CartesianSchool

PYTHON 3.14 · OD ZERA

Python od zera

programowanie, grafika,
aplikacje i gry

Siergej Sobolewski

Software & AI Engineer, założyciel Cartesian School

PYTHON 3.14 · OD ZERA

Python od zera

programowanie, grafika, aplikacje i gry

Siergej Sobolewski
Software & AI Engineer, założyciel Cartesian School

Python od zera: programowanie, grafika, aplikacje i gry

Siergej Sobolewski — Software & AI Engineer, założyciel Cartesian School

Wydanie Cartesian School, 2026. Python 3.14.

Wydanie elektroniczne. Wersja online kursu, interaktywna praktyka w przeglądarce i kod źródłowy wszystkich projektów — cartesianschool.org

Spis treści

MATERIAŁY WPROWADZAJĄCE

PROJEKTY

KOMPENDIUM

O autorze

Siergej Sobolewski — inżynier z ponad dwudziestoletnią praktyką w programowaniu systemowym i AI, założyciel Cartesian School. Jego doświadczenie — od embedded i awioniki po produkcyjne AI/ML — legło u podstaw metody inżynierskiej tego kursu: wyjaśnienie musi być nie tylko zrozumiałe, ale precyzyjne i weryfikowalne w rzeczywistym kodzie. Ta książka to próba pokazania programowania tak, jak pokazuje się je inżynierowi przygotowywanemu do prawdziwej pracy, ale już od pierwszego kroku.

Siergej Sobolewski, założyciel Cartesian School. Ta książka powstała z praktycznego pytania: jak wytłumaczyć programowanie osobie, która nigdy wcześniej nie napisała ani linijki kodu, nie zamieniając wyjaśnienia ani w nudny podręcznik, ani w powierzchowną „książeczkę z obrazkami”.

Cartesian School to projekt edukacyjny oparty na idei, że nowoczesne podejście inżynierskie i zrozumiałe wyjaśnienie dla początkujących wcale sobie nie przeczą: te same zajęcia mogą być jednocześnie precyzyjne i wciągające.

O recenzencie technicznym

> **Otwarty punkt**
> Nazwisko recenzenta technicznego zostanie tutaj dodane po Product Owner wyznaczeniu recenzenta do publikacji. Rozdział została zachowana zgodnie z kanoniczną strukturą spisu treści i nie została usunięta — jednocześnie nie wymyśliliśmy osoby nieistniejącej.

Każdy rozdział tej książki przechodzi kontrolę techniczną: cały kod jest automatycznie kompilowany, sprawdzany przez lintera `ruff`, a przykłady w formacie Jupyter Notebook są naprawdę wykonywane od początku do końca — wyniki obliczeń w tekście książki są uzyskiwane nie „na oko”, ale prawdziwym uruchomieniem kodu na Python 3.14.

Wprowadzenie

Krótko o tym, jak zbudowana jest ta książka i jak wydobyć z niej maksimum.

Dla kogo jest ta książka

Ta książka jest dla tych, którzy nigdy wcześniej nie pisali kodu. Wcale. Nie ma znaczenia, ile masz lat: dziesięć, dwadzieścia pięć czy pięćdziesiąt — jeśli potrafisz obsługiwać komputer i jesteś gotów to rozgryźć, Książka prowadzi cię od pierwszej wydrukowanej linii do własnych gier, aplikacji i rysunków na ekranie.

Jak działa każdy rozdział

Prawie w każdym rozdziale spotkasz ten sam porządek: najpierw — po co to jest potrzebne, potem — proste wyjaśnienie idei, potem — działający przykład, który można uruchomić od razu, a na końcu — praktyka i typowe błędy. Celowo nie zaczynamy od suchej definicji — najpierw powinno być jasne, po co ci to, a termin pojawi się potem.

Klasyczne podejście i nowoczesny Python 3.14

Python istnieje od ponad trzydziestu lat i w tym czasie jest wygodniejsza sposoby robienia znanych rzeczy. Tam, gdzie jest to ważne, książka najpierw pokazuje klasykę sztuczka — bo właśnie to znajdziesz w starym kodzie, artykułach i odpowiedziach na forach — oraz Następnie nowoczesna wersja Python 3.14 i wyjaśnia, czym się różnią i czego używać dzisiaj. Jest to oznaczone specjalnym blokiem „Klasyczne podejście → nowoczesne Python”

Teoria na miejscu, praktyka w Jupyter Notebook

Teorię czytasz tutaj, na stronie Cartesian School — z podświetleniem kodu, wyszukiwaniem i nawigacją. A do praktyki do każdej sekcji dołączony jest osobny plik Jupyter Notebook:, w którym można zmieniać kod i od razu widzieć wynik, nie otwierając nic dodatkowego, poza Visual Studio Code lub PyCharm.

Poziomy trudności praktyki

Zadania oznaczone są gwiazdkami: ★ — podstawowa praktyka, powtarzająca przykład z rozdziału; ★★ jest samodzielnym zadaniem, które wymaga odrobiny refleksji; ★★★ jest zadaniem o rosnącej złożoności dla tych, którzy chcą więcej. Nie w każdym przypadku sekcja ma wszystkie trzy poziomy – tylko tam, gdzie jest uzasadniona.

Czego potrzebujesz

Python 3.14, zainstalowany zgodnie z instrukcjami z Rozdziału 2, oraz jeden z dwóch edytorów kodu: Visual Studio Code lub PyCharm są szczegółowo opisane w książce. Prawie nic więcej nie jest potrzebne: Wszystkie projekty z tej książki można uruchomić na zwykłym laptopie.

Czy wiedziałeś?

Pierwszy rozdział książki „Python od zera”

Czym jest programowanie?

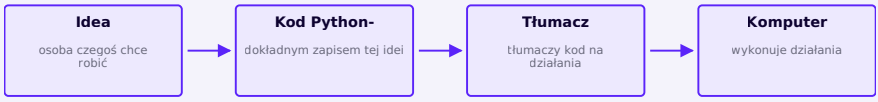
Przed napisaniem kodu warto zrozumieć, co się dzieje, gdy komputer „wykonuje program”

Po tym rozdziale będziesz mógł: rozróżniać program, kod źródłowy, algorytm, instrukcję i wyrażenie; wyjaśnić rolę bajt-kodu w CPython; ustawić główne etapy historii Python; rozróżniać python.org, dokumentację, PyPI, PSF i PEP; nazwać obszary zastosowań i granice Python; uruchomić `print(...)` i czytaj Komunikat o błędzie.

Czym jest programowanie?

Komputer jest niesamowicie szybki, ale całkowicie niesamodzielny. Sam z siebie nie potrafi w ogóle niczego — ani otworzyć gry, ani pokazać zdjęcia, ani policzyć reszty w sklepie. Wszystko, co robi komputer, robi dlatego, że ktoś wcześniej napisał dla niego dokładną sekwencję poleceń. Ta sekwencja nazywa się **program**, a proces jego pisania jest **programowanie**.

Program tekstowy nazywa się **kodem źródłowym** (source code). Procesor nie jest wykonuje ją bezpośrednio: używa w zeszycie ćwiczeń **CPython** przygotowuje kod i wykonuje ją. Na pierwszym poziomie nazwiemy ten proces pracą **tłumacz**; dokładna schematyka z bajtkodem podana poniżej.



Od pomysłu do rezultatu: ścieżka programu

Co się dzieje
Gdy klikasz „Uruchom”

Algorytm, instrukcja, wyrażenie

Kilka słów, które pojawią się w każdym rozdziale tej książki – raz się przyda zgadzam się co do tego, co mają na myśli:

- **Algorytm** jest ostateczną, jednoznacznie podaną sekwencją kroków, co dla poprawnych wejść prowadzi do wyniku lub zakończenia. Wynik może zależeć od danych wejściowych lub losowości — nie musi być zawsze taka sama.

- **Instrukcja** (statement) to jedna pełna instrukcja dla programu, na przykład "Wyświetl tekst na ekranie" lub "Zapisz numer do pamięci".
- **Ekspresja** (expression) to część kodu, która ma wartość, taką jak `2 + 2` lub `"Привет" + "!"`. Wyrażenia często znajdują się wewnątrz instrukcji.
- **Program** jest uporządkowanym zbiorem instrukcji w języku programowania, implementacja jednego lub więcej algorytmów.
- **Kod maszynowy** to ciąg liczb, który wykonuje się bezpośrednio CPU. Ludzie prawie nigdy nie piszą go ręcznie, a to właśnie lubią języki Python oraz tłumaczy, którzy zamieniają przyjazny dla ludzi kod w kod maszynowy.

Interpreter i kompilator

Poziom 1 (wystarczający, by zacząć). Implementacja Python przyjmuje kod źródłowy i organizuje jego wykonanie zgodnie z zasadami sterowania przepływem. Zwykle nazywa się ją **tłumacz**. To celowo uproszczony model.

Dla porównania, niektóre języki (np. C lub Rust) mają osobny krok — **Kompilator** tłumaczy cały kod źródłowy na kod maszynowy z góry, oraz Dopiero wtedy uruchamiany jest gotowy program. W normalnym CPython uruchamianiu użytkownik nie wykonuje maszynowy plik z wyprzedzeniem, więc Python nazywa się *interpretacja*. To jest opis sposobu uruchamiania, a nie stwierdzenie, że kompilacji nie ma.

> **Powszechny błąd**
> Stwierdzenie „Python w ogóle się nie kompiluje”

Poziom 2 (jeśli zastanawiasz się, co tak naprawdę się dzieje). Oficjalny („referencyjna” **bajtcode** (kompaktowe instrukcje, niezrozumiałe dla człowieka, ale szybkie do wykonania), a następnie wykonuje ten bytecode przy pomocy Python VM (maszyny wirtualnej). Wszystko dzieje się automatycznie i niezauważalnie dla ciebie — po prostu teraz wiesz, że „interpretowane” nie znaczy „bez ani jednego kroku kompilacji”.

Poziom 2: Co tak naprawdę CPython robi w środku

Więcej informacji o CPython można znaleźć w sekcji „Wspólnota i filozofia Python”

Dlaczego twoje dzieci powinny nauczyć się programowania?

Programowanie nie jest zastosowaniem dla zawodu „programisty”

- **Błąd — to normalne.** Program prawie nigdy nie działa za pierwszym razem, i nie jest to powód do zmartwienia — to część procesu. Programowanie uczy spokojnie znajdować przyczynę i próbować ponownie, zamiast bać się popełnić błąd.
- **Pomysł zamienia się w efekt.** Myśl „a co jeśli zrobisz to tak”
- **To umiejętność przenośna.** Umiejętność rozłożenia złożonego zadania na proste kroki Przydaje się znacznie poza programowaniem – w szkole, pracy i codziennym życiu Zadania.

Dlaczego Python?

Istnieją setki języków programowania. Python wybranych jako pierwszy język w trzech praktycznych Przyczyny:

- **Czyta się prawie jak zwykły tekst.** String `if age >= 18:` jest zrozumiałe bez tłumaczenia. Mniej abstrakcyjne składnia — więcej uwagi poświęconej samej idei.
- **Jeden język – bardzo różne rezultaty.** Gry są pisane i rysowane na Python grafika, składanie aplikacji desktopowych i webowych, trenowanie modeli sztucznej inteligencji. Wszystko, o czym przeczytasz w tej książce, jest napisane w tym samym języku.
- **Ogromna społeczność.** Jeśli masz pytanie, prawdopodobnie ktoś inny już o to pytałem i otrzymałem odpowiedź. Gotowe narzędzia (nazywane bibliotekami lub pakietów) dla Python napisano ich ogromną liczbę — więcej o tym później Ten sam rozdział.

Historia Python

Python nie pojawił się przypadkowo i nie od razu — krótka, ale prawdziwa historia o tym, dlaczego język działa tak, a nie inaczej.

Python

Przed Python: CWI i językiem ABC

Python nie pojawił się znikąd. W latach 80., w Holenderskim Instytucie Badawczym, Instytut **CWI** (Centrum Wiskunde & Informatica, Centrum Matematyki oraz informatyki") opracował język **ABC** musiało być proste i wygodne do nauczania programowania dla osób nieprofesjonalnych. ABC miał dobre pomysły (np. wgłębienia zamiast kręconych aparatów), ale istniały poważne ograniczenia — nie było przeznaczone dla dla dużych „prawdziwych”

Narodziny Python: 1989–1991

Jeden z twórców ABC, holenderski programista **Guido van Rossum** (Guido van Rossum), doskonale znał mocne i słabe strony ABC. Pod koniec grudnia 1989 roku, w Podczas świąt Bożego Narodzenia zaczął pisać nowy język – jako projekt hobbysty, by się czymś zająć Na święta i jako sposób na zachowanie pomysłów ABC lubiący, pozbywając się ich Ograniczenia.

Guido van Rossum na konferencji PyCon US

Guido van Rossum, twórca Python, PyCon US 2024 roku. Zdjęcie: Kushal Das, licencja CC BY-SA 4.0 (obraz przycięty i skompresowany na potrzeby strony).

Pierwsze publiczne wydanie, **Python 0.9.0**, został wydany w lutym 1991 roku. W nim Istniały już funkcje, obsługa wyjątków, klasy z dziedziczeniem oraz wbudowane typy danych Na przykład listy i słowniki, wiele z tego, czego użyjesz w tej książce.

> **Idea**
> Python rozpoczęło się jako osobisty projekt jednej osoby dla własnej przyjemności. Nie trzeba od samego początku „zmieniać branży” — czasami wystarczy rozwiązać własne małe zadanie.

Skąd wzięła się nazwa „Python”

Pomimo logo węża, język nie nosi nazwy gada. Guido van Rossum jest wielkim fanem brytyjskiego programu komediowego **Latający Cyrk Monty Pythona** (Monty Python's Flying Circus) i chciał czegoś

krótkiego, trochę nietypowego i zapadającego w pamięć Imię. Motyw węża w logo pojawił się później, jako wizualne skojarzenie ze słowem „Python”

Python 1.x i Python 2.0

Wersja **Python 1.0** został wydany w styczniu 1994 roku. Język nadal się rozwijał: Pojawiły się nowe moduły Standardowej Biblioteki, a społeczność deweloperów rosła.

W październiku 2000 roku **Python 2.0**, przyniósł ważne rzeczy dla języka — na przykład wyrażenia listowe (list comprehensions) oraz pełnoprawny garbage collector, potrafiący znajdować i zwalniać pamięć nawet w obiektach odniesionych do siebie nawzajem w pętli.

W 2001 roku został stworzony **Python Software Foundation (PSF)** - Non-Profit organizacja, która od tego czasu posiada prawa do Python i kierowała rozwojem język. Dowiedz się więcej o PSF w kolejnej części tego rozdziału.

Python 3: Dlaczego potrzebujesz niekompatybilnej wersji

Do połowy lat 2000. język nabrał cech, których nie dało się poprawić bez nich Poprzez łamanie części starego kodu — na przykład sposobu, w jaki Python 2 obsługiwał tekst i bajty — było źródłem ciągłego zamieszania, zwłaszcza przy pracy z językami nieangielskimi (w tym rosyjski). Twórcy Python świadomie dążyli do wydania **Python 3.0** (grudzień 2008) — wersja, która miejscami *celowo niekompatybilna* ze starym kodem Aby skorygować te nagromadzone problemy: pojedynczy model tekstowy (Unicode domyślnie), `print()` stała się zwykłą funkcją, a nie specjalnym słowem języka, dzielenie całkowite stało się bardziej przewidywalne i wiele więcej.

> **Oficjalne źródło**
> Ogólny plan Python 3000 opisany jest w PEP 3000, a zmiany zauważalne dla użytkownika opisane są w oficjalnej sekcji What's New In Python 3.0. PEP 3100 Miscellaneous Python 3.0 Plans uzupełnia je listą indywidualnych planowanych zmian, ale nie zawiera pełnej listy powodów zmiany.

Przejście trwało znacznie dłużej niż się spodziewano: społeczność potrzebowała czasu, by Przepisz stary kod i biblioteki. Oficjalne wsparcie **Python 2 zakończyła się 1 stycznia 2020 roku** – od Python nawet Naprawy krytycznych podatności.

Droga do nowoczesnych Python i wersja 3.14

Po 2008 roku Python rozwijał się w zauważalnych, ale już kompatybilnych etapach: f-strings do łatwego formatowania tekstu (3.6), operator walrus `:=` (3.8), bardziej zrozumiałe i dokładne komunikaty o błędach oraz dopasowanie strukturalne do wzorca (3.10), zauważalny wzrost szybkości wykonywania w kilku ostatnich wydaniach. To właśnie ciągły, ale ostrożny rozwój, definiuje to, co dzisiaj nazywa się „nowoczesnym Python”.

Ta książka została napisana i przetestowana w całości **Python 3.14** — tematyczne Stabilne wydanie w chwili pisania.

CWI, Holandia
instytut badawczy, w którym rodzi się pomysł

ABC języka
Python poprzednik jest prosty, ale z ograniczeniami

koniec 1989 roku
Guido van Rossum zaczyna Python w wakacje bożonarodzeniowe

1991 · Python 0.9.0
pierwsze publiczne wydanie

1994 · Python 1.0
pierwsza pełna wersja

2000 · Python 2.0
listę dodatków, garbage collector z pętlami

2001 · utworzono PSF
Python Software Foundation jest organizacją non-profit

2008 · Python 3.0
świadomie niekompatybilne poprawy językowe

2020 · Koniec Python 2
oficjalne wsparcie dla Python 2 zostało zakończone

Dziś · Python 3.14
aktualne stabilne wydanie, baza tej książki

Krótka chronologia Python

python.org, dokumentacja i PyPI

Trzy miejsca, które warto poznać już na samym początku: oficjalna strona internetowa, oficjalna dokumentacja oraz katalog pakietów firm trzecich.

Python

python.org — oficjalna strona internetowa

python.org — oficjalna strona języka Python, którą zarządza Python Software Foundation. To pierwsze miejsce, gdzie warto szukać oficjalnych informacji: jak zainstalować Python, gdzie przeczytać dokumentację, co nowego w ostatniej wersji.

python.org jest oficjalną stroną Python. Zwróć uwagę na górne menu (About, Downloads, Documentation, Community) oraz blok „Download”

Na stronie głównej warto pamiętać o czterech sekcjach menu górnego:

- **Downloads** są Python instalatorzy Windows, macOS i Linux (to jest to, co masz Będziesz go już używać w rozdziale 2).
- **Documentation** — link do docs.python.org, oficjalnej dokumentacji (więcej szczegółów poniżej).
- **Community** – fora, listy mailingowe, lokalne społeczności Python-deweloperów.
- **PSF** (zakładka na górze strony) — sekcja dotycząca samej organizacji, Python Software Foundation.

Typowy błąd

W internecie jest wiele stron o podobnych nazwach i «ułatwionym instalowaniu Python», które w rzeczywistości nie mają nic wspólnego z oficjalnym projektem. Pewny znak autentyczności — domena `python.org` w pasku adresu przeglądarki.

Oficjalna dokumentacja: docs.python.org

Dokumentacja to nie jest coś, co czytają tylko doświadczeni programiści. Dobrze Dokumentacja to narzędzie działające na co dzień, z którego korzysta absolutnie każdy, w tym autorzy samego języka.

Strona główna docs.python.org 3.14 z sekcjami Tutorial, Library reference Language reference

docs.python.org jest dokumentacją oficjalną. Trzy sekcje w lewym górnym rogu strony powinny być rozróżnione od samego początku.

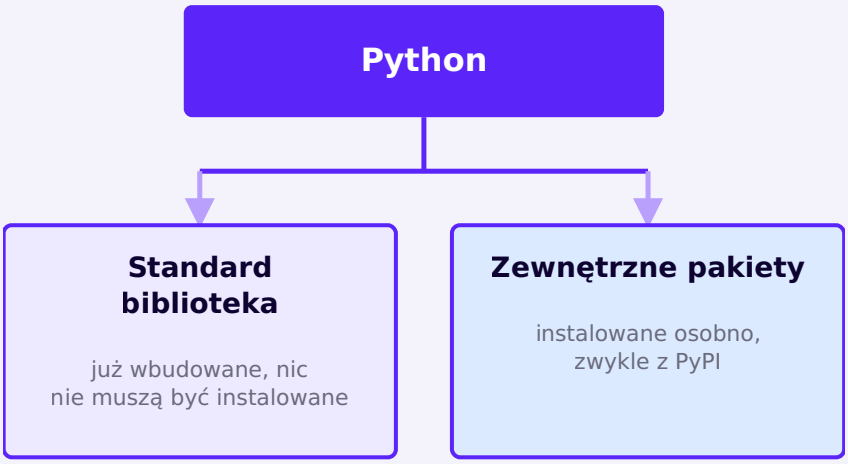
Na stronie dokumentacji od razu widzisz trzy różne sekcje i warto zrozumieć różnicę między nimi Oni:

- **Tutorial** jest edukacyjnym wprowadzeniem do języka, napisanym dla tych, którzy już Trochę znam programowanie. Dobre źródło do recenzji, ale nie zastępuje tego Książka dla zupełnego początkującego.
- **Library Reference** - Szczegółowy opis biblioteki standardowej: co Każdy wbudowany moduł i funkcja mogą to zrobić. To tutaj będziesz szukać najczęściej, Kiedy chcesz wiedzieć, co jeszcze potrafi, na przykład, `print()`.
- **Language Reference** jest formalnym, najściślejszym opisem zasad Język. Zazwyczaj nie jest potrzebny początkującym – jest zaprojektowany do bardzo precyzyjnych problemów technicznych.

Idea
Nie musisz jeszcze rozumieć całej dokumentacji. Celem tej sekcji jest po prostu przestanie bać się słowa „dokumentacja”

PyPI: pakiety Python firm trzecich

W Python już wbudowano wiele przydatnego — to nazywa się **standard Biblioteka**. Ale wciąż jest ogromna liczba gotowych narzędzi, napisane przez innych deweloperów i wydane osobno — nazywają się **pakiety** (lub bibliotek). Rozszerzają możliwości Python znacznie dalej Granice tego, co jest „nietypowe”



Dwie części tego, co można wykorzystać w Python

Główne repozytorium takich pakietów nazywa się **PyPI** — Python Package Index (pypi.org). Kiedy ktoś mówi „włóż paczkę do pip”

pypi.org strona główna z polami wyszukiwania i statystykami liczby projektów

pypi.org to Python Package Index, katalog pakietów firm trzecich Python.

Oficjalne źródło

Szczegółowy praktyczny przewodnik po instalacji pakietów – packaging.python.org. bardzo praktyczna możliwość korzystania `pip` i środowiska wirtualne będą potrzebne później w tej książce — na razie wystarczy zrozumieć ideę ekosystemu pakietów.

Wspólnota i filozofia Python

Kto decyduje, kim Python się stanie, jakie zasady go kierują i dlaczego "Python" i "CPython" to nie do końca to samo.

Python Software Foundation (PSF)

Python Software Foundation — organizacja non-profit w USA, założona w 2001 roku. Posiada prawa do Python, wspiera rozwój języka, organizuje konferencje (np. coroczną PyCon US)) i finansuje infrastrukturę społeczności — w tym samą stronę python.org i PyPI.

PSF nie sprzedaje Python ani nie wydaje oficjalnych certyfikatów w imieniu języka – jego rola raczej administracyjne i organizacyjne: by zapewnić Python przyszłość i że Rozwój był prowadzony przez otwartą społeczność, a nie przez jedną firmę.

PEP: jako Python podejmuje decyzje dotyczące zmian

Ważne zmiany w języku nie zachodzą spontanicznie. Istnieje formalny proces — **PEP** (Python Enhancement Proposal, „propozycja ulepszenia Python"). To dokument, w którym ktoś szczegółowo opisuje proponowaną zmianę, wyjaśnia, dlaczego jest potrzebna, i omawia ją ze społecznością, zanim zostanie zaakceptowana lub odrzucona.

Wszystkie PEP ponumerowane i opublikowane na peps.python.org. Two PEP warto znać z nazwy Teraz, żeby lepiej ich poznać w przyszłości:

- **PEP 8** - Python przewodnik po stylu kodu (jak nazywać zmienne, ile wcięć i przestrzeni pozostawić). Wrócimy do tego szczegółowo później w tej książce.
- **PEP 20** znany jest również jako „Python zen"

Zen Python

Python ma własne krótkie lista zasady, które wyjaśniają, jak język próbuje Być. Został napisany przez dewelopera Tima Petersa i można to zobaczyć, uruchamiając go w interpreterze Python jedną linijkę:

dzen.py

```
import this
```

Własnymi słowami, a nie cytatem dosłownym, idea sprowadza się do kilku rzeczy, które Python najbardziej docenia:

- **Czytelność ważniejsza niż zwięzłość.** Czysty kod jest lepszy niż kod, który zmieścić się w jednej linijce kosztem przejrzystości.

- **Jawność jest lepsza niż ukryta.** Kodeks powinien, jeśli to możliwe, bezpośrednio pokazywać, co robi, i nie polegać na ukrytej „magii”
- **Prostota jest lepsza niż złożoność — ale nie kosztem możliwości.** Jeśli zadanie Złożone, rozwiązanie może być trudne, ale nie bardziej niż powinno być.

Podstawowa praktyka

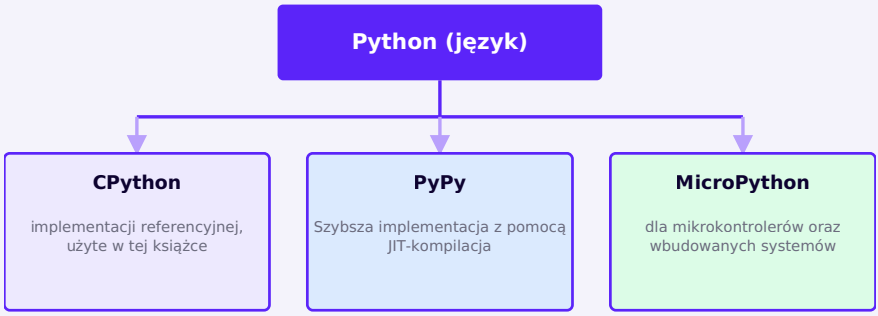
Spróbujmy

Otwórz praktykę w tej sekcji i uzupełnij komórkę słowami `import this`. Przeczytaj cały wynik — te linie będą pojawiać się w podpowiedziach książki jeszcze nie raz.

CPython i inne implementacje

Warto rozróżniać dwie rzeczy: **Python jako język** (zasady składni i zachowania, które Language Reference) i **konkretny program, który Ten język działa** nazywa się realizacją.

Najczęściej stosowaną implementacją jest **CPython**, napisane w języku C. Jej używasz przez całą tę książkę, i to samo pobierają z python.org domyślnie. Istnieją też inne implementacje, stworzone dla szczególnych zadań:



Python jest język; Język posiada kilka implementacji

- **PyPy** jest implementacją z kompilacją JIT- (Just-In-Time), czyli W niektórych zadaniach wykonuje kod znacznie szybciej CPython dzięki kompilacji na bieżąco w czasie rzeczywistym.

- **MicroPython** jest zwartą implementacją Python dla mikrokontrolerów i Urządzenia wbudowane z bardzo ograniczoną pamięcią, od robotów po inteligentne czujniki.

> **Idea**
> W tej książce wystarczy wiedzieć, że gdy gdzieś jest napisane „Python 3.14.2”

Python — to zabawa!

Krótka wycieczka po tym, co zbudujesz własnymi rękami na kartach tej książki — i szczere spojrzenie na to, gdzie Python ma zastosowanie w prawdziwym świecie.

Games!

W rozdziałach 17, 19 i 21 złożysz trzy prawdziwe gry w całości: Kółko, krzyżyk, Snake and space shooter — od pustego okna do gotowej gry z punktami i zasadami. Każdy z nich jest zbudowany w małych, zrozumiałych krokach i nie pojawia się w jednym dużym kawałku kod.

Grafika i animacja

Już w rozdziale 6 zaczniesz rysować na ekranie, używając `turtle` to wirtualny żółw z piórem, który porusza się na twoje polecenia. Oto jak wygląda kod, Rysowanie kwadratu – na przykład, szczegółowo przeanalizujemy je później:

podglyadyvaem_vpered.py

```
import turtle

artist = turtle.Turtle()
for _ in range(4):
    artist.forward(100)   # jedziemy dalej
    artist.right(90)      # obróć się o 90°
```

> **Patrząc w przyszłość**
> Nie próbuj teraz rozumieć tego kodu – `for` (pętla) pojawi się dopiero w rozdziale 10. Tutaj wystarczy zobaczyć, że prawdziwy rysunek powstał tylko z kilku linii.

Strony internetowe

W rozdziale 22 dowiesz się, jak działa sieć — co robią HTML, CSS i JavaScript w przeglądarce, jaką rolę pełni Python na serwerze, — i uruchomisz własną działającą stronę na Flask.

Zastosowania

Używając `Tkinter` (rozdziały 16–18) zbudujecie prawdziwe aplikacje okienkowe z przyciskami i polami wprowadzania — kalkulator napiwków i własną aplikację do rysowania — a nie tylko programy wyświetlające tekst w terminalu.

Gdzie Python jest używany w rzeczywistym świecie

Gry, grafika, aplikacje i strony internetowe w tej książce to tylko część obrazu. Poza tym Kurs szkoleniowy Python stosowany w bardzo różnych obszarach:

- **Automatyzacja i skrypty** — zmienić nazwę setek plików, przetworzyć tabelę, uruchomić rutynowe zadanie zgodnie z harmonogramem.
- **Tworzenie stron internetowych** jest częścią serwerową dużych stron (Django, Flask oraz podobne narzędzia).
- **Analiza danych i nauka** – przetwarzanie dużych tabel danych, statystyk, Obliczenia naukowe (pandas, NumPy).
- **Sztuczna inteligencja i uczenie maszynowe** – Szkolenia i start-up modele (PyTorch, TensorFlow i inne).
- **Serwery i usługi backendowe** - API, kolejki zadań, wewnętrzne Narzędzia firm.
- **Testowanie programów** — automatyczne testy dla innego oprogramowania, niezależnie od języka, w którym zostało napisane.
- **DevOps i infrastruktura** - Skrypty wdrożeniowe, automatyzacja serwerów.
- **Edukacja** — jeden z najpopularniejszych pierwszych języków programowania na świecie, w tym w tej książce.
- **Narzędzia do cyberbezpieczeństwa** — skanery, automatyzacja analizy słabości.
- **Wbudowane systemy** - MicroPython na mikrokontrolerach oraz Małe urządzenia.

Tam Python gdzie nie jest najlepszym wyborem

Dla uczciwości: Python nie pasuje tak samo dobrze do absolutnie wszystkiego. Kilka przykładów, gdzie zwykle wybiera się inne narzędzia:

- **Twarde systemy czasu rzeczywistego** — tam, gdzie opóźnienie w ułamkach milisekundy jest niedopuszczalne (na przykład w niektórych wbudowanych systemach sterowania), zazwyczaj wybiera się języki bardziej przewidywalne pod względem czasu wykonania.
- **Niektóre natywne aplikacje mobilne** iOS i Android częściej Korzystaj z narzędzi natywnych dla tych platform.
- **Niskopoziomowa praca z jądrem systemu operacyjnego i sterownikami urządzeń** — ten obszar zwykle wymaga języków takich jak C, które dają bezpośrednią kontrolę nad pamięcią i sprzętem.
- **Środowiska o silnym ograniczeniu pamięci** to właśnie MicroPython okazuje się, że są zbyt „ciężkie”

To w porządku — nie ma języka programowania, universal. znać mocne i słabe strony Bok instrumentu jest częścią rozwoju zawodowego.

> **Praktyka: Pierwsze uruchomienie kodu**
> interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji
>
> Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/01-01/index.html)

Pierwsze zmienne i pierwsze błędy

Jedno poprawne mentalne wyobrażenie zmiennej i pierwsze spokojne spotkanie z komunikatem o błędzie, na długo zanim cię zaskoczy.

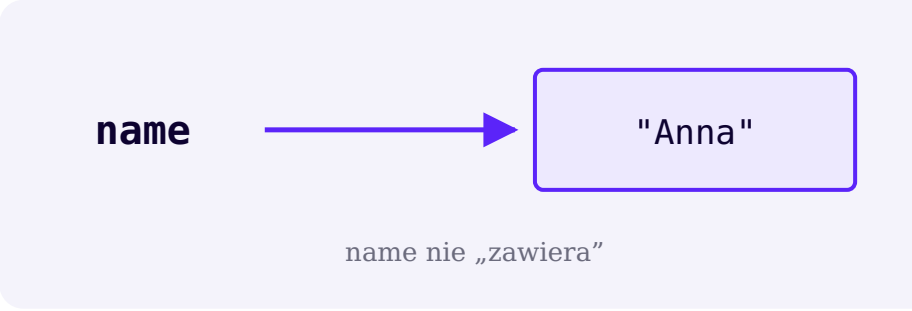
Pierwsze zmienne: jak Python je faktycznie przechowywać

W następnym rozdziale zaczniesz aktywnie używać zmiennych. Jedna jest mała, ale Ważny obraz w głowie należy opanować już teraz — uchroni cię przed późniejszym zamieszaniem.

peremennaya.py

```
name = "Anna"
print(name)
```

Łatwo (i błędnie) jest myśleć o zmiennej jako o „pudełku” **imię — to skrót, który wskazuje na wartość**, a nie kontener, gdzie fizycznie znajduje się wartość.



Co się dzieje

`name = "Anna"` znaczy "niech nazwa zostanie `name` teraz oznacza wartość `"Anna"` " zamiast "włożyć tekst do pola o nazwie name". Szczegółowa struktura pamięci Python to temat na kolejne rozdziały; na razie ważne jest, by nie przyzwyczaić się do błędnego obrazu "pudełka".

Podstawowa praktyka

Eksperyment

W praktyce tej sekcji stwórz zmienną `name` swoim imieniem i nazwiskiem i wyjść przez `print(name)` . To przydziel `name` inne znaczenie i wydrukuj je ponownie — zauważ, że „etykieta”

Błędy są normalne

prędzej czy później każdy program wyrzuca błąd — i to nie jest znak, że coś poszło nie tak katastrofalnie błędny. Spójrzmy na przykład teraz, gdy stopy są niskie.

```
oshibka.py
print("Hello"
```

Temu kodowi brakuje nawiasu zamykającego. Jeśli wykonasz taką linię, Python nie zgaduje, o co ci chodziło — zamiast tego zatrzymuje się i wpisuje szczegółowy opis Wywołano komunikat o błędzie **traceback** („śledzenie”

Typowy błąd początkującego
Gdy widzisz długi tekst z błędem, łatwo wpadnie w panikę i pomyśleć, że „coś jest zepsute na zawsze”

Przydatny nawyk przy widoku błędu — przeczytać **ostatnie zdanie** traceback: tam zwykle krótko napisano, co dokładnie poszło nie tak. Wszystko powyżej — to droga, którą Python dotarł do problemu.

★★ Zadanie samodzielne
Debug Lab
W praktyce tej sekcji znajdziesz specjalnie zepsuty wiersz kodu. Przeczytaj traceback, znajdź wiersz z problemem i napraw go samodzielnie — tak samo jak w sekcji „Typowy błąd” poprzedniej praktyki tego rozdziału.

Jak w pełni wykorzystać tę książkę

Kilka nawyków, które sprawią, że czytanie będzie znacznie skuteczniejsze — oraz pełne podsumowanie rozdziału 1.

Cztery proste nawyki sprawią, że czytanie książki będzie znacznie skuteczniejsze:

- **Wpisuj kod ręcznie**zamiast kopiować. Literówka, którą znajdziesz, i Sam to naprawiasz, uczy się znacznie lepiej niż kod, który działał za pierwszym razem.
- **Zmiana liczb i słów w przykładach** — co się stanie, jeśli wpiszesz Nie 100, ale 300? Przewidywanie wyniku *do* start to najlepszy trening.
- **Nie pomijaj sekcji „Typowy błąd”** Prawdziwe są tam rozebrane Błędy początkujących polegają na tym, że lepiej jest ich zobaczyć z wyprzedzeniem niż spotykać się z nimi na ślepo.
- **otwórz laptop ćwiczeniowy** dla każdej sekcji to teoria na stronie Wyjaśnia tę ideę, a laptop to miejsce, gdzie testujesz to rękami.

Jeśli coś nie działa

To nie jest znak, że programowanie nie jest dla ciebie — to najczęstsza część procesu. W rozdziale 3 i później w książce omówiliśmy cały system: jak odczytać komunikat o błędzie i znaleźć przyczynę, zamiast się jej bać.

Podsumowanie

Czego nauczyliśmy się w tym rozdziale

- Program to dokładna sekwencja poleceń dla komputera; zamienia kod źródłowy w akcje **interpreter**, a wewnątrz CPython kod wzdłuż ścieżki również jest kompilowany do bajtkodu.
- Python ukazał się w CWI pod koniec 1989 roku – został stworzony przez Guido van Rossuma jako rozwinięcie idei języka ABC; nazwa pochodzi na cześć Monty Python Flying Circus.
- Python 3 jest celowo niekompatybilny z Python 2 – ta cena była świadomym wyborem rozwiązania długoletnich problemów językowych; wsparcie dla Python 2 zakończyło się w 2020 roku.
- python.org jest oficjalna strona internetowa, docs.python.org oficjalna dokumentacja (Tutorial/Library Reference/Language Reference), PyPI katalog pakietów firm trzecich.
- Python Software Foundation kieruje rozwojem języka; ważne zmiany przechodzą przez proces PEP; „Zen Python” `import this`) opisuje wartości języka.
- CPython jest główną implementacją Python, ale nie jedyną: istnieją także PyPy i MicroPython dla różnych zadań.
- Python jest używany w wielu dziedzinach – ale nie jest najlepszym wyborem absolutnie wszędzie i jest to normalne w każdym języku.
- Nazwa zmiennej wskazuje na wartość, a nie „zawiera”
- W rozdziale 2 zainstalujesz Python na komputerze i zaczniesz pisać kod poza przeglądarką.

ROZDZIAŁ 2 · STR. 29

Ustalmy Python!

Pełna konfiguracja miejsca pracy Python- programisty: instalacja na Windows, macOS i Linux, terminal i PATH, VS Code i PyCharm, środowiska wirtualne, pip/pipx/venv/uv/conda — i jak to wszystko jest ze sobą powiązane.

Co dokładnie instalujemy?

Zanim zaznaczysz pola w instalatorach, warto zrozumieć, z jakich warstw w ogóle składa się stanowisko robocze programisty Python-.

Komputer jest niesamowicie szybki, ale całkowicie niesamodzielny. Sam z siebie nie potrafi w ogóle niczego — ani otworzyć gry, ani pokazać zdjęcia, ani policzyć reszty w sklepie. Wszystko, co robi komputer, robi dlatego, że ktoś wcześniej napisał dla niego dokładną sekwencję poleceń. Ta sekwencja nazywa się **program**, a proces jego pisania jest **programowanie**.

Kiedy instalujesz Python, to właśnie instalujesz **interpreter** — program, który potrafi czytać i wykonywać kod w Python (rozmawialiśmy o tym w rozdziale 1). Ale pomiędzy "Python zainstalowany" i "mój kod działa" to w rzeczywistości cały łańcuch warstw, A ten rozdział dotyczy szczerej analizy każdego z nich.

siedem warstw między sprzętem a pierwszym programem

Cztery rzeczy, które łatwo pomylić

To być może najważniejsza różnica w całym rozdziale — i jeden z głównych powodów, dla których Początkujący mają zamieszanie w kwestii instalacji Python:

- **Python-interpreter ≠ VS Code.** VS Code — to edytor kodu. Sam nie potrafi wykonywać Python — znajduje i uruchamia zainstalowany na komputerze interpreter.
- **Python-interpreter ≠ PyCharm.** Podobnie: PyCharm jest środowiskiem rozwojowym, który również korzysta z zewnętrznego interpretera i nie zawiera go „wewnątrz siebie”

- **Interpreter Pythona ≠ pip.** pip to program do instalowania pakietów; Pracuje sama *na wierzchu* tłumacz, nie tłumacz.
- **Python-interpreter ≠ środowisku wirtualnym.** Wirtualne środowisko (szczegółowo opisane w sekcjach 2.10–2.11) — to lekka, izolowana kopia interpretera dla konkretnego projektu, a nie coś oddzielnego od niego.

> **Idea**
> VS Code i PyCharm to narzędzia, które *używają* Python-interpreter zamiast go zastępować. Instalacja edytora kodu i instalacja Python to dwa różne, niezależne kroki.

Później w tym rozdziale przejdziemy do końca w kolejności: zainstaluj sam interpreter na twoim System operacyjny (sekcje 2.3–2.5), naucz się sprawdzać, który Python jest realistyczny. Uruchamia (2.6), konfiguruje edytor kodu (2.7–2.9) i zajmuje się środowiskami wirtualnymi (2.10–2.12) i z otoczeniem zoo narzędzi wokół nich (2.13–2.14), a na końcu wszystko złożymy razem oraz ćwiczyć znajdowanie i naprawianie typowych błędów (2.15–2.17).

Terminal, shell i PATH

Jedno z ważniejszych wyjaśnień tego rozdziału: co naprawdę dzieje się, gdy wprowadzasz polecenie i naciskasz Enter.

Terminal i shell

Terminal jest okno programu, w którym wpisujesz polecenia tekstowe i widzisz Odpowiedź tekstowa na komputerze. **Shell** (powłoka poleceń) to program, który Wewnątrz terminala faktycznie odczytuje twoje polecenie i decyduje, co z nim zrobić. Na Windows tego Zazwyczaj PowerShell lub klasyczne cmd, na macOS i Linux – najczęściej `zsh` lub `bash`.

Komenda zazwyczaj składa się z nazwy programu i, w razie potrzeby, **argumenty** — dodatkowe słowa po niej: w `python --version` sam w sobie `python` — drużyna, a `--version` jest argumentem.

Ścieżka, system plików i aktualny folder

Shell zawsze „stoi” **obecna działalność folder** (current directory). **Ścieżka** (path) to adres pliku lub folderu w systemie plików. Ścieżka może wyglądać:

- **absolutny** — zaczyna się od korzenia dysku, np. `C:\Users\anna\project` lub `/home/anna/project`, i działa z dowolnego miejsca;

- **krewny** jest na przykład liczona z bieżącego folderu `./project` czy po prostu `project`.

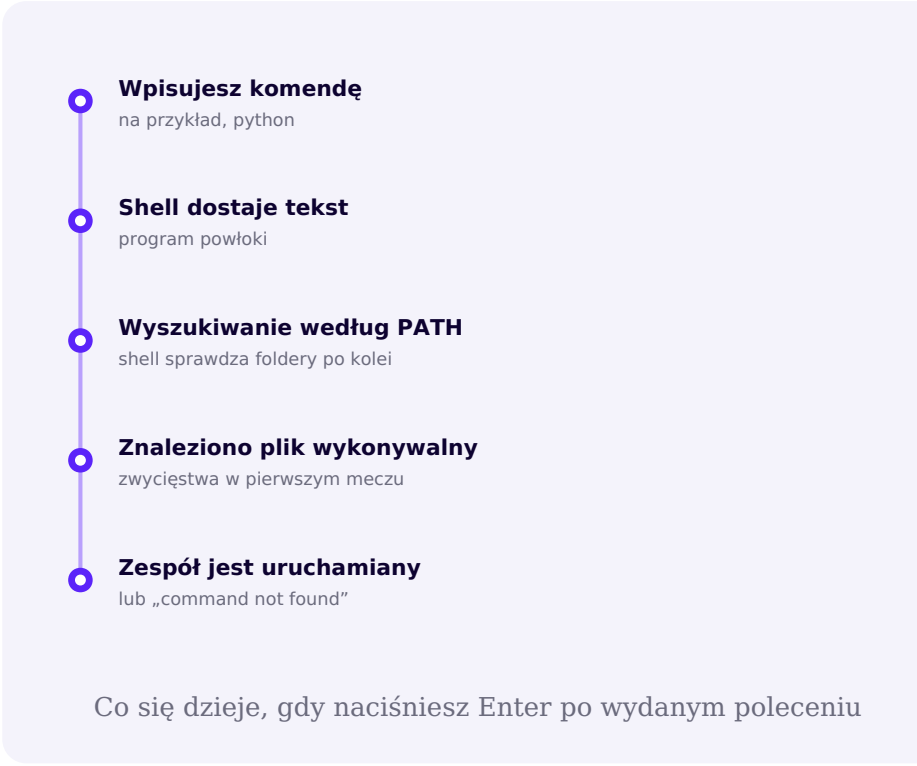
PATH zmienna środowiskowa

Kiedy wpisujesz krótką komendę typu `python` komputer nie szuka pliku na całym dysku — to zajęłoby zbyt dużo czasu. Zamiast tego shell patrzy na specjalny folder lista — **zmienna środowiskowa PATH** — i sprawdza je względem kolejki w określonej kolejności, aż znajdzie plik wykonywalny o tej nazwie.

PATH (Linux/macOS, uproszczone)

```
/usr/local/bin
/usr/bin
/bin
~/.local/bin
```

Jeśli wybierzesz numer `python`, shell sprawdzimy `/usr/local/bin/python`, więc `/usr/bin/python` i tak dalej, i uruchamia pierwszy znany plik. Jeśli żaden z folderów nie PATH takim plikiem Nie – zobaczysz błąd taki jak `command not found` (macOS/Linux) lub `'python' is not recognized...` (Windows).



Co się dzieje, gdy naciśniesz Enter po wydanym poleceniu

Typowy błąd

Jeśli Twój komputer ma **dwa różne Python**, a w PATH nie są one wymienione w oczekiwanej kolejności – `python` może mieć zupełnie inną wersję niż ta, którą właśnie zainstalowałeś. Rozdział 2.6 nauczy cię, jak to sprawdzić.

Oficjalne źródło

Szczegółowe wyjaśnienie zmiennych środowiskowych znajduje się w dokumentacji Twojego systemu operacyjnego; o samym Python zobacz sekcję "Using Python on Unix platforms" / "Using Python on Windows" na docs.python.org.

Instalacja Python na Windows

Cztery kroki: pobierz z python.org, zaznacz PATH, zainstaluj, sprawdź.

Istnieją dwa oficjalne sposoby na umieszczenie Python: klasycznego instalatora na Windows `.exe`, który istnieje od wielu lat, i nowy **Python install manager** — lekki narzędzie do zarządzania kilkoma wersjami Python, które Python Software Foundation promuje jako

główny sposób instalacji na Windows począwszy od ostatnich wydań. Traditional-instalator **pozostaje dostępny na Linie 3.14 i 3.15**, więc obie opcje to nie błąd, a dwie oficjalne Jak.

Dla tego kursu
Będziemy używać klasycznego instalatora `.exe` dla pierwszego znajomego jest to bardziej przejrzyste i daje dokładnie ten sam efekt: pracujący zespół `python` w terminalu.

Krok 1. Pobierz instalator

Open **python.org/downloads** jest jedynym źródłem, które Zaufaj podczas pobierania Python. Sama strona wykrywa Twój system operacyjny i oferuje Obecna wersja stabilna.

Strona do pobrania Python Windows na python.org

python.org/downloads/windows — oficjalna strona pobierania dla Windows.

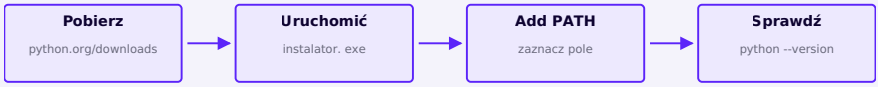
Pobieraj tylko z python.org
Strony firm trzecich czasami oferują „przyjazne użytkownikowi”

Krok 2. Uruchom instalator

Uruchom pobrany plik. Na pierwszym ekranie instalatora znajduje się najważniejszy znaczek w całym pliku Instalacja:

Nie przegap: „Add python.exe to PATH”
Na dole pierwszego ekranu instalatora jest pole wyboru **Add python.exe to PATH**. Koniecznie go zainstaluj. Jeśli pominiesz ten krok, Python zostanie zainstalowany, ale `python` w terminalu nie będzie działać — a powód będzie całkowicie nieoczywisty, jeśli jeszcze nie wiesz, co to jest PATH (omówiliśmy to w sekcji 2.2).

Następnie kliknij **Install Now** do standardowego montażu – odpowiedni dla W zdecydowanej większości przypadków i instalacji Python, pip oraz standardowej biblioteki w Profil użytkownika.



Cztery kroki instalacji Python na Windows

Krok 3. Sprawdź instalację

Otwórz PowerShell (lub zwykły wiersz poleceń) i wpisz:

```PowerShell
python --version
```

Powinieneś zobaczyć coś takiego `Python 3.14.7`. Jeśli zamiast Terminal odpowiada, że polecenie nie zostało rozpoznane — najprawdopodobniej przegapiłeś zaznaczenie «Add python.exe to PATH». Rozwiązaniem jest ponowne uruchomienie instalatora i wybór **Modify**, ponownie zaznaczając tę opcję, lub zainstalować od nowa.

python czy py?

Zespół często pracuje dla Windows `py` jest osobnym launcherem Python, który jest instalowany natywnie wraz z interpreterem i może uruchomić wybraną wersję, nawet jeśli jest ich kilka (na przykład, `py -3.14`). Do tego kursu użyj `python` — ale nie martw się, jeśli zobaczysz `py` w czyichś innych instrukcjach.

O Microsoft Store

The Microsoft Store również ma Python – to oficjalny pakiet, ale ma swoje specyficzne cechy Pracuj z uprawnieniami do plików, które czasem zaskakują początkujących. Do tego kursu Polecamy instalatora z python.org — jest bardziej przewidywalny i bliższy temu, co spotkasz na prawdziwych projektach i innych systemach operacyjnych.

Instalacja Python na macOS

Oficjalny uniwersalny instalator – i dlaczego lepiej nie dotykać systemu Python.

Na współczesnym macOS już istnieje drużyna `/usr/bin/python3` — ale To nie jest „twoja” **Apple- Narzędzie kontrolowane**: nadchodzi wraz z Xcode / Command Line Tools for Xcode i jest zaprojektowany z myślą o wewnętrznych potrzebach macOS i oprogramowanie firm trzecich, które na nim polega. Zazwyczaj jest zauważalnie starszy i niekompletna wersja Python niż obecna wersja z python.org. Install od firm trzecich Pakiety, aktualizowanie lub usuwanie to zły pomysł: można z niego odłączyć narzędzia zależny.

> **Nie dotykaj Apple-Python**
> Nigdy nie instaluj pakietów „na wierzch” `/usr/bin/python3` i nie odinstalowuj go – to oficjalna rekomendacja, docs.python.org: ten Python Apple- jest kontrolowany i używany przez oprogramowanie systemowe i zewnętrzne. Zamiast tego umieść osobną, „własną”

Krok 1. Pobierz instalator

Open **python.org/downloads/macos**. Oficjalnym instalatorem macOS jest **uniwersalny (universal2)** pakiet `.pkg` : jedno i drugie Ten sam plik działa na Mac z procesorem Apple Silicon (M1/M2/M3/M4)), a na Mac z procesorem Intel.

python.org/downloads/macos - Oficjalna strona do pobrania macOS.

Krok 2. Uruchom instalator

Otwórz pobrane `.pkg`-plik i przejdź przez standardowy kreator instalacji macOS (Wprowadzenie → Licencja → Lokalizacja instalacji → Instalacja). W przeciwieństwie do Windows, tutaj nie ma osobnego zaznaczenia dotyczącego PATH — instalator macOS sam konfiguruje wszystko, co potrzebne.

Krok 3. Sprawdź instalację

Otwórz aplikację **Terminal** i wejdź:

```zsh
python3 --version
```

Dlaczego python3, a nie python?
Na współczesnym macOS (i w Linux) drużynie `python` bez sufiksu zazwyczaj w ogóle nie istnieje. Tak nie było zawsze: w bardzo starych wersjach macOS (do Catalina, rok 2019) polecenie `python` historycznie wskazywał na system Python 2 — ale to już przeszłość, a nie obecne zachowanie. Oficjalny instalator python.org nie ustawia polecenia dokładnie tak jak `python3`. Omówimy, jak zrobić polecenie `python` wygodne w wewnętrznym środowisku wirtualnym, w rozdziale 2.10.

Powinieneś zobaczyć coś takiego `Python 3.14.7`. Jeśli terminal pokazuje starszą wersję (na przykład, `Python 3.9.6`) to, prawdopodobnie sterowane przez Apple- `/usr/bin/python3`, nie o to interpreter, który właśnie zainstalowałeś; sprawdź kolejność PATH (sekcja 2.4) lub jawnie Określ ścieżkę przez `sys.executable` (Rozdział 2.6), aby zapewnić, że który `python3` faktycznie jest uruchomiony.

Instalacja Python na Linux

Linux to pełnoprawna, pierwszorzędna platforma dla Python, z własnymi zasadami dobrych manier: menedżerem pakietów systemowych i obowiązkowymi środowiskami wirtualnymi.

Linux jest tak wysokiej klasy platformą dla Python jak Windows macOS i wielu profesjonalne zespoły, to Linux — główny system działania (i prawie wszystkie serwery, na którym działa Twój przyszły kod, również Linux). Podobnie jak macOS, większość dystrybucji Linux już włączyli system Python — i ta sama zasada obowiązuje tutaj: **nie ruszaj systemowego Python**.

PEP 668: «externally-managed-environment»
W zakresie współczesnych dystrybucji (Ubuntu 23,04+, Debian 12+ i inne) zespół `pip install` celowo bez wirtualnego środowiska **odmówi pracy** i pokaże błąd `error: externally-managed-environment` . To nie jest błąd – to ochrona opisana w oficjalnym standardzie PEP 668, aby przypadkowo nie zepsuć narzędzi systemowych zapisanych również w Python. Poprawną odpowiedzią na ten błąd nie jest „ominięcie”

Instalacja Menedżera Pakietów

Na Linux Python zwyczajowo instaluje się ją przez menedżera pakietów systemowych dystrybucji, a nie Pobranie instalatora osobno to standardowa, oficjalnie zalecana metoda.

Debian / Ubuntu

```
sudo apt update
sudo apt install python3 python3-venv python3-pip
```

Fedora

```
sudo dnf install python3 python3-pip
```

Arch Linux

```
sudo pacman -S python python-pip
```

Dlaczego stawiać osobny python3-venv?
Debian/Ubuntu moduł do tworzenia środowisk wirtualnych bywa czasem umieszczany w osobnym pakiecie `python3-venv` — bez niego drużyna `python3 -m venv` (Rozdział 2.11) nie zadziała. Jeśli planujesz korzystać ze środowisk wirtualnych (a my będziemy), zainstaluj je od razu.

Sprawdzanie instalacji

bash

```
python3 --version
pip3 --version
```

Dlaczego jest tak surowo Linux

Sama system operacyjna na Linux często jest częściowo napisana w Python i używa systemowego interpretera do swoich narzędzi (menedżera pakietów, narzędzi systemowych). Jeśli wydasz polecenie `pip install` bez środowiska zastąp wersję biblioteki, iz której zależy sam system — można zepsuć coś ważnego w systemie operacyjnym. PEP 668 — to standard branżowy, który chroni właśnie przed tym scenariuszem, a nie kaprys konkretnej dystrybucji.

Praktyczne zakończenie
Linux zasada „zawsze pracuj w środowisku wirtualnym”

Który Python tak naprawdę się rozbiega

To samo pytanie uchroni cię przed połową przyszłych problemów środowiskowych: „Który tłumacz obecnie pracuje?”

Jeśli komputer może mieć wiele Python (system + zainstalowany przez Ciebie i późniejsze wirtualne środowiska), prędzej czy później pojawia się pytanie: **który Python obecnie jest zarządzany przez `python`**? Dobra wiadomość — sam Python potrafi odpowiedzieć na to pytanie.

sys.executable jest najbardziej szczerą odpowiedzią

Każdy działający interpreter Python zna dokładną ścieżkę do swojej własnej pliku wykonywalnego. Jest przechowywany w `sys.executable`.

python

```
import sys
print(sys.executable)
print(sys.version)
```

Przeprowadź to na dwa sposoby i porównaj wynik – z normalnego terminala i (gdy Przejdźmy do VS Code, sekcja 2.8) z wbudowanego terminala edytora. Jeśli ścieżki są różne — Oznacza to, że w tych dwóch miejscach *różne* tłumaczy, i to wiele wyjaśnia, jeśli pakiet jest „zainstalowany”

which / where – skrócona wersja z terminala

Nie uruchamiając Python wcale, można zapytać samego shell, który plik znajdzie jako pierwszy według PATH (przypomnijmy rozdział 2.2):

macOS / Linux

```
which python3
```

Windows (PowerShell)

```
where.exe python
```

Laboratorium: „dwa Python”

Jeśli Twój komputer ma zarówno system Python, jak i Python z python.org, uruchom `which python3` (lub `where.exe python`) i porównaj ścieżkę z tym, co się drukuje `sys.executable` od środka Python. Muszą się zgadzać – jeśli nie, to nie chodzi zmienna PATH o tłumacza, którego się spodziewałeś.

Oficjalne źródło

Więcej o `sys.executable` oraz `sys.version` — w module `sys` na docs.python.org.

VS Code: instalacja i rozszerzenia

VS Code sam nie rozumie Python — ta zdolność jest mu nadawana przez rozszerzenia. Przeanalizujmy to, co umieścić i dlaczego każde z nich jest potrzebne.

VS Code (Visual Studio Code) — darmowy edytor kodu od Microsoft. Sam w sobie, „od razu po wyjęciu z pudełka”, nie wie nic o Python — to uniwersalny edytor tekstu dla dziesiątek języków. Zrozumienie Python zapewniają mu **rozbudowa** (extensions) - indywidualnie Małe wtyczki, które instalujesz na edytorze.

Rozszerzenie ≠ pakiet

To są różne rzeczy, choć słowa są podobne. **Rozszerzenie VS Code** jest wtyczką instalowaną bezpośrednio w edytorze i dodającą nowe funkcje (podświetlanie składni, automatyczne uzupełnianie). **Pakiet Python** (Rozdział 2.12) to biblioteka kodu, którą instalujesz pip w określonym środowisku, aby móc korzystać z jej funkcji w programie. Rozszerzenia nie widzą ani nie zastępują pakietów i odwrotnie.

Instalacja VS Code

pobierz instalator z oficjalnej strony internetowej **code.visualstudio.com** dla twojego system operacyjny i przejście przez standardowego kreatora instalacji — nie ma tu specjalnych pułapek jak Pole PATH z sekcji 2.3.

Expansion Python – i to, co sama stawia

Otwórz zakładkę Extensions (ikonę czterokwadratową w pasku bocznym, lub `Ctrl+Shift+X`) i szukaj przedłużenia **Python** od Microsoft.

Python strona rozszerzenia (Microsoft) w VS Code Marketplace

Oficjalne rozszerzenie Python od Microsoft na VS Code Marketplace — ponad 232 mln instalacji.

Po instalacji tego rozszerzenia automatycznie pojawia się VS Code Marketplace więcej Trzy dodatki wraz z nim:

- **Pylance** — szybki analizator kodu (autouzupełnianie, podpowiedzi typów, przejście do definicji);
- **Python Debugger** — osobny silnik do krokowej debugizacji;
- **Python Environments** - Nowy interfejs do tworzenia i przełączania Wirtualne środowiska bezpośrednio z VS Code.

Rozszerzenie Jupyter

Jeśli planujesz pracować z notatnikami (notebooks, pliki `.ipynb` — już ich poznaliśmy podczas interaktywnej praktyki tego kursu), będziemy musieli się rozszerzyć **Jupyter**.

Jupyter stronę rozszerzenia w VS Code Marketplace

Oficjalne rozszerzenie Jupyter od Microsoft — ponad 108 mln instalacji.

Rozszerzanie Jupyter nie jest samym Jupyter

Deweloperzy szczerze ostrzegają na stronie rozszerzenia: **nie sedno Jupyter** (not a Jupyter kernel). Rozszerzenie to interfejs tylko w VS Code; aby faktycznie uruchamiać notebooki, potrzebujesz także środowiska Python z zainstalowanym pakietem `jupyter` (sekcja 2.15) — dokładnie tak samo, jak edytor i interpreter są rozdzieleni w sekcji 2.1.

Rozbudowa Ruff Projekt portfelowy

Ruff jest szybko standaryzowanym narzędziem do przeglądania i formatowania kodu z Astral Software (twórcy uv, Rozdział 2.13). Zastępuje kilka starych jednocześnie Narzędzia – Flake8, Black, isort – Jedno szybkie.

Strona rozszerzenia Ruff w VS Code Marketplace

Oficjalna ekspansja Ruff z Astral Software — ponad 4,3 miliona instalacji.

co umieścić w tabeli przestawnej

Rozbudowa	Wydawca	Dlaczego potrzebujesz	Status
Python	Microsoft	Podstawowe wsparcie języka: podświetlanie, uruchamianie, debugowanie, wybór interpretera	Wymagane
Pylance	Microsoft	Szybka analiza kodu, automatyczne uzupełnianie, wskazówki typu	Ustawia automatycznie z Python
Python Debugger	Microsoft	Osobny silnik debugowania krok po kroku (breakpoints, krok po kroku)	Ustawia automatycznie z Python
Python Environments	Microsoft	Nowy interfejs tworzenia i przełączania środowisk wirtualnych	Ustawia automatycznie z Python
Jupyter	Microsoft	Praca z notatnikami .ipynb wewnątrz VS Code	Polecam
Ruff	Astral Software	Natychmiastowa kontrola stylu i automatyczne formatowanie kodu	Polecam

Minimalny zestaw do tego kursu
Wystarczy zainstalować jedno przedłużenie **Python** – automatycznie dokręca Pylance, Python Debugger zarówno Python Environments. Jupyter, jak i Ruff – opcjonalnie, ale zalecamy oba.

VS Code: interpreter i przepływ pracy

Rozszerzenie jest zainstalowane, ale VS Code jeszcze nie wie, którego Python użyć dla Twojego konkretnego projektu. Przyjrzyjmy się, jak to określić i jak to działa.

Zainstalowane rozszerzenie Python jeszcze nie wie *który* interpreter używać w swoim projekcie — szczególnie jeśli na komputerze jest ich kilka (systemowy, z python.org, a wkrótce także środowiska wirtualne). Należy to wskazać wyraźnie — raz na projekt.

Wybór interpretatorów

Otwórz folder projektu w VS Code, a potem otwórz dowolny plik `.py`. W prawym dolnym rogu okna pojawi się inskrypcja z aktualnie wybraną wersją Python – kliknij na (lub otwórz paletę poleceń `Ctrl+Shift+P` / `Cmd+Shift+P` i wpisz „Python: Select Interpreter”

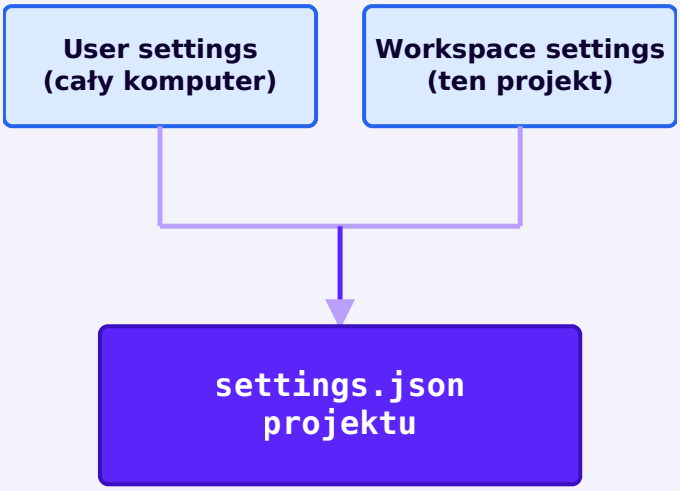
Pojawi się lista wszystkich interpreterów, którzy VS Code zostały znalezione na komputerze — z pełnymi ścieżkami do każdego. Wybierz potrzebny.

> **Test Bojowy**
> Wybrany interpreter powinien odpowiadać tej ścieżce, którą drukuje `sys.executable` (Rozdział 2.6), jeśli przepuścisz plik przez VS Code. Otwórz wbudowany terminal VS Code (`Ctrl+``) i porównaj.

Ustawienia użytkownika i ustawienia środowiska pracy

VS Code ma dwa poziomy ustawień z tymi samymi nazwami pól, ale różnym zasięgiem:

- **User settings** - Aplikuj do wszystkich projektów, na które się otwiera Ten komputer
- **Workspace settings** - Aplikuj tylko do obecnie otwartego folderu (projektu) oraz są przechowywane w `.vscode/settings.json` wewnątrz samego projektu.



Wybór interpretera jest powiązany z projektem, a nie „na zawsze”

Gdy wybierasz interpretera przez "Select Interpreter", wybór jest powiązany z projekt, a nie "na zawsze" dla całego komputera — różne projekty na komputerze mają cichość może być inna Python. Technicznie rzecz biorąc, rozszerzenie **Python Environments** (my instalowaliśmy go razem z rozszerzeniem Python w rozdziale 2.7) przechowuje to przypisanie w `.vscode/settings.json` projektu — nie jako bezpośredniej ścieżki do pliku, lecz jako jako odniesienie do pożądanego środowiska, które samo rozszerzenie odnajduje na nowo za każdym razem Otwarcie projektu:

.vscode/settings.json

```
{
  "python-envs.pythonProjects": [
    { "path": ".", "envManager": "ms-python.python:venv" }
  ]
}
```

A co z python.defaultInterpreterPath?
Starsze ustawienie `python.defaultInterpreterPath` — to nie jest zapis twojego wyboru, a **plan awaryjny**: rozszerzenie odczytuje go tylko wtedy, gdy żadne środowisko nie zostało przypisane do projektu w inny sposób. Ręczne zaznaczenie przez „Select Interpreter” `settings.json` nawet po wybraniu tłumacza.

Wbudowany terminal

Terminal wewnątrz VS Code (`Ctrl+``) — to ten sam shell, co zwykły terminal systemu operacyjnego (rozdział 2.2), po prostu otwarty bezpośrednio w edytorze, w folderze projektu. Kiedy dla projektu wybrane jest środowisko wirtualne, VS Code zwykle **aktywuje go automatycznie** w nowym wbudowanym terminalu zobaczysz `(.venv)` przed wywołaniem instrukcji poleceń (więcej informacji zobacz 2.11).

Uruchamianie i debugowanie pliku

Przycisk ▷ („Run Python File”

Run używa niewłaściwego Python"
Jeśli przepuszczanie pliku przez VS Code zachowuje się inaczej niż uruchamianie tego samego pliku w zwykłym terminalu, prawie zawsze wynika to z wybrania niewłaściwego interpretera. Otwórz Select Interpreter ponownie i porównaj wybraną ścieżkę z tym, co wypisuje `sys.executable` (Rozdział 2.6).

PyCharm: projekt, interpretator, środowisko

Specjalistyczne IDE dla Python – z innym modelem pracy niż VS Code, ale z tymi samymi podstawowymi ideami w środku.

PyCharm (JetBrains) to specjalistyczne środowisko programistyczne (IDE) specjalnie dla Python, w przeciwieństwie do VS Code, który jest wszechstronny i rozumie Python poprzez rozszerzenia. Począwszy od wersji 2025.1, PyCharm jest **pojedynczy produkt**: poprzedniego dzielenia na Community- i Professional-edycji już nie ma. Podstawowe, „codzienne” możliwości (w tym wsparcie Jupyter-notatników) są darmowe dla wszystkich, a płatna subskrypcja **Pro** dodaje bardziej zaawansowane narzędzia (na przykład do tworzenia stron internetowych i baz danych); gdy Przy pierwszej instalacji każdy użytkownik otrzymuje bezpłatny okres próbny Pro. Na ten kurs, Zdecydowanie wystarczająco dużo darmowych funkcji.

Jeśli natkniesz się na „PyCharm Community”
Do połączenia w 2025 roku PyCharm rzeczywiście miał osobne edycje Community i Professional — nadal możecie spotkać te nazwy w starych artykułach i filmach. Istoty to nie zmienia: bezpłatne podstawowe funkcje dzisiejszego PyCharma są bezpośrednim następcą dawnej edycji Community.

IDE ≠ interpreter
Podobnie jak w VS Code (Rozdział 2.1): PyCharm sam nie jest interpreterem Python-. To program, który znajduje już zainstalowany interpreter na twoim komputerze (lub pomaga stworzyć nowe środowisko wirtualne) i używa go do uruchamiania oraz analizy kodu.

i jego interpretator

W PyCharm jednostce pracy — **projekt** (Project): plus folder kodu Ustawienia powiązane konkretnie z tym światem. Tworząc nowy projekt, PyCharm od razu pyta Którego interpretera użyć, oraz oferuje wygodny przycisk do stworzenia nowego wirtualnego środowiska specjalnie dla tego projektu (co to oznacza w sekcji 2.10).

Interpretera można zmienić później przez **Settings → Project → Python Interpreter** — tam można zobaczyć pełne lista każdego pakiety zainstalowane w obecnym środowisku, co jest wygodne do szybkiej weryfikacji.

PyCharm Wtyczki

Podobnie jak VS Code, PyCharm ma system wtyczek – ale większość podstawowego wsparcia Python jest już zbudowany od razu, bez konieczności instalowania dodatkowych rozbudów. Via **Settings → Plugins** można dodać wsparcie dla innych języków i narzędzi; wsparcie dla notatników Jupyter w jednym produkcie PyCharm również jest wliczone w cenę.

VS Code czy PyCharm?

	VS Code	PyCharm (bezpłatne możliwości)
Co to jest	Uniwersalny Edytor + Rozszerzenia	IDE szczególnie dla Python

	VS Code	PyCharm (bezpłatne możliwości)
Obsługa Python	Przedłużenia przejściowe (sekcja 2.7)	Wbudowane natywnie
Waga i prędkość rozruchu	Zapalniczki	Cięższy
Inne języki	Doskonale – dziesiątki języków	Głównie Python (+wtyczki)
Dla tego kursu	Zalecane	Też pasuje

Którą wybrać
Obie opcje są właściwym wyborem i obie są używane przez profesjonalistów. Jeśli już pracowałeś w VS Code, zostań w niej. Jeśli chcesz środowiska „wyostrzonego”

Środowiska wirtualne – dlaczego są potrzebne

Zanim wpiszesz polecenie do stworzenia środowiska, przeanalizujmy problem, który ono rozwiązuje. Bez tego polecenie to tylko kolejna linia do zapamiętania.

Zanim pokażemy polecenie tworzenia środowiska wirtualnego, przyjrzyjmy się, do czego służy Ogólnie rzecz biorąc, jest to konieczne – w przypadku konkretnego problemu, z którym prędzej czy później wszyscy się zmierzą.

Wyobraź sobie następującą sytuację

Pracujesz nad dwoma projektami na tym samym komputerze:

- **Projekt A** to stary projekt edukacyjny wykorzystywany przez bibliotekę `requests` wersja 2.25 (napisana dawno temu, na razie ją aktualizuj nie ma takiej potrzeby);
- **Projekt B** jest nowym projektem, który potrzebuje świeżości `requests` wersji 2.32 — są funkcje, których nie było w Stara wersja.

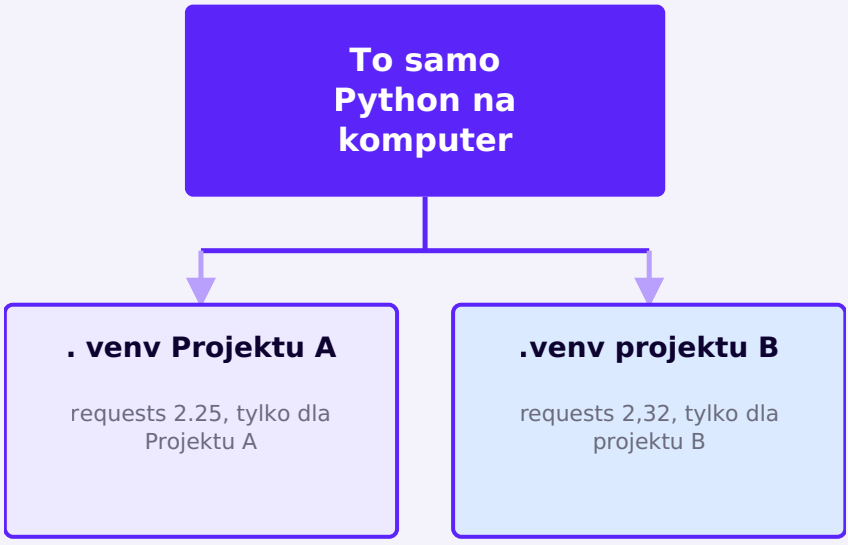
Gdyby istniał tylko jeden, współdzielony przez cały komputer Python własnym zestaw pakietów — Instalacja obu wersji tej samej biblioteki jednocześnie byłaby niemożliwa. Zainstalowanie nowej wersji Projektu B automatycznie przerwało Projektowi A.

To nie jest hipotetyczny scenariusz

Ten problem jest głównym powodem, dla którego środowiska wirtualne stały się standardem w świecie Python. Bez nich dwa projekty na tym samym komputerze muszą dzielić te same wersje wszystkich bibliotek — a każda aktualizacja jednego projektu grozi cichym uszkodzeniem drugiego.

Rozwiązanie: Własna kopia środowiska dla każdego projektu

Środowisko wirtualne (virtual environment, w skrócie venv) jest światłem, Izolowana „kopia”



Każdy projekt otrzymuje swoją izolowaną kopię — wersje pakietów nie kolidują

Technicznie rzecz biorąc, nie jest to pełna kopia samego interpretera (nie marnuje ani gigabajta miejsca na każdy projekt) — venv ponownie wykorzystuje główną zainstalowaną Python, ale przygotowuje ją pod potrzeby projektu osobny, niezależny folder z pakietami. Dla ciebie jako programisty różnica jest niezauważalna: W środowisku aktywowanym pakiety zainstalowane dla tego samego projektu po prostu nie są widoczne do drugiego.

> **Analogia**
> Wirtualne środowisko — jak osobne, oznakowane pudełko z narzędziami dla każdego projektu, a nie jedna wspólna półka dla wszystkich. Wzięli młotek z pudełka projektu A — projekt B nawet się o tym nie dowie.

Nie chodzi tylko o konflikty wersji

Środowiska wirtualne mają inne praktyczne zalety:

- **Powtarzalność** - lista pakietów projektów można zapisywać w pliku oraz Dokładnie przywracaj to samo środowisko na innym komputerze.
- **Czystość** - Usunięcie środowiska projektu jest tak proste, jak usunięcie jego folder, nie wpływając na resztę systemu;
- **Bezpieczeństwo systemów** na macOS, a zwłaszcza na Linux (sekcja 2.5), jest To także sposób, by w ogóle nie dotykać systemowego Python.

Następna sekcja to polecenie stworzenia środowiska i rozróżnienia, co jest ono prawdziwe tworzy go na dysku.

Stwórz . venv

Od polecenia, przez to, co faktycznie pojawia się na dysku, oraz jak włączać i wyłączać środowisko.

Teraz, gdy rozumiemy dlaczego, stwórzmy pierwsze środowisko wirtualne. Moduł `venv` jest zawarta w standardowej bibliotece Python – osobno Nic nie jest potrzebne (poza Linux, gdzie czasem wymagany jest pakiet systemowy `python3-venv`, zobacz sekcję 2.5).

Tworzenie środowiska

Otwórz terminal w folderze projektu i uruchom:

Windows / macOS / Linux

```
python -m venv .venv
```

Co oznacza kropka przed nazwą
Nazwa `.venv` zaczyna się od kropki na macOS i Linux jest to standardowa konwencja dla „ukrytych” `ls` bez flag nie są wyświetlane, aby nie zagracać lista plików serwisowych. Folder może być także nazwany bez kropki (na przykład `venv`), ale `.venv` jest najczęstszym porozumieniem i jest również domyślnie uznawane VS Code i PyCharm.

Co pojawiło się na płycie

Zespół stworzył folder `.venv` ze swoją wewnętrzną strukturą:

.venv/ (macOS / Linux)

```
.venv/
├── bin/
│   ├── python -> /usr/local/bin/python3.14
│   ├── pip
│   └── activate
├── lib/
│   └── python3.14/site-packages/
└── pyvenv.cfg
```

.venv\ (Windows)

```
.venv\
├── Scripts\
│   ├── python.exe
│   ├── pip.exe
│   └── activate.bat
├── Lib\
│   └── site-packages\
└── pyvenv.cfg
```

Najważniejsze jest tutaj `site-packages`: właśnie tutaj trafią pakiety, które zainstalujesz dla tego projektu, i właśnie ten folder jest odizolowany od innych projektów (rozdział 2.10). Plik `pyvenv.cfg` przechowuje odniesienie do „rodzica”

Aktywacja środowiska

Samo stworzenie środowiska nie oznacza, że jest ono wykorzystywane – jest konieczne **aktywuj** w obecnym terminalu:

macOS / Linux (bash/zsh)

```
source .venv/bin/activate
```

Windows (PowerShell)

```
.venv\Scripts\Activate.ps1
```

Windows (cmd.exe)

```
.venv\Scripts\activate.bat
```

Po aktywacji polecenie w wierszu poleceń się zmieni — przed nim pojawi się `(.venv)`. To jest sygnał wizualny: teraz polecenia `python` oraz `pip` w tym terminalu wskazują dokładnie na interpreter i pakiety tego środowiska, a nie na systemowy lub globalnie zainstalowany Python.

Terminal po aktywacji

```
(.venv) anna@laptop project %
```

Aktywacja – dla każdej nowej karty/okna terminala
Aktywacja jest ważna tylko w oknie terminala, w którym ją wykonałeś. Jeśli otworzysz nową zakładkę terminala, ponownie aktywuj środowisko. VS Code zwykle robi to automatycznie dla wbudowanego terminala (sekcja 2.8), jeśli ten interpreter jest wybrany dla projektu.

Dezaktywacja

Aby powrócić do normalnego, „globalnego”

Każdy system operacyjny

```
deactivate
```

Podstawowa praktyka

Stwórz i aktywuj swoje pierwsze środowisko

Utwórz folder o dowolnej nazwie, otwórz w nim terminal, uruchom `python -m venv .venv`, aktywuj go poleceniem dla twojego systemu operacyjnego i upewnij się, że zachęta w wierszu poleceń zmieniła się na `(.venv)`.

Zainstaluj pierwszy pakiet

pip install jest też diagnozą najczęstszego błędu początkujących: „zainstalowany, ale z jakiegoś powodu nie zaimportowany”

Po aktywacji środowiska (sekcja 2.11) możesz zainstalować pierwszy pakiet. Weźmiemy `requests` jest popularną biblioteką do żądań do serwerów WWW, Później używamy go w projektach kursowych.

pip install

Terminal (środowisko aktywowane)

```
(.venv) $ pip install requests
```

pip pobiera pakiet z **PyPI** (Python Package Index — poznaliśmy go w Rozdział 1) i rozpakowuje ją bezpośrednio do `site-packages` twoja aktywne środowisko (Rozdział 2.11).

Sprawdzanie instalacji

Terminal

```
pip show requests
pip list
```

python

```
import requests
print(requests.__version__)
```

Laboratorium: „Zainstalowane, ale import nie działa”

Jednym z najczęstszych i najbardziej mylących problemów dla początkujących: właśnie zainstalowałeś Pakiet, zobacz go w `pip list` — a `import` w kodzie nadal daje `ModuleNotFoundError`.

Prawie zawsze przyczyna jest jedna
Pakiet zainstalowany w **jedno** środowisko, a kod uruchamiany jest przez **inne**. Na przykład: aktywowałeś `.venv` do terminala i umieszcza tam pakiet, ale VS Code uruchamia plik przez globalny interpreter, ponieważ dla projektu wybrano zły Python (sekcja 2.8).

Kolejność diagnostyki:

1. Sprawdź `sys.executable` (Rozdział 2.6) bezpośrednio z miejsca Gdzie leży import – który interpreter jest faktycznie używany?
2. Sprawdź, co pokazuje komunikat terminala `(.venv)` — Czy środowisko jest naprawdę aktywne?
3. Wykonaj `pip show requests` w tym samym terminalu jest paczka Czy to naprawdę jest stąd widoczne?
4. Jeśli jesteś w VS Code — otwórz „Select Interpreter” (rozdział 2.8) i upewnij się, że wybrana jest ścieżka dokładnie do `.venv`, a nie do systemowego lub globalnego Python.

★★ Zadanie samodzielne
Znajdź i napraw „dwa środowiska”
Stwórz nowe środowisko `.venv2` obok istniejącego `.venv` (ale nie aktywuj go). W aktywnym `.venv` instalacja `requests`. Dezaktywuj środowisko, aktywuj `.venv2` i spróbuj `import requests` – Musisz zobaczyć `ModuleNotFoundError`. To ten sam scenariusz „zainstalowany, ale nie zaimportowany”

pip, pipx, venv, virtualenv, uv

Pięć podobnie brzmiących instrumentów — i bardzo różne zadania, które każdy z nich rozwiązuje.

Do tego momentu już korzystałeś `pip` oraz `venv` (Sekcje 2.11–2.12). W ekosystemie Python jest ich jeszcze kilka instrumenty o podobnym brzmieniu – analizujmy każdy osobno, a nie jako jedną listę, żeby nie było tego pomylić ich ze sobą.

pip

pip instaluje, aktualizuje i usuwa pakiety *do już istniejącego Środowisko aktywne*. Nie tworzy niczego, wypełnia tylko to, co już istnieje.

Terminal

```
pip install requests
pip install requests==2.31.0
pip uninstall requests
```

venv

venv jest standardowym modułem bibliotecznym (sekcja 2.11), który sprawia, że jedno: tworzy nowe, izolowane środowisko. Wchodzi do samego Python — by je ustanowić Nie musisz robić tego osobno.

virtualenv

virtualenv to pakiet firmy trzeciej, poprzednik `venv` (moduł venv w standardowej bibliotece w dużej mierze z niej wyrosł). Rozwiązuje to samo Task, ale historycznie był szybszy i wspierał nieco więcej scenariuszy (np. starsze wersje Python, gdzie nie było jeszcze wbudowanej venv). Na tym kursie jesteś całkowicie Dość zwykłych rzeczy `venv` — virtualenv warto znać z nazwy, jeśli napotkasz go w cudzym projekcie.

pipx

pipx rozwiązuje inny problem — nie "pakiet do mojego projektu", lecz "program z interfejs wiersza poleceń, którego chcę używać z dowolnego folderu." Przykład: Narzędzie Formatowanie kodu `black` – nie importujesz tego do swojego kodu, tylko ty Uruchom to jako komendę. pipx automatycznie tworzy osobny program dla każdego takiego programu oddzielne środowisko, ale udostępnia samo polecenie globalnie, z dowolnego folderu — bez Aktywacja.

Terminal

```
pipx install black
black my_script.py
```

uv

uv (od firmy Astral — to oni tworzą także Ruff, rozdział 2.7) — stosunkowo nowe narzędzie, które opisuje siebie jako jedną zamianę od razu dla pip, pip-tools, pipx, poetry, pyenv, virtualenv i innych. Jego główną zaletą jest szybkość: uv jest 10–100 razy szybszy od pip w typowych operacjach, ponieważ napisany jest w Rust i agresywnie buforuje pakiety.

Terminal

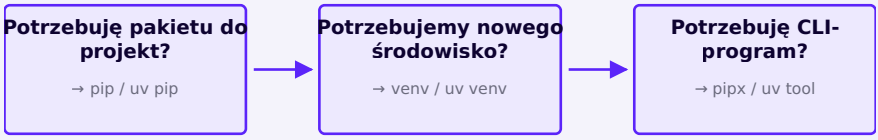
```
uv venv
uv pip install requests
uv run my_script.py
```

Oficjalne źródło
docs.astral.sh/uv — aktualna dokumentacja; w momencie pisania kursu stabilna wersja to 0.12.5.

Porównanie

Tool	Co robi	Instalacja pakietów	Tworzy środowiska
pip	Instaluje pakiety w obecnie aktywnym środowisku	Tak	Tak
venv	Tworzy izolowane środowiska (zawarte w standardowej bibliotece)	Tak	Tak
virtualenv	Starszy i szybszy zewnętrzny odpowiednik venv	Tak	Tak

Tool	Co robi	Instalacja pakietów	Tworzy środowiska
pipx	Ustawia CLI-aplikacje na Python globalnie, każdą — w swoim izolowanym środowisku	Tak (do aplikacji)	Tak (jeden na aplikację, automatyczny)
uv	Jedno szybkie narzędzie do wymiany pip, venv, virtualenv, pipx i innych	Tak	Tak



Trzy najczęściej zadawane pytania oraz narzędzie odpowiedzi na każde z nich

Czego używać na tym kursie

Pokażemy przykłady przez `pip` oraz `venv` — to standard, który działa wszędzie „od razu” i na którym najłatwiej zrozumieć, co się naprawdę dzieje. Jeśli później spodoba ci się szybkość uv — łatwo przełączyć: komendy niemal lustrzane.

conda, Miniconda, Miniforge, Anaconda

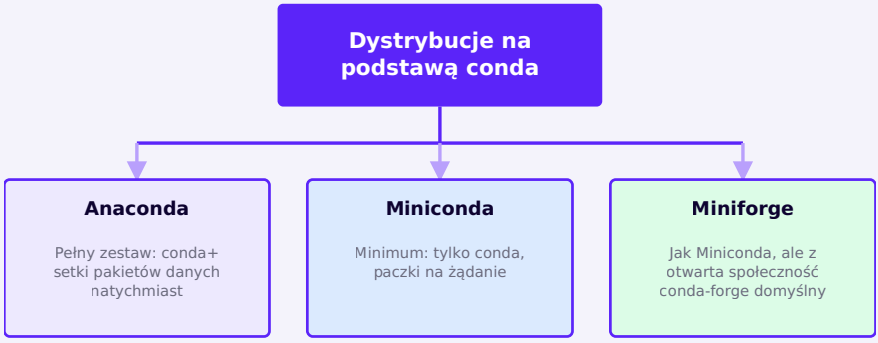
Kolejna rodzina narzędzi pochodząca z data science — ze swoim modelem środowisk i swoim podejściem do pakietów.

conda — jeszcze jeden menedżer pakietów i środowisk, pierwotnie stworzony dla data science i obliczeń naukowych przez firmę Anaconda, Inc.. Jego kluczowa różnica od pip: conda polega na tym, że potrafi instalować nie tylko pakiety Python-, ale także **nie-Python**

zależności – na przykład kompilatory, CUDA biblioteki dla kart graficznych, biblioteki systemowe dla Komputery naukowe to coś, z czym pip nie poradzi sobie sam.

Anaconda, Miniconda, Miniforge — jaka jest różnica

Wszystkie trzy są *sposoby, by conda przenieść na komputer*zamiast trzech różnych narzędzi:



Wszystkie trzy używają tego samego narzędzia conda — różni się tylko to, co jest „w zestawie”

- **Anaconda** jest najbardziej „trudna”
- **Miniconda** — oficjalny minimalny instalator: tylko sam conda, resztę instalujesz według potrzeby;
- **Miniforge** - Podobny do Miniconda, ale używany Repozytorium społeczności conda-forge zamiast domyślnego kanału z Anaconda, Inc.

Środowiska w conda

Terminal

```
conda create -n my-project python=3.14
conda activate my-project
conda install numpy pandas
```

Idea jest dokładnie taka sama jak w venv (Rozdział 2.10) – izolowane środowisko na projekt. Różnice Szczegółowo: conda przechowuje środowiska nie w folderze projektu, lecz w współdzielonym miejscu na dysku, a ty zwracaj się do nich po imieniu (`-n my-project`), nie po drodze.

Jak conda i pip żyją razem

W obrębie aktywowanego conda-okreslenia polecenie `pip` też działa – i często jest używany dla pakietów, które nie znajdują się w repozytoriach conda. Rule Dobre maniery: najpierw przepuść to `conda install` wszystko, co jest dostępna w ten sposób, a tylko reszta jest dostępna `pip install`, aby nie pomieszać własnego systemu rozwiązywania zależności conda.

Nie mieszaj venv i conda dla tego samego środowiska
conda i venv to dwa niezależne sposoby tworzenia odizolowanych środowisk. Nie próbuj aktywować venv *w środku* conda- otoczenia (lub odwrotnie) dla tego samego projektu — wybierz jedno narzędzie na projekt.

Kiedy wybrać conda

Praktyczne porady
Jeśli zajmujesz się analizą danych, nauką maszynową lub obliczeniami naukowymi i korzystasz z bibliotek z złożonymi zależnościami nie-Python (na przykład niektóre wersje PyTorch z GPU) wsparciem, conda jest często wygodniejsze. Dla tego kursu i większości typowych projektów Python pakiet venv + pip z sekcji 2.10–2.13 jest całkiem wystarczający.

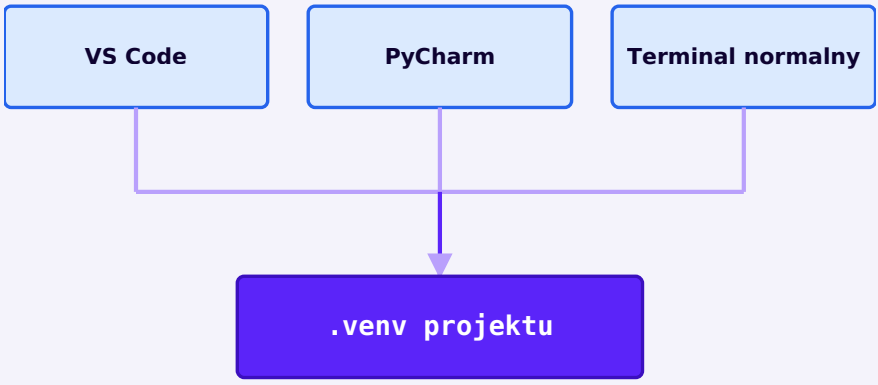
Jak IDE, Python i środowisko są powiązane

Połączenie wszystkich części rozdziału w jeden spójny obraz — w tym częsty źródło zamieszania: różnica między zeszytem a podstawowym Jupyter.

Zainstalowaliśmy Python (2.3–2.5), edytor kodu (2.7–2.9) i nauczyliśmy się tworzyć środowiska (2.10–2.12). Teraz połączmy to wszystko w jedną spójną całość — jak te części są naprawdę ze sobą powiązane.

Główna idea: IDE nie jest właścicielem środowiska

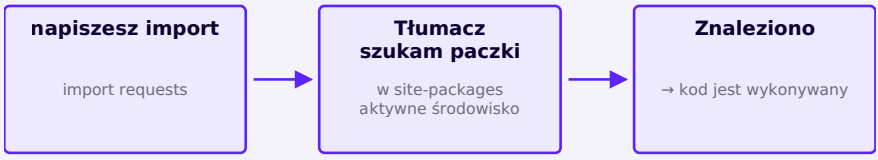
Warto powiedzieć to wprost: **VS Code PyCharm nie „zawstydzają”** — one jedynie znajdują na dysku już istniejący interpreter (i jego środowisko) i uruchamiają go w swoim imieniu. To samo środowisko `.venv` może być równie skutecznie stosowany nawet z VS Code, z PyCharm czy z zwykłych terminalny — i efekt będzie ten sam, bo to nie edytor naprawdę działa, lecz tłumacz.



Trzy różne „frontendy”

Model prostego procesu

Gdy uruchamiasz plik, nie ma znaczenia, który edytor czy terminal – dzieje się dokładnie to samo To samo:



Prosty model: interpreter szuka pakietu tylko we własnym środowisku

Notatniki: notebook i kernel — też nie to samo

Z zeszytami (`.ipynb`) dodana jest kolejna warstwa, która często jest Dezorientacja: **notebook** jest sam plik z kodem i komórkami tekstowymi, oraz **kernel** (rdzeń) to proces Python tła, który faktycznie jest wykonuje kod z komórek. Ten sam plik notatnika można otworzyć i podłączyć do różnych na przykład jądra z jądra ze środowiska Projektu A lub Projektu B.

Rozszerzenie Jupyter w VS Code (Rozdział 2.7) wyraźnie ostrzega: **nie jest rdzeniem** sam w sobie jest po prostu interfejsem, który pokazuje komórki i wysyła je kodu do wybranego jądra. Aby jądro w ogóle pojawiło się na liście wyborów, w odpowiednim środowisku musi być zainstalowany `jupyter`:

```
Terminal (środowisko aktywowane)
pip install jupyter
```

Potem, gdy otworzysz `.ipynb`-Wpisz VS Code po prawej W górnym rogu możesz wybrać jądro — coś w rodzaju `Python 3.14.7 ('.venv': venv)` , wyraźnie pokazując, z którego okrążenie zostało zdobyte.

„W notesie są inne paczki niż w terminalu”
Jeśli `import` działa w zwykłym pliku . py-, ale nie w notatniku (lub odwrotnie) — prawie zawsze wybiera się inny kernel dla notesu, czyli inne środowisko. Sprawdź wybór kernelu w prawym górnym rogu notesu, tak jak zaznaczyłeś „Select Interpreter”

Ostatnie zdjęcie rozdziału

Kto jest kim

- **Interpreter Python** jest programem, który faktycznie wykonuje kod.
- **Środowisko wirtualne** jest izolowanym zbiorem pakietów powiązanych z interpreterem.
- **VS Code / PyCharm** to redaktorzy, którzy znajdują i używają interpretera + środowiska, ale nie zawierają ich w sobie.
- **pip / uv** - Instaluj pakiety w obecnie aktywnym środowisku.
- **Notebook** — plikować z komórkami; **kernel** jest Python proces, który je wykonuje.

Typowe problemy i diagnoza

Osiem Nazwanych Laboratoriów – Katalog typowych problemów w tym rozdziale i sposoby ich rozwiązania.

Zebraliśmy osiem najczęstszych problemów tego rozdziału w jednym miejscu — jako odniesienie do których możesz wrócić, gdy coś pójdzie nie tak. Każdy z nich został już uporządkowany w trakcie rozdziału; Oto zwięzła wersja do szybkiej diagnozy.

Laboratorium 1.

«command not found: python»

Objaw: Komenda `python` nie znaleziono w terminalu.

Przyczyna i rozwiązanie: Python nie jest zainstalowane ani dodane do PATH (Windows: pola „Add python.exe to PATH” `python3` zamiast `python` (sekcja 2.4).

Laboratorium 2. «externally-managed-environment»

Objaw: `pip install` odmawia pracy dla Linux/macOS.

Przyczyna i rozwiązanie: To jest ochrona PEP 668 (rozdział 2.5): utwórz i aktywuj środowisko wirtualne (rozdział 2.11) i instaluj tam pakiety — nie do systemowego Python.

Laboratorium 3. ModuleNotFoundError po instalacji

Objaw: Pakiet jest zainstalowany, ale import upada.

Przyczyna i rozwiązanie: Pakiet jest zainstalowany nie w tym środowisku, z którego uruchamiany jest kod. Sprawdź `sys.executable` (Rozdział 2.6) z aktywnym środowiskiem terminalnym oraz z interpreterem wybranym w VS Code (Rozdział 2.8) – zobacz laboratorium Sekcji 2.12.

Laboratorium 4. VS Code uruchamia „niewłaściwy” Python

Objaw: Zachowanie przy rozpoczęciu z VS Code różni się od startu z terminala.

Przyczyna i rozwiązanie: Sprawdź „Select Interpreter” `.vscode/settings.json` projektu.

Laboratorium 5. Zapomniano aktywować środowisko

Objaw: Komunikat terminala się nie wyświetla `(.venv)` paczki są „zniknęły”

Przyczyna i rozwiązanie: Środowisko nie jest aktywowane w tym oknie terminala – aktywacja nie jest zapisywana między zakładkami (sekcja 2.11). Uruchom ponownie polecenie aktywacji dla swojego systemu operacyjnego.

Laboratorium 6.

Dwa Python - Konflikt wersji

Objaw: `python --version` pokazuje inną wersję niż ta, którą właśnie zainstalowałeś.

Przyczyna i rozwiązanie: Na komputerze jest kilka interpreterów, a PATH znajduje inny interpreter niż oczekiwano (Rozdział 2.2). Sprawdź kolejność w `which` / `where.exe` (Rozdział 2.6).

Laboratorium 7.

Notatnik używa innego jądra

Objaw: B `.ipynb` import nie znajduje pakietu, który jest dokładnie zainstalowany.

Przyczyna i rozwiązanie: Notes ma inne jądro (kernel) nie jest środowiskiem, w którym umieszczasz pakiet (sekcja 2.15). Zmień jądro w prawym górnym rogu notesu.

Laboratorium 8.

„pip: command not found”

Objaw: Zespół pip wciąż nie znajduje się w aktywowanym środowisku.

Przyczyna i rozwiązanie: Rzadka sytuacja uszkodzonego środowiska — utwórz je na nowo: usuń folder `.venv` i wykonaj `python -m venv .venv` nowe (Rozdział 2.11).

Ogólna metoda diagnostyczna

Prawie wszystkie problemy tego rozdziału sprowadzają się do jednego pytania: **który dokładnie interpreter i jakie środowisko są aktualnie faktycznie używane?** Sprawdź to przede wszystkim — przez `sys.executable` (Rozdział 2.6), komunikat terminala `(.venv)` i wybór tłumacza w edytorze (Rozdział 2.8) — zanim podejrzesz coś bardziej skomplikowanego.

Rekomendacje Cartesian School i efekty

Trzy sposoby na zorganizowanie miejsca pracy — i ostatnia praktyka: pełna lista kontrolna do tworzenia miejsca pracy na własnym komputerze.

Trzy Zalecane Ścieżki Cartesian School

Wszystkie narzędzia w tym rozdziale — legalne, działające opcje. Oto trzy kombinacje, które rekomendujemy w zależności od twoich celów:

Ścieżka	Edytor	Zarządzanie pakietami	Dla kogo jest odpowiedni?
Standardowy	VS Code	venv + pip	Zalecamy dla tego kursu — minimum ruchomych części, maksimum zrozumienia
Szybko	VS Code ani żadnego innego	uv	Jeśli chcesz szybkości i jednolitego narzędzia od pierwszego dnia
Data science	PyCharm, VS Code lub Jupyter	conda / Miniforge	Jeśli już skupiasz się na analizie danych, ML lub obliczeniach naukowych

W razie wątpliwości

Zacznij od **standardowej ścieżce** to VS Code + venv + pip. Dokładnie tego użyliśmy we wszystkich przykładach w tym rozdziale i najłatwiej zrozumieć, co naprawdę dzieje się pod maską. Zawsze możesz później przejść do uv lub conda i pomysły się nie zmienią.

Praktyka: Zorganizuj miejsce pracy (local-required)

Ta praktyka jest **wymagane, aby działać na własnym komputerze**. W przeciwieństwie do interaktywnych ćwiczeń w przeglądarce z innych rozdziałów, nie możesz zainstalować Python (i nie musisz) emulować online — chodzi o to, by mieć Prawdziwe, działające miejsce pracy.

local-required

Przejdź każdy punkt po kolei na swoim komputerze i zaznacz wykonane. Nic nie trzeba wysyłać do sprawdzenia — to lista kontrolna dla Ciebie.

★★★ Obowiązkowa praktyka rozdziału

Lista kontrolna w miejscu pracy (18 kroków)

- ☐ Python zainstalowany, oraz `python --version` (lub `python3 --version`) drukuje wersję 3.x
- ☐ Zespół pracuje od **jak zwykle** terminal systemu operacyjnego (nie tylko z edytora)
- ☐ `sys.executable` wyświetla oczekiwaną ścieżkę do interpretera
- ☐ VS Code (lub PyCharm) zainstalowana i otwarta bez błędów
- ☐ Rozszerzenie jest Python zainstalowane w VS Code (lub potwierdzone jest wbudowane wsparcie dla PyCharm)
- ☐ W panelu bocznym VS Code widać, że wraz z Python pojawiły się Pylance, Python Debugger i Python Environments
- ☐ Utworzono folder testowy projektu
- ☐ W tym folderze `python -m venv .venv`
- ☐ Środowisko `.venv` aktywowany pomyślnie (w poleceniu terminala możesz zobaczyć `(.venv)`)
- ☐ W aktywnym środowisku wykonano `pip install requests` bezbłędne
- ☐ `import requests` działa w trybie interaktywnym `python`
- ☐ W VS Code dla folderu testowego wybrano interpreter właśnie z `.venv` („Select Interpreter”
- ☐ Utworzony plik `main.py` z `import requests`, a uruchomienie przez przycisk ▷ w VS Code odbywa się bez błędów
- ☐ Ten sam plik jest pomyślnie uruchamiany z wbudowanego terminala VS Code
- ☐ Ustawia się co najmniej jeden punkt przerwania, a debugger VS Code się na nim zatrzymuje
- ☐ Środowisko dezaktywowane przez polecenie `deactivate`, a potem `import requests` nie działa już w zwykłym terminalu (jeśli requests nie został zainstalowany globalnie) – potwierdza to izolację
- ☐ Środowisko ponownie aktywowane

- ☐ Możesz wyjaśnić własnymi słowami różnicę między tłumaczem, redaktorem, pip a środowiskiem wirtualnym — na głos, partnerowi do nauki lub sobie

Podsumowanie rozdziału

Co możemy teraz zrobić

- Zrozum siedem warstw między sprzętem a kodem — oraz że interpreter, edytor, pip i środowisko to cztery różne rzeczy.
- Zrozum, czym jest terminal, shell zmienna PATH i jak komputer znajduje polecenie po nazwie.
- Zainstalowaliśmy Python na swoim systemie operacyjnym — Windows, macOS lub Linux — i znamy jego cechy (Znak wyboru PATH-, systemowy Python, PEP 668).
- Wiemy, jak sprawdzić, który interpreter faktycznie działa przez sys.executable.
- skonfigurować edytor kodu — VS Code lub PyCharm — i wiedzieć, jak wybrać interpretera do projektu.
- Rozumiemy, dlaczego potrzebne są środowiska wirtualne, wiemy, jak tworzyć, aktywować i instalować w nich pakiety.
- Rozróżniamy pip, pipx, venv, virtualenv i uv — i wiemy, do jakiego zadania każdy z nich służy.
- Wiemy, jak działają conda i Anaconda/Miniconda/Miniforge rodzina oraz kiedy ją wybrać.
- Potrafimy diagnozować osiem najczęstszych problemów z instalacją i konfiguracją.
- Zebraliśmy pełne, działające stanowisko pracy programisty Python na swoim komputerze.

co dalej

Z gotowym stanowiskiem roboczym jesteś w pełni przygotowany do rozdziału 3 — tam zaczniemy pisać prawdziwy, treściwy kod.

Python VS Code PyCharm Jupyter PySH

Technologia rozdziału: Oficjalne logotypy używane do identyfikacji, brak oświadczenia o partnerstwie

ROZDZIAŁ 3 · STR. 70

Twój pierwszy program na Python

Jak naprawdę rozmawiać z plikami Python: i jak je uruchamiać, terminal i powłoki, PySH, interaktywne REPL, print()/input(), nazwy, błędy i traceback, debugger Jupyter — oraz pierwszy mały projekt.

Tworzenie i uruchamianie programów na Python

Program Python zaczyna się od pliku tekstowego w formie zwykłego tekstu. Przyjrzyjmy się, co to naprawdę oznacza — i jak stworzyć taki plik oraz go uruchomić.

Program na Python — to **plik tekstowy**. Nic magicznego: te same bajty, co w każdym `.txt` -pliku, tylko Python potrafi je czytać i zamieniać na działania.

Czym jest plik programu? py

Plik programu zawiera kilka elementów, które warto wyróżnić:

- **Kod źródłowy** (source code) — sam tekst programu, który pisze człowiek;
- **Nazwa pliku** — na przykład, `privet.py`; Rozbudowa `.py` jest umowa, na mocy której sami redaktorzy Python rozpoznać plik kodu Python;
- **Folder Projektu** jest miejsce na dysku, gdzie znajduje się plik (i być może inne) akta tego samego projektu);
- **Kodowanie** jest sposób, w jaki tekst jest przechowywany w formie bajtów. Python domyślnie odczytuje kod źródłowy w **UTF-8** — w uniwersalnym kodowaniu, które obsługuje praktycznie każdy alfabet, w tym cyrylicę. Dlatego w kodzie źródłowym Python można swobodnie pisać rosyjskie łańcuch znaków i komentarze — nie są potrzebne żadne specjalne ustawienia.

Plik nie jest programem, który „już działa”

Dopóki plik leży po prostu na dysku, nic się nie dzieje — to tylko tekst. Program zaczyna działać dopiero w momencie, gdy jest URUCHAMIANY — przekazany interpreterowi Python do wykonania. Różnica między „plik istnieje” a „plik jest wykonywany” — to samo rozróżnienie, co między przepisem na papierze a prawdziwym przygotowaniem potrawy.

Aktualny folder i ścieżka pliku

Aby uruchomić plik, shell (sekcja 3.2) musi wiedzieć, gdzie się znajduje. Jeśli jesteś *w tym samym folderze*jako plik, jego nazwa wystarczy. Jeśli nie, potrzebujesz ścieżki: pełnej (z korzenia dysku) lub relacji (z aktualnego folderu). Analizowaliśmy ścieżkę i aktualny folder Więcej szczegółów w rozdziale 2.

Prosty model: Jak Python uruchamia program

Poziom 1: Plik do wyniku

Wewnątrz „Python działa”

Poziom 2: Co CPython robi wewnętrznie – kompiluje, a następnie wykonuje

Co się naprawdę dzieje
CPython (implementacja Python, której używamy) najpierw przekształca Twój kod źródłowy w pośredni, bardziej kompaktowy format — **bajtcode** — a potem wykonuje go na własnej maszynie wirtualnej. To nie znaczy, że Python „nigdy się nie kompiluje”

A co z __pycache__?
Kiedy później zaczniesz *importować* pliki jako moduły do innych plików (rozdziały o funkcjach i modułach), zauważysz, że `__pycache__` — Python przechowuje tam już skompilowany bajtkod, dzięki czemu nie musi go przeliczać za każdym razem, gdy go uruchamiasz. Dla pliku, który po prostu uruchamiasz bezpośrednio (jako `privet.py` poniżej), ten folder zwykle nie jest tworzony – nie zdziw się, jeśli teraz nie jest widoczny.

Twój pierwszy program

privet.py

```
print(„Cześć, Python!”)
```

Podzielmy polecenie startu na części:

Terminal

```
python privet.py
```

- `python` jest plikiem wykonywalnym interpretera (który zainstalowaliśmy zob. rozdział 2);
- `privet.py` jest argument: który plik wykonać.

Po uruchomieniu powinieneś zobaczyć:

wyjście programu

```
Привет, Python!
```

Utwórz i uruchom w VS Code

1. Otwórz folder projektu przez **File → Open Folder** (ustawiliśmy VS Code w Rozdziale 2 — interpreter powinien być już wybrany dla tego folderu).

2. Utwórz nowy plik `privet.py`.
3. Wprowadź kod i kliknij trójkąt ▷ „Run Python File” w prawym górnym rogu — lub otwórz wbudowany terminal (`Ctrl+``) i wybierz `python privet.py`.

Powstanie i uruchomienie w PyCharm

1. Otwórz lub stwórz projekt z wybranym interpreterem Python (Rozdział 2).
2. Kliknij prawym przyciskiem myszy na folder projektu w panelu po lewej → **New → Python File**, nazwij go `privet`.
3. Wpisz kod i uruchom przez **Run → Run 'privet'** w górnym menu lub kliknij prawym przyciskiem w edytorze i wybierz **Run**.

O zielonym trójkącie obok if __name__

W niektórych cudzych samouczkach prosi się o szukanie zielonego trójkąta ▷ dokładnie obok linii `if __name__ == "__main__":` — ale dla programu tak prostego jak `privet.py`, ta konstrukcja wcale nie jest potrzebna. Na razie uruchom plik przez **Run** w górnym menu lub kliknij prawym przyciskiem myszy wystarczy. Co to oznacza `if __name__ == "__main__":`, wymyślimy to później, gdy naprawdę przyjdzie jej kolej.

Praktyka: stwórz i uruchom swój pierwszy program
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/03-01/index.html)

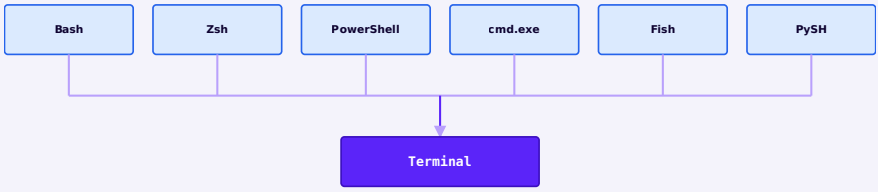
Terminal, shell i Python REPL — to różne rzeczy

Jedną z najczęstszych przyczyn zamieszania u nowicjuszy są te trzy pojęcia. Omówmy je raz na zawsze.

Już widzieliśmy słowa "terminalny" i "shell" w rozdziale 2 — ale żeby poruszać się pewnie Ponadto ważne jest, aby ostatecznie utrwalić to rozróżnienie i dodać do niego trzecie pojęcie: siebie **Python REPL**. Zamieszanie między tymi trzema rzeczami jest jedną z najczęstszych przyczyn, dla których początkującym wydaje się, że „zespół łańcuch znaków” to coś dużego i zagmatwanego.

Trzy różne rzeczy

- **Terminal** (terminal) to okno aplikacji, w którym Wejście i wyjście tekstu. Sam terminal niczego „nie rozumie”
- **Shell** (powłoka poleceń) — program W TERMINALU, który czyta twoje polecenia systemu operacyjnego: `cd`, `ls`, uruchamianie programów. Bash, Zsh, PowerShell, cmd.exe, Fish, PySH każdy z nich jest shell.
- **Python REPL** (znany też jako Python Shell) to osobny program (sam interpreter Python interaktywny), który URUCHAMIASZ z shell Drużyna `python`. Ma własne zaproszenie `>>>`, i rozumie już nie polecenia systemu operacyjnego, lecz kod na Python.



Inne shell – ten sam „ekran”

Oddzielna ścieżka: Terminal → → Python REPL Interpreter

Dlaczego wszyscy mylą "shell" i "Shell"
Istnieje niefortunny historyczny zbieg okoliczności: tryb interaktywny Python często nazywany jest w dokumentacji anglojęzycznej „Python Shell” `$` lub `>` zazwyczaj oznacza typowy shell systemu operacyjnego, a `>>>` jest dokładnie Python REPL.

Jak rozróżnić za pomocą oka

	Normalna shell (Bash/ PowerShell/...)	Python REPL
Zaproszenie	`$` , `%` lub `>`	`>>>`
rozumie	Polecenia systemu operacyjnego: `cd` , `ls` , uruchamianie programów	Kod na Python: wyrażenia, przypisania, wywołania funkcji
Jak dostać się	Otwórz terminal — jesteś już wewnątrz	Wybrać `python` w środku shell
Jak się wylogować	Zamknąć terminal lub `exit`	`exit()` czy Ctrl+D / Ctrl+Z

Sprawdź sam

Otwórz terminal i wpisz `python` . Zaproszenie zmieni się z zwykłego (na przykład, `$`) do `>>>` jest pewnym znakiem, że właśnie przeszedłeś z systemu operacyjnego shell na Python REPL. Dial `exit()` wrócić.

Rodzina Shell

cmd.exe, PowerShell, Bash, Zsh, Fish – krótko o tym, gdzie ich spotkasz i czym są ogólnie.

Skoro już o shell mówimy, przyjrzyjmy się krótko najczęstszym. Cel Częścią tej sekcji nie jest nauka składni każdego z nich, lecz rozpoznawanie ich przez Nazwij i zrozum, gdzie się z nimi spotkasz.

Shell	Gdzie się pojawia	Co to jest
cmd.exe	Windows (klasyka, Team łańcuch znaków)	Najstarsza powłoka poleceń Windows; wciąż spotykana w starych podręcznikach i systemach korporacyjnych

Shell	Gdzie się pojawia	Co to jest
PowerShell	Windows (Nowoczesny domyślny system)	Potężniejszy wrapper od Microsoft z obiektowym podejściem do poleceń; czego używaliśmy w Rozdziale 2 do Windows
Bash	Linux (często domyślnie), macOS (starsze wersje)	Jeden z najczęściej spotykanych shell na świecie — prawie każdy serwer i podręcznik ją zna
Zsh	macOS (domyślnie od 2019 roku), popularny na Linux	Podobny do Bash, ale z wygodniejszymi funkcjami „prosto z pudełka” (autouzupełnianie, motywy)
Fish	Linux/macOS (wybór użytkownika)	Przyjazna dla nowicjuszy powłoka z promptami i podkreśleniami już podczas rekrutacji zespołu
PySH	Linux/macOS/Windows (zainstalowane osobno)	Python-first powłoka — zamiast własnego minijęzyka poleceń używa prawdziwego Python (sekcja 3.4)

Dlaczego deweloper Python- miałby to wiedzieć

Nie musisz znać składni wszystkich shell — ale pomocne jest rozpoznanie ich na zaproszenie i w tytule, gdy natkniesz się na nie w artykułach, logach, CI/CD lub instrukcjach instalacji innych osób. Często instrukcja brzmi "zrób to w Bash" lub "w PowerShell" – i teraz dokładnie wiesz, co to znaczy.

PySH logo	**PySH** jest ostatnim łańcuch znaków powyższej tabeli, a następna sekcja jest mu poświęcona: jedyna skorupa tutaj napisana jako Python-first alternatywa, a nie kolejna wariant klasycznego języka shell-.

Nie będziemy szczegółowo opisywać składni każdej powłoki — to nie jest część kursu Python. Zamiast tego w następnej części poznamy PySH — ułożoną skorupę W przeciwnym razie: zamiast mini-poleceń używa rzeczywistych Python.

PySH: Python-first skorupa

Kolejne spojrzenie na linię poleceń — co by było, gdybyśmy zamiast mini-języka poleceń użyli prawdziwego Python?

PySH logo	`Oficjalne logo projektu PySH (pysh-shell.com) — używane do identyfikacji technologia omawiana na tym kursie, bez oświadczenia o partnerstwie czy sponsoringu.`

PySH jest prawdziwym programem zainstalowanym na tym komputerze, a wszystkie przykłady na nim Page jest prawdziwym, zweryfikowanym wnioskiem tego konkretnego settingu, a nie wymyślonym tekstem.

pysh-shell.com oficjalna strona internetowa

pysh-shell.com — oficjalna strona projektu PySH („The Python-Native Automation Platform”). Wersja na stronie (0.8.2) odpowiada wersji zainstalowanej na tym komputerze.

Tylko ten projekt

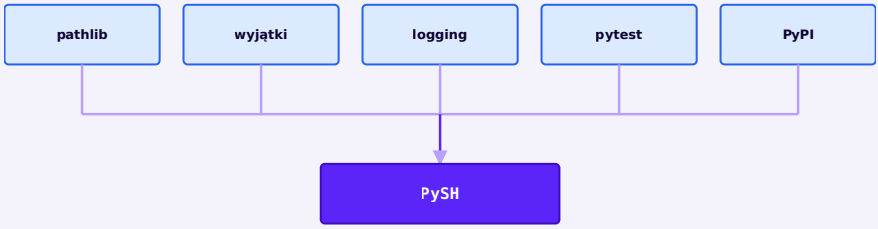
Na świecie istnieje kilka pakietów o podobnej nazwie „pysh” `pysh-shell`, zespół — `pysh`), a nie inne narzędzia o tej samej nazwie.

Co to jest PySH

PySH jest interaktywną powłoką (shell to Rozdział 3.3) napisaną w czystej formie Python i pozycjonowanie się jako **Python-first alternatywa** tradycyjnie shell wydaje się Bash. Oficjalny opis ze strony: "The Python-Native Automation Platform for Developers, System Administrators, and DevOps Engineers».

Dwa podejścia do linii poleceń

Tradycyjne podejście: terminal → shell-język → shell-skrypt



Python-first podejście PySH: zwykłych Python-tools bezpośrednio w wierszu poleceń

Tradycyjny shell doskonale nadaje się do krótkich interaktywnych zadań — ale im bardziej skomplikowany staje się skrypt, tym bardziej odczuwalny jest brak prawdziwego języka programowania: wyjątków strukturalnych, testów, czytelnych modułów. PySH proponuje inny kierunek: pisać automatyzację od razu w Python, nie przełączając się między dwoma różnymi językami.

Nie mylić z pełną wymianą Bash

PySH sam szczerze wskazuje w oficjalnej dokumentacji: **nie** pełna POSIX-kompatybilna zamiana `/bin/sh` nie jest klonem Zsh i nie gwarantuje zgodności z żadnym istniejącym skryptem Bash-. To samodzielny scenariusz z własnym, Python--orientowanym podejściem – a nie udawaniem, że prawdziwe Bash. jest ukryte w środku

Launch PySH – prawdziwe zakończenie

Na tym komputerze PySH już zainstalowane. Sprawdźmy to tymi samymi komendami, co ty Sprawdziłbym każdy inny program:

Terminal

```
command -v pysh
pysh --version
```

Rzeczywiste wyjście

```
/usr/bin/pysh
pysh 0.8.2
```

Przejdźmy PySH i zobaczmy prawdziwy baner oraz zaproszenie:

Terminal - Prawdziwa Sesja PySH

```
   PySH 0.8.2 | Python 3.13.5 | GPL-2.0-only
System: Debian GNU/Linux 13 | Kernel 6.12.101 | 11th Gen Intel
Core i3-1115G4 | RAM 19 GiB
Type 'exit' or press Ctrl+D to quit.
┌─   astra@soi — [~/Projects/Python_001] — git:feat/curriculum-
v2-chapter-03
│  py3.13 · uv0.11.24 · ruff0.15.20 · rust1.85.0 · node26.3.1 ·
npm11.16.0
└─❯
```

O zrzutach ekranu na tej stronie

Ta strona pokazuje rzeczywiste, dosłownie skopiowane wyjście faktycznie zainstalowanego PySH – wyjście terminala jest odtwarzane jako tekst, a nie jako zdjęcie ekranowe, ponieważ środowisko, w którym kurs został złożony, technicznie nie jest w stanie robić zrzutów ekranu graficznych okien. Każde polecenie na tej stronie zostało wykonane rzeczywiście.

Prompt PySH jest dwulinijowy i informacyjny: nazwa użytkownika, host, aktualny folder, gałęzi git, a druga łańcuch znaków z wersjami narzędzi projektowych (Python, uv, ruff i innych, jeśli zostaną znalezione). To ten sam duch co „zaawansowane”

PySH rozumie powszechne komendy

Pomimo Python-first filozofii, znane komendy również działają — PySH wie, jak uruchomić programy zewnętrzne, potoki i podstawowe operatory, jak zwykłe shell:

Prawdziwa sesja

```
└─❯ pwd
/home/astra/Projects/Python_001
└─❯ ls scripts | head -3
build_book.py
build_chapter_01.py
build_chapter_02.py
```

Python bezpośrednio w wierszu poleceń

I to właśnie odróżnia PySH od zwykłej shell: drużyny `py` wykonuje prawdziwy Python na miejscu, w stałym (utrzymującym się między połączeniami) Kontekst:

Prawdziwa sesja

```
└─❯ py print("hello from python")
hello from python
└─❯ py x = 10
└─❯ py print(x)
10
```

Zmienne i importy są zachowywane między poszczególnymi wywołaniami `py` — `x` z pierwszego polecenia jest dostępny w drugim. Dla dłuższego kodu jest też blok wieloliniowy:

Prawdziwa sesja

```
└❯ py {
import os
targets = [p for p in os.environ.get("PATH", "").split(":") if
p]
print(f"PATH entries: {len(targets)}")
}
PATH entries: 43
```

Istnieje również trzeci, bardziej „ciężki” tryb — pełnoprawny zagnieżdżony Python REPL ze swoim `>>>` zaproszeniem, które włącza się komendą `#py` bezpośrednio z zwykłego zaproszenia PySH:

Prawdziwa sesja

```
└❯ #py
PySH Python Command Execution Layer | GPL-2.0-only
Python 3.13.5
Type #help for commands, Ctrl+D or #exit to return to PySH.

>>> 2 + 2
4
>>> print("Привет, PySH!")
Привет, PySH!
>>> #exit
```

Bash-script i PySH-approach: jeden przykład

Mała ilustracja różnicy w stylu — nie po to, by ogłosić jeden sposób „złym”

Zadanie: Policz pliki w aktualnym folderze

Bash

```
count=$(ls | wc -l)
echo Plików: $count
```

PySH (py-blok)

```
py {
    import os
    count =
len(os.listdir('.'))
    print(f'Pliki: {count}')
}
```

Co używać dzisiaj: Dla krótkiego, jednorazowego polecenia Bash nadal szybciej się wpisuje. Ale gdy automatyzacja zostaje przepełniona warunkami, obsługą błędów i testami, czytelny Python zaczyna wygrywać. PySH daje się w ten sposób, nie zmuszając do przełączania się na osobny plik . py-plik.

Laboratorium

Lokalne: Spróbuj PySH (opcjonalnie)

To jest opcjonalny ćwiczenie dla twojego własnego komputera. Nie da się go zweryfikować Automatycznie i nie jest warunkiem koniecznym do Python kursu – PySH jest tego warte **wiedzieć**, a korzystać z niego dalej czy nie — decyzja należy do ciebie.

Nieobowiązkowo · lokalnie na twoim komputerze

Lista kontrolna: pierwsze wprowadzenie PySH

- ☐ Ustaw PySH: `pip install pysh-shell` (lub znajdź gotową instalację przez `command -v pysh`)
- ☐ Uciekaj `pysh` i spójrzcie na prawdziwy baner i zaproszenie
- ☐ Wykonaj jedno proste polecenie, na przykład `pwd` lub `ls`
- ☐ Wypróbuj polecenie `py print("hello")` - Python bezpośrednio z zaproszenia PySH
- ☐ Wyjdź poleceniem `exit` lub Ctrl+D

Oficjalne źródła

pysh-shell.com — strona internetowa projektu; pypi.org/project/pysh-shell — strona pakietu (`pip install pysh-shell`); pełna dokumentacja znajduje się w repozytorium projektu, link jest wymieniony na stronie PyPI-.

Tryb interaktywny Python (Python REPL)

Drugim sposobem pracy z Python jest wpisywanie kodu linia po linii i natychmiastowe zobaczenie efektu. Przyjrzyjmy się, jak to faktycznie działa.

Oprócz uruchamiania plików (sekcja 3.1), Python posiada drugi tryb działania — **interaktywny**. W niej wpisujesz kod linijka po linii i od razu widzisz bez pliku i bez wyraźnego „run”

Uruchomienie

Otwórz terminal (w VS Code, PyCharm lub jako osobna aplikacja — rozdział 3.2) i wpisz:

```
Terminal
python
```

Zobaczysz coś w stylu:

```
Rzeczywiste wyjście
Python 3.14.7 (main, Aug  5 2026, 00:00:00) [GCC ...] on linux
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

Zaproszenie `>>>` oznacza: Python jest gotowy i czeka na twoją drużynie.

Co oznacza R-E-P-L

REPL to skrót od **Read-Eval-Print Loop** („czytaj — wykonuj — drukuj — powtarzaj”

Pierwsze przykłady

Python REPL

```
>>> 2 + 2
4
>>> "Py" + "thon"
'Python'
>>> name = "Анна"
>>> name
'Анна'
```

Proszę zauważyć: łańcuch znaków `name = "Анна"` NIE pokazał wyniku — samo przypisanie nie jest wyrażeniem, którego znaczenie trzeba drukować. Ale Następne łańcuch znaków tam, gdzie właśnie pisaliśmy `name` już jest wyrazu twarzy, a REPL natychmiast je obliczył i wpisał.

Dlaczego REPL drukuje plik, a nie
REPL automatycznie pokazuje wynik *ostatnie obliczone wyrażenie* w każdej wprowadzonej linii to wygoda trybu interaktywnego. Normalny `.py` plik -jest wykonywany w całości i bezgłośnie — chyba że jest wyraźnie wywoływany gdziekolwiek `print()` nic nie pojawi się na ekranie, nawet jeśli gdzieś pośrodku pliku pojawi się wyrażenie takie jak `2 + 2` .

Twój shell potrafi liczyć

Gdy REPL natychmiast pokazuje wynik każdego wiersza, wygodnie jest go użyć jako szybkiego Kalkulator – Bez jednego `print()`:

Python REPL

```
>>> 2 + 2
4
>>> 10 / 4
2.5
>>> 3 * 7
21
```

Działa tylko w samej powłodzie: w zwykłej powłoce `.py` -plik łańcuch znaków `2 + 2` sam z siebie nic nie wywoła.

Praktyka: Interaktywny tryb laptopa
Interaktywny notatnik bezpośrednio w przeglądarce — porównaj komórkę Jupyter z Python REPL

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/03-02/index.html)

Praktyka: użyj Python jako kalkulatora
Interaktywny laptop bezpośrednio w przeglądarce – dodawaj, odejmuj, mnż, dziel

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/03-03/index.html)

Dwa rodzaje zaproszeń

- `>>>` — **główne** zaproszenie: Python czekając na start nowej drużyny;
- `...` — **zaproszenie do kontynuowania**: Python widzi, że bieżące polecenie nie zostało jeszcze ukończone (na przykład, nawias jest otwarty lub blok został rozpoczęty), oraz czeka na resztę.

Python REPL — przykład wieloliniowy

```
>>> total = (
...     2
...     + 2
... )
>>> total
4
```

Jak się wylogować

Wpisz `exit()` i kliknij Enter. klawiaturę Kombinacje: `Ctrl+Z` potem Enter na Windows, albo `Ctrl+D` na macOS/Linux.

REPL jako narzędzie badawcze

type(), help(), dir() to, jak zadawać Python pytania podczas pracy i jaka jest różnica między script, REPL, notebook a IDE.

REPL to nie tylko sposób na uruchamianie kodu, ale także wygodne narzędzie do ustawiania Python pytania podczas pracy. Trzy wbudowane komendy są szczególnie przydatne do badania.

type() — jakiego typu jest ta wartość?

Python REPL

```
>>> type(42)
<class 'int'>
>>> type("Python")
<class 'str'>
```

help() - wbudowana pomoc

Python REPL

```
>>> help(print)
```

Opis funkcji pojawi się bezpośrednio z oficjalnej dokumentacji Python – działa nawet bez niej Internet, bo pomoc jest wbudowana w samego tłumacza.

dir() — co w ogóle ma wartość

Python REPL

```
>>> dir(42)
```

Długi lista — to normalne

`dir()` pokaże długą lista nazw takich jak `__add__`, `__class__` i inne — teraz nie trzeba rozumieć każdego z nich. Sama przyzwyczajenie „zapytaj Python bezpośrednio” jest znacznie ważniejsze niż natychmiastowe zrozumienie każdej nazwy w odpowiedzi.

Praktyka: przewidzieć i sprawdzić

Interaktywny notatnik bezpośrednio w przeglądarce — type(), help(), dir() w praktyce

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/03-05/index.html)

Script vs REPL vs Notebook vs IDE

Spotkaliśmy już wszystkie cztery pojęcia osobno — połączmy je w jedną tabelę, aby widzieć pełny obraz:

	Script (.py)	REPL	Notebook (.ipynb)	IDE/editor
Co to jest	Zapisany plik kodu	Tymczasowa Sesja Interaktywna	Komórki kodowe + Tekst + Wyniki	Narzędzie do edycji/uruchamiania/debugowania
Czy jest przechowywana?	Tak, na dysku	Brak – znika przy wyjściu	Tak, w pliku .ipynb	—
Najlepsze dla	Programy gotowe do użycia	Szybkie eksperymenty	Szkolenia, badania danych	Pisanie kodu i debugowanie
Python-language?	Tak	Tak	Tak	No to narzędzie, które wykorzystuje Python

A co z praktyką przeglądania w tym kursie?

Interaktywne ćwiczenia, które wykonujesz bezpośrednio na tej stronie kursu, są bardziej Piąta opcja: prawdziwy Python jest realizowany w Twojej przeglądarce za pomocą technologii **Pyodide** (szczegóły można znaleźć w sekcji 3.16 dotyczącej notebook i kernel). To także Pełnoprawny Python — tylko z innym czasem działania, bez instalacji na komputerze.

Dane wyjściowe za pomocą Python

print() może robić więcej, niż się na pierwszy rzut oka wydaje: wiele wartości jednocześnie, separatory i kontrolę końca linii.

W pliku `.py` nic nie jest wyświetlane samo na ekranie, musisz Wyraźnie poproś polecenie o to `print()`. Już go użyliśmy w Rozdział 1 i REPL (Rozdział 3.5), przyjrzyjmy się teraz bliżej pierwszemu prawdziwemu narzędziu, które będziemy stosować w każdym programie tego kursu.

Wiele wartości w jednym poleceniu

print_demo.py

```
print(„Wiek:”, 10, „lat”)
```

Możesz przekazać dowolną liczbę wartości, oddzielonych przecinkami — `print()` wstawi między nimi odstęp i pokaże wszystko w jednej linii.

Parametr `sep` — separator

Domyślnie wartości są oddzielone przez spację. Można to zmienić przez `sep`:

print_sep.py

```
print("2024", "01", "15", sep="-")
```

wyjście programu

```
2024-01-15
```

Parametr `end` — jak zakończyć linię

Domyślnie każdy `print()` kończy się podłączeniem linii (`\n`) — więc następne połączenie zaczyna się na nowej linii. Parametr `end` pozwala to zmienić:

print_end.py

```
print(„Ładowanie”, end="")
print("...", end="")
print(„Gotowe!”)
```

wyjście programu

```
Загрузка...готово!
```

Pusty łańcuch znaków
Challenge `print()` bez argumentów po prostu drukuje pusty łańcuch znaków — przydatne do rozdzielania bloków tekstu od siebie.

Oficjalne źródło
Pełny opis wszystkich parametrów `print()` — w sekcji „Built-in Functions”

Praktyka: sep, end i formatowanie wyjścia
interaktywny laptop bezpośrednio w przeglądarce – print() opcje w praktyce
Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/03-04/index.html)

input(): program zaczyna rozmawiać z tą osobą

Pierwsze polecenie, które sprawia, że program staje się naprawdę interaktywny.

Do tej pory nasze programy mówiły tylko o – teraz nauczymy ich słuchać. `input()` — pierwsza komenda, która sprawia, że program staje się naprawdę interaktywny: wstrzymuje wykonanie i czeka, aż człowiek coś wpisze.

Pierwszy dialog

dialog.py

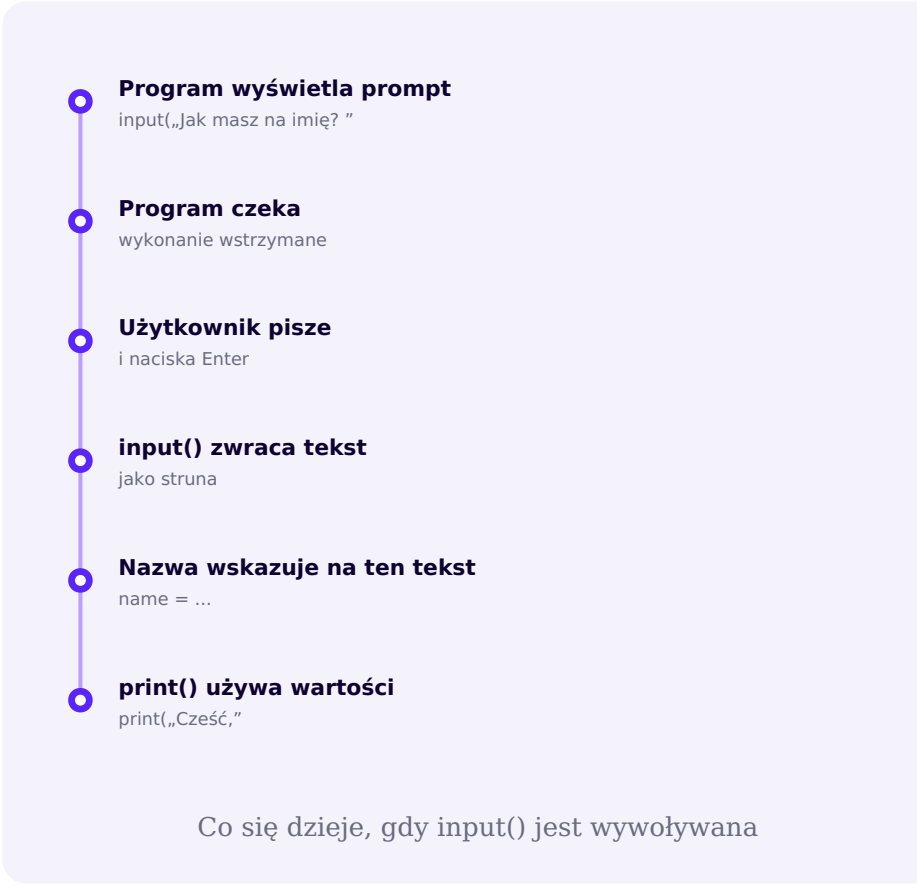
```
name = input(„Jak masz na imię?”)
print(„Cześć,”, name)
```

Po uruchomieniu w terminalu pojawi się komunikat, program poczeka na wejście, a potem Cię powita:

Terminal

```
Как вас зовут? Анна
Привет, Анна
```

Co dzieje się krok po kroku



Co się dzieje, gdy input() jest wywoływana

input() zawsze zwraca wiadomość

Nawet jeśli użytkownik wpisze liczby, `input()` zwroty **linia** (tekst), a nie liczba. Aby uzyskać liczbę rzeczywistą, tekst musi być jawnie przekonwertowany — na przykład za pomocą polecenia `int(...)`. W następnym rozdziale omówimy więcej o liczbach i konwersjach typów; tutaj wystarczy przypomnieć sobie sam fakt: wynik `input()` zawsze jest SMS.

Przykład konwersji (Jumping Ahead)

```
age_text = input(„Ile masz lat? ”)
print(int(age_text) + 1)
```

Praktyka: input() i pierwszy dialog
Interaktywny laptop bezpośrednio w Twojej przeglądarce – prawdziwe pole wejściowe

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/03-06/index.html)

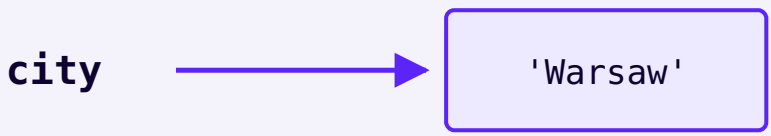
Imiona i znaczenia

Poprawny model mentalny zmiennej w Python nie jest pudełkiem danych, lecz nazwą wskazującą na wartość.

Już stworzyliśmy imiona takie jak `name` w poprzednich sekcjach — Czas wyjaśnić, co to naprawdę oznacza.

Nie „pudełko z danymi”

Powszechna, choć nie do końca trafna metafora: „zmienna to pudełko, do którego wkłada się znaczenia.”



Nazwa wskazuje na znaczenie — a nie „zawiera”

Nazwa jest raczej strzałką wskazującą na wartość. Przypisanie `city = "Warsaw"` nie „wkłada” `city` — zmusza ono nazwę `city` wskazują na cechę łańcuch znaków `"Warsaw"` istnieje samodzielnie.

Tworzenie i czytanie imienia

names.py

```
city = "Warsaw"
print(city)
```

Części tej linii:

- `city` — nazwa (zmienna);
- `=` jest operatorem przypisania: „niech nazwa po lewej wskaże czyli po prawej stronie”
- `"Warsaw"` — wartość (obiekt).

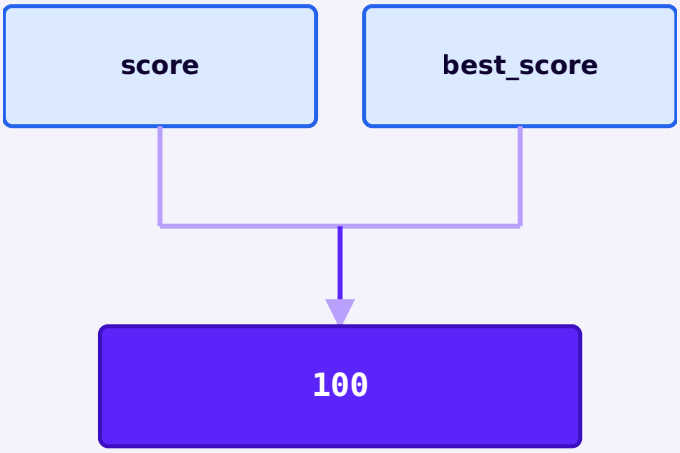
Wiele nazw — jeden obiekt

Ponieważ nazwa jest tylko wskaźnikiem, ten sam obiekt może mieć wiele imion jednocześnie. Weźmy przykład:

score.py

```
score = 100
best_score = score
print(score)
print(best_score)
```

Tutaj **dwa imiona** — `score` oraz `best_score` — ale **jeden obiekt**, na który oboje wskazują:



Dwa odwołania do tego samego obiektu — a nie dwie osobne liczby 100

Częsty błąd myślowy
NIE MYŚL, że Python „skopiował 100 na dwie zmienne pudełkowe” **link** (reference): "nazwa jest związana z obiektem" zamiast "nazwa zawiera znaczenie".

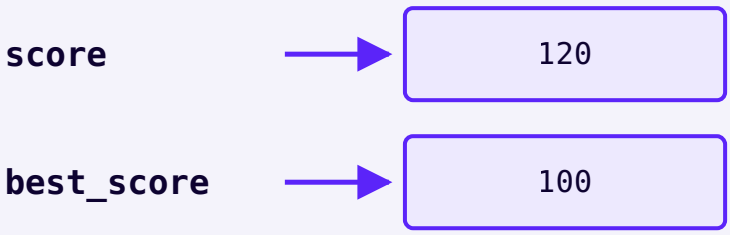
Przeniesienie nie zmienia starej nazwy

Kontynuujmy ten sam przykład:

score2.py

```
score = 100
best_score = score
score = 120
print(score)       # 120
print(best_score)  # wciąż 100
```

String `score = 120` jest **ponowne przypisanie** (rebinding): nazwa `score` zaczyna wskazywać na inny obiekt. Istniejący obiekt `100` jednocześnie nie zmienia się w żaden sposób — `best_score` nadal na to wskazuje:



Po score = 120: Każda nazwa ma teraz własne, niezależne powiązanie

To ważna idea, do której będziemy wracać więcej niż raz, gdy mówimy o niezmienności liczb oraz strings (Rozdział 4), o listach i ich różnicach względem liczb, o tym, jak jeden i ten sam obiekt może być być dostępnym z wielu lokalizacji oraz w jaki sposób wartości są przekazywane do funkcji. Na razie Wystarczy pamiętać o samej zasadzie: **zmiana zmienia relację z nazwą, nie obiekt.**

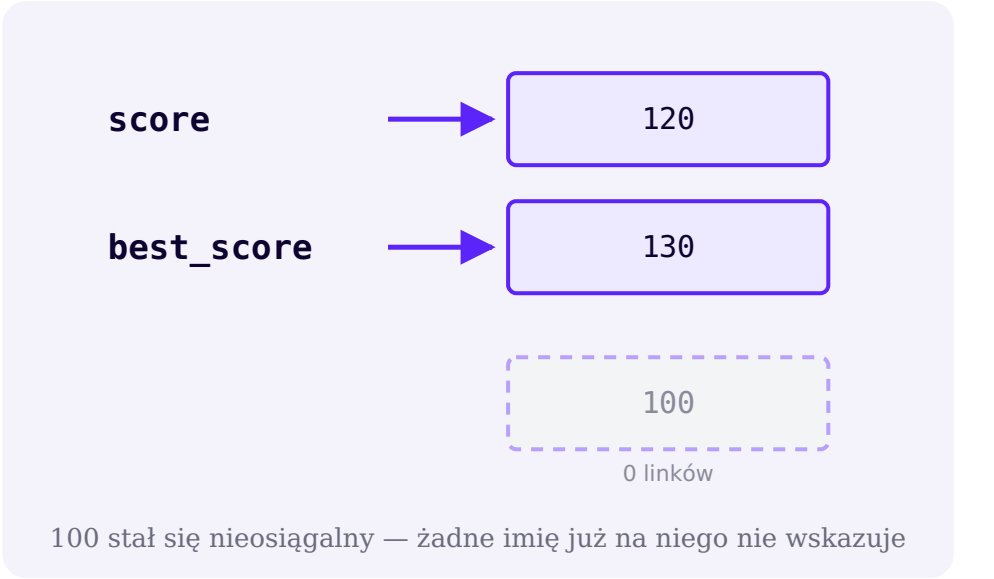
Kiedy obiekt „znika”

Logiczne pytanie: jeśli `score` już nie wskazuje na `100`, co się dzieje z samym obiektem `100`? Kontynuujmy ten przykład o krok dalej:

score3.py

```
best_score = 130
print(score)       # 120
print(best_score)  # 130
```

Teraz nie ma `score`, ani `best_score` nie wskazywanie obiektu `100` — nie ma już żadnego połączenia:



100 stał się nieosiągalny — żadne imię już na niego nie wskazuje

Obiekt `100` stał się **nieosiągalne** (unreachable) — nie można do niego „dotrzeć”

To nieścisłe tak pisząc
„Gdy tylko zmienna przestaje wskazywać na wartość, garbage collector natychmiast usuwa tę wartość” `best_score` wskazywał na `100` nawet po `score` przypisany). Obiekt staje się kandydatem do usunięcia tylko wtedy, gdy **żadnej** nie utrzymuje z nim kontaktu.

Garbage collector: automatyczne zarządzanie pamięcią

Zazwyczaj nigdy nie będziesz musiał ręcznie „zwalniać” **odśmiecacz (garbage collector)** (garbage collector, GC) jest częścią samego Python, co automatycznie uwalnia pamięć od obiektów nieosiągalnych.

Wystarczy do codziennej pracy
Python śledzi obiekty i zwalnia pamięć, gdy obiekt nie może być już używany pod żadną nazwą w programie. Dlatego w Python (w przeciwieństwie do języków takich jak C) prawie nigdy nie piszesz kodu specjalnie po to, by zwolnić pamięć.

Co idzie głębiej:

Liczenie linków

To opcjonalne, głębsze spojrzenie – możesz bezpiecznie pominąć go za pierwszym razem czytam i wracam później.

W CPython (implementacji Python, której używamy) każdy obiekt ma licznik: ile odniesień do niego istnieje teraz. Przejdźmy jeszcze raz przez nasz przykład, ale tym Kontrowersja:

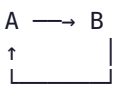
Obiekt 100 — licznik referencji

```
score = 100          # объект 100: ссылок = 1
best_score = score   # объект 100: ссылок = 2
score = 120          # объект 100: ссылок = 1  (score теперь
указывает на 120)
best_score = 130     # объект 100: ссылок = 0  (недостижим)
```

Dla zwykłych obiektów, nie wchodzących w cykliczne odniesienie w CPython moment, gdy licznik osiąga zero, zazwyczaj pokrywa się z momentem zwalniania pamięci — prawie od razu, bez opóźnienia. To wygodne, praktyczne zachowanie, ale nie gwarancja ustalona przez język Python jako całość — inne implementacje Python (na przykład PyPy)) zwalniają pamięć według innych zasad.

Liczenie ogniw ma jedną fundamentalną trudność — **cykliczne linki**:

Połączenie Kołowe (koncepcyjnie)



Wyobraź sobie dwa obiekty, każdy odnoszący się do drugiego. Nawet jeśli nie ma nazwy w Program już nie wskazuje na A ani B, nadal odnoszą się do siebie nawzajem — oraz liczba odniesień każdego z nich

nie spada do zera. Dlatego w CPython, z wyjątkiem liczenia linki, jest osobna **okrągły kontener śmieci**, który okresowo szuka właśnie takich nieosiągalnych z zewnątrz cykli i je zwalnia.

> **Idźmy dalej, nie teraz**
> Celowo tu się zatrzymujemy. Do zarządzania pamięcią, czasu życia obiektu i modułu `gc` wrócimy do tego później w kursie, gdy pojawią się struktury danych, które naprawdę mogą tworzyć cykle (np. obiekty odwołujące się do siebie nawzajem). Na razie wystarczy wiedzieć, że taka sytuacja istnieje i Python ma mechanizm dla niej.

Mała notatka o del

Komenda `del` nie usuwa obiektu, lecz konkretną relację (Imię):

del_demo.py

```
x = 100
y = x
del x
print(y)  #100 — Obiekt wciąż zniknął
```

`del x` usuwa imię `x` przestrzeń nazw — ale NIE oznacza „usuń obiekt 100, bez względu na wszystko.” `y` nadal wskazuje na `100`, Obiekt pozostaje w pełni dostępny przez `y`.

Przestrzeń nazw

Podczas działania programu Python przechowuje lista wszystkie utworzone nazwy oraz co każda z nich z nich teraz oznacza — to się nazywa **przestrzeń nazw** (namespace). Powyższe diagramy pokazują fragment takiej przestrzeni nazw w różnych momentach Wdrażanie programu.

Odnosi się do nazwy, która jeszcze nie została stworzona

Jeśli nazwa jest wspomniana przed jej utworzeniem w ramach cesji, Python nie może Zgadnij, o co chodzi — i zgłasza błąd:

broken.py

```
print(favourite_city)
favourite_city = "Warsaw"
```

Błąd

```
NameError: name 'favourite_city' is not defined
```

Więcej informacji o błędach i sposobie czytania takich komunikatów można znaleźć w sekcji 3.13.

Praktyka: Imiona i znaczenia
Interaktywny laptop bezpośrednio w przeglądarce – przypisuj, czytaj, NameError

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/03-07/index.html)

Komentarze i czytelny kod

Drobne nawyki, które znacznie ułatwiają życie – dla twojego przyszłego ja i wszystkich, którzy będą czytać twój kod.

Komentarze

linijka zaczynająca się od `#` jest **komentarz**: Python całkowicie go ignoruje podczas uruchamiania. Komentarze nie istnieją dla komputera, ale Dla ludzi – w tym dla siebie za sześć miesięcy.

privet.py

```
# Poproś o imię, aby spersonalizować powitanie
name = input(„Jak masz na imię?”)
print(„Cześć,”, name)
```

Dobry komentarz wyjaśnia "dlaczego" zamiast "co"
Komentuj jak `# складываем a и b` powyżej linii `c = a + b` bezużyteczny — kod to i tak pokazuje. Przydatny komentarz wyjaśnia to, czego nie widać w samym kodzie: dlaczego wybrano właśnie takie rozwiązanie, jaka logika biznesowa za tym stoi, na co warto zwrócić uwagę w przyszłości.

To całkowicie normalne, że zostawiasz więcej komentarzy na etapie nauki niż byś zrobił Doświadczony programista w gotowym projekcie – pomaga ci opowiedzieć, co robi Co łańcuch znaków. Nie martw się, jeśli nadal chcesz komentować prawie wszystko.

Nazwy plików i zmiennych

Dobry styl w nazwach oszczędza czas — zarówno Twój, jak i wszystkich, którzy czytają kod po Tobie:

- zastosowanie **małych liter**: `privet.py`, a nie `Privet.PY`;
- zastosowanie **sensowne nazwy**: `calculator.py`, a nie `file1.py`;
- dla wielu słów użyj **podkreślenie**: `my_first_program.py` zamiast spacji czy skrzynek.
- unikać spacji w nazwach plików — są wygodne w graficznym interfejsie, ale niewygodne w terminalu, gdzie musisz uciec z każdego pola.

Zły	Dobrze
`New File FINAL 2!!.py`	`calculator.py`
`a.py`	`temperature_converter.py`
`MyFirstProgram.py`	`my_first_program.py`

PEP 8
Oficjalny przewodnik stylowy Python-code to PEP 8 (peps.python.org/pep-0008) — wspomnieliśmy o nim już w Rozdziale 1. Opisuje znacznie więcej niż tylko nazwy plików, ale na razie wystarczy wiedzieć: istnieje i jego rekomendacje są podzielane przez niemal całą społeczność Python.

IDLE – czym jest i kiedy jest przydatny

Python oferuje prosty edytor — IDLE. Sprawdźmy, jak go używać i kiedy przejść na potężniejsze narzędzie.

Python posiada wbudowany prosty edytor kodu, który instaluje się automatycznie razem z Python samym, **IDLE** (Integrated Development and Learning Environment). Możesz go otworzyć, szukając „IDLE” `idle3` w terminalu (Linux).

Dlaczego w ogóle IDLE istnieje
IDLE nie jest „przestarzałym śmieciem”

Python Shell jest interaktywnym oknem powłoki

Gdy IDLE się zaczyna, interaktywna powłoka się otwiera, tak samo Python REPL my analizowane w sekcjach 3.5–3.6, tylko w osobnym oknie z niewielkimi udogodnieniami takimi jak Podkreślanie składni.

Tryb Scenariusza (Script Mode)

Aby napisać pełnoprawny program, potrzebny jest oddzielny plik: **File → New File** otwiera puste okno edytora — to jest „tryb skryptowy”

1. Wpisz kod w nowym oknie edytora.
2. Zapisz plik: **File → Save** (rozszerzenie `.py` samo się podstawia).
3. Bieg: **Run → Run Module**, albo po prostu naciśnij `F5`.
4. Wynik pojawi się w interaktywnym oknie powłoki IDLE.

Kiedy używać IDLE

IDLE → nowoczesny rozwój

IDLE (wchodzi w skład Python „z pudełka”)

```
print(„Cześć z IDLE!”)
#start: menu Run -> Run Module (F5)
# jeden plik, minimum ustawień
```

VS Code / PyCharm

```
print(„Pozdrowienia od VS Code!”)
# Start: Run przycisk lub wbudowany terminal
# podświetlanie błędu na bieżąco, automatyczne uzupełnianie,
#debugger, git, rozszerzenie zbiór
```

Co używać dzisiaj: IDLE świetnie jest napisać i uruchomić pierwszą linię kodu w zaledwie minutę po instalacji Python — nie musisz instalować niczego dodatkowego. Ale gdy projekty wykraczają poza granice pojedynczego pliku, zauważalny jest brak automatycznego uzupełniania, debugera i podświetlania błędów na bieżąco. Najczęściej używamy VS Code lub PyCharm — ale warto IDLE wiedzieć: to właśnie zobaczysz, jeśli spróbujesz Python na czyimś komputerze bez dodatkowej konfiguracji.

Praktyka: Zdobądź swoje imię (i coś jeszcze)

Łączymy w całość plik, terminal, print i wybór edytora — w jednym małym ćwiczeniu, zanim pójdziemy dalej.

Mały punkt kontrolny: użyjmy pliku, REPL, `print()` i wybór redaktora (w tym IDLE) w jednym ćwiczeniu, zanim przejdziemy dalej do błędów, Debugowanie i pierwszy prawdziwy mini-projekt.

Podstawowa praktyka
Rozgłoszić swoje nazwisko
Utwórz plik `o_sebe.py` i jeden zespół `print()` napisz swoje imię.

★★ Zadanie samodzielne
I coś jeszcze
W tym samym pliku dodaj jeszcze dwie linie: jedną z twoim miastem, drugą z dowolną liczbą, używając `sep` wypisywać nazwę, miasto i numer w jednej linii za pomocą „· ”

Challenge
Ten sam efekt na trzy sposoby
Uzyskaj ten sam wynik na trzy różne sposoby: uruchamiając plik z terminala (`python o_sebe.py`), przez przycisk Run w VS Code/ PyCharm oraz przez Run Module w IDLE.

Błędy, traceback i jak je czytać

Błędy to dane, a nie powód do paniki. Przeanalizujmy trzy najczęstsze typy i nauczmy się czytać traceback sensownie.

Czerwony tekst błędu nie jest powodem do paniki. Jest **dane**: Python szczegółowo wyjaśnia, co dokładnie poszło nie tak i gdzie dokładnie szukać problemu. Ucz się spokojnie Czytanie tych wpisów to jedna z najbardziej przydatnych umiejętności dla początkujących.

SyntaxError

Python nawet nie rozumie struktury kodu — w ogóle nie został wykonany. Częściej Jedynym powodem jest brak nawiasu, cudzysłowu lub dwukropku.

broken.py

```
print("Привет"
```

Traceback

```
  File "broken.py", line 1
    print("Привет"
                  ^
SyntaxError: '(' was never closed
```

NameError

Python napotkał nazwę, która jeszcze nigdzie nie została utworzona przez przypisanie — często jest to po prostu literówka.

broken2.py

```
print(usr_name)
```

Traceback

```
Traceback (most recent call last):
  File "broken2.py", line 1, in <module>
    print(usr_name)
          ^^^^^^^^
NameError: name 'usr_name' is not defined
```

IndentationError

W Python wcięcie jest częścią składni (wrócimy do tego szczegółowo w rozdziale o warunkach i pętle). Niespójne wcięcia to także błąd:

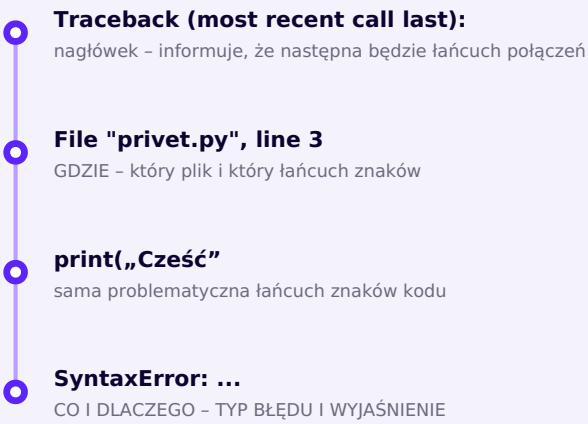
broken3.py

```
name = "Cartesian"
    print(name)
```

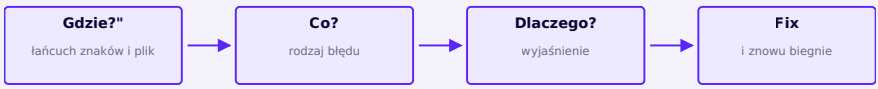
```
Traceback
  File "broken3.py", line 2
    print(name)
IndentationError: unexpected indent
```

Anatomia traceback

Traceback najlepsza lektura **od dołu do góry** najważniejsze informacje poniżej:



Anatomia traceback – czytanie od dołu do góry



Kolejność postępowania z każdym błędem

Nie bój się czerwonego tekstu

Błąd — to Python, uczciwie wyjaśniający, co się stało. Profesjonalni programiści widzą traceback- dziesiątki razy dziennie — różnica polega tylko na tym, że potrafią szybko znaleźć w nim potrzebną linię. Teraz i ty potrafisz.

Praktyka: Przeczytaj traceback (NameError)

Interaktywny laptop bezpośrednio w przeglądarce – znajdź i poprawij literówkę

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/03-08/index.html)

Laboratoria debugowania

Pięć krótkich, specjalnie popsutych przykładów — poćwiczymy tę samą umiejętność: przewidzieć, uruchomić, zbadać, naprawić.

Pięć krótkich laboratoriów – Złamany kod do ćwiczenia znajdowania i Rozwiąż problem według tego samego algorytmu: przewiduj → przejdź → badaj → napraw → wyjaśnić sobie, dlaczego to zadziałało.

Laboratorium 1.

Brakujący cudzysłów lub nawias

1. Przewidywanie: przeczytaj poniższy kod – co myślisz, że się wydarzy?

```
zepsuty kod
```

```
print("Привет, мир!
```

2. Uciekaj — zobaczycie: `SyntaxError: unterminated string literal`

3. Poprawka:

```
naprawione
```

```
print(„Witaj, świecie!”)
```

4. Dlaczego: cudzysłowy otwierające nie miały pary, Python nie rozumiały końca tekstu.

Laboratorium 2.

Nieprawidłowa nazwa zmiennej

1. Przewidywanie: przeczytaj poniższy kod – co myślisz, że się wydarzy?

zepsuty kod

```
user_name = "Cartesian"
print(usr_name)
```

2. Uciekaj — zobaczycie: `NameError: name 'usr_name' is not defined`

3. Poprawka:

naprawione

```
user_name = "Cartesian"
print(user_name)
```

4. Dlaczego: literówka w nazwie — `usr_name` zamiast `user_name`. Python nie „zgaduje”

Laboratorium 3.

Błąd wcięcia

1. Przewidywanie: przeczytaj poniższy kod – co myślisz, że się wydarzy?

zepsuty kod

```
print(„Początek”)
  print(„Kontynuacja”)
```

2. Uciekaj — zobaczycie: `IndentationError: unexpected indent`

3. Poprawka:

naprawione

```
print(„Początek”)
print(„Kontynuacja”)
```

4. Dlaczego: druga linijka ma wcięcie, którego nie powinno tam być — poza blokami (if/for/def pojawią się w kolejnych rozdziałach), wszystkie linie powinny zaczynać się od tej samej pozycji.

Laboratorium 4.

Program działa, ale generuje niewłaściwe wyniki

1. Przewidywanie: przeczytaj poniższy kod – co myślisz, że się wydarzy?

zepsuty kod

```
name = „Anna”
city = „Moskwa”
print(„Cześć,”, city)
```

2. Uciekaj — zobaczycie:
`Привет, Москва (а должно быть — имя)`

3. Poprawka:

naprawione

```
name = „Anna”
city = „Moskwa”
print(„Cześć,”, name)
```

4. Dlaczego: program wykonał się bez żadnego błędu, ale zmienne były pomieszane. To najbardziej podstępny typ błędu: Python nie mogą wiedzieć, że nie miałeś na myśli tego, co napisałeś.

Laboratorium 5. Zły tłumacz – Pozdrowienia z rozdziału 2

1. Przewidywanie: przeczytaj poniższy kod – co myślisz, że się wydarzy?

```
zepsuty kod
```

```
import requests
print(„Pakiet znaleziony!”)
```

2. Uciekaj — zobaczycie: `ModuleNotFoundError: No module named 'requests' (хотя вы точно его ставили!)`

3. Poprawka:

```
naprawione
```

```
# w VS Code: Ctrl+Shift+P -> „Python: Select Interpreter”
# Wybierz interpretera z aktywnego projektu . venv
```

4. Dlaczego: pakiet zainstalowany w jednym środowisku, a kod uruchamiany przez inne — klasyczny problem z rozdziału 2 (sekcja 2.16). Sprawdź `sys.executable` oraz wybranego interpretera w edytorze.

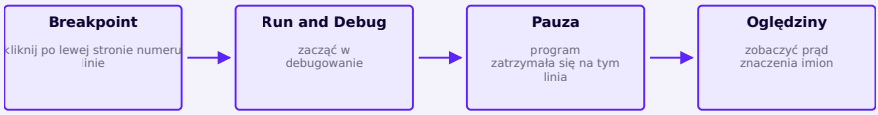
Praktyka: Przeczytaj traceback (SyntaxError)
Interaktywny laptop bezpośrednio w przeglądarce – znajdź brakujące nawiasy i cudzysłowy

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/03-09/index.html)

Pierwszy debugger w IDE

Debugger może zrobić to, czego nie print() — zatrzymać program i zajrzeć do środka.

Debugger (debugger) to narzędzie, które może **wstrzymywać** program wykonywany na konkretnej linii i pokaż, z czym wszyscy są w danym momencie równi nazwy (Rozdział 3.9). To znacznie potężniejsze niż aranżowanie `print()` w całym kodzie i uruchamiać ponownie.



Ogólna idea debuggera jest taka sama w VS Code jak w PyCharm

Weźmy ten sam malutki przykład dla obu środowisk:

debug_demo.py

```
name = „Anna”
city = „Warszawa”
greeting = „Cześć, ” + name
print(greeting)
```

Debugger w VS Code

1. Kliknij po lewej stronie od numeru linii `greeting = "Привет, " + name` - Pojawia się czerwona kropka (breakpoint, punkt przerwania).
2. Otwórz kartę **Run and Debug** na bocznym panelu (lub naciśnij `F5`).
3. Program zaczyna i kończy się dokładnie w punkcie przerwy, łańcuch znaków się zapala.
4. W panelu **Variables** po lewej stronie widać aktualne wartości: `name` już istnieje, ale `greeting` — jeszcze nie (zatrzymaliśmy się PRZED wykonaniem tej linijki).
5. **Step Over** (ikona strzałki nad kropką) wykonuje dokładnie jeden linia i znowu stopy — teraz `greeting` też pojawi się na liście zmiennych.
6. **Continue** (▷) uwalnia program do końca lub do następnego punktu wyłączenie.

Debugger w PyCharm

Idea jest ta sama – różni się tylko nazwy przycisków:

Akcja	VS Code	PyCharm
Postawić punkt zostańova	Kliknij po lewej stronie numeru linii	Kliknij po lewej stronie numeru linii
Uruchom z debugowaniem	Run and Debug / F5	ikona chrząszcza obok Run
Wykonaj jeden wiersz	Step Over	Step Over (F8)
Przejdź do końca/Następny punkt	Continue	Resume Program
Zobacz wartości zmiennych	Panel Variables	Variables panel (na dole)

O zrzutach ekranu na tej stronie

Ta strona opisuje rzeczywisty, standardowy interfejs debugera VS Code oraz PyCharm tekst i wykres, a nie zrzut ekranu — środowisko, w którym powstał ten kurs, nie pozwala robić zrzutów ekranu aplikacji graficznych. Sekwencja działań (punkt przerwania → uruchomienie z debugowaniem → zatrzymaniem → sprawdzeniem wartości → Step Over/Continue) jest dokładnie taka sama jak to, co widzisz na ekranie.

Główna idea

Debugger pozwala PAUSE działającego programu i zajrzeć do jego wnętrza — zobaczyć prawdziwe znaczenia nazw w momencie uruchomienia, zamiast zgadywać z kodu, co miało się wydarzyć.

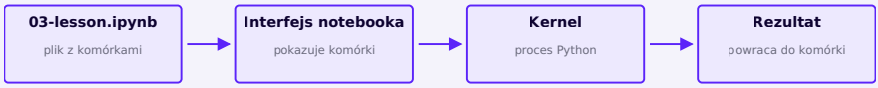
Notebook i kernel

Co tak naprawdę dzieje się, gdy uruchamiasz komórkę – zarówno w Notatniku, jak i w przeglądarce tego kursu.

Już ćwiczyliśmy ten kurs w przeglądarce, więc czas wyjaśnić, co znajduje się na faktycznie się to dzieje, z punktu widzenia wykonywania kodu.

Notebook to nie to samo co kernel

- **notebook** (`.ipynb`) to sam plik: zbiór komórek z kodem, tekstem i już zapisanymi wynikami poprzedniego przebiegu;
- **kernel** (jądro) to osobny proces roboczy, Python który faktycznie jest Wykonuje kod z komórek, gdy je uruchamiasz.



Od pliku notatnika do wyniku – przez jądro

Ten sam plik notatnika można otworzyć i dołączyć do różnych jąder, takich jak jądro z jednym zestawem zainstalowanych pakietów lub z innym (rozmawialiśmy o tym w rozdziale 2, sekcja 2.15, w odniesieniu do VS Code).

Ważna cecha: stan jest zapisywany między komórkami

W przeciwieństwie do zwykłego skryptu, który wykonuje się za każdym razem **od początku**, W Notatniku zmienne utworzone w jednej komórce pozostają dostępne w kolejnych — podczas gdy Ten sam kernel działa.

```
Komórka 1
```

```
x = 10
```

```
Komórka 2 (uruchomiona PO komórce 1)
```

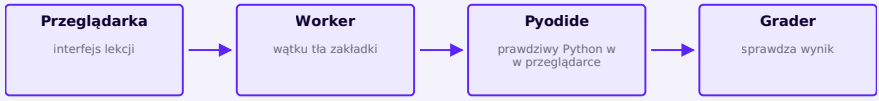
```
print(x)  #10 — x wciąż istnieje
```

Kolejność wystrzeliwania komórek nie jest kolejnością komórek na ekranie

Jeśli uruchomić komórki nie w kolejności (na przykład drugą przed pierwszą) lub ponownie uruchomić jądro, nie wykonując wszystkiego od nowa — wynik może nie zgadzać się z tym, co widać na ekranie. To jedna z najczęstszych przyczyn zamieszania u początkujących w notatnikach: to, co pokazane na ekranie, jest wynikiem OSTATNIEGO uruchomienia, a nie gwarancją tego, co powstanie przy wykonywaniu od góry do dołu teraz.

Jak działa praktyka przeglądania tego kursu

Wszystkie interaktywne ćwiczenia na kursie są prawdziwym Python, a nie imitacją. Działa to tak:



Jak działa praktyka przeglądania tego kursu

Pyodide jest prawdziwym interpreterem Python skompilowanym w WebAssembly i działając bezpośrednio w Twojej przeglądarce, bez instalowania na komputerze. To jest kolejna Runtime — z własnymi ograniczeniami: na przykład nie ma dostępu do natywnego dostępu graficzne okna systemu operacyjnego, więc pakiety takie jak `tkinter` nie pracują tam w czystej formie (na kursie takie ćwiczenia są szczerze oznaczane jako „lokalne”

Oficjalne źródło

Więcej o samym Jupyter — jupyter.org; o Pyodide — pyodide.org.

Mini-projekt i streszczenie rozdziałów

Zebranie wszystkiego, co omówiliśmy, w jeden mały, ale realny projekt — i podsumowanie rozdziału.

Czas zebrać niemal wszystko, co omówiliśmy w tym rozdziale, w jeden mały, ale prawdziwy rozdział Projekt.

Mini-projekt: wizytówka w terminalu

Program prosi o nazwę, miasto i to, co chcesz stworzyć na Python — a następnie wydrukuje Mały, spersonalizowany profil.

card.py

```
name = input(„Twoje imię: ”)
city = input(„Twoje miasto: ”)
goal = input(„Co chcesz stworzyć na Python?”)

print("========================")
print(„MY PYTHON-CARD”)
print("========================")
print(„Imię:”, name)
print(„City:”, city)
print(„Chcę tworzyć:”, goal)
print("========================")
```

Przykład tego, co powinno powstać:

Przykład wyjścia

```
========================
 МОЯ PYTHON-КАРТОЧКА
========================
Имя: Анна
Город: Варшава
Хочу создать: игру
========================
```

Tylko to, co już znajome

Cały projekt opiera się wyłącznie na narzędziach z tego rozdziału: `input()`, przywłaszczenie, `print()`. Warunki i pętle, które uczyniłyby kod bardziej elastycznym, pojawią się w kolejnych rozdziałach — na razie to wystarczy, by stworzyć prawdziwy, kompletny program.

Challenge

Challenge: własny projekt

Wymyśl własny projekt wizytówki — inną ramkę (na przykład z `*`), wyrównanie przez `sep` lub dodaj czwartą linię z ulubionym językiem programowania.

Praktyka: mini-projekt „Python-wizytówka”
Interaktywny notebook bezpośrednio w przeglądarce — zbierz wszystko razem

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/03-10/index.html)

Podsumowanie rozdziału

Co możemy teraz zrobić

- Zrozum, że Python program to zwykły tekst. py- plik i jak CPython przekształca kod źródłowy w kod bajtowy i następnie go wykonuje.
- Wyraźnie rozróżnić terminal, system operacyjny shell i Python REPL — trzy różne rzeczy o podobnych nazwach.
- Znamy podstawowe powłoki (Bash, Zsh, PowerShell, cmd.exe, Fish) zapoznaliśmy się z PySH — Python-first alternatywą na lokalnym przykładzie.
- Pewnie korzystamy z Python REPL: R-E-P-L, zaproszeń >>> i ..., type()/help()/dir() jako narzędzi badawczych.
- Rozróżnić script, REPL, notebook i IDE i wiedzieć, do czego każdy z nich najlepiej się nadaje.
- Opanowałem print() szczegółowo (kilka znaczeń, sep, end) i input() (pierwszy dialog, tekst w jej imieniu).
- Rozumiej nazwy jako wskaźniki znaczeń, a nie jako „pudełka danych”
- Umieć pisać zrozumiałe komentarze i wybierać czytelne nazwy plików i zmiennych.
- czytać SyntaxError spokojnie, NameError i IndentationError, rozumieć anatomię traceback i przeszedłem przez pięć laboratoriów debugowania.
- Wiedzieć, jak ustawić punkt przerwania i używać debuggera w VS Code i PyCharm.
- Zrozum różnicę między notebook a kernel — oraz jak działa praktyka przeglądania tego kursu na Pyodide.
- Zorganizowaliśmy pierwszy prawdziwy mini-projekt — interaktywną wizytówkę.

co dalej

W rozdziale 4 przyjrzymy się bliżej liczbom — ich formom, przekształceniom i matematyce na Python.

ROZDZIAŁ 4 · STR. 119

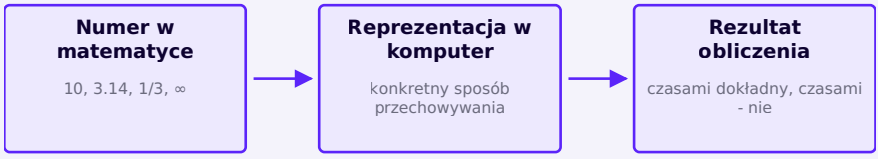
Python lubi liczby

Pełne bazy liczbowe Python 3.14: co komputer nazywa liczbą, systemy int i liczbowe, operatorzy i ich kolejność, dzielenie i reszta, float i dlaczego 0.1 + 0.2 nie jest równe 0.3, zaokrąglenie, Decimal, Fraction, complex, math, random/secrets, statistics — oraz kilka mini-projektów.

Czym jest liczba dla komputera

Zanim przejdziemy do typów liczb, zastanówmy się, co oznacza „liczba”

Człowiek widzi liczby jako pojęcia abstrakcyjne: `10`, `3.14`, `1/3`, nawet `∞`. Komputer natomiast nie „rozumie” **Wydajność**: Specyficzna sekwencja bajtów przez Konkretne zasady.



Liczba matematyczna i jej reprezentacja w komputerze — są powiązane, ale nie zawsze się pokrywają

Dla liczb całkowitych to rozróżnienie jest niemal niezauważalne— reprezentacja komputerowa zachowuje się Tak jak liczba w matematyce. Ale dla liczb ułamkowych różnica staje się istotna — i my

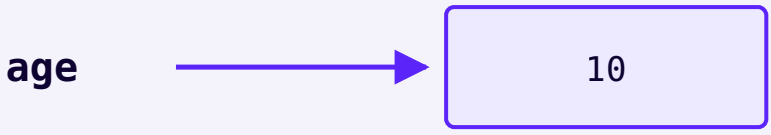
również Przeanalizujemy ją szczegółowo w sekcjach 4.10–4.11, gdy przejdziemy do `float`. Na razie wystarczy pamiętać o samej zasadzie: co piszesz w kodzie i co faktycznie jest przechowywane wewnątrz znajdują się powiązane, ale niekoniecznie identyczne rzeczy.

Nazwy wskazują na cechy liczbowe

W rozdziale 3 omówiliśmy ważną zasadę: nazwa (zmienna) nie jest „pudełkiem”

age.py

```
age = 10
print(age)
```



age wskazuje obiekt liczbą 10 — nie „zawiera” go

Co się dzieje, gdy age = age + 1

Liczby w Python — **niezmienne** (immutable) obiekty: sam obiekt `10` nie można „edytować” w 11. Kiedy piszemy:

age2.py

```
age = 10
age = age + 1
print(age)  # 11
```

nie jest edycją, lecz ponownym przypisaniem, Python oblicza nową wartość i Zmusza nazwę do wydania `age` wskazać na NOWY obiekt:

PRZED: age oznacza 10

`obliczenie: 10 + 1 → 11`

PO: age wskazuje na nowy obiekt 11 – stary obiekt 10 nie jest już potrzebny

Precyzyjne sformułowanie jest ważne

Python nie „zmienił liczby 10 na 11” **niezmienne**: Python oblicza wynik `10 + 1` otrzymał nowy przedmiot `11` i przydzielono nazwę na nowo `age` do tej nowej placówki. To dokładnie ten sam model "nazw → obiekt" i "przypisanie zmienia relację, a nie sam obiekt", o którym rozmawialiśmy w rozdziale 3.

Będziemy wracać do zasady niezmienności przez cały kurs – to wyjaśnia zachowanie łańcuch znaków, krotk oraz (w przeciwieństwie do tego) dlaczego listy się zachowują w przeciwnym razie, gdy do nich dochodzimy w kolejnych rozdziałach.

Spróbujmy

Wpisz ten przykład w przeglądarce lub w REPL (Rozdział 3) — i sprawdź po każdej linii `id(age)` . Liczba (adres obiektu w pamięci dla CPython) zmieni się po ponownym przypisaniu jest wyraźnym potwierdzeniem, że obiekt faktycznie się zmienił.

Praktyka: Liczby jako obiekty, nie pudełka
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/04-01/index.html)

int — liczby całkowite bez obaw o duże wartości

Liczby całkowite w Python mogą rosnąć znacznie bardziej niż w większości innych języków — i jednocześnie pozostają obiektami niezmiennymi.

`int` (integer jest liczbą całkowitą) jest typem liczb bez ułamków części: dodatnie, ujemne i zero.

celye.py

```
print(0)
print(42)
print(-17)
print(type(42))
```

Liczby z niemal zerowym limitem wielkości

W przeciwieństwie do wielu innych języków programowania, gdzie liczby całkowite są ograniczone stały zakres (np. 32 lub 64 bity), w Python `int` może rosnąć tak bardzo – dokładność **dowolny**, ograniczone w praktyce tylko dostępna pamięć komputera, nie język.

bolshoe_chislo.py

```
huge = 10 ** 100
print(huge)
```

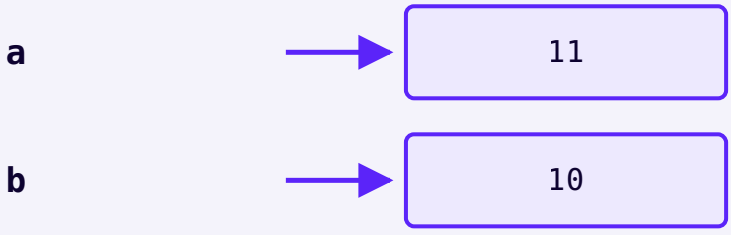
Dokładne sformułowanie
Poprawnie mówi się „dowolna precyzja ograniczona dostępnymi zasobami”, a nie „nieograniczona wielkość”. Formalnie istnieje limit — ale w praktyce prawie nigdy go nie osiągniesz w zwykłych programach.

liczby są niezmienne – ciąg dalszy

Widzieliśmy to już w sekcji 4.1 na przykładzie `age`. Znowu, już z dwoma nazwami:

a_b.py

```
a = 10
b = a
a += 1
print(a)  # 11
print(b)  # 10 — b się nie zmienił
```



Po a += 1: a ma nowe połączenie; b nadal wskazuje na stary obiekt

Nie używaj id() jako triku naukowego do „kasha liczb całkowitych”
Możesz gdzieś natknąć się na porównanie `id(256)` oraz `id(257)` jako „dowód”

bool jest szczególnym krewnym int

Wartości `True` oraz `False` forma Typ `bool` — i historycznie jest ściśle związany z `int`: `True` zachowuje się jak `1`, a `False` — jak `0`.

bool_int.py

```
print(int(True))   # 1
print(int(False))  # 0
print(True + True) # 2
```

Ale w normalnym kodzie — to wartości logiczne
To, co `bool` technicznie „liczbowy” — element języka, a nie wezwanie do dodawania `True + True` w rzeczywistym kodzie. Więcej informacji o warunkach i logice można znaleźć w rozdziale o warunkach; tutaj wystarczy znać związek z liczbami, jeśli gdzieś go znajdziesz.

Podkreślenia separatorów dla czytelności

W długich liczbach łatwo stracić rachubę zer. Python pozwala na podzielenie liczby Podkreślenia — nie wpływają one na znaczenie, a jedynie na czytelność:

naselenie.py

```
population = 38_000_000
print(population)  # 38000000
```

Skrócone przydziały

Operatorzy tacy jak `+=` jest zwartym zapisem "liczenia i przydzielić" zamiast "zmienić numer na miejscu":

augmented.py

```
x = 10
x += 1   # to samo co x = x + 1
x -= 2   # tak samo jak x = x - 2
x *= 3   # tak samo jak x = x * 3
```

Praktyka: duże liczby i niezmienność int
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/04-06/index.html)

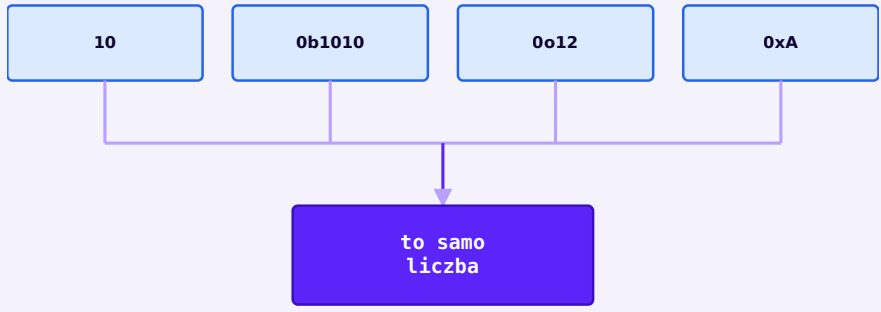
Jak zapisywać liczby: systemy liczbowe

System dziesiętny nie jest jedynym. Przyjrzyjmy się notacjom binarnym, ósemkowym i szesnastkowym oraz dlaczego programista w ogóle ich potrzebuje.

Jesteśmy przyzwyczajeni do zapisywania liczb w **dziesiętnym** system (base 10) — dziesięć cyfry, od 0 do 9. Ale to nie jedyny sposób, a komputery często korzystają z innych Systemy liczbowe.

Cztery systemy, które napotkasz

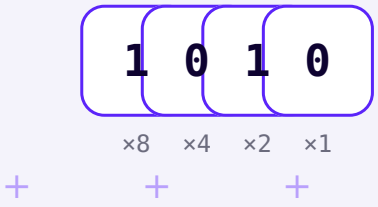
System	Fundacja	Liczby	Python-literal
Dziesiętna (decimal)	10	0–9	`10`
Binarność (binary)	2	0–1	`0b1010`
Oktalny (octal)	8	0–7	`0o12`
Szesnastkowy (hexadecimal)	16	0–9, A–F	`0xA`



Cztery zgłoszenia, ten sam wynik

Jak czytać notację pozycyjną

W każdym systemie liczbowym każda pozycja cyfry ma swoją „wagę” — potęgę podstawy. Weźmy liczbę binarną `1010`:



$1010_2 = 1\times8 + 0\times4 + 1\times2 + 0\times1 = 10$

Dokładnie ta sama zasada działa również dla systemu dziesiętnego, którego używasz codziennie — po prostu wagi pozycji tam są 1000, 100, 10, 1 (moc dziesiątki), a nie potęgi dwójki.

Dlaczego programiście potrzebny jest system szesnastkowy

Zapis szesnastkowy spotyka się w praktyce znacznie częściej, niż się wydaje:

- **Kolory** w internecie i grafice: `#FF0000` — jaskrawoczerwony;
- **Wzorce bajt/bit** — kompaktowe zapisywanie danych binarnych;
- **Protokoły sieciowe i formaty plików** — MAC- adresy, sumy skrótu;
- **Symbole Unicode** — kod symbolu często jest zapisywany w hex (patrząc w przyszłość, Więcej o Unicode w rozdziale o liniach).

cvet.py

```
red = 0xFF
print(red)  # 255
```

Funkcje bin(), oct(), hex()

Przekonwertować liczbę Z notacji dziesiętnej na łańcuch znaków o innym systemie liczbowym:

v_druguyu_sistemu.py

```
print(bin(10))  # '0b1010'
print(oct(10))  # '0o12'
print(hex(10))  # '0xa'
```

int() z oznaczeniem podstawy – transformacja odwrotna

Funkcja `int()` dwoma argumentami robi odwrotnie — odczytuje tekst w określonym układzie liczbowym i zwraca regularną (dziesiętną) liczbę:

iz_teksta.py

```
print(int("1010", 2))  # 10
print(int("FF", 16))   # 255
```

Spróbujmy

Przekonwertuj swój rok urodzenia na notację binarną i szesnastkową za pomocą `bin()` oraz `hex()` — a następnie przetłumaczyć wynik z powrotem `int(..., 2)` oraz `int(..., 16)` upewnić się, że dostaniesz ten sam numer.

Praktyka: tłumacz systemów liczbowych

Interaktywny laptop bezpośrednio w Twojej przeglądarce – bin(), oct(), hex(), int(x, base)

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/04-07/index.html)

Komentarze w obliczeniach: jednostki, wzory i zamiar

Same liczby nie mają sensu — komentarze i sensowne nazwy pomagają określić, co dokładnie oznaczają.

Już szczegółowo przeanalizowaliśmy komentarze w rozdziale 3 — tutaj nie będziemy powtarzać całej lekcji, ale Skupmy się na tym, dlaczego komentarze są szczególnie przydatne w obliczeniach: jednostki pomiary, wzory i ukryte założenia.

Liczba sama w sobie nie niesie jednostki miary

Spójrz na tę linię:

distance.py

```
distance = 120  # km
```

Komentarz tutaj pomaga — ale pozostaje kruchy: żyje oddzielnie od wartości i nikt nie sprawdza, czy jest nadal prawdziwy. Bardziej niezawodny (choć nie idealny) sposób — uwzględnić jednostkę bezpośrednio w nazwie:

distance2.py

```
distance_km = 120
```

Tak wyraźniej widać z samego kodu, co dokładnie jest przechowywane — bez potrzeby szukania komentarza obok. Wrócimy do bardziej rygorystycznych sposobów łączenia wartości z ich znaczeniem (typy, struktury danych) w następnych rozdziałach — tutaj wystarczy świadomie uświadomić sobie problem: **Python nie wiedzą, że liczba 120 oznacza kilometry**. Ta wiedza istnieje tylko w głowie tego, kto napisał kod — i co najwyżej w nazwie lub komentarzu.

Komentarze do wzorów i założeń

Oddzielna, bardzo przydatna rola komentarzy — wyjaśnić formułę lub zarejestrować przypuszczenie, które w innym przypadku pozostałoby „magiczną liczbą”:

nds.py

```
#VAT w przykładzie przykładowym: 23%
vat_rate = 0.23
price = 100
total = price * (1 + vat_rate)
print(total)
```

Dobry komentarz wyjaśnia, czego nie widzisz w kodzie
`vat_rate = 0.23` – sama liczba nie wyjaśnia, skąd pochodzi i co oznacza. Komentarz `# НДС в учебном примере: 23 %` odpowiada dokładnie na to pytanie – "dlaczego ta liczba", a nie "co robi ten łańcuch znaków".

Ten sam trik przyda się dosłownie w każdej sekcji tego rozdziału — formuły na powierzchnię koła, konwersję czasu, rabaty i procenty czyta się znacznie łatwiej, jeśli obok zaznaczono, co oznacza każda liczba.

Praktyka: Jednostki miary i komentarze do wzorów
Interaktywny laptop bezpośrednio w przeglądarce – wyraźnie wyczyść kod numeryczny
Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/04-02/index.html)

Operatory arytmetyczne

Pełny zestaw operatorów do pracy z liczbami, od dodawania do potęgowania.

Python obsługuje pełny zestaw operatorów arytmetycznych — większość z nich już jest Znam to z matematyki, ale kilka z nich ma szczególne, „programistyczne”

Operator	Nazwa	Przykład	Rezultat
`+`	Dodawanie	`7 + 3`	10
`-`	Odejmowanie	`7 - 3`	4
`*`	Mnożenie	`7 * 3`	21
`/`	Dywizja (zawsze float)	`7 / 3`	2.333...
`//`	Dzielenie całkowite	`7 // 3`	2
`%`	Pozostałość dywizji	`7 % 3`	1
`**`	Potęgowanie	`7 ** 3`	343

Unary + i -

Plus i minus również działają jako operatory unarne — przed jedną liczbą, bez drugiej Operand:

unarnye.py

```
x = 5
print(-x)   # -5
print(+x)   #5 (unarny + prawie nie używany, ale istnieje)
```

Operatorzy w związku z rzeczywistym zadaniem

Weźmy na zniżoną cenę produktu:

skidka.py

```
price = 1000
discount_percent = 15
discount_amount = price * discount_percent / 100
final_price = price - discount_amount
print(final_price)  # 850.0
```

Praktyka: Operatory arytmetyczne
Interaktywny laptop bezpośrednio w przeglądarce – przewiduj i sprawdź wynik

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/04-08/index.html)

Kolejność wykonywania operacji

2 + 3 * 4 czy 14 czy 20? Przyjrzyjmy się kolejności, w jakiej Python faktycznie ocenia wyrażenia.

Podobnie jak w matematyce, w Python operacje nie wykonuje się ściśle od lewej do prawej — dla wszystkich operator ma własny **priorytet**.

primer.py

```
print(2 + 3 * 4)     #14, nie 20
print((2 + 3) * 4)   # 20 — nawiasy zmieniają kolejność
```

`()`	nawiasy — zawsze wykonywane jako pierwsze
`**`	potęgowanie
`+x -x`	operator jednoargumentowy plus/minus
`* / // %`	mnożenie, dzielenie, dzielenie całkowite, reszta
`+ -`	dodatku, odejmowanie

Kolejność operacji w tym rozdziale to od najwyższego do najniższego priorytetu

To nie jest kompletna tabela

Oficjalna tabela priorytetów Python zawiera znacznie więcej operatorów — porównań, boolea, bit i tak dalej — do których jeszcze nie dotarliśmy. Oto co odnosi się do arytmetyki tego rozdziału. Pełną tabelę można znaleźć w sekcji „Operator precedence”

Zasada dobrych manier

Nawet gdy dokładnie pamiętasz priorytet operatorów, nawiasy prawie nic nie kosztują, a poprawiają czytelność znacząco. Jeśli masz wątpliwości, jak wyrażenie odczyta inna osoba (lub ty sam za pół roku) — dodaj nawiasy.

Praktyka: Ustalcie kolejność operacji

Interaktywny laptop bezpośrednio w przeglądarce – przewidź wynik, a następnie zablokuj

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/04-09/index.html)

Dzielenie: /, //, % i divmod()

Trzy powiązane operatory dzielenia — i dlaczego dzielenie ujemnych liczb całkowitych nie działa tak, jak się wydaje na pierwszy rzut oka.

W dzieleniu w Python faktycznie są trzy powiązane, ale różne operatory — i mylenie ich jest jednym z najczęstszych błędów u początkujących.

17 cukierków dla 5 dzieci

Wyobraźmy sobie zadanie: mamy 17 cukierków i dzielimy je równo między 5 dzieci.

konfety.py

```
candies = 17
children = 5

print(candies / children)   # 3.4 — dokładne dzielenie ułamkowe
print(candies // children)  #3 — 3 cukierki dla każdego dziecka
print(candies % children)   #2 — 2 cukierki pozostawione
nierozdanymi
```

Każde dziecko otrzymuje `candies // children` całe cukierki, i `candies % children` to, ile zostało „zostało”

divmod_demo.py

```
print(divmod(17, 5))  # (3, 2)
```

Trochę głębiej: Negatywny podział floor-

Z liczbami ujemnymi `//` zachowuje się inaczej niż „po prostu odrzuć część ułamkową” **na bok, minus nieskończoność** (floor):

otricatelnoe.py

```
print(-7 // 3)  #-3 zamiast -2
print(-7 % 3)   # 2
```

-7 // 3 zaokrągla w dół, do -3 (w stronę minus nieskończoności), a nie do -2

Dlaczego to, a nie inaczej

`//` w Python definiuje się jako „floor division”

Praktyka: dzielenie, reszta i divmod()

interaktywny laptop bezpośrednio w przeglądarce – w tym negatywny floor-division

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/04-10/index.html)

Stopnie, pow() i abs()

Wykładnictwo to także funkcja abs(), która przyda się w każdym rozdziale kursu.

Potęgowanie w Python - Operator `**` albo funkcja `pow()`:

stepeni.py

```
print(2 ** 10)     # 1024
print(pow(2, 10))  #1024 – To samo
```

Opcjonalny trzeci argument pow()

`pow()` może też użyć trzeciego, opcjonalnego argumentu — Moduł:

pow_mod.py

```
print(pow(2, 10, 1000))
#24 jest tym samym co (2 ** 10)% 1000, ale bardziej efektywny
dla większych liczb
```

Ta „modularna potęgowanie”

---

abs() — moduł liczby

`abs()` zwraca liczbę nieznaną — odległość do zera:

abs_demo.py

```
print(abs(-7))   # 7
print(abs(7))    # 7
print(abs(-3.5)) # 3.5
```

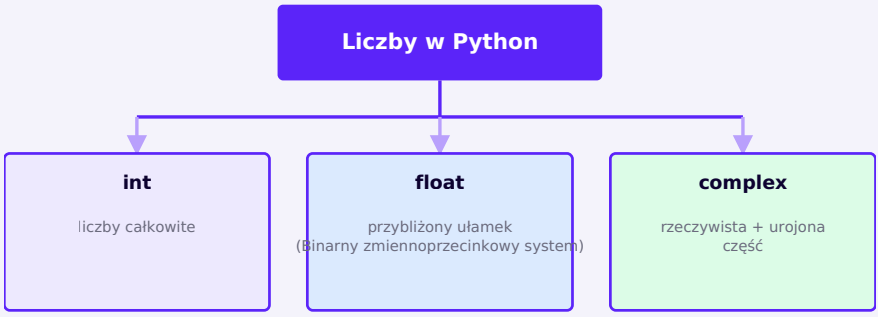
Praktyka: stopnie, pow() i abs()
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/04-11/index.html)

Mapa typów liczb Python

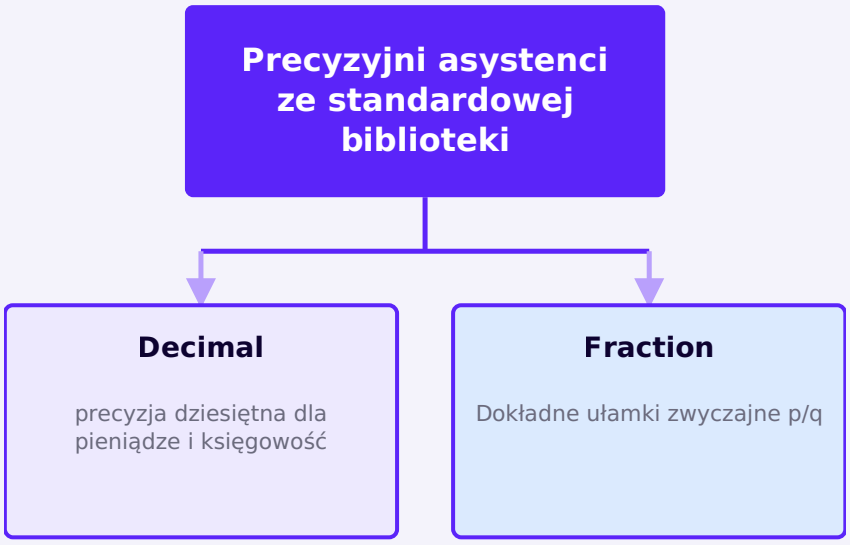
Zanim przejdziemy głębiej do każdego typu osobno, zobaczmy szerszy obraz.

Zanim przejdziemy głębiej — zbierzmy ogólną mapę. W Python są trzy **wbudowanych** typów liczbowych, których literały rozumie sam język:



Trzy wbudowane typy liczb – literali rozumieją sam język

Oprócz nich, standardowa biblioteka (moduły instalowane z Python, ale wymagać explicitnych `import`) istnieją jeszcze dwa ważniejsze numeryczne Narzędzia:



Decimal i Fraction — NIE wbudowane literały, a klasy z modułów standardowej biblioteki

Istotna różnica

`int` , `float` oraz `complex` są częścią samego języka: ich literaly (`42` , `3.14` , `3+4j`) pracują bez ani jednego `import` . `Decimal` oraz `Fraction` są regularne zajęcia Python moduły `decimal` oraz `fractions` i bez niego `import` są niedostępne. Omówimy oba szczegółowo w sekcjach 4.14–4.15.

Później w tym rozdziale szczegółowo omówimy każde z pięciu narzędzi — oraz Na końcu rozdziału (sekcja 4.22) zebramy je wszystkie w jedną ostateczną kartę wyboru: jaki typ jest potrzebny dla Jakie zadanie.

float — liczby zmiennoprzecinkowe

Pierwsze spotkanie z float następuje przed zrozumieniem, dlaczego jego zachowanie czasem jest zaskakujące (następna sekcja).

`float` — typ dla liczb zmiennoprzecinkowych: wymiary, ceny, ułamki, wielkości naukowe. W przeciwieństwie do `int` , który ma taki prosta notacja, y `float` są dwie.

Regularne nagrania

float_obychnaya.py

```
pi = 3.14
print(type(pi))

whole = 2.0
print(type(whole))  #float mimo braku części ułamkowej
```

Kropka sprawia, że liczba float
Even `2.0` — `float`, nie `int`, chociaż matematycznie jest to liczba całkowita. Typ definiuje wpis w kodzie, a nie matematyczne znaczenie wartości.

Naukowy (wykładniczy) rekord

Dla bardzo dużych lub bardzo małych liczb wygodna jest notacja naukowa — `e` oznacza „pomnożyć przez 10 do potęgi”

nauchnaya.py

```
print(1e6)     # 1000000.0  (1 × 10^6)
print(2.5e-3)  # 0.0025     (2.5 × 10^-3)
```

float w informatyce

Operator `/` (normalny podział, sekcja 4.7) zawsze wraca `float`, nawet jeśli liczby są całkowicie podzielne:

delenie_float.py

```
print(10 / 2)  #5.0, nie 5
```

Praktyka: float - Zapis i pierwsze obliczenia
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/04-12/index.html)

Dlaczego 0,1 + 0,2 nie jest dokładnie 0,3

Najbardziej znana „niespodzianka”

Jedno z najsłynniejszych „niespodzianek”

surpriz.py

```
print(0.1 + 0.2)
```

Rzeczywiste wyjście

```
0.30000000000000004
```

To nie jest błąd Python

Zobaczysz dokładnie takie samo zachowanie w JavaScript, Java, C C++ niemal każdym innym języku powszechnym. Powodem nie jest błąd Python, lecz sposób, w jaki KAŻDY komputer przechowuje liczby ułamkowe w binarnym.

Prosta analogia: 1/3 w zapisie dziesiętnym

W systemie dziesiętnym `1/3` nie można zapisać jako liczba skończona Tylko liczby `0.333333...`, bez końca. Nie dlatego, że System dziesiętny jest „zły”

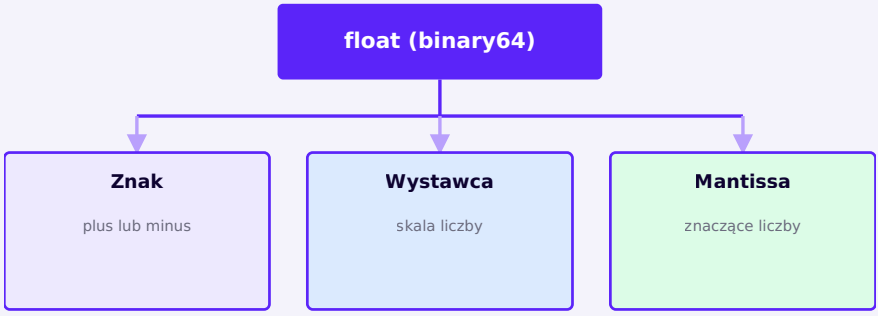
Komputer w systemie binarnym ma dokładnie ten sam problem, tylko z innymi liczbami. `0.1` w zapisie dziesiętnym wygląda „prosto” — ale w systemie binarnym zamienia się w nieskończenie powtarzającą się ułamek, dokładnie jak `1/3` w systemie dziesiętnym. Komputer jest zmuszony przyciąć go do skończonej liczby bitów — stąd ten mały błąd.

0.1 nie jest przechowywane równomiernie — przechowywane jest najbliższe możliwe przybliżenie binarne

---

Co dzieje się głębiej

Na większości nowoczesnych komputerów i w standardowym CPython `float` spełnia format sprzętowy o podwójnej precyzji, znany jako **binary64** (część standardu IEEE 754). Koncepcyjnie liczba jest przechowywana w trzech częściach:



Z czego składa się typowy float w nowoczesnym CPython (standardowy binary64)

Nie trzeba się uczyć na pamięć
Dokładna liczba bitów na każdą część (1/11/52 w standardzie binary64) — szczegół, którego nie trzeba się uczyć od razu. Ważne jest zrozumienie samej zasady: wartość jest przechowywana przybliżona, a nie to, że Python „źle liczy”. To powszechne, dobrze zbadane zachowanie — standard binary64, a nie własna zasada języka Python.

Python może pokazać dokładną wartość zapisaną

Dobrą wiadomością jest to, że podejście nie jest przypadkowe ani tajemnicze – Python może Ci pokazać, Co dokładnie jest przechowywane:

tochnoe.py

```
print((0.1).as_integer_ratio())
# (3602879701896397, 36028797018963968) to dokładny ułamek,
który faktycznie jest przechowywany
```

hex_predstavlenie.py

```
print((0.1).hex())
# '0x1.999999999999ap-4' to dokładna reprezentacja w formie
szesnastkowej
```

Główne zakończenie sekcji
0,1 nie jest przechowywane „mniej więcej”

Praktyka: 0,1 + 0,2 — przeanalizuj przybliżenie
Interaktywny notatnik bezpośrednio w przeglądarce — as_integer_ratio(), hex() i przewidywanie wyniku

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/04-13/index.html)

Poprawne porównanie float

== prawie nigdy nie jest odpowiednie dla float. Przyjrzyjmy się odpowiedniemu narzędziu — math.isclose().

razy `0.1 + 0.2` nie równa dokładnie `0.3` (Rozdział 4.11), bezpośrednie porównanie za pomocą `==` float jest zły Pomysł:

sravnenie_ploho.py

```
print(0.1 + 0.2 == 0.3)  # False!
```

math.isclose() – Porównanie „Wystarczająco blisko”

Poprawne narzędzie – porównaj z tolerancją (tolerance): zamiast "dokładnie równe" "Wystarczająco blisko":

isclose_demo.py

```
from math import isclose

print(isclose(0.1 + 0.2, 0.3))  # True
```

Nie wymyślaj własnej tolerancji na oko

Często jest to domowy czek, taki jak `abs(a - b) < 0.000001`. Nie zawsze to działa — dla bardzo dużych lub bardzo małych liczb stały próg może być zbyt surowy lub zbyt łagodny. `math.isclose()` domyślnie **względny** tolerancja (proporcjonalna do rozmiaru samych liczb), która jest znacznie bardziej wiarygodna dla liczb różnych skali.

otnositelnyj_dopusk.py

```
from math import isclose

print(isclose(1000.0001, 1000.0002))  #True — Różnica jest
niewielka w stosunku do rozmiaru liczb
print(isclose(0.0001, 0.0002))        #False — ta sama różnica
absolutna, ale ogromna w porównaniu do rozmiaru
```

Praktyka: Napraw porównanie float

Interaktywny notebook bezpośrednio w przeglądarce — zamień == na math.isclose()

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/04-14/index.html)

Zaokrąglenie: round(), floor(), ceil(), trunc()

Kilka sposobów na przekształcenie liczby ułamkowej w liczbę całkowitą — i dlaczego dają różne odpowiedzi dla liczb ujemnych.

Istnieje kilka sposobów na przekształcenie liczby ułamkowej w liczbę całkowitą – i zachowują się one Na różne sposoby, zwłaszcza dla wartości ujemnych.

round() jest regularne zaokrąglanie

round_demo.py

```
print(round(3.14159, 2))  # 3.14
print(round(7.5))          # 8
```

Zaokrąglanie „do parzystej” na granicy .5

Dokładne spojrzenie na zaokrąglenie dokładnie pośrodku daje na pierwszy rzut oka nieoczekiwane Wynik:

granica.py

```
print(round(2.5))  #2, nie 3!
print(round(3.5))  # 4
```

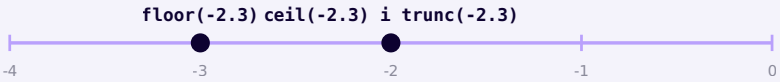
Zaokrąglanie do najbliższego parzystego

To nie jest pomyłka ani przypadek: `round()` w Python zaokrągla wartość dokładnie między dwiema liczbami całkowitymi do NAJBLIŻSZEJ PARZYSTEJ jednej, standardowej techniki statystycznej („zaokrąglanie bankowe”

math.floor(), math.ceil(), math.trunc()

floor_ceil_trunc.py

```
from math import floor, ceil, trunc

print(floor(-2.3))  # -3 — do mniejszej liczby całkowitej
print(ceil(-2.3))   # -2 — Aż do większej całości
print(trunc(-2.3))  # -2 — po prostu odpuść część ułamkową
```



floor zaokrągla w dół do -3; ceil i trunc pokrywają się tutaj – oba dają -2

Dla liczb ujemnych są trzy różne odpowiedzi

Dla liczb dodatnich `floor()` oraz `trunc()` często się pokrywają, więc różnica łatwo umyka uwadze. W przypadku negatywnych prawie nigdy się nie pokrywają, co widać na powyższej linii liczbowej.

Praktyka: round, floor, ceil, trunc

Interaktywny laptop bezpośrednio w przeglądarce – przewiduj wynik dla ujemnych wyników

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/04-15/index.html)

Decimal – gdy potrzebujesz dokładnej arytmetyki dziesiętnej

W przypadku pieniędzy i księgowości nawet najmniejszy błąd może float być nie do przyjęcia – przyjrzyjmy się narzędziu, które go usuwa.

Już to widzieliśmy `float` przechowuje przybliżenie (Rozdział 4.11). W większości zadań nie stanowi to problemu – ale przy pieniądzach nawet minimalny margines błędu kumuluje się i może stać się zauważalna. W tym celu biblioteka standardowa ma `decimal`.

decimal_demo.py

```
from decimal import Decimal

print(Decimal("0.1") + Decimal("0.2"))
#0.3 — Dokładność, brak marginesu błędu
```

Stwórz Decimal z łańcuch znaków nie float

To chyba najważniejszy praktyczny szczegół w całej sekcji:

stroka_vs_float.py

```
from decimal import Decimal

print(Decimal("19.99"))  #Decimal('19.99') to dokładnie to, co
napisałeś
print(Decimal(19.99))    #
Decimal('19.9900000000000000000000000000000000000000131628...')
— odziedziczył błąd float!
```

Dlaczego tak się dzieje?

`Decimal(19.99)` najpierw tworzy zwykły `float` `19.99` — już z tym samym przybliżeniem z sekcji 4.11 — a następnie opakuje tę przybliżoną wartość w Decimal. Dokładność jest już utracona w tym momencie. `Decimal("19.99")` czyta tekst bezpośrednio i w ogóle nie przechodzi przez float — dlatego zachowuje dokładnie wartość, którą wpisałeś.

Przykład: zakup

pokupka.py

```
from decimal import Decimal

price = Decimal("19.99")
quantity = 3
total = price * quantity
print(total)  # 59.97 — dokładnie
```

Decimal nie jest „zawsze lepsze”

	float	Decimal
Speed	Szybki — sprzętowe wsparcie	Swolniejszy – Implementacja programowa
Typowe zastosowanie	Obliczenia naukowe, pomiary, ogólne zastosowania	Pieniądze, księgowość, gdzie ważna jest dokładność dziesiętna
Dokładność	Przybliżenie binarne	Dokładny zapis dziesiętny

Nie wymieniaj każdego float na Decimal
Decimal jest właściwym wyborem dla pieniędzy i księgowości, gdzie precyzja dziesiętna i przewidywalne zaokrąglanie są ważne. W większości obliczeń naukowych i codziennych stosuje się standardowe `float` szybsze i absolutnie wystarczające.

Praktyka: Decimal i pieniądze
Interaktywny laptop bezpośrednio w przeglądarce – Decimal(str) vs Decimal(float)

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/04-16/index.html)

Fraction są dokładnymi ułamkami zwyczajnymi

Gdy potrzebna jest nie dziesiętna, ale zwykła dokładna ułamek — licznik przez mianownik, bez przybliżenia.

Innym dokładnym narzędziem numerycznym ze standardowej biblioteki jest `Fraction` zwykłym ułamkiem wymiernym „licznik/mianownik”

fraction_demo.py

```
from fractions import Fraction

third = Fraction(1, 3)
print(third)  # 1/3
```

1/3

Fraction(1, 3) — dokładnie jedna trzecia, bez przybliżenia

Arytmetyka z Fraction jest dokładna

fraction_math.py

```
from fractions import Fraction

result = Fraction(1, 3) + Fraction(1, 6)
print(result)  # 1/2 — dokładnie, nie przybliżeniowo
```

Fraction z tekstu i float też jest różnicą

fraction_stroka.py

```
from fractions import Fraction

print(Fraction("0.1"))  #1/10 — czytane z tekstu dziesiętnego poprawnie
print(Fraction(0.1))    #3602879701896397/36028797018963968 — odziedziczyłem podejście float!
```

Ta sama logika co z `Decimal` w 4.14: Stwórz najpierw `float` takie podejście już wcześniej miało miejsce `Fraction` zdążyłem go otrzymać.

float, Decimal i Fraction obok siebie

Tool	Przedstawia	Dokładny dla	Typowe zastosowanie
`float`	Przybliżenie binarne	Potęgi dwójki i ich sumy	Nauka, pomiary, ogólne obliczenia
`Decimal`	Dziesiętny	Wartości dziesiętne	Pieniądze, księgowość
`Fraction`	Liczba wymierna p/q	Dowolny ułamek normalny	Dokładna arytmetyka wymierna

float vs Decimal vs Fraction

Czy potrzebujesz regularnych obliczeń lub pomiarów? → **`float`**

Czy potrzebujesz precyzji dziesiętnej (pieniędzy)? → **Decimal**

Czy potrzebujesz dokładnej części p/q? → **Fraction**

Trzy narzędzia do trzech różnych rodzajów precyzji

Praktyka: Fraction i dokładne ułamki
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/04-17/index.html)

complex są liczbami zespolonymi

Ostatni z trzech wbudowanych typów liczb dotyczy problemów inżynieryjnych i naukowych, gdzie potrzebne są zarówno części rzeczywiste, jak i wyimaginowane.

`complex` jest typem dla liczb zespolonych: wartości postaci „prawdziwa część + wyimaginowana część" `j` (i nie `i` jak w matematyce — konwencja inżynierska, nie mylić z obecnym określeniem).

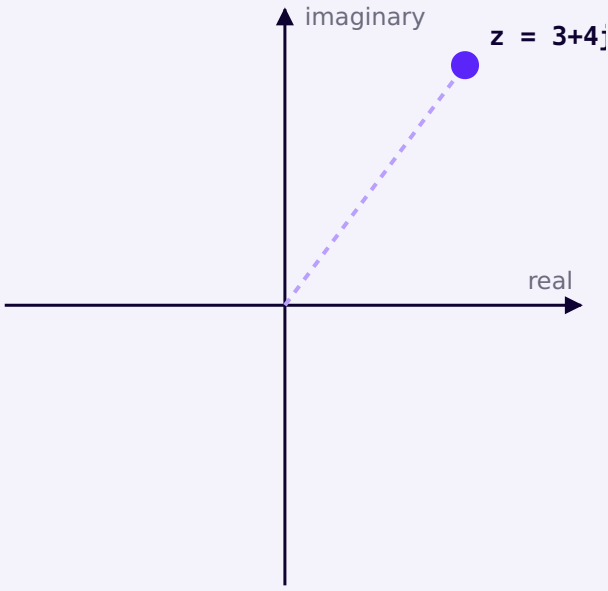
complex_demo.py

```
z = 3 + 4j
print(z)
print(type(z))
```

Część rzeczywista i wyimaginowana

real_imag.py

```
z = 3 + 4j
print(z.real)  # 3.0
print(z.imag)  # 4.0
print(abs(z))  #5.0 to długość wektora od początku
```



z = 3 + 4j na płaszczyźnie zespolonej

Gdzie występuje w praktyce

W zwykłym kodzie aplikacyjnym liczby zespolone są rzadkością. Ale w niektórych obszarach są rzadkie są niezastąpione:

- **Inżynieria elektryczna** — obliczanie prądu przemiennego;
- **przetwarzanie sygnałów** to transformata Fouriera i powiązane obliczenia;
- **Systemy sterowania** – analiza stabilności;
- **Fizyka** — mechanika kwantowa, procesy falowe.

Jeśli żadna z tych dziedzin nie należy do Twoich najbliższych planów — całkowicie normalne jest po prostu wiedzieć, że `complex` istnieje i wracam do niego Rozdział, gdy jest to naprawdę potrzebne.

cmath jest matematyką dla liczb zespolonych

Moduł Regularny `math` (Rozdział 4.18) nie wie, jak z nim pracować Liczby zespolone — istnieje dla nich osobny moduł `cmath`:

cmath_demo.py

```
import cmath

z = 3 + 4j
print(cmath.phase(z))  # kąt wektora z w radianach
```

Praktyka: Płaszczyzna zespolona
Interaktywny laptop bezpośrednio w przeglądarce – real, imag, abs() dla liczb złożonych

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/04-18/index.html)

Konwersja typów liczb

Liczby można przekształcić w siebie nawzajem i w tekst, ale nie zawsze odwrotnie i nie zawsze, jak się wydaje na pierwszy rzut oka.

Każdy typ numeryczny ma własną funkcję transformatora o tej samej nazwie: `int()`, `float()`, `complex()`. Ponadto każdą liczbę można przekonwertować na tekst funkcją `str()` — i odwrotnie, tekst podobny do liczby może być Odwróć to.

preobrazovanie.py

```
whole = int(3.99)
print(whole)      #3 – część ułamkowa jest odrzucana zamiast zaokrąglonej

decimal = float(7)
print(decimal)    # 7.0

text = str(42)
print(text, type(text))   # "42" <class 'str'>
```

int() odrzuca część ułamkową, zamiast zaokrąglać
`int(3.99)` równa się `3`, nie `4` jest **obcięcie** (truncation) ku zeru, nie zaokrąglając. `round()` (Rozdział 4.13).

Kierunek obcięcia jest również ważny dla liczb ujemnych — `int()` zawsze zmierza do zera, a nie „w dół” w sensie matematycznym:

int_otricatelnoe.py

```
print(int(3.9))   # 3
print(int(-3.9))  #-3 zamiast -4
```

Przekształć tekst na liczbę

tekst_v_chislo.py

```
age_text = "10"
age = int(age_text)
print(age + 5)   # 15 — teraz to prawdziwa liczba
```

Nie każdy tekst da się przekonwertować

`int("десять")` spowoduje `ValueError` - Python nie potrafi odczytać liczb w słowach. Można przekonwertować tylko tekst wyglądający jak liczba.

Parsowanie tekstu z określeniem systemu liczbowego

Widzieliśmy to już w sekcji 4.3 – `int()` potrafi odczytać tekst, który nie jest w stanie Tylko w systemie dziesiętnym:

s_osnovaniem.py

```
print(int("1010", 2))  # 10
print(int("FF", 16))   # 255
```

Tworzenie łańcuch znaków z liczb: konkatenacja → f-łańcuch znaków

Klasyczne podejście

```
age = 10
message = „Wiek: ” +
str(age) + „ lat”
print(message)
# koniecznie ręcznie opakować
liczbę w str()
```

Nowoczesny Python 3.14

```
age = 10
message = f"Wiek: {age} lat"
print(message)
#f-łańcuch znaków wykonuje
konwersję wewnątrz samego {}
```

Co używać dzisiaj: f-stringi (f-strings), pojawiły się w Python 3.6 i od tego czasu stały się standardem. Są krótsze, mniej podatne na błędy (nie trzeba pamiętać o str()) i pozwalają od razu pisać wyrażenia wewnątrz nawiasów klamrowych.

Praktyka: int(), float(), str() i parsowanie baz
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/04-04/index.html)

Moduł math

Gotowy zestaw narzędzi matematycznych ze standardowej biblioteki — uporządkowany według znaczenia, a nie alfabetycznie.

Moduł `math` ze standardowej biblioteki — zestaw gotowych materiałów Narzędzia matematyczne, których nie musisz sam pisać.

import.py

```
import math

print(math.sqrt(16))  # 4.0
```

math

STAŁE

pi · e · tau · inf · nan

PIERWIASTKI I POTĘGI

`sqrt · isqrt · pow`

ZAOKRĄGLANIE

`floor · ceil · trunc`

MATEMATYKA CAŁKOWITA

`factorial · gcd · lcm`
`comb · perm`

GEOMETRIA

`hypot · dist`

TRYGONOMETRIA

`sin · cos · tan`
`radians · degrees`

math jako zorganizowany zestaw cech, a nie przypadkowy lista nazw

Stałe

konstanty.py

```python
import math

print(math.pi)   # 3.141592653589793
print(math.e)    # 2.718281828459045
print(math.tau)  # 6.283185307179586 – to 2 * pi
```

Pierwiastki i potęgi

korni.py

```python
import math

print(math.sqrt(16))    #4.0 to pierwiastek kwadratowy (float)
print(math.isqrt(16))   #4 to pierwiastek całkowity kwadratowy
(int)
```

Matematyka całkowita

celochislennaya.py

```python
import math

print(math.factorial(5))  # 120 – 5!
print(math.gcd(12, 18))   #6 jest największym dzielnikiem
wspólnym
print(math.lcm(4, 6))     #12 – Najniższy wspólny wielokrotnik
print(math.comb(5, 2))    #10 – Liczba kombinacji
print(math.perm(5, 2))    # 20 – liczba permutacji
```

Geometria

geometriya.py

```python
import math

print(math.hypot(3, 4))       # 5.0 — przeciwprostokątna (długość wektora)
print(math.dist((0, 0), (3, 4)))  #5.0 — Odległość między punktami
```

Trygonometria – Szybki przegląd

Funkcje trygonometryczne działają w **radiany**, nie dyplomy:

trig.py

```python
import math

angle_deg = 90
angle_rad = math.radians(angle_deg)
print(math.sin(angle_rad))  # 1.0
```

Jeśli trygonometria nie jest teraz częścią twojego zadania, to całkowicie normalne, że wracasz do niej podrozdział później.

Praktyka: Moduł math
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/04-19/index.html)

Liczby losowe: random i secrets

Dwa moduły do losowości — i zasada, którą warto zapamiętać od pierwszego dnia: nie każda losowość jest równie bezpieczna.

Moduł `random` z biblioteki standardowej generuje **pseudo-losowe** liczby — wyglądają na losowe i są wystarczająco dobre do gier, symulacji i przykładów edukacyjnych.

random_demo.py

```
import random

print(random.randint(1, 6))
# losowa liczba całkowita od 1 do 6 włącznie
print(random.random())        # liczba losowa między 0,0 a 1,0
print(random.choice([„Eagle”, reszki]))  # losowy wybór z listy
```

Powtarzalność poprzez seed

Czasem chcesz, aby powtarzała się „losowa” `seed()`:

seed_demo.py

```
import random

random.seed(42)
print(random.randint(1, 100))  # to zawsze ta sama liczba dla
tej samej seed
```

Wybór modułu zależy od tego, czy stawką jest bezpieczeństwo

BEZPIECZEŃSTWO: random – nie dotyczy haseł i tokenów

Nie generuj haseł i tokenów za pomocą modułu random

`random` zoptymalizowany pod kątem szybkości i jakości statystycznej, a nie nieprzewidywalności wobec atakującego — jego wynik zasadniczo można przewidzieć, znając wewnętrzny stan generatora. Do wszystkiego, co „wrażliwe na bezpieczeństwo” (hasła, tokeny sesji, klucze) używaj modułu `secrets`.

secrets_demo.py

```
import secrets

token = secrets.token_hex(16)
print(token)  # kryptograficznie bezpieczny token losowy
```

Praktyka: random vs secrets
Interaktywny laptop bezpośrednio w przeglądarce – wybierz odpowiednie narzędzie do zadania

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/04-20/index.html)

statistics, inf i nan

Gotowe narzędzia statystyczne — i specjalne stany liczbowe, które warto umieć rozpoznać.

Moduł statistics

Proste obliczenia statystyczne są już w standardowej bibliotece — nie trzeba ich pisać ręcznie:

statistics_demo.py

```
import statistics

scores = [85, 90, 78, 92, 88]

print(statistics.mean(scores))    #86.6 — Medium
print(statistics.median(scores))  #88 to mediana (średnia w kolejności)
```

tryb i odchylenie standardowe

```mode_stdev.py
import statistics

scores = [85, 90, 78, 92, 88]

print(statistics.mode(scores))    # Najczęściej stosowana wartość
print(statistics.pstdev(scores))  # odchylenie standardowe populacji
```

To tylko niewielka część modułu `statistics` jest zadanie tego Rozdział nie polega na przekształceniu kursu w statystykę, lecz po to, by pokazać: typowe obliczenia statystyczne są już gotowe w Python roku i nie musisz za każdym razem ręcznie pisać średniej formuły.

Szczególne znaczenia: inf i nan

Oprócz regularnych liczb, `float` potrafi reprezentować kilka specjalnych stanów:

```inf_nan_demo.py
positive_infinity = float("inf")
negative_infinity = float("-inf")
not_a_number = float("nan")

print(positive_infinity)  # inf
print(not_a_number)       # nan
```

Szczególne float znaczenia

INF

+∞

nieskończoność

`math.isinf()`

-INF

−∞

minus nieskończoność

`math.isinf()`

NAN

nie liczba

not a number

`math.isnan()`

Trzy szczególne stany, które może przyjąć float, są sprawdzane przez trzy różne funkcje

nan nie jest równy nawet samemu sobie

Jedna z najbardziej nieoczekiwanych właściwości `nan`: porównanie `nan == nan` zwroty `False`. To nie jest błąd Python, a część standardu IEEE 754 (ten sam standard, o którym wspominaliśmy w rozdziale o float) — nan jest specjalnie zdefiniowany jako „nierówny niczemu, włącznie z samym sobą.")

nan_sravnenie.py

```
x = float("nan")
print(x == x)  # False!
```

Jak poprawnie sprawdzać te stany

proverki.py

```
import math

x = float("nan")
y = float("inf")

print(math.isnan(x))     # True
print(math.isinf(y))     # True
print(math.isfinite(5.0))  #True jest regularną liczbą
skończoną
```

Takie wartości nie są wymyślone — naprawdę pojawiają się w danych i obliczeniach: dzielenie, które daje przepełnienie, brakujące dane w analizie, wynik nieokreślonej operacji. Warto umieć je rozpoznać i świadomie obsłużyć, a nie dziwić się, skąd się wzięły.

Praktyka: statistics, inf i nan
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/04-21/index.html)

Błędy numeryczne i debugowanie

Dziewięć krótkich laboratoriów — od prawdziwych wyjątków po prawidłowe, ale nieoczekiwane wyniki liczbowe.

zebrałem dziewięć krótkich laboratoriów — niektóre z nich dotyczyły prawdziwych błędów z wyjątekm, część dotyczy ważnego, ale nieoczekiwanego wyniku numerycznego. Ważne jest, aby umieć rozróżnić te dwa Scenariusz.


Co poszło nie tak?
Python przestał z wyjątekm
ZeroDivisionError
ValueError
TypeError
Przeczytaj traceback i napraw przyczynę
Kod wykonywany bez wyjątku
Ale efekt wygląda niespodziewanie
Sprawdzanie modelu matematycznego i reprezentacji liczb
Poprawiamy kod, oczekiwania lub wybór typu liczbowego

Laboratorium 1.

Dlaczego 10/2 daje 5,0?

przewidzieć i uruchomić

```
print(10 / 2)
```

Obserwacja: wynik — `5.0`, nie `5`.

Wyjaśnienie: Operator `/` w Python ZAWSZE zwraca float (Rozdział 4.7), nawet gdy dzieli się całkowicie.

Laboratorium 2.

Dlaczego int(3.99) jest równe 3?

przewidzieć i uruchomić

```
print(int(3.99))
```

Obserwacja: wynik — `3`, nie `4`.

Wyjaśnienie: `int()` odrzuca część ułamkową (obcinanie do zera), a nie zaokrągla (dział 4.16).

Laboratorium 3.
Dlaczego porównanie 0,1 + 0,2 == 0,3 nie działa?

przewidzieć i uruchomić

```
print(0.1 + 0.2 == 0.3)
```

Obserwacja: wynik — `False`.

Wyjaśnienie: float przechowuje przybliżenie (Rozdział 4.11) – zastosowanie `math.isclose()` zamiast `==`.

Laboratorium 4.
Dlaczego Decimal(0.1) wygląda dziwnie?

przewidzieć i uruchomić

```
from decimal import Decimal
print(Decimal(0.1))
```

Obserwacja: wynikiem jest długa, niekołowa frakcja, a nie `0.1`.

Wyjaśnienie: argument jest float, co oznacza, że przybliżenie z sekcji 4.11 miało miejsce PRZED utworzeniem Decimal. Use `Decimal("0.1")`.

Laboratorium 5.

Dlaczego 7 // 3 różni się od 7/3?

przewidzieć i uruchomić

```
print(7 // 3)
print(7 / 3)
```

Obserwacja: `2` przeciw `2.333...`.

Wyjaśnienie: `//` jest dzieleniem całkowitym (odrzuca część ułamkową wyniku), `/` — zwykłe dzielenie, zawsze float.

Laboratorium 6.

Przewidź: -7 // 3 i -7 % 3

przewidzieć i uruchomić

```
print(-7 // 3)
print(-7 % 3)
```

Obserwacja: `-3` oraz `2` nie jest tym, czego wielu się spodziewa.

Wyjaśnienie: `//` zaokrągla do minus nieskończoności zamiast zera (Rozdział 4.7).

Laboratorium 7.

Popraw: int(„FF”

przewidzieć i uruchomić

```
print(int("FF"))
```

Obserwacja: ValueError: invalid literal for int() with base 10: 'FF'

Wyjaśnienie: bez wskazania podstawy `int()` oczekuje zapisu dziesiętnego. Należy: `int("FF", 16)` .

Laboratorium 8.

Napraw ZeroDivisionError

przewidzieć i uruchomić

```
price = 100
quantity = 0
print(price / quantity)
```

Obserwacja: ZeroDivisionError: division by zero.

Wyjaśnienie: przed dzieleniem warto sprawdzić dzielnik pod kątem zera, jeśli może być zerowy — szczegółowe warunki będą w następnym rozdziale; tutaj wystarczy poznać sam typ błędu.

Laboratorium 9.

Wybierz właściwy typ numeru

przewidzieć i uruchomić

```
# Zadanie: oblicz dokładną kwotę czeku w rublach i
kopiejkach
```

Obserwacja: float może dawać błąd w ułamkowych ilościach.

Wyjaśnienie: właściwy wybór pieniędzy — `Decimal` (Rozdział 4.14) zamiast `float`.

Praktyka: Debugowanie numerycznych numerów błędów
interaktywny laptop bezpośrednio w przeglądarce – znajdź i napraw typowe problemy numeryczne

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/04-22/index.html)

Mini-projekty i wyniki rozdziałów

Zebranie wszystkiego, co omówiliśmy, w trzy małe, ale realne projekty — i podsumowanie rozdziału ostateczną mapą wyboru typu numerycznego.

Trzy małe, ale prawdziwe projekty — każdy używa tylko tego, co już omówiliśmy w tym rozdziale (i w rozdziale 3).

Kalkulator zakupu A: projektu

wpisz cenę i ilość, oblicz kwotę. Porównaj float i Decimal na tym samym Przykład:

kalkulyator_pokupki.py

```python
from decimal import Decimal

# Wersja na float — zwykła, ale z ryzykiem błędu
price_float = 19.99
quantity = 3
print(price_float * quantity)  # 59,97 — zbiegło się to tutaj,
ale nie zawsze tak szczęśliwie

# Decimal Wersja — Przewidywalna dokładność za pieniądze
price_decimal = Decimal("19.99")
print(price_decimal * quantity)  # 59,97 — dokładnie,
gwarantowane
```

Podstawowa praktyka
Challenge
Dodaj podatek (na przykład 23%) do kwoty Decimal. wersji

Projekt B: Przetwornik Czasu

Wprowadzamy łączną liczbę sekund — liczymy godziny, minuty i sekundy przez `//` oraz `%` (Rozdział 4.7) — tutaj operatorzy mają teraźniejszość Zastosowanie:

konverter_vremeni.py

```python
total_seconds = 3725

hours = total_seconds // 3600
minutes = (total_seconds % 3600) // 60
seconds = total_seconds % 60

print(f"{hours}godziny {minutes}m {seconds}z")  # 1h 2m 5s
```

★★ Zadanie samodzielne
Challenge
Testuj swoją funkcję przez 0 sekund i przez 90000 sekund (więcej niż jeden dzień).

Projekt C: kalkulator geometryczny

Wprowadzamy promień okręgu — liczymy średnicę, obwód i pole za pomocą `math.pi`:

geometricheskij.py

```python
import math

radius = 5
diameter = radius * 2
circumference = 2 * math.pi * radius
area = math.pi * radius ** 2

print(f"Średnica: {diameter}")
print(f"Obwód: {circumference:.2f}")
print(f"Obszar: {area:.2f}")
```

Challenge

Challenge · Systemy liczbowe

Opcjonalnie: napisz mini-„tłumacza”

Praktyka: mini-projekty rozdziału 4

Interaktywny laptop bezpośrednio w przeglądarce – kup kalkulator, konwerter czasu, geometrię

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/04-05/index.html)

Ostatnia karta: Jakiego typu numeru potrzebuję?

Jaki typ/liczbowy/ narzędzie jest mi potrzebne?

Potrzebujesz całego rachunku?" → **int**

Potrzebny zwykły pomiar ułamkowy? → **float**

Potrzebujesz dokładnej kwoty dziesiętnej (pieniędzy)?	→ `Decimal`
Czy potrzebujesz dokładnej części p/q?	→ `Fraction`
Czy potrzebujesz prawdziwej + wyimaginowanej części?	→ `complex`
Czy potrzebujesz gotowych funkcji matematycznych?	→ `math / cmath`
Potrzebujesz losowości w grach/ symulacjach?	→ `random`
Czy potrzebna jest losowość dla bezpieczeństwa?	→ `secrets`

Rozdział 4 Mapa Podsumowania – Wracaj do niej w przyszłych projektach

Podsumowanie rozdziału

Co możemy teraz zrobić

- Zrozumieć różnicę między liczbą matematyczną a jej reprezentacją w komputerze oraz że nazwy odnoszą się do niezmiennych obiektów liczbowych — a nie do „pudełek”
- Pewnie pracuj z int: arbitralną precyzją, łącząc bool, podkreślenia i separatory.
- Rozumiemy układy liczbowe — decimal/binary/octal/hex — i wiemy, jak je przeliczać.
- Znamy pełny zestaw operatorów arytmetycznych i ich kolejność wykonywania.
- Rozróżniamy /, // i % — i rozumiemy, jak działa ujemne floor- dzielenie.
- Zrozumieć, dlaczego 0,1 + 0,2 nie jest dokładnie 0,3 i wiemy, jak porównać float przez math.isclose().
- Potrafimy zaokrąglać liczby przy użyciu round(), floor(), ceil() i trunc() — i znamy różnicę dla liczb ujemnych.
- Wiedzieć, kiedy używać Decimal (pieniądze) i Fraction (dokładne ułamki) zamiast float.
- Rozumiemy complex i złożoną płaszczyznę, wiemy, jak korzystać z math/cmath, random/secrets i statistics.
- Potrafimy rozpoznawać inf i nan oraz debugować typowe błędy liczbowe.
- Zebrałem trzy mini-projekty, które wykorzystują Python liczby do rzeczywistych zadań.

co dalej

W rozdziale 5 będziemy kontynuować pracę z liczbami — porównania, operacje logiczne i pierwsze konstrukcje warunkowe.

ROZDZIAŁ 5 · STR. 173

Pobawmy się liczbami!

Głębokie zanurzenie w obliczenia matematyczne Python: wyrażenia, operatory i ich priorytety, tłumaczenie formuł z matematyki na kod, moduł math (pierwiastki, geometria, trygonometria, logarytmy), moduł random (pseudolosowość, seed, odtwarzalność) oraz debugowanie obliczeń.

Wyrażenia i podstawowe operacje

Wyrażenie to operandy plus operator. Zacznijmy od czterech operacji znanych ze szkoły, ale teraz zobaczymy je nie tylko w kodzie, ale także na osi liczbowej i w formie prostokąta.

W rozdziale 4 zrozumieliśmy, czym są liczby. Teraz nauczymy Python prawdziwie z nimi pracować — liczyć, porównywać, łączyć. Zaczniemy od najważniejszego słowa tego rozdziału: **wyraz twarzy**.

---

czym jest wyrażenie

`5 + 3` jest **wyraz twarzy**: Ma **operandy** (wartości, z którymi pracujemy — `5` oraz `3`) i **Operator** (akcja — `+`). Każde wyrażenie ma wynik — samo wyrażenie można podstawić tam, gdzie oczekiwana jest wartość:

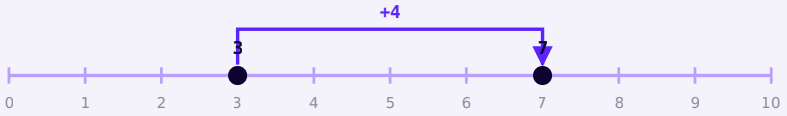
vyrazhenie.py

```
print(5 + 3)        # wyraz twarzy tuż w print()
summa = 5 + 3       # wynik wyrażenia jest zapisywany do
print(summa * 2)    # summa — też operand w nowym wyrażeniu
```

Wyrażenie nie jest tym samym co polecenie (statement)
`summa = 5 + 3` jest całkowicie **nauczanie** (przywłaszczenie), oraz `5 + 3` w środku — **wyraz twarzy**: Wyrażenie zawsze ma wartość, ale instrukcja nie. Dlatego `print(x = 5)` — błąd (`x = 5` nic nie zwraca), oraz `print(x == 5)` jest zwykłym kodem.

Dodawanie i odejmowanie — ruch po osi liczbowej

Najprostszy model dla `+` oraz `-` — Skok wzdłuż osi liczb: sumowanie skacze w prawo, odejmowanie w lewo.



3 + 4 = 7 – skocz 4 kroki w prawo od punktu 3

slozhenie.py

```
print(3 + 4)    #7 — Skok w prawo
print(3 - 4)    # -1 — przeskocz w lewo, możesz przejść do
wartości ujemnych
```

Mnożenie to prostokąt komórek

Mnożenie `a * b` to powierzchnia prostokąta z bokami `a` oraz `b`: wiersze autorstwa `a` komórki i takie rzędy `b` sztuk.

4 × 3 = 12

4 × 3 = 12 – 3 rzędy po 4 komórki

umnozhenie.py

```
print(4 * 3)    # 12
print(4 * 0)    #0 — Prostokąt bez wysokości nie ma powierzchni
```

Dzielenie jest zawsze liczbą ułamkową

W Python `/` jest **zwykłe dzielenie**, i to **zawsze** zwroty `float`, nawet jeśli liczby dzielą się bez reszty:

delenie.py

```
print(6 / 3)    #2.0 — float, nie 2
print(7 / 2)    # 3.5
```

Nie da się dzielić przez zero
`5 / 0` przyczyny `ZeroDivisionError` Python sam nie wymyśla tego wyniku, ale szczerze informuje, że operacja nie ma sensu.

Operatorzy i prawdziwe życie

Operator	Co robi	Przykład z życia codziennego
`+`	dodawanie	liczba paragonów w koszyku
`-`	odejmowanie	saldo konta po zakupie
`*`	mnożenie	cena × ilość
`/`	dywizja	suma rachunku, podzielona między przyjaciół

Pro `//` (dzielenie całkowite) i `%` (reszta) znajduje się w następnej sekcji: to osobny i bardzo ważny temat.

Praktyka: wyrażenia i +, -, *, /
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/05-01/index.html)

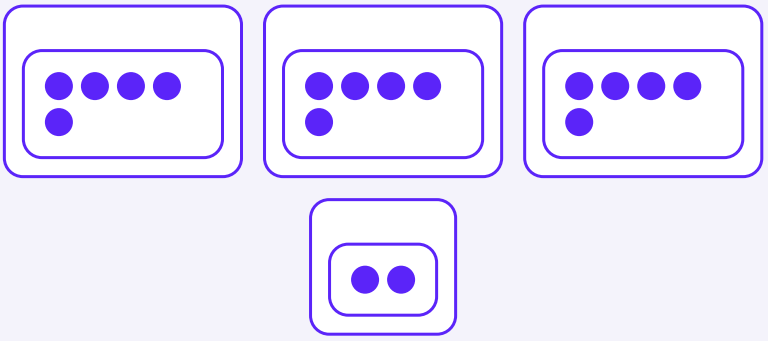
Podział z resztą

Podziel liczby na grupy i zapoznaj się z dwoma operatorami, które nie są dostępne w zwykłym kalkulatorze.

Oprócz zwykłego dzielenia `/`, Python ma dwa operatory, które Nie jest to na kalkulatorze szkolnym, ale prawie żaden program nie może się bez nich obejść.

Dzielenie całkowite `//`

Zwraca tylko część całkowitą wyniku – „ile razy jedna liczba mieści się w innej całość”



`17 // 5 = 3, 17 % 5 = 2`

17 cukierków po 5 w paczce — 3 pełne paczki i 2 cukierki pozostały

celochislennoe.py

```
print(17 // 5)   #3 — Trzy pełne paczki
print(17 / 5)    #3.4 to zwykły podział, dla porównania
```

Pozostałość dywizji `%`

Zwraca co **lewo** po włożeniu do worków – na diagramie powyżej znajdują się 2 cukierki. Jeden z najbardziej użytecznych operatorów w programowaniu: na przykład dla sprawdzanie parzystości liczby.

ostatok.py

```
print(17 % 5)    #2 — Tyle zostało
print(10 % 2)    # 0 — jeśli reszta to 0, liczba jest parzysta
print(11 % 2)    #1 — a jeśli 1 jest nieparzyste
```

Sprawdzanie parzystości to klasyczna technika

`число % 2 == 0` jest najczęstszym sposobem sprawdzenia, czy liczba jest parzysta. Ta technika pojawi się wiele razy na kursie.

`divmod()` — oba wyniki jednocześnie

Jeśli potrzebna jest zarówno część całkowita, jak i reszta jednocześnie, nie trzeba pisać `//` oraz `%` osobno — jest wbudowany Funkcja `divmod()`:

divmod_primer.py

```
print(divmod(17, 5))   # (3, 2) — krotka: cała część i reszta
chast, ostatok = divmod(17, 5)
print(chast, ostatok)  # 3 2
```

Przykłady z rzeczywistego świata

Zadanie	Ekspresja	Rezultat
Ile pełnych drużyn po 4 osoby z 22	`22 // 4`	5
Ilu ludzi pozostanie bez zespołu	`22 % 4`	2
Ile godzin i minut to 137 minut	`divmod(137, 60)`	(2, 17)

Praktyka: //, % i divmod()
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji
Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/05-07/index.html)

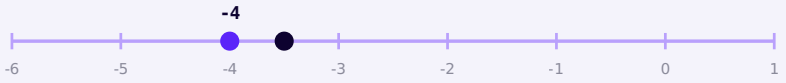
Ujemne dzielenie całkowite przez floor

Najczęstsza niespodzianka w tym temacie — co robi // z liczbami ujemnymi. Analizujemy na osi liczbowej.

Co się stanie, jeśli podzielisz się resztą **negatywne** liczba? Tutaj Python zachowuje się inaczej, niż wielu oczekuje ze szkoły — i ważne jest, aby to zrozumieć teraz, a nie w momencie debugowania.

Zaokrąglenie jest zawsze w dół — czyli do minus nieskończoności

`//` w Python nazywa się *floor division* — „dzielenie z zaokrąglanie na podłogę” **mniejszego** , a dla liczb ujemnych "mniej" oznacza "bardziej ujemne":



-7 // 2 = -4 — zaokrąglenie na podłogę oznacza przesunięcie się w LEWO wzdłuż osi liczby, do −∞

otricatelnoe_floor.py

```
print(-7 // 2)    #-4, nie -3! Python zaokrągleń do minus nieskończoności
print(7 // 2)     #3 — Dla liczb dodatnich nie ma różnicy przy zaokrągleniu „do zera”
print(-7 / 2)
# -3.5 — Dzielenie normalne uczciwie pokazuje część ułamkową
```

Nie „zaokrąglanie do zera”
W odniesieniu do innych języków programowania `-7 // 2` dałoby `-3` (zaokrąglone do zera). W Python wynik to `-4` : To świadomy wybór języka, aby poniższy wzór działał zawsze, bez wyjątków dla liczb ujemnych.

Reszta na dzieleniu negatywnym

Reszta wpisu Python zawsze pokrywa się z tym znakiem **dzielnik**:

otricatelnyj_ostatok.py

```
print(-7 % 2)     #1 – Reszta jest dodatnia, ponieważ dzielnik
(2) jest dodatni
print(7 % -2)     #-1 – Reszta jest ujemna, ponieważ dzielnik
(-2) jest ujemny
```

Tożsamość, która zawsze działa

Bez względu na to, jak zmieniają się znaki, dla dowolnych `a` oraz `b` (z wyjątkiem dzielenia przez 0) jest prawdziwe:

```
a == (a // b) * b + a % b
                ↓
-7 == (-7 // 2) * 2 + (-7 % 2)
                ↓
      -7 == (-4) * 2 + 1
                ↓
         -7 == -8 + 1
                ↓
          -7 == -7 ✓
```

Tożsamość dzielenia z resztą jest testowana w przykładzie -7 i 2

tozhdestvo.py

```
a, b = -7, 2
print((a // b) * b + a % b == a)   #True — Tożsamość zawsze działa
```

Praktyka: Ujemna część floor-a.
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/05-08/index.html)

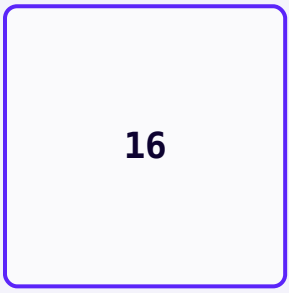
Stopnie naukowe i korzenie

Podnoszenie do potęgi i pierwiastkowanie — ten sam obraz geometryczny, odczytany w dwóch kierunkach.

Teraz jest operacja, która ma najpiękniejszy obraz geometryczny ze wszystkich: Dyplom.

Kwadrat liczby to pole kwadratu

`x ** 2` to powierzchnia kwadratu z bokiem `x`:



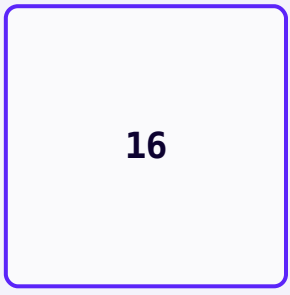
4 ** 2 = 16 to pole kwadratu o boku 4

kvadrat.py

```
print(4 ** 2)    # 16
print(4 ** 3)    #64 — Cube: Ten sam pomysł, ale w trzech
wymiarach (tom)
```

Pierwiastek kwadratowy to operacja odwrotna: od pola do boku

Jeżeli znana jest powierzchnia kwadratu, można znaleźć długość boku — to właśnie pierwiastek kwadratowy. Ta sama ilustracja, tylko czytamy ją w odwrotną stronę: mamy liczbę 16 w kwadracie i szukamy boku.



√16 = 4 — problem odwrotny: mając daną powierzchnię, szukając boku.

koren.py

```
print(16 ** 0.5)     # 4.0 — stopień 0.5 to pierwiastek
kwadratowy
import math
print(math.sqrt(16))  # 4.0 — to samo, ale wyraźnie i
zrozumiale dla czytelnika
```

math.sqrt() czyta się lepiej niż ** 0,5
Obie opcje działają równie poprawnie, ale `math.sqrt(16)` natychmiast mówi czytelnikowi kodu: „Oto oni szukają źródła.” `16 ** 0.5` przypomina, że 0,5 to pierwiastek.

Trzy sposoby potęgowania — i w czym jest różnica

Metoda	Co wraca	Przykład
`**`	int jeśli oba operandy int; w przeciwnym razie float	`2 ** 10` → `1024`
`pow(a, b)`	to samo co `a ** b`	`pow(2, 10)` → `1024`
`math.pow(a, b)`	zawsze float, nawet dla liczb całkowitych	`math.pow(2, 10)` → `1024.0`

pow() ma trzeci, opcjonalny argument
`pow(a, b, m)` liczy `(a ** b) % m`, ale znacznie szybciej — dla dużych liczb przyspiesza pracę tysiąckrotnie. Jest wykorzystywana na przykład w kryptografii. Na razie wystarczy wiedzieć, że taka możliwość istnieje.

Praktyka: Stopnie naukowe i korzenie
Ten sam laptop, który łączy //, % i ** razem — czas ćwiczyć wszystkich operatorów specjalnych naraz
Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/05-02/index.html)

Operatory unarne

Jeden operand, a nie dwa — i jedna bardzo znana pułapka priorytetu operacji.

`-5` — to nie „minus pięć jako osobna liczba”, tylko zastosowanie **unary** operator `-` do liczby `5`. Unary oznacza „z jednym operandem” `8 - 3`).

unarnyj.py

```
x = 5
print(-x)     # -5 — Unary minus changes sign
print(+x)     # 5 — jednolity plus prawie nic nie robi (rzadko
używany)
print(8 - 3)  #5 — I to jest odejmowanie binarne, dwa operandy
```

Klasyczna pułapka: `-2 ** 2`

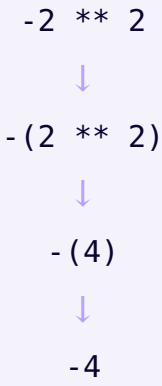
Wielu się spodziewa `-4`, ale podejrzewana `4` (jakby najpierw liczyć `(-2) ** 2`). Sprawdźmy:

lovushka.py

```
print(-2 ** 2)     # -4
print((-2) ** 2)   #4 to inny numer!
```

`( )`	nawiasy zawsze są pierwsze
`**`	potęgowanie
`+x -x`	operator jednoargumentowy plus/minus
`* / // %`	mnożenie, dzielenie, dzielenie całkowite, reszta
`+ -`	dodatku, odejmowanie

** stoi WYŻEJ niż minus jednoargumentowy — dlatego -2 ** 2 najpierw podnosi 2 do kwadratu



** wykonuje się przed unarnym minusom — najpierw 2 ** 2 = 4, następnie stosuje się minus

**** to jedyny tego wyjątek tego typu**

Dla wszystkich innych operatorów priorytet zachowuje się zgodnie z oczekiwaniami. To właśnie dzięki tej funkcji `**` doświadczony programista prawie zawsze pisze `(-2) ** 2` wyraźnie, nawiasami — nawet gdy priorytet jest formalnie jasny.

Praktyka: Operatory unarne

interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/05-09/index.html)

Kolejność przypisania i obliczeń

Krótszy wpis o zmiennych – oraz pełna mapa tego, co pojawia się najpierw, a co później.

Operacje przydziałowe

Zmiana zmiennej „opartej na samej sobie” `score = score + 10` możliwe, ale Python oferuje krótszy wpis:

prisvaivanie.py

```
score = 0
score += 10   # to samo, co score = score + 10
print(score)  # 10

score -= 3    # score = score - 3
print(score)  # 7
```

To tworzy NOWY obiekt, a nie zmienia stary

Jak dowiedzieliśmy się w rozdziałach 3 i 4, liczby w Python są niezmienne. `score += 10` nie „dodaje" `0` — Python oblicza `score + 10`, tworzy nowy obiekt `10` i ponownie wiąże nazwę `score` do tego. Na zewnątrz wygląda to na zmianę, ale w środku to dokładnie ten sam mechanizm, który widzieliśmy w rozdziale 4 `age = age + 1`.

Takie skróty istnieją dla wszystkich głównych operatorów:

Skrócone nagranie	Pełne nagranie
`x += n`	`x = x + n`
`x -= n`	`x = x - n`
`x *= n`	`x = x * n`
`x /= n`	`x = x / n`
`x //= n`	`x = x // n`
`x %= n`	`x = x % n`
`x **= n`	`x = x ** n`

Co jest pierwsze?

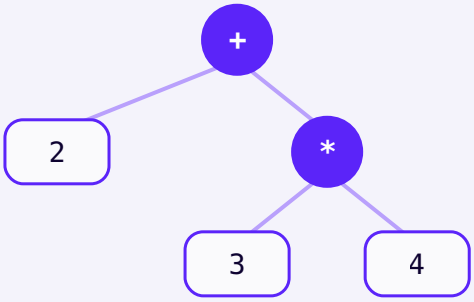
Już widzieliśmy kawałki tego obrazu osobno — teraz zbierzemy pełną mapę priorytetu operatorów rozdziału 5, od najwyższego do najniższego:

`( )`	nawiasy zawsze są pierwsze
`**`	potęgowanie
`+x -x`	operator jednoargumentowy plus/minus
`* / // %`	mnożenie, dzielenie, dzielenie całkowite, reszta
`+ -`	dodatku, odejmowanie

Pełna mapa priorytetu operatorów w tym rozdziale — od najwyższego do najniższego

poryadok.py

```
print(2 + 3 * 4)     #14 - Najpierw 3 * 4 = 12, potem + 2
print((2 + 3) * 4)   # 20 — nawiasy zmieniają kolejność
```



2 + 3 * 4 = 14 – mnożenie jest wykonywane głębiej w drzewie, czyli wcześniej

Drzewo pokazuje to samo co priorytet: `*` zainwestowany głębszy, więc najpierw oblicza się, a jego wynik staje się jednym z operandów `+`.

W razie wątpliwości używaj nawiasów

Nawet gdy kolejność jest formalnie jasna, nawiasy sprawiają, że kod jest bardziej uczciwy dla osoby, która go będzie czytać (w tym dla ciebie za sześć miesięcy). `(a + b) * c` czyta się szybciej niż trzeba przypominać sobie zasady priorytetu.

Praktyka: Przypisanie i kolejność obliczeń

Ten sam laptop co w sekcji Degrees and Roots – porusza ten temat również

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/05-02/index.html)

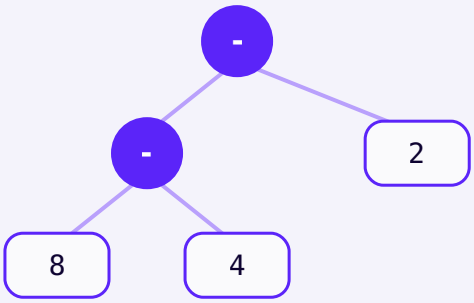
Skojarzenie operatorów

Priority pokazuje, który operator jest pierwszy. Asocjowalność odnosi się do kolejności, w jakiej kilka operatorów o tym samym priorytecie powinno być liczonych w rzędzie.

Priorytet odpowiada na pytanie „który operator jest pierwszy” **asocjowalność** — na pytanie „w jakiej kolejności liczyć kilku operatorów TEGO samego priorytetu po kolei”.

Większość operatorów jest od lewej do prawej

`8 - 4 - 2` jest uważane za `(8 - 4) - 2`, to znaczy, od lewej do prawej. Nazywa się to **lewicowa asocjowalność**, i tak się zachowują Prawie wszyscy operatorzy Python: `+ - * / // %`.



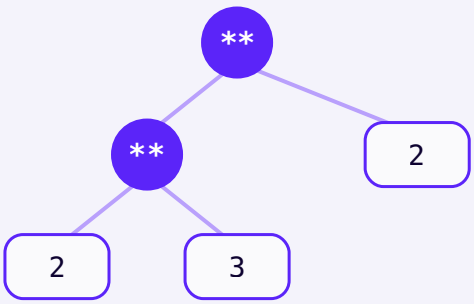
8 - 4 - 2 — lewostronnie asocjacyjne: najpierw lewa para (8 - 4), potem wynik minus 2

levaya.py

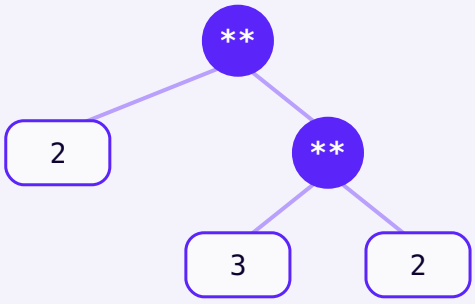
```
print(8 - 4 - 2)     #2, bo to (8 – 4) – 2 = 4 – 2
print((8 - 4) - 2)   #2 – Wyraźnie to samo
print(8 - (4 - 2))   # 6 – a oto inna liczba!
```

Wyjątek: `**` — od prawej do lewej

Potęgowanie — jedyny operator w tym rozdziale z **prawą** asocjowalności. Porównajmy, ile każda opcja dałaby `2 ** 3 ** 2`:



Gdyby ** było lewostronnie asocjacyjne: (2 ** 3) ** 2 = 64 – ale to NIE jest to, co Python robi



Właściwie ** asocjacyjne: 2 ** (3 ** 2) = 512

stepen_associativnost.py

```
print(2 ** 3 ** 2)      # 512 — Python liczy jak 2 ** (3 ** 2)
print((2 ** 3) ** 2)    # 64 — wariant lewostronny dałby inną liczbę
print(2 ** (3 ** 2))    #512 — Odpowiada rzeczywistemu zachowaniu Python
```

```
2 ** 3 ** 2
      ↓
2 ** (3 ** 2)
      ↓
  2 ** 9
      ↓
```

512

Prawo-asocjowalność: najpierw liczy się wykładnik GÓRNY (prawy)

Po co ** w ogóle to zrobiono

Prawa asocjowalność stopnia pokrywa się z matematyczną tradycją „wieży stopni” *tetration*): w matematyce a^{b^c} zawsze odczytywany jest jako $a^{(b^c)}$, a nie odwrotnie. Python tutaj po prostu podąża za długo utrzymywaną konwencją matematyczną.

Praktyka: Operator Asocjowalność

interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/05-10/index.html)

Nawiasy i wzory ze zmiennych

Od pojedynczych operatorów do rzeczywistych formuł o znaczących nazwach, tak jak w prawdziwym kodzie.

Widzieliśmy już, że nawiasy zmieniają kolejność obliczeń. Ale nawiasy mają też drugą rolę: Nie mniej ważne: sprawiają, że formuły są zrozumiałe dla **człowieka**, nawet gdy Priorytet i tak dałby właściwy rezultat.

Nawiasy dla ludzi, a nie tylko dla Python

skobki_dlya_lyudej.py

```python
# Technicznie nawiasy nie są potrzebne — priorytet jest już poprawny:
srednee = 4 + 7 + 9 / 3
print(srednee)
# 14.0 — NIEPRAWIDŁOWO! To 4 + 7 + (9 / 3), nie to, co było zamierzone

# Ale z jawnymi nawiasami wynik odpowiada intencji:
srednee = (4 + 7 + 9) / 3
print(srednee)      # 6,666... — średnia trzech liczb
```

Zapomniane nawiasy są źródłem prawdziwych błędów
Powyższy przykład nie jest wymyśloną pułapką: „zapomnij nawiasów wokół sumy przed podziałem”

Formuły zmiennych nazwanych

Kod rzeczywisty prawie nigdy nie jest pisany „gołymi”

formula_pryamougolnika.py

```python
shirina = 6
vysota = 3

perimetr = 2 * (shirina + vysota)
ploshad = shirina * vysota

print(f"Obwod: {shirina + shirina}")
print(f"Obwod: {perimetr}")
print(f"Obszar: {ploshad}")
```

Powyższa literówka jest celowa
Zwróć uwagę na pierwszą linijkę `print()` w przykładzie: `shirina + shirina` zamiast `perimetr` — nie wyrzuca błędu, ale daje błędną liczbę. Formuła, która nie zawiesza się, ale liczy niewłaściwie — to najpodstępniejszy błąd w obliczeniach. Takie błędy przeanalizujemy szczegółowo w rozdziale o debugowaniu obliczeń.

Trzy kolejne klasyczne formuły

eshche_formuly.py

```
# prędkość = dystans/czas
rasstoyanie_km = 120
vremya_ch = 2
skorost = rasstoyanie_km / vremya_ch
print(f"Szybkość: {skorost} km/h")

# Przeliczenie temperatury z Celsjusza na Fahrenheita
celsius = 20
fahrenheit = celsius * 9 / 5 + 32
print(f"{celsius}°C = {fahrenheit}°F")

# Proste odsetki: Kwota * Stopa * Lata / 100
summa = 1000
stavka = 5
gody = 3
procenty = summa * stavka * gody / 100
print(f"Procenty: {procenty}")
```

1/3

Średnia trzech równych uderzeń jest taka sama jak dzielenie przez 3 w powyższym wzorze

Praktyka: nawiasy i wzory z zmiennych
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/05-11/index.html)

Tłumaczenie wzorów: matematyka → Python

Podręcznik zapisuje wzory inaczej niż Python. Przeanalizujmy zasady tłumaczenia — oraz najczęstsze błędy w tym tłumaczeniu.

Wzory w podręcznikach matematyki są zapisywane inaczej niż kod Python – mnożenie jest często stosowane ogólnie nie pisz, stopień jest podniesiony ponad linię, dzielenie jest rysowane linią. Przetłumacz takie wzory na Python trzeba być ostrożnym.

Tabela Tłumaczeń

W matematyce	W Python	Częsty błąd
2x	`2 * x`	pisać `2x` jest SyntaxError, mnożenie nigdy nie jest implikowane
x2	`x ** 2`	pisać `x^2` — zadziała, ale da zupełnie inną liczbę!
a+bc	`(a + b) / c`	pisać `a + b / c` – dzieli się tylko b, nie całość
x	`x ** 0.5` lub `math.sqrt(x)`	zapomnieć, że pierwiastek jest również stopniem (0,5)
πr2	`math.pi * radius ** 2`	zapomnieć zaimportować `math`, albo pisać `pi` zamiast `math.pi`
a(b+c)	`a * (b + c)`	pisać `a(b + c)` — Python zdecyduje, że `a` to funkcja i spróbuje ją nazwać
a+bc+d	`(a + b) / (c + d)`	pisać `a + b / c + d` – tylko b zostanie podzielone na c

^ w Python NIE jest stopniem naukowym

W matematyce `^` czasem oznacza podniesienie stopnia do pewnego stopnia. W Python `^` jest zupełnie innym operatorem (bitowo wyłączny OR, przeanalizujemy go w rozdziale o bitach): `5 ^ 3` równa się `6`, nie `125`. Do potęgowania zawsze używaj `**`.

proverka_karety.py

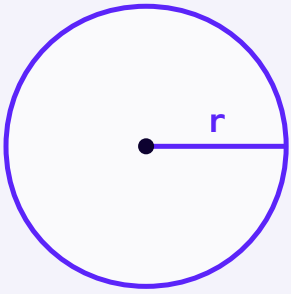
```
print(5 ** 3)   #125 — Stopień
print(5 ^ 3)    #6 to zupełnie inna operacja!
```

Od matematyki przez geometrię do Python

Tabela pokazuje przesunięcie linia po linijce. Ale wzór A=πr2 istnieje trzeci, najbardziej ilustracyjny poziom — sama figura geometryczna, którą formuła opisuje:

$$A=\pi r2$$

A = πr² to pole okręgu przebiegającego przez jego promień



Promień r łączy środek koła z jego krawędzią, z której oblicza się pole A = πr²

ploshad_kruga_perevod.py

```python
import math

radius = 5
ploshad = math.pi * radius ** 2
print(round(ploshad, 2))   # 78.54
```

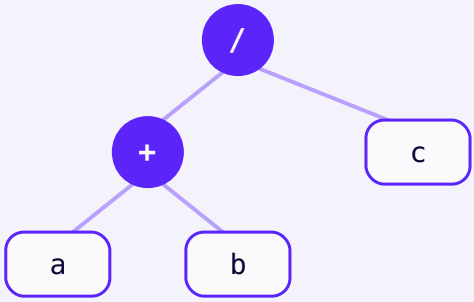
radius ** 2, nie (radius) ** 2
Nawiasy wokół pojedynczej zmiennej nie są potrzebne — `**` jest już stosowane tylko do `radius`, i nie na cały wyraz twarzy `math.pi * radius` (mnożenie i moc to różne poziomy priorytetu, moc jest wyższa).

Dlaczego nawiasy wokół sumy są obowiązkowe

Wracamy do ułamka a+bc spojrzeć na to z trzech stron: jako formułę, jako drzewo obliczeniowe i jako kod.

a+bc

Ryza ułamka GRUPUJE a + b w liczniku — jest to wizualnie oczywiste



(a + b) / c — drzewo pokazuje, że cała suma musi znajdować się PONIŻEJ linii ułamków

Drzewo pokazuje to samo co kreska ułamka: `a + b` — jednolity blok, który w całości staje się licznikiem. W Python kreski nie ma, więc tę grupę trzeba wyraźnie oznaczyć — za pomocą nawiasów okrągłych:

pochemu_skobki.py

```
a, b, c = 8, 4, 3

# POPRAWNIE — nawiasy odtwarzają linię ułamku:
print((a + b) / c)   # 4.0

# NIEPRAWIDŁOWO — Tylko b: jest dzielone bez nawiasów
print(a + b / c)     # 9.333... to a + (b / c), zupełnie inna formuła!
```

Przykład tłumaczenia: wzór na średnią

Wzór na średnią trzech liczb $a+b+c3$ jest przeliczany na Python według tej samej zasady — cała kwota musi być podzielona w nawiasach przed podziałem:

srednee_perevod.py

```
a, b, c = 4, 7, 9
srednee = (a + b + c) / 3
print(srednee)   # 6.666...
```

Praktyka: Tłumaczenie wzorów z matematyki na Python
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/05-12/index.html)

Moduł math – Od katalogu do narzędzi

Czym jest moduł, dlaczego potrzebujesz import i co oznacza punkt w math.sqrt() — zanim przejdziemy do konkretnych funkcji, zastanówmy się, jak działa ten zestaw narzędzi ogólnie.

Python już potrafi liczyć coś bez żadnego przygotowania:

vstroennoe.py

```
print(abs(-7))     # 7
print(round(4.5))  # 4
print(pow(2, 10))  # 1024
```

Ale na świecie jest znacznie więcej prawdziwej matematyki niż cztery wbudowane funkcje: pierwiastki, sinusy, logarytmy, praca z kątami i odległościami. Gdyby tylko **wszystko** takie narzędzia byłyby wbudowane bezpośrednio w język — Python stałby się zagraconym lista z setkami nazw, i byłoby niemożliwe zapamiętać, co jest w ogóle dostępne.

Moduł to osobna, podpisana skrzynka narzędziowa

Zamiast tego standardowa biblioteka Python organizuje powiązane narzędzia w **moduły** — oddzielne „pudełka", które leżą obok siebie, ale nie mieszają się bezpośrednio z językiem ani między sobą. Jedno z takich pudełek nazywa się `math` — Zawiera narzędzia matematyczne: pierwiastki, trygonometrię, logarytmy i wiele innych.

Python: to, co jest wbudowane, i to, co znajduje się w modułach

WBUDOWANE W JĘZYK — DOSTĘPNE ZAWSZE

`print()` · `abs()`
`round()` · `pow()`
nie wymaga import

MODUŁ MATH - ODDZIELNA SKRZYNKA NARZĘDZIOWA

`sqrt()` · `hypot()` · `gcd()`

`sin()` · `log()` · `pi`

wymaga `import math`

Moduł to osobny, wcześniej podpisany zestaw narzędzi, niebędący częścią samego języka

`import math` — otwieramy szufladę

Używać narzędzi z szuflady `math`, musi tak być Na pierwszy rzut oka jest to oczywiste **otwórz** — drużyna `import`:

import_math.py

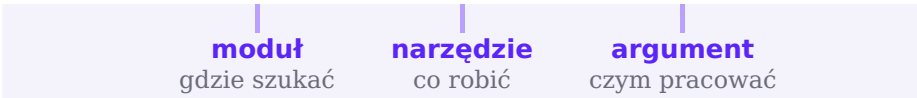
```python
import math

# teraz cała zawartość skrzynki pocztowej math dostępna w tym programie
print(math.sqrt(2))
```

import math nie chodzi o „włączenie matematyki”

String `import math` mniej więcej oznacza: „udostępnij zawartość modułu math w tym programie.” `math.sqrt(25)` spowoduje `NameError` — Python jeszcze nie wie, gdzie szukać `math`.

Co oznacza kropka

Przeanalizujmy nagranie `math.sqrt(25)` części:



Kropka oznacza „weź sqrt Z modułu math”

Kropka tutaj oznacza "weź `sqrt` **z** modułu `math`" jest Python rozróżnia, na przykład, `math.sqrt` z innej funkcji o podobnej nazwie z zupełnie innego modułu: każdy Narzędzie ma swój wyraźny adres.

A co jeśli nie potrzebujesz argumentu: `math.pi`

`math.pi` nie jest funkcją, lecz gotową **znaczenie** (stała): Liczba π została już obliczona i znajduje się wewnątrz modułu pod nazwą `pi`. Dlatego nie ma nawiasów — nawiasy oznaczają „wywołanie funkcji” `pi` nie musisz tego nazywać, wystarczy przeczytać:

math_pi.py

```
import math

print(math.pi)        # 3.141592653589793 – wartość, nie
wywołanie
print(math.sqrt(25))  #5.0 – i to jest wywołanie funkcji, więc
są nawiasy
```

Pierwsze eksperymenty

Kilka małych przykładów to kod i jego wynik:

eksperiment_1.py

```
import math

print(math.sqrt(25))
```

WYNIK: `5.0` to pierwiastek kwadratowy z 25.

eksperiment_2.py

```
import math

print(math.pi)
```

WYNIK: `3.141592653589793` to liczba π z dokładnością float.

eksperiment_3.py

```
import math

print(math.gcd(18, 24))
```

WYNIK: `6` — największy wspólny dzielnik 18 i 24.

eksperiment_4.py

```
import math

print(math.hypot(3, 4))
```

WYNIK: `5.0` to długość przeprostokąta trójkąta prostokątnego o bokach 3 i 4.

Wbudowany vs math – nie myl

Jeszcze raz wyznaczmy wyraźną granicę — które narzędzia są wbudowane bezpośrednio w język, a które muszą importować wprost, a skąd:

Tool	Skąd	Potrzebujesz import?
`abs()`, **`round()`**, **`pow()`**	są wbudowane w język	nie
`math.sqrt()`, **`math.floor()`**, **`math.sin()`**, **`math.log()`**	moduł `math`	`import math`
`Decimal`	moduł `decimal` (rozdział 4)	`from decimal import Decimal`

Tool	Skąd	Potrzebujesz import?
`Fraction`	moduł `fractions` (rozdział 4)	`from fractions import Fraction`

Decimal i Fraction NIE są częścią math
To częste zamieszanie: `Decimal` oraz `Fraction` z rozdziału 4 to narzędzia do dokładnych liczb, ale istnieją we WŁASNYCH modułach (`decimal` oraz `fractions`), osobno od `math` . Trzy różne skrzynki — trzy różne importy.

Pełna mapa: Co math może zrobić

Teraz, gdy rozumiesz, czym jest moduł i dlaczego potrzebujesz punktu, możesz spokojnie spojrzeć na Mapa czegoś `math` sugeruje, kolejne sekcje rozdziału będą obejmować Każda karta szczegółowo, z rzeczywistymi zadaniami:

Co potrafi moduł math

KORZENIE I ODLEGŁOŚCI

`sqrt · isqrt`
`hypot · dist`

MATEMATYKA CAŁKOWITA

`gcd · lcm`
`factorial · comb · perm`

GEOMETRIA

`pi`
`hypot · dist`

TRYGONOMETRIA

`sin · cos · tan`

`radians · degrees`

LOGARYTMY I WZROST

`log · log2 · log10 · exp`

Każda karta to osobna sekcja poniżej, z prawdziwymi problemami i zdjęciami

Jak szukać czegoś, co nie znajduje się na mapie

U `math` około pięćdziesięciu funkcji – ta mapa pokazuje tylko Najbardziej przydatne na początek. Nikt nie pamięta o całej lista i to jest w porządku: Profesjonalna umiejętność to nie „znać wszystko na pamięć” **mniej więcej**to i szybko znaleźć dokładną część.

Oficjalna dokumentacja nie jest wrogiem, lecz narzędziem

Kompletny i dokładny lista wszystko w `math` , na oficjalnej stronie dokumentacji Python 3.14: docs.python.org/3.14/library/math (Standard Library → math). Doświadczeni programiści często tam patrzą — to nie jest oznaka ignorancji, lecz normalna część pracy.

Praktyka: moduł math — czym jest moduł i pierwsze wywołania

interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

`Otwórz praktykę →` (https://www.cartesianschool.org/pl/practice/05-04/index.html)

Korzenie i odległości

Od twierdzenia pitagorejskiego ręcznie do gotowych funkcji, które obliczają dla ciebie odległość.

`isqrt()` — pierwiastek całkowity

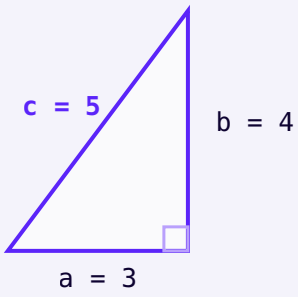
`math.sqrt()` zawsze wraca `float`. Jeśli potrzebujesz `int` — na przykład dla tak dużych liczb, To, co float zaczyna tracić na celności, jest tam `math.isqrt()`: on odrzuca część ułamkową i pracuje tylko z liczbami całkowitymi.

isqrt.py

```
import math

print(math.sqrt(10))     # 3.1622776601683795
print(math.isqrt(10))
#3 — cała część korzenia, brak ułamków i brak utraty precyzji
```

`hypot()` – przeciwprostokąt bez ręcznego Pitagorasa

Jeśli znane są długości dwóch nóg trójkąta prostokątnego, można znaleźć długość przepustki Formuła `c = sqrt(a**2 + b**2)` — ale `math` już wie, jak to zrobić za pomocą jednej funkcji:



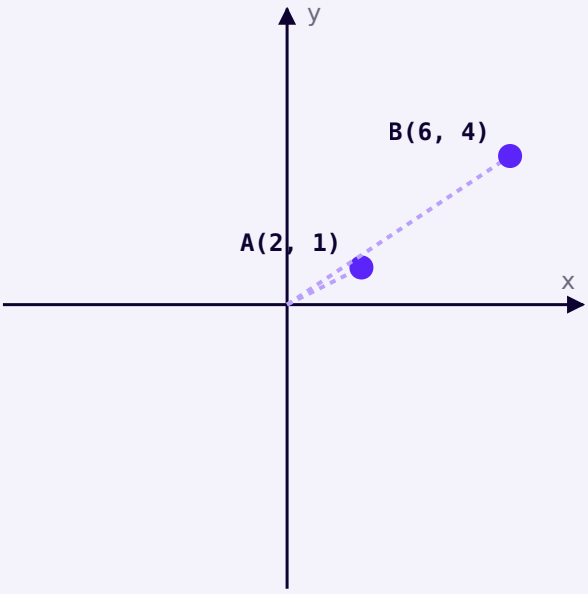
hypot(3, 4) = 5 – klasyczny trójkąt prostokątny 3-4-5

hypot.py

```
import math

print(math.sqrt(3 ** 2 + 4 ** 2))   #5.0 — Podręcznik
Twierdzenia Pitagorasa
print(math.hypot(3, 4))             # 5.0 — to samo, krócej i
precyzyjniej
```

`dist()` to odległość między dwoma punktami

`math.dist()` jest `hypot()`, ale dla dwóch punktów na płaszczyźnie (lub w przestrzeni), a nie dla jednego trójkąta: sama oblicza przyprostokątne jako różnicę współrzędnych.



Odległość między A i B to przeciwprostokątna trójkąta prostokątnego, zbudowanego na ich współrzędnych

dist.py

```
import math

A = (2, 1)
B = (6, 4)
print(math.dist(A, B))   #5.0 — Ta sama odległość obliczona na
podstawie współrzędnych
```

Skąd wziął się ten numer 5.0

Różnica x: 6 − 2 = 4. Różnica y: 4 − 1 = 3. To dokładnie nogi trójkąta 3-4-5 powyżej — `dist()` pod maską robi dokładnie to samo, co `hypot()`, sam tylko liczy przyprostokątne za ciebie.

Praktyka: Korzenie i odległości

interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/05-13/index.html)

gcd, lcm, czynnikowy, comb, perm

Pięć funkcji do pracy z liczbami całkowitymi i liczenia wariantów – z rzeczywistymi problemami dla każdej z nich.

`gcd()` — największy wspólny dzielnik

Największy wspólny dzielnik (GCF) dwóch liczb jest największą liczbą, przez którą obie są podzielne bez reszty. Klasyczne zastosowanie – redukcja ułamków:

gcd.py

```python
import math

print(math.gcd(12, 18))   # 6 — i 12, i 18 dzielą się przez 6 bez reszty

# zredukować ułamek 12/18:
chislitel, znamenatel = 12, 18
nod = math.gcd(chislitel, znamenatel)
print(f"{chislitel // nod}/{znamenatel // nod}")   #2/3 to ten sam ułamek, ale w najprostszej formie
```

`lcm()` — najmniejsza wspólna wielokrotność

Najniższy wspólny mnożnik (LCM) to najmniejsza liczba podzielna przez obie liczby jednocześnie. Przyda się na przykład zrozumienie, kiedy dwa powtarzające się zdarzenia znów się pokrywają:

lcm.py

```python
import math

print(math.lcm(4, 6))
#12 — Autobus nr 1 kursuje co 4 minuty, autobus nr 2 co 6 minut
                        # obaj będą na przystanku o tej samej porze co 12 minut
```

`factorial()` — liczyć permutacje

Factorial `n!` jest iloczynem wszystkich liczb całkowitych od 1 do `n`. Odpowiada na pytanie „na ile sposobów można to zorganizować n różne obiekty w kolejności”

factorial.py

```python
import math

print(math.factorial(5))   #120 — 5 książek na półce można ułożyć na 120 różnych sposobów
```

`comb()` oraz `perm()` – Wybór i porządek

`comb(n, k)` — ile sposobów wyboru `k` przedmioty z `n` jeśli kolejność wyboru **nie jest ważne**. `perm(n, k)` jest takie samo, ale gdy kolejność **ważne**.

comb_perm.py

```
import math

# ile różnych trójek liczb można wybrać z 49 w loterii,
kolejność nie ma znaczenia:
print(math.comb(49, 3))   # 18424

# ile różnych podium (1., 2., 3.) może mieć 10 zawodników,
kolejność jest ważna:
print(math.perm(10, 3))   # 720
```

perm() zawsze większy lub równy comb()

Za to samo `n` oraz `k` `perm()` liczy KAŻDE zamówienie osobno, oraz `comb()` są tylko unikalnymi zestawami. Jeśli nie jesteś pewien, którą funkcję użyć, zadaj sobie pytanie: „Czy kolejność jest ważna?” `perm()` nie ma → `comb()` .

Praktyka: gcd, lcm, silnia, comb, perm

interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

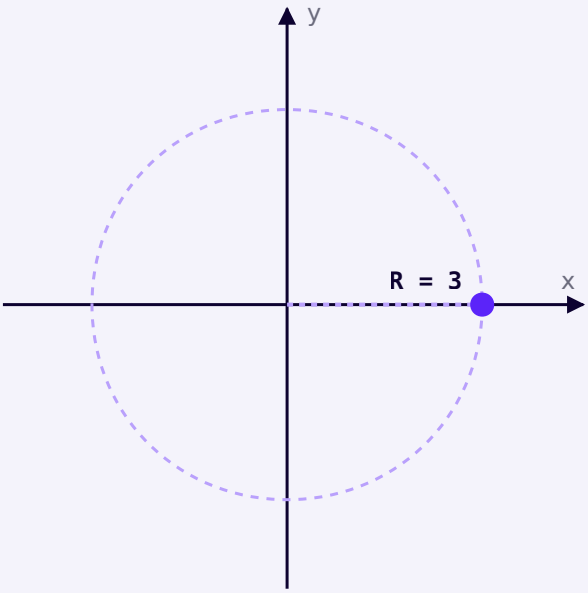
`Otwórz praktykę →` (https://www.cartesianschool.org/pl/practice/05-14/index.html)

Geometria z math

math nie rysuje kółek — ale daje wszystko, by je dokładnie liczyć.

Moduł `math` nie umie rysować kółek, ale daje wszystko, czego potrzebujesz ich **policz**: stały `math.pi` i zwykłe operatory arytmetyczne, które już znamy.

Obwód i obwód



Koło o promieniu 3 — powierzchnia i obwód liczone przez jego promień

krug.py

```
import math

radius = 3
ploshad = math.pi * radius ** 2
dlina_okruzhnosti = 2 * math.pi * radius

print(f"Obszar: {round(ploshad, 2)}")               # 28.27
print(f"Obwód: {round(dlina_okruzhnosti, 2)}")  # 18.85
```

math.pi jest float z ograniczoną dokładnością

`math.pi` przechowuje π z dokładnością wystarczającą do praktycznych zadań (15-16 cyfr znaczących) — ale nie jest to „prawdziwe" nieskończone π, a jego `float` – reprezentacja. W rozdziale 4 szczegółowo omówiliśmy, dlaczego float ograniczenia dokładności.

Połącz kilka wzorów: powierzchnia pierścienia

Prawdziwe zadania geometryczne często wymagają łączenia kilku wzorów. Powierzchnia pierścienia między dwoma okręgami — powierzchnia dużego koła minus powierzchnia małego:

kolco.py

```
import math

vneshnij_radius = 5
vnutrennij_radius = 3

ploshad_kolca = math.pi * vneshnij_radius ** 2 - math.pi *
vnutrennij_radius ** 2
print(round(ploshad_kolca, 2))   # 50.27
```

Przydatni pomocnicy: `prod()` oraz `fsum()`

Do mnożenia lub dokładnego sumowania kilku liczb jednocześnie, takich jak objętość równoległostniki w trzech wymiarach:

prod_fsum.py

```
import math

izmereniya = [4, 5, 6]
obyom = math.prod(izmereniya)
print(obyom)   # 120 — jak 4 * 5 * 6

#fsum dokładniej float niż zwykłe sum() — mniejszy skumulowany
błąd zaokrąglenia
chisla = [1, 1e16, -1e16]
print(sum(chisla))
#0.0 — Normalna sum() straciła jeden z powodu kolejności
dodawania
print(math.fsum(chisla))   #1.0 — fsum() uważa, że jest
bardziej trafny w takich przypadkach
```

Praktyka: geometria z math
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/05-15/index.html)

Trygonometria bez strachu

sin i cos to po prostu współrzędne punktu na okręgu o promieniu 1. Cała trygonometria opiera się na tym jednym obrazku.

Trygonometria przeraża wielu ludzi nie samą matematyką, lecz notacją. W rzeczywistości idea jest prosta: Dla dowolnego punktu na okręgu o promieniu 1 (okrąg jednostkowym), współrzędne tego punktu są I jest `cos` oraz `sin` róg.

Stopnie i radiany

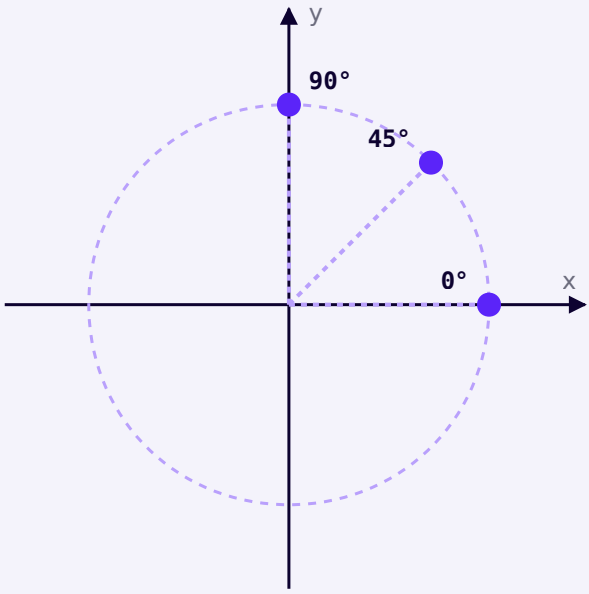
Python (jak prawie cała matematyka poza szkolną geometrią) liczy kąty w **radiany**, nie w stopniach. Pełny obrót to 360° lub, co jest tym samym, `2π` radianów:

gradusy_radiany.py

```
import math

print(math.radians(180))    # 3.141592653589793 — czyli π
print(math.degrees(math.pi)) # 180.0
```

math.sin(90) NIE jest tym, co myślisz
`math.sin(90)` liczy sinus kąta na poziomie 90 **radiany** (to ogromny kąt, wiele pełnych obrotów), nie 90 stopni! Przed funkcjami trygonometrycznymi prawie zawsze potrzebujesz `math.radians()`: `math.sin(math.radians(90))`.

Unit Circle



Jednostkowy okrąg (promień 1) — Współrzędne punktu są równe (cos kąta, sin kąta)

edinichnaya_okruzhnost.py

```python
import math

ugol_gradusy = 30
ugol_radiany = math.radians(ugol_gradusy)

print(round(math.cos(ugol_radiany), 3))
#0,866 to współrzędna x punktu na okręgu
print(round(math.sin(ugol_radiany), 3))
# 0.5   — współrzędna y punktu na okręgu
```

round() tutaj nie jest przypadkiem
Bez `round()` `math.sin(math.radians(30))` wydaje `0.49999999999999994`, nie równomiernie `0.5` — ta sama cecha float, którą omawialiśmy w rozdziale 4. Do porównania kątów również warto użyć tolerancji (na przykład, `math.isclose()`), nie `==`.

`tan()` i funkcje odwrotne

tan_obratnye.py

```python
import math

print(round(math.tan(math.radians(45)), 3))   # 1.0

# problem odwrotny: znane tan, szukanie kąta
ugol = math.degrees(math.atan(1))
print(ugol)   # 45.0
```

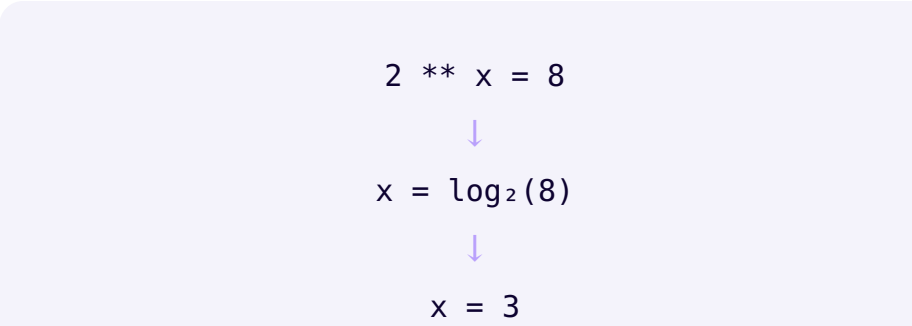
Praktyka: Trygonometria bez strachu
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/05-16/index.html)

Logarytmy i wykładniki

Logarytm jest po prostu odwrotnością działania mocy: nie "wynikiem konstrukcji", lecz "jaką mocą potrzebujemy".

Logarytm odpowiada na pytanie odwrotne do stopnia: nie "2 w jakim stopniu daje 8", lecz "w jakim stopniu Musisz podnieść stopień 2, żeby dostać 8."



Logarytm jest odwrotnością operacji potęgowej: szukamy wskaźnika, a nie wyniku

log.py

```
import math

print(2 ** 3)          # 8 — zadanie bezpośrednie: stopień znany, szukamy wyniku
print(math.log2(8))    # 3.0 — zadanie odwrotne: wynik jest znany, szukamy stopnia
```

Trzy warianty podstawy

Funkcja	Fundacja	Stosowane
`math.log2(x)`	2	informatyka: ile bitów potrzeba dla x wartości
`math.log10(x)`	10	rząd wielkości liczby (ile ma cyfr)
`math.log(x)`	e ≈ 2,718 (naturalny)	wzrost/upadek, finanse, nauka

Funkcja	Fundacja	Stosowane
`math.log(x, base)`	żadnym, jest dane przez drugi argument	dowolnej podstawie

osnovaniya.py

```
import math

print(math.log10(1000))   #3.0 — Numer 1000 ma trzy zera
print(math.log(8, 2))     #3.0 jest tym samym co log2(8), ale
poprzez wspólną funkcję
```

`exp()` jest operacją odwrotną do logarytmu

`math.exp(x)` liczy `e ** x` — że Jaki logarytm i moc przechodzą ze sobą:

exp.py

```
import math

x = 2
print(math.exp(x))          # 7.38905609893065
print(math.log(math.exp(x)))  # 2.0 — log i exp znoszą się
nawzajem
```

Gdzie logarytmy spotykają się ponownie

Logarytmy nie będą potrzebne od razu, ale spotkamy je jeszcze nie raz: w ocenie złożoności algorytmów (dlaczego wyszukiwanie w posortowanej liście jest szybkie), w analizie danych, w skalach takich jak decybele czy trzęsienia ziemi według skali Richtera. Na razie wystarczy zrozumieć samą ideę — „operacja odwrotna do potęgi”.

Praktyka: logarytmy i wykładniki

interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/05-17/index.html)

Moduł pseudolosowości random:

Trudno sobie wyobrazić jakąkolwiek grę bez odrobiny przypadkowości — ale ta „przypadkowość” jest zorganizowana o wiele ciekawiej, niż się wydaje.

Moduł `random` dodaje do Python element losowości — bez niego prawie żadna gra się nie obejdzie: losowy przeciwnik, losowa mapa, losowa liczba dla gry „Zgadnij liczbę” w rozdziale 9.

„losowa”

Komputer nie może stworzyć prawdziwego wypadku z niczego— zamiast tego, `random` zastosowania **generator pseudolosowy**: wzór obliczający następną liczbę z jednej liczby („stan”

psevdosluchajnost.py

```
import random

print(random.random())   # losowa liczba ułamkowa między 0,0 a
1,0 (nie licząc 1,0)
print(random.random())   # inny numer — status generatora
został zaktualizowany
```

Po co to wiedzieć, skoro możesz po prostu tego używać?
Ponieważ prowadzi to do ważnej praktycznej właściwości: generator może być **start od restartu** z tego samego stanu, a następnie ponownie generuje ten sam ciąg liczb. Nazywa się to powtarzalnością i szczegółowo rozłożymy to na kilka sekcji — jest niezbędna przy debugowaniu programów losowych.

Co dalej w tej sekcji

RANDINT · RANDRANGE · UNIFORM

losowe liczby w zakresie

CHOICE · CHOICES · SAMPLE · SHUFFLE

losowy wybór ze zbioru

SEED

powtarzalność i random vs secrets

Każda karta znajduje się osobną sekcją poniżej

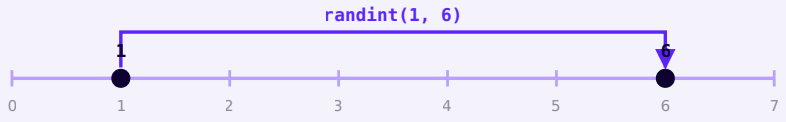
Praktyka: podstawy random
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/05-05/index.html)

randint, randrange, uniform

Trzy podobne funkcje z trzema różnymi zasadami dotyczącymi granic zasięgu — i jednym klasycznym błędem, który pojawia się, gdy je pomieszasz.

`randint(a, b)` jest losową całością, obie granice są uwzględnione



randint(1, 6) może zwrócić 1, może zwrócić 6 — obie granice są wliczone

randint.py

```
import random

kubik = random.randint(1, 6)   # może być 1, 2, 3, 4, 5 LUB 6
print(kubik)
```

Powszechny błąd „na jeden”
Jeśli potrzebujesz liczb od 1 do 5 (nie licząc 6), ale zapisz `random.randint(1, 6)` — program czasem źle zwraca 6. Obie granice `randint()` są włączone — warto o tym pamiętać za każdym razem.

`randrange(start, stop, step)` — jak `range()`, ale przypadkowo

W przeciwieństwie do `randint()`, u `randrange()` górna granica **nie uwzględnione** — całkiem jak w zwykłym `range()`, który szczegółowo przeanalizujemy w rozdziale 10. Dodatkowo jest opcjonalny krok:

randrange.py

```
import random

print(random.randrange(1, 7))       # od 1 do 6 włącznie – 7
NIE jest uwzględnione, analogowe randint(1, 6)
print(random.randrange(0, 10, 2))   # Losowa liczba parzysta:
0, 2, 4, 6 lub 8
```

Funkcja	Górna granica	Krok
`randint(a, b)`	włączona	nie, tylko sąsiednie całości
`randrange(a, b)`	NIE UWZGLĘDNIONE	nie (domyślnie 1)

Funkcja	Górna granica	Krok
`randrange(a, b, step)`	NIE UWZGLĘDNIONE	tak — można przeskakiwać

`uniform(a, b)` jest liczbą ułamkową losową

Działa jako `randint()`, ale dla `float` jest dowolną wartością ułamkową w zakresie, nie tylko liczbami całkowitym:

uniform.py

```
import random

temperatura = random.uniform(18.0, 24.0)
print(round(temperatura, 1))   # np. 21.3
```

Praktyka: randint, randrange, uniform
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji
Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/05-18/index.html)

choice, choices, sample, shuffle

Cztery podobne narzędzia do losowania różnią się liczbą wybranych przedmiotów i możliwością ich powtórzenia.

Poza okazjonalnymi *liczby*, `random` wie, jak wybierać losowo of **gotowy zestaw** wartości — lista opcji, talia kart, lista imion.

Cztery podobne, ale różne narzędzia

Funkcja	Ilu wybiera	Czy powtórzenia są możliwe?	zmienia oryginalny zestaw?
`choice(seq)`	1 przedmiot	—	nie
`choices(seq, k=n)`	n elementów	tak – ten sam element można powtórzyć	nie

Funkcja	Ilu wybiera	Czy powtórzenia są możliwe?	zmienia oryginalny zestaw?
`sample(seq, k=n)`	n elementów	nie – każdy jest inny	nie
`shuffle(seq)`	wszystkie elementy	—	tak, miesza na miejscu

`choice()` jest jednym elementem losowym

choice.py

```
import random

varianty = [„kamień”, „nożyczki”, „papier”]
print(random.choice(varianty))
```

`choices()` — kilka, z możliwymi powtórzeniami

Wyobraźmy sobie ruletkę z nagrodami – ta sama nagroda może wypaść ponownie:

choices.py

```
import random

prizy = ["  ", "  ", "  "]
vypavshie = random.choices(prizy, k=5)
print(vypavshie)   # na przykład, ["  ", "  ", "  ", "  ",
"  "] — powtórzenia są dozwolone
```

choices() mają opcjonalne obciążniki

`random.choices(prizy, weights=[1, 1, 10], k=5)` sprawią, że trzecia nagroda jest dziesięć razy bardziej prawdopodobna niż pozostałe, co jest przydatne przy „nieuczciwych”

`sample()` — kilka, bez powtórzeń

Wyobraźmy sobie losowanie nagród między uczestnikami – ten sam uczestnik nie może wygrać Dwa razy w tym samym losowaniu:

sample.py

```
import random

uchastniki = [„Ania”, „Borys”, „Faith”, „Gleb”, „Dane”]
pobediteli = random.sample(uchastniki, k=2)
print(pobediteli)   # na przykład, ["Vera", "Anya"] są różne
```

k nie może być dłuższy niż długość zbioru
`random.sample(uchastniki, k=10)` z 5 uczestnikami powoduje `ValueError` — bez powtórzeń nie można wybrać więcej elementów niż jest w zestawie. U `choices()` nie ma takiego ograniczenia, ponieważ powtórzenia są dozwolone.

`shuffle()` to mieszać wszystko na bieżąco

W przeciwieństwie do trzech powyższych funkcji, `shuffle()` nie wybiera Pod zbiór – to się porusza **wszystkim** lista i nic nie zwraca (`None`), ponieważ zmienia lista bezpośrednio „na miejscu”:

shuffle.py

```
import random

koloda = [1, 2, 3, 4, 5]
random.shuffle(koloda)
print(koloda)   # na przykład, [3, 1, 5, 2, 4] to ten sam
lista, nowy porządek
```

shuffle() zwraca None – nie przypisuj wyniku ponownie
`koloda = random.shuffle(koloda)` — częsty błąd: po niej `koloda` stanie się `None` , chociaż mieszanie już się powiodło PRZED przypisaniem. Po prostu wywołaj `random.shuffle(koloda)` oddzielnym wierszem.

Praktyka: choice, choices, sample, shuffle
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji
Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/05-19/index.html)

seed i powtarzalność

Pseudolosowość może zostać „zamrożona”

`seed()` — ustaw stan początkowy generatora

Ponieważ generator pseudolosowy jest formułą o stanie, to przez określenie tego samego początkowego stan („ziarnista”

seed.py

```
import random

random.seed(42)
print(random.randint(1, 100))   # zawsze jest tą samą liczbą,
gdy seed(42)

random.seed(42)                 # zresetuj stan
print(random.randint(1, 100))   # ta sama liczba co za
pierwszym razem
```

Dlaczego jest to potrzebne w praktyce?

Powtarzalność jest kluczowa podczas debugowania: jeśli błąd pojawia się tylko „czasami” `random.seed()` pozwala przechwycić konkretną „losową”

Niezależne generatory: `random.Random()`

Wszystkie funkcje modułu `random` pracowaliśmy z Jeden powszechny, „globalny”

random_instance.py

```
import random

generator_a = random.Random(1)
generator_b = random.Random(1)

print(generator_a.randint(1, 100))   # oba generatory są
tworzone z tą samą seed —
print(generator_b.randint(1, 100))
# oznacza, że uzyskają ten sam wynik niezależnie od siebie
```

random nie jest bezpieczny dla haseł i kluczy

razy `random` — **przewidywalny** generator (ten sam seed → tej samej sekwencji) nie może być używana tam, gdzie losowość chroni przed hakowanie: hasła, tokeny sesji, klucze szyfrujące. Aby to zrobić, standardowa biblioteka Python Jest osobny moduł — `secrets`, zaprojektowany dokładnie dla bezpieczeństwa, a nie dla szybkości.

random czy secrets?

Czy potrzebujesz losowości w grze, symulacji, przypadku testowym?	→ **zastosowanie** `random`
Potrzebna losowość dla hasła, tokena, klucza bezpieczeństwa?	→ **zastosowanie** `secrets`

random zoptymalizowany pod kątem szybkości i przewidywalności jednocześnie seed – NIE nadaje się do ochrony danych

secrets_primer.py

```python
import secrets

token = secrets.token_hex(16)   # kryptograficznie bezpieczny losowy token
print(token)
```

Nie będziemy secrets szczegółowo analizować w tym rozdziale

Wystarczy pamiętać o samej zasadzie: `random` — do gier, symulacji i zadań edukacyjnych; `secrets` — tam, gdzie przypadek chroni coś cennego. O bezpieczeństwie porozmawiamy dokładniej w późniejszych rozdziałach.

Praktyka: seed i powtarzalność
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/05-20/index.html)

Obliczania debugowania

Niepoprawna liczba bez żadnego błędu — najbardziej podstępny typ błędu. Omówimy trzy przyczyny i niezawodny sposób ich wyszukiwania.

Wzór z błędną liczbą jest znacznie bardziej podstępny niż błąd, który odpada Program: Python nie powie ci, że coś jest nie tak. Spójrzmy, jak szukać takich problemów świadomie, a nie metodą „zmieniałem się i patrzył”

Trzy różne typy problemów — i trzy różne sposoby ich wyszukiwania

1. Błąd składniowy

Najprostszy przypadek to taki, że program w ogóle się nie uruchamia, a Python wskazuje na konkretny Nić i charakter jeszcze przed rozpoczęciem egzekucji:

sintaksicheskaya.py

```
# itog = (a + b * c
#SyntaxError:'(' was never closed — Brakuje nawiasów końcowych
```

Liczenie nawiasów w parach
Najpewniejszy sposób znalezienia niezakończonego nawiasu — policzyć otwierające i zamykające nawiasy w podejrzanym łańcuch znaków osobno; powinno ich być tyle samo.

2. Błąd w wzorze

Najbardziej niebezpieczną kategorią jest sytuacja, gdy program działa bez żadnego błędu, ale wynik jest nieprawidłowy. Zazwyczaj istnieje jeden z trzech powodów, które już omówiliśmy w tym rozdziale:

Przyczyna	Przykład	Gdzie to już widzieliśmy
zapomnianych nawiasów	`4 + 7 + 9 / 3` zamiast `(4 + 7 + 9) / 3`	sekcja „Nawiasy i formuły"
nie ta zmienna	`print(shirina + shirina)` zamiast `print(perimetr)`	sekcja „Nawiasy i formuły"
Zdezorientowany priorytet	`-2 ** 2` nie jest tym samym co `(-2) ** 2`	sekcja „Operatorzy uniarni”

Sztuczka: Przerysuj wzór przez nazwane kroki

Gdy długa formuła daje podejrzany wynik, najpewniejszy sposób na znalezienie Problem — podziel go na nazwane kroki pośrednie i wpisz każdy:

trassirovka.py

```
# był jednym z łańcuch znaków, nie wiadomo, co poszło nie tak:
itog = (cena - skidka) * kolichestvo + dostavka / kolichestvo

# stał się — możesz zobaczyć każdy wynik pośredni:
cena_so_skidkoj = cena - skidka
print(„Cena po rabacie:”, cena_so_skidkoj)

summa_za_tovar = cena_so_skidkoj * kolichestvo
print(„Ilość za przedmiot:”, summa_za_tovar)

dostavka_na_edinicu = dostavka / kolichestvo
print(„Dostawa na jednostkę:”, dostavka_na_edinicu)

itog = summa_za_tovar + dostavka_na_edinicu
print(„Podsumowanie:”, itog)
```

łącznie = (100 − 10) × 3 + 15 / 3

↓

cena po rabacie = 90

↓

ilość na przedmiot = 270

↓

dostawa na jednostkę = 5.0

↓

łącznie = 275,0

Trace zamienia pojedynczy tajemniczy łańcuch znaków w sekwencję zrozumiałych, weryfikowalnych kroków

To tymczasowy kod, a nie stały
Podział na nazwaną kroki — technika właśnie do **debugowanie**. Gdy wzór działa poprawnie, pośredni `print()` zwykle usuwane, ale czasem należy pozostawić znaczące nazwy zmiennych pośrednich — czynią kod jaśniejszym, nawet jeśli już nie jest debugowany.

3. Cechy reprezentacji liczb

Trzecia kategoria to kod poprawny składniowo, wzór również jest poprawny, ale wynik nadal jest niespodzianki. Zazwyczaj powodem jest to, że już szczegółowo omówiliśmy to w rozdziale 4:

float_lovushka.py

```
print(0.1 + 0.2 == 0.3)
# False! nie jest błędem, lecz cechą float
print(round(0.1 + 0.2, 10) == 0.3)  #True — Porównanie
tolerancji działa poprawnie
```

Zobacz rozdział 4, aby uzyskać więcej szczegółów
Dlaczego tak się dzieje i jak poprawnie float porównać, szczegółowo omówiono w sekcji „Poprawne porównanie float” `==` między float zachowuje się nieoczekiwanie — oznacza to, że sprawa dotyczy reprezentacji liczb, a nie błędu w formule.

Przewiduj przed startem

Najlepszym nawykiem przy debugowaniu jest budowanie go zanim pojawią się błędy: zanim go uruchomisz kod, przewidź wynik. Jeśli przewidywanie nie pokrywa się z rzeczywistością, albo znalazłeś błąd, albo źle zrozumieć kod. Oba przypadki należy analizować bez dalszego uruchamiania kodu.

Podstawowa praktyka
Przewidzieć wynik
Nie uruchamiając kodu, określ wynik: `итог = 2 + 3 * 4 ** 2 // 5`. Potem się sprawdz.

★★ Zadanie samodzielne
Znajdź błąd w wzorze
W kodzie `srednyaya_ocenka = ocenka1 + ocenka2 + ocenka3 / 3` jest błąd priorytetu. Znajdź i napraw go.

Challenge
Śledzenie wzoru
Weź formułę obliczania łącznej kwoty zamówienia z uwzględnieniem rabatu i dostawy z powyższego przykładu i dodaj do niej śledzenie z czytelnymi nazwami zmiennych pośrednich.

Praktyka: Obliczania debugowania
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji
Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/05-21/index.html)

Mini-projekty i podsumowania

Zebranie całego rozdziału w pięć małych, ale realnych projektów i podsumowanie wyników.

Pięć małych, ale realnych projektów – każdy wykorzystujący tylko to, co już omówiliśmy tym rozdziale.

Projekt 1: inteligentny kalkulator

Liczymy wynik w zależności od wybranego operatora — `+`, `-`, `*` lub `/`:

wyprzedzanie faktów
Użyty tutaj projekt `if / elif` — omówimy to szczegółowo w rozdziale 8. Na razie wystarczy czytać to jak zwykły tekst: „jeśli operator taki-to — zrób tak, a jeśli inny — inaczej”.

umnyj_kalkulyator.py

```
a, b = 12, 4
operator = "+"

if operator == "+":
    rezultat = a + b
elif operator == "-":
    rezultat = a - b
elif operator == "*":
    rezultat = a * b
elif operator == "/":
    rezultat = a / b

print(rezultat)   # 16
```

Podstawowa praktyka
Dodaj //, % i **
Rozszerzyć kalkulator tak, aby mógł także wykonywać dzielenie liczb całkowitoliczbowych, reszty i potęgowanie.

Projekt 2: Konwerter Czasu – Krótsza wersja

Problem rozwiązaliśmy już w rozdziale 4 — przez `//` oraz `%` osobno. Teraz, kiedy wiemy `divmod()` (sekcja „Dzielenie z resztą”

konverter_vremeni_divmod.py

```
total_seconds = 3725

hours, ostatok = divmod(total_seconds, 3600)
minutes, seconds = divmod(ostatok, 60)

print(f"{hours}godziny {minutes}m {seconds}z")   # 1h 2m 5s
```

★★ Zadanie samodzielne
Challenge
Sprawdź swój konwerter na 0 sekund i na 90000 sekund (więcej niż dobę).

Projekt 3: Laboratorium Kostek

Rzuć dwoma kostkami i policz sumę oraz średnią – użyj `random.randint()` już znanych operatorów:

laboratoriya_kubika.py

```
import random

kub1 = random.randint(1, 6)
kub2 = random.randint(1, 6)
summa = kub1 + kub2
srednee = summa / 2

print(f"Kostka 1: {kub1}, sześcian 2: {kub2}")
print(f"Suma: {summa}, średnia: {srednee}")
```

★★ Zadanie samodzielne

Podwójnie!

Dodaj sprawdzenie: jeśli obie kości są takie same, wypisz „Double!”

Projekt 4: losowe wyzwanie — wielokrotności liczb

Program, który znajduje wszystkie wielokrotności losowo wybranego jednego – użyj `random` oraz `%` razem:

kratnye_chisla.py

```
import random

# liczba losowa, której liczności szukamy
kratnoe_chemu = random.randint(2, 9)
print(f"Szukamy liczb podzielnych przez {kratnoe_chemu}, od 1
do 50")

chislo = 1
while chislo <= 50:
    if chislo % kratnoe_chemu == 0:
        print(chislo)
    chislo += 1
```

Pospieszmy się jeszcze raz

Cykl `while` omówimy szczegółowo w rozdziale 10. Tutaj ważne jest, aby zobaczyć, jak `%` z tego rozdziału znajduje wszystkie wielokrotności danej liczby: reszta dzielenia jest równa zero dokładnie wtedy, gdy liczba jest podzielna jako liczba całkowita.

Challenge

Własny zasięg

Zmień granice wyszukiwania z 1–50 na 1–100.

Projekt 5: laboratorium geometryczne

Zebranie kilku wzorów w tym rozdziale jednocześnie: odległość między dwoma punktami oraz pole okręgu o następującym promieniu:

geometricheskaya_laboratoriya.py

```
import math

tochka_a = (0, 0)
tochka_b = (3, 4)

radius = math.dist(tochka_a, tochka_b)
ploshad_kruga = math.pi * radius ** 2

print(f"Odległość między punktami: {radius}")
print(f"Pole okręgu o takim promieniu: {round(ploshad_kruga,
2)}")
```

Challenge

Obwód trójkąta

Dodaj trzeci punkt i oblicz obwód trójkąta utworzonego przez wszystkie trzy punkty, używając math.dist() trzech razy.

Praktyka: Mini-projekty w rozdziale 5

interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/05-06/index.html)

Ostateczna mapa rozdziału

Jakiego narzędzia do rozdziału 5 potrzebuję?

Potrzebujesz sprawdzenia salda lub mnożności?	→ `// i %`
Potrzebujesz dyplomu czy root?	→ `** lub math.sqrt()`
Formuła nie mieści się w jednej zrozumiałej linii?	→ `nazwane kroki pośrednie`
Czy potrzebujesz geometrii — odległości, powierzchni, kąta?	→ `math.hypot() / math.dist() / math.pi`
Potrzebujesz losowości do gry?	→ `random`
Musisz powtórzyć ten sam losowy wynik?	→ `random.seed()`

Wracaj do tej mapy w przyszłych projektach — zbiera ona cały rozdział w sześć pytań

Podsumowanie rozdziału

Co możemy teraz zrobić

- Wyrażenie — operandy plus operator; odróżniamy wyrażenie od instrukcji (statement).
- Główne operatory `+ - * /` i ich modele geometryczne: prosta liczbowa dla +/-, prostokąt dla mnożenia.
- `//`, `%` oraz `divmod()` — w tym jak działają z liczbami ujemnymi (zaokrąglanie do minus nieskończoności).
- `**`, `pow()` oraz `math.pow()` to klasyczna pułapka `-2 ** 2`.
- Skrócone operatory przypisania, pełna mapa priorytetów oraz asocjowalność (w tym prawicowa asocjatiwność `**`).
- Tłumaczenie wzorów z notacji matematycznej na Python — i typowe błędy tłumaczeniowe.
- Głęboki moduł `math`: pierwiastki i odległości, gcd/lcm/factorial/comb/perm, geometria, trygonometria, logarytmy.
- Głęboki moduł `random`: pseudo-losowość, randint/randrange/uniform, choice/choices/sample/shuffle, seed i random vs secrets.
- Obliczania debugowania: trzy typy błędów oraz technika śledzenia formuły przez nazwane kroki.

co dalej

W rozdziale 6 weźmiemy liczby, które teraz umiemy liczyć, i zaczniemy je **rysować** — zapoznajmy się z modułem `turtle` narysować pierwsze kształty na ekranie. Współrzędne, kąty i odległości z tego rozdziału od razu się przydadzą.

ROZDZIAŁ 6 · STR. 234

Rysowanie fajnych rzeczy Turtle

Pierwsza prawdziwa grafika na Python: współrzędne, ruch i obroty, wielokąty, pióro i wypełnienie, okręgi i kolory, grafika losowa — i mandala na koniec. Każdy przykład pokazuje naprawdę wykonany rysunek, a nie opis tego, co powinno powstać.

Zaczynamy

Poznaj dwa główne elementy Turtle grafiki — ekran i samego żółwia — oraz ideę, że żółw ma fortunę.

Moduł `turtle` wchodzi w standardową dostawę Python — nic więcej nie trzeba instalować.

Trochę historii

Idea „grafiki żółwi” `turtle` w Python bezpośrednim potomkiem tego samego żółwia Logo-.

Ekran i żółw to dwa różne obiekty

Przed rysowaniem potrzebujesz dwóch obiektów: **ekran** (`turtle.Screen()`) to okno, w którym odbywa się rysunek i **żółw** (`turtle.Turtle()`) jest tym, co w rzeczywistości, remisuje.

screen_i_cherepashka.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=480, height=360)
artist = turtle.Turtle()

screen.exitonclick()
```

REZULTAT

Białe okno Turtle z małym czarnym trójkątem żółwia w centrum wskazującym w prawo

Puste okno 480×360 z żółwiem u początku, kierując się w prawo

Mały trójkąt w centrum to żółw w pozycji startowej. Trasa Kierunek żółwia jest domyślnie skierowany w prawo – na wschód.

STAN ŻÓŁWIA

position	**(0, 0)**
heading	**0° (Wschód)**
pen	**down**

Żółw ma fortunę

Zauważ: żółw w każdym momencie ma kilka cech jednocześnie — gdzie się znajduje (*stanowisko*), gdzie patrzy (*kurs*), i czy rysuje teraz (*długopis*). Każde polecenie, które będziemy studiować w tym rozdziale, zmienia jedno z tych cech, nie dotykając reszty. Wróćmy do tego paska statusu na przez cały rozdział.

Dlaczego dwa oddzielne obiekty?

Podzielony ekran i żółwie pozwalają mieć na tym samym ekranie jednocześnie **kilka** żółwi — każdy będzie miał swój własny stan, niezależny od pozostałych.

Praktyka: Zaczynam od ustawienia ekranu

Moduł turtle otwiera natywne okno Python – uruchamiane lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/06-02/index.html)

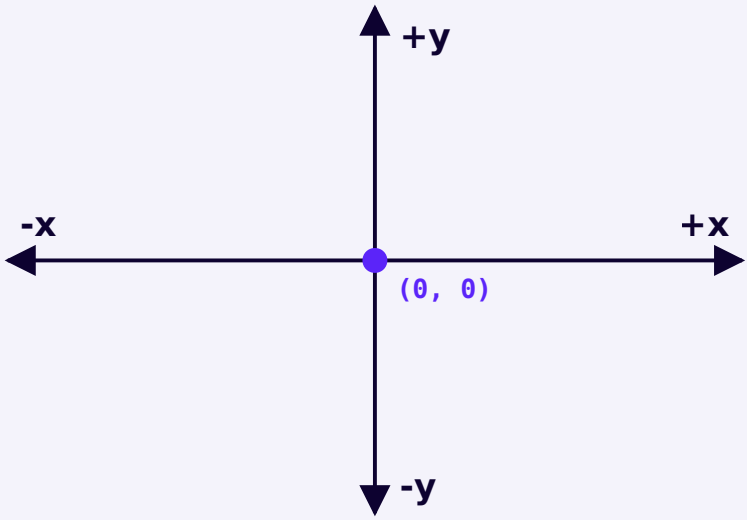
Współrzędne: środek (0, 0)

Liczby z rozdziału 5 stają się miejscem na ekranie – poznając układ współrzędnych okna Turtle.

Już widzieliśmy współrzędne w rozdziale 5 — punkty na płaszczyźnie z współrzędnymi x i y. Window Turtle jest ułożona dokładnie według tego samego systemu i dosłownie kontroluje, co rysuje Żółw.

środku ekranu znajduje się kropka (0, 0)

W przeciwieństwie do wielu innych systemów graficznych (gdzie oś Y rośnie *na ziemię*), Turtle używa zwykłego matematycznego układu współrzędnych:



Środek okna — punkt (0, 0). Oś X rośnie w prawo, oś Y rośnie w górę.

- **x** — poziomo: wartości dodatnie po prawej, ujemne po lewej.
- **y** — pionowo: wartości dodatnie w górę, wartości ujemne w dół.
- środku okna jest punkt `(0, 0)` tam, gdzie żółw pojawia się domyślnie.

Współrzędna	Tam, gdzie wskazuje
`(100, 0)`	po prawej stronie od środka
`(-100, 0)`	na lewo od środka
`(0, 100)`	w górę od środka
`(0, -100)`	w dół od środka

Współrzędne faktycznie kontrolują okno

To nie jest tylko abstrakcyjny schemat — układ współrzędnych naprawdę decyduje, gdzie Żółw tam dotrze, jeśli zostanie tam wysłany:

koordinaty.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=480, height=480)
artist = turtle.Turtle()
artist.hideturtle()
artist.speed(0)

# oś X
artist.penup(); artist.goto(-200, 0); artist.pendown()
artist.pencolor("#0D0230"); artist.pensize(2)
artist.goto(200, 0)

# oś Y
artist.penup(); artist.goto(0, -200); artist.pendown()
artist.goto(0, 200)

# oznacz początek i jeden punkt
artist.penup(); artist.goto(0, 0)
artist.dot(10, "#5B24F9")
artist.goto(120, 80)
artist.dot(10, "#5B24F9")
artist.write("  (120, 80)", font=("Arial", 12, "normal"))
artist.goto(0, -20)
artist.write("  (0, 0)", font=("Arial", 12, "normal"))

screen.exitonclick()
```

REZULTAT

Okno Turtle z przecinającymi się osiami poziomymi i pionowymi w centrum oraz dwoma oznaczonymi punktami: (0, 0) i (120, 80)

osi, początek (0, 0) i punkt (120, 80) zaznaczone bezpośrednio w oknie Turtle

write() - Podpisz punkt na ekranie

W powyższym przykładzie polecenie `artist.write("текст")` — drukuje tekst w aktualnej pozycji żółwia. Wygodne do oznaczania punktów na diagramach, jak tutaj.

Praktyka: współrzędne i goto()

Moduł turtle otwiera natywne okno Python – uruchamiane lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/06-09/index.html)

Poruszanie się do przodu i do tyłu

Pierwsza rzecz, jaką musi umieć żółw — poruszać się. W tej sekcji napiszesz swój pierwszy program, który naprawdę rysuje.

Po co to potrzebne

Wszystko, co rysujesz dalej — wielokąty, domy, mandale — jest zbudowane z tego samego Pary działań: **„trochę pojeździć”** oraz **„Za rogiem za rogiem”**. Opanując cztery polecenia z tej części, będziesz mieć fundamenty na cały rozdział.

Składnia

- `forward(расстояние)` polega na przesunięciu się do przodu o określoną liczbę pikseli (skrót to `fd`)
- `backward(расстояние)` — wracać bez skręcania (skróty — `bk`, `back`)
- `right(угол)` — obróć zgodnie z ruchem wskazówek zegara o kąt w stopniach (skrót — `rt`)
- `left(угол)` — obróć przeciwnie do ruchu wskazówek zegara o kąt wyliczony w stopniach (skrót — `lt`)

Pierwszy krok: po prostu naprzód

vpered_120.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=480, height=300)
artist = turtle.Turtle()
artist.pensize(3)
artist.pencolor("#5B24F9")

artist.forward(120)

screen.exitonclick()
```

REZULTAT

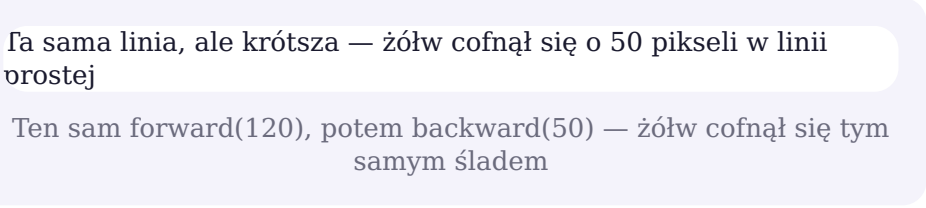
artist.forward(120) — żółw przejechał 120 pikseli w prawo, zostawiając linię

Do przodu, potem do tyłu

vpered_nazad.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=480, height=300)
artist = turtle.Turtle()
artist.pensize(3)
artist.pencolor("#5B24F9")

artist.forward(120)
artist.backward(50)

screen.exitonclick()
```

REZULTAT



Ta sama linia, ale krótsza — żółw cofnął się o 50 pikseli w linii prostej

Ten sam forward(120), potem backward(50) — żółw cofnął się tym samym śladem

Porady

Liczby w `forward()` mogą być również ujemne. `forward(-50)` oraz `backward(50)` robią to samo.

Pierwszy prawdziwy program: Litera „G”

Jedźmy do przodu, skręćmy i jedźmy dalej:

bukva_g.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=480, height=360)
artist = turtle.Turtle()
artist.pensize(3)
artist.pencolor("#5B24F9")

artist.forward(120)
artist.left(90)
artist.forward(80)

screen.exitonclick()
```

REZULTAT

Analiza krok po kroku

- `turtle.Screen()` otwiera okno do rysowania.
- `turtle.Turtle()` tworzy sam żółw — na początku współrzędnych, kurs skierowany w prawo.
- `artist.forward(120)` - Żółw porusza się 120 pikseli w kierunku toru, zostawiając linię.
- `artist.left(90)` — tor jest obracany o 90° przeciwnie do ruchu wskazówek zegara.
- Drugie Wyzwanie `forward()` już zmierza w nowym kierunku – stąd róg litery „G”

Typowy błąd

Uwaga
Początkujący często się mylą `right()` / `left()` ruchem — wydaje się, że żółw powinien przesunąć się na bok. W rzeczywistości te komendy **tylko obracają** oczywiście i nie ruszaj żółwia ani o piksel.

Diagnoza: czy kształt wygląda na „spłaszczony” `forward()` po obrocie, albo pomylono miejscami obrót i ruch.

Spróbuj się zmienić

Podstawowa praktyka
Inny dystans i kąt
Zmień odległość z 120 do 250 i kąt z 90 na 45. Przed rozpoczęciem spróbuj przewidzieć: w którą stronę żółw „wygląda”

★★ Zadanie samodzielne
Trzeci odcinek
Dodaj trzecią parę poleceń left(90) i forward(120) na końcu programu. Jaka figura powstanie?

Modern Option

W starym kodzie żółw często jest rysowany bez wyraźnego obiektu, czyli modułu `turtle` automatycznie tworzy „żółwia domyślnego”.

Klasyczne podejście → nowoczesny Python 3.14

Klasyczne podejście

```
import turtle

# bez dopełnienia jawnego —
# używane ukrycie
# żółw domyślny
turtle.forward(120)
turtle.left(90)
turtle.forward(80)
turtle.done()
```

Nowoczesny Python 3.14

```
import turtle

# obiekt jawny — rozumiem,
# który żółw rysuje,
# łatwo dodać drugi
artist = turtle.Turtle()
artist.forward(120)
artist.left(90)
artist.forward(80)
turtle.done()
```

Co używać dzisiaj: obiekt jawny `turtle.Turtle()`. Nie komplikuje niczego, ale natychmiast przygotowuje do rozdziału o obiektach i jest łatwo skalowalny — jeśli później będziesz musiał narysować scenę z kilkoma żółwiami, każdy będzie miał własną nazwę i swój stan. Funkcje modułowe (`turtle.forward()` bez obiektu) wciąż spotyka się w starych kodach i podręcznikach — warto je znać, ale od razu zacznij zwyczaj od obiektu.

Praktyka: eksperymenty, zadanie i samodzielna praktyka
Moduł turtle otwiera natywne okno Python – uruchamiane lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/06-02/index.html)

Co pamiętać

- `forward()` / `backward()` przesuwać żółwia w linii prostej; `left()` / `right()` tylko skręcać trasę.
- Pozycja i kurs to dwie niezależne właściwości żółwia.
- Nowoczesny styl — tworzyć jawny obiekt `turtle.Turtle()` zamiast polegać na domyślnym ukrytym żółwiu.

Kierunek i kąt

Żółw może stać w miejscu i patrzeć w cztery różne strony — analizujemy, czym kurs różni się od pozycji.

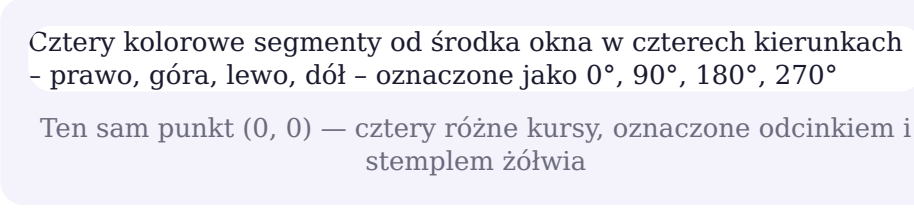
Ważne jest, aby rozróżnić dwie rzeczy, które łatwo pomylić: **pozycja** żółwie (gdzie się znajduje) i jej **kurs** (gdzie patrzy). Żółw może stać w ten sam punkt i jednocześnie patrzenie w cztery zupełnie różne strony.

Cztery stany w jednym punkcie

chetyre_napravleniya.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=480, height=480)
artist = turtle.Turtle()
artist.shape("classic")
artist.speed(0)
artist.penup()

headings = [(0, Wschód · 0°"), (90, Północ · 90°"), (180,
„Zachód · 180°”), (270, Południe · 270°")]
colors = ["#5B24F9", "#DB2777", "#059669", "#D97706"]
for (angle, label), color in zip(headings, colors):
    artist.setheading(angle)
    artist.goto(0, 0)
    artist.pencolor(color)
    artist.pendown()
    artist.forward(90)
    artist.penup()
    artist.write("  " + label, font=("Arial", 11, "normal"))
    artist.goto(0, 0)

screen.exitonclick()
```

REZULTAT



Cztery kolorowe segmenty od środka okna w czterech kierunkach – prawo, góra, lewo, dół – oznaczone jako 0°, 90°, 180°, 270°

Ten sam punkt (0, 0) — cztery różne kursy, oznaczone odcinkiem i stemplem żółwia

Obrót zmienia kurs, ale nie pozycję

Oto, co dzieje się ze stanem żółwia po odwróceniu – zwróć uwagę, że `position` nie zmieniło ani jednego piksela:

STAN ŻÓŁWIA

position	**(0, 0)**
heading	**0° (Wschód)**
pen	**down**

→ left(90) →

STAN ŻÓŁWIA

position	**(0, 0)**
heading	**90° (północ)**
pen	**down**

Kąty z rozdziału 5

Te kąty widzieliśmy już w rozdziale 5, gdy rozmawialiśmy o trygonometrii i kręgu jednostkowym - Teraz kontrolują bezpośredni kierunek żółwia:

Corner	Co to oznacza dla żółwia
360°	pełny obrót — żółw patrzy ponownie tam, skąd zaczął
180°	odwrócenie — żółw patrzy dokładnie w przeciwną stronę
90°	ćwierć obrotu — kąt prosty, kurs staje się prostopadły

Corner	Co to oznacza dla żółwia
45°	połowa kąta prostego to tor diagonalny

Kierunek stopni zaczynający od wschodu
Na Turtle 0° jest na wschód (domyślny kurs), a stopnie wznoszą się przeciwnie do ruchu wskazówek zegara: 90° na północ, 180° na zachód, 270° na południe. To ta sama zasada co na okręgu jednostkowym z rozdziału 5.

Praktyka: kierunek i stan żółwia
Moduł turtle otwiera natywne okno Python – uruchamiane lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/06-10/index.html)

Praktyka: Przewiduj kurs i koordynuj (bez Turtle)
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/06-19/index.html)

Zmuszamy żółwia do zmiany kierunku

Oprócz względnych skrętów left()/right(), żółw ma sposób na absolutne ustawienie kierunku, niezależnie od tego, gdzie wcześniej patrzył.

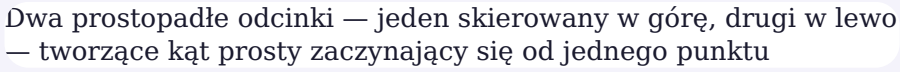
Drużyny `left()` / `right()` z poprzedniego rozdziału obracają żółwia **krewny** jej obecnym kursie. Czasem wygodniej jest ustawić kierunek **absolutnie** — „Patrz wyłącznie na północ”

Kurs absolutny: `setheading()`

setheading.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=480, height=400)
artist = turtle.Turtle()
artist.pensize(3)
artist.pencolor("#5B24F9")

artist.setheading(90)
artist.forward(100)

artist.setheading(180)
artist.forward(100)

screen.exitonclick()
```

REZULTAT



Dwa prostopadłe odcinki — jeden skierowany w górę, drugi w lewo — tworzące kąt prosty zaczynający się od jednego punktu

setheading(90) → forward(100), potem setheading(180) → forward(100) — kurs jest ustalony absolutnie, niezależnie od poprzedniego kierunku

setheading_kod.py

```
artist.setheading(90)   # patrz prosto w górę, gdziekolwiek wcześniej patrzyłeś
artist.forward(100)

artist.setheading(180)  # patrzymy ściśle w lewo
artist.forward(100)
```

Powrót do domu: `home()`

Wraca żółwia do pierwotnego punktu (0, 0) z pierwotnym kursem (0°) – jest wygodne Zacznij rysunek od nowa, nie tworząc nowego obiektu:

home.py

```
artist.home()
```

heading() — sprawdź aktualny kurs wymiany

Jeśli nie chcesz wyznaczać kursu, ale *się przekonać* to, użyj `artist.heading()` : zwraca bieżący kierunek w stopniach, nie zmieniając go. U `setheading()` jest też krótki pseudonim — `seth()` .

Praktyka: setheading(), heading() i home()

Moduł turtle otwiera natywne okno Python – uruchamiane lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/06-03/index.html)

Pierwsze liczby i wzór 360/n

Od pojedynczych segmentów do rzeczywistych wielokątów i do wzoru, który wyjaśnia je wszystkie naraz.

Użyj ruchu i obrotu, aby narysować pierwsze prawdziwe kształty — jeszcze żadnych pętli, Po prostu powtarzając tę samą parę komend tyle razy, ile chcesz.

Triangle

Trójkąt ma trzy boki i trzy kąty obrotu. Suma wszystkich zewnętrznych narożników dowolnego Wielokąt zawsze ma 360°, więc jeden obrót tutaj to `360 / 3 = 120` stopni:

treugolnik.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=480, height=400)
artist = turtle.Turtle()
artist.pensize(3)
artist.pencolor("#5B24F9")

artist.forward(120)
artist.right(120)
artist.forward(120)
artist.right(120)
artist.forward(120)
artist.right(120)

screen.exitonclick()
```

REZULTAT

Prawidłowy trójkąt — trzy obroty po 120°

Prostokąt

Prostokąt nadal ma kąty proste (90°), ale boki są naprzemiennie – długie, krótkie, Krótko:

pryamougolnik.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=480, height=360)
artist = turtle.Turtle()
artist.pensize(3)
artist.pencolor("#5B24F9")

artist.forward(160)
artist.right(90)
artist.forward(90)
artist.right(90)
artist.forward(160)
artist.right(90)
artist.forward(90)
artist.right(90)

screen.exitonclick()
```

REZULTAT

Prostokąt szerszy niż wyżej narysowany, z jednolitą fioletową linią

Prostokąt – boki o różnych długościach, wszystkie obroty mają kąt 90°

Pentagon

pyatiugolnik.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=480, height=440)
artist = turtle.Turtle()
artist.pensize(3)
artist.pencolor("#5B24F9")

artist.forward(90)
artist.right(72)
artist.forward(90)
artist.right(72)
artist.forward(90)
artist.right(72)
artist.forward(90)
artist.right(72)
artist.forward(90)
artist.right(72)

screen.exitonclick()
```

REZULTAT

Regularny pięciokąt narysowany jednolitą fioletową linią

Poprawny pięciokąt — pięć obrotów po 72°

Wzór dla dowolnego regularnego wielokąta

Ogólna zasada działająca dla wszystkich powyższych kształtów:

$$\text{obrót} = \frac{360}{n}$$

gdzie `n` — liczba boków. Za każdym razem, gdy żółw przechodzi poprawny wielokąt dookoła i wraca do punktu wyjścia z pierwotnym kursem, suma wszystkich obrotów musi wynosić dokładnie jeden pełny obrót — 360°.

Figura	Partie (n)	Obrót = 360 / n
Triangle	3	120°
Square	4	90°
Pentagon	5	72°
Hexagon	6	60°

Kwadrat i sześciokąt w następnej sekcji
Celowo zostawiliśmy kwadrat (360 / 4 = 90°) i sześciokąt (360 / 6 = 60°) na kolejny mini-projekt, któremu przyjrzymy się bliżej.

Praktyka: pierwsze liczby
Moduł turtle otwiera natywne okno Python – uruchamiane lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/06-11/index.html)

Praktyka: Formula 360/n (bez Turtle)
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/06-20/index.html)

Mini-projekt — narysuj kwadrat i sześciokąt

Zastosuj wzór 360/n do dwóch najbardziej rozpoznawalnych wielokątów.

Zastosowanie wzoru `360 / n` z poprzedniej sekcji do dwóch najbardziej Rozpoznawalne postacie.

Mini-projekt — rysowanie kwadratu

Kwadrat ma cztery identyczne boki i cztery kąty 90°: `360 / 4 = 90`. Więc trzeba powtórzyć cztery razy tę samą parę poleceń: jechać do przodu, skręcić o 90°.

kvadrat.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=400, height=400)
artist = turtle.Turtle()
artist.pensize(3)
artist.pencolor("#5B24F9")

artist.forward(100)
artist.right(90)
artist.forward(100)
artist.right(90)
artist.forward(100)
artist.right(90)
artist.forward(100)
artist.right(90)

screen.exitonclick()
```

REZULTAT

Regularny kwadrat narysowany jednolitą fioletową linią

Kwadrat: cztery boki po 100 pikseli, cztery obroty o 90°

Zauważyłeś powtórzenie?
Cztery identyczne bloki w rzędzie to wyraźny znak, że w powietrzu jest cykl `for`. Wrócimy do tego samego kwadratu w rozdziale na pętlach i przepiszemy go w 2 linijki zamiast 8.

Mini-projekt – narysuj sześciokąt

Sześciokąt ma sześć równych boków: `360 / 6 = 60` stopni przez Każdy krok.

shestiugolnik.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=420, height=400)
artist = turtle.Turtle()
artist.pensize(3)
artist.pencolor("#5B24F9")

artist.forward(80)
artist.right(60)
artist.forward(80)
artist.right(60)
artist.forward(80)
artist.right(60)
artist.forward(80)
artist.right(60)
artist.forward(80)
artist.right(60)
artist.forward(80)
artist.right(60)

screen.exitonclick()
```

REZULTAT

Prawidłowy sześciokąt, narysowany pełną fioletową linią

Heksagon: Sześć boków o średnicy 80 px, sześć obrotów o 60°

Praktyka: kwadrat, sześciokąt i inne wielokąty

Moduł turtle otwiera natywne okno Python – uruchamiane lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/06-04/index.html)

Techniki skrócone

Każda komenda ruchu ma krótszy pseudonim — warto umieć czytać obie wersje.

Każdy zespół ruchu, który badaliśmy, ma krótszy pseudonim – oni Rób dokładnie to samo, tylko drukuj krócej:

- `forward()` → `fd()`
- `backward()` → `bk()` (lub `back()`)
- `left()` → `lt()`
- `right()` → `rt()`

Pełne nazwy drużyn → krótkie aliasy

Imiona i nazwiska	Krótkie aliasy
`artist.forward(100)`	`artist.fd(100)`
`artist.backward(50)`	`artist.bk(50)`
`artist.left(90)`	`artist.lt(90)`
`artist.right(90)`	`artist.rt(90)`

Co używać dzisiaj: oba warianty są oficjalne i robią dokładnie to samo — w modułu turtle istnieją aliasy specjalnie dla szybszego pisania. W tej książce zwykle używamy pełnych imion, ponieważ są czytelniejsze przy pierwszym czytaniu, ale warto nauczyć się krótkich form: w kodzie innych osób i przykładach w Internecie znajdziesz obie opcje.

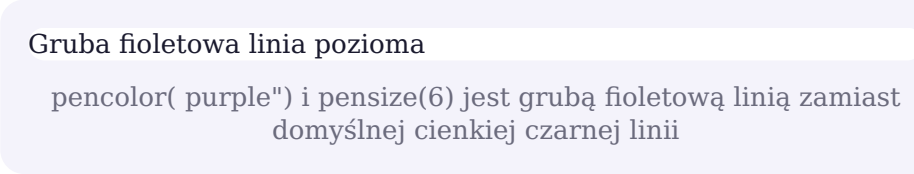
Pierwsze spojrzenie na formatowanie linii

Oprócz ruchu, w następujących sekcjach spotkają się jeszcze dwie drużyny — Poznajmy ich teraz krótko:

oformlenie.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=420, height=300)
artist = turtle.Turtle()

artist.pencolor("purple")
artist.pensize(6)
artist.fd(200)

screen.exitonclick()
```

REZULTAT



pencolor(purple") i pensize(6) jest grubą fioletową linią zamiast domyślnej cienkiej czarnej linii

oformlenie_kod.py

```
artist.pencolor("purple")   # Kolor linii
artist.pensize(6)            # grubość linii
```

Praktyka: Obejmuje treningi z drużynami krótkimi
Ten sam laptop co Mini-Projects: Shapes – to też obejmuje

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/06-04/index.html)

Podnieś i opuść długopis

Czasem trzeba się ruszać bez rysowania — poznać penup() i pendown().

Do tej pory każdy ruch żółwia zostawiał linię. Ale czasem trzeba się ruszyć, nic Bez rysowania — na przykład żeby zacząć nowy kształt gdzie indziej na ekranie, nie łącząc go z nim poprzednie.

Model mentalny

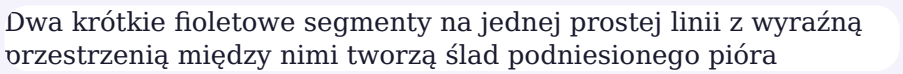
Wyobraź sobie prawdziwy długopis na papierze: gdy dotyka kartki, każdy ruch ręki Podnieś uchwyt, a ręka może się poruszać gdziekolwiek, nie zostawiając ani jednej ręki. linii.

`penup()` oraz `pendown()`

pero.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=480, height=260)
artist = turtle.Turtle()
artist.pensize(3)
artist.pencolor("#5B24F9")

artist.forward(60)      # Długopis w dół — losowanie
artist.penup()
artist.forward(60)      #pen podniesiony — tylko się ruszam,
bez kolejki
artist.pendown()
artist.forward(60)      # długopis opuszczony ponownie —
losowanie

screen.exitonclick()
```

REZULTAT



Linia-Przestrzeń-Linia: penup() między dwoma ruchami pozostawia widoczną lukę

pero_kod.py

```
artist.forward(60)      # Długopis w dół – losowanie
artist.penup()
artist.forward(60)      #pen podniesiony – tylko się ruszam,
bez kolejki
artist.pendown()
artist.forward(60)      # długopis opuszczony ponownie –
losowanie
```

Nie zapomnij włączyć pióra z powrotem

Najczęstszy błąd z `penup()` to zapomnieć zadzwonić `pendown()` po nim. Wtedy wszystko, co jest rysowane później w programie, stanie się niewidoczne — żółw porusza się uczciwie, po prostu nic nie zostawia na ekranie.

isdown() - Sprawdź stan długopisu

Jeśli chcesz wiedzieć, czy długopis jest teraz nieaktywny — `artist.isdown()` wróci `True` lub `False` bez zmiany samego stanu.

Praktyka: penup() i pendown()

Moduł turtle otwiera natywne okno Python – uruchamiane lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/06-12/index.html)

goto() i panel współrzędnych

Ten sam kwadrat co wcześniej — ale tym razem poprzez bezpośrednie podanie współrzędnych każdego rogu.

Wróćmy do kwadratu z sekcji mini-projektów — tym razem narysujemy go bez ani jednego obrót poprzez bezpośrednie ustawienie czterech punktów narożnych za pomocą komendy `goto()`.

Absolutne Przemieszczenie: `goto(x, y)`

kvadrat_goto.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=400, height=400)
artist = turtle.Turtle()
artist.pensize(3)
artist.pencolor("#5B24F9")

artist.goto(100, 0)
artist.goto(100, 100)
artist.goto(0, 100)
artist.goto(0, 0)

screen.exitonclick()
```

REZULTAT

Regularny kwadrat narysowany przez goto() jest wizualnie identyczny z kwadratem z poprzedniej sekcji

Ten sam kwadrat — tym razem każdy wierzchołek otrzymuje współrzędną bezpośrednio, bez pojedynczej rotacji

kvadrat_goto_kod.py

```
artist.goto(100, 0)
artist.goto(100, 100)
artist.goto(0, 100)
artist.goto(0, 0)
```

Dwa sposoby, jeden wynik

`forward()` + `right()` opisać ruch *względem żółwia* („przejechać, skręcić”); `goto()` opisuje ruch *względem ekranu* („znajdź się w tym punkcie”

Lišta Nawigacji Żółwia

Z wyjątkiem `goto()` żółw ma jeszcze kilka komend do wykorzystania z współrzędnymi bezpośrednimi:

Komenda	Co robi
`position()` (lub `pos()`)	zwraca bieżące współrzędne (x, y)
`xcor()`	zwracać tylko współrzędną x
`ycor()`	zwracać tylko współrzędną y
`setx(x)`	się przemieszczać, zmieniając tylko x
`sety(y)`	przemieścić się, zmieniając tylko y
`home()`	wróć do (0, 0) z kursem 0°

navigacionnaya_panel.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=480, height=360)
artist = turtle.Turtle()
artist.pensize(3)
artist.pencolor("#5B24F9")

artist.penup()
artist.dot(8, "#B9A0FC")
artist.write(„ Start (0, 0)”, font=("Arial", 11, "normal"))
artist.setx(150)
artist.sety(80)
artist.dot(8, "#5B24F9")
artist.write(„Po setx(150), sety(80)”, font=("Arial", 11,
"normal"))

screen.exitonclick()
```

REZULTAT

Na ekranie są dwie kropki: jedna na środku oznaczona jest jako „start (0, 0)”

Punkt startowy (0, 0) i punkt po setx(150), sety(80) — oba zaznaczone i podpisane

navigaciya_kod.py

```
print(artist.position())   # bieżące współrzędne

artist.setx(150)
artist.sety(80)
print(artist.position())   # współrzędnych zmienia się o jedną
oś na raz
```

Praktyka: Rysowanie kształtów za pomocą współrzędnych
Moduł turtle otwiera natywne okno Python – uruchamiane lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/06-07/index.html)

Kolor, grubość i wygląd zewnętrzny

Kontrolowanie tego, jak wygląda linia i sam żółw — oraz ustalanie, co speed() zmienia, a co nie.

Jak dotąd rysowaliśmy domyślną fioletową linią. Czas nauczyć się tego Tak wygląda rysunek – i tak wygląda sam żółw.

Grubość i kolor linii

tonkaya_chernaya.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=380, height=200)
artist = turtle.Turtle()

artist.pensize(1)
artist.pencolor("black")
artist.forward(220)

screen.exitonclick()
```

REZULTAT

Cienka czarna linia pozioma

pensize(1), pencolor(„black”

tolstaya_sinyaya.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=380, height=200)
artist = turtle.Turtle()

artist.pensize(10)
artist.pencolor("#2563EB")
artist.forward(220)

screen.exitonclick()
```

REZULTAT

Gruba niebieska, pozioma linia o tej samej długości

pensize(10), pencolor(#2563EB") to ta sama linia, ale gruba i niebieska

cvet_i_tolschina.py

```
artist.pensize(10)
artist.pencolor("#2563EB")
artist.forward(220)
```

Kolor można ustawiać na różne sposoby

`pencolor("blue")` (nazwa koloru), `pencolor("#2563EB")` (kod szesnastkowy, jak CSS) – obie opcje działają w ten sam sposób.

Wygląd żółwia i tło okien

vid_i_fon.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=380, height=300)
screen.bgcolor("#FAFAFC")
artist = turtle.Turtle()
artist.shape("turtle")
artist.pencolor("#5B24F9")
artist.pensize(2)
artist.forward(120)
artist.left(90)
artist.forward(60)

screen.exitonclick()
```

REZULTAT

Narożnik z dwóch odcinków, narysowany żółwiem w kształcie sylwetki żółwia zamiast trójkąta

shape(„turtle”

vid_kod.py

```
screen.bgcolor("#FAFAFC")
artist.shape("turtle")
# zamiast trójkąta-strzałki — sylwetka żółwia
```

Komenda	Co się zmienia
`screen.title("...")`	pasek tytułowy w oknie
`screen.bgcolor("...")`	kolor tła
`artist.shape("turtle")`	forma kursora żółwia

Komenda	Co się zmienia
`artist.hideturtle()` / `showturtle()`	ukryj/pokaż kursor (linia sama w sobie nie znika)

Speed to animacja, a nie rysowanie

`artist.speed(0)` maksymalnie przyspiesza animacje rysowania (0 jest najszybsze, 1 najwolniejsze, 6 to zwykle). Dotyczy to tylko dla **jak szybko** linia pojawi się na ekranie, ostateczny rysunek będzie całkowicie tak samo przy prędkości 1 i przy prędkości 0. Statyczny obraz nie pokaże różnicy W Speed — jest widoczny tylko na żywo, gdy kod jest uruchamiany.

skorost.py

```
artist.speed(0)   # jak najszybciej – wygodnie przy złożonych wzorcach
```

Praktyka: Kolor, grubość i wygląd
Moduł turtle otwiera natywne okno Python – uruchamiane lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/06-13/index.html)

Wypełnienie, Okrąglenie, Łuk i Punkt

Od pustej ścieżki do wypełnionych kształtów, ukończonych okręgów, łuków i punktów.

Kształty wypełniające

Kontur — to nie wszystko: często chce się wypełnić figurę do środka. W tym celu żółwik musi «zapamiętać», gdzie zaczyna się kontur, i «puścić», gdy kontur się zamknie.

kontur_treugolnika.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=380, height=360)
artist = turtle.Turtle()
artist.pensize(3)
artist.pencolor("#5B24F9")

artist.forward(140)
artist.left(120)
artist.forward(140)
artist.left(120)
artist.forward(140)
artist.left(120)

screen.exitonclick()
```

REZULTAT

Trójkąt równoboczny, narysowany tylko konturem, bez wypełnienia

Trójkąt bez wypełnienia – tylko linia konturowa

zalityj_treugolnik.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=380, height=360)
artist = turtle.Turtle()
artist.pensize(3)
artist.pencolor("#5B24F9")
artist.fillcolor("#B9A0FC")

artist.begin_fill()
artist.forward(140)
artist.left(120)
artist.forward(140)
artist.left(120)
artist.forward(140)
artist.left(120)
artist.end_fill()

screen.exitonclick()
```

REZULTAT

Ген sam trójkąt równoboczny, ale z jasno-fioletowym wypełnieniem wewnątrz konturu

ten sam trójkąt, ale begin_fill()/end_fill() pomalował obszar wewnątrz konturu

zalivka_kod.py

```
artist.pencolor("#5B24F9")
artist.fillcolor("#B9A0FC")

artist.begin_fill()
artist.forward(140)
artist.left(120)
artist.forward(140)
artist.left(120)
artist.forward(140)
artist.left(120)
artist.end_fill()
```

Jak to działa

`begin_fill()` zapamiętuje punkt startowy. Następnie zwykłe polecenia rysowania (`forward()` , `left()` i inne) rysują zamknięty kontur, jak zwykle. `end_fill()` wypełnia obszar wewnątrz tego konturu kolorem `fillcolor()` .

`circle()` jest gotowym poprowadzeniem dla kręgów

Nie musisz rysować ręcznie koła (wiele małych prostych linii pod lekkim kątem) — jest gotowy Drużyna:

krug.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=380, height=380)
artist = turtle.Turtle()
artist.pensize(3)
artist.pencolor("#5B24F9")

artist.circle(80)

screen.exitonclick()
```

REZULTAT

Poprawne koło, narysowane ciągłą fioletową linią

circle(80) to pełne koło o promieniu 80 px

krug_kod.py

```
artist.circle(80)   # promień w pikselach
```

Łuk jest częścią koła

Po drugie, argument opcjonalny `circle()` jest `extent` ile stopni okręgu narysować:

duga.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=380, height=300)
artist = turtle.Turtle()
artist.pensize(3)
artist.pencolor("#5B24F9")

artist.circle(80, 90)

screen.exitonclick()
```

REZULTAT

Łuk to ćwierć okręgu narysowana jednolitą fioletową linią

circle(80, 90) – ten sam okrąg, ale tylko ćwierć (90° z 360°)

duga_kod.py

```
artist.circle(80, 90)   # promień 80, ale tylko 90° z 360°
```

`dot()` jest punktem bez ruchu

W przeciwieństwie do wszystkich poprzednich komend, `dot()` nie porusza żółwia Nie o piksel — po prostu umieszcza wypełnione koło na bieżącej pozycji:

tochki.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=380, height=260)
artist = turtle.Turtle()
artist.hideturtle()
artist.penup()

artist.goto(-100, 0); artist.dot(20, "#5B24F9")
artist.goto(0, 0); artist.dot(30, "#DB2777")
artist.goto(100, 0); artist.dot(40, "#059669")

screen.exitonclick()
```

REZULTAT

Trzy kolorowe kropki o różnych rozmiarach w rzędzie

dot() rysuje punkt o określonej średnicy i kolorze — bez żadnego ruchu żółwia

tochki_kod.py

```
artist.dot(20, "#5B24F9")
artist.forward(100)
artist.dot(30, "#DB2777")
```

Praktyka: Wypełnienie, Okrąg, Łuk i Punkt
Moduł turtle otwiera natywne okno Python – uruchamiane lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/06-14/index.html)

Przechodzimy do losowych punktów na ekranie

goto() przesuwa żółwia bezpośrednio na ekran — wraz z losowymi liczbami daje ciekawy efekt.

Do tej pory żółw poruszał się tylko względem swojej aktualnej pozycji lub po wcześniej znanych współrzędnych. Ale można od razu „przeteleportować" go do konkretnego punktu ekranu poleceniem `goto(x, y)` — a razem z losowymi liczbami z rozdziału 5 to otwiera interesujące możliwości.

Losowe punkty na ekranie

sluchaynye_tochki.py

```
import turtle
import random

random.seed(7)
screen = turtle.Screen()
screen.setup(width=420, height=340)
artist = turtle.Turtle()
artist.hideturtle()
artist.penup()

for _ in range(10):
    x = random.randint(-200, 200)
    y = random.randint(-150, 150)
    artist.goto(x, y)
    artist.dot(14, "#5B24F9")

screen.exitonclick()
```

REZULTAT

10 losowych punktów – random.seed(7) każdy przebieg daje ten sam obraz

sluchaynye_tochki_kod.py

```
import turtle
import random

random.seed(7)   # poprawka seed – zdjęcie będzie takie samo
przy każdej premierze
screen = turtle.Screen()
artist = turtle.Turtle()
artist.penup()

for _ in range(10):
    x = random.randint(-200, 200)
    y = random.randint(-150, 150)
    artist.goto(x, y)
    artist.dot(14, "#5B24F9")
```

penup() przed goto()

Bez `penup()` żółw za każdym razem, gdy się porusza, rysuje linię `goto()` — co zwykle nie jest tym, czego potrzeba do „skoku”

Dlaczego random.seed(7) tutaj

Rozbieraliśmy `random.seed()` w rozdziale 5 — tutaj jest potrzebny dokładnie po to, by KAŻDY przebieg tego kodu dawał ten sam obraz, a nie nowy za każdym razem. Usuń linię z seed a kropki stają się naprawdę losowe przy każdym przebiegu.

Praktyka: goto(), penup()/pendown() i random

Moduł turtle otwiera natywne okno Python – uruchamiane lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/06-06/index.html)

Losowy ruch

Mały krok i losowy zakręt, powtarzany wiele razy, i otrzymujesz krętą, niepowtarzalną trajektorię.

Łącząc mały krok do przodu z losowym obrotem w każdej iteracji, można uzyskać niepowtarzalna, organicznie wyglądająca trajektoria — technika zwana "losową" wędrując" (random walk).

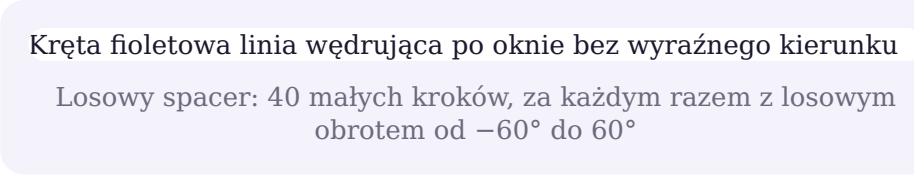
Pomysł — najpierw bez kodu

Wyobraź sobie: zrób mały krok, lekko skręć w losowym kierunku, krok jeszcze raz, jeszcze raz Przypadkowy zwrot akcji – i to tyle razy z rzędu. Brak planu, tylko powtarzanie tego samego Małe rozwiązanie.

sluchaynoe_dvizhenie.py

```
import turtle
import random

random.seed(3)
screen = turtle.Screen()
screen.setup(width=420, height=420)
artist = turtle.Turtle()
artist.pensize(2)
artist.pencolor("#5B24F9")
artist.speed(0)

for _ in range(40):
    artist.forward(15)
    artist.right(random.randint(-60, 60))

screen.exitonclick()
```

REZULTAT



Losowy spacer: 40 małych kroków, za każdym razem z losowym obrotem od −60° do 60°

wyprzedzanie faktów

Poniższy kod wykorzystuje pętlę `for` — omówimy go szczegółowo później. Na razie nie trzeba rozumieć całej składni pętli: ważny jest sam pomysł — mały krok i przypadkowy obrót, powtarzane wiele razy.

sluchaynoe_dvizhenie_kod.py

```python
import turtle
import random

random.seed(3)   # ustalić seed odtwarzanego obrazu
screen = turtle.Screen()
artist = turtle.Turtle()
artist.pensize(2)
artist.pencolor("#5B24F9")
artist.speed(0)

for _ in range(40):
    artist.forward(15)
    artist.right(random.randint(-60, 60))
```

Spróbuj się zmienić

Usuń `random.seed(3)` - Ścieżka będzie nowa za każdym razem, gdy ją przebijesz. Zmień zasięg `randint(-60, 60)` na przykład węższą `(-15, 15)` ścieżka stanie się bardziej bezpośrednia i przewidywalna.

Praktyka: Ruch losowy

Moduł turtle otwiera natywne okno Python – uruchamiane lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/06-15/index.html)

Rysowanie według współrzędnych

Plan na papierze w współrzędnych — i ten sam plan przetłumaczony na kod goto().

Do tej pory budowaliśmy figurki za pomocą ruchu („jedź, skręcaj” `circle()`). Ale możesz zaplanować rysunek od razu w współrzędnych — na papier lub w głowie — i po prostu wypisz punkty dla `goto()`.

Po pierwsze, plan na płaszczyźnie współrzędnych

Zanim napiszesz kod, zdecydujmy, gdzie będą wierzchołki trójkąta:

Szczyt	Współrzędna
Dolny Lewy Róg	(-60, -50)
Dolny prawy róg	(60, -50)
Górny róg	(0, 70)

Teraz jest kod

risuem_po_koordinatam_kod.py

```
artist.fillcolor("#B9A0FC")
artist.penup()
artist.goto(-60, -50)
artist.pendown()

artist.begin_fill()
artist.goto(60, -50)
artist.goto(0, 70)
artist.goto(-60, -50)
artist.end_fill()
```

I wreszcie, efekt

risuem_po_koordinatam.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=380, height=380)
artist = turtle.Turtle()
artist.pensize(3)
artist.pencolor("#5B24F9")
artist.fillcolor("#B9A0FC")
artist.penup()
artist.goto(-60, -50)
artist.pendown()

artist.begin_fill()
artist.goto(60, -50)
artist.goto(0, 70)
artist.goto(-60, -50)
artist.end_fill()

screen.exitonclick()
```

REZULTAT

Cieniowany jasnofioletowy trójkąt zbudowany na trzech wyraźnych współrzędnych

Trójkąt zbudowany według trzech z góry ustalonych współrzędnych, z wypełnieniem

Planowanie w współrzędnych to potężna technika

To podejście jest szczególnie wygodne, gdy figura jest złożona i wymaga uzyskania **na pewno** zgodnie z zamierzeniem jest znacznie bardziej niezawodny niż wybieranie kątów i odległości na oko. W następnej sekcji złożymy kilka postaci w jedną kompozycję, dokładnie taką — najpierw planuj, potem koduj.

Praktyka: Rysuj według współrzędnych
Moduł turtle otwiera natywne okno Python – uruchamiane lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/06-16/index.html)

Debugowanie Turtle

Grafika dodaje kilka nowych sposobów popełniania błędów — analizujmy je według schematu „symptom → powodować → korektę”

Grafika dodaje kilka nowych, Turtle specyficznych sposobów na zrobienie czegoś źle – Przeanalizujmy najczęstsze z nich według schematu „objawy → powodują → korekcję”

Objaw	Przyczyna	Poprawka
Okno otwiera się i od razu zamyka	Program się zakończył, a Python zamknął okno razem z nim	Dodaj `screen.exitonclick()` (czekając na kliknięcie) lub `turtle.done()` na samym końcu programu
Żółw „patrzy nie tam”	Pomyłki `left()` / `right()`, albo jeden z zwrotów w sekwencji zostaje zapomniany	Przeczytaj program linijka po linijce i zapytaj każdą drużynę: „Czy to ruch czy obrót?”
Część obrazu nie pojawia się na ekranie	Zapomniane `pendown()` po `penup()` – żółw porusza się uczciwie, tylko nic nie rysuje	Sprawdź, czy długopis jest opuszczony przed każdym „prawdziwym”
Kształt jest narysowany, ale nie ma wypełnienia	Zapomniane `end_fill()`, albo kontur nie zamknął się dokładnie tam, gdzie się zaczął	Upewnij się, że ścieżka między `begin_fill()` oraz `end_fill()` wraca do punktu wyjścia
Część rysunku „znika” za krawędzią okna	Współrzędne wyszły poza widoczny obszar — okno nie jest nieskończone	Sprawdź `screen.setup()` i nie wychodź poza połowę szerokości/wysokości okna od środka

Objaw	Przyczyna	Poprawka
`TclError` lub okno w ogóle się nie otwiera na serwerze/Linux bez monitora	Turtle używa Tk, natywnego GUI, który potrzebuje prawdziwego (lub wirtualnego) wyświetlacza	Uruchamiaj Turtle na komputerze z tradycyjnym komputerem stacjonarnym; automatyzacja bez monitora wymaga wirtualnego wyświetlacza, jak Xvfb – to już nie jest poziom podstawowy
Praktyka Turtle nie otwiera się bezpośrednio w przeglądarce na tej stronie	Środowisko przeglądarkowe Python (Pyodide) nie potrafi otwierać natywnych okien Tk- — to ograniczenie techniczne, a nie błąd w Twoim kodzie)	Takie praktyki są oznaczone jako „wymagane Python lokalne”

Dlaczego nie Turtle uruchomić bezpośrednio w Twojej przeglądarce na tej stronie

Zwykły Python (CPython) rysuje Turtle przez Tk — bibliotekę natywnych okien systemu operacyjnego. Przeglądarka Python, na której działają interaktywne ćwiczenia tego kursu, w ogóle nie ma dostępu do natywnych okien — dlatego ćwiczenia Turtle są oznaczone jako lokalne od samego początku rozdziału, a nie dlatego, że coś nie działa.

Przewiduj przed startem

Podobnie jak w przypadku liczb w rozdziale 5, najlepszym nawykiem jest przewidywanie kursu i przybliżanie współrzędnych Żółwie **do** startu, a nie później. Jeśli przewidywanie nie zgadza się z rzeczywistością — albo znalazłeś błąd, albo nie rozumiesz dokładnie zespołu. W obu przypadkach warto to przeanalizować.

Podstawowa praktyka

Przewiduj przebieg

Żółw stoi na (0, 0), kurs 0°. Wykonano polecenia: left(90), right(45). Jaki jest końcowy kurs? Sprawdź siebie, wykonując to w prawdziwym oknie Turtle.

★★ Zadanie samodzielne

Znajdź błąd

W programie, który rysuje kwadrat, po trzecim obrocie otrzymujesz pięciokąt, a nie kwadrat. Ile razy para forward()/right(90) powtarza się w kodzie? Ile powinno ich być?

Praktyka: debugowanie Turtle

Moduł turtle otwiera natywne okno Python – uruchamiane lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/06-17/index.html)

Mini-projekty

Cztery małe, ale naprawdę ukończone projekty — zbieramy wszystko, co zostało omówione w tym rozdziale, w gotowe obrazki.

Cztery małe, ale naprawdę ukończone projekty – każdy wykorzystujący tylko to, co robimy zostały już ukończone w tym rozdziale. Piąty projekt, mandala, czeka na samym końcu rozdziału — to jest finał.

Projekt A: Dom Kolorowych

Trzy oddzielne postacie (ściany, dach, drzwi), każda z własnym wypełnieniem, złożone w jedną Kompozycja z wykorzystaniem `penup()` / `goto()` pomiędzy:

cvetnoj_dom.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=420, height=420)
artist = turtle.Turtle()
artist.pensize(3)
artist.speed(0)

# ściany
artist.penup(); artist.goto(-80, -80); artist.pendown()
artist.pencolor("#5B24F9"); artist.fillcolor("#EDE9FE")
artist.begin_fill()
for _ in range(4):
    artist.forward(160)
    artist.left(90)
artist.end_fill()

# dach
artist.penup(); artist.goto(-100, 80); artist.pendown()
artist.pencolor("#B91C1C"); artist.fillcolor("#FCA5A5")
artist.begin_fill()
artist.goto(0, 160)
artist.goto(100, 80)
artist.goto(-100, 80)
artist.end_fill()

# drzwi
Model
artist.penup(); artist.goto(-20, -80); artist.pendown()
artist.pencolor("#78350F"); artist.fillcolor("#92400E")
artist.begin_fill()
for _ in range(2):
    artist.forward(40)
    artist.left(90)
    artist.forward(70)
    artist.left(90)
artist.end_fill()

screen.exitonclick()
```

REZULTAT

Projekt A: Kolorowy dom — ściany, dach i drzwi, trzy oddzielne wypełnienia

Podstawowa praktyka

Window

Dodaj kwadratowe okno w ścianie domu – kolejny mały begin_fill()/ end_fill() blok.

Cel B: Projektu

Cztery okręgi jeden w drugim, o malejącym promieniu i naprzemiennych kolorach:

mishen.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=380, height=380)
artist = turtle.Turtle()
artist.speed(0)
artist.hideturtle()

rings = [(90, "#DC2626"), (65, "#FAFAFC"), (40, "#DC2626"),
(15, "#FAFAFC")]
for radius, color in rings:
    artist.penup()
    artist.goto(0, -radius)
    artist.pendown()
    artist.fillcolor(color)
    artist.pencolor("#0D0230")
    artist.begin_fill()
    artist.circle(radius)
    artist.end_fill()

screen.exitonclick()
```

REZULTAT

Okrągły cel z naprzemiennie czerwonymi i białymi koncentrycznymi pierścieniami

Project B: Target – cztery koncentryczne koła z naprzemiennymi wypełnieniami

mishen_kod.py

```
rings = [(90, "#DC2626"), (65, "#FAFAFC"), (40, "#DC2626"),
(15, "#FAFAFC")]
for radius, color in rings:
    artist.penup()
    artist.goto(0, -radius)
    artist.pendown()
    artist.fillcolor(color)
    artist.begin_fill()
    artist.circle(radius)
    artist.end_fill()
```

wyprzedzanie faktów

Pętla użyta tutaj `for` listę par (promień, kolor) — później szczegółowo rozłożymy listy i pętle. Na razie wystarczy zobaczyć ideę: powtarza się ta sama sekwencja działań dla każdej pary wartości.

★★ Zadanie samodzielne

Niestandardowe kolory

Zamień kolory pierścionków na własne, takie jak odcienie niebieskiego i żółtego.

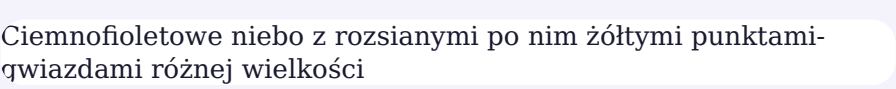
Projekt C: Gwiaździstym Niebie

Ciemne tło, wiele punktów o losowym rozmiarze w losowych miejscach — ten sam zabieg, co w rozdziale o losowych punktach, ale w innym nastroju:

zvezdnoe_nebo.py

```python
import turtle
import random

random.seed(11)
screen = turtle.Screen()
screen.setup(width=420, height=340)
screen.bgcolor("#0D0230")
artist = turtle.Turtle()
artist.hideturtle()
artist.penup()

for _ in range(35):
    x = random.randint(-200, 200)
    y = random.randint(-150, 150)
    size = random.choice([6, 8, 10])
    artist.goto(x, y)
    artist.dot(size, "#FDE68A")

screen.exitonclick()
```

REZULTAT



Project C: Starry Sky – ciemne tło i losowo rozmieszczone gwiazdy

★★ Zadanie samodzielne

Moon

Dodaj jedną dużą białą kropkę w rogu ekranu — „księżyc”, narysowany wcześniej, przed przypadkowymi gwiazdami.

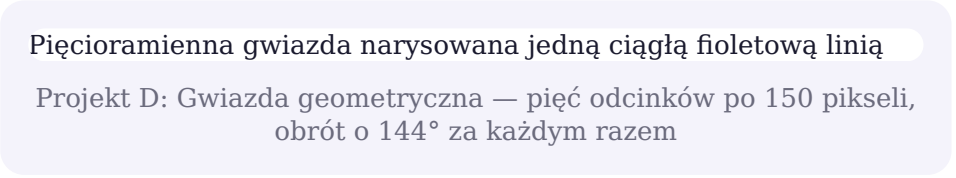
Projekt D: gwiazda geometryczna

Pięcioramienna gwiazda jest rysowana jedną linią, bez ani jednego oderwania pióra – sekret polega na tym, że Obrót o 144° (zamiast 72°) powoduje przecięcie się linii:

geometricheskaya_zvezda.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=380, height=380)
artist = turtle.Turtle()
artist.pensize(3)
artist.pencolor("#5B24F9")
artist.speed(0)

artist.forward(150)
artist.right(144)
artist.forward(150)
artist.right(144)
artist.forward(150)
artist.right(144)
artist.forward(150)
artist.right(144)
artist.forward(150)
artist.right(144)

screen.exitonclick()
```

REZULTAT



Projekt D: Gwiazda geometryczna — pięć odcinków po 150 pikseli, obrót o 144° za każdym razem

geometricheskaya_zvezda_kod.py

```
for _ in range(5):
    artist.forward(150)
    artist.right(144)
```

Skąd wzięła się liczba 144

144° — to `2 × 360 / 5` : Aby gwiazda prawidłowo się „złożyła”

Challenge

Gwiazda Siedmiu Kończy

Zamień 5 na 7 i 144° pod odpowiednim kątem (podpowiedź: ta sama zasada to obracanie się dwa razy w siedmiu krokach). Co się stanie?

Praktyka: mini-projekty

Moduł turtle otwiera natywne okno Python – uruchamiane lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/06-18/index.html)

Mini-projekt — rysowanie mandali wyłącznie z linii prostych

Zebrać wszystkie techniki rozdziału w jednym wzorze — od pierwszego motywu po pełny efekt — i podsumować wyniki.

Zbierz wszystkie techniki rozdziału w jednym finalnym wzorze: mandali składającej się tylko z linii prostych — zbiór promienie tej samej długości, za każdym razem obrócone o niewielki kąt.

Planowanie: co się dzieje

Pomysł prosty: stać w jednym punkcie, patrzeć w określonym kierunku `ugol`, Jedź do przodu i natychmiast wracaj tą samą linią – a następnie lekko skręcaj trasę i powtarzać. Jeśli powtórzymy wystarczająco wiele razy, obejmując całe 360° wokół środka, promienie połączy się w symetryczny wzór.

Jeden motyw

Na początku tylko kilka promieni, by zobaczyć samą technikę, zanim zamieni się w Bogaty wzór:

motiv.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=420, height=420)
artist = turtle.Turtle()
artist.speed(0)
artist.pencolor("#5B24F9")

shag_ugla = 10
ugol = 0
while ugol < 60:
    artist.setheading(ugol)
    artist.forward(150)
    artist.backward(150)
    ugol += shag_ugla

screen.exitonclick()
```

REZULTAT

Wachlarz z kilku fioletowych promieni, wychodzących z jednego punktu pod małym kątem względem siebie

Jeden motyw: kilka promieni pod lekkim kątem względem siebie nie jest jeszcze kompletną mandalą

motiv_kod.py

```
shag_ugla = 10
ugol = 0
while ugol < 60:
    artist.setheading(ugol)
    artist.forward(150)
    artist.backward(150)
    ugol += shag_ugla
```

wyprzedzanie faktów
Cykl `while` zostanie omówione szczegółowo później — tutaj wystarczy zobaczyć, że `setheading()` pozwala narysować złożony wzór w zaledwie kilku łańcuch znakówx, okrążając rogi w kółko.

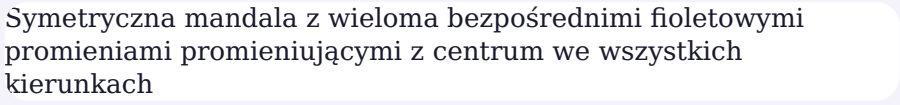
Pełna tura - Ukończona mandala

Ten sam kod, tylko pętla teraz nie sięga 60°, lecz pełnego 360°:

mandala.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=420, height=420)
artist = turtle.Turtle()
artist.speed(0)
artist.pencolor("#5B24F9")

shag_ugla = 10
ugol = 0
while ugol < 360:
    artist.setheading(ugol)
    artist.forward(150)
    artist.backward(150)
    ugol += shag_ugla

screen.exitonclick()
```

REZULTAT



Ta sama technika, obrócona o pełne 360° – mandala złożona z prostych linii

mandala_kod.py

```
shag_ugla = 10
ugol = 0
while ugol < 360:
    artist.setheading(ugol)
    artist.forward(150)
    artist.backward(150)
    ugol += shag_ugla
```

Podstawowa praktyka

Drugie Kątowe Kroki

Zmień shag_ugla na 5 lub na 30 — jak zmienia się wzór?

★★ Zadanie samodzielne

Różne długości wiązki

Zmień długość forward(150) – spróbuj 80 i 250.

Challenge

Kolorowa Mandala

Dodaj artist.pencolor(„violet”

Praktyka: Mandala prostych linii

Moduł turtle otwiera natywne okno Python – uruchamiane lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/06-08/index.html)

Ostateczna mapa rozdziału

Jakie polecenie powinienem Turtle?

muszę wiedzieć, gdzie jest żółw?" → **position() / xcor() / ycor()**

Musisz się ruszać bez rysowania?	→ `penup() / pendown()`
Trzeba trafić w konkretny punkt?	→ `goto(x, y)`
Potrzebujesz odpowiedniego wielokąta?	→ `tura = 360 / n`
Czy potrzebny jest wypełniony kształt?	→ `begin_fill() / end_fill()`
Czy potrzebujesz koła czy łuku?	→ `circle(radius, [extent])`
Czy potrzebujesz powtarzalnej losowości?	→ `random.seed()`

Główna mapa streszczenia rozdziału 6 — wracaj do niej w przyszłych projektach

Podsumowanie

Czego nauczyliśmy się w tym rozdziale

- Ekran (`turtle.Screen()`) i żółwia (`turtle.Turtle()`) to dwa różne obiekty; żółw ma stan: pozycję, kurs, pióro, kolor.
- Okno Turtle używa tego samego układu współrzędnych co matematyka: środek (0, 0), x prawo, y w górę.
- `forward()` / `backward()` poruszają żółwia; `left()` / `right()` skręcić tor względem bieżącego kierunku; `setheading()` ustala kierunek absolutnie.
- Kąt obrotu dla poprawnego wielokąta — `360 / n` .
- `penup()` / `pendown()` kontrolować, czy linia zostanie narysowana podczas ruchu; `goto(x, y)` przesuwa żółwia bezpośrednio w określony punkt na ekranie.
- `begin_fill()` / `end_fill()` wypełniają zamknięty kontur; `circle()` rysuje koła i łuki; `dot()` daje punkt bez ruchu.
- random.seed() sprawia, że losowa grafika jest powtarzalna — przydatne zarówno do dokumentacji, jak i debugowania.

co dalej

W rozdziale 7 wrócimy do Turtle i dopracujemy go jeszcze dokładniej, ucząc się, jak dostrojić ekran, Rysuj tekst, opanuj łuki w większym stopniu i złóż własne emotikony. Wszystko, czego się nauczyłeś tutaj — współrzędne, kąty, pióro, wypełnienie — staną się fundamentem, na którym opiera się więcej niż Dopracowywanie pracy.

ROZDZIAŁ 7 · STR. 297

Dogłębne zanurzenie się w Turtle

Turtle jak prawdziwy system graficzny: okno i płótno, tryby kolorów i colormode, osobno pióro i wypełnienie, geometria głębokiego koła, znaczki, tekst i współrzędne, wiele żółwi i clone(), debugowanie grafiki — oraz cztery mini-projekty, w tym zegar i cel współrzędnych.

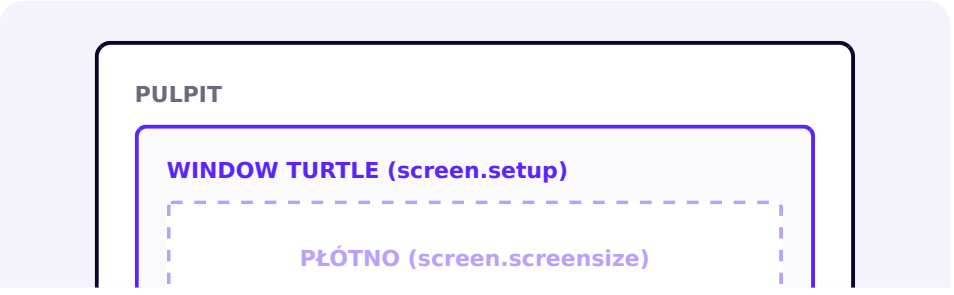
Ekran jako system graficzny

Zanim narysujemy coś bardziej złożonego niż kwadraty, zastanówmy się, co oznacza „rozmiar”

W rozdziale 6 tworzyliśmy już okno i żółwia — ale nie analizowaliśmy, co w ogóle się dzieje w środku. Zacznijmy od tego, że Turtle — to nie tylko „narysować linię”, lecz prawdziwy system graficzny z kilkoma warstwami.

Trzy różne rzeczy, które łatwo pomylić

Kiedy mówimy o „rozmiarze”



Trzy różne rzeczy: okno (widoczny rozmiar), płótno (obszar do rysowania, może być większe niż okno) oraz świat współrzędnych (liczby bez żadnego limitu)

- **Window** jest widoczny na ekranie komputera, rozmiar jest ustawiony `screen.setup()`.
- **Płótno** — obszar, na którym można rysować; zwykle pokrywa się z oknem, ale może być większy (wtedy pojawiają się paski przewijania) — rozmiar jest ustawiony `screen.screensize()`.
- **Coordinate World** — liczby x i y, którymi operujemy w kodzie; nie są ograniczone i nie muszą zgadzać się z pikselami okna jeden do jednego.

Na początek setup() wystarczy

W przeważającej większości programów edukacyjnych wystarczy jeden `screen.setup()` — okno i płótno się pokrywają i nie ma potrzeby myśleć o różnicy. `screensize()` jest potrzebne rzadziej, na przykład, gdy rysunek jest większy, niż rozsądnie byłoby pokazać go na ekranie jako całość.

Konfigurujemy okno: przed i po

Trzy komendy ekranu — nagłówek, kolor tła i rozmiar:

svetlaya_tema.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=500, height=360)
screen.title("Cartesian Turtle Studio")
screen.bgcolor("#F5F2FF")
artist = turtle.Turtle()
artist.pencolor("#5B24F9")
artist.pensize(3)
artist.forward(150)

screen.exitonclick()
```

REZULTAT

Światło Turtle jasnofioletowym oknem z nagłówkiem Cartesian Turtle Studio i krótką fioletową linią

setup(width=500, height=360), title(...) i lekki bgcolor()

svetlaya_tema_kod.py

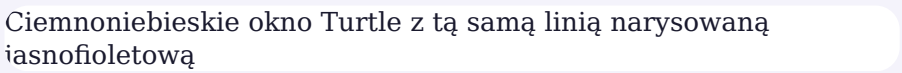
```
screen.setup(width=700, height=500)
screen.title("Cartesian Turtle Studio")
screen.bgcolor("#F5F2FF")
```

Zmień tylko kolor tła — cała reszta logiki rysowania pozostaje taka sama:

temnaya_tema.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=500, height=360)
screen.title("Cartesian Turtle Studio — Dark")
screen.bgcolor("#0D0230")
artist = turtle.Turtle()
artist.pencolor("#B9A0FC")
artist.pensize(3)
artist.forward(150)

screen.exitonclick()
```

REZULTAT



Ten sam kod, ale ze ciemnym bgcolor() — motyw okna nie zmienia logiki rysowania

Praktyka: Konfiguracja ekranu jako systemu
Moduł turtle otwiera natywne okno Python – uruchamiane lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/07-01/index.html)

colormode i kolory

Nazwa, kod HEX-code lub trójka RGB liczb to trzy sposoby na powiedzenie Turtle tego samego: „ten kolor”

Już ustaliliśmy kolory według nazwy — `"purple"`, `"gold"`. Ale Turtle ma też sposób numeryczny — przez komponenty czerwonego, zielonego i niebieskiego (RGB).

Dwa tryby: 0.0-1.0 i 0-255

`screen.colormode()` określa, w jakim zakresie zapisywane są liczby RGB:

Mode	każdy zakres liczb	Przykład czerwieni
`colormode(1.0)` (domyślnie)	0.0 — 1.0	`(1.0, 0.0, 0.0)`
`colormode(255)`	0 — 255	`(255, 0, 0)`

colormode_kod.py

```
screen.colormode(255)
artist.pencolor(255, 0, 0)   # Czerwony, Składniki 0-255

screen.colormode(1.0)
artist.pencolor(1.0, 0, 0)  # ten sam czerwony, komponenty
0.0-1.0
```

Liczby poza zakresem są błędem

Jeśli zostanie zainstalowany `colormode(1.0)`, a przekażesz to dalej `(255, 0, 0)` – Turtle nie zrozumie, co miałeś na myśli z czerwonym: 255 to poza zakresem 0,0-1,0 i pojawi się błąd. Reżim i liczebność muszą się zgadzać.

RGB- – cała paleta

rgb_svatchi.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=460, height=200)
screen.colormode(255)
artist = turtle.Turtle()
artist.hideturtle()
artist.penup()

swatches = [
    ((255, 0, 0), "red"),
    ((0, 200, 0), "green"),
    ((0, 0, 255), "blue"),
    ((255, 210, 0), "yellow"),
]
x = -170
for color, label in swatches:
    artist.goto(x, 0)
    artist.fillcolor(color)
    artist.pencolor("#0D0230")
    artist.pendown()
    artist.begin_fill()
    for _ in range(4):
        artist.forward(70)
        artist.left(90)
    artist.end_fill()
    artist.penup()
    artist.goto(x, -25)
    artist.write(label, font=("Arial", 12, "normal"))
    x += 110

screen.exitonclick()
```

REZULTAT

Cztery kolorowe kwadraty w rzędzie — czerwony, zielony, niebieski i żółty — z podpisami pod każdym

colormode(255) — cztery kolory podane w trójkach (r, g, b) od 0 do 255

rgb_svatchi_kod.py

```
screen.colormode(255)

artist.fillcolor(255, 0, 0)     # czerwony
artist.fillcolor(0, 200, 0)     # zielony
artist.fillcolor(0, 0, 255)     # niebieski
artist.fillcolor(255, 210, 0)   #yellow
```

Trzy sposoby na ustawienie tego samego koloru

Metoda	Przykład	wtedy, gdy jest to wygodne
Nazwa	`"red"`	szybki, jak na standardowe kolory
HEX-kod	`"#FF0000"`	dokładny odcień, jak w CSS/ projekcie
RGB- krotka	`(255, 0, 0)`	gdy kolor jest obliczany w kodzie (na przykład losowo)

Nie zagłębiając się w teorię kolorów
To wystarczy do ćwiczenia tego rozdziału. To, jak RGB jest ułożone „pod maską”

Praktyka: colormode i RGB
Moduł turtle otwiera natywne okno Python – uruchamiane lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/07-10/index.html)

Konfiguracja samego Turtle

Ekran jest ustawiony — teraz skonfigurujmy samego żółwia: prędkość, grubość, kolor, kształt wskaźnika.

W poprzedniej części ustawiliśmy **ekran**. Teraz skonfigurujmy samą **żółw**: prędkość rysowania, grubość i kolor linii oraz kształt wskaźnika.

nastrojka_grafiki.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=420, height=260)
artist = turtle.Turtle()

artist.speed(3)
artist.pensize(4)
artist.pencolor("purple")
artist.shape("turtle")
artist.forward(150)

screen.exitonclick()
```

REZULTAT

Gruba fioletowa linia narysowana przez żółwia o kształcie kursora w sylwetce żółwia

speed(3), pensize(4), pencolor("purple"), shape("turtle") — cztery ustawienia żółwia naraz

nastrojka_grafiki_kod.py

```
artist.speed(3)          # prędkość od 1 (wolno) do 10
(szybko), 0 – natychmiast
artist.pensize(4)        # Grubość linii w pikselach
artist.pencolor("purple") # Kolor linii
artist.shape("turtle")    # kształt wskaźnika – żółw zamiast
strzałki
```

speed(0) - Tryb artysty

Kiedy liczy się wynik, a nie sam proces malowania (np. w skomplikowanych wzorach jak mandala z rozdziału 6), wygodnie jest ustawić `artist.speed(0)` — żółw rysuje natychmiast, bez animacji ruchowej. Przeanalizujemy, dlaczego nie wpływa to na ostateczny rysunek w sekcji „speed, tracer i update”

Praktyka: Umożliwia personalizację grafiki Turtle
Ten sam laptop co Display as a System – to też obejmuje
Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/07-01/index.html)

Pióro i wypełnienie — to nie to samo

Kontur figury i jej obszar wewnętrzny malowane są różnymi poleceniami — ustalamy, które odpowiada za co.

Już użyliśmy i `pencolor()`, i `fillcolor()` osobno – ale ważne jest, aby jasno zrozumieć, czym ona jest **dwa różne** kolory odpowiadające za różne rzeczy.

Kontur to nie to samo co wypełnienie

Komenda	Za co odpowiada
`pencolor()`	kolor LINII — ramki figury
`fillcolor()`	KOLOR WYPEŁNIENIA—obszar wewnątrz kształtu (widoczny tylko pomiędzy begin_fill()/end_fill())
`color()`	ustawia oba naraz: color(kontur, wypełnienie)

tolko_pencolor.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=340, height=320)
artist = turtle.Turtle()
artist.pensize(4)
artist.pencolor("blue")

for _ in range(4):
    artist.forward(120)
    artist.right(90)

screen.exitonclick()
```

REZULTAT

Kwadrat narysowany tylko z niebieskim konturem, bez wypełnienia wewnątrz

pencolor(blue") bez fillcolor — tylko ścieżka, brak wypełnienia

tolko_pencolor_kod.py

```
artist.pencolor("blue")

for _ in range(4):
    artist.forward(120)
    artist.right(90)
```

Oba kolory przez jedną drużynę

color_vmeste.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=340, height=320)
artist = turtle.Turtle()
artist.pensize(4)
artist.color("blue", "gold")

artist.begin_fill()
for _ in range(4):
    artist.forward(120)
    artist.right(90)
artist.end_fill()

screen.exitonclick()
```

REZULTAT

Ген sam kwadrat z niebieskim konturem, ale teraz z złotym wypełnieniem w środku

color("blue", "gold") określa zarówno kontur, jak i wypełnienie jednym poleceniem

color_vmeste_kod.py

```
artist.color("blue", "gold")   # (konspekt, wypełnienie)

artist.begin_fill()
for _ in range(4):
    artist.forward(120)
    artist.right(90)
artist.end_fill()
```

color() bez dyskusji to poznać aktualne kolory

`artist.color()` zwraca parę bez argumentów `(pencolor, fillcolor)` — wygodne do debugowania, jeśli nie wiadomo, dlaczego figura wygląda inaczej niż oczekiwano.

Praktyka: Pióro i wypełnienie

Moduł turtle otwiera natywne okno Python – uruchamiane lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/07-11/index.html)

speed, tracer i update

Szybkość animacji i ostateczna geometria rysunku to zupełnie różne rzeczy. A dla naprawdę złożonych wzorów istnieje technika znacznie potężniejsza niż tylko speed(0).

speed() zmienia animację, a nie efekt

`artist.speed()` kontroluje, co **jak szybko** możesz zobaczyć rysunek, nie z tym, co ostatecznie masz. Ostateczny rysunek w `speed(1)` i na `speed(0)` wyjdzie geometrycznie **dokładnie to samo** — różnica jest tylko taka, czy widzisz sam proces, czy od razu gotowy wynik. Statyczny obraz fizycznie nie może pokazać różnicy w szybkości — widać ją tylko na żywo, przy uruchamianiu kodu.

tracer() i update(): pobierają partie

Nawet przy naprawdę złożonych rysunkach (setki i tysiące zespołów) `speed(0)` może wydawać się wolny — ekran odświeża się po **każdy** drużyny. `screen.tracer(0)` wyłączenia W ogóle aktualizacje pośrednie, oraz `screen.update()` aktualizuje ekran raz, ręcznie, gdy rysunek jest gotowy:

TRYB NORMALNY

→ ekran zaktualizowany

→ ekran zaktualizowany

→ ekran zaktualizowany

TRYB WSADOWY - tracer(0)

tracer(0)

drużyna, drużyna,

drużyna, ...

update() → ekran odświeżył się RAZ

tracer_batch.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=420, height=420)
screen.tracer(0)
artist = turtle.Turtle()
artist.hideturtle()
artist.pencolor("#5B24F9")

for i in range(36):
    artist.circle(80)
    artist.right(10)
screen.update()

screen.exitonclick()
```

REZULTAT

Misterny, symetryczny wzór 36 nałożonych na siebie okręgów tworzących wzór kwiatowy

36 kółek, każde obracane o 10° – narysowane natychmiast dzięki tracer(0) + update()

tracer_batch_kod.py

```
screen.tracer(0)   # wyłącz aktualizacje pośrednie

for i in range(36):
    artist.circle(80)
    artist.right(10)

screen.update()    # ostatnia aktualizacja i cały wzór pojawia
się naraz
```

Po co to potrzebne
Dla 10 poleceń różnica prawie niezauważalna. Ale dla tysięcy operacji `tracer(0)` + `update()` może sprawić, że rysowanie będzie wielokrotnie szybsze — ekran po prostu nie traci czasu na przerysowywanie po każdym kroku.

`delay()` - Samodzielne otoczenie

`screen.delay()` ustawia opóźnienie w milisekundach między Klatki animacji — koncepcyjnie podobne do `speed()`, ale Działa na poziomie ekranu, a nie pojedynczego żółwia:

delay_kod.py

```
screen.delay(15)   # milisekundy między klatkami animacji
print(screen.delay())  # bez argumentu zwróci aktualną wartość
```

Klasyczna pułapka debugująca

tracer(0) bez update() – pusty ekran
Jeśli wywołać `tracer(0)` coś narysować, ale zapomnieć `update()` — ekran pozostanie pusty! To jeden z najczęstszych i nieoczywistych błędów z Turtle — kod wykonuje się bez żadnego błędu, a na ekranie nic nie ma. Szczegółowo omówimy to w rozdziale o debugowaniu grafiki.

Praktyka: speed, tracer i update
Moduł turtle otwiera natywne okno Python – uruchamiane lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/07-12/index.html)

Kształty bez linii, okręgów, punktów

Trzy narzędzia z rozdziału 6, ale teraz z prawdziwym rezultatem — i mostem do głębszej geometrii koła przed nami.

Trzy narzędzia, które już spotkaliśmy mimochodem w rozdziale 6 – teraz zatrzymajmy się Umówić i przyjrzeć się im uważniej, zanim zagłębimy się w to.

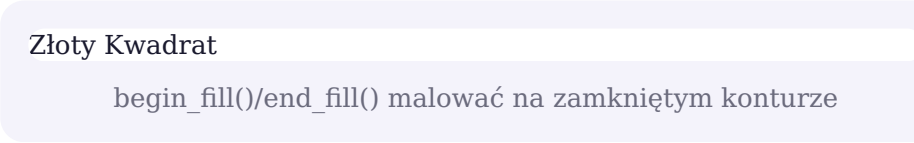
Figury bez linii: wypełnienie kolorem

Aby narysować nie tylko kontur, ale wypełnioną figurę, owiń rysowanie między `begin_fill()` oraz `end_fill()`:

zalivka.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=340, height=320)
artist = turtle.Turtle()
artist.fillcolor("gold")

artist.begin_fill()
for _ in range(4):
    artist.forward(100)
    artist.right(90)
artist.end_fill()

screen.exitonclick()
```

REZULTAT



zalivka_kod.py

```
artist.fillcolor("gold")
artist.begin_fill()
for _ in range(4):
    artist.forward(100)
    artist.right(90)
artist.end_fill()
```

Kręgi

Nie musisz rysować ręcznie koła po zbiór krótkich fragmentach — Turtle Zespół gotowy `circle(радиус)`:

okruzhnost.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=340, height=320)
artist = turtle.Turtle()
artist.pensize(3)
artist.pencolor("#5B24F9")

artist.circle(80)

screen.exitonclick()
```

REZULTAT

Fioletowe koło o promieniu 80 px

circle(80) jest gotowym poleceniem dla koła, bez ani jednego ręcznego obrotu

Punkty

`dot()` — mały (albo nie za mały) zakolorowany okrąg w bieżącej pozycji, bez ruchu żółwia:

tochka.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=260, height=200)
artist = turtle.Turtle()
artist.hideturtle()

artist.dot(50, "red")

screen.exitonclick()
```

REZULTAT

Jedna duża czerwona kropka o średnicy 50 pikseli

dot(50 „red”

tochka_kod.py

```
artist.dot(50, "red")   # kropka o średnicy 50, czerwona
```

Dużo głębiej przed nami

To tylko wstęp. W kolejnych trzech sekcjach szczegółowo przeanalizujemy geometrię koła – skąd pochodzi centrum, co oznacza ujemny promień i jak okrąg zamienia się w wielokąt.

Praktyka: Wypełnienie, Okręgi i Punkty

Moduł turtle otwiera natywne okno Python – uruchamiane lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/07-03/index.html)

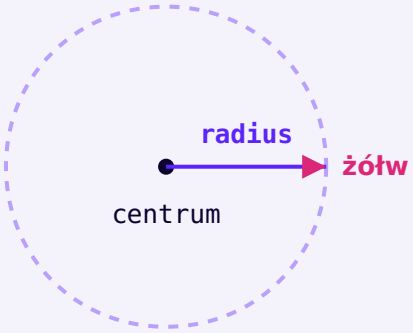
Geometria koła

circle() nie jest magią: ma precyzyjną geometrię, którą warto zrozumieć, a nie tylko zapamiętać jak zaklęcie.

`circle()` wygląda jak jedno proste polecenie, ale za nim stoi prawdziwa geometria — rozbierzmy ją naprawdę, a nie jak „magiczną” funkcję.

Gdzie naprawdę jest centrum

Oficjalne postępowanie `circle(radius)`: środek okręgu jest w odległości `radius` **lewo** od żółwie (względem kierunku kursu) – nie w obecnej pozycji żółwia, jeśli to możliwe Można by pomyśleć:



Środek okręgu znajduje się w odległości radius na lewo od żółwia — nie tam, gdzie sam żółw stoi

Na jednym końcu łuku stoi żółw

Jeśli narysujesz łuk (`extent` mniej niż 360°) — jeden z końców łuku zawsze będzie aktualną pozycją żółwia. To wygodne: nie musisz wcześniej obliczać, od którego łuku się zacznie.

Promień dodatni i ujemny

Promień dodatni rysuje okrąg przeciwnie do ruchu wskazówek zegara; promień ujemny rysuje wzdłuż Sentinel, o tym samym promieniu modulo:

krug_polozhitelnyj.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=340, height=320)
artist = turtle.Turtle()
artist.pensize(3)
artist.pencolor("#5B24F9")

artist.circle(80)

screen.exitonclick()
```

REZULTAT

Krąg narysowany fioletem przeciwnie do ruchu wskazówek zegara

circle(80) ma promień dodatni, przeciwnie do ruchu wskazówek zegara

krug_otricatelnyj.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=340, height=320)
artist = turtle.Turtle()
artist.pensize(3)
artist.pencolor("#DB2777")

artist.circle(-80)

screen.exitonclick()
```

REZULTAT

Okrąg o tej samej wielkości narysowany różem zgodnie z ruchem wskazówek zegara

circle(-80) ma ten sam promień modulo, ale zgodnie z ruchem wskazówek zegara

napravlenie_kod.py

```
artist.circle(80)    # przeciwnie do ruchu wskazówek zegara
artist.circle(-80)   #Clockryse — ten sam rozmiar, inny kierunek
```

Kierunek zmienia także kurs żółwia na końcu

Po pełnym okręgu kurs żółwia wraca do początkowego w obu przypadkach. Ale jeśli rysować łuk (nie pełne koło), końcowy kurs będzie RÓŻNY dla dodatniego i ujemnego promienia — żółw obróci się w przeciwnych kierunkach.

Praktyka: Geometria okręgu
Moduł turtle otwiera natywne okno Python – uruchamiane lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/07-13/index.html)

Łuki

circle() potrafi rysować nie tylko pełne okręgi, ale także ich części — łuki o określonym rozmiarze.

Wątek jest częścią kręgu. U `circle()` istnieje drugi, opcjonalny argument `extent` to ile stopni Trzeba narysować kręgi.

Od ćwiartki do pełnego koła

dugi_podpisannye.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=460, height=460)
artist = turtle.Turtle()
artist.speed(0)
artist.hideturtle()

arcs = [(60, "#5B24F9"), (90, "#DB2777"), (180, "#059669"),
(270, "#D97706"), (360, "#0D0230")]
for extent, color in arcs:
    artist.penup()
    artist.goto(0, 0)
    artist.setheading(0)
    artist.pendown()
    artist.pencolor(color)
    artist.pensize(3)
    artist.circle(90, extent)
    artist.penup()
    artist.write(f"  {extent}°", font=("Arial", 11, "bold"))

screen.exitonclick()
```

REZULTAT

Pięć kolorowych łuków o różnych długościach wychodzących z tego samego punktu, każdy oznaczony inną wartością extent stopni

Pięć łuków o tym samym promieniu, ale różnym extent – od 60° do pełnego koła 360°

dugi_kod.py

```
arcs = [(60, "#5B24F9"), (90, "#DB2777"), (180, "#059669"),
(270, "#D97706"), (360, "#0D0230")]
for extent, color in arcs:
    artist.pencolor(color)
    artist.circle(90, extent)
```

extent	Co jest rysowane
90°	ćwiartka koła
180°	Półkoło (półkoło)
270°	trzy czwarte obwodu
360° (lub nieokreślone)	pełne koło

Półokrąg + prosta = kopuła
Łącząc łuk z zwykłą linią, łatwo uzyskać figury złożone — na przykład półokrągłą kopułę domeczku lub łuk. W mini-projekcie „uśmieszek” na końcu rozdziału używamy właśnie takiego łuku — uśmiech będzie miał kąt 120°.

Praktyka: Łuki o różnych rozmiarach
Moduł turtle otwiera natywne okno Python – uruchamiane lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/07-04/index.html)

circle(steps=...) — z okręgu na wielokąt

Sekretem circle(): pod maską zawsze jest wielokąt — tylko czasem z bardzo dużą liczbą boków.

Oto sekret, który wiele wyjaśnia: `circle()` właściwie **nie** potrafi narysować idealne koło – przybliża je do właściwego wielokąt z małymi bokami zbiórm. Trzeci argument, `steps`, pozwala na jawne zarządzanie liczbą tych boków.

Od trójkąta do sześciokąta, ten sam promień

steps_3.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=320, height=300)
artist = turtle.Turtle()
artist.pensize(3)
artist.pencolor("#5B24F9")

artist.circle(80, steps=3)

screen.exitonclick()
```

REZULTAT

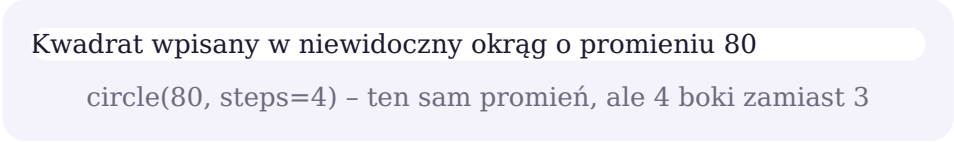
Trójkąt wpisany w niewidzialne koło o promieniu 80

circle(80, steps=3) to okrąg przybliżony trójkątem

steps_4.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=320, height=300)
artist = turtle.Turtle()
artist.pensize(3)
artist.pencolor("#5B24F9")

artist.circle(80, steps=4)

screen.exitonclick()
```

REZULTAT



steps_6.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=320, height=300)
artist = turtle.Turtle()
artist.pensize(3)
artist.pencolor("#5B24F9")

artist.circle(80, steps=6)

screen.exitonclick()
```

REZULTAT

Prawidłowy sześciokąt wpisany w niewidoczny okrąg o promieniu 80

circle(80, steps=6) – 6 boków, bardziej przypominających koło

circle_steps_kod.py

```
artist.circle(80, steps=3)   # trójkąt
artist.circle(80, steps=4)   # kwadrat
artist.circle(80, steps=6)   # sześciokąt
```

Most do formuły 360/n z rozdziału 6

To ten sam pomysł regularnego wielokąta, który zbudowaliśmy ręcznie w Rozdziale 6 aż do `forward()` / `right(360 / n)`. `circle(radius, steps=n)` jest po prostu krótszym sposobem uzyskania tego samego rezultatu, jeśli znasz pożądany promień opisanego okręgu.

A jeśli steps nie jest wskazane?

Jeśli `steps` nie jest określone, Turtle automatycznie wybiera wystarczająco dużą liczbę boków, aby wielokąt wyglądał jak gładkie koło — zazwyczaj kilka dziesiątek. Po prostu nie zauważamy, że to też jest wielokąt.

Praktyka: circle(steps=...) jako wielokąt

Moduł turtle otwiera natywne okno Python – uruchamiane lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/07-14/index.html)

Praktyka: przewidzieć wielokąt na podstawie steps (bez Turtle)

interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/07-24/index.html)

Stempelki i inne techniki

stamp() jest nowy pomysł: odcisk kształtu żółwia bez ruchu i bez linii.

Znaczek to nie linia ani kursor

`stamp()` zostawia odcisk obecnego kształtu żółwia na Bez rysowania linii jest istotna różnica w stosunku do normalnego ruchu. Przydatne dla różnych identyczne obiekty na ekranie (gwiazdy nieba, wrogowie w grze, znaki na wykresie).

shtampy.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=460, height=200)
artist = turtle.Turtle()
artist.shape("circle")
artist.color("#5B24F9")
artist.penup()

for _ in range(6):
    artist.stamp()
    artist.forward(60)

screen.exitonclick()
```

REZULTAT

Sześć identycznych fioletowych kółek do pieczątek w rzędzie, bez linii łączącej je

stamp() sześć razy z rzędu — pozostawia odcisk formy, nie rysując linii między nimi

shtampy_kod.py

```
artist.shape("circle")
artist.color("#5B24F9")
artist.penup()

for _ in range(6):
    artist.stamp()
    artist.forward(60)
```

stamp() zwraca numer — zapamiętaj go

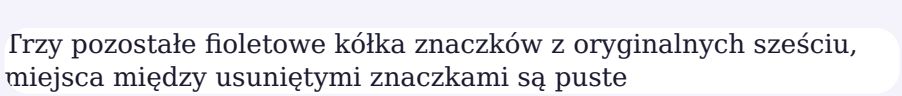
Każde wyzwanie `stamp()` zwraca liczbę całkowitą, która jest identyfikatorem danego znaczka. Bez niej nie możesz później usunąć konkretnego znaczka.

Usuń konkretne znaczki

clearstamp.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=460, height=200)
artist = turtle.Turtle()
artist.shape("circle")
artist.color("#5B24F9")
artist.penup()

stamp_ids = []
for _ in range(6):
    stamp_ids.append(artist.stamp())
    artist.forward(60)

artist.clearstamp(stamp_ids[0])
artist.clearstamp(stamp_ids[2])
artist.clearstamp(stamp_ids[4])

screen.exitonclick()
```

REZULTAT



Te same sześć znaczków, ale trzy z nich zostają usunięte po clearstamp(id)

clearstamp_kod.py

```
stamp_ids = []
for _ in range(6):
    stamp_ids.append(artist.stamp())
    artist.forward(60)

artist.clearstamp(stamp_ids[0])
artist.clearstamp(stamp_ids[2])
artist.clearstamp(stamp_ids[4])
```

Komenda	Co robi
`clearstamp(id)`	usuwa JEDEN konkretny znaczek przez jego numer
`clearstamps(n)`	usuwa pierwsze n znaczki (lub ostatnie n, jeśli n negatywne)
`clearstamps()`	usuwa WSZYSTKIE znaczki żółwi

Żółw chowaj się i pokazuj

skryt.py

```
artist.hideturtle()   # ukryj wskaźnik żółwia (sama linia pozostanie)
artist.showturtle()   # pokaż z powrotem
```

Dlaczego ukrywać żółwia?
W gotowych rysunkach wskaźnik żółwia może być wizualnie niepokojący. `hideturtle()` na końcu programu sprawia, że końcowy efekt jest czystszy — używamy tego we wszystkich mini-projektach rozdziału.

Cofnij ostatnią akcję: `undo()`

undo.py

```
artist.forward(100)
artist.undo()   # cofa ostatni ruch jak Ctrl+Z
```

Praktyka: zawiera stemple i inne metody
Ten sam laptop co sekcja Kształty, Kółka, Kropki – to też obejmuje

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/07-03/index.html)

Jak rysować tekst na ekranie

Turtle potrafi nie tylko linie — nauczymy ją podpisywać własne rysunki.

Turtle może nie tylko rysować linie, ale także wyświetlać tekst bezpośrednio na płótnie — za pomocą polecenia `write()`.

tekst_bazovyj.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=420, height=200)
artist = turtle.Turtle()
artist.hideturtle()

artist.write(„Cześć, Turtle!”, font=("Arial", 20, "normal"))

screen.exitonclick()
```

REZULTAT

Tekst „Hello, Turtle!”

write() wyświetla tekst na aktualnej pozycji żółwia

tekst_bazovyj_kod.py

```
artist.write(„Cześć, Turtle!”, font=("Arial", 20, "normal"))
```

Parametr `font` to zestaw trzech znaczeń razem: imię czcionka, rozmiar i styl (`"normal"` , `"bold"` lub `"italic"`). Taki grupowanie wielu wartości w jedną (*kondukt*) omówimy formalnie w rozdziale o Kolekcja — na razie wystarczy namalować ją zgodnie z próbką, jak w przykładzie.

Tekst pojawia się z obecnej pozycji

`write()` nie porusza żółwiem ani nie podnosi/opuszcza długopisu — po prostu drukuje tekst zaczynając od bieżącej współrzędnej. Aby umieścić tekst w konkretnym miejscu, najpierw przesuń się tam `goto()` (zwykle z `penup()` żeby nie wyznaczyć dodatkowej granicy).

Praktyka: write() i opcje czcionek

Moduł turtle otwiera natywne okno Python – uruchamiane lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/07-06/index.html)

Wyrównanie i czcionka

Tekst nie tylko może być wyświetlany, ale także kontrolować, jak rośnie z kropki — i jak wygląda.

U `write()` są dwie opcje, które zmieniają tekst z Etykiety „Gdziekolwiek pójdziesz”

align — skąd wywodzi się tekst

Miejsce, w którym przenieśliśmy żółwia, nie zawsze jest początkiem tekstu. Parametr `align` determinuje, jak tekst jest ustawiony WZGLĘDEM tego Punkty:

vyravnivanie.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=460, height=260)
artist = turtle.Turtle()
artist.hideturtle()
artist.penup()

for y, align, text in [(60, "left", "left"), (0, "center",
"center"), (-60, "right", "right")]:
    artist.goto(0, y)
    artist.dot(6, "#5B24F9")
    artist.write(text, align=align, font=("Arial", 16,
"normal"))

screen.exitonclick()
```

REZULTAT

Trzy linie tekstu (left, center, right), każda wyrównana inaczej do oznaczonej kropki na początku linii

Ten sam punkt (0, y), trzy różne align — tekst rozwija się w różnych kierunkach od niego

vyravnivanie_kod.py

```
for y, align, text in [(60, "left", "left"), (0, "center",
"center"), (-60, "right", "right")]:
    artist.goto(0, y)
    artist.dot(6, "#5B24F9")
    artist.write(text, align=align, font=("Arial", 16,
"normal"))
```

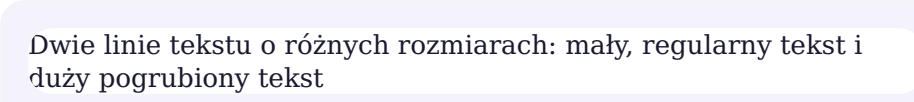
align	Gdzie punkt względem tekstu
`"left"` **(domyślnie)**	kropka to początek tekstu, tekst rośnie w prawo
`"center"`	punkt — środek tekstu
`"right"`	punkt to koniec tekstu, tekst rośnie w lewo

font — rozmiar i krój

shrift.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=420, height=220)
artist = turtle.Turtle()
artist.hideturtle()
artist.penup()

artist.goto(-180, 40)
artist.write("Arial 12 normal", font=("Arial", 12, "normal"))
artist.goto(-180, -30)
artist.write("Arial 26 bold", font=("Arial", 26, "bold"))

screen.exitonclick()
```

REZULTAT



Dwie różne czcionki i rozmiary – ta sama write() metoda, różne opcje font

shrift_kod.py

```
artist.write("Arial 12 normal", font=("Arial", 12, "normal"))
artist.write("Arial 26 bold", font=("Arial", 26, "bold"))
```

align=„center”
Gdy oznaczamy podziały osi współrzędnych lub środek kształtu, `align="center"` prawie zawsze wygląda schludniej — w przeciwnym razie podpis przesuwa się w stronę punktu, który opisuje.

Praktyka: Wyrównanie tekstu i czcionka
Moduł turtle otwiera natywne okno Python – uruchamiane lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/07-15/index.html)

Tekst i układ współrzędnych

Trzy narzędzia osobno to po prostu komendy. Razem tworzą one wykres współrzędnych ze podpisanymi proporcjami.

Przyłóżmy trzy narzędzia z tego rozdziału — `goto()`, `dot()` oraz `write()` — oraz współrzędne z Rozdział 6, aby uzyskać coś podobnego do prawdziwego wykresu z oznaczonymi znaczkami.

Prawdziwy harmonogram zaczyna się od planu

Plan jest prosty: narysować dwie osie, a następnie w kilku punktach wzdłuż każdej osi umieścić punkt-podział i podpisać go liczbą.

koordinatnyj_chart.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=460, height=460)
artist = turtle.Turtle()
artist.hideturtle()
artist.speed(0)

artist.penup()
artist.goto(-180, 0)
artist.pendown()
artist.goto(180, 0)
artist.penup()
artist.goto(0, -180)
artist.pendown()
artist.goto(0, 180)

artist.penup()
for value in [-100, -50, 50, 100]:
    artist.goto(value, 0)
    artist.dot(6, "#5B24F9")
    artist.goto(value, -20)
    artist.write(str(value), align="center", font=("Arial",
11, "normal"))
    artist.goto(0, value)
    artist.dot(6, "#5B24F9")
    artist.goto(15, value)
    artist.write(str(value), font=("Arial", 11, "normal"))
artist.goto(0, 0)
artist.dot(8, "#0D0230")
artist.goto(10, 6)
artist.write("0", font=("Arial", 11, "normal"))

screen.exitonclick()
```

REZULTAT

Płaszczyzna współrzędnych z osiami X i Y, punktami dzielenia co 50 pikseli oraz oznaczonymi liczbami -100, -50, 0, 50, 100

Osie z numerowanymi podziałkami — goto(), dot() i write() razem

koordinatnyj_chart_kod.py

```
# osie
artist.penup(); artist.goto(-180, 0); artist.pendown();
artist.goto(180, 0)
artist.penup(); artist.goto(0, -180); artist.pendown();
artist.goto(0, 180)

# podziały i podpisy
artist.penup()
for value in [-100, -50, 50, 100]:
    artist.goto(value, 0)
    artist.dot(6, "#5B24F9")
    artist.goto(value, -20)
    artist.write(str(value), align="center", font=("Arial",
11, "normal"))
    artist.goto(0, value)
    artist.dot(6, "#5B24F9")
    artist.goto(15, value)
    artist.write(str(value), font=("Arial", 11, "normal"))
```

Trzy rozdziały w jednym rysunku

Ten obraz nie jest przypadkiem: liczby i współrzędne z rozdziału 5, idea świata współrzędnych z rozdziału 6 oraz `write()` / `dot()` z tego rozdziału zostały połączone w jedno użyteczne narzędzie. Używamy tego samego pomysłu w miniprojekcie „coordinate target”

Praktyka: tekst i układ współrzędnych

Moduł turtle otwiera natywne okno Python – uruchamiane lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/07-16/index.html)

Turtle jako obiekt graficzny

Kształt, rozmiar i nachylenie kursora – ustawienia wyglądu niezależne od tego, gdzie i jak żółw faktycznie się porusza.

Do tej pory kształt żółwia był dla nas tylko „wskaźnikiem”

Formy wbudowane

galereya_form.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=560, height=180)
artist = turtle.Turtle()
artist.penup()
artist.pencolor("#5B24F9")

shapes = ["arrow", "turtle", "circle", "square", "triangle",
"classic"]
x = -220
for shape_name in shapes:
    artist.goto(x, 20)
    artist.shape(shape_name)
    artist.stamp()
    artist.goto(x, -25)
    artist.write(shape_name, align="center", font=("Arial",
10, "normal"))
    x += 85

screen.exitonclick()
```

REZULTAT

Wszystkie wbudowane kształty kursorów żółwi są arrow classic

galereya_form_kod.py

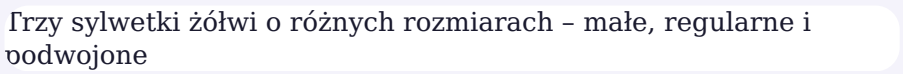
```
shapes = ["arrow", "turtle", "circle", "square", "triangle",
"classic"]
for shape_name in shapes:
    artist.shape(shape_name)
    artist.stamp()
```

shapesize() to rozmiar kursora, a nie rozmiar obrazu

razmer_formy.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=420, height=220)
artist = turtle.Turtle()
artist.shape("turtle")
artist.pencolor("#5B24F9")
artist.penup()

sizes = [(0.5, -140), (1, -20), (2, 140)]
for factor, x in sizes:
    artist.goto(x, 0)
    artist.shapesize(factor, factor)
    artist.stamp()

screen.exitonclick()
```

REZULTAT



shapesize() rozciąga widok kursora — 0.5×, 1× i 2× tej samej formy

razmer_formy_kod.py

```
artist.shape("turtle")
for factor in [0.5, 1, 2]:
    artist.shapesize(factor, factor)
    artist.stamp()
```

shapesize() nie zmienia współrzędnych
Powiększony żółw wygląda na większy, ale to tylko wygląd kursora — wszystkie współrzędne, obroty i odległości w kodzie pozostają dokładnie takie same jak wcześniej.

tilt() jest nachylenie widoku, a nie przebieg podróży

Tutaj ważne jest, aby rozróżnić dwa podobne, ale różne pojęcia:

Koncepcja	Co oznacza
heading (kurs)	tam, gdzie żółw się PORUSZA, zmienia się left()/right()/setheading()
tilt (pochylenie)	jak obrócony jest WID kursora na ekranie — zmienia to tylko tilt()/tiltangle()

naklon.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=420, height=220)
artist = turtle.Turtle()
artist.shape("arrow")
artist.shapesize(3, 3)
artist.pencolor("#5B24F9")
artist.penup()

artist.goto(-100, 0)
artist.setheading(0)
artist.stamp()

artist.goto(100, 0)
artist.tilt(45)
artist.stamp()

screen.exitonclick()
```

REZULTAT

Dwa wskaźniki strzałki o tym samym rozmiarze — lewy patrzy prosto, prawy jest odchylony pod kątem 45 stopni

tilt(45) obraca WIDOK kursora o 45°, nie zmieniając kierunku ruchu

naklon_kod.py

```
artist.shape("arrow")
artist.shapesize(3, 3)

artist.stamp()          # zwykły widok

artist.tilt(45)          # tylko obróć WIDOK, kurs ruchu się
nie zmienił
artist.stamp()
```

Dlaczego możesz go potrzebować

Wyobraź sobie samolot lub samochód w grze: może LATAĆ ściśle w prawo (tor jest stały), ale lekko się kołysać wizualnie – to jest różnica między heading a tilt. Do podstawowych rysunków treningowych rzadko jest tilt() potrzebne, ale warto wiedzieć, że kurs ruchu i widok to różne rzeczy.

Praktyka: Żółw jako obiekt graficzny

Moduł turtle otwiera natywne okno Python – uruchamiane lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/07-17/index.html)

Wiele żółwi

Jeden ekran może pomieścić dowolną liczbę żółwi, a każdy będzie miał swój własny, całkowicie niezależny stan.

Stworzyliśmy jednego żółwia do całego scenariusza. Ale nic nie stoi na przeszkodzie, by stworzyć kilka naraz — Każdy będzie miał własny, całkowicie niezależny państwo.

Jeden ekran, różne żółwie

dve_cherepashki.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=420, height=320)

alice = turtle.Turtle()
alice.color("#5B24F9")
alice.shape("turtle")
alice.penup()
alice.goto(-100, -80)
alice.pendown()
alice.setheading(70)
alice.forward(140)

bob = turtle.Turtle()
bob.color("#DB2777")
bob.shape("turtle")
bob.penup()
bob.goto(100, -80)
bob.pendown()
bob.setheading(110)
bob.forward(140)

screen.exitonclick()
```

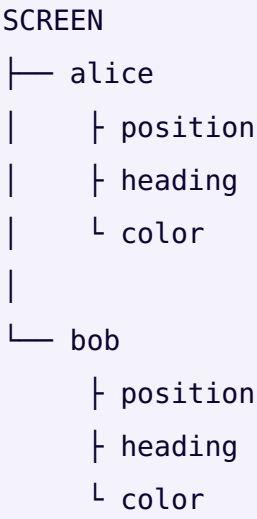
REZULTAT

Dwa segmenty o różnych kolorach z różnych punktów – fioletowy z Alice i różowy z Bob

alice i bob — dwa niezależne obiekty Turtle na jednym ekranie

dve_cherepashki_kod.py

```
alice = turtle.Turtle()
alice.color("#5B24F9")
alice.penup(); alice.goto(-100, -80); alice.pendown()
alice.setheading(70)
alice.forward(140)

bob = turtle.Turtle()
bob.color("#DB2777")
bob.penup(); bob.goto(100, -80); bob.pendown()
bob.setheading(110)
bob.forward(140)
```



Twój własny stan oznacza niezależne zachowanie

Zmiana kursu lub koloru `alice` NIE ma wpływu `bob` istnieją na tym samym ekranie, ale żyją całkowicie osobno. To jest istota tego, co `turtle.Turtle()` nie jest tylko funkcją, lecz fabryką niezależnych obiektów.

sravnenie_sostoyaniya.py

```
print(alice.position(), alice.heading())
print(bob.position(), bob.heading())
```

Nazwa tego jest przed nami

Później, w rozdziale o programowaniu obiektowym, nadamy temu zjawisku formalną nazwę i przeanalizujemy je znacznie głębiej. Na razie wystarczy samo obserwacje: każdy `Turtle()` jest odrębnym, niezależnym obiektem o własnym stanie.

Praktyka: Wiele żółwi

Moduł turtle otwiera natywne okno Python – uruchamiane lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/07-18/index.html)

clone() — kopiować stan

Nie "nowy żółw od podstaw", ale "dokładna kopia istniejącego" – i wtedy są całkowicie niezależne.

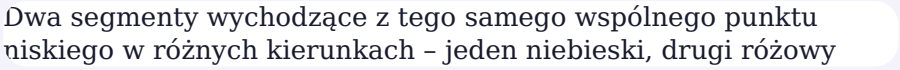
`clone()` inny sposób na zdobycie drugiego żółwia, ale fundamentalnie inne: nie "stwórz od zera", lecz "skopiuj stan istniejącego".

Kopia zaczyna się z tego samego miejsca

clone.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=420, height=320)

original = turtle.Turtle()
original.color("#5B24F9")
original.penup()
original.goto(0, -100)
original.setheading(90)

copy = original.clone()
copy.color("#DB2777")

original.pendown()
original.left(30)
original.forward(160)

copy.pendown()
copy.right(30)
copy.forward(160)

screen.exitonclick()
```

REZULTAT



original i copy zaczynają się od tego samego punktu z tym samym stanem, a następnie się rozchodzą

clone_kod.py

```
original = turtle.Turtle()
original.color("#5B24F9")
original.penup(); original.goto(0, -100); original.pendown()
original.setheading(90)

copy = original.clone()   # Ten sam kolor, ta sama pozycja, ta
sama szybkość
copy.color("#DB2777")     # zmieniać tylko kolor, aby odróżnić
go na zdjęciu

original.left(30)
original.forward(160)

copy.right(30)
copy.forward(160)
```

clone() ≠ link do tego samego żółwia

`copy` jest całkowicie NOWYM, niezależnym obiektem. Zaczyna się od tej samej pozycji, kursu i ustawień co `original` w czasie klonowania, ale dalsze zmiany u jednego żółwia nie wpływają na drugi, co już widzieliśmy w przypadku `alice` oraz `bob`.

Kiedy jest przydatny

Wyobraź sobie wzór z kilkoma identycznymi promieniami, ale obracanymi pod różnymi kątami – zamiast ustawiać każdego żółwia od zera, możesz ustawić „referencyjnego”

Praktyka: clone() i niezależne egzemplarze

Moduł turtle otwiera natywne okno Python – uruchamiane lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/07-19/index.html)

clear(), reset() i home()

Trzy polecenia brzmiące podobnie, ale każde wymazuje i przesuwa coś innego.

Trzy polecenia brzmiące podobnie, ale z różnymi efektami. Rozłóżmy je na części jedno po drugim, Używając TEGO SAMEGO scenariusza: narysuj pole, odsuń się, potem usuń z pola.

Komenda	Usuwa rysunek?	Czy to porusza żółwia?"
`clear()`	Tak	nie – żółw pozostaje tam, gdzie był
`reset()`	Tak	Tak – Zwroty do (0, 0) przy wskaźniku niewy-płacalności
`home()`	nie	tak – wraca do (0, 0), ale NIE dotyka rysunku

`clear()` usuwa rysunek, nie porusza żółwia

clear_effekt.py

```python
import turtle

screen = turtle.Screen()
screen.setup(width=340, height=260)
artist = turtle.Turtle()
artist.pencolor("#5B24F9")

for _ in range(4):
    artist.forward(100)
    artist.right(90)
artist.penup()
artist.goto(120, 80)
artist.clear()
# clear() usuwa rysunek, ale NIE dotyka pozycji żółwia –
# Punkt pojawi się w tym samym miejscu, gdzie go właśnie
przesunęliśmy
artist.dot(14, "#DB2777")

screen.exitonclick()
```

REZULTAT

Pusty ekran z różową kropką w prawym górnym rogu i czarnym kursorem żółwia obok niej — kwadrat został wymazany, ale pozycja żółwia pozostała ta sama

Po clear() rysunek zniknął, ale czubek pojawił się w tym samym miejscu, gdzie się przesunęliśmy PRZED czyszczeniem – pozycja się nie resetowała

clear_effekt_kod.py

```
for _ in range(4):
    artist.forward(100)
    artist.right(90)

artist.penup()
artist.goto(120, 80)
artist.clear()
# clear() usuwa rysunek, ale NIE dotyka pozycji żółwia —
# Punkt pojawi się w tym samym miejscu, gdzie go właśnie
przesunęliśmy
artist.dot(14, "#DB2777")
```

`reset()` — wymazuje rysunek I wraca do domu

reset_effekt.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=340, height=260)
artist = turtle.Turtle()
artist.pencolor("#5B24F9")

for _ in range(4):
    artist.forward(100)
    artist.right(90)
artist.penup()
artist.goto(120, 80)
artist.reset()
#reset() usuwa wzór I przywraca żółwia do (0, 0) –
# punkt pojawi się w miejscu początkowym, a nie tam, gdzie
byliśmy
artist.dot(14, "#DB2777")

screen.exitonclick()
```

REZULTAT

Pusty ekran z różową kropką dokładnie na środku — kwadrat zostaje wymazany, a żółw wraca do punktu początkowego

Po reset() wzór zniknął. A żółw wrócił do (0, 0) — kropka pojawiła się na środku

reset_effekt_kod.py

```
for _ in range(4):
    artist.forward(100)
    artist.right(90)

artist.penup()
artist.goto(120, 80)
artist.reset()
#reset() usuwa wzór I przywraca żółwia do (0, 0) –
# punkt pojawi się w miejscu początkowym, a nie tam, gdzie
byliśmy
artist.dot(14, "#DB2777")
```

`home()` — tylko ruch, rysunek pozostaje

Już się spotkaliśmy `home()` w rozdziale 6 to się przesuwa żółw w (0, 0) z domyślnym wskaźnikiem, ale w przeciwieństwie do `reset()`, NIE usuwa tego, co już zostało narysowane:

home_kod.py

```
artist.forward(100)
artist.left(45)
artist.forward(60)
artist.home()   # rysunek pozostaje na miejscu, tylko żółw
wraca do domu
```

Jak wybrać odpowiedni zespół

Trzeba wymazać rysunek, ale pozostać na miejscu? `clear()`. Czy musisz zaczynać od nowa – zarówno od rysunku, jak i od pozycji? `reset()`. Wystarczy po prostu wrócić do środka, nic nie wymazując? `home()`.

Praktyka: clear, reset, home

Moduł turtle otwiera natywne okno Python – uruchamiane lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/07-20/index.html)

Praktyka: Przewiduj efekt clear/reset/home (bez Turtle)
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/07-25/index.html)

Debugowanie grafiki

Bardziej zaawansowane grafiki wprowadzają nowe, konkretne sposoby popełniania błędów — analizujmy je według schematu „objawy → powodują → korekcję”

Grafika dodaje kilka nowych, bardziej zaawansowanych Turtle sposobów Aby zrobić coś źle, przeanalizujmy je według schematu „objawy → wywołać → korektę”

Objaw	Przyczyna	Poprawka
Okrąg jest narysowany w niewłaściwym kierunku niż się spodziewano	Promień ma nieprawidłowy znak — dodatni zamiast ujemnego lub odwrotnie	Sprawdź znak: promień dodatni jest przeciwny do ruchu wskazówek zegara, promień ujemny to
Tekst wcale nie pojawił się tam, gdzie się spodziewałeś	Zapomniałem przesunąć żółwia `goto()` / `penup()` wcześniej `write()` — tekst wyświetlany jest z pozycji CURRENT	Pan do pożądanego punktu przed rozpoczęciem `write()`
Fill nie przemalował, mimo że fillcolor() był ustawiony	Kontur pomiędzy `begin_fill()` oraz `end_fill()` nie zbliżył się – żółw nie wrócił do punktu startowego	Sprawdź, czy ścieżka tworzy ZAMKNIĘTĄ figurę — koniec pokrywa się z początkiem
RGB-color powoduje błąd typu „nieprawidłowa wartość koloru”	Liczby nie odpowiadają aktualnemu `colormode` — na przykład, `(255, 0, 0)` at `colormode(1.0)`	Sprawdź liczby w trybie aktywnym – 0-255 lub 0,0-1,0

Objaw	Przyczyna	Poprawka
Kod działa bez błędów, ale na ekranie nic się nie pojawia	Wezwany `screen.tracer(0)`, ale zapomniany `screen.update()` — ekran ani razu się nie odświeżył	Dodaj `screen.update()` po wszystkich komendach rysowania
Na ekranie pozostała pieczątka, chociaż wydawało się, że powinna zniknąć	`clearstamp()` wywołana z nieprawidłowym id, albo id w ogóle nie został zapisany podczas wywołania `stamp()`	Zachowaj id każdego stamp() w zmiennej/liście od razu przy tworzeniu
Dwa żółwie poruszają się w ten sam sposób, choć powinny się od siebie oddalić	Obie zmienne faktycznie odnoszą się do TEGO samego żółwia (np. `bob = alice` zamiast `bob = turtle.Turtle()`)	Twórz każdą żółwia osobnym wywołaniem `turtle.Turtle()` czy przez `clone()`
Kursor żółwia wygląda w złą stronę, mimo że porusza się poprawnie	Zdezorientowany `heading` (kierunek ruchu) i `tilt` (wizualna rotacja kursora) – są niezależne	heading zmieniać left()/right()/setheading(); wygląd – tylko tilt()/tiltangle()
Część obrazu nie jest widoczna, choć kod rysuje ją dokładnie	Współrzędne wyszły poza granice okna — nie jest ono nieskończone	Sprawdź `screen.setup()` / `screensize()` i trzymaj współrzędne w obrębie widocznego obszaru

`tracer(0)` bez `update()` — przeanalizujmy to szczegółowo

To tak powszechny i tak dyskretny błąd, że zasługuje na osobną analizę. Program działa całkowicie, bez żadnego błędu, ale ekran pozostaje nieskazitelnie pusty:

tracer_bug.py

```
screen.tracer(0)

for _ in range(4):
    artist.forward(100)
    artist.right(90)

# jeśli nie ma tu screen.update(), okno pozostanie puste!
```

Model mentalny

`tracer(0)` — to nie „wyłączyć rysowanie”, lecz „nie pokazywać go na ekranie, dopóki nie poproszą”. Rysowanie nadal się odbywa naprawdę, po prostu jest niewidoczne, dopóki nie zostanie wywołane `update()`.

Praktyka: Debugowanie grafiki (bez Turtle)

interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/07-23/index.html)

Mini-projekt – koło wewnątrz kwadratu

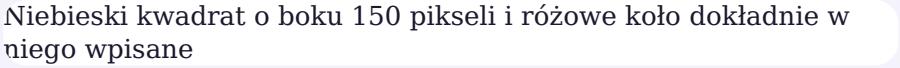
Najpierw wynik, planuj później, krok po kroku – naucz się świadomie budować kompozycję postaci, a nie metodą wyboru.

Co zbudujemy

okruzhnost_v_kvadrate.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=340, height=340)
artist = turtle.Turtle()
artist.pensize(3)

razmer = 150
artist.pencolor("#5B24F9")
artist.penup()
artist.goto(-razmer / 2, -razmer / 2)
artist.pendown()
for _ in range(4):
    artist.forward(razmer)
    artist.left(90)

artist.pencolor("#DB2777")
artist.penup()
artist.goto(0, -razmer / 2)
artist.setheading(0)
artist.pendown()
artist.circle(razmer / 2)

screen.exitonclick()
```

REZULTAT



Gotowy wynik: kwadrat o boku 150 i wpisane w niego koło

Rozłóżmy na części

Zanim napiszesz kod, podzielmy rysunek na proste kształty: kwadrat z bokiem `razmer` oraz okrąg o promieniu `razmer / 2`, wyryte dokładnie na środku.

Plan współrzędnych

Aby okrąg idealnie pasował do kwadratu, musisz zacząć rysować go od punktu na Do połowy dolnej strony kwadratu, nie od narożnika i nie od środka:

Co rysować	Od czego zacząć	Kurs na początek
Square	lewym dolnym rogu: (-razmer/2, -razmer/2)	0° (prawa)
Obwód	środku dolnej strony: (0, -razmer/2)	0° (prawa)

Krok 1: Tylko kwadrat

kvadrat_plan.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=340, height=340)
artist = turtle.Turtle()
artist.pensize(3)
artist.pencolor("#5B24F9")

razmer = 150
artist.penup()
artist.goto(-razmer / 2, -razmer / 2)
artist.pendown()
for _ in range(4):
    artist.forward(razmer)
    artist.left(90)

screen.exitonclick()
```

REZULTAT

Krok 1 – Na początku tylko kwadrat, bez koła

kvadrat_plan_kod.py

```
razmer = 150
artist.penup()
artist.goto(-razmer / 2, -razmer / 2)
artist.pendown()
for _ in range(4):
    artist.forward(razmer)
    artist.left(90)
```

Krok 2: Dodaj koło z inskrypcją

okruzhnost_v_kvadrate_kod.py

```
artist.pencolor("#DB2777")
artist.penup()
artist.goto(0, -razmer / 2)
artist.setheading(0)
artist.pendown()
artist.circle(razmer / 2)
```

Skąd wzięła się ta kropka

Promień wpisanego okręgu — dokładnie połowa boku kwadratu. A środek okręgu znajduje się po lewej stronie żółwia (patrz rozdział „Geometria okręgu”) — więc aby środek pokrywał się ze środkiem kwadratu, żółw musi zacząć rysować DOKŁADNIE z punktu na dolnej krawędzi, w odległości radius od środka.

Challenge

koncentricheskie.py

```python
import turtle

screen = turtle.Screen()
screen.setup(width=340, height=340)
artist = turtle.Turtle()
artist.pensize(3)

razmer = 150
artist.pencolor("#5B24F9")
artist.penup()
artist.goto(-razmer / 2, -razmer / 2)
artist.pendown()
for _ in range(4):
    artist.forward(razmer)
    artist.left(90)

colors = ["#DB2777", "#059669", "#D97706"]
for i, color in enumerate(colors):
    radius = razmer / 2 - i * 20
    artist.pencolor(color)
    artist.penup()
    artist.goto(0, -radius)
    artist.setheading(0)
    artist.pendown()
    artist.circle(radius)

screen.exitonclick()
```

REZULTAT

Ген sam kwadrat z trzema wielokolorowymi koncentrycznymi kółkami w środku zamiast jednego

Zadanie-wywołanie: kilka okręgów jeden w drugim zamiast jednego

★★ Zadanie samodzielne
Własny zestaw pierścionków
Zmień kod, aby uzyskać własny zestaw koncentrycznych okręgów — inne promienie, inne kolory.

★★ Zadanie samodzielne
Krąg w Heksagonie
Zamień kwadrat na sześciokąt z rozdziału 6 i wpisz w niego okrąg o odpowiednim promieniu.

Challenge
Gruby kontur
Zwiększ pensize() kwadratu i koła – jak zmienia się postrzeganie figury?

Praktyka: Koło z inskrypcją
Moduł turtle otwiera natywne okno Python – uruchamiane lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/07-07/index.html)

Zmieniamy kierunek rysowania

Kierunek, znak promienia i tryb mierzenia kątów — trzy powiązane idee, zebrane razem przed finalnymi mini-projektami.

Geometrię koła szczegółowo przeanalizowaliśmy już na początku rozdziału. Tutaj zebramy trzy powiązane elementy Pomysły razem—kierunek rysowania, znak promienia i tryb kąta—i przypinają je przed ostatnimi mini-projektami.

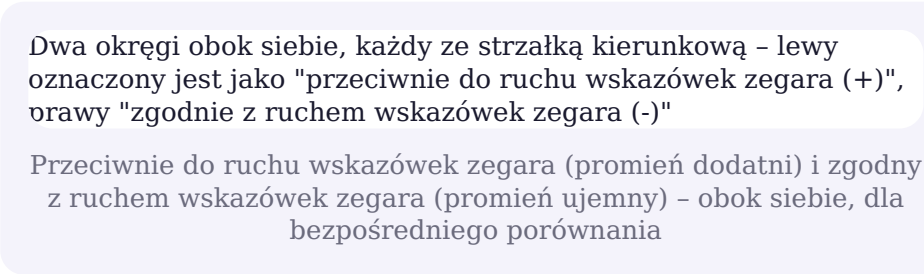
zgodnie z ruchem wskazówek zegara lub przeciwnie – Definiuje znak promienia

cw_ccw.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=560, height=340)
artist = turtle.Turtle()
artist.speed(0)
artist.pensize(3)

#circle(70) – środek na 70 px LEWO od toru: start poniżej
środka na 70,
# aby sam okrąg ustawił się wokoło (-150, 0), nie opierając się
o krawędź okna
artist.penup()
artist.goto(-150, -70)
artist.pendown()
artist.pencolor("#5B24F9")
artist.circle(70)
artist.penup()
artist.goto(-150, 115)
artist.write("circle(70)", align="center", font=("Arial", 13,
"bold"))
artist.goto(-150, 95)
artist.write(„przeciwnie do ruchu wskazówek zegara (+)”,
align="center", font=("Arial", 11, "normal"))

# circle(-70) jest centrum na 70 px PRAWEJ stronie trasy: start
to 70 stopni powyżej środka,
# tak że okrąg stoi wokół (150, 0), jest symetryczny względem
pierwszego
artist.goto(150, 70)
artist.pendown()
artist.pencolor("#DB2777")
artist.circle(-70)
artist.penup()
artist.goto(150, 115)
artist.write("circle(-70)", align="center", font=("Arial", 13,
"bold"))
artist.goto(150, 95)
artist.write(„zgodnie z ruchem wskazówek zegara (-)”,
align="center", font=("Arial", 11, "normal"))
```

```
screen.exitonclick()
```

REZULTAT



Przeciwnie do ruchu wskazówek zegara (promień dodatni) i zgodny z ruchem wskazówek zegara (promień ujemny) – obok siebie, dla bezpośredniego porównania

cw_ccw_kod.py

```
artist.circle(60)    # przeciwnie do ruchu wskazówek zegara – dodatni promień
artist.circle(-60)   # zgodnie z ruchem wskazówek zegara to promień ujemny
```

Tryby pomiaru kąta

Domyślnie Turtle mierzy kąty w stopniach. Jeśli z jakiegoś powodu wygodniej jest pracować w radianach (np. dla zgodności z formułami z modułu `math` z rozdziału 5) — są rozkazy `degrees()` oraz `radians()`:

radiany.py

```
import math

artist.radians()
artist.left(math.pi / 2)   # 90° obrotu wyrażonego w radianach

artist.degrees()            # wracając do zwykłych stopni
```

Nie zapomnij wrócić do stopni

Jeśli wywołać `radians()` i zapomnij `degrees()` na końcu wszystkie kolejne obroty w programie będą interpretowane jako radiany, a nie stopnie, co niemal zawsze prowadzi do nieoczekiwanych rezultatów.

Praktyka: obejmuje praktykę z kierunkiem rysowania
Ten sam notebook, co w rozdziale „Okrąg w kwadracie” — obejmuje również ten temat

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/07-07/index.html)

Mini-projekt – ponadczasowe zegary

Rozpoznawalna tarcza zegara — po prostu geometria: podział przez równe kąty, liczby, dwie wskazówki w ustalonych kierunkach.

Co zbudujemy

Timeless Watch to statyczna tarcza zegarka analogowego. Moduł `datetime` jeszcze nie potrzebujemy — strzałki zostaną naprawione kąt, po prostu po to, by pokazać, jak rozpoznawalny obiekt jest złożony z geometrii.

chasy_polnye.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=420, height=420)
artist = turtle.Turtle()
artist.speed(0)
artist.pencolor("#0D0230")
artist.pensize(3)

#circle(150) wyśrodkowuje okrąg 150 px LEWO od obecnego kursu –
# więc zacznij 150 niżej (0, -150), tak aby środek tarczy
spadał do (0, 0)
artist.penup()
artist.goto(0, -150)
artist.pendown()
artist.circle(150)

for hour in range(12):
    artist.penup()
    artist.goto(0, 0)
    artist.setheading(90 - hour * 30)
    artist.forward(135)
    artist.pendown()
    artist.forward(15)
    artist.penup()

for hour, label in [(0, "12"), (3, "3"), (6, "6"), (9, "9")]:
    artist.goto(0, 0)
    artist.setheading(90 - hour * 30)
    artist.forward(110)
    artist.write(label, align="center", font=("Arial", 16,
"bold"))

artist.pensize(5)
artist.pencolor("#5B24F9")
artist.goto(0, 0)
artist.setheading(90 - 12 * 30)
artist.pendown()
artist.forward(80)
artist.penup()

artist.pensize(3)
artist.pencolor("#DB2777")
```

```
artist.goto(0, 0)
artist.setheading(90 - 2 * 30)
artist.pendown()
artist.forward(120)

screen.exitonclick()
```

REZULTAT

Analogowy tarcza zegarka z 12-godzinnymi podziałami, z liczbami 12, 3, 6, 9 i dwoma wskazówkami wskazującymi stały czas

Gotowy wynik: tarcza zegara z podziałkami, liczbami 12/3/6/9 i dwiema wskazówkami

---

Plan geometryczny

12 podziałów pokrętła jest ułożonych w okręgu w regularnych odstępach – znana formuła `360 / 12 = 30` stopni między sąsiednimi godzinami. Ale w przeciwieństwie do zwykłego wielokąta, tutaj wygodniej ustawiać kąt każdego podziału ABSOLUTNIE, przez `setheading()`, a nie kumulować się na przemian:

ugol_deleniya.py

```
# godzina 0 (12 godzin) — w górę, 90°
# godzina 3            — kierunek w prawo, 0°
# godzina 6            — kierunek w dół, -90°
# godzina 9            — kierunek w lewo, 180°
# ogólnie: ugol = 90 - hour * 30
```

Krok 1: Koło i znaczniki

chasy_delenia.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=420, height=420)
artist = turtle.Turtle()
artist.speed(0)
artist.pencolor("#0D0230")
artist.pensize(3)

#circle(150) wyśrodkowuje okrąg 150 px LEWO od obecnego kursu –
# więc zacznij 150 niżej (0, -150), tak aby środek tarczy
spadał do (0, 0)
artist.penup()
artist.goto(0, -150)
artist.pendown()
artist.circle(150)

for hour in range(12):
    artist.penup()
    artist.goto(0, 0)
    artist.setheading(90 - hour * 30)
    artist.forward(135)
    artist.pendown()
    artist.forward(15)
    artist.penup()

screen.exitonclick()
```

REZULTAT

Okrągła tarcza z dwunastoma krótkimi kreskami wokół okręgu, bez cyfr i wskazówek

Krok 1 – Okrąg i 12 haków, bez liczb i strzałek

chasy_delenia_kod.py

```
artist.circle(150)

for hour in range(12):
    artist.penup()
    artist.goto(0, 0)
    artist.setheading(90 - hour * 30)
    artist.forward(135)
    artist.pendown()
    artist.forward(15)
    artist.penup()
```

Dlaczego goto(0, 0) przed każdą podziałem
Każda dywizja zaczyna się od środka, w przeciwnym razie żółw rysowałby podziały jedna po drugiej w okręgu, a nie jako niezależne promienie od środka. penup()/pendown() wokół każdego kroku zapewnia, że nie ma linii między podziałami.

Krok 2: Liczby i strzałki

chasy_chisla_strelki_kod.py

```
for hour, label in [(0, "12"), (3, "3"), (6, "6"), (9, "9")]:
    artist.goto(0, 0)
    artist.setheading(90 - hour * 30)
    artist.forward(110)
    artist.write(label, align="center", font=("Arial", 16,
"bold"))

# wskazówka godzinowa — krótsza, grubsza, ustawiona na 12
artist.pensize(5)
artist.pencolor("#5B24F9")
artist.goto(0, 0)
artist.setheading(90 - 12 * 30)
artist.pendown()
artist.forward(80)
artist.penup()

# wskazówka minutowa — dłuższa, cieńsza, przymocowana do 2
artist.pensize(3)
artist.pencolor("#DB2777")
artist.goto(0, 0)
artist.setheading(90 - 2 * 30)
```

```
artist.pendown()
artist.forward(120)
```

Końcowy efekt

chasy_polnye.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=420, height=420)
artist = turtle.Turtle()
artist.speed(0)
artist.pencolor("#0D0230")
artist.pensize(3)

#circle(150) wyśrodkowuje okrąg 150 px LEWO od obecnego kursu –
# więc zacznij 150 niżej (0, -150), tak aby środek tarczy
spadał do (0, 0)
artist.penup()
artist.goto(0, -150)
artist.pendown()
artist.circle(150)

for hour in range(12):
    artist.penup()
    artist.goto(0, 0)
    artist.setheading(90 - hour * 30)
    artist.forward(135)
    artist.pendown()
    artist.forward(15)
    artist.penup()

for hour, label in [(0, "12"), (3, "3"), (6, "6"), (9, "9")]:
    artist.goto(0, 0)
    artist.setheading(90 - hour * 30)
    artist.forward(110)
    artist.write(label, align="center", font=("Arial", 16,
"bold"))

artist.pensize(5)
artist.pencolor("#5B24F9")
artist.goto(0, 0)
artist.setheading(90 - 12 * 30)
artist.pendown()
artist.forward(80)
artist.penup()

artist.pensize(3)
artist.pencolor("#DB2777")
```

```
artist.goto(0, 0)
artist.setheading(90 - 2 * 30)
artist.pendown()
artist.forward(120)

screen.exitonclick()
```

REZULTAT

Analogowy tarcza zegarka z 12-godzinnymi podziałami, z liczbami 12, 3, 6, 9 i dwoma wskazówkami wskazującymi stały czas

Gotowy wynik: tarcza zegara z podziałkami, liczbami 12/3/6/9 i dwiema wskazówkami

Podstawowa praktyka
Wszystkie 12 liczb
Dodaj pozostałe 8 liczb (1, 2, 4, 5, 7, 8, 10, 11) według tej samej zasady.

★★ Zadanie samodzielne
Seconds
Dodaj trzecią, najcieńszą rękę – drugą wskazówkę – do dowolnego ustalonego kąta.

Challenge
Inne razy
Zmień kąty wskazówek godzinowych i minutowych, aby zegarek pokazywał inny (również stały) czas.

Praktyka: Zegar Ponadczasowy
Moduł turtle otwiera natywne okno Python – uruchamiane lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/07-21/index.html)

Mini-projekt – koordynować cel

Naprawdę mała infografika – cel, osiery, losowe kropki i podpis złożony ze znanych narzędzi.

Co zbudujemy

koordinatnaya_mishen.py

```
import turtle
import random

random.seed(4)
screen = turtle.Screen()
screen.setup(width=420, height=420)
artist = turtle.Turtle()
artist.speed(0)
artist.hideturtle()

rings = [(140, "#DC2626"), (100, "#FAFAFC"), (60, "#DC2626"),
(20, "#FAFAFC")]
for radius, color in rings:
    artist.penup()
    artist.goto(0, -radius)
    artist.pendown()
    artist.fillcolor(color)
    artist.pencolor("#0D0230")
    artist.begin_fill()
    artist.circle(radius)
    artist.end_fill()

artist.penup()
artist.goto(-160, 0)
artist.pendown()
artist.goto(160, 0)
artist.penup()
artist.goto(0, -160)
artist.pendown()
artist.goto(0, 160)

artist.penup()
for _ in range(6):
    x = random.randint(-120, 120)
    y = random.randint(-120, 120)
    artist.goto(x, y)
    artist.dot(10, "#059669")

artist.goto(-190, 170)
artist.write(„Skoordynuj cel”, font=("Arial", 13, "bold"))
```

```
screen.exitonclick()
```

REZULTAT

Cel z czerwono-białych koncentrycznych pierścieni z czarnym celownikiem, sześcioma zielonymi punktami życia i tytułem „Coordinate Target”

Gotowy wynik: koncentracyjna tarcza, krzyż osi, losowe trafienia i podpis

Pełnoprawna, mała infografika to nie tylko cel, ale cel z osiami współrzędnych, punktami „hits” `goto()`, losowe punkty z rozdziału 6 i Podpis przez `write()`.

Rozłóżmy na warstwy

Warstwa	Z czego się składa?	Co już jest znane
Pierścienie Celowe	4 zacienione okręgi o malejącym promieniu	begin_fill()/circle()/ end_fill()
Rozdroża	dwie linie przez środek	goto() z rozdziału 6
Punkty trafień	współrzędne losowe	random + dot() z rozdziału 6
Podpis	tekst w rogu	write() z tego rozdziału

Cały kod

koordinatnaya_mishen_kod.py

```
import random

random.seed(4)   # stały seed — punkty są takie same przy
każdym uruchomieniu

# pierścienie celu
rings = [(140, "#DC2626"), (100, "#FAFAFC"), (60, "#DC2626"),
```

```
(20, "#FAFAFC")]
for radius, color in rings:
    artist.penup()
    artist.goto(0, -radius)
    artist.pendown()
    artist.fillcolor(color)
    artist.begin_fill()
    artist.circle(radius)
    artist.end_fill()

# Celownik na osi
artist.penup(); artist.goto(-160, 0); artist.pendown();
artist.goto(160, 0)
artist.penup(); artist.goto(0, -160); artist.pendown();
artist.goto(0, 160)

# losowe punkty życia
artist.penup()
for _ in range(6):
    x = random.randint(-120, 120)
    y = random.randint(-120, 120)
    artist.goto(x, y)
    artist.dot(10, "#059669")

# podpis
artist.goto(-190, 170)
artist.write(„Skoordynuj cel”, font=("Arial", 13, "bold"))
```

Dlaczego random.seed(4)

Jak w rozdziale 6, ustalony seed sprawia, że obraz jest powtarzalny — przy każdym uruchomieniu punkty będą dokładnie w tych samych miejscach. Usuń ten wiersz, aby otrzymać nowy zestaw punktów przy każdym uruchomieniu.

★★ Zadanie samodzielne

Oznaczanie współrzędnych

Obok każdego punktu życia dodaj write() z jego współrzędnymi.

Challenge

Bullseye

Policz (wpisz do konsoli), ile losowych punktów wpadło do centralnego czerwonego pierścienia (promień < 60) – będziesz potrzebować wzoru na odległość z rozdziału 5.

Praktyka: cel współrzędnych
Moduł turtle otwiera natywne okno Python – uruchamiane lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/07-22/index.html)

Mini-projekt — emotikon

Zbieramy wszystkie techniki rozdziału w jednym przyjaznym rysunku, etapami — i podsumowujemy wyniki.

Co zbudujemy

Ostateczny mini-projekt rozdziału — i całego bloku o Turtle.. Złożymy przyjazną twarz z prymitywów, które omawialiśmy przez cały rozdział: wypełnienie, punkty, łuk.

smajlik.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=340, height=340)
artist = turtle.Turtle()
artist.speed(0)

artist.penup()
artist.goto(0, -100)
artist.pendown()
artist.fillcolor("yellow")
artist.pencolor("#0D0230")
artist.pensize(2)
artist.begin_fill()
artist.circle(100)
artist.end_fill()

artist.penup()
artist.goto(-35, 35)
artist.pendown()
artist.dot(20, "black")
artist.penup()
artist.goto(35, 35)
artist.pendown()
artist.dot(20, "black")

artist.penup()
artist.goto(-45, -25)
artist.setheading(-60)
artist.pendown()
artist.pensize(4)
artist.circle(55, 120)

artist.hideturtle()

screen.exitonclick()
```

REZULTAT

Żółty kółko z dwoma czarnymi kropkami – oczami i zakrzywionym uśmiechem z grubej linii

Efekt końcowy: żółta twarz, dwa oczy i łukowy uśmiech

Rozłożymy twarz na prymitywy

Szczegóły	Z czego się składa?	Tool
Face	zacienionym okręgu o promieniu 100	fillcolor() + circle() + begin_fill()/end_fill()
Oczy	dwa punkty	dot()
Uśmiech	łuk okręgu o promieniu 60, extent 120°	circle(radius, extent)

Plan współrzędnych

Szczegóły	Współrzędna/Kąt
Lewe Oko	(-35, 120)
Prawe oko	(35, 120)
Smile Start	(-50, 60), kurs -60°

Etap 1: Twarz

smajlik_lico.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=340, height=340)
artist = turtle.Turtle()
artist.speed(0)

# circle(100) centruje okrąg w 100 px PO LEWEJ od kursora —
zaczynamy na
# 100 poniżej (0, -100), aby twarz znalazła się dokładnie w
centrum okna (0, 0)
artist.penup()
artist.goto(0, -100)
artist.pendown()
artist.fillcolor("yellow")
artist.pencolor("#0D0230")
artist.pensize(2)
artist.begin_fill()
artist.circle(100)
artist.end_fill()

screen.exitonclick()
```

REZULTAT

Żółto malowane koło bez żadnych detali twarzy

Etap 1 - Twarz: Malowane koło

smajlik_lico_kod.py

```
artist.fillcolor("yellow")
artist.pencolor("#0D0230")
artist.pensize(2)

artist.begin_fill()
artist.circle(100)
artist.end_fill()
```

Etap 2: oczy

smajlik_glaza.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=340, height=340)
artist = turtle.Turtle()
artist.speed(0)

artist.penup()
artist.goto(0, -100)
artist.pendown()
artist.fillcolor("yellow")
artist.pencolor("#0D0230")
artist.pensize(2)
artist.begin_fill()
artist.circle(100)
artist.end_fill()

artist.penup()
artist.goto(-35, 35)
artist.pendown()
artist.dot(20, "black")
artist.penup()
artist.goto(35, 35)
artist.pendown()
artist.dot(20, "black")

screen.exitonclick()
```

REZULTAT

Ген sam żółty kółko, teraz z dwoma czarnymi kropkami na oku na górze

Krok 2 – Dodaj oczy: Dwie kropki

smajlik_glaza_kod.py

```
artist.penup()
artist.goto(-35, 120)
artist.pendown()
artist.dot(20, "black")

artist.penup()
artist.goto(35, 120)
artist.pendown()
artist.dot(20, "black")
```

Krok 3: Uśmiechnij się i po wszystkim

smajlik_ulybka_kod.py

```
artist.penup()
artist.goto(-50, 60)
artist.setheading(-60)
artist.pendown()
artist.pensize(4)
artist.circle(60, 120)

artist.hideturtle()
```

smajlik.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=340, height=340)
artist = turtle.Turtle()
artist.speed(0)

artist.penup()
artist.goto(0, -100)
artist.pendown()
artist.fillcolor("yellow")
artist.pencolor("#0D0230")
artist.pensize(2)
artist.begin_fill()
artist.circle(100)
artist.end_fill()

artist.penup()
artist.goto(-35, 35)
artist.pendown()
artist.dot(20, "black")
artist.penup()
artist.goto(35, 35)
artist.pendown()
artist.dot(20, "black")

artist.penup()
artist.goto(-45, -25)
artist.setheading(-60)
artist.pendown()
artist.pensize(4)
artist.circle(55, 120)

artist.hideturtle()

screen.exitonclick()
```

REZULTAT

Żółty kółko z dwoma czarnymi kropkami – oczami i zakrzywionym uśmiechem z grubej linii

Efekt końcowy: żółta twarz, dwa oczy i łukowy uśmiech

Podstawowa praktyka

Inny nastrój

Zmień łuk uśmiechu na łuk zmarszczonych brwi – odwróć kierunek extent.

★★ Zadanie samodzielne

Kolorowy emotikon

Zmień fillcolor twarzy na dowolny inny kolor.

Challenge

Policzki

Dodaj dwa małe różowe dot() pod oczami, aby uzyskać różowe policzki.

Praktyka: Rysowanie uśmiechniętej buźki

Moduł turtle otwiera natywne okno Python – uruchamiane lokalnie w VS Code, PyCharm lub Jupyter

`Otwórz praktykę →` (https://www.cartesianschool.org/pl/practice/07-09/index.html)

Podsumowanie rozdziału

Czego nauczyliśmy się w tym rozdziale

- Okno, płótno i świat współrzędnych — trzy różne rzeczy, które Turtle pozwala konfigurować osobno.
- colormode(1.0) i colormode(255) to dwa sposoby nagrywania tego samego RGB-colora.
- pencolor() koloruje kontur, fillcolor() — wypełnienie; color() ustawia oba naraz.
- speed() wpływa tylko na animację, a nie na ostateczną geometrię; tracer(0) + update() przyspieszają złożone grafiki, rysując je jako partię.
- circle() rysuje wpisany wielokąt — center znajduje się po lewej stronie żółwia, znak promienia określa kierunek, a steps steruje liczbą boków jawnie.
- stamp() pozostawia odcisk formy bez ruchu i bez linii.
- write() wyświetla tekst z konfigurowalnym align i font, zaczynając od bieżącej pozycji.
- Wiele turtle.Turtle() obiektów żyje niezależnie na tym samym ekranie, każdy w swoim własnym stanie; clone() kopiuje stan początkowy, a następnie obiekty się rozchodzą.
- clear(), reset() i home() usuwają i przesuwają różne kombinacje wzoru i pozycji.

co dalej

Turtle- grafika nie kończy się na tym na zawsze, ale wtedy trasa idzie nie tak nowe narzędzia językowe. W rozdziale 8 zaczniemy pracować z tekstem na poważnie—z łańcuch znakówe i słowa, które dotąd pojawiały się jedynie jako podpisy na ekranie.

ROZDZIAŁ 8 · STR. 388

Zabawa literami i słowami

Python linie do pełnej głębi: tworzenie i cytowanie, escape, wieloliniowe i raw-stringi, indeksy i slicery z wykresami wizualnymi, niezmienność, dziesiątki metod, formatowanie f-XQ0łańcuch znakówme, wprowadzanie użytkowników, Unicode, debugowanie — oraz osiem mini-projektów.

czym są struny?

Tekst w Python to stringi: sekwencja znaków, z którymi możesz pracować równie pewnie jak z liczbami.

Tekst to także dane

Prawie każdy program pracuje z tekstem: nazwa użytkownika, wiadomość na czacie, tytuł plik, adres lokalizacji, polecenie terminal. W Python tekst jest przechowywany w specjalnym rodzaju danych — **linia** (`str`). łańcuch znaków — to **sekwencja znaków**: litery, cyfry, znaki interpunkcyjne, sprady, Emotikony – pojawiają się jedna po drugiej w kolejności, jak koraliki na nici.

primery_strok.py

```
imya = „Ada”
soobshchenie = „Cześć, jak się masz?”
faijl = "otchet_2026.pdf"
adres = "https://cartesianschool.org"
komanda = "git commit -m fix"
```

Wszystko, co piszesz z klawiatury aż do `input()` (rozdział 8.15), również przychodzi jako łańcuch — nawet jeśli wygląda jak liczba.

Stwórz wiersz

Struna jest otoczona cudzysłowem — pojedynczym `'...'` lub podwójnie `"..."`. W Python są one całkowicie równoważne: wybierz dowolne, Najważniejsze to nie mieszać cudzysłowów w tej samej parze.

sozdaem_stroki.py

```
greeting = „Cześć”
name = 'Cartesian'
print(greeting, name)
#Hello Cartesian
```

Kiedy które cudzysłowy są wygodniejsze

Nie ma różnicy dla Python, ale czasami ludziom wygodniej jeden typ: jeśli w tekście będzie apostrof — `it's` — łatwiej jest owinąć go podwójnymi cudzysłowami; jeśli w środku jest mowa bezpośrednia w cudzysłowie — łatwiej jest owinąć go w pojedyncze cudzysłowy. Więcej na ten temat w sekcji 8.4.

Pusty łańcuch znaków

łańcuch znaków może nie zawierać pojedynczego znaku – to również pełnoprawny łańcuch znaków, tylko **pusty**: `""` lub `''`. Często służy jako wartość początkowa, do której później coś jest dodawane.

pustaya_stroka.py

```
pusto = ""
print(pusto)          # (nic nie zostanie wytworzone — łańcuch
znaków jest puste)
print(len(pusto))     # 0
```

type() – jak dowiedzieć się, co masz przed sobą łańcuch znaków

proverka_tipa.py

```
print(type("Python"))   # <class 'str'>
print(type(5))          # <class 'int'>
print(type("5"))
# <class 'str'> — cyfra w cudzysłowie i tak łańcuch znaków!
```

„5”
String `"5"` składa się z symbolu „pięć” `input()` .

Praktyka: tworzenie łańcuch znaków i type()
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/08-01/index.html)

Ucieczka: \n, \t i inni

Jak wstawić w łańcuch znaków znak, który nie ma własnego klawisza — i dlaczego to, co napisane w kodzie, nie zawsze zgadza się z tym, co wyświetlane jest na ekranie.

Znaki, których nie można wydrukować bezpośrednio

Czasem potrzebujesz znaku w linii, która nie ma własnego klucza — na przykład "end strings" (łącznik) lub "tabulator". Dla nich Python ma **sekwencje pomocnicze** (escape- sekwencja) to ukośnik wtwórczy `\` list plus:

Twórczość	oznacza
`\n`	znak nowej linii (new line)
`\t`	tabulator (wcięcie)
`\\`	sam backslash
`\"`	podwójny cudzysłów w łańcuch znaków w cudzysłowie
`\'`	pojedynczy apostrof wewnątrz łańcuch znaków w pojedynczych cudzysłowach

Tekst źródłowy vs wynik drukowany

kluczowa idea: Że ty **piszesz** w kodzie (z ukośnikami), i to, co naprawdę **wydrukowane** na ekranie (już bez nich), — dwa różne teksty:

escape_primer.py

```
print(„Pierwsza łańcuch znaków\nDruga łańcuch znaków”)
# Pierwszy łańcuch znaków
# Druga łańcuch znaków

print(„Imię:\tWage:”)
print(„Ada:\t28”)
# Imię: Wiek:
#Ada: 28
```

Symbol `\n` w kodzie źródłowym to DWA znaki (ukośnik i "n"), ale Python rozumie je jako JEDEN specjalny znak "łamanie linii", który Druk zamienia się w prawdziwy transfer — linia odwrotna nie jest już widoczna na ekranie.

Po co w ogóle tarcza

Ukośnik wpleczny to znak serwisowy Python wewnątrz łańcuch znaków. Escaping wyjaśnia Python: „to nie następuje polecenie, lecz ten konkretny znak.” `print()` .

repr() - Zobacz symbole serwisowe „tak, jak jest”

`repr()` drukuje łańcuch znaków tak, jak pojawia się w kodzie — z ze wszystkimi odwrotnymi cechami, a nie z ich efektem. To świetne narzędzie do debugowania, Do którego wrócimy w sekcji 8.17:

repr_vs_print.py

```
text = „Linia 1\nstroka 2”
print(text)          # wypisuje DWA prawdziwe łańcuch znaków
print(repr(text))    # 'Linia 1\nstroka 2' — widoczna \n jako
tekst
```

Podstawowa praktyka

Tablet z trzema liniami

Jednym print() komendy z \n wydrukuj trzy linie: "Imię: Ada", "Zawód: programist", "Rok: 1843".

Praktyka: uciekanie i repr()

interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/08-11/index.html)

Struny wieloliniowe i raw-struny

Dwa specjalne sposoby tworzenia łańcuch znaków tekstów: potrójne cudzysłowy dla tekstu wielolinijowego oraz litera r dla tekstu niewybieganego.

Wielolinijkowe łańcuch znaków: potrójne cudzysłowy

Zamiast wstawiać `\n` ręcznie możesz napisać tekst w kilku linijkach naraz — jeśli zawiniesz **potrójne cytaty** `"""` lub `'''`. Wszystkie podziały linii wewnątrz zachowały się tak, jak są:

mnogostrochnaya.py

```
poem = „Kod za kodem,
krok po kroku –
tak rodzi się
program.”
print(poem)
# Kod po kodzie,
#step krok po kroku –
# tak powstaje
# program.
```

Ostrożnie z wcięciami

Jeśli kod wewnątrz funkcji lub bloku jest wcięty, a potrójny łańcuch znaków zapisany z tym samym wcięciem, te space znajdą się WEWNĄTRZ tekstu — Python nie usuwa ich automatycznie. W prostych samouczkach, gdzie potrójny łańcuch znaków zaczyna się na początku linii kodu, nie stanowi to problemu; w prawdziwych projektach istnieją specjalne techniki na to (na przykład `textwrap`), których na razie nie potrzebujemy.

Surowe łańcuch znaków: r„...”

Czasem powinno być DUŻO tylnych linii w tekście, a ucieczka od każdego z nich jest żmudna i trudna do odczytania. Klasycznym przykładem jest ścieżka do pliku w Windows:

bez_raw.py

```
path = "C:\\Users\\Cartesian\\Documents"
print(path)
# C:\Users\Cartesian\Documents
```

Każda linia odwrotna musiała być podwojona. Jeśli położysz ją przed cudzysłowem otwierającym List `r` (od *raw* — «surowy»), Python przestaje rozumieć `\` jako początek sekwencji serwisowej i przyjmuje tekst dokładnie Jak jest napisane:

s_raw.py

```
path = r"C:\Users\Cartesian\Documents"
print(path)
# C:\Users\Cartesian\Documents — ten sam rezultat, ale bez
podwójnych kresek
```

Raw- struny usuwają „szum”
łańcuch znaków surowe często używa się do ścieżek do plików i do wzorców wyrażeń regularnych (temat na przyszłe rozdziały) — tam szczególnie dużo jest znaków odwrotnego ukośnika, i bez `r"..."` kod byłby trudny do odczytania.

Podstawowa praktyka
własną drogą
Wyprowadź ścieżkę r„D:\Projects\python-from-zero\notebooks”

Praktyka: potrójne i raw-łańcuchy
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/08-12/index.html)

W mojej linii są cudzysłowy! :O

Cudzysłów w łańcuch znaków tekstu, łącząc łańcuch znaków z operatorem + i powtarzając je z operatorem *.

W mojej linii są cudzysłowy! :O

Co powinienem zrobić, jeśli potrzebuję cudzysłowu tego samego typu, który oprawia łańcuch znaków tekstu w ramce? Najbardziej Prostym sposobem jest oprawienie łańcuch znaków w cudzysłowie **drugiej** typu:

kavychki.py

```
quote = „Powiedziała: 'Cześć!'”
quote2 = 'Ona powiedziała: „Cześć!”
print(quote)
print(quote2)
# Powiedziała: 'Cześć!'
# Powiedziała: „Cześć!”
```

Jeśli potrzebujesz tych samych cudzysłowów co w oprawie, są one pomijane (sekcja 8.2) backslash `\`:

ekranirovanie_kavychek.py

```
quote = „Powiedziała: 'Cześć!'”
print(quote)
# Powiedziała: „Cześć!”
```

Dołącz do kolejek: +

łańcuch znaków, podobnie jak liczby, można dodawać przez `+` — to nazywa się **łączenie** („klejenie”

konkatenaciya.py

```
first_name = „Ada”
last_name = „Lovelace”
full_name = first_name + " " + last_name
print(full_name)
# Ada Lovelace
```

łańcuch znaków z liczbą nie może być bezpośrednio dodany
`"Возраст: " + 10` spowoduje `TypeError` — Python nie wie, co miałeś na myśli: przykleić cyfrę „10” jako tekst czy dodać liczby. Trzeba jasno powiedzieć: albo `str(10)` lub f-łańcuch znaków (Rozdział 8.14).

tipeerror_i_ispravlenie.py

```
# print(„Wiek: ” 
print(„Wiek: ” + str(10))  # Wiek: 10 — Stały
```

Powtórz linijkę: *

łańcuch znaków można pomnożyć przez liczbę całkowitą — powtórzy się tyle razy:

povtorenie.py

```
stroka = "Ha" * 3
print(stroka)     # HaHaHa

razdelitel = "-" * 20
print(razdelitel) # --------------------
```

Konkatenacja w print()

Przypomnienie: `print()` istnieje łatwiejszy sposób na wycofanie się Wiele wartości oddzielonych przecinkami, bez jawności `+`:

print_konkatenaciya.py

```
print(first_name, last_name)          #Ada Lovelace – sam doda
miejsce
print(first_name + " " + last_name)   # Ada Lovelace – spację
trzeba dodawać samemu
```

Podstawowa praktyka

Rama z powtarzania

Użyj łańcuch znaków "=" pomnożonego przez 30, aby złożyć górne i dolne pola dla nagłówka "ROZDZIAŁ 8" i wyjść trzy linie: klatka, tytuł, ramka.

Praktyka: Cudzysłowy, łączenie i powtarzanie

Ten sam laptop, co w sekcji „Czym są łańcuch znaków?” — obejmuje też ten temat

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/08-01/index.html)

Długość linii: len()

Liczymy znaki w łańcuch znaków — w tym spacje — i przygotowujemy się do indeksów.

Ilu znaków jest w kolejce?

Funkcja `len()` („length”

dlina.py

```
word = "Python"
print(len(word))         # 6

phrase = „Python od zera”
print(len(phrase))       # 13 — spacje też się liczą!
```

Policz spacje samodzielnie

"Python od zera" to P-y-t-h-o-n (6) + spacja (1) + c (1) + spacja (1) + n-u-l-ya (4) = 13 znaków. Łatwo przypadkowo zapomnieć liczyć spację "okiem" – len() nigdy nie popełnia błędów.

Po co długość łańcuch znaków

`len()` przyda się w kolejnej części: aby zrozumieć jakie indeksy istnieją dla tego łańcuch znaków w ogóle. łańcuch znaków o długości `n` indeksy znaków zaczynają się od `0` do `n - 1` jest głównym powodem, aby znać długość linii z wyprzedzeniem.

dlina_i_indeks.py

```
word = "Python"
print(len(word))     # 6
print(word[len(word) - 1])   #n to ostatni znak, indeks
„długość minus jeden”
```

Podstawowa praktyka

Najdłuższe słowo

Dane trzy łańcuch znaków: "Python", "programowanie", "kod". Wyświetl długość każdego i określ (na podstawie wyświetlonych liczb), który łańcuch znaków jest najdłuższy.

Praktyka: len() i granice wiersza
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/08-13/index.html)

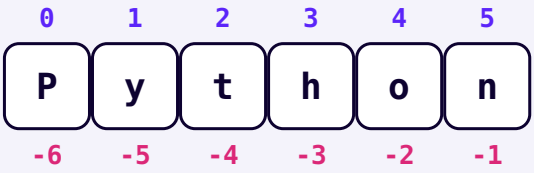
Indeksy wierszowe

Każdy znak w łańcuch znaków można pobrać po liczbie, zarówno na początku, jak i na końcu. Przeanalizujmy to bardzo jasno.

Kod pocztowy to numer miejsca

łańcuch znaków to łańcuch znaków, z których każdy ma swój własny **numer seryjny** — **indeks**. Indeks zapisuje się w kwadrat Nawiasy bezpośrednio po linijce: `строка[индекс]`.

Liczenie zaczyna się od zera!
To najważniejszy szczegół w tej sekcji. Pierwszy znak łańcuch znaków ma indeks **0**, nie 1. Drugi symbol to indeks 1. I tak dalej. Ta metoda liczenia nazywana jest **indeksowanie od zera** (zero-based indexing) – prawie wszystkie języki programowania go używają i szybko się do niego przyzwyczajasz.



„Python”

indeksy.py

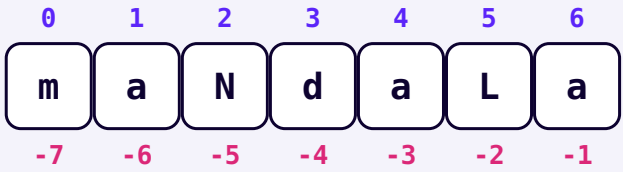
```
word = "Python"
print(word[0])   # P — pierwszy znak (indeks 0)
print(word[1])   # y — drugi znak (indeks 1)
print(word[5])   # n — szósty znak (indeks 5 — ostatni,
ponieważ jest 6 znaków)
```

Ujemne indeksy: liczenie od końca

Indeks `-1` oznacza „ostatni znak” `-2` jest „przedostatni”

otricatelnye_indeksy.py

```
word = "Python"
print(word[-1])   #n jest ostatnią postacią
print(word[-2])   # o — przedostatni
print(word[-6])   # P — ten sam symbol co word[0]!
```



Ten sam zasad działa także dla cyrylicy — „mandala”, 7 znaków, indeksy 0…6 i -7…-1

IndexError - Indeks Off-Line

`word[10]` dla sześcioliterowego słowa powoduje `IndexError: string index out of range`. Indeksy istnieją tylko od `0` do `len(word) - 1` (rozdział 8.5), i od `-1` do `-len(word)` od końca.

indexerror.py

```
word = "Python"
# word[10]   # IndexError: string index out of range
print(len(word))          #6 oznacza maksymalny indeks 5
```

Podstawowa praktyka

Pierwszy i ostatni list

Dla słowa „Cartesian”

Praktyka: Indeksy i indeksy ujemne
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/08-03/index.html)

Plasterki rzędowe

Slicer wyciąga cały fragment ze sznurka naraz, a nie tylko jeden znak. Przeanalizujmy granice przecięcia wizualnie, według pudełek.

Wycinanie — to kawałek łańcucha

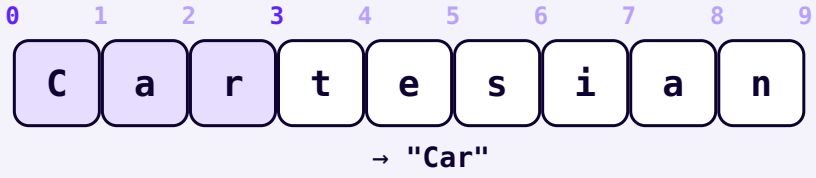
Wycinanie (slice) pobiera z łańcucha od razu kilka znaków z rzędu: `строка[start:stop]`. Proszę `start` — indeks, z którego pochodzi fragment **się zaczyna** (włącznie), ale `stop` to indeks, na którym wycink **końców** (NIE włącznie).

stop nie jest uwzględniany — i to celowo
Prawy obramowanie przekroju nie jest uwzględnione w wyniku. Brzmi to dziwnie, ale ma wygodną konsekwencję: długość kawałka jest zawsze równa `stop - start`. Wygodnie jest wizualizować indeksy nie jako liczby LITER, lecz jako liczby GRANIC MIĘDZY literami – spójrz na liczby nad poniższymi ramkami: są dokładnie pomiędzy.

Pierwsze trzy postacie

srez_pervye_tri.py

```
word = "Cartesian"
print(word[0:3])   # Car
```

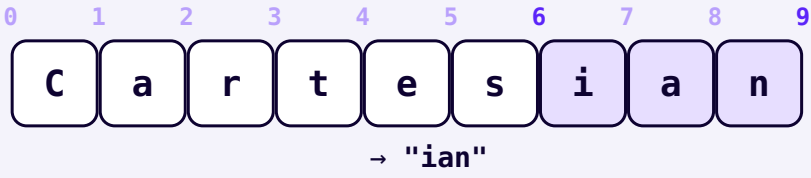


word[0:3] — od granicy 0 do granicy 3 (nie włączając jej)

Ostatnie trzy znaki

srez_poslednie_tri.py

```
word = "Cartesian"
print(word[6:9])   # ian
```

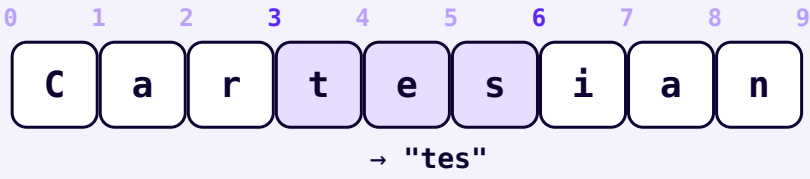


word[6:9] — od granicy 6 do końca linii (granica 9)

Fragment ze środka

srez_seredina.py

```
word = "Cartesian"
print(word[3:6])   # tes
```



word[3:6] – „średni” kawałek, od granicy 3 do granicy 6

Niepodane granice

Jeśli `start` pominąć – cięcie zaczyna się od samego początku Linie. Jeśli pominiesz `stop` — wycinek idzie aż do samego końca. Można pominąć oba granice naraz — wtedy powstaje kopia całego wiersza:

```
propushchennye_granicy.py
```

```
word = "Cartesian"
print(word[:3])    # Car — to samo co word[0:3]
print(word[6:])    #ian jest tym samym co word[6:9]
print(word[:])     #Cartesian - Kopia całego łańcuch znaków
```

Cut pitch

Fragment ma także trzeci, opcjonalny parametr — **krok**: `строка[start:stop:step]`. Krok 2 oznacza „bierz każdy drugi znak”:

```
shag_sreza.py
```

```
word = "Cartesian"
print(word[::2])    #Craea jest każda inna postać, od początku
do końca
```

Odwrócenie linii: [::-1]

Ujemny krok przechodzi linię w przeciwnym kierunku. Jeśli pominiesz i `start`, i `stop`, ale zrób krok `-1` — odwrócisz łańcuch znaków na odwrotną stronę:

```
razvorot_sreza.py
```

```
word = "Cartesian"
print(word[::-1])   # naisetraC
```

Najkrótszy sposób na rozciągnięcie sznurka

`строка[::-1]` to klasyczna technika Python. Trzeci parametr przekroju to „krok”

★★ Zadanie samodzielne

Środkowe litery w jednym kroku

Dla słowa „programowanie” uzyskaj przez wycinek każdą trzecią literę (krok 3), a następnie — samo słowo, odwrócone do tyłu.

Praktyka: Sekacze wierszy
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/08-14/index.html)

Rzędy nie mogą być zmieniane

Struny nie można zmienić „na miejscu”

Wiersza nie można zmienić „na miejscu”

Może się wydawać, że skoro wiemy, jak uzyskać symbol za pomocą indeksu (`word[0]`), można też **zastąpić** go tak samo Sposób. To nieprawda – linie w Python **niezmienne** (immutable):

popytka_izmenit.py

```
word = "Cat"
word[0] = "B"
# TypeError: 'str' object does not support item assignment
```

Dlaczego tak dziwnie?
To świadome urządzenie językowe, a nie ograniczenie. Niezmienne łańcuch znaków są łatwiejsze i bezpieczniejsze do przekazywania między różnymi częściami programu, więc możesz być pewien, że żaden inny fragment kodu „niezauważalnie”

Poprawny sposób: utworzyć nowy łańcuch

Zamiast zmieniać linię „na miejscu”

pravilnyj_sposob.py

```
word = "Cat"
word = "B" + word[1:]     # sklej "B" z "at" i uzyskaj nową
linię
print(word)                # Bat
```

Właśnie dlatego wszystkie metody łańcuch znaków — `upper()`, `replace()` i inni (Sekcje 8.9–8.10) – nie zmieniać oryginału linia, oraz **przywrócić nową**. Jeśli chcesz zapisać wynik, musisz go zapisać Jawnie przypisz zmienną do:

metody_ne_menyayut.py

```
text = "python"
text.upper()          # wynik stworzony, ale nigdzie nie
zapisany — i utracony!
print(text)            #python - Oryginalna łańcuch znaków
niezmieniona

text = text.upper()   # tak — zapisaliśmy wynik z powrotem do
text
print(text)            # PYTHON
```

Podstawowa praktyka

Zamiana pierwszej litery

Dla linii "home" zbuduj nową linię "dome", zastępując pierwszy znak przez cięcie i łączenie – jak pokazano powyżej dla "Cat" → "Bat".

Praktyka: niezmienność łańcuch znaków

Ten sam laptop co w sekcji Line Slicers – ten temat również porusza

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/08-14/index.html)

Metody strunowe to magia pracy z łańcuch znakówme!

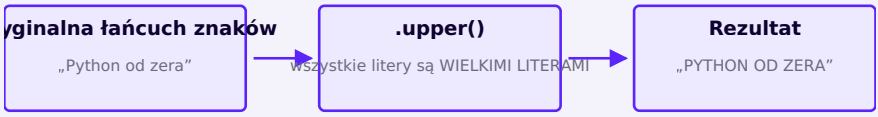
Akcje gotowe do wyjęcia z pudełka na łańcuch znakówme, wywoływane przez kropkę, od zmiany rejestru do czyszczenia przestrzeni.

Każdy rząd w Python ma wbudowane **metody** — gotowe działania, co można wykonać na nim za pomocą kropki: `строка.метод()`. Jak ustaliliśmy w rozdziale 8.8, wszystkie one zwracają NOWY łańcuch znaków, nie zmieniając oryginału.

Wielkie litery: upper, lower, title, capitalize, swapcase

registr.py

```
text = „Python od zera”
print(text.upper())      # PYTHON OD PODSTAW
print(text.lower())      # python od podstaw
print(text.title())      # Python Od Zera – pierwsza litera
każdego słowa wielka
print(text.capitalize()) # Python od zera – pisane wielkimi
literami tylko przy pierwszym słowie
print(text.swapcase())   # pYTHON OD ZERA – Przypadek każdej
litery odwróconej
```



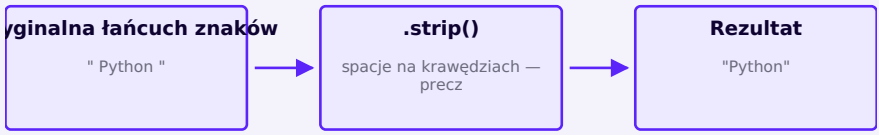
upper() zwraca NOWĄ linię (sekcja 8.8) – oryginalny zmienna się nie zmienia

Zbędne spacje: strip, lstrip, rstrip

Tekst wprowadzony przez użytkownika lub pobrany z pliku często zawiera przypadkowe spacje na brzegach. `strip()` usuwa ich z obu stron, `lstrip()` — tylko po lewej (left), `rstrip()` - Tylko prawo (right):

probely.py

```
text = "   Python   "
print(repr(text.strip()))    # 'Python'
print(repr(text.lstrip()))   # 'Python   '
print(repr(text.rstrip()))   # '   Python'
```



strip() usuwa spacje tylko na brzegach — nie wewnątrz łańcuch znaków

repr() znów się przydało
Używamy `repr()` z sekcji 8.2, aby zobaczyć przestrzenie na krawędziach linii – normalne `print()` wizualnie je „zjada” i nie zawsze jest jasne, czy pozostały.

★★ Zadanie samodzielne
Czyszczenie i formatowanie nazwy
Dana łańcuch znaków "Ada Lovelace" (z dodatkowymi spacjami i małymi literami). W jednym łańcuchu metod doprowadza się do formy "Ada Lovelace" – usuwa odstępy na krawędziach i stosuje wybraną metodę wielkości liter.

Praktyka: Metody rejestru i luk
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji
Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/08-04/index.html)

Metody łańcuchowe: wyszukiwanie i parsowanie

Zastępujemy podłańcuchy, dzielimy łańcuch znaków na części i ponownie sklejamy — a także szukamy podłańcuchów w tekście.

replace() - Wymiana podciągu

replace.py

```
text = „Kocham Java”
print(text.replace("Java", "Python"))   # Kocham Python
```

text.replace("Java", "Python") - zastępuje WSZYSTKIE wystąpienia podciągu

split() to podzielić linię na części

`split()` bez argumentów dzieli łańcuch znaków na spacji (oraz grupy spacji) przez lista pojedyncze słowa:

split.py

```
sentence = „Kot i Pies”
words = sentence.split()
print(words)          # ['kot', 'i', 'pies']
print(len(words))     #3 słowa
```

split() zamienia łańcuch znaków w lista słów (więcej na liście w rozdziale 10)

Możesz określić własny separator, na przykład przecink:

split_razdelitel.py

```
csv_row = „Ada, 28 lat, programistka”
fields = csv_row.split(",")
print(fields)          # ['Ada', '28', 'programmer']
```

join() jest operacją odwrotną: sklej lista w linię

join.py

```
words = [„kot”, „i”, „pies”]
result = "-".join(words)
print(result)          #cat-i-pies
```

join() to działanie odwrotne split(): które przykleja lista z powrotem do sznurka

count() — ile razy się pojawia

count.py

```
text = Mississippi
print(text.count(„c”))     # 4
print(text.count(„si”))    # 2
```

find() i index() – gdzie się pojawia

Obie metody szukają podciągu i zwracają indeks jego pierwszego znaku. Różnica polega na zachowaniu, Gdy łańcuch znaków nie zostaje znaleziona:

find_i_index.py

```
text = "Python"
print(text.find("th"))     #2 — Znalezione, zaczyna się od
indeksu 2
print(text.find("zz"))
# -1 — nie znaleziono, find() zwraca -1

print(text.index("th"))    # 2 — to samo co find(), jeśli
znaleziono
# text.index("zz") # ValueError: substring not found — index()
"zawiesza się" z błędem!
```

Kiedy wybrać co

Jeśli łańcuch znaków podciągu może nie być w tekście — użyj `find()` i sprawdź wynik dla `-1`. Jeśli jesteś pewien, że łańcuch znaków musi być (i chcesz od razu wiedzieć, czy nie), `index()` zgłasza błąd wyraźnie, zamiast cicho zwracać -1.

startswith() i endswith()

startswith_endswith.py

```
filename = "otchet_2026.pdf"
print(filename.startswith("otchet"))   # True
print(filename.endswith(".pdf"))       # True
print(filename.endswith(".docx"))      # False
```

★★ Zadanie samodzielne

Parsowanie nazwy pliku

Dla łańcuch znaków "report_final_v2.pdf" sprawdź endswith(". pdf"), znajdź pozycję znaku "_" metodą find() i przez split("_") uzyskaj lista części nazwy.

Praktyka: Wyszukiwanie i parsowanie łańcuchów

interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/08-16/index.html)

Metody weryfikacji: isalpha i inne

Metody odpowiadające na True/False tego, z czego składa się łańcuch znaków, są niezbędne przed przekształceniem tekstu w liczby.

Metody spełniające True lub False

Stringi mają całą grupę metod, które sprawdzają, Z CZEGO łańcuch znaków się składa i reagują `True` lub `False`. Szczególnie oni przydatne do weryfikacji danych wejściowych użytkownika (sekcja 8.15) – przed podjęciem próby Przekonwertuj tekst na liczby.

metody_proverki.py

```
print("Python".isalpha())      #True - Tylko listy
print("Python3".isalpha())     #False – jest ich liczba

print("2026".isdigit())        #True – Tylko liczby
print(„28 lat”.isdigit())      #False – jest spacja i litery

print("Python3".isalnum())     #True - Tylko litery i/lub cyfry
print("Python 3".isalnum())    #False – jest przestrzeń

print("   ".isspace())         #True - tylko znaki w odstępach
print("".isspace())
#False – Pusty łańcuch znaków się nie liczy
```

Metoda	Checki
`isalpha()`	tylko litery (dowolnego alfabetu)
`isdigit()`	tylko liczby 0-9
`isalnum()`	tylko litery i/lub cyfry
`isspace()`	tylko spacji/tabulatorów/łączników

isnumeric() i isdecimal() — krewni isdigit()
To rzadsze warianty tej samej idei: `isdecimal()` — najbardziej rygorystyczny (tylko zwykłe cyfry 0-9), `isdigit()` — nieco szerzej (obejmuje też niektóre specjalne znaki cyfrowe), `isnumeric()` jest najszerszy (rozumie także cyfry tekstowe, takie jak cyfry rzymskie w niektórych systemach pisma). Do nauki zadań i testowania normalnych danych wejściowych użytkownika `isdigit()` prawie zawsze wystarcza.

Praktyczne zastosowanie: weryfikacja przed przekształceniem

proverka_pered_int.py

```
age_text = "28"
if age_text.isdigit():
    age = int(age_text)
    print(f"Po 5 latach: {age + 5}")
else:
    print(„To nie wygląda jak liczba”)
```

Pełny operator `if` zostanie szczegółowo omówione w rozdziale 9 — ale już Teraz możesz zobaczyć, jak `isdigit()` pomaga zapobiegać awariom programu z błędem podczas próby `int()` z tekstu nienumerycznego.

Podstawowa praktyka
Jaki łańcuch znaków?
Dla trzech linii „Python3”, „2026”, „ ” wyświetl wynik isalpha(), isdigit() i isspace() dla każdej — i upewnij się, że rozumiesz, dlaczego otrzymano właśnie taki wynik.

Praktyka: Metody walidacji łańcuch znaków
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji
Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/08-17/index.html)

in, porównywanie łańcuchów i prawdziwość

Sprawdź, czy jedno łańcuch znaków pasuje do drugiego, porównaj linie między sobą i dowiedz się, kiedy łańcuch znaków zachowuje się jak „kłamstwo”

in i not in — sprawdzenie przynależności

Operator `in` sprawdza, czy jeden łańcuch znaków znajduje się w środku Druga to bardzo częsta i przydatna kontrola:

in_i_not_in.py

```
print("thon" in "Python")       #True – "thon" występuje
wewnątrz "Python"
print("java" in "Python")       # False
print("java" not in "Python")   # True – kontrola odwrotna
```

Porównanie stringów

łańcuch znaków można porównywać ze sobą za pomocą operatora `==` — Efekt zawsze będzie taki `True` lub `False`:

sravnenie_strok.py

```
print("Python" == "Python")     #True – struny idealnie się
pokrywają
print("Python" == "python")     # False – wielkość liter ma
znaczenie!
```

== porównuje wartość is porównuje obiekt

Do łańcuch znaków prawie zawsze potrzebny jest dokładnie `==`. Operator kamery `is` sprawdza, czy jest to pojedynczy obiekt w pamięci, znacznie rzadszy i bardziej szczegółowy przypadek, do którego wrócimy w Rozdziale 14.

Pusty łańcuch znaków to „kłamstwo”

W logicznym sprawdzaniu (pełne `if` — w rozdziale 9) pusty łańcuch znaków zachowuje się jak `False`, a każdy niepusty łańcuch znaków jako `True`:

pustaya_stroka_logika.py

```
print(bool(""))         # False
print(bool("Python"))   # True
print(bool(" "))        # True — spacja — to też znak!
```

Podstawowa praktyka

Walidacja domeny

Dla adresu "support@cartesianschool.org" sprawdź in, czy zawiera "@" i czy zawiera ". ru».

Praktyka: in, porównania i prawda strun

Ten sam laptop co w sekcji Indeksy wierszy – porusza ten temat również

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/08-03/index.html)

Iteracja przez linię w pętli

Struna to łańcuch znaków, przez które można przejść jeden po drugim —pierwsze wprowadzenie do idei pętli.

wyprzedzanie faktów

Jeszcze nie przerabialiśmy szczegółowo pętli — to temat rozdziału 9. Tutaj wystarczy rozumieć `for ch in text:` dosłownie: „powtórz blok kodu dla każdego znaku text kolejno, za każdym razem umieszczając kolejny znak w zmiennej ch”. Pełna konstrukcja cyklu `for` omówię później.

Struna to sekwencja, przez którą można przejść

Ponieważ łańcuch znaków jest zestawem znaków w kolejności (sekcja 8.1), możesz go używać **to uporządkować** — czyli przyjrzyć się każdemu symbolowi po kolei:

perebor_stroki.py

```
word = "Python"
for ch in word:
    print(ch)
# P
# y
# t
# h
# o
# n
```

Studium przypadku: Liczenie samogłosek

schitaem_glasnye.py

```
text = „programowanie”
glasnye = „aeeyoueyyuya”
count = 0
for ch in text:
    if ch in glasnye:
        count += 1
print(f"Samogłoski: {count}")
# Samogłosek: 7
```

Tutaj już trochę się przyglądaliśmy i `if` (Rozdział 9), ale Sama idea „przechodzenia przez każdy symbol i robienia z nim czegoś”

Podstawowa praktyka

Liczymy konkretną literę

Powtórz linię "mississippi" w pętli for i policz liczbę powtórzeń litery "i" — ręcznie, bez użycia count() z sekcji 8.10.

Praktyka: Iteracja na łańcuch znaków w pętli

interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/08-18/index.html)

Formatowanie łańcuch znaków

f-strings to główne nowoczesne narzędzie do wstawiania wartości do tekstu. Przyjrzyjmy się im szczegółowo: od prostej podstawienia po wyrównanie kolumn.

Już korzystaliśmy z f-łańcuch znaków począwszy od rozdziału 4 — teraz dokładnie przeanalizujemy formatowanie tekstu.

f-łańcuch znaków jest prostą podstawieniem

Cudzysłowem otwierającym poprzedza się litera `f`, ale w środku Aparaty kręcone `{}` — nazwa zmiennej. Python samo wstawi tam wartość:

f_stroka_prostaya.py

```
name = „Ada”
print(f"Cześć, {name}!")
# Cześć, Ada!
```

Wewnątrz {} można pisać wyrażenia

To nie jest tylko podstawienie imienia – KAŻDE Python- wyrażenie działa w nawiasach: Arytmetyka, metoda wywoływania, cokolwiek:

f_stroka_vyrazheniya.py

```
age = 5
print(f"Za rok będzie {age + 1}.")           # Za rok będzie
ich 6.

name = "cartesian"
print(f"Wielka litera: {name.upper()}")            # Wielkimi
literami: CARTESIAN
```

Formatowanie liczb: precyzja i separatory

Po wartości można dodać dwukropek i **specyfikacja formatu** — jak Aby wyświetlić numer:

format_specifikatory.py

```
pi = 3.14159265
print(f"Pi zaokrąglone: {pi:.2f}")
# Pi zaokrąglone: 3,14 – 2 miejsca po przecinku
print(f"{1234567:,}")                #1,234,567 – Separator
Tysięcy
print(f"{0.4567:.1%}")               #45,7% – udział procentowy
```

`.2f` brzmi tak: `f` jest liczbą z zmiennoprzecinkowe, `.2` — zaokrąglić do 2 miejsc po przecinku.

Wyrównanie i szerokość

Liczba po dwukropku (bez kropki) określa minimalną szerokość pola — wygodne do wyrównywania tekstu w kolumnie:

vyravnivanie.py

```
for name, score in [(„Ada”, 98), („Alan”, 87), („Grace”, 100)]:
    print(f"{name:<8} {score:>5}")
# Ada 98
#Alan 87
# Grace     100
```

Specyfikator	Znaczenie
{value:<10}	po lewej szerokość pola wynosi 10
{value:>10}	po prawej krawędzi, szerokość pola 10
{value:^10}	na środku, szerokość 10 kwadratów
{value:.2f}	liczba ułamkowa, 2 miejsca po przecinku
{value:,}	separator przecinków tysiąca przecinków

Technika debugowania: f„{{value=}}”

Jeśli umieścisz znak wewnątrz linii f- po wyrażeniu `=` — Python wyświetli zarówno wyrażenie, jak i jego wartość. Doskonałe do szybkiego sprawdzenia: `print(f"{{age=}}")` wycofa się `age=28` - bez ręcznego wpisywania „age =”

Formatowanie łańcuch znaków: % → . format() → Struny f-

Podejście klasyczne (% i . format())

```
name = "Cartesian"
age = 5

# operator % – najstarszy sposób
print(„Witaj %s! Masz %d lat.” % (name, age))

#. format() jest nowszy, ale wciąż rozwlekły
print(„Cześć, {}! Masz {} lat.”.format(name, age))
```

Modern Python 3.14 (struny f-strings)

```
name = "Cartesian"
age = 5

#f-łańcuch znaków wartości znajdują się dokładnie wewnątrz {}
print(f"Cześć, {name}! Ty {age} lat.")

# działają także wyrażenia, nie tylko zmienne:
print(f"Za rok będzie {age + 1}.")
```

Co używać dzisiaj: struny f-struny – pojawiły się w Python 3.6 i od tego czasu są standardem. Operator `%` , najstarsza metoda odziedziczona z języka C, nadal znajduje się w starym kodzie. `.format()` jest pośrednim krokiem między nimi. f-linie są najlepsze do czytania: wartość jest widoczna dokładnie tam, gdzie zostanie podstawiona, i możesz zapisać dowolne wyrażenia, nie tylko nazwy zmiennych.

★★ Zadanie samodzielne

Zaznacz „na stole”

Wyjść linijka "Łącznie: 1234,5 → 1 234,50 ₽" (używając: ,,2f) dla wartości 1234,5), a osobno - "Zniżka: 15,0%" dla udziału 0,15 (stosując :,1%).

Praktyka: %, . format() i struny f-struny

interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/08-06/index.html)

Otrzymywanie danych wejściowych od użytkowników

Początek automatyzacji: Program, który pyta i odpowiada na odpowiedź.

Do tej pory wszystkie dane w programach były „podłączone”

vvod.py

```
name = input(„Jak masz na imię?”)
print(f"Cześć, {name}!")

# Dialog w terminalu:
# Jak masz na imię? Ada
# Cześć, Ada!
```

`input()` zatrzymuje program i czeka, aż użytkownik wpisze tekst i naciśnie Enter — to, co wpisał, wraca jako zwykły łańcuch znaków.

input() w praktyce przeglądarkowej — naprawdę interaktywny
Nasz przeglądarkowy laptop ćwiczeniowy wie, jak naprawdę czekać na Twoją odpowiedź `input()` jak zwykły `.py` -plik, uruchomiony na komputerze: program zatrzymuje się, pokazuje wskazówkę i czeka, aż wpiszesz tekst.

Prosta zasada: input → łańcuch znaków → przekonwertować, jeśli potrzebujesz liczby

`input()` **zawsze** zwraca niczkę—nawet jeśli użytkownik wpisał numer. Aby obliczyć coś o tej wartości, musisz Przemiana (tak jak w rozdziale 4):

vvod_chisla.py

```
age_text = input(„Ile masz lat? ”)
print(type(age_text))
# <class 'str'> — nawet jeśli wpisane jest „28”

age = int(age_text)
print(f"Za 5 lat będziesz {age + 5}.")
```

Częsty błąd powtarany przez początkujących
Jeśli zapomnisz o `int()` i napisać `age + 5`, gdzie `age` jest łańcuch znaków `input()`, Python to zdradzi `TypeError`: nie można bezpośrednio dodawać łańcuch znaków i liczby.

Jeszcze kilka dialogów

dialogi.py

```
city = input(„W jakim mieście mieszkasz?”)
print(f"{city} to świetne miasto!")

height_text = input(„Czy twój wzrost jest w cm?”)
height = float(height_text)
print(f"Twój wzrost w metrach: {height / 100:.2f}")
```

Podstawowa praktyka
Witamy czekiem
Po input() poproś o nazwę, a następnie sprawdź (przez isalpha() z sekcji 8.11), czy składa się tylko z liter – wyjdź inny tekst w zależności od wyniku.

Praktyka: input() i konwersja typów
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji
Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/08-07/index.html)

Cyrylica, unicode i emoji

Strings Python równie dobrze rozumieją każdy język na świecie — a nawet emotikony.

Python współpracuje z każdym językiem

Strings Python mogą przechowywać symbole DOWOLNEGO języka na świecie – cyrylicy, łaciny, hieroglifów, Nawet emotikony dzięki systemowi kodowania **Unicode** (Unicode) wbudowany w Python od razu po wyjęciu z pudełka. Nie musisz nic konfigurować:

unikod_primery.py

```
russkij = „Witaj, świecie!”
english = "Hello, world!"
smajlik = „Python jest    i    ”
print(russkij)
print(english)
print(smajlik)
```

Długość linii emoji

dlina_s_emodzi.py

```
text = „kod   ”
print(len(text))    #6 — "do", "o", "y", ","  " — ale spacja +
emotikony liczą się osobno
```

Wystarczy znać na tym poziomie

Pełna teoria kodowania (UTF-8, bajty, tabele znaków) — temat na przyszłe, bardziej zaawansowane rozdziały. Obecnie ważne jest tylko jedno: w Python 3 cyrylica, łacina i emoji działają równie dobrze i bez specjalnej konfiguracji — można swobodnie pisać kod i teksty programów po rosyjsku.

Wszystko, co już wiemy, działa tutaj

metody_na_kirillice.py

```
text = „Witaj, Świecie”
print(text.upper())     #HELLO ŚWIAT
print(text.lower())     #hello świat
print(text[::-1])       # riM ,tveirP
```

Podstawowa praktyka

Pozdrowienie emotikoną

Utwórz łańcuch znaków za pomocą konkatenacji z trzech części: „Witamy” + spacja + emoji według wyboru i wypisz jego długość przez len().

Praktyka: łańcuch znaków w różnych językach
Ten sam laptop, co w sekcji „Czym są łańcuch znaków?” — obejmuje też ten temat

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/08-01/index.html)

Debugowanie problemów z łańcuchami

Struny często wyglądają poprawnie dla oka, ale pękają przez jeden niewidzialny znak. Naucz się znajdować i naprawiać takie błędy.

Struny są źródłem szczególnie podstępnych błędów: często wyglądają prosto na oko, oraz Złamać się przez jeden niewidzialny symbol. Przyjrzyjmy się najczęstszym sytuacjom.

Przestrzenie niewidzialne

nevidimye_probely.py

```
password = „sekret”     #random miejsce na końcu!
print(password == „tajemnica”)   # False — i nie da się tego rozpoznać na oko.
print(repr(password))          # 'sekret' — repr() daje przestrzeń
```

repr() jest twoim głównym narzędziem
Widzieliśmy już repr() w sekcji 8.2 – pokazuje on łańcuch znaków „taki, jaki jest”

Zapomniany cudzysłow

zabytaya_kavychka.py

```
# text = "Witaj świecie! # Zapomniałem cytatu końcowego
# SyntaxError: unterminated string literal
```

Niepoprawne maskowanie

nepravilnoe_ekranirovanie.py

```
#path=„C:\new_folder”
print("C:\new_folder")
# C:
# ew_folder wcale nie jest tym, czego chcieli

print(r"C:\new_folder")
# C:\new_folder — raw-łańcuch znaków (rozdział 8.3) rozwiązuje
problem
```

Łączenie łańcuch znaków z liczbą

stroka_plyus_chislo.py

```
score = 95
# print(„Wynik: ”
print(„Wynik: ” + str(score))   # poprawiono — jawna konwersja
print(f"Wynik: {score}")        # jeszcze lepiej — f-łańcuch
znaków samo przekształca
```

find() vs index() gdy nie ma podciągu

find_vs_index_oshibka.py

```
text = "Python"
print(text.find("java"))    # -1 — Cicho zwraca -1, łatwo
pominąć w kodzie
# text.index(„java”
```

Częsty błąd — zapomnieć, że `find()` w przypadku braku podciągu zwraca `-1`, nie `False` lub `None`. Sprawdzam jak `if text.find("java"):` nie będzie działać poprawnie, ponieważ `-1` jest prawda w logiczności Sprawdzone! Dokładnie — żeby porównać wprost: `if text.find("java") != -1:`.

input() „naprawdę”

input_ne_chislo.py

```
age = input(„Wiek? ”)
# nawet jeśli wpisane zostanie "28", to łańcuch znaków "28"
# print(age + 1)           # TypeError – zapomniano int()
print(int(age) + 1)         # poprawna
```

Końcowa ściąga do debugowania stringów

Objaw	Przyczyna	Jak to naprawić
Porównanie `==` niespodziewanie False	niewidzialną przestrzeń lub inny przypadek	`repr()`, `.strip()`, `.lower()`
`SyntaxError: unterminated string`	zapomniany znak zamykający cudzysłów	sprawdzić parzystość cudzysłowów
Tekst „zepsuty”	losowe `\n` w ścieżce/tekście	raw-łańcuch znaków `r"..."`
`TypeError: can only concatenate str`	dodawanie łańcuch znaków o liczbie oddzielonej przez +	`str()` lub f-łańcuch znaków
`if text.find(...)` zachowuje się dziwnie	-1 to prawda w czeku boolowskim	porównać `!= -1` wyraźnie
`TypeError` w arytmetyce z input()	input() zawsze zwraca łańcuch znaków	`int()` / `float()`

★★ Zadanie samodzielne
Znajdź błąd
W wierszu `total = „Suma: ”

Praktyka: debugowanie błędów w ciągach znaków
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/08-20/index.html)

Mini-projekt – przeniesienie Turtle tekstu na wyższy poziom!

Połącz input() z sekcji 8.15 z write() z rozdziału 7 — osobiste pozdrowienia umieszczone bezpośrednio na płótnie.

Połącz znaki znaków, dane wejściowe od użytkownika i Turtle z rozdziałów 6-7: poproś o nazwę i wynik Osobiste powitanie bezpośrednio na płótnie.

turtle_tekst_imya.py

```
import turtle

screen = turtle.Screen()
artist = turtle.Turtle()
artist.hideturtle()

name = input(„Jak masz na imię?”)

artist.penup()
artist.goto(0, 0)
artist.write(f"Cześć, {name}!", align="center", font=("Arial",
24, "bold"))

screen.exitonclick()
```

★★ Zadanie samodzielne

Powitanie w zależności od pory dnia

Poproś użytkownika o liczbę (godzinę dnia) w input() + int() i wypisz różne teksty w zależności od wartości — używając tylko tego, co już wiesz o łańcuch znakówx (pełny if będzie w Rozdziale 9, tutaj wystarczy wyświetlić jeden tekst z zastąpioną godziną).

Praktyka: Wprowadź nazwę (logikę, brak okna Turtle)

Ten sam notebook, co w sekcji „Wejście od użytkownika” — obejmuje też ten temat

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/08-07/index.html)

Mini-projekt — powitanie i formatowanie imienia i nazwiska

Składanie łańcuchów f-stringów, metod rejestrów i strip() w dwóch praktycznych mini-projektach.

Mini-projekt – generator pożyczeń

Łączymy porę dnia i imię w jedno uprzejme powitanie — używając f-łańcuchów (rozdział 8.14) i metod rejestru (rozdział 8.9):

generator_privetstvij.py

```
daytime = input(„Czy to rano, popołudnie, wieczór czy noc?”)
name = input(„Jak masz na imię?”)

privetstviya = {
    „dzień dobry”: „Dzień dobry”,
    „dzień”: „Dzień dobry”,
    „wieczór”: „Dobry wieczór”,
    „Noc”: „Dobranoc”,
}
privetstvie = privetstviya.get(daytime.lower().strip(),
„Cześć”)
print(f"{privetstvie}, {name.strip().title()}!")
#Good dzień dobry, Ada!
```

wyprzedzanie faktów

Tutaj już używamy słownik `{...}` to struktura „klucz → wartość” `.get(ключ, значение_по_умолчанию)` : „Znajdź klucz, a jeśli go nie masz, zwróć opcję zapasową.”

Mini-projekt – formater pełnej nazwy

Częstym zadaniem praktycznym: nadać wpisanemu imieniu i nazwisku schludny wygląd Niezależnie od sposobu wpisania przez użytkownika — wielkimi literami, małymi literami, z dodatkowymi spacjami:

formatirovshik_fio.py

```
raw_first = input(„Nazwa: ”)
raw_last = input(„Nazwisko: ”)

first = raw_first.strip().capitalize()
last = raw_last.strip().capitalize()
full_name = f"{first} {last}"
initials = f"{first[0]}. {last[0]}."

print(f"Pełne imię i nazwisko: {full_name}")
print(f"Inicjały: {initials}")
# wejście: "  ada  " / "LOVELACE"
#Full Imię: Ada Lovelace
# Inicjały: A. L.
```

★★ Zadanie samodzielne

Podpis elektroniczny

Rozszerz kreator imienia i nazwiska: dodaj trzeci input() dla zawodu i stwórz e-mail- podpis w rodzaju „Ada Lovelace, programista” jednym f- wierszem.

Praktyka: Generator Powitań i Pełne Imię

interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/08-21/index.html)

Mini-projekt — krzyk do ekranu

Dwa krótkie, ale ilustracyjne mini-projekty dotyczące metod i linii.

Mini-projekt — krzyk do ekranu

Prosty, ale zabawny program: zamienia dowolną frazę w „krzyk” — wielkimi literami i z mnóstwem znaków wykrzyknienia.

krik.py

```
phrase = input(„Co chcesz krzyczeć?”)
krik = phrase.upper() + "!!!"
print(krik)
```

Mini-projekt – odwrócenie nazwy

Użyj kawałka `[::-1]` z sekcji 8.7 na rozszerzenie Wpisane imię od tyłu:

perevorot_imeni.py

```
name = input(„Jak masz na imię?”)
perevernutoe = name[::-1]
print(f"Twoje imię odwrócone: {perevernutoe}")
```

Podstawowa praktyka

Krzyk z ograniczeniem

Zmień krik.py tak, aby było tyle wykrzykników, ile liter w frazie (podpowiedź: mnożenie linii przez liczbę — str * n — powtarza ją n razy, sekcja 8.4).

★★ Zadanie samodzielne

Palindrom?

Dodaj perevorot_imeni.py tak, aby określał, czy wpisane słowo jest palindrom — czy czyta się tak samo w obu kierunkach (porównaj name z odwróconą wersją).

Praktyka: Krzyk i zmiana nazwy

interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/08-09/index.html)

Mini-projekt — weryfikacja hasła i e-mail

Łączenie metod walidacji łańcuch znaków w dwa praktyczne programy walidacji danych wejściowych.

Mini-projekt – łatwa weryfikacja haseł

Zbierzmy kilka zasad (długość, obecność numeru, obecność litery) w jednym sprawdzeniu — Stosując metody weryfikacji z sekcji 8.11:

proverka_parolya.py

```
password = input(„Wymyśl hasło: ”)

dostatochno_dlinnyj = len(password) >= 8
est_cifra = any(ch.isdigit() for ch in password)
est_bukva = any(ch.isalpha() for ch in password)

print(f"Długość od 8 znaków: {dostatochno_dlinnyj}")
print(f"Jest cyfra: {est_cifra}")
print(f"Jest list: {est_bukva}")

nadyozhnyj = dostatochno_dlinnyj and est_cifra and est_bukva
print(f"hasło jest silne: {nadyozhnyj}")
```

wyprzedzanie faktów

`any(...)` z wyrażeniem wewnątrz jest zwartą formą pętli z sekcji 8.13: „Czy istnieje przynajmniej jeden znak, dla którego warunek jest prawdziwy?” `and` / `or` i takie zwarte struktury zostaną przeanalizowane w rozdziałach 9-10.

Mini-projekt – łatwe e-mail sprawdzenia

Prawdziwa email check to trudne zadanie (istnieją specjalne biblioteki do tego zadania), ale Podstawowe „kontrole zdrowego rozsądku” `in` (Rozdział 8.12), `count()` oraz `find()` (Rozdział 8.10):

proverka_email.py

```
email = input(„Twój e-mail: ”).strip()

est_sobachka = "@" in email
odna_sobachka = email.count("@") == 1
est_tochka_posle = "." in email[email.find("@"):]

pohozhe_na_email = est_sobachka and odna_sobachka and
est_tochka_posle
print(f"Wygląda na e-mail: {pohozhe_na_email}")
# wejście: „ada@cartesianschool.org”
# Wygląda na e-mail: True
```

★★ Zadanie samodzielne
Własne zasady
Dodaj jeszcze jedną zasadę do sprawdzania hasła: hasło nie może zawierać spacji (użyj isspace() lub in „”

Praktyka: Hasło i weryfikacja e-mail
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji
Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/08-22/index.html)

Mini-projekt – licznik słów i czyste zdania

Liczyć słowa w tekście i ucz się automatycznie zastępować niechciane słowa.

Mini-projekt – licznik słów

Policz ile słów znajduje się w tekście i ile razy każde z nich się pojawia, używając `split()` (Rozdział 8.10) oraz liczenie cykliczne (Rozdział 8.13):

schetchik_slov.py

```
text = input(„Wprowadź zdanie: ”)
words = text.lower().split()

print(f"Łączna liczba słów: {len(words)}")

schetchik = {}
for word in words:
    schetchik[word] = schetchik.get(word, 0) + 1

for word, count in schetchik.items():
    print(f"{word}: {count}")
# wejście: „kot i pies i kot”
# Łączna liczba słów: 5
# kot: 2
# i: 2
#dog: 1
```

wyprzedzanie faktów
Słownik `schetchik` jest struktura „słowa → ile razy się pojawiło” `.get(word, 0)` zwraca aktualny licznik słowa (lub 0, jeśli słowo pojawiło się po raz pierwszy), a my dodajemy do niego jeden.

Mini-projekt — czyszczenie propozycji (cenzor tekstowy)

Zastąpienie „zakazanych” `replace()` (Rozdział 8.10) i powtarzanie linii (Rozdział 8.4):

cenzor.py

```
text = input(„Wprowadź tekst: ”)
zapreshchennye = [„bad_word”, „tajemnica”]

ochishchennyj = text
for word in zapreshchennye:
    zamena = "*" * len(word)
    ochishchennyj = ochishchennyj.replace(word, zamena)

print(ochishchennyj)
#input: „To tajemnica, nikomu nie mów”
#to ******, nikomu nie mów
```

★★ Zadanie samodzielne
Najczęściej używane słowo
Rozszerz licznik słów: znajdź słowo z maksymalnym count i wyświetl je w osobnej linii „Najczęstsze słowo: ...”. Podpowiedź — przejrzyj słownik schetchik.items() i zapamiętuj aktualnego lidera.

Praktyka: licznik słów i cenzor tekstu
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji
Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/08-23/index.html)

Mini-projekt – kolorowa i dynamiczna matematyka

Złożenie rozdziału: dane wejściowe użytkownika, liczby, łańcuch znaków i Turtle w jednym interaktywnym mini-kalkulatorze.

Ostatni mini-projekt rozdziału łączy niemal wszystko: dane wejściowe od użytkownika, liczby z Rozdziały 4-5, kwestie i Turtle z rozdziałów 6-7 to mały „kalkulator z postacią”

dinamicheskaya_matematika.py

```
import turtle

screen = turtle.Screen()
artist = turtle.Turtle()
artist.hideturtle()

a = float(input(„Pierwszy numer: ”))
b = float(input(„Drugi numer: ”))

artist.penup()
artist.goto(0, 60)
artist.pencolor("purple")
artist.write(f"{a} + {b} = {a + b}", align="center",
font=("Arial", 18, "bold"))

artist.goto(0, 0)
artist.pencolor("blue")
artist.write(f"{a} * {b} = {a * b}", align="center",
font=("Arial", 18, "bold"))

screen.exitonclick()
```

Challenge

Jeszcze dwie operacje

Dodaj dwie kolejne linie na płótno: wynik odejmowania i dzielenia jest inny kolor dla każdej z nich.

Praktyka: Matematyka dynamiczna (logika, Turtle) bez okien

interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/08-10/index.html)

Podsumowanie

Czego nauczyliśmy się w tym rozdziale

- łańcuch znaków są tworzone w cudzysłowie pojedynczym, podwójnym lub potrójnym; potrójne łańcuch znaków przechowują podziały linii, struny raw- (`r"..."`) wyłącz osłony.
- Sekwencje serwisowe takie jak `\n` oraz `\t` wstawiać znaki, które nie mają własnego klucza; `repr()` pokazuje wiersz „jak w kodzie”.
- Znaki w łańcuch znaków są dostępne przez indeks od zera i przez indeks ujemny od końca — pierwszy znak zawsze ma indeks 0.
- Wycinanie `[start:stop:step]` usuwa część linii: `start` w tym `stop` — nie; `[::-1]` odwraca łańcuch.
- łańcuch znaków są niezmienne: metody takie jak `upper()` zwrócić NOWĄ strunę bez dotykania oryginalnej.
- Struny mają dziesiątki gotowych metod, z obudowy (`upper()` , `strip()`) do wyszukiwania i analizowania (`split()` , `replace()` , `find()`) i sprawdzanie składu (`isdigit()` , `isalpha()`).
- `in` sprawdza wystąpienie podciągu znaków; łańcuch znaków ten może być zapętlony `for ch in text` .
- Stringi f-strings to nowoczesny sposób formatowania, obsługujący wyrażenia, precyzję i wyrównanie; `%` oraz `.format()` — starsze, ale nadal spotykane w rzeczywistym kodzie.
- `input()` zawsze zwraca łańcuch znaków — liczby wymagają jawnej konwersji `int()` / `float()` .
- Python działa równie dobrze z cyrylicą, łaciną i emotikonami bez specjalnych ustawień.

ROZDZIAŁ 9 · STR. 434

Wykonaj moje polecenie!

Jak dotąd nasze programy w większości wykonywały polecenia jedno po drugim. Teraz nauczymy ich, jak wybierać, które polecenie wykonać następne. Przeanalizujmy algorytmy i kontrolujmy przepływ, jak warunki tworzą rozdroża i jak Python podejmuje decyzje na True, False if/elif/else — i złóżmy pierwszą prawdziwą grę.

Algorytmy i polecenia

Zanim nauczymy program wyboru między opcjami, zastanówmy się, co oznacza "wykonywanie poleceń" w ogóle — i skąd pochodzi termin "algorytm".

Jak dotąd nasze programy wykonywały te same polecenia w tej samej kolejności na Każdy bieg. Zanim nauczysz programu **wybrać**co robić dalej, Ustalmy, co w ogóle oznacza „wykonywanie poleceń”

Kształt bloku nie jest ozdobą

Każda figura na schemacie blokowym ma ścisłe znaczenie: podpowiada, jaki rodzaj kroku algorytmu jest przed tobą. Będziemy korzystać z tego języka przez całą rozdział:

Początek / Koniec
terminator

Akcja
proces/zespół

Wejście/Wyjście
równoległobiegi

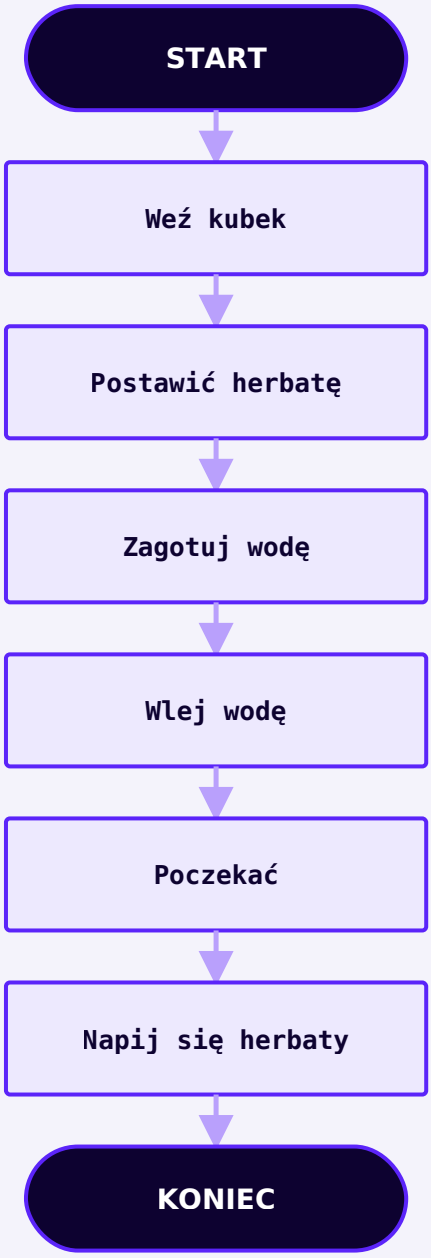
Warunku?
decyzja (diament)

Czym jest algorytm

Algorytm — to zrozumiała sekwencja działań, która prowadzi nas od sytuacji początkowej do pożądanego rezultatu. Korzystamy z algorytmów codziennie, nawet nie nazywając ich tak:

ALGORYTM: przygotować herbatę

1. Weź kubek.
2. Położyć herbatę.
3. Zagotować wodę.
4. Wlej wodę.
5. Czekaj.
6. Napij się herbaty.



Algorytm „parzenia herbaty”

Każdy pojedynczy krok jest **komenda** (instrukcja): Instruuj komputer (lub osoby) do wykonania konkretnej czynności. Cały uporządkowany zestaw kroków to algorytm.

Dowodzenie i ekspresja to różne rzeczy

Używaliśmy już poleceń od samego pierwszego programu w rozdziale 1 — tylko nie nazywaliśmy ich w ten sposób:

znakomye_komandy.py

```
print(„Cześć”)

name = input(„Jak masz na imię?”)

score = 10
```

- `print(...)` jest polecenie „pokaż coś na ekranie”
- `input(...)` jest polecenie „Request information from the user”
- `score = 10` jest poleceniem „Przypisz nazwę z wartością”

2 + 2 to wyrażenie, a nie polecenie

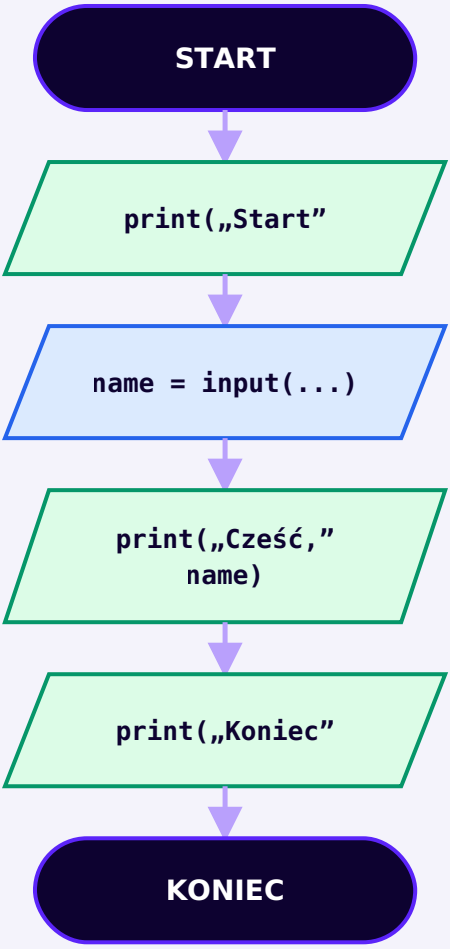
`2 + 2` samo w sobie nic nie „robi” **wyraz twarzy**: jest obliczany i daje wartość (`4`). Ale `print(2 + 2)` — już polecenie: pobiera wartość wyrażenia i coś z nią robi (wyświetla na ekranie). Formalne rozróżnienie nie musi być teraz zapamiętane dosłownie — ale nie należy mylić „obliczyć wartość” z „wykonać działanie”.

Program to ciąg poleceń

Rozważmy prosty program i prześledźmy, jak Python wykonuje go komenda po komendzie. Zwróć uwagę: `input()` jest losowany jako ENTER, oraz `print()` jako KONKLUZJĘ, z różnymi kształtami, nie tymi samymi Prostokąty:

posledovatelnaya_programma.py

```
print(„Start”)

name = input(„Jak masz na imię?”)

print(„Cześć,”, name)

print(„Koniec”)
```



Ten sam kod co schemat przepływu – wejście i wyjście rysowane inaczej niż te same prostokąty

Zazwyczaj Python ściśle wykonuje polecenia **od góry do dołu**, jeden po drugim. To jest jest nazywana **kolejno** (lub liniowe) wykonanie: co następne Krok następuje zaraz po poprzednim, bez pomijania i bez wyboru.

Praktyka: Algorytmy, Polecenia i Wykonanie Sekwencyjne
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/09-07/index.html)

Trzy struktury algorytmu i rozgałęzienie

Każdy algorytm składa się z trzech idei: sekwencji, rozgałęzienia i powtórzenia. Ten rozdział dotyczy drugiego z nich.

Trzy struktury, z których budowany jest każdy algorytm

Prawie każdy algorytm — od przepisu na herbatę po ogromny program — składa się tylko z trzech podstawowych pomysłów:

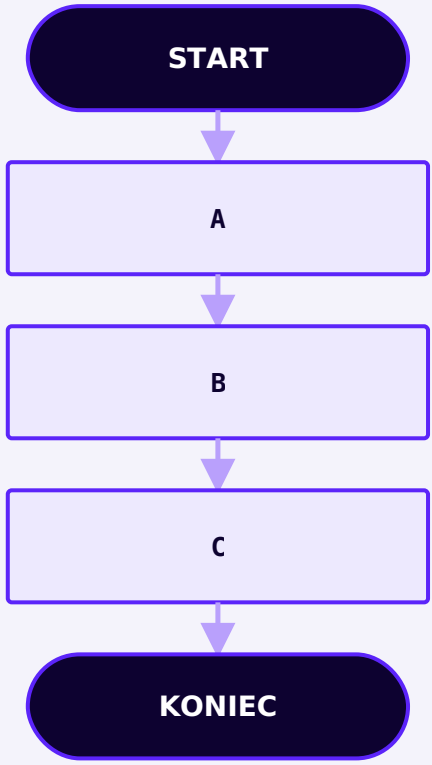
Jaka struktura jest potrzebna?

Komendy idą jedna po drugiej, bez wyboru?	→ **Sekwencja — już znana**

Trzeba wybrać JEDNĄ ścieżkę spośród kilku, w zależności od warunku?	→ **Rozgałęzienie — ten rozdział**
Czy trzeba powtarzać tę akcję kilka razy?	→ **Powtórzenie — rozdział 10**

Sekwencja

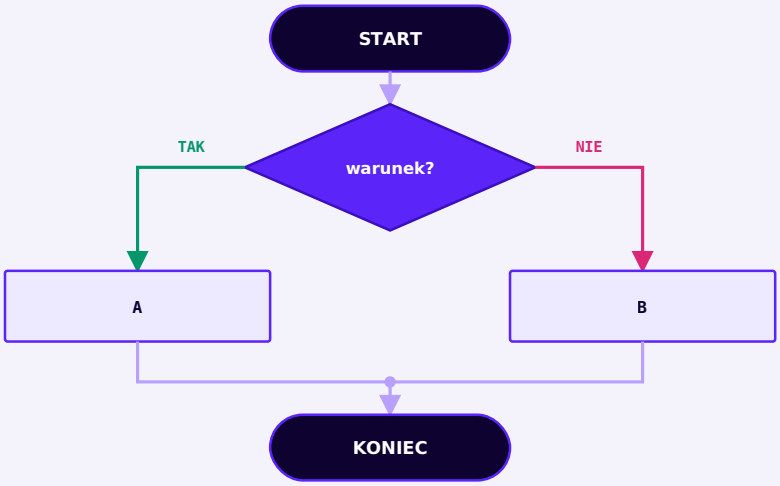
Polecenia są wykonywane pojedynczo, bez wyboru i powrotu.



Sekwencja — każdy krok następuje bezpośrednio po poprzednim

Rozgałęzienie

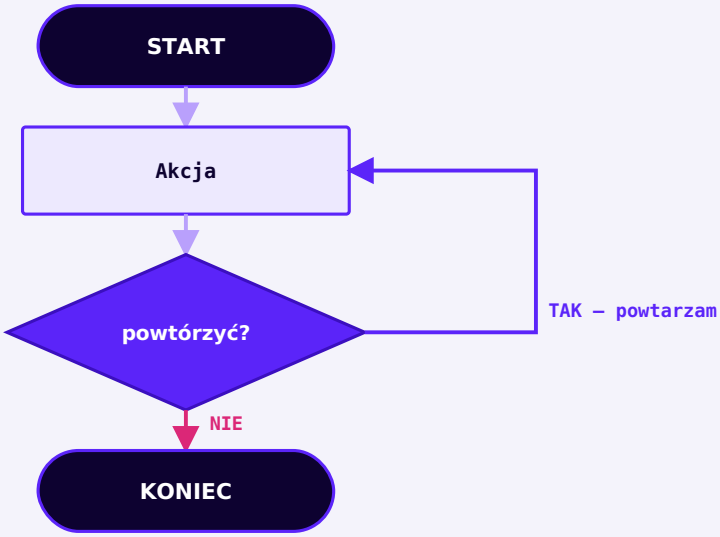
Pytanie logiczne wybiera jedną z dwóch ścieżek; po wykonaniu rozgałęzień ścieżki się zbiegają.



Rozgałęzienie – wybiera się dokładnie jedną ścieżkę, obie opcje się zbiegają

Powtarzalność

Po akcji pytanie logiczne decyduje, czy powtórzyć akcję, czy zakończyć cykl.



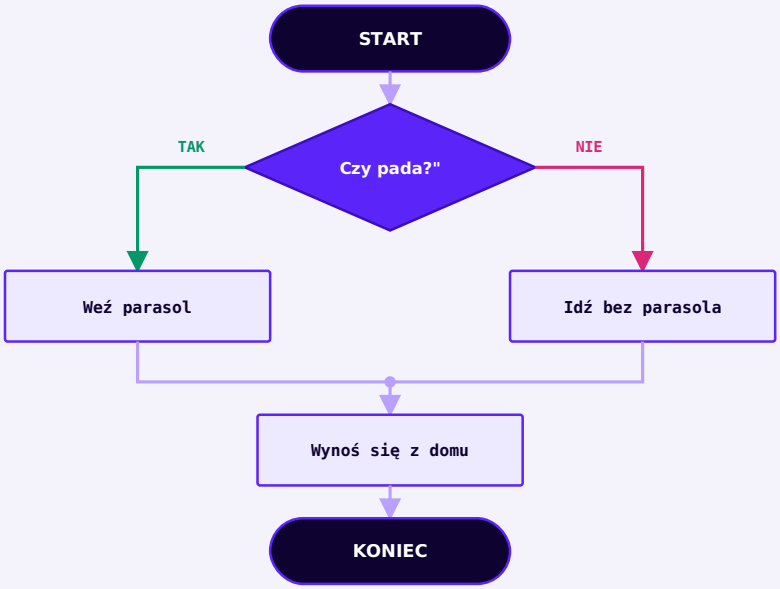
Powtarzanie — wstępny schemat, składnię pętli poznamy w rozdziale 10

Gdy brakuje jednej sekwencji

Prosty algorytm liniowy — „wyjdź z domu, idź na spacer”

Czym jest rozgałęzienie

Rozgałęzienie — to miejsce w algorytmie, gdzie dalsze działania zależą od odpowiedzi na pytanie. Każda rozgałęzienie zaczyna się od **warunki** — pytanie, na które można odpowiedzieć „tak” lub „nie”. Program nie wybiera gałęzi losowo: oblicza warunek, a wynik determinuje ścieżkę. Niezależnie od tego, którą gałąź wybierze program, dalsze wykonywanie zazwyczaj kontynuuje się z tego samego następnego kroku:



Rozgałęzienie i kontynuacja: obie gałęzie zbiegają się i algorytm przechodzi dalej

To już nie jest czysto sekwencyjny algorytm — **gałęzie**i obie gałęzie ponownie zbiegają się w kroku „Opuść dom”

Od tak/nie do True/False

Ludzkie „tak” i „nie” w Python zamieniają się w dwie szczególne wartości:

Ludzie	Python
Tak	`True`
nie	`False`

Dlatego następna część dotyczy `True` oraz `False` nie jest już tematem abstrakcyjnym „znikąd”

Praktyka: Struktury trzech algorytmów i rozgałęzienia
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/09-08/index.html)

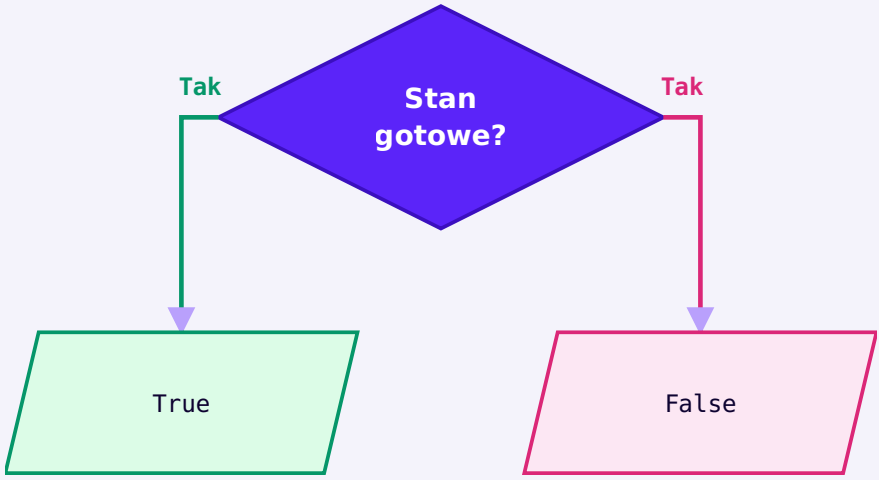
Prawda czy fałsz

Zanim program będzie mógł podejmować decyzje, musi mieć sposób na przechowywanie wyniku samej decyzji — logicznego typu bool.

Dowiedzieliśmy się, że program odpowiada na pytania tak lub nie z wartościami `True` oraz `False`. Czas się z tym zapoznać Rodzaj danych, które je przechowują.

bool jest typem Boole'a o dwóch wartościach

Typ `bool` ma dokładnie dwa możliwe znaczenia:



Pytanie logiczne zawsze daje jeden z dwóch wyników: True lub False

Proszę zauważyć: `True` oraz `False` są pisane wielkimi literami — to Python słowa kluczowe, a nie zwykły tekst.

bool.py

```
is_sunny = True
is_raining = False
print(is_sunny, type(is_sunny))
# True <class 'bool'>
```

bool nie jest łańcuch znaków!

Łatwo pomylić `True` (wartość typu bool) ze stringiem `"True"` — ale to zupełnie różne rzeczy:

bool_ne_stroka.py

```
print(type(True))     # <class 'bool'>
print(type("True"))   # <class 'str'>
print(True == "True")  # False – różne typy, różne wartości
```

„True”
String `"True"` nie ma szczególnego sensu dla Python — jest to zwykły, niepusty tekst, który, jak zobaczymy później, zachowuje się jak `True`, ale to są dwa różne zjawiska, które łatwo pomylić.

Jaki jest najczęstszy sposób na uzyskanie bool

Zapisz ręcznie `True` / `False` są rzadkie — zwykle uzyskuje się je przez porównanie (następna sekcja):

sravnenie_bool.py

```
print(5 > 3)     # True
print(5 == 3)    # False
```

Praktyka: typ bool i True/False
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

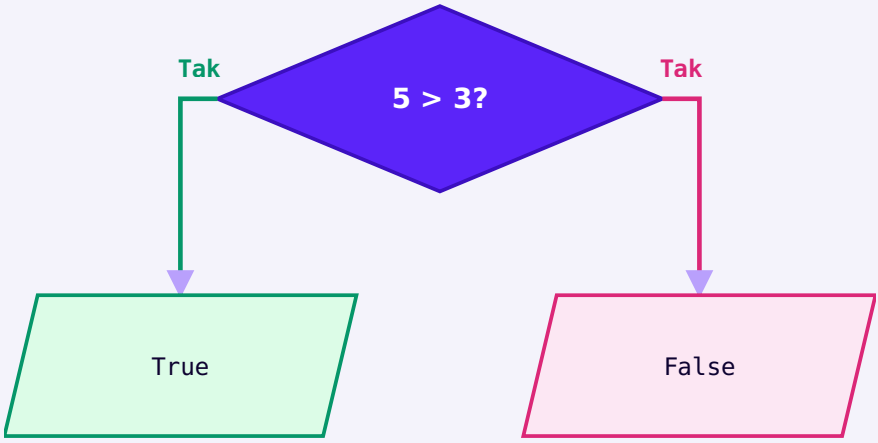
Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/09-01/index.html)

Porównujemy i podejmujemy decyzję

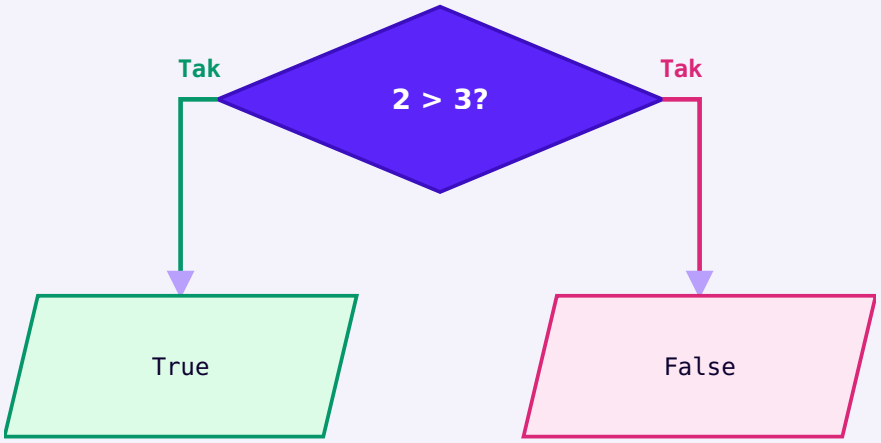
Sześć operatorów, które zamieniają dwie wartości w jeden bool — oraz linię liczbową do zobaczenia ich granic.

Porównanie to krok algorytmu, który generuje bool

Weź dwie wartości, zastosuj operator porównawczy i otrzymasz dokładnie jeden z nich: `True` lub `False`. Oba scenariusze są następujące: **równie udane** efekt: porównanie nie "upada" ani "nie działa", Gdy odpowiedź brzmi False, to po prostu szczerze mówiąc "nie".



5 > 3 – porównanie było SKUTECZNE i dało True



2 > 3 – porównanie też było skuteczne, tylko dało False

sravnenie.py

```
print(5 > 3)     # True
print(5 == 3)    # False
```

False jest też poprawną odpowiedzią

`2 > 3` odnosi sukces równie dobrze jak `5 > 3` po prostu ich wyniki są różne. Nie myl "warunek jest fałszywy" z "program nie zadziałał": to dwa zupełnie różne zjawiska i do tego rozróżnienia wrócimy w sekcji o `if`.

Wszystkie sześć operatorów porównawczych

Pytanie	Matematyka	Python	Przykład	Rezultat
to ma znaczenie?	=	`==`	`5 == 5`	True
nie równe?	≠	`!=`	`5 != 3`	True

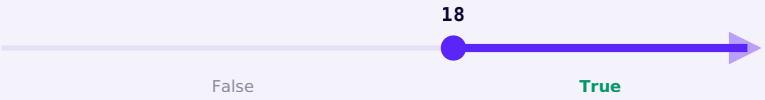
Pytanie	Matematyka	Python	Przykład	Rezultat
więcej?	>	`>`	`5 > 3`	True
mniej?	<	`<`	`5 < 3`	False
większe lub równe?	≥	`>=`	`5 >= 5`	True
mniej lub równe?	≤	`<=`	`5 <= 3`	False

operatory_sravneniya.py

```
print(5 == 5)   # równa się
print(5 != 3)   # nie jest równe
print(5 > 3)    # więcej
print(5 < 3)    # mniej
print(5 >= 5)   # większe lub równe
print(5 <= 3)   # mniejsze lub równe
```

Oś Liczb – Zobacz porównanie

`age >= 18` oznacza „age większe niż LUB równe 18”

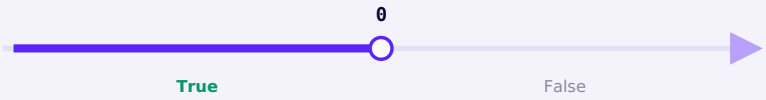


age >= 18 — wypełniony punkt: 18 wliczone

Na lewo od 18 — obszar, gdzie warunek jest fałszywy; sam punkt 18 i wszystko na prawo to obszar, gdzie warunek jest prawdziwy. Punkt na 18 **zamalowany**: oznacza to, że 18 wpisuje się w wartość prawda Zakres (`>=` zawiera ramkę).

Otwarta i zamknięta granica

Porównaj z `temperature < 0`:



temperature < 0 - Pusty okres: 0 NIE włączony

Oto punkt na 0 **pusty** (Nie pomalowane) - Ramka NIE włączona: Wartość `0` nie spełnia warunku `< 0`. Zgódźmy się w tym przez cały rozdział:

- **zacieniony punkt** - Granica włączona (`>=` , `<=`)
- **pusty punkt** jest granicą wykluczoną (`>` , `<`)

Praktyka: Sześć operatorów porównawczych i granic
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

`Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/09-02/index.html)`

Jakie są różnice = oraz ==i jak porównywać struny

Jednym z najczęstszych błędów początkujących jest to, jak linie z rozdziału 8 są zaangażowane w warunki.

= — to nie to samo, co ==

Ta część zasługuje na szczególną uwagę – pomyłka między pojedynczym a podwójnym „równi”

Dwóch różnych operatorów: = oraz ==

= (jedna cyfra) - PRZYDZIAŁ

```
name = "Anna"

# sens: powiązać nazwę name
# z wartością „Anna”
# Nic się nie równa!
```

== (dwie cyfry) - PORÓWNANIE

```
name == "Anna"

#meaning: Czy wartości są
równe?
# wynik to True lub False,
# zmienna sam się nie zmienia
```

Co używać dzisiaj: Pomylić `=` oraz `==` jest jednym z najczęstszych błędów popełnianych przez początkujących w każdym języku programowania. Jeśli `if` przypadkowo napisać singla `=`, Python od razu zgłosi błąd składni — przypisania nie można używać jako warunku, więc ten konkretny błąd, na szczęście, nie przechodzi niezauważony.

!= — „nie równy”

`!=` — operator odwrotny `==`:

ne_ravno.py

```
password = ""
print(password != "")   #False – hasło jest po prostu puste

password = "secret123"
print(password != "")   #True – Hasło NIE jest puste
```

Dlaczego != a nie ≠

W matematyce „nie równa się” zapisuje się jednym znakiem — ≠. Na klawiaturze nie ma takiej ikony, a Python (jak C, Java JavaScript prawie wszystkie języki programowania) nie rozumie jej jako kodu — musisz zapisać dokładnie dwa zwykłe symbole w rzędzie: wykrzyknik i znak równości, `!=`. Są to dwa różne, ale równoważne określenia tej samej idei: ≠ jest zapisem dla osoby na papierze, `!=` jest wpisem dotyczącym Python. If w kodzie na tej stronie `!=` wygląda jak pojedyncza przekreślona ikona — to po prostu cecha czcionki edytora (ligatury), która jest piękna *ekspozycje* dwóch postaci obok siebie; nadal musisz pisać `!` oraz `=` osobno — nie musisz szukać osobnej „ikony ≠”

Porównanie stringów

Widzieliśmy w rozdziale 8: struny są porównywane przez postacie, a przypadek jest ważny:

sravnenie_strok.py

```
print("Python" == "python")   #False – Sprawa jest ważna
print("cat" < "dog")          # True
```

"cat" < "dog" nie jest do końca "alfabetycznie"

Python porównuje linie przez **pozycje kodów znaków**, a nie według „ludzkiego” alfabetycznego porządku słownika. Dla zwykłych liter łacińskich w jednej wielkości rezultat zwykle pokrywa się z intuicją, ale przy mieszaniu wielkości liter lub różnych alfabetów nie zawsze jest to „porządek alfabetyczny”

TROCHĘ GŁĘBIEJ JEST KOD SYMBOLU

Każdy znak odpowiada liczbie — swojemu kodowi. Funkcja `ord()` to pokazuje: `ord("A")` → 65, `ord("a")` → 97. To są liczby Python faktycznie porównuje. W tym rozdziale nie musisz znać dokładnych liczb — ważne jest tylko, by zrozumieć, że porównanie łańcuch znaków działa przez nie, a nie przez słownik.

Normalizacja przed porównaniem

Użytkownik może wpisywać "Tak", "TAK", "Tak" – z różnymi skrzynkami i spacjami. Metody wersy z rozdziału 8 rozwiązują ten problem jednym zdaniem:

normalizaciya.py

```
answer = input(„Kontynuować?”).strip().lower()

if answer == „tak”:
    print(„Kontynuujemy”)
```

`.strip()` usuwa przypadkowe szczeliny na krawędziach, `.lower()` małymi literami — nie ma znaczenia jak To użytkownik wpisał odpowiedź.

Praktyka: == vs. =, != i porównanie łańcuch znaków
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/09-09/index.html)

Łańcuchy porównawcze

Jak sprawdzić, czy wartość mieści się w określonym zakresie — w jednej eleganckiej linii zamiast dwóch oddzielnych porównań.

Zakres wartości – łańcuch porównań

Częste zadanie: sprawdzić, czy wartość mieści się POMIĘDZY dwoma granicami. W Python można to zapisać za pomocą jednej eleganckiej sekwencji:

cepochka.py

```
age = 30
print(18 <= age <= 65)   # True
```

18 <= age <= 65 to zakres objęty

To dokładnie to samo co następujące:

cepochka_razvernuto.py

```
age = 30
print(age >= 18 and age <= 65)   # ten sam wynik
```

ale łańcuch `18 <= age <= 65` czyta się bliżej tego, jak Formułujemy to w ludzkim języku – „age między 18 a 65 latami”

Jeszcze przykłady

primery_cepochek.py

```
print(0 <= score <= 100)      # score od 0 do 100 włącznie
print(-10 < temperature < 30)
# temperatura ściśle między -10 a 30
print(1 <= month <= 12)       #month jest ważną liczbą miesiąca
```

-10 < temperature < 30

Uwaga: tutaj są oba ograniczenia **otwarte** (surowa `<`) — wartości -10 i 30 same NIE wchodzą w zakres, co widać po pustych punktach na prostej.

Podstawowa praktyka

Dozwolony miesiąc

Podano wartości month = 0, month = 1, month = 12, month = 13. Dla każdego przewidź wynik 1 <= month <= 12, a następnie sprawdź w kodzie.

Praktyka: Łańcuchy porównawcze

interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/09-10/index.html)

Truthiness i None

Stan nie zawsze wymaga wyraźnego porównania — Python potrafi zapytać o każde znaczenie, czy jest prawdziwe, czy fałszywe.

Stan nie zawsze wymaga wyraźnego porównania

już widzieliśmy w rozdziale 8: Pusty łańcuch znaków zachowuje się jak `False`, a niepusty — jak `True`. To działa nie tylko dla łańcuch znaków — Python potrafi zapytać o DOWOLNĄ wartość: „Czy zachowujesz się jak prawda, czy jak fałsz?” To nazywa się **truthiness** („prawda”

Falsy są wartości, które zachowują się jak False

Znaczenie	bool(...)	Dlaczego
`False`	False	już fałsz
`None`	False	„to nie ma znaczenia”
`0`, `0.0`	False	zero jest uznawane za fałsz
`""`	False	pusty łańcuch znaków
`[]`, `{}`	False	pustych kolekcjach szczegółowo przyjrzy-my się później

Truthy to prawie wszystko inne

Znaczenie	bool(...)	Dlaczego
`True`	True	już prawda
`-10`, `1`, `42`	True	dowolną niezerową liczbą
`"нет"`, `"0"`, `"Fal-se"`	True	każdy niepusty łańcuch znaków — nawet taki!

truthiness.py

```
print(bool(0))        # False
print(bool(1))        # True
print(bool(-10))      #True – ujemne też nie jest zerem!
print(bool(""))       # False
print(bool("False"))  #True jest niepustym łańcuch znaków
łańcuch znaków, a nie wartością False
print(bool("0"))      # True również nie jest pustym łańcuch
znaków
```

Tekst „False”
`bool("False")` równa się `True` bo łańcuch znaków `"False"` składa się z pięciu znaków — nie jest pusty, co oznacza, że truthy. To samo z `"0"` : To jest tekst z pojedynczym znakiem, a nie z cyfrą zero.

None – „Tu nie ma sensu”

`None` jest szczególnym znaczeniem oznaczającym coś w stylu „nie ma wartości”

none.py

```
value = None

print(value is None)       # True
print(bool(value))          #False – None jest też falsy
print(value == 0)           #False – None to nie jest 0
print(value == "")          #False – None to nie jest pusta
łańcuch znaków
```

Zwróć uwagę na operator `is` zamiast `==` — do różnicy między nimi wrócimy w sekcji 9.17.

Praktyka: truthiness i None
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

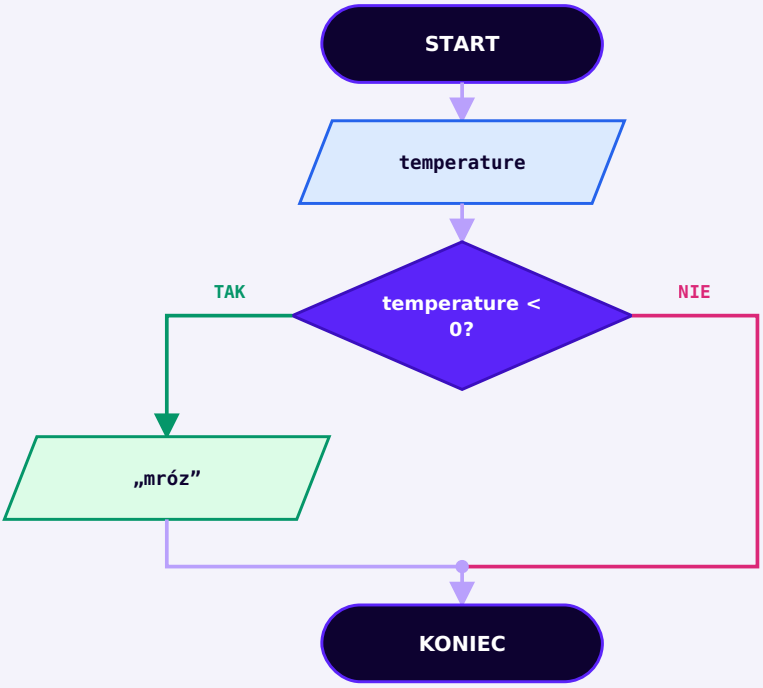
Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/09-11/index.html)

Jeśli tak się stanie, wykonuj polecenie!

Pierwsze prawdziwe stwierdzenie warunkowe: kod, który wykonuje się tylko w określonych okolicznościach, najpierw jako schemat, potem na Python.

najpierw schemat, kod

Weźmy problem: "jeśli temperatura spadnie poniżej zera, pokaż "szron". Przed pisaniem Python, rozwiążmy to jako schemat:



Najpierw rozwiązujemy problem jako schemat, a następnie przenosimy go do Python

Gdzie tu jest pytanie? Która gałąź zostanie wykonana? Co się stanie, jeśli warunek jest fałszywy? Tylko Gdy odpowiedzi są jasne, zmieniamy schemat na Python.

Pierwsza if

if_prostoj.py

```
temperature = -5

if temperature < 0:
    print(„Frost”)
```

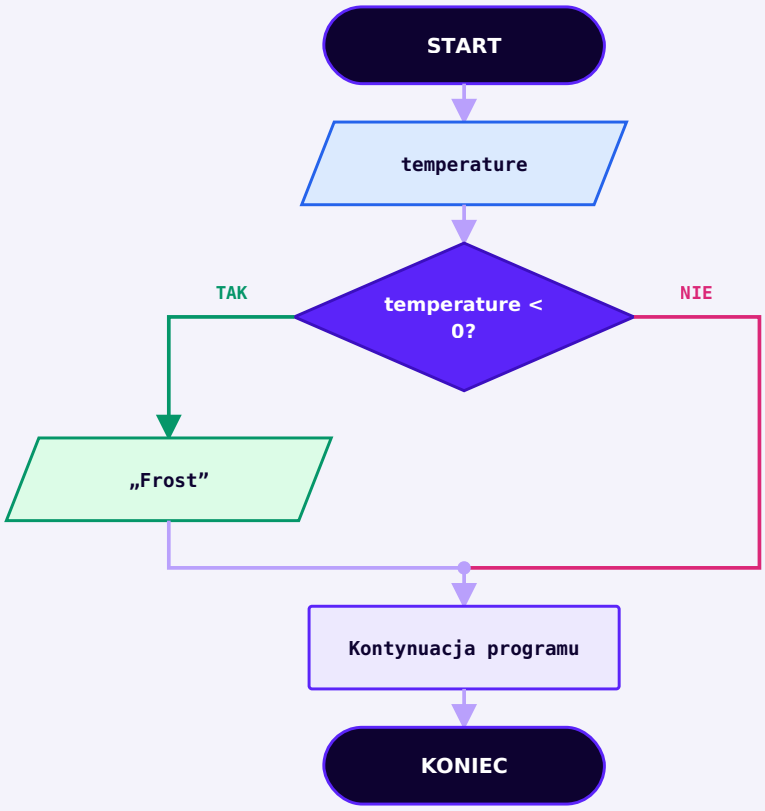
Wyjaśnijmy to słowami:

- `if` — „jeśli”
- `temperature < 0` — warunek (pytanie)
- `:` — tutaj zaczyna się blok poleceń
- łańcuch znaków z wcięciem — polecenie, należące do tego bloku

Pełny przykład z obiema gałęziami

if.py

```
age = 20

if age >= 18:
    print(„Dostęp dozwolony.”)
```



Pierwszym if jest False blok jest po prostu pomijany, wykonanie idzie dalej

Jeśli warunek jest fałszywy, `if` łatwe **pomija się** — program nie "zawiesza się", nie wyświetla błędu, lecz kontynuuje, jakby nic się nie stało. Fałsz Warunek ten nie oznacza "program się zatrzymał": kontynuuje się od następnego kroku.

Wcięcie — to część programu

W Python wgłębienie to nie tylko projekt piękna. To wgłębienie determinuje Python Które wiersze należą do bloku `if`, a które już nie:

blok_komand.py

```
if temperature < 0:
    print(„Na zewnątrz jest lodowato.”)
    print(„Załóż ciepłą kurtkę.”)
    print(„Nie zapomnij rękawiczek.”)

print(„Program się skończył”)
```

Pierwsze trzy komendy mają takie samo wcięcie – wszystkie należą do bloku `if`. Ostatni łańcuch znaków nie ma wcięcia — zostanie spełniony **zawsze**, niezależnie od stanu.

> **Zalecane: 4 przestrzenie**
> Standardowy odstęp w Python — 4 spacje na jeden poziom zagnieżdżenia. Nie mieszaj spacji i tabulacji w tym samym pliku — Python w niektórych przypadkach uzna to za błąd.

IndentationError

Jeśli zapomnisz o wcięciu, Python nie zrozumie, co odnosi się do bloku:

indentationerror.py

```
if age >= 18:
print("OK")
# IndentationError: expected an indented block after 'if'
statement
```

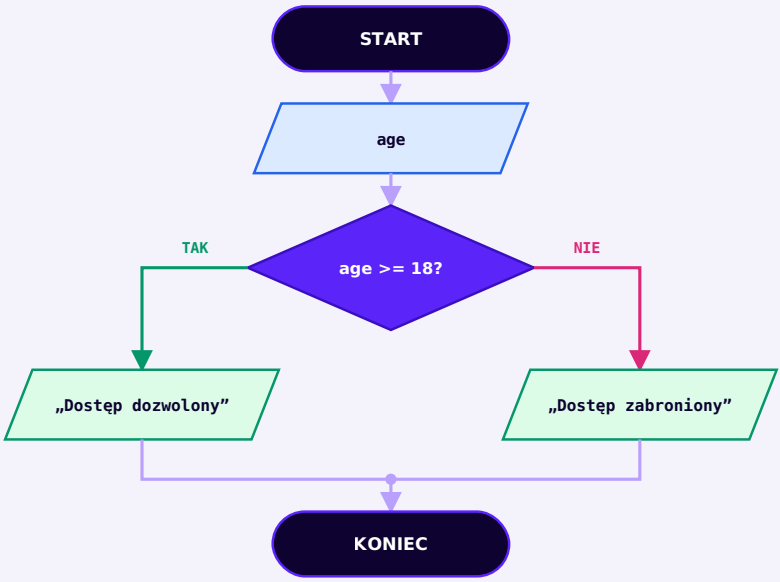
Poprawka: dodaj wcięcie przed `print`.

Inaczej?" — if/else

`else` określa blok kodu, który zostanie wykonany, jeśli okazało się to nieprawdą:

if_else.py

```
age = 15

if age >= 18:
    print(„Dostęp dozwolony.”)
else:
    print(„Dostęp odrzucony — masz jeszcze mniej niż 18 lat.”)
```



if/else: dwóch ścieżek, jedna z nich zostanie spełniona i obie zbiegną się jeszcze bardziej

Gałęzie ponownie się łączą

Która gałąź zostanie wykonana, `if` lub `else` — wtedy program zwykle trwa dalej następny krok. Na powyższym diagramie widać to po zbiegu strzałek do wspólnego punktu przed `КОНЕЦ`.

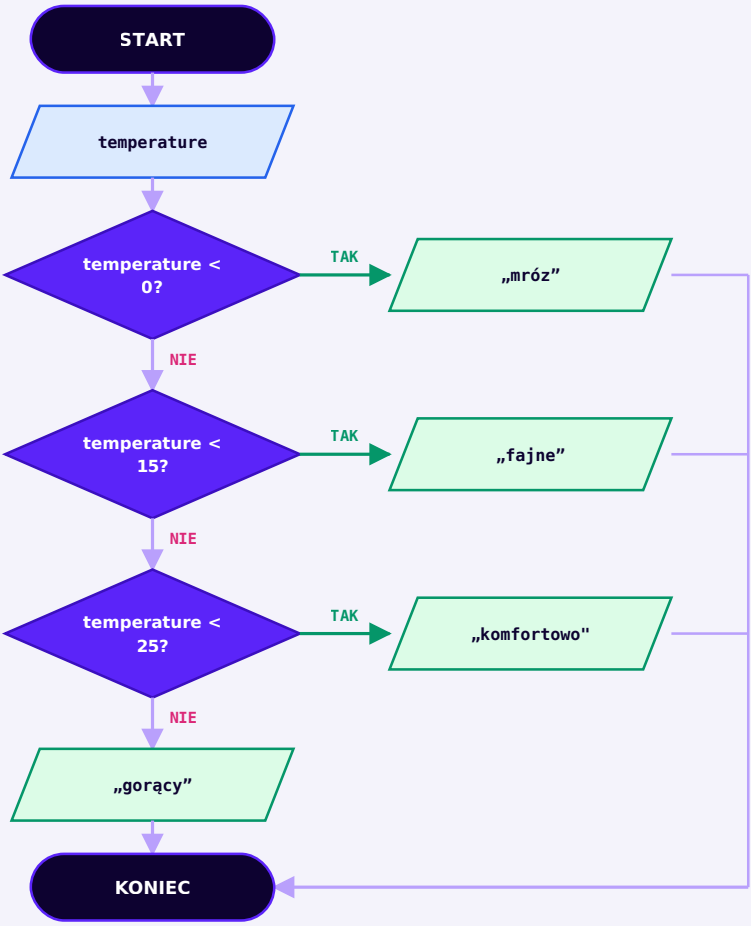
Praktyka: pierwszy if, wcięcie i if/else
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji
Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/09-03/index.html)

elif — kilka opcji

Gdy jest więcej niż dwa wyniki, elif sprawdza warunki kolejno, a pierwszy prawdziwy wygrywa.

Kiedy jest więcej niż dwa wyniki

Temperatura: poniżej zera – "mróz", od 0 do 14 – "chłodno", od 15 do 24 – "komfortowo", w przeciwnym razie "gorąco". Są tu cztery wyniki, nie dwa `if/else` już brakuje.



if/elif/else jako drabinka warunków – pierwszy prawdziwy warunek wygrywa, reszta jest pomijana

elif_cepochka.py

```
temperature = 10

if temperature < 0:
    print(„mróz”)
elif temperature < 15:
    print(„fajne”)
elif temperature < 25:
    print(„komfortowo")
else:
    print(„gorący”)
```

`elif` — skrót od *else if*, „w przeciwnym razie jeśli” `if` oraz `else`.

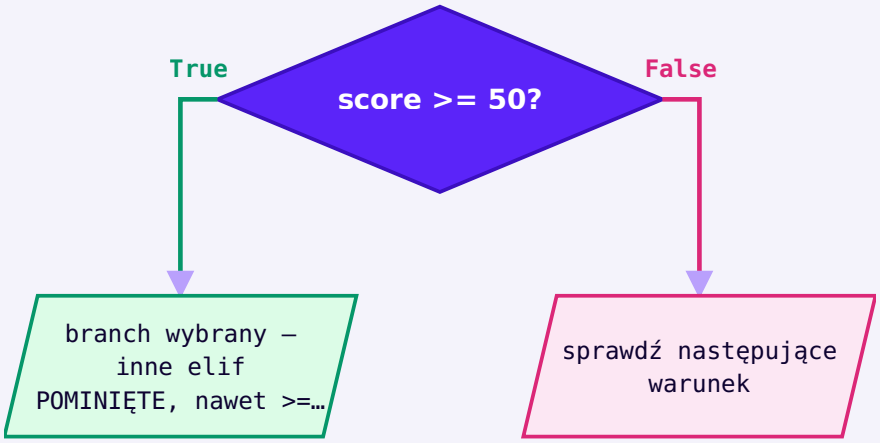
Pierwszy prawdziwy warunek wygrywa

Python sprawdza warunki **w porządku, od góry do dołu**, i zatrzymuje się na pierwszym prawdziwym — nawet jeśli kolejne warunki również by pasowały. Reszta łańcucha jest po prostu pomijana.

Kolejność warunków ma znaczenie

nepravilnyj_poryadok.py

```
score = 95

if score >= 50:
    bukva = "D"
elif score >= 90:
    bukva = "A"
# nigdy nie zostanie spełnione przez score >= 50!
```



score = 95: Jeśli >= 50 pojawi się pierwsze, program nigdy nie osiągnie >= 90

score = 95 trafi do "D" zamiast do "A"
razy `score >= 50` jest pierwszym i `95 >= 50` prawda, Python wybiera tę konkretną gałąź i nawet nie sprawdza `score >= 90`.

Poprawka polega na uporządkowaniu warunków od bardziej rygorystycznych do mniej rygorystycznych:

pravilnyj_poryadok.py

```
score = 95

if score >= 90:
    bukva = "A"
elif score >= 75:
    bukva = "B"
elif score >= 50:
    bukva = "C"
else:
    bukva = "D"
```

★★ Zadanie samodzielne

Trzy poziomy rabatu

Zbuduj drabinę elif-: kwota zakupu >= 5000 - 15% zniżka, >= 2000 - 10% zniżka, >= 500 - 5% zniżka, w przeciwnym razie brak rabatu. Sprawdź kwotę 3000 - która zniżka wygra?

Praktyka: Kolejność drabiny elif- i stan

interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/09-12/index.html)

Kilka if ≠ if/elif/else

Bardzo częsty problem wśród początkujących: kilka oddzielnych if może zadziałać wszystkie naraz, a w łańcuchu elif — tylko jeden.

Blok poleceń i niewidoczny wskaźnik

Gałęzia warunków może zawierać wiele poleceń, jak już widzieliśmy. Przydatne Wyobraź sobie, że Python trzyma niewidzialny „wskaźnik” `if` wybiera, na którą gałąź „wejść”.

To uproszczony model mentalny

Wewnątrz komputera istnieje podobna idea (nazywa się ją „przepływem sterowania” — control flow), ale do naszych celów wystarczy ten prosty obraz: polecenia idą po kolei, a if/elif/else tymczasowo kieruje przepływ do jednej z gałęzi.

Przepływ sterowania

Przepływ sterowania — to kolejność, w jakiej faktycznie wykonywane są polecenia programu. Do tej rozdziału była to prawie zawsze prosta linia od góry do dołu. Teraz może **się rozwijać**. W rozdziale 10 nauczy się też, jak cofnąć się — Powtarzaj się.

Wiele if NIE jest tym samym co if/elif/else

To jedno z najczęstszych nieporozumień wśród początkujących — przyjrzyjmy się konkretnemu przykładowi:

Niezależne if kontra łańcuch if/elif

Dwa NIEZALEŻNE if

```
temperature = 30

if temperature > 20:
    print(„ciepło”)

if temperature > 25:
    print(„gorący”)

# OBA warunki są prawdziwe —
# OBIE linie się wydrukują!
```

Łańcuch if / elif

```
temperature = 30

if temperature > 25:
    print(„gorący”)
elif temperature > 20:
    print(„ciepło”)

# tylko JEDNA gałąź
# w tym łańcuchu wykonam
```

Co używać dzisiaj: Każdy samodzielny `if` jest sprawdzana niezależnie od pozostałych — Python może wykonać kilka takich bloków z rzędu. I w łańcuchu `if/elif/else` wykonuje się DOKŁADNIE JEDNA gałąź — pierwsza prawdziwa. To bardzo częste zamieszanie u początkujących: jeśli potrzebny jest „wybór jednej opcji spośród kilku” — potrzebny jest dokładnie łańcuch elif, a nie kilka osobnych if.

★★ Zadanie samodzielne

Znajdź różnicę

Weź score = 85. Zbuduj wariant z trzema niezależnymi if (score > 50, score > 70, score > 90) oraz wariant z łańcuchem elif-, w tych samych warunkach. Porównaj, ile linii wydrukuje każdy wariant.

Praktyka: Niezależny if kontra elif

interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/09-13/index.html)

and — wszystkie warunki naraz

Czasem decyzja zależy nie od jednego, lecz od wszystkich wymienionych warunków jednocześnie.

Wszystkie warunki muszą być True

Prawdziwa sytuacja: wpuszczasz się na korytarz tylko wtedy, gdy masz 18 lat I masz bilet. Oba warunki są obowiązkowe:

Wejście do sali jest dozwolone tylko wtedy, gdy spełniają się OBA warunki

age >= 18? → **potrzeba TAK**

has_ticket? → **też potrzebuje TAK**

and_primer.py

```
age = 20
has_ticket = True

if age >= 18 and has_ticket:
    print(„Proszę wejść do sali.”)
```

Stół Prawdy and

A	B	A and B
False	False	False
False	True	False
True	False	False
True	True	True

Rezultat `and` jest prawdziwe tylko wtedy, gdy są prawdziwe **obu** operand, we wszystkich pozostałych trzech przypadkach, wynik jest fałszywy.

Przetłumacz na ludzki język

Przydatna umiejętność — czytać warunek na głos jak zdanie:

chitaem_vsluh.py

```
age >= 18 and country == "PL"
# „age nie mniejsze niż 18 I country równe PL”
```

Podstawowa praktyka

Własny warunek z and

Zapisz warunek: can_drive – możesz prowadzić samochód, jeśli age >= 18 I has_license jest równe True. Sprawdź age=20, has_license=False.

Praktyka: and i Tabela Prawdy

interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/09-14/index.html)

or — przynajmniej jeden

Czasami wystarczy, by przynajmniej jeden z wymienionych schorzeń został wywołany.

Jeden True

Zniżka jest przeznaczona dla studentów LUB emerytów – wystarczy, by kwalifikować się do przynajmniej jednej Kategoria:

Przysługuje rabat, jeśli prawdziwe jest PRZYNAJMNIEJ JEDNO warunek

student? → **TAK wystarczy**

senior? → **lub TAK tutaj**

```
or_primer.py
```

```
is_student = False
is_senior = True

if is_student or is_senior:
    print(„Zastosowano zniżkę.”)
```

Tabela prawdziwości or

A	B	A or B
False	False	False
False	True	True
True	False	True
True	True	True

Rezultat `or` jest fałszywe tylko wtedy, gdy są fałszywe **obu** operand — we wszystkich pozostałych przypadkach jest prawdziwy.

Podstawowa praktyka

Weekend lub święto

Zapisz warunek: can_rest – nie możesz pracować, jeśli is_weekend jest True LUB is_holiday równa się True. Sprawdź is_weekend=False, is_holiday=True.

Praktyka: or i tabela prawdy

interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/09-15/index.html)

not — odwróć warunek

Najprostszy operator logiczny: zamienia True w False i odwrotnie.

Odwrócenie wartości boole'a

`not` jest najprostszym operatorem logicznym: jest po prostu Flipy `True` w `False` i odwrotnie.

A	not A
True	False
False	True

not_primer.py

```
is_raining = False

if not is_raining:
    print(„Możesz chodzić”)
```

Przeczytaj na głos: „jeśli NIE pada” `not is_raining` naprawdę dokładnie kiedy `is_raining` nieprawda.

not_s_sravneniem.py

```
age = 15
print(not (age >= 18))   #True – ponieważ age >= 18 samo w
sobie False
```

Podstawowa praktyka

Podwójna not

Przewidź wynik not not True — a następnie sprawdź w kodzie. Wyjaśnij własnymi słowami, dlaczego otrzymano właśnie taką odpowiedź.

Praktyka: not i odwrócenie warunków

interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/09-16/index.html)

Więcej niż jeden warunek! :O

and, or i not razem stanowią priorytet operatorów oraz tłumaczenia ludzkiej frazy na stan Python-.

Rozmontowaliśmy `and`, `or` oraz `not` osobno — teraz połączymy je razem i przeanalizujemy, jak Python rozumie bardziej skomplikowane warunki.

logicheskie_operatory.py

```
age = 20
has_ticket = True

if age >= 18 and has_ticket:
    print(„Proszę wejść do sali.”)

is_weekend = False
is_holiday = True
if is_weekend or is_holiday:
    print(„Nie musisz dziś iść do pracy.”)

if not has_ticket:
    print(„Najpierw musisz kupić bilet.”)
```

Priorytet operatora

Kiedy w jednym warunku spotykają się `and` oraz `or` razem, Python najpierw oblicza `not`, więc `and`, a dopiero potem `or` — tak jak mnożenie odbywa się przed dodawaniem do Arytmetyka:

not	jest wykonywany jako pierwszy
and	jest wykonywany przez drugiego
or	wykonywany jest na końcu

Priorytet operatorów boolowskich jest podobny do +/- i */÷ w arytmetyce

prioritet.py

```
print(True or False and False)
#and zostanie wykonany pierwszy: False and False -> False
# to or: True or False -> True
```

Nawiasy pomagają odczytywać skomplikowane warunki
Nie zmuszaj czytelnika (i siebie w przyszłości) do pamiętania tabeli priorytetów. Gdy warunków jest wiele, nawiasy czynią kolejność wyraźną: `is_admin or (is_owner and is_active)` jest jednoznacznie czytelna, w przeciwieństwie do opcji bez nawiasów.

Przetłumacz ludzki język na stan — i odwrotnie

Ludzkie Słowo	Python-stan
„wiek nie mniejszy niż 18”	`age >= 18`
„temperatura poniżej zera”	`temperature < 0`
„Nazwa nie jest pusta”	`name` (wystarczy sama nazwa — truthiness)
„odpowiedź brzmi: tak lub yes”	`answer in ("да", "yes")`
„dorosły i z mandatem”	`age >= 18 and has_ticket`

To podstawowa umiejętność: nauczyć się formułować pytanie w ludzkim języku, oraz następnie przetłumacz to na Python-condition, a odwrotnie, odczytaj gotowy warunek jako zdanie.

Praktyka: and, or, not razem i priorytet
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/09-04/index.html)

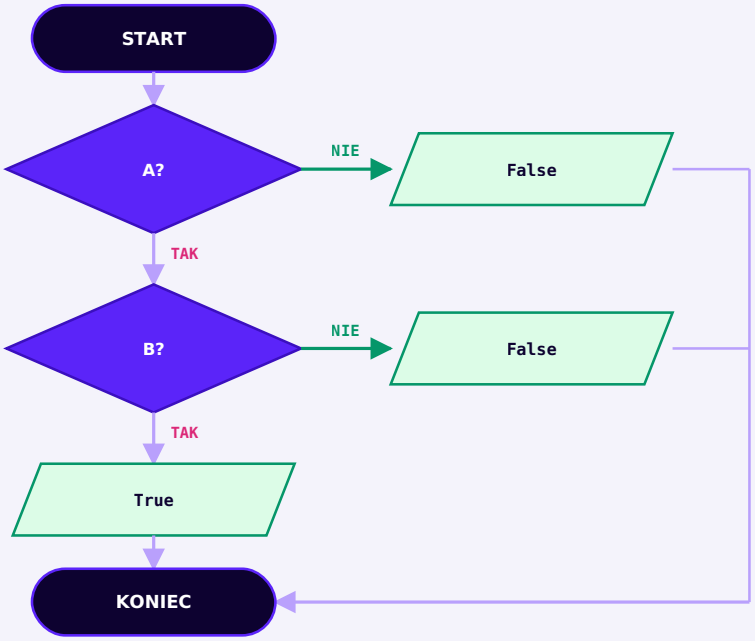
Short-circuit: Python czasem leniwy

and i or nie zawsze oceniają oba operandy — i można wykorzystać tę właściwość do ochrony przed błędami.

Python czasem nie sprawdza całego stanu

U `and` oraz `or` jest ważna Własność — **short-circuit** (zwarcie): Python oblicza sekundę operand, tylko jeśli naprawdę jest to konieczne do odpowiedzi.

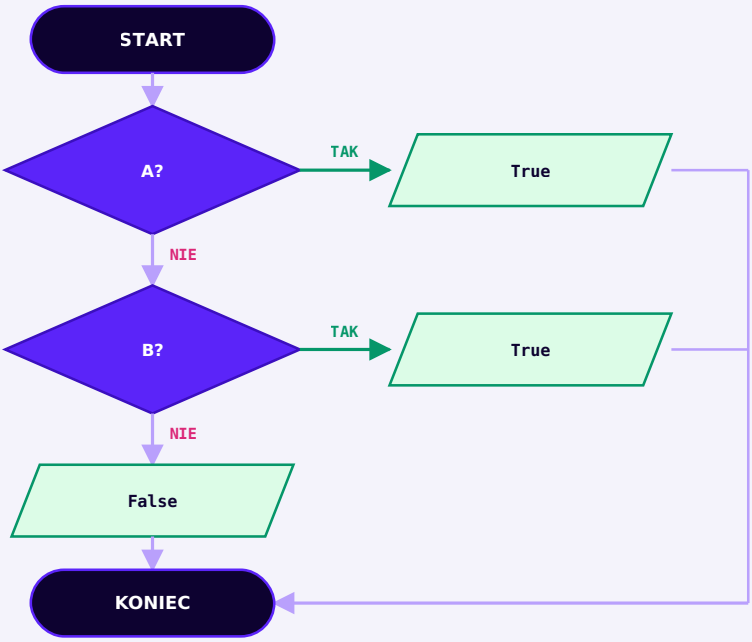
and: jeśli A już False



and: NO w dowolnym kroku natychmiast daje False – drugi warunek sprawdza się tylko wtedy, gdy pierwszy jest prawdziwy

Jeśli `A` fałszywe, wszystko `A and B` już musi być fałszywe, niezależnie od `B`. Python to rozumie i nie rozumie poświęca czas na obliczanie `B`.

or: jeśli A już True



or: TAK na dowolnym etapie natychmiast daje True – drugi warunek jest sprawdzany tylko wtedy, gdy pierwszy jest fałszywy

Podobnie: jeśli `A` prawdziwe, całe `A or B` musi już być prawdą.

Prawdziwa korzyść: Ochrona przed błędami

To nie tylko optymalizacja szybkości — czasem to właśnie ona ratuje program przed awarią:

short_circuit_zashchita.py

```
name = ""

if name and name[0] == "A":
    print(„Nazwa zaczyna się na A”)
else:
    print(„Warunek niespełniony”)
```

Jeśli `name` jest pustym łańcuch znaków, jest falsy sam w sobie (rozdział 8), Zatem `name and ...` już jest fałszywe i Python **nawet nie próbuje** obliczyć `name[0]`. Gdyby kolejność była odwrotna...

bez_zashchity.py

```
name = ""
print(name[0] == "A" and name)
# IndexError: string index out of range!
```

Kolejność operandów ma znaczenie

`name and name[0] == "A"` jest bezpieczne: najpierw sprawdza się, czy name wcale nie jest pusty. `name[0] == "A" and name` jest niebezpieczne: Python spróbuje to rozgryźć `name[0]` nawet przed jakimkolwiek sprawdzeniem pustki i otrzyma `IndexError` dla pustej linii (rozdział 8, sekcja o indeksach).

★★ Zadanie samodzielne

Zabezpiecz Ostatnią Kontrolę Postaci

Napisz warunek, który bezpiecznie sprawdza, czy niepusty łańcuch znaków text kończy się na znak '!' — tak, aby dla pustego łańcuch znaków nie pojawiał się IndexError.

Praktyka: short-circuit i ochrona przed błędami

interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/09-17/index.html)

in / not in jako warunek

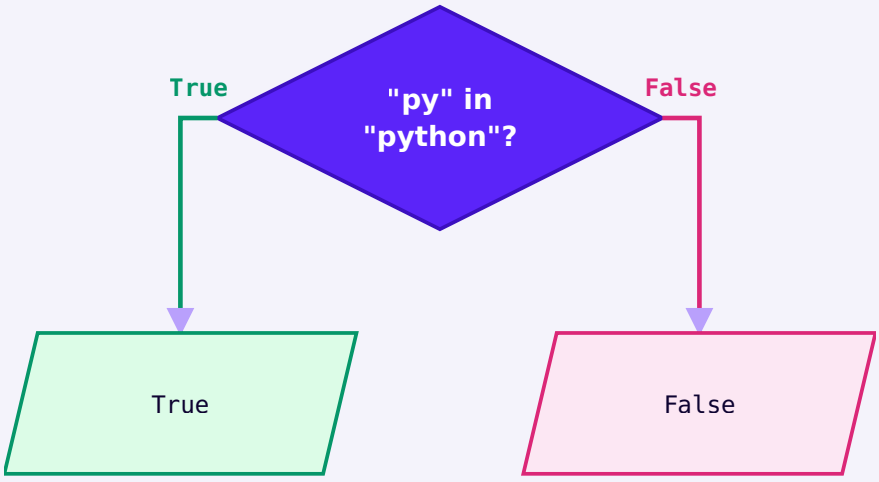
Rozdział 8 Sprawdzanie zdarzeń – teraz bezpośrednio wewnątrz if. warunków

Sprawdzenie wystąpień jako warunek

Poznaliśmy się z `in` / `not in` w Rozdział 8 — teraz używamy ich bezpośrednio w warunkach `if`.

in_kak_uslovie.py

```
text = "python"
print("py" in text)        # True
print("java" not in text)  # True
```



in pytanie „czy znaczenie zostało odnalezione w środku”

Grupa dopuszczalnych odpowiedzi

Częste zadanie: sprawdzić, czy odpowiedź użytkownika mieści się w zestawie przyjętych opcji. Na razie nie studiowaliśmy dokładnie list (to będzie w rozdziale 10-11), ale już możemy użyć prostej grupy wartości w nawiasach — **kondukt**:

gruppa_otvetov.py

```
answer = input(„Kontynuować?”).strip().lower()

if answer in („tak”, "yes", "y"):
    print(„Kontynuujemy”)
elif answer in („nie”, "no", "n"):
    print(„Stopping”)
else:
    print(„Nie zrozumiałem odpowiedzi”)
```

Co to jest ("tak", "yes", "y")
To **kondukt** to grupa wartości w nawiasach. `answer in (...)` sprawdza: „czy answer pokrywa się przynajmniej z jedną z wymienionych wartości?”

Uważaj na in podciągu
`"да" in "давай"` również daje `True` bo `in` dla łańcuch znaków poszukuje PODCIĄGU zamiast dokładnego dopasowania całego słowa. Aby porównać z zestawem opcji, użyj wartości krotk/lista, jak w powyższym przykładzie, zamiast sprawdzania „jeden łańcuch znaków wewnątrz drugiego”

Praktyka: in / not in w warunkach
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji
Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/09-18/index.html)

is wady ==

Dwóch podobnych operatorów pyta o zupełnie różne rzeczy – i jest dokładnie jeden właściwy powód, by is użyć właśnie teraz.

Dwa różne pytania

`==` oraz `is` wyglądają podobnie, ale pytają o zupełnie różne rzeczy:

==

„Czy wartości są równe?”

is

„Czy to ten sam obiekt?”

Głównym praktycznym zastosowaniem jest None

W zdecydowanej większości przypadków liczby, łańcuch znaków i zwykłe porównania ogólnie wymagają Dokładnie tak `==`. Właściwe i naprawdę ważne zastosowanie `is` na tym etapie to sprawdzić `None`:

is_none.py

```
value = None

print(value is None)       # True – właściwy sposób
print(value is not None)   # False
```

Dlaczego nie value == None

Technicznie `value == None` też działa w większości przypadków – ale `is None` jest uważany za bardziej poprawny styl, Python: jednoznacznie pyta "czy to dokładnie None?", zamiast "czy znaczenie jest równe czemuś, co zachowuje się jak None?". Zgódźmy się, że zawsze będziemy używać `is None / is not None`.

Nie używaj is do liczb i łańcuch znaków

`256 is 256` lub `"a" is "a"` mogą zachowywać się nieoczekiwanie — ich wynik zależy od wewnętrznych szczegółów implementacji Python, które nie są gwarantowane i mogą się różnić w zależności od sytuacji. To nie jest coś, czego trzeba się nauczyć teraz: aby porównywać wartości, zawsze używaj `==`. Pomysł „porównanie po obiekcie” powróci szczegółowo w rozdziale o OOP.

Praktyka: is kontra==
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

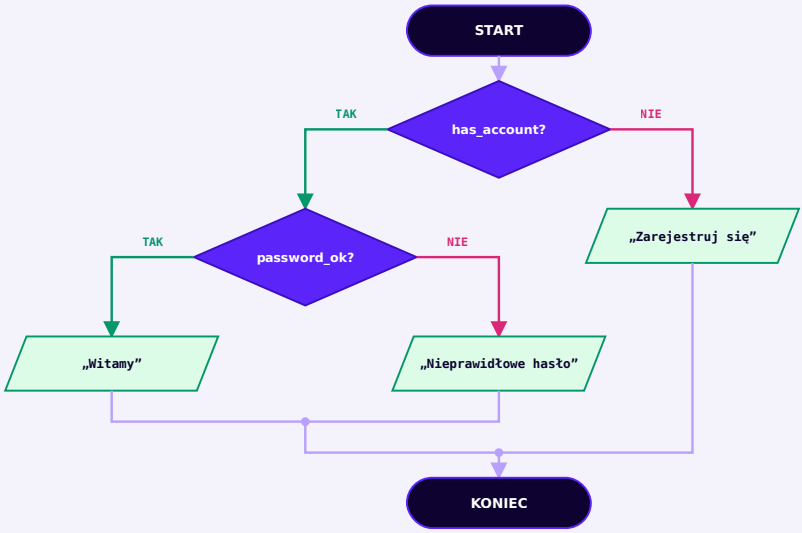
Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/09-19/index.html)

Zagnieżdżone warunki

Warunek w warunku — i kiedy warto uprościć go jednym and zamiast głębokiego zagnieżdżenia.

Stan w stanie

Wewnątrz jednostki `if` może być jeszcze jeden `if` nazywa się **warunek zagnieżdżony**. Rozwiążemy zadanie logowania do konta w całości, jednym schematem.



Warunek zagnieżdżony: password_ok sprawdza się tylko wewnątrz has_account = True gałęzi

vlozhennye_usloviya.py

```
has_account = True
password_ok = False

if has_account:
    if password_ok:
        print(„Witamy”)
    else:
        print(„Nieprawidłowe hasło”)
else:
    print(„Zarejestruj się”)
```

Poziomy wcięć = poziomy zagnieżdżenia

Każdy dodatkowy poziom wcięcia oznacza głębszy poziom warunkowej kontroli:

urovni_otstupa.py

```
# Poziom 0
if has_account:
    # Poziom 1
    if password_ok:
        # Poziom 2
        print("...")
```

Nie angażuj się w głębokie zagnieżdżenie
Trzy lub cztery poziomy zagnieżdżonych if w rzędzie („piramida” `and` – jak pokazano poniżej.

Upraszczamy: zagnieżdżenie → and

Warunki zagnieżdżenia vs jednym and

Zagnieżdżone if

```
if has_account:
    if password_ok:
        print("Login")
```

Jeden if z and

```
if has_account and
password_ok:
    print("Login")
```

Co używać dzisiaj: Obie opcje są poprawne i w tym prostym przypadku dają taki sam rezultat. Wersja zagnieżdżona jest wygodna, gdy każda gałąź potrzebuje SWOJEGO osobnego przetwarzania (na przykład różne komunikaty dla „brak konta” i „niepoprawne hasło”, jak w przykładzie poniżej). Jeden `and` jest bardziej zwarta, gdy obie gałęzie są potrzebne tylko razem — wybór między nimi zależy od czytelności, a nie od „poprawności”

★★ Zadanie samodzielne

Trzeci poziom

Dodaj trzeci poziom zagnieżdżenia: wewnątrz „Witamy” sprawdź również is_active (konto nie zablokowane) — jeśli False, wyświetl „Konto zablokowane”.

Praktyka: Warunki zagnieżdżone

interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/09-20/index.html)

Projektowanie warunku: input, walidacja, granice

Powtarzalny sposób podejścia do każdego warunku — od wymagań po przetestowaną wartość graniczną.

Walidacja to nie to samo co rozwiązanie

Przed podjęciem decyzji na podstawie wprowadzonych danych musisz sprawdzić, czy dane znaczące. To są dwa różne kroki:

- **Walidacja** — czy wkład jest akceptowalny? (np. czy to w ogóle liczba?)
- **Rozwiązanie** - Co zrobić z dozwolonym wejściem?

validaciya_i_reshenie.py

```
age_text = input(„Wiek: ”).strip()

if age_text.isdigit():          # 1. walidacja
    age = int(age_text)
    if age >= 18:                #2. Rozwiązanie
        print(„Dostęp dozwolony”)
    else:
        print(„Dostęp zabroniony”)
else:
    print(„To nie wygląda jak liczba”)
```

Pełna obsługa błędów — przed nami

Tutaj sprawdzamy ręczne wejście przez `isdigit()` (Rozdział 8). Obecny mechanizm obsługi wyjątków to `try/except` — pojawi się w osobnym rozdziale później; na razie wystarczy to proste podejście.

Granice są źródłem większości błędów

„Dozwolone są wartości od 1 do 10 włącznie”

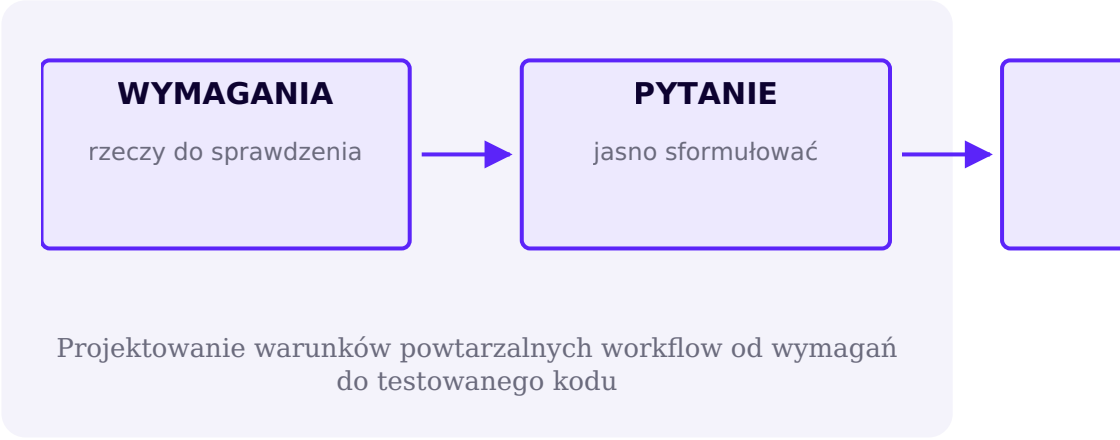
granicy.py

```
number = 1
print(1 <= number <= 10)   #True – Zgadza się, zawiera
krawędzie
print(1 < number < 10)     #False – TRAP: Odrzuceni 1 i 10!
```

Znaczenie	1 <= number <= 10	Komentarz
0	False	poniżej granicy
1	True	dokładnie na granicy — wchodzi
10	True	dokładnie na granicy — wchodzi
11	False	powyżej granicy

Off-by-one — klasyczny błąd
Jeśli sformułowanie mówi „inkluzywnie” `<` / `>` zamiast `<=` / `>=` wartości brzegowe są niezauważalnie odrzucane. Ten błąd nazywamy **off-by-one** („błąd o jednego”

Powtarzalny sposób projektowania warunku



Projektowanie warunków powtarzalnych workflow od wymagań do testowanego kodu

Zanim napiszesz kod:

1. Jakie pytanie zadajemy?
2. Jakie wartości w nim uczestniczą?
3. Co oznacza efekt True?
4. Co powinnam zrobić, jeśli True? A jeśli False?
5. Czy jest więcej niż dwa scenariusze?
6. Jakie są tutaj wartości graniczne?

Zawsze testuj **wartość tuż poniżej granicy, dokładnie na granicy i tuż nad nią** prosta zasada, która wykrywa ogromną część błędów logicznych, zanim do nich dojdzie do użytkownika.

★★ Zadanie samodzielne
Zaprojektuj go samodzielnie
Wymóg: „Rabat przysługuje przy kwocie zakupu od 1000 rubli”. Przejdź przez cały workflow — od pytania po przetestowany kod — i sprawdź wartości graniczne 999, 1000, 1001.

Praktyka: potwierdzenie, granice i off-by-one
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji
Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/09-21/index.html)

Debugowanie błędów logicznych

Program może działać bez żadnego błędu i nadal dawać błędny wynik. Nauka metodycznego znajdowania takich błędów.

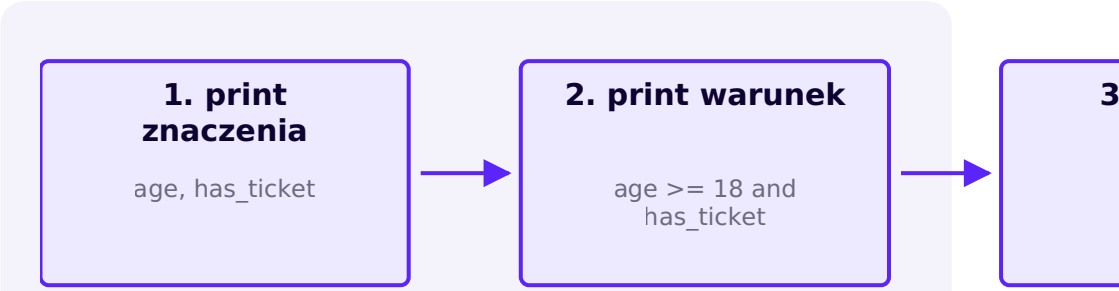
Błąd logiczny — to nie jest błąd Python

Program `if age > 18:` jest wykonane perfekcyjnie i nie zdradza Nie ma wątpliwości. Ale jeśli zgodnie z warunkami problemu „dostęp od 18. roku życia” **błąd logiczny**: Python nie rozumiem, co miałeś na myśli.

Typ błędu	Co się dzieje	Przykład
SyntaxError	Python nawet nie potrafi zacząć wykonywać kodu	`if age >= 18` (zapomniany dwukropek)
Runtime- błąd	program się zaczyna, ale mieści się w środku	`age_text + 5` (str + int)
Błąd logiczny	program wykonuje się poprawnie, ale wynik jest błędny	`if age > 18` zamiast `>=`

Metoda debugowania warunków

Gdy wynik warunku wydaje się niewłaściwy, nie zgaduj — wyprowadź wartości pośrednie:



Metoda debugowania warunków – od wyświetlania wartości po sprawdzanie każdej części osobno

otladka_uslovij.py

```
age = 20
has_ticket = False
eligible = age >= 18 and has_ticket

print("age >= 18:", age >= 18)
print("has_ticket:", has_ticket)
print("eligible:", eligible)
# age >= 18: True
# has_ticket: False
# eligible: False
```

Rozkładając skomplikowany stan na części i dedukując każdą osobno, możesz od razu zobaczyć, CO dokładnie sprawiło, że wynik był fałszywy.

Znacząco nazywać zmienne booleowskie

Dobre imię	Zła reputacja
`is_ready`	`flag1`
`has_ticket`	`thing`
`can_enter`	`value2`
`needs_update`	`x`

Konsolki `is_`, `has_`, `can_`, `needs_` - opcjonalne składnia, ale przydatna zasada: od razu widzisz, co zmienna przechowuje `True` / `False`.

Zapisz zespolony warunek do zmiennej

imenovannoe_uslovie.py

```
can_enter = age >= 18 and has_ticket

if can_enter:
    print(„Przechodzicie”)
```

`can_enter` czyta od razu – nie trzeba składać ponownie ekspresja `age >= 18 and has_ticket` za każdym razem, gdy Wróć do tego kodu.

TROCHĘ GŁĘBIEJ TO WYRAŻENIE WARUNKOWE

Gdy if/else jest już dobrze zrozumiały, warto znać zwartą formę dla PROSTEGO wyboru wartości: `status = "adult" if age >= 18 else "minor"`. To nie jest zamiennik zwykłego if/else — jedynie wygoda dla jednej linii.

match/case na przyszłość

Istnieje także porównanie strukturalne w Python `match/case` w niektórych sytuacjach wielozmiennych. Przeanalizujemy to szczegółowo później — na razie wystarczy wiedzieć, że istnieje.

Praktyka: Debugowanie błędów logicznych

interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/09-22/index.html)

Mini-projekt – Gra Zgadnij Liczbę

Pierwsza prawdziwa gra w książce polega na tym, że komputer zgaduje liczbę, a ty próbujesz zgadnąć.

Pierwsza prawdziwa gra w tej książce! Komputer zgaduje liczbę, ty wpisujesz jedną Alternatywnie, program pokazuje, czy dobrze zgadłeś, czy nie.

Najpierw — jak to wygląda dla gracza

terminal_dialog.txt

```
Я загадал число от 1 до 10.

Ваш вариант: 7
Слишком большое!
Я загадал: 4
```

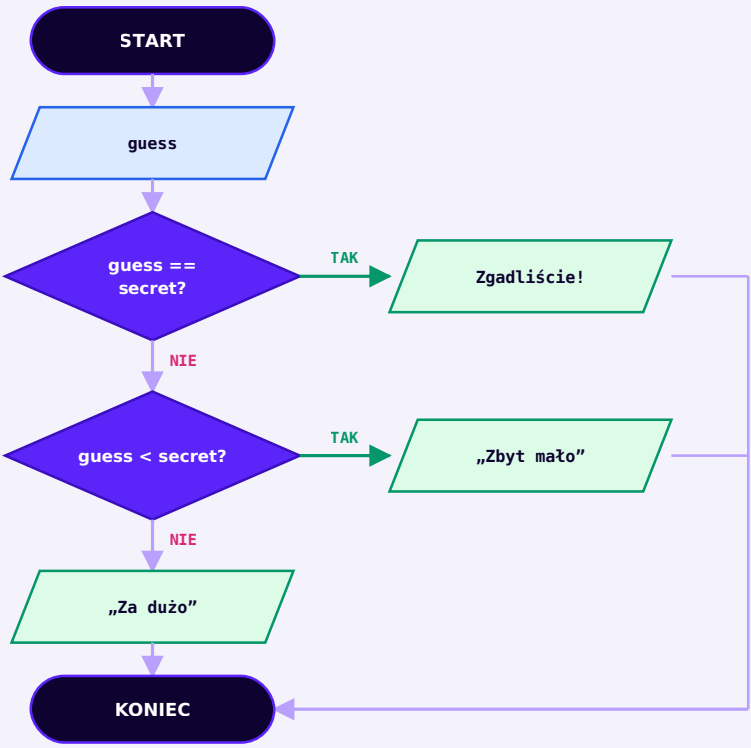
Albo, jeśli zgadliście za pierwszym razem:

terminal_dialog_win.txt

```
Я загадал число от 1 до 10.

Ваш вариант: 4

   Вы угадали!
```

Jedna próba — i to w porządku
Jeszcze nie przeszliśmy przez pętle (Rozdział 10), więc gracz ma dokładnie jedną próbę. Pojawi się tam pełna, powtarzalna wersja z powtórką do czasu zgadywania, więc miejcie to na uwadze jako motywację do kolejnego rozdziału.

Drzewo Decyzyjne w grze



Decyzyjne drzewo gry — tłumaczymy bezpośrednio do if / elif / else

reshenie_v_kod.py

```
if guess == secret:
    print(„   Zgadłeś dobrze!”)
elif guess < secret:
    print(„Za mało!”)
else:
    print(„Za dużo!”)
```

Budowanie gry krok po kroku

Krok 1 – Zrób liczbę

shag_1.py

```
import random

secret = random.randint(1, 10)
```

Krok 2 – Przeczytaj odpowiedź gracza

shag_2.py

```
guess_text = input(„Twoja opcja: ”)
```

Krok 3 – Przelicz na liczbę

shag_3.py

```
guess = int(guess_text)
```

Zapomniane int() są źródłem błędów
Jeśli porównać `guess_text < secret` bezpośrednio (łańcuch znaków z liczbą), Python da `TypeError` - Nie można porównywać łańcuch znaków i liczby. Konwersja `int()` koniecznie.

Krok 4 – Sprawdź dokładne dopasowanie

shag_4.py

```
if guess == secret:
    print(„   Zgadłeś dobrze!”)
```

Krok 5 – Dodaj „za mało”

shag_5.py

```
if guess == secret:
    print(„   Zgadłeś dobrze!”)
elif guess < secret:
    print(„Za mało!”)
```

Krok 6 — else oznacza „za dużo”

shag_6.py

```
if guess == secret:
    print(„   Zgadłeś dobrze!”)
elif guess < secret:
    print(„Za mało!”)
else:
    print(„Za dużo!”)
```

Krok 7 - Wersja ostateczna

ugadaj_chislo.py

```
import random

secret = random.randint(1, 10)
guess = int(input(„Twoja opcja: ”))

if guess == secret:
    print(„   Zgadłeś dobrze!”)
elif guess < secret:
    print(„Za mało!”)
else:
    print(„Za dużo!”)

print(f"Wylosowana liczba była: {secret}")
```

Stała liczba dla przykładowych kursów
W przykładach dokumentacji czasami używa się stałej liczby zamiast `random.randint(...)` tak, by efekt był przewidywalny po pokazaniu. W prawdziwej grze nie jest to konieczne — to losowość czyni grę interesującą.

Debug-laboratorium: pięć błędów

Błąd 1 — zapomniano int()

bug_1.py

```
guess = input("Ваш вариант: ")
if guess < secret:
    ...
# TypeError: '<' not supported between instances of 'str' and
'int'
```

Poprawka: `guess = int(input(...))`.

Błąd 2 – Nieprawidłowa kolejność gałęzi

bug_2.py

```
if guess < secret:
    ...
elif guess == secret:
    ...
```

Sam się nie zepsuje, ale kolejność „najpierw niedokładna, potem dokładna”

Błąd 3 — = zamiast ==

bug_3.py

```
if guess = secret:
    ...
# SyntaxError: invalid syntax
```

Single `=` w tym warunku jest błąd składniowy, Python nawet nie uruchomi programu (Rozdział 9.5). Metoda

Błąd 4 - Zła granica

bug_4.py

```
secret = random.randint(1, 10)
# но подсказка написана как "от 1 до 9" — граница не совпадает
с кодом
```

Błąd 5 – Dwa niezależne if zamiast elif

bug_5.py

```
if guess < secret:
    print("Слишком мало!")
if guess > secret:
    print("Слишком много!")
# при guess == secret НИ ОДНО сообщение не покажется — а if/
elif/else это учитывает
```

★★ Zadanie samodzielne

Podpowiedź „gorąco/zimno"

Dodaj czwarte warunek: jeśli różnica między guess a secret jest mniejsza niż 3 (użyj abs()) — wyświetl „Bardzo blisko!" przed głównym komunikatem.

Praktyka: Zgadywanie całej gry w zgadywanie liczb

interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/09-05/index.html)

Mini-Projekty – Doradca ds. Klubu i pogody

Dwa praktyczne mini-projekty na łączenie warunków z rozdziałów 9.11–9.17.

Zrobimy dwa krótkie, ale pouczające projekty o łączeniu warunków.

Mini-projekt – „Club Python”

Wejście do klubu dozwolone tylko dla pełnoletnich gości z biletem. To czysto edukacyjne zadanie logiczne — nie prawdziwy system bezpieczeństwa!

klub_python.py

```
age = int(input(„Twój wiek: ”))
has_ticket = input(„Masz bilet? (tak/nie): ”).strip().lower()
== „tak”

if age >= 18 and has_ticket:
    print(„Witamy w Club Python!”)
elif age < 18:
    print(„Wstęp tylko od 18 roku życia.”)
else:
    print(„Potrzebuję biletu.”)
```

★★ Zadanie samodzielne

VIP bez biletu

Dodaj zmienną is_vip. VIP-guests przechodzi nawet bez zgłoszenia – użyj or dla tego wyjątku.

Mini-projekt — doradca pogodowy

Program rekomenduje na podstawie temperatury i tego, czy pada deszcz:

sovetchik_pogody.py

```
temperature = int(input(„Temperatura: ”))
is_raining = input(„Czy pada? (tak/nie): ”).strip().lower() ==
„tak”

if temperature < 10 and is_raining:
    print(„Ciepła kurtka i parasol”)
elif temperature < 10:
    print(„Ciepła kurtka”)
elif is_raining:
    print(„Tylko parasol”)
else:
    print(„Lekkie ubrania, bez parasola”)
```

Kolejność warunków przemyślana z góry
Zwróć uwagę, że najbardziej szczegółowy stan (zimno I deszcz) pojawia się jako pierwszy — dokładnie ta sama zasada „pierwszy prawdziwy wygrywa”

★★ Zadanie samodzielne
Wietrzna pogoda
Dodaj trzeci czynnik is_windy. Przy silnym wietrze dodaj do zalecenia „i czapkę” do którejkolwiek z czterech możliwości.

Praktyka: Doradca ds. klubu i pogody
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji
`Otwórz praktykę →` (https://www.cartesianschool.org/pl/practice/09-23/index.html)

Mini-projekty — klasyfikator ocen i interpretator poleceń

Uporządkowane elif-łańcuchy i powiązanie wierszy z rozdziału 8 z warunkami tego rozdziału.

Mini-projekt – Klasyfikacja Punktów

Przełóż wynik liczbowy (0–100) na poziom — przykład edukacyjny, granice kursu są warunkowe, a nie odzwierciedlają rzeczywiste standardy edukacyjne:

score	poziom
0–49	niezadowalające
50–74	satysfakcjonujące
75–89	dobrze
90–100	świetnie

klassifikator_ballov.py

```
score = int(input(„Twój wynik (0-100): ”))

if score >= 90:
    level = „doskonały”
elif score >= 75:
    level = „dobrze”
elif score >= 50:
    level = „satysfakcjonujące”
else:
    level = „niezadowalające”

print(f"Poziom: {level}")
```

★★ Zadanie samodzielne

Sprawdź wszystkie granice

Uruchom klasyfikator na wartościach 0, 49, 50, 74, 75, 89, 90, 100 — upewnij się, że każdy z nich mieści się w oczekiwanym poziomie (sekcja 9.19 dotycząca granic).

Mini-projekt - Interpreter poleceń tekstowych

Użytkownik wprowadza polecenie tekstowe, a program odpowiada. Bezpośrednie połączenie między łańcuch znakówme z rozdziału 8 i warunki tego rozdziału:

interpretator_komand.py

```
command = input(„Wprowadź polecenie (start/stop/
help):”).strip().lower()

if command == "start":
    print(„Start...”)
elif command == "stop":
    print(„Zatrzymanie...”)
elif command == "help":
    print(„Dostępne komendy: start, stop, help”)
else:
    print(f"Nieznane dowództwo: {command}")
```

W następnym rozdziale

Teraz program reaguje na JEDEN rozkaz i kończy działanie. Z pętlą `while` (Rozdział 10) będzie mógł czekać na kolejne polecenia jak prawdziwa mini-konsola – aż użytkownik wpisze „exit”

Podstawowa praktyka

Niezależność rejestru

Sprawdź interpretator na wejściu „START”, „ Stop ”, „HELP” — upewnij się, że .strip().lower() sprawia, że program jest niewrażliwy na wielkość liter i spacje.

Praktyka: klasyfikator ocen i interpreter poleceń

interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

`Otwórz praktykę →` (https://www.cartesianschool.org/pl/practice/09-24/index.html)

Warunki nadal się kumulują!

Mapa końcowa: które narzędzie wybrać do jakiego zadania — i co dalej.

Warunki mogą się kumulować tak bardzo, jak tylko chcesz — `elif`, `and` / `or` i gniazdowanie pozwalają Przedstawić arbitralnie złożone rozwiązanie. Podsumujmy rozdział.

Mapa decyzji

Mapa Decyzji – Czego użyć?

Trzeba zadać jedno pytanie tak/nie?	→ **`if`**
Potrzebne są dwie alternatywy?	→ **`if / else`**

Czy potrzebujesz kilku wzajemnie wykluczających się opcji?	→ `if / elif / else`
Czy WSZYSTKIE warunki muszą być prawdziwe?	→ `and`
CZY PRZYNAJMNIEJ JEDEN prawdziwy stan wystarczy?	→ `or`
Czy trzeba odwrócić ten stan?	→ `not`
Musisz sprawdzić, czy pojawia się SMS/grupa?	→ `in`
Czy powinienem porównać to z None?	→ `is`
Potrzebujesz porównać wartości?	→ `==, !=, <, >, <=, >=`
Trzeba sprawdzić zasięg?	→ `łańcuch porównań`

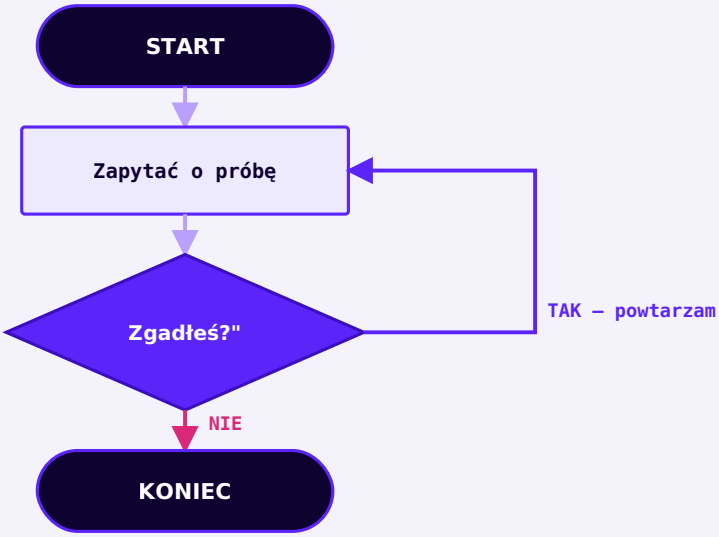
Co możemy teraz zrobić

Czego nauczyliśmy się w tym rozdziale

- Algorytm — zrozumiała sekwencja działań; składa się z trzech struktur: sekwencja, rozgałęzienie, powtarzanie.
- Typ `bool` sklepy `True` lub `False` ; Sześć operatorów porównawczych zamienia wartości w bool.
- Truthiness: wartości zero i puste zachowują się jak kłamstwo, wszystko inne zachowuje się jak prawda (ale tekst `"False"` jest prawdą!).
- `if` / `elif` / `else` wybierają dokładnie JEDNĄ gałąź — w przeciwieństwie do kilku niezależnych `if` , które mogą zadziałać wszystkie naraz.
- `and` , `or` , `not` warunki łączą; and/or ma short-circuit Python nie zawsze oblicza drugi operand.
- `is` oraz `==` — różne kwestie; `is None` jest właściwy sposób sprawdzania None.
- Warunki można inwestować, ale często należy je uprościć poprzez and; Zawsze sprawdzaj wartości brzegowe.

Most do rozdziału 10

Teraz program potrafi podjąć JEDNĄ decyzję. Ale gry i prawdziwe aplikacje zazwyczaj podejmują decyzje PONOWNIE I PONOWNIE. „Zgadnij liczbę” na razie daje tylko jedną próbę — w następnym rozdziale nauczymy program **powtarzać** akcji, a gra będzie mogła kontynuuj, aż zgadniesz liczbę.



Rozdział 10: Ten sam backward-arrow dodaje powtórzenie – „Zgadnij liczbę”

Praktyka: Mapa decyzji
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/09-06/index.html)

ROZDZIAŁ 10 · STR. 502

Trochę automatyzacji!

Pętle for i while jako trzecia struktura algorytmu po rozgałęzieniu z rozdziału 9, range() i enumerate() bez mitów, break/continue/loop-else, zagnieżdżonych pętli, debugowania typowych błędów pętli — i wreszcie prawdziwej automatyzacji wszystkich rysunków z rozdziałów 6-7.

Cykle magiczne! Cykle for

Wreszcie prawdziwa automatyzacja: zamiast powtarzać kod ręcznie, prosimy Python, by powtórzył go sam.

Zanim Python: powtarzalność wokół nas

Powtarzamy czynności cały czas, nawet nie myśląc: mycie zębów tymi samymi ruchami, Maszerujemy tymi samymi krokami, śpiewamy zwrotkę za zwrotką tej samej melodii. **Powtarzalność jest odrębnym, niezależnym sposobem organizowania działania**na równi z "zrób jedną rzecz po drugiej" oraz "wybierz jedną z dwóch ścieżek".

Trzy struktury algorytmu: przypominamy sobie

W rozdziale 9 omówiliśmy pierwsze dwie z trzech podstawowych struktur, z których składa się każdy algorytm. Trzecia — **powtarzalność** jest tematem tego rozdziału.

Trzy struktury, z których budowany jest każdy algorytm

1 · KOLEJNOŚĆ

Kroki następują jeden po drugim

„Najpierw… potem… potem…”

Rozdział 9

2 · ROZGAŁĘZIENIE

Wybór jednej ze ścieżek według warunku

`if / elif / else`

Rozdział 9

3 · RECENZJA

Ten sam blok w kółko

`for` / `while`

Ten rozdział

Co jest nie tak z tym kodem?

Oto kwadrat z rozdziału 6 — już znany kod: cztery identyczne bloki „krok do przodu, obrót”

kvadrat_bez_cikla.py

```
artist.forward(100)
artist.right(90)
artist.forward(100)
artist.right(90)
artist.forward(100)
artist.right(90)
artist.forward(100)
artist.right(90)
```

Teraz wyobraź sobie, że nie potrzebujesz kwadratu, ale **Dwudziestu**. Skopiuj te dwie linie jeszcze 16 razy ręcznie — długo i łatwo popełnić błąd. Właśnie do takich powtarzających się bloków istnieją **pętle**: Sposób na powiedzenie Python „Powtarzaj to N raz”

Terminy pętli

Zanim napiszemy pierwszy cykl, ustalmy słowa, jakimi go opiszemy — Potem spotkają się w całym rozdziale:

Termin	Co to jest
pętla	konstrukcja, która powtarza ten sam blok kodu wielokrotnie
Korpus Cyklu	tym samym bloku kodu, który jest powtarzany (linie wcięte pod `for` / `while`)
iteracja	jednym przejściem przez treść cyklu, jednym "raz" z "powtórz N razy"
warunek	to, co pętla sprawdza, by zdecydować, czy kontynuować, czy zakończyć
licznik	zmienna, która liczy numer iteracji lub zmienia się z każdym krokiem

Pętle for

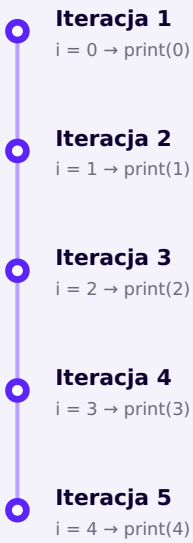
Najczęstszym typem cyklu w Python jest `for`. Razem z `range()` może powtórzyć blok kodu określoną liczbę razy:

cikl_for.py

```
for i in range(5):
    print(i)
```

Rozkładamy na słowa

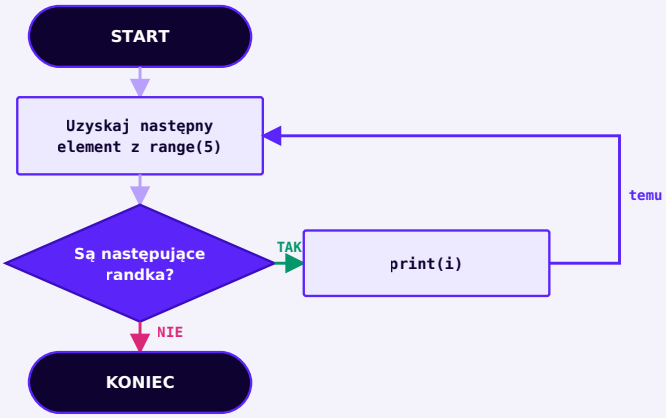
Część kodu	Co oznacza
`for`	„rozpoczęcie cyklu”
`i`	zmienna pętli — na każdym kroku pobiera kolejną wartość z `range(5)`
`in range(5)`	Źródło wartości: Liczby od 0 do 4
`:`	wtedy zaczyna się treść cyklu (z wcięciem)
`print(i)`	ciało pętli — to, co powtarza się przy każdej iteracji



for i in range(5): print(i) to pięć iteracji tego samego ciała pętli

Kanoniczny flowchart cyklu for

Ważne jest, aby od razu przyzwyczaić się do właściwego obrazu tego, co się dzieje: `for` nie "magicznie wykonuje ciała N czasie" – na każdym etapie pyta "czy jest następny element?", a jeśli tak, wykonuje ciało i ponownie pyta:



for i in range(5): print(i) - Pętla za każdym razem żąda kolejnego elementu

Trasa Egzekucji

Tabela śledzenia to sposób śledzenia tego, co dzieje się w każdej iteracji, linia po linii. My Będziemy korzystać z tej tabeli przez cały rozdział:

Iteracja	i	Zakończenie (print)
1	0	0
2	1	1
3	2	2

Iteracja	i	Zakończenie (print)
4	3	3
5	4	4

range() z różnymi argumentami — krótko

`range(5)` — od 0 do 4. `range(2, 8)` — od 2 do 7. `range(0, 10, 2)` — od 0 do 8 z krokiem 2: 0, 2, 4, 6, 8. Szczegółowa analiza znajduje się na następnej stronie.

Gdy cykl zmienna nie jest potrzebny

Jeśli samo liczba powtórzeń jest ważna, a nie wartość licznika — tradycyjnie nazywa się ją `_` (podkreślenie), sygnalizując „ta wartość świadomie nie jest używana”:

Rozdział 6 Kwadrat: 8 linijek → 2 wersy

Brak Pętli (Rozdział 6)

```
artist.forward(100)
artist.right(90)
artist.forward(100)
artist.right(90)
artist.forward(100)
artist.right(90)
artist.forward(100)
artist.right(90)
```

Z cyklem for

```
for _ in range(4):
    artist.forward(100)
    artist.right(90)
```

Co używać dzisiaj: pętla `for`. Nie jest „nowoczesny”

_ jest konwencją, a nie specjalną składnią

`_` to nazwa zmiennej powszechnej Python nie obsługuje jej w żaden specjalny sposób. To po prostu umowa między programistami: „celowo nie używamy tutaj wartości pętli.” `for x in range(4):` też zadziała, ale czytelnik kodu będzie miał mniej jasny sygnał.

Praktyka: for cykl i terminologia
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/10-01/index.html)

range() szczegóły

range() nie jest lista, lecz generatorem liczb na żądanie. Przeanalizujmy wszystkie trzy jego formy i skąd wzięła się zasada „stop nie jest uwzględniona”

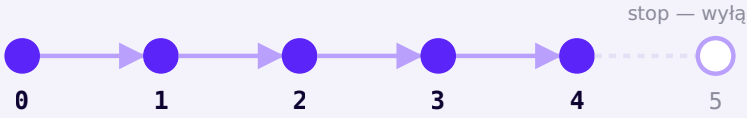
range() nie jest lista

`range()` jest jedną z najczęstszych przyczyn zamieszania wśród początkujących. Ustalmy to od razu i dokładnie.

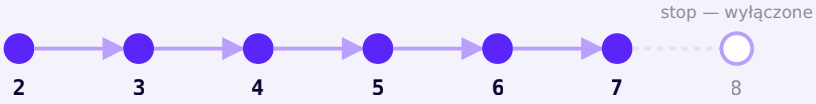
range() NIE tworzy lista
`range(5)` tworzy specjalny obiekt generatora liczb — generuje liczby pojedynczo na żądanie, zamiast przechowywać je wszystkie naraz w pamięci jako lista. `list(range(5))` naprawdę się ustąpi `[0, 1, 2, 3, 4]` — ale to jest osobna, oczywista przemiana, a nie tym, czym jest `range`.

Trzy Charakterystyczne Poziomy

Challenge	Co oznacza	Przykład
`range(stop)`	od 0 do stop, nie licząc stop	`range(5)` → 0, 1, 2, 3, 4
`range(start, stop)`	od start do stop, nie wliczając stop	`range(2, 8)` → 2, 3, 4, 5, 6, 7
`range(start, stop, step)`	od start do stop z krokiem step, nie wliczając stop	`range(0, 10, 2)` → 0, 2, 4, 6, 8



range(5) — pięć wartości: 0, 1, 2, 3, 4; sam stop=5 nie wchodzi do wyniku



range(2, 8) — od 2 do 7 włącznie; 8 nie jest uwzględnione

Dlaczego stop nie chce się włączyć

To nie jest dowolna fanaberia języka — to ta sama zasada, którą już widzieliśmy w rozdziale 8 w wycinkach łańcuch znaków. `"Python"[0:3]` bierze znaki o indeksach 0, 1, 2 — i nie obejmuje znaku o indeksie 3. `range(0, 3)` jest ułożone dokładnie tak To samo: 0, 1, 2, bez 3. Python spójne: "koniec zakresu" wszędzie oznacza "do tego miejsca, bez uwzględnienia tego."

Przydatna konsekwencja

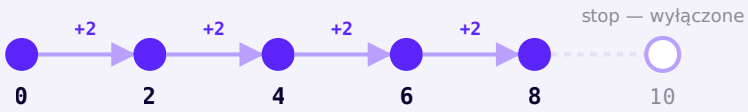
Dzięki tej zasadzie `range(n)` zawsze daje dokładnie `n` liczby oraz długość kawałka `s[a:b]` zawsze jest równy `b - a` — liczenie ilości elementów wychodzi bez dodatkowych +1/-1 w głowie.

Krok: step

Trzeci argument określa, o ile liczba rośnie (lub maleje) na każdym kroku:

shag.py

```
for number in range(0, 10, 2):
    print(number)
```



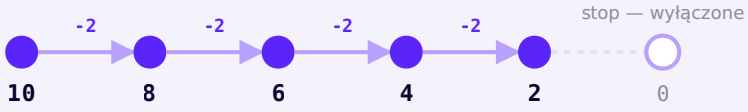
range(0, 10, 2) - Krok 2: 0, 2, 4, 6, 8

Negatywny krok

Etap może być ujemny – wtedy liczby idą w przeciwnym kierunku, ale start też stop Konieczne jest także zmienienie kolejności zgodnie z jego znaczeniem: start powinno być więcej niż stop.

otricatelnyj_shag.py

```
for number in range(10, 0, -2):
    print(number)
```



range(10, 0, -2) – odliczanie wstecz w krokach po 2: 10, 8, 6, 4, 2

Ujemny range bez ujemnego kroku jest pusty

`range(10, 0)` (bez kroku) nic nie zwróci — domyślny krok to +1, a dzięki niemu nigdy nie przejdziesz z 10 do 0. Aby zakres spadł, krok musi być ujemny i odpowiadać kierunkowi.

Krok 0 to błąd, a nie nieskończona pętla

`range(0, 10, 0)` nie zawiesza się i nie zamienia w nieskończoną pętlę — Python od razu podnosi `ValueError: range() arg 3 must not be zero` dlatego, że przy zerowym kroku nigdy nie da się przejść z start do stop.

Praktyka: range() — start, stop, krok

interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/10-09/index.html)

Indeks, wartość i enumerate()

for potrafi iterować nie tylko przez liczby, ale także licznik i akumulator, co pozwala przekształcić proste wyszukiwanie w prawdziwe narzędzie analizy danych.

for potrafi przechodzić nie tylko przez liczby

łańcuch znaków — to sekwencja znaków (rozdział 8), a więc, `for` potrafi iterować i jej — znak po znaku, bez indeksów:

perebor_strok.py

```
for letter in "Python":
    print(letter)
```

To samo dotyczy list i innych **iterowalne** obiektów — tak nazywa się wszystko, co `for` potrafi przechodzić przez jeden żywioł na raz. Struna, lista, Zakres `range()` wszystkie są obiektami iteracyjnymi; Dokładny mechanizm techniczny, jak dokładnie Python to robi „pod maską”

Indeks vs Wartość

W pętli `for letter in "Python":` dostajemy sam symbol ("wartość"), ale nie numer konta ("indeks"). Czasami liczba jest ważna, na przykład dla wyjść "symbol No1: P".

Klasyczną opcją jest cykl indeksowy

cikl_po_indeksam.py

```
slovo = "Python"
for i in range(len(slovo)):
    print(i, slovo[i])
```

„Pythoniczna" wersja — enumerate()

enumerate_variant.py

```
slovo = "Python"
for i, letter in enumerate(slovo):
    print(i, letter)
```

i	letter
0	P
1	y
2	t
3	h
4	o
5	n

Kiedy używać enumerate()

`enumerate()` jest potrzebne dokładnie wtedy, gdy zarówno indeks, jak i wartość są ważne jednocześnie. Jeśli w ogóle nie potrzebujesz indeksu, użyj `for letter in slovo:`, bez niej. Pętla indeksowa (`range(len(...))`) nie jest „złym" `enumerate()` poradzi sobie krócej.

Licznik to zmienna, który się liczy

Licznik to zmienna, który rośnie z każdą iteracją (lub zmniejsza się) o stały krok, zwykle 1. Śledzi „ile razy już jesteśmy załatwione.”

schetchik.py

```
glasnye = „aeeyoueyyuya”
slovo = „programowanie”
kolichestvo = 0

for letter in slovo:
    if letter in glasnye:
        kolichestvo += 1

print(kolichestvo)
```

Symbol	letter in glasnye?	kolichestvo po kroku
p	nie	0
r	nie	0
o	Tak	1
g	nie	1
r	nie	1
a	Tak	2
...	...	...

Urządzenie pamięci masowej (akumulator) to zmienna, która gromadzi wynik

Przechowywanie jest zaprojektowany jak licznik, ale nie sumuje liczby kroków, lecz Sam wynik to suma, iloczyn, klejony sznurek:

nakopitel.py

```
chisla = [4, 8, 15, 16, 23, 42]
summa = 0

for n in chisla:
    summa += n

print(summa)
```

Iteracja	n	summa po kroku
1	4	4
2	8	12
3	15	27
4	16	43
5	23	66
6	42	108

Typowe w liczniku i na dysku: status

Przykładami są licznik i urządzenie magazynujące **stany**: zmienna, która żyje na zewnątrz cyklu, ale zmienia się w każdej iteracji wewnątrz niego. Przed cyklem stan trzeba obowiązkowo zainicjalizować (`kolichestvo = 0`) — bez tego kroku `kolichestvo += 1` zawiesza się już w pierwszej iteracji, ponieważ zmienna jeszcze nie istnieje.

Mini-Projekt Analizatora Tekstu

Złóżmy licznik i urządzenie do przechowywania w jednym małym narzędziu – liczmy samogłoski, długość najdłuższego słowa oraz łączna liczba słów w tekście:

analizator_teksta.py

```
tekst = „python to proste i proste”
slova = tekst.split()

glasnye_count = 0
for letter in tekst:
    if letter in „aeeyoueyyuya”:
        glasnye_count += 1

samoe_dlinnoe = ""
for slovo in slova:
    if len(slovo) > len(samoe_dlinnoe):
        samoe_dlinnoe = slovo

print(f"słowa: {len(slova)}, samogłoski: {glasnye_count},
najdłuższy: {samoe_dlinnoe}")
```

Trzy niezależne napędy obok siebie

Proszę zauważyć: `glasnye_count` oraz `samoe_dlinnoe` — dwa różne akumulatory, każdy ze swoim cyklem i własną inicjalizacją początkową. Łączenie wielu stanów w jednym programie to normalna rzecz, najważniejsze, żeby nie pomylić, który stan zmienia się w której pętli.

Praktyka: indeks, wartość i enumerate()

interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/10-10/index.html)

Praktyka: Mini-projekt Analizatora Tekstu

interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/10-11/index.html)

if warunki w for cyklach

Pętle i warunki świetnie ze sobą współpracują — a pętle można zagnieżdżać w sobie.

if warunki w for cyklach

`if` z rozdziału 9 działa perfekcyjnie w pętli — na każdym z nich Krok możesz podjąć własną decyzję:

if_v_cikle.py

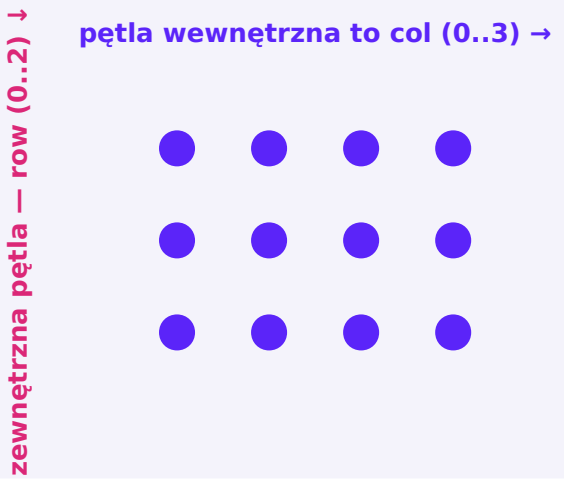
```
for number in range(1, 11):
    if number % 2 == 0:
        print(number, „— nawet”)
```

Zagnieżdżone pętle for

pętlę można umieścić w innej pętli – wtedy wewnętrzna pętla jest w pełni wykonana na każdym kroku zewnętrznej. Wyobraź sobie siatkę: zewnętrzna pętla biegnie wzdłuż łańcuch znakówm, wewnętrzna pętla biegnie wzdłuż kolumny każdego wiersza:

vlozhennye_cikly.py

```
for row in range(3):
    for col in range(4):
        print(f"({row}, {col})", end=" ")
    print()  # nowy łańcuch znaków po każdym wierszu
```



Zewnętrzna pętla to 3 wiersze, wewnętrzna pętla to 4 kolumny każda: 3 × 4 = 12 komórek

Ile razy wykona się wewnętrzna pętla?

linie×kolumny=iteracji ciała

Wzór liczby iteracji zagnieżdżonej pętli

row	col	print(...) raz zadziałało. Nie.
0	0	1
0	1	2
0	2	3
0	3	4
1	0	5
...	...	...
2	3	12

Zagnieżdżone pętle — częsta przyczyna nieoczekiwanej powolności

Cykl zewnętrzny wykonywany jest 3 razy, cykl wewnętrzny 4 razy *na każdym z nich* etap zewnętrznego — całkowity `print(...)` w środku się sprawdzi `3 * 4 = 12` raz. Jeśli obie pętle idą nie do 3-4, lecz do tysięcy, końcowa liczba iteracji mnoży się — i program może działać zdecydowanie dłużej, niż się wydaje na pierwszy rzut oka.

Mini Projekt: Tablica mnożenia

Klasycznym problemem zagnieżdżonej pętli jest wydrukowanie tablicy mnożenia od 1 do 5:

tablica_umnozheniya.py

```
for a in range(1, 6):
    for b in range(1, 6):
        print(a * b, end="\t")
    print()
```

Wystarczy jeden lub dwa przykłady
Nie będziemy konkretnie mnożyć przykładów wzorców na zagnieżdżonych cyklach — tablica mnożenia (i powyższa siatka) dają wystarczającą intuicję, by samodzielnie czytać i zapisywać podobne konstrukcje.

Pętla + łańcuch znaków + warunek

Zagnieżdżanie działa także tam, gdzie wewnętrzna „pętla”

cikl_stroka_uslovie.py

```
slova = ["python", „kod”, „cykl”]

for slovo in slova:
    bukvy_o = 0
    for letter in slovo:
        if letter == „o”:
            bukvy_o += 1
    print(slovo, "—", bukvy_o)
```

Praktyka: warunki i zagnieżdżone pętle
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/10-02/index.html)

Praktyka: mini-projekt „Tabliczka mnożenia”
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/10-12/index.html)

Praktyka: Ile iteracji jest w zagnieżdżonej pętli?
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/10-24/index.html)

Cykle while

for potrafi przechodzić nie tylko przez liczby, ale while powtarza tę czynność, dopóki warunek pozostaje spełniony, bez względu na to, ile razy to potrwa.

Gdy liczba powtórzeń nie jest znana z góry

`for` jest świetne, gdy wiemy z góry, ile razy Musisz powtórzyć czynność – "4 boki kwadratu", "wszystkie litery słowa". A co jeśli Liczba powtórzeń nie jest znana z góry – na przykład: "Powtarzaj, aż użytkownik zgadnie liczba"? Dla takich problemów istnieje drugi typ cyklu — `while`.

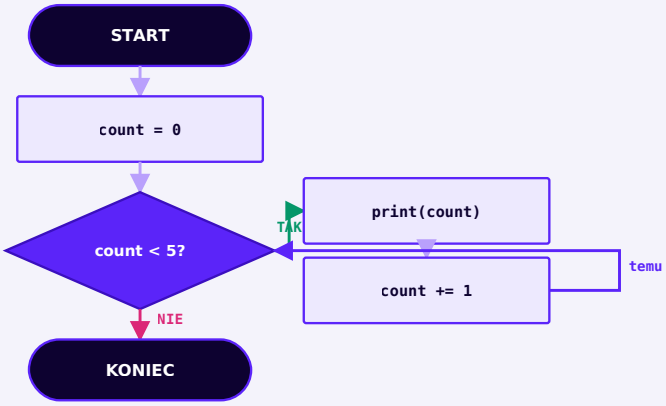
Cykle while

W przeciwieństwie do `for`, `while` powtarza się **ten warunek nadal jest prawdziwy**:

cikl_while.py

```
count = 0
while count < 5:
    print(count)
    count += 1
```

Kanoniczne flowchart cyklu while



while count < 5: – sprawdzanie stanu PRZED każdym ciałem, w tym pierwszym

Trzy części cyklu while

Część	Kodeks	Rola
Inicjalizacja	`count = 0`	przygotowuje stan PRZED pierwszym sprawdzeniem warunku
Stan	`count < 5`	sprawdzane przed każdą iteracją, w tym pierwszą
Aktualizacja	`count += 1`	zmienia stan tak, aby warunek w końcu stał się fałszywy

Iteracja	Sprawdzanie count lt 5	Podsumowanie	count po kroku
1	0 < 5 → tak	0	1

Iteracja	Sprawdzanie count lt 5	Podsumowanie	count po kroku
2	1 < 5 → tak	1	2
3	2 < 5 → tak	2	3
4	3 < 5 → tak	3	4
5	4 < 5 → tak	4	5
—	5 < 5 → nie	pętla zakończona	—

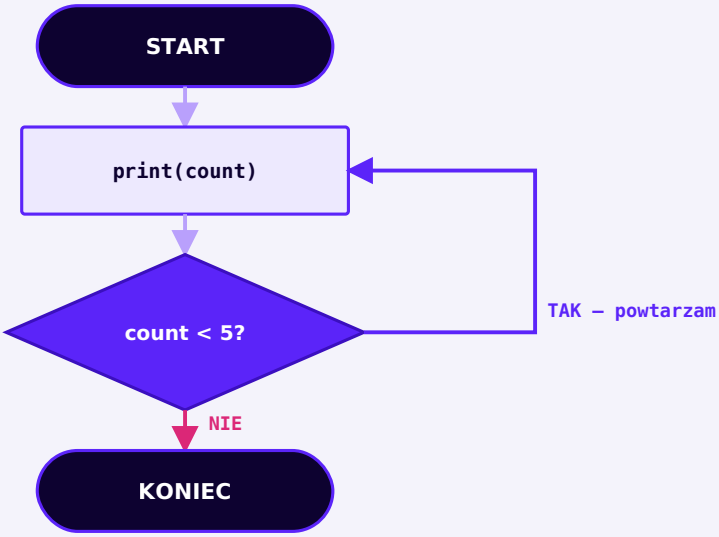
while jest pre-test cykl
Stan w `while` sprawdzane **do** ciało, co oznacza, że jeśli warunek jest początkowo fałszywy, ciało nie zostanie w ogóle wykonane (zero iteracji). W Python nie ma osobnej pętli „najpierw wykonaj, potem sprawdź” `do/while`) — jeśli takie zachowanie jest potrzebne, symulują je poprzez `while True z break` na końcu ciała (tę technikę zobaczymy na następnej stronie).

Nieskończona pętla – celowo i przypadkowo

Dokładnie przyjrzyj się temu kodowi:

beskonechnyj_cikl.py

```
count = 0
while count < 5:
    print(count)
    # zapomniałem count += 1!
```



Stan (count) nie zmienia się między iteracjami – warunek ten pozostaje prawdziwy na zawsze

Jak zatrzymać zamrożony program

Jeśli program utknął w nieskończonej pętli, kliknij `Ctrl+C` (lub przycisk „Stop”/„Interrupt kernel” w Jupyter). Zawsze sprawdzaj, co jest w środku `while` istnieje krok, który prędzej czy później sprawi, że stan zostanie fałszywy.

while True też jest normalne

`while True` tworzy pętlę, której stan sam nigdy się nie powtórzy nie stanie się fałszywym — jedynym wyjściem: `break` wewnątrz ciała. Jest to powszechna i absolutnie poprawna technika, gdy stan zatrzymania jest bardziej naturalny Sprawdź na środku ciała, a nie na początku – użyjemy tego na następnej stronie.

while True samo w sobie nie jest błędem

`while True:` nie jest samo w sobie „złą praktyką” — to świadome narzędzie. Problem pojawia się dopiero wtedy, gdy wewnątrz zapomni się ustawić `break` – wtedy ten cykl naprawdę nigdy się nie skończy.

Iteracja po łańcuchach znaków — przypomnienie

Struna to łańcuch znaków (Rozdział 8) — `for` potrafi iterować po niej bezpośrednio, bez indeksów (szczegółowa analiza indeksu i wartości — na stronie o enumerate()):

perebor_strok.py

```
for letter in "Python":
    print(letter)
```

for czy while?

Wytyczne, a nie sztywne reguły

Liczba powtórzeń jest znana wcześniej („narysuj 4 boki”, „przejdź przez wszystkie litery”) → **for**

Liczba powtórzeń nie jest znana z góry („dopóki użytkownik nie zgadnie” → **while**

Bardziej naturalne jest sprawdzanie stanu zatrzymania się w środku ciała → **while True + break**

Musisz przejść przez gotową sekwencję (łańcuch znaków, lista, range) → **for**

Oba rodzaje pętli są równie potężne pod względem Python — wybór wpływa na czytelność kodu, a nie na to, co w ogóle jest możliwe

Praktyka: pętle while
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/10-03/index.html)

Praktyka: for czy while – którą wybrać wybrać?
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/10-23/index.html)

Przerwij misję!" break i continue

Dwa sposoby sterowania pętlą od środka — zatrzymać ją całkowicie lub pominąć jeden krok — i specjalny blok else, który wie, czy break.

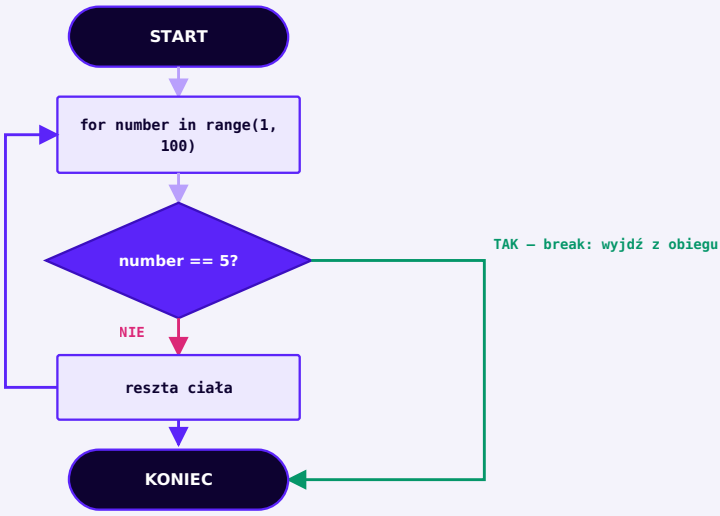
Czasem trzeba wyjść z pętli wcześniej albo pominąć jeden krok, nie kończąc pętli W całości — są dwa słowa kluczowe i jeden szczególny sposób, by sprawdzić, czy break.

`break` to całkowicie przerwać ten cykl

break.py

```
for number in range(1, 100):
    if number == 5:
        break  # cykl zatrzymuje się na dobre
    print(number)
```

Będzie wydawać tylko 1, 2, 3, 4 – tak szybko, jak `number` staje się 5, pętla zostaje przerwana bez oczekiwania na pozostałe 94 wartości `range()`.



break wychodzi z obiegu NATYCHMIAST – ale nie z programu

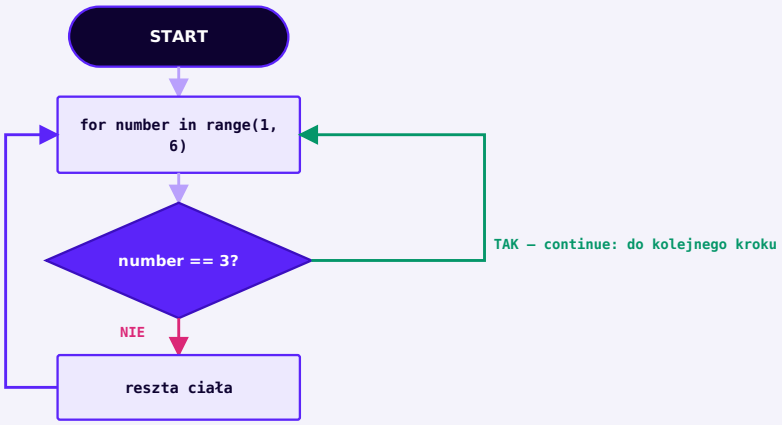
break nie kończy programu
`break` przerywa jedynie pętlę, w której się znajduje — kod po pętli działa normalnie. To nie jest analogia wyjścia z programu, lecz wyjścia z powtarzania.

`continue` to pominąć ten krok

continue.py

```
for number in range(1, 6):
    if number == 3:
        continue  # pomiń resztę ciała w tym kroku
    print(number)
```

Wyjście 1, 2, 4, 5 – liczba 3 jest pomijana, ale cykl trwa dalej, w przeciwieństwie do `break`.



continue pomija resztę CIAŁA i przechodzi do kolejnej iteracji – cykl trwa dalej

continue nie wychodzi z cyklu
`continue` jedynie przerywa bieżący przepływ ciała — cykl sam trwa dalej i od następnego elementu. To jest odwrotne `break`, a nie jego łagodniejsza wersja.

break i continue w pobliżu

	break	continue
Co się dzieje	cykl zatrzymuje się na zawsze	obecna iteracja się przerwie, pętla trwa dalej
Kod po pętli	wykonany natychmiast	kursuje dopiero po zakończeniu pętli
Częsty przykład	znaleźli to, czego szukali — nie ma potrzeby kontynuować	ten element nie pasuje — pomijamy właśnie go

break w while True

Najczęstsze miejsce dla `break` — cykl `while True`, gdzie warunek zatrzymania łatwiej sprawdzić w środku ciała:

while_true_break.py

```
komanda = ""
while True:
    komanda = input(„Wprowadź komendę (stop - wyjście): ”)
    if komanda == "stop":
        break
    print(„Wykonuję:”, komanda)
```

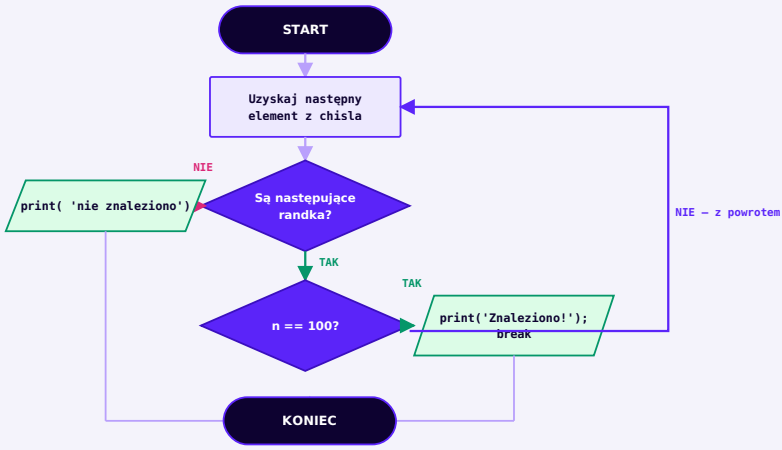
continue przed aktualizacją licznika to częsta pułapka
W pętli `while` do opublikowania `continue` PRZED wierszem, który zmienia stan (na przykład, `count += 1`), niebezpieczne: ten łańcuch znaków będzie pomijany za każdym razem, gdy zostanie wywołany `continue`, a ten cykl może się nigdy nie skończyć. Przyjrzymy się bliżej tutorialowi dotyczącemu pętli debugowania.

loop-else jest else cyklu, a nie if

Cykle `for` oraz `while` rzadko jedzą Używaną, ale czasem bardzo wygodną funkcją jest blok `else`, które są wykonywane, **tylko wtedy, gdy pętla kończyła się bez break**:

loop_else.py

```
chisla = [4, 8, 15, 16, 23, 42]

for n in chisla:
    if n == 100:
        print(„Znalazłem 100!”)
        break
else:
    print(„100 nie znalezionych ani razu”)
```



else dla pętli jest wykonywana tylko wtedy, gdy pętla osiągnęła koniec BEZ break

loop-else NIE jest if/else

Chociaż słowo `else` tak samo jak `if`, sens jest zupełnie inny. `else` u `if` oznacza „jeśli warunek jest fałszywy”. `else` w pętli oznacza „jeśli pętla zakończyła się sama, a nie przez break”. Jeśli wewnątrz pętli nie było `break` w ogóle — blok `else` zawsze zostanie spełnione zaraz po zakończeniu cyklu.

Mini-projekt: cykl zespołów

Będziemy zbierać `while True`, `break` oraz pracuj z łańcuch znaków/ warunkami z rozdziałów 8-9 w małej, interaktywnej pętli. To jest tutorial Demonstracja techniki, a nie zastępstwo dla interpretera PySH z wprowadzenia kursu:

cikl_komand.py

```
zhurnal = []
while True:
    komanda = input(„Drużyna (help/list/
stop):”).strip().lower()
    if komanda == "stop":
        print(„Kończę pracę.”)
        break
    elif komanda == "help":
        print(„Dostępne: help, list, stop”)
    elif komanda == "list":
        print(„Magazyn:”, zhurnal)
    else:
        zhurnal.append(komanda)
        print(f"Dodane do dziennika: {komanda}")
```

Praktyka: break i continue
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/10-04/index.html)

Praktyka: Mini-projekt i loop-else Cykl Teams
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/10-13/index.html)

Wyszukiwanie, filtrowanie i streszczenie

Pętla w połączeniu z if jest już prawdziwym algorytmem: trzema najczęstszymi wzorcami, na których większość programów opiera się na cyklach.

Pętla + if = rzeczywiste algorytmy

Większość rzeczywistych problemów z pętlami to `for` lub `while` w związku z `if`. Omówmy trzy najczęstsze wzorce: wyszukiwanie, filtrowanie, sumowanie.

Szukanie (search)

poisk.py

```
chisla = [4, 8, 15, 16, 23, 42]
iskomoe = 16
najdeno = False

for n in chisla:
    if n == iskomoe:
        najdeno = True
        break

print(najdeno)
```

Liczenie (counting)

podschet.py

```
chisla = [4, 8, 15, 16, 23, 42, 8, 4]
chetnye = 0

for n in chisla:
    if n % 2 == 0:
        chetnye += 1

print(chetnye)
```

Podsumowanie: ręczne i sum()

summa_vruchnuyu.py

```
chisla = [4, 8, 15, 16, 23, 42]
summa = 0

for n in chisla:
    summa += n

print(summa)
```

summa_cherez_sum.py

```
chisla = [4, 8, 15, 16, 23, 42]
summa = sum(chisla)
print(summa)
```

sum() nie jest magią, lecz ten sam cykl w środku
`sum()` — funkcja wbudowana, która robi dokładnie to samo, co ręczna pętla z akumulatorem powyżej, tylko w jednej linijce. Warto najpierw napisać ręcznie (żeby zrozumieć, co się dzieje), a potem wiedzieć, że dla gotowego wyniku istnieje krótsza droga.

Podgląd: Znajdź niskie i wysokie

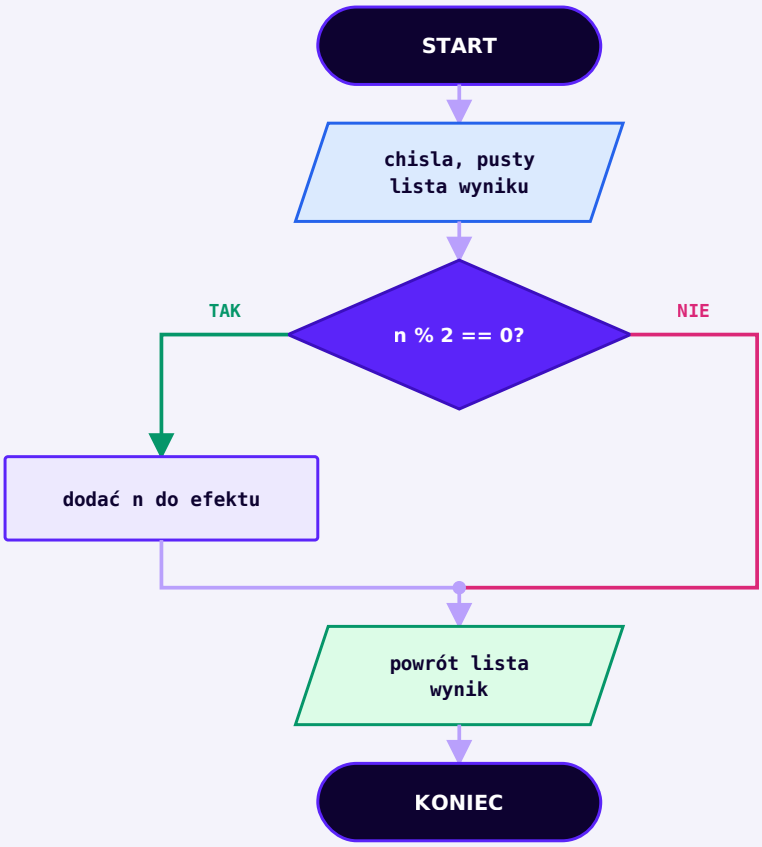
min_max_vruchnuyu.py

```
chisla = [4, 8, 15, 16, 23, 42]
maksimum = chisla[0]

for n in chisla:
    if n > maksimum:
        maksimum = n

print(maksimum)
# to samo, krócej: print(max(chisla))
```

Filtrowanie — zbieramy tylko potrzebne

filtraciya.py

```
chisla = [4, 8, 15, 16, 23, 42]
chetnye_chisla = []

for n in chisla:
    if n % 2 == 0:
        chetnye_chisla.append(n)

print(chetnye_chisla)
```



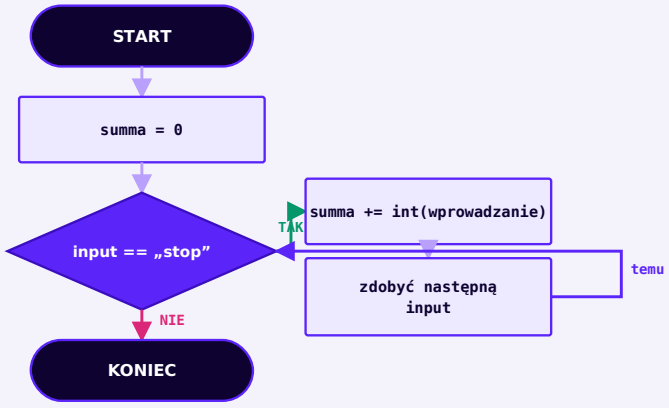
Wzorzec filtrowania: przejdź przez wszystkie elementy, zostawiając tylko te, które spełniły dany warunek

Pętla strażnicza — specjalna wartość jako sygnał zatrzymania

Sentinel jest szczególnym znaczeniem, które samo w sobie nie jest częścią danych, ale sygnalizuje „dane na zewnątrz”

sentinel_cikl.py

```
summa = 0
while True:
    chislo = input(„Wprowadź liczbę (stop - koniec): ”)
    if chislo == "stop":
        break
    summa += int(chislo)

print(„Ilość:”, summa)
```



cykl Sentinel: użytkownik decyduje, kiedy wyczerpują się dane

Opcjonalne: FizzBuzz

Klasyczne, ale nieobowiązkowe ćwiczenie rozgrzewkowe na pętle i warunki — jeśli ciekawi, spróbuj napisać je samodzielnie zanim zobaczysz rozwiązanie:

fizzbuzz.py

```
for n in range(1, 21):
    if n % 15 == 0:
        print(FizzBuzz)
    elif n % 3 == 0:
        print(„Fyzzz”)
    elif n % 5 == 0:
        print(Buzz)
    else:
        print(n)
```

Praktyka: wyszukiwanie, liczenie, stremywanie, filtrowanie
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/10-14/index.html)

Praktyka: pętla sentinelowa
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/10-22/index.html)

Kontrola wejścia pętli

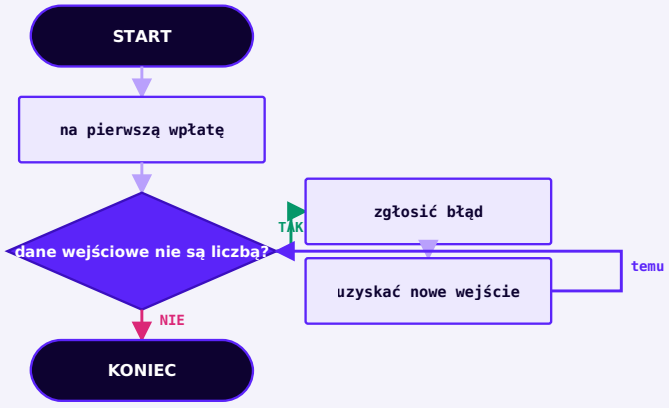
Jednym z najbardziej praktycznych wzorców while jest ponowne zapytanie użytkownika, aż dane będą odpowiednie.

Problem: użytkownik może wpisać cokolwiek

Program w rozdziale 9 oczekiwał od użytkownika liczby — ale co jeśli wpisał to słowo lub Zostawić pole puste? Pętla `while` pozwala ci zapytać ponownie, aż Wkład nie stanie się odpowiedni:

proverka_vvoda.py

```
vozrast = input(„Wejdź w wiek: ”)
while not vozrast.isdigit():
    print(„Musisz wpisać numer!”)
    vozrast = input(„Wejdź w wiek: ”)

vozrast = int(vozrast)
print(„Dziękuję! Za ciebie”, vozrast, „lat”)
```



Pętla powtórkową: Pytaj ponownie, aż dane wejściowe będą odpowiednie

isdigit() jest prosty, ale ograniczony sprawdzenie

`str.isdigit()` sprawdza, że łańcuch znaków składa się tylko z cyfr — to wystarczy dla liczb całkowitych dodatnich, ale nie odróżnia `"-5"` (liczba ujemna) lub `"3.5"` (ułamek) od nieprawidłowego wejścia. Bardziej wiarygodnym sposobem sprawdzenia jest projekt `try/except` — nauczymy się jej w następnych rozdziałach; teraz wystarczy umieć sprawdzać liczby całkowite dodatnie.

Ograniczenie liczby prób

Czasem nie warto pytać bez końca – rozsądne jest ograniczenie liczby prób:

proverka_s_limitom.py

```
popytki = 0
max_popytok = 3

while popytki < max_popytok:
    vozrast = input(„Wejdź w wiek: ”)
    if vozrast.isdigit():
        print(„Przyjęte:”, vozrast)
        break
    print(„To nie brzmi jak liczba.”)
    popytki += 1
else:
    print(„Za dużo złych prób.”)
```

Tu znów przydało się loop-else

Blok `else` u `while` odniesie sukces tylko wtedy, gdy próby zakończą się niepowodzeniem `break` dokładnie to, czego potrzeba, by zgłosić „zbyt wiele nieudanych prób”

Praktyka: Kontrola wejścia pętli

interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/10-15/index.html)

Mini-projekt — gra „Zgadnij liczbę”, wersja 2

Ta sama gra co w rozdziale 9, ale teraz z nieograniczoną liczbą prób, zaprojektowanych od pseudokodu i schematu do gotowego kodu.

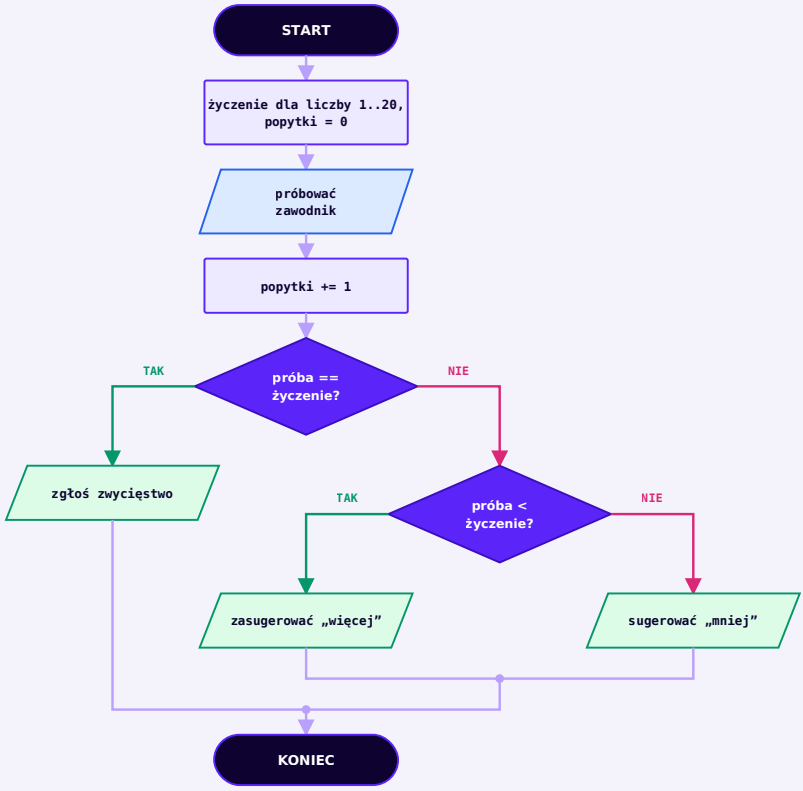
Pamiętasz grę „Zgadnij liczbę” `while` oraz `break`, Dajmy graczowi tyle prób, ile chce. Zaprojektujmy to jak prawdziwy algorytm — najpierw słowa, potem schemat, a potem kod.

Krok 1 — pseudokod

psevdokod.txt

```
загадать случайное число от 1 до 20
счётчик попыток = 0
повторять пока не угадано:
    получить попытку игрока
    увеличить счётчик попыток
    если попытка равна загаданному — сообщить победу и
остановиться
    иначе если попытка меньше — подсказать «больше»
    иначе — подсказать «меньше»
```

Krok 2 — flowchart



Zgadnij liczbę v2 – algorytm jest kompletny, do jednej linii Python

Krok 3 - Tabela Śladów (Przykładowa Gra)

Próba Nie	Wkład gracza	Ukryte	Rezultat
1	10	14	mniej
2	17	14	więcej
3	14	14	trafił!

Krok 4 - Kod

ugadaj_chislo_v2.py

```
import random

zagadannoe = random.randint(1, 20)
popytki = 0

while True:
    popytka = int(input(„Zgadnij liczbę od 1 do 20:”))
    popytki += 1

    if popytka == zagadannoe:
        print(f"Gratulacje, zgadłeś dobrze {popytki} prób!")
        break
    elif popytka < zagadannoe:
        print(„Ukryta liczba jest większa.”)
    else:
        print(„Ukryta liczba jest mniejsza.”)
```

while True jest celowo nieskończoną pętlą

`while True:` nigdy nie zatrzyma się samo z siebie, to jedyna droga wyjścia: `break` do środka. Bardziej naturalne jest sprawdzanie warunku zatrzymania („gracz zgadł”

Zadanie utrwalające: ograniczenie liczby prób i loop-else

★★ Zadanie samodzielne

Limit prób

Przerób grę tak, aby gracz miał nie więcej niż 5 prób – użyj range(5) loop for zamiast while True, a bloku else loop, aby dać poprawną odpowiedź, jeśli gracz nie odpowie poprawnie w 5 próbach.

Praktyka: „Zgadnij liczbę”
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/10-05/index.html)

Praktyka: „Zgadnij liczbę”
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/10-16/index.html)

Debugowanie pętli

Dziesięć typowych błędów pętli i uniwersalna metoda ich znajdowania — tabela śledzenia stworzona ręcznie, łańcuch znaków linia po linii.

Jak czytać czyjś (i własny) zepsuty cykl

Pętle — najczęstsze miejsce, w którym program *dzieła*, ale daje błędny wynik, albo całkowicie się zawiesza. Dobrą wiadomością jest to, że zdecydowana większość takich błędów pasuje do nich Łącznie istnieje tuzin typowych scenariuszy. Przyjrzyjmy się im po kolei.

Ważne: NIE będziemy uruchamiać zawieszonego kodu

Niektóre z poniższych przykładów opisują nieskończone pętle — w praktyce tej strony będziesz **czytać i przewidywać** zachowanie kodu, zamiast uruchamiać prawdziwą pętlę zawieszającą się w przeglądarce. Jeśli kod w twoim programie jest naprawdę zamrożony, zatrzymaj go ręcznie (`Ctrl+C` na terminalu, „Interrupt kernel”

10 Częstych błędów w pętli

1. Zapomniałem zaktualizować status

bug.py

```
while count < 5:
    print(count)
    # zapomniałem count += 1
```

Warunek nigdy nie stanie się fałszywy — nieskończona pętla. Sprawdź, czy w while znajduje się łańcuch znaków, która zmienia zmienną względem warunku.

2. Błąd na jeden na range()

bug.py

```
for i in range(1, 10):
    print(i)  # nie wyświetli 10!
```

range(1, 10) daje 1..9 — jeśli trzeba dodać 10, to range(1, 11). Klasyczny błąd off-by-one.

3. Nieprawidłowe wgniecenie korpusu

bug.py

```
for n in chisla:
print(n)  # SyntaxError: wcięcia oczekiwano
```

Ciało pętli musi być wcięte. Bez tego Python nawet nie uruchomi programu.

4. Licznik jest zerowany wewnątrz pętli

bug.py

```
for n in chisla:
    kolichestvo = 0
    kolichestvo += 1
```

kolichestvo jest resetowany przy każdej iteracji — na końcu zawsze ma 1, a nie całkowitą liczbę. Inicjalizacja powinna nastąpić PRZED pętlą, a nie wewnątrz.

5. break na niewłaściwym poziomie wcięcia

bug.py

```
for n in chisla:
    if n == 5:
    break  # nie w ciele if!
```

break, stojący poza if (z powodu wcięcia), zadziała przy pierwszej iteracji niezależnie od warunku. Sprawdzaj wcięcia tak samo uważnie, jak warunki.

6. continue przed aktualizacją

bug.py

```
while count < 5:
    if count == 2:
        continue
    print(count)
    count += 1
```

Gdy count == 2, continue pomija count += 1 — i count na zawsze pozostanie równe 2. Nieskończona pętla, i jest szczególnie trudno ją zauważyć.

7. Pomyłka zmiennych zagnieżdżonych pętli

bug.py

```
for i in range(3):
    for i in range(4):
        print(i)
```

Wewnętrzna pętla używa tej samej nazwy i, co zewnętrzna — zewnętrzna wartość i zostaje nadpisana i utracona. Użyj różnych nazw, na przykład i i j.

8. Nieprawidłowy krok negatywny

bug.py

```
for n in range(10, 0):
    print(n)  # nic nie wywoła
```

range(10, 0) z domyślnym krokiem +1 nigdy nie przejdzie od 10 do 0 — potrzebny jest range(10, 0, -1).

9. Niezgodność stanu i warunku

bug.py

```
while spisok:
    print(spisok[0])
    # zapomniałem usunąć przedmiot z spisok
```

Warunek sprawdza spisok (o ile lista nie jest pusty), ale ciało nigdy nie zmniejsza lista — nieskończona pętla, choć wygląda inaczej niż licznik.

10. Warunek fałszywy od samego początku

bug.py

```
n = 10
while n < 5:
    print(n)
    n += 1
```

Ciało nie wykona się ani razu — zero iteracji. To nie jest błąd sam w sobie, ale częsta niespodzianka, jeśli wcześniej nie sprawdzono warunku.

Metoda debugowania: Ręczna tabela śledzenia

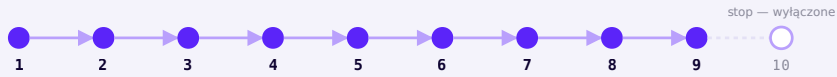
Kiedy pętla działa nie tak, jak się tego spodziewano, najpewniejszy sposób, żeby to sprawdzić, to zbudować tę samą tabelę śledzenia, której używaliśmy przez cały ten rozdział, ale teraz dla prawdziwego, zepsutego kodu:

Iteracja	Stan	Zmienne PRZED ciałem	tego, co jest wydalane	Zmienne PO CIAŁA
1	…	…	…	…
2	…	…	…	…
…	…	…	…	…

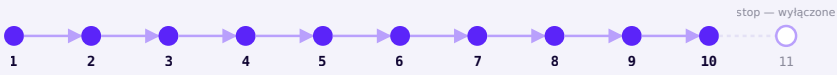
Wypełniaj tabelę linijka po linijce, jakbyś był Python
Nie próbuj zgadywać całego wyniku. Weź pierwszą iterację, zapisz warunek uczciwie, wartości zmiennych przed i po ciele, i powtarzaj, aż zobaczysz, gdzie oczekiwanie odbiega od tego, co faktycznie się dzieje.

Off-by-one — osobna analiza

„Error per one”



range(1, 10) to ostatnia wartość 9, a nie 10! Projekt



range(1, 11) — poprawne, jeśli chcesz osiągnąć 10 włącznie

Lista kontrolna przypadków granicznych

Pytanie	Po co sprawdzać
Co się stanie, jeśli nie ma żadnych powtórek (pusta sekwencja)?	ciało pętli w ogóle się nie wykonuje, niezależnie od tego, czy kod się zepsuje po pętli
Co się stanie przy jednym powtórzeniu?	najczęstszym źródłem błędów na jednostkę
Czy ostatnia granica jest uwzględniona tam, gdzie jest potrzebna?	stop nie jest domyślnie włączony dla range() i slicerów
Czy kierunek kroku pokrywa się z kierunkiem start→stop?	pozytywny krok dla wzrostu, negatywny dla spadku

Sprawa	Przykład	Ile iteracji
Zero	for n in []:	0 — ciało nie zostanie stracone ani razu
Jeden	for n in [7]:	1
Dużo	for n in range(1000):	1000

Zmienna cyklu po zakończeniu

peremennaya_posle_cikla.py

```
for n in range(5):
    pass

print(n)  #4 to ostatnia wartość n którą przyjął
```

Zmienna nie znika po for

Po ukończeniu `for` zmienna pętli pozostaje powiązana z ostatnią uzyskaną wartością — w przeciwieństwie do niektórych innych języków, w Python pętle nie mają osobnego „zakresu widoczności”. Wyjątek: jeśli sekwencja była pusta, ciało nie wykonało się ani razu, a zmienna pętli w ogóle nie została utworzone — odwołanie się do niej po pętli w tym przypadku spowoduje `NameError`.

Nazwy zmiennych pętli

Krótkie nazwy takie jak `i`, `j`, `k` — dawna tradycja dla prostych liczników (szczególnie w zagnieżdżonych pętlach), ale gdy tylko zmienna pętli ma realny sens — nazwij ją sensownie: `for slovo in slova:` jest łatwiejszy do czytania niż `for x in y:`.

Zmienne cieniowanie (shadowing)

Jeśli ręcznie przypiszesz nową wartość zmiennej pętli wewnątrz pętli lub użyjesz nazwy, która już istniała poza pętlą, możesz przypadkowo usunąć pożądaną wartość. Szczególnie łatwo jest popełnić ten błąd w zagnieżdżonych pętlach o tej samej nazwie (patrz błąd nr 7 powyżej).

Podgląd: Zmień listę podczas wyszukiwania

Nie zmieniaj lista, którą teraz próbujesz
Usuwanie lub dodawanie elementów w lista wprost podczas cyklu `for элемент in spisok:` może prowadzić do tego, że niektóre elementy zostaną pominięte lub przetworzone dwukrotnie. Dokładnie omówimy ten temat i bezpieczne sposoby jego obejścia w rozdziale o listach — na razie wystarczy wiedzieć, że tak nie należy robić.

Trochę o występach

Każde dodatkowe zagnieżdżenie pętli mnoży liczbę iteracji (widzieliśmy to już na przykładzie tabliczki mnożenia). Na razie musisz policzyć tysiące, a nie miliony wartości – różnica jest niewidoczna. Formalne wyjaśnienie szybkości algorytmów czeka na Ciebie już teraz Intuicja wystarczy: zagnieżdżona pętla w pętli zwykle działa zauważalnie wolniej niż ta cykl o tej samej wielkości.

Praktyka: Znajdź błęda w pętli
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji
Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/10-17/index.html)

Praktyka: off-by-one — liczyć iteracje
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji
Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/10-18/index.html)

Mini-projekt — automatyzacja rysowania kwadratu

Wróćmy do liczb z rozdziału 6 — tym razem z cyklami zamiast ręcznym powtarzaniem i z rzeczywistymi wynikami.

Mini-projekt — automatyzujemy kwadrat

Zastosujmy pętlę do kwadratu z rozdziału 6. Obie wersje poniższego kodu rysują dokładnie tak samo Kwadrat to jedyna różnica w liczbie wierszy, które zajmował:

Kwadrat: 8 rzędów ręcznie kontra 2 linie z pętlą

Brak Pętli (Rozdział 6)

```
artist.forward(100)
artist.right(90)
artist.forward(100)
artist.right(90)
artist.forward(100)
artist.right(90)
artist.forward(100)
artist.right(90)
```

Z cyklem for

```
for _ in range(4):
    artist.forward(100)
    artist.right(90)
```

Co używać dzisiaj: efekt jest ten sam, ale cztery razy krótszy i z jednym miejscem na zmianę liczby stron.

avto_kvadrat.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=300, height=300)
artist = turtle.Turtle()
artist.speed(0)
artist.pencolor("#5B24F9")
artist.pensize(3)

for _ in range(4):
    artist.forward(100)
    artist.right(90)

screen.exitonclick()
```

REZULTAT

Kwadrat narysowany Turtle

Rzeczywisty efekt jest taki, że oba warianty powyższego kodu rysują dokładnie ten kwadrat

Automatyzujemy dowolną prostą figurę

Uogólnimy kwadrat do wzoru, który rysuje **żadnych** prawidłowy wielokąt — wykorzystując wzór kąta obrotu z rozdziału 6:

$$360° \div \text{liczba stron} = \text{kąt obrotu}$$

Im więcej boków, tym mniejszy kąt obrotu przy każdym kroku

avto_lyubaya_figura.py

```
storony = 8       # Chcę ośmiokąt — po prostu zmień ten numer
dlina = 60
ugol = 360 / storony

for _ in range(storony):
    artist.forward(dlina)
    artist.right(ugol)
```

Jednym zmienna jest dowolna liczba

Zmiana `storony` przez 3, 5, 6, 20 — program automatycznie narysuje trójkąt, pięciokąt, sześciokąt lub prawie koło, bez żadnej zmiany reszty kodu. Poniżej znajduje się rzeczywisty wynik dla kilku wartości `storony`.

Trójkąt (storony = 3)

treugolnik.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=300, height=300)
artist = turtle.Turtle()
artist.speed(0)
artist.pencolor("#5B24F9")
artist.pensize(3)

storony = 3
dlina = 110
ugol = 360 / storony

for _ in range(storony):
    artist.forward(dlina)
    artist.right(ugol)

screen.exitonclick()
```

REZULTAT

Trójkąt wylosowany Turtle

storony = 3, dlina = 110 → trójkąt regularny

Pentagon (storony = 5)

pyatiugolnik.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=300, height=300)
artist = turtle.Turtle()
artist.speed(0)
artist.pencolor("#5B24F9")
artist.pensize(3)

storony = 5
dlina = 75
ugol = 360 / storony

for _ in range(storony):
    artist.forward(dlina)
    artist.right(ugol)

screen.exitonclick()
```

REZULTAT

Pentagon narysowany Turtle

storony = 5, dlina = 75 → regularnym Piątokątie

Heksagon (storony = 6)

shestiugolnik.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=300, height=300)
artist = turtle.Turtle()
artist.speed(0)
artist.pencolor("#5B24F9")
artist.pensize(3)

storony = 6
dlina = 65
ugol = 360 / storony

for _ in range(storony):
    artist.forward(dlina)
    artist.right(ugol)

screen.exitonclick()
```

REZULTAT

Hexagon narysował Turtle

storony = 6, dlina = 65 → regularny sześciokąt

Ośmiokąt (storony = 8)

vosmiugolnik.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=300, height=300)
artist = turtle.Turtle()
artist.speed(0)
artist.pencolor("#5B24F9")
artist.pensize(3)

storony = 8
dlina = 55
ugol = 360 / storony

for _ in range(storony):
    artist.forward(dlina)
    artist.right(ugol)

screen.exitonclick()
```

REZULTAT

Oktagon wylosowany Turtle

storony = 8, dlina = 55 → prawidłowy ośmiokąt

Mini-projekt: studio automatycznych wielokątów

★★ Zadanie samodzielne

Studio wielokątów

Zaprojektuj powyższy program tak, aby użytkownik storony wywoływał input() (nie zapomnij int()). testować na kilku wartościach — w tym dużych, jak 30 czy 50 — i zobaczyć, jak kształt się zmienia.

Praktyka: Automatyczne Studio Wielokątne
Moduł turtle otwiera natywne okno Python – uruchamiane lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/10-06/index.html)

Mini-projekt — automatycznie rysujemy mandalę

Ta sama mandala co w rozdziale 6 — teraz etapami, krócej i czytelniej dzięki range() z trzema argumentami.

W rozdziale 6 mandala rysowana była cyklem `while` z manualem Zwiększanie kąta. Teraz malujmy to etapami — od jednego płatka do pełnego wzoru — Spójrzmy na rzeczywisty efekt każdego etapu.

Etap 1 — jeden motyw

`circle(60)` bez parametru extent rysuje pełne koło i Wraca żółwia dokładnie do punktu startowego i z oryginalnym kierunkiem:

mandala_stadiya_1.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=300, height=300)
artist = turtle.Turtle()
artist.speed(0)
artist.pencolor("#5B24F9")
artist.pensize(2)

artist.circle(60)

screen.exitonclick()
```

REZULTAT

Mandala Jednego Kręgu

Jednym z motywów jest po prostu circle(60)

Etap 2 – Cztery motywy

mandala_stadiya_2.py

```
for _ in range(4):
    artist.circle(60)
    artist.left(90)
```

mandala_stadiya_2.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=300, height=300)
artist = turtle.Turtle()
artist.speed(0)
artist.pencolor("#5B24F9")
artist.pensize(2)

for _ in range(4):
    artist.circle(60)
    artist.left(90)

screen.exitonclick()
```

REZULTAT

Cztery Kręgi Mandali

4 motywy obracały się względem siebie o 90°

Etap 3 – Dwanaście motywów

mandala_stadiya_3.py

```
for _ in range(12):
    artist.circle(60)
    artist.left(30)
```

mandala_stadiya_3.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=300, height=300)
artist = turtle.Turtle()
artist.speed(0)
artist.pencolor("#5B24F9")
artist.pensize(1)

for _ in range(12):
    artist.circle(60)
    artist.left(30)

screen.exitonclick()
```

REZULTAT

Dwanaście kręgów mandali

12 motywów obróconych o 30° to już prawdziwy wzór

Etap 4 – Pełna Mandala

mandala_polnaya.py

```
cvieta = ['#5B24F9', '#DB2777', '#059669']
for i in range(24):
    artist.pencolor(cvieta[i % 3])
    artist.circle(80)
    artist.left(15)
```

mandala_polnaya.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=340, height=340)
artist = turtle.Turtle()
artist.speed(0)
artist.pensize(1)

cvieta = ["#5B24F9", "#DB2777", "#059669"]
for i in range(24):
    artist.pencolor(cvieta[i % 3])
    artist.circle(80)
    artist.left(15)

screen.exitonclick()
```

REZULTAT

Pełna wielokolorowa mandala

24 motywy naprzemiennie z trzema kolorami w i% 3

range() z trzema argumentami — to samo, co ręczny while

`for ugol in range(0, 360, shag_ugla):` generuje dokładnie te same liczby, które otrzymywaliśmy ręcznie w rozdziale 6: `while ugol < 360: ... ugol += shag_ugla`. Pętla `for` po prostu skraca i nie ryzykuje, że zapomnisz zwiększyć licznik.

cvieta[i% 3] to pozostała część dzielenia jako indeks „pętlowany”

`i % 3` daje 0, 1, 2, 0, 1, 2, ... — bez względu na to, jak duże są `i`, wynik reszty zawsze mieści się w zakresie indeksów listy składającej się z 3 kolorów. Ten zabieg — cykliczne przeglądanie skończonego zestawu wartości — występuje bardzo często.

★★ Zadanie samodzielne
Własna mandala
Zmień liczbę motywów (24 → 36), promień (80 → 50) oraz lista kolory, aby stworzyć własny wzór.

Praktyka: Mandala przez for + range(), etapami
Moduł turtle otwiera natywne okno Python – uruchamiane lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/10-07/index.html)

Losowe wzorce Turtle

Każda iteracja pętli może wykonać własny, wcześniej nieznany ruch — a seed() sprawia, że losowy wynik jest powtarzalny.

Cykle i losowość

Moduł `random` z rozdziału 9 dobrze pasuje do cykli — Każda iteracja może podjąć własną, przedniezнaną decyzję. Dzięki temu wynik może być miał na celu reprodukcję i testowanie, poniższe przykłady używają `random.seed(...)` — tworzy „losową”

Przypadkowe wędrowanie

sluchajnoe_bluzhdanie.py

```
import random
random.seed(7)

for _ in range(100):
    artist.setheading(random.randint(0, 360))
    artist.forward(10)
```

sluchajnoe_bluzhdanie.py

```
import turtle
import random

random.seed(7)
screen = turtle.Screen()
screen.setup(width=420, height=420)
artist = turtle.Turtle()
artist.speed(0)
artist.pencolor("#5B24F9")
artist.pensize(2)

for _ in range(100):
    artist.setheading(random.randint(0, 360))
    artist.forward(10)

screen.exitonclick()
```

REZULTAT

Losowa trajektoria chodzenia

100 kroków w losowym kierunku, seed(7) – jeśli pobiegniesz ponownie, otrzymasz tę samą ścieżkę

seed() nie chodzi o losowość figury, lecz o powtarzalność

Bez `random.seed(...)` wynik byłby inny za każdym razem – co jest ciekawe dla kreatywności, ale niewygodne dla praktyki, gdzie ważne jest, by sprawdzić konkretny, powtarzalny wynik.

Pole Gwiezdne

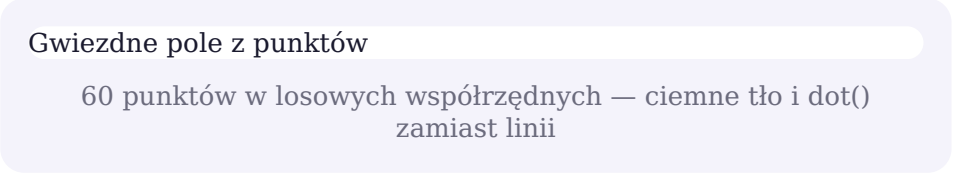
zvezdnoe_pole.py

```
import random
random.seed(3)

for _ in range(60):
    x = random.randint(-190, 190)
    y = random.randint(-190, 190)
    artist.goto(x, y)
    artist.dot(6, "white")
```

zvezdnoe_pole.py

```
import turtle
import random

random.seed(3)
screen = turtle.Screen()
screen.setup(width=420, height=420)
screen.bgcolor("#0D0230")
artist = turtle.Turtle()
artist.speed(0)
artist.hideturtle()
artist.penup()

for _ in range(60):
    x = random.randint(-190, 190)
    y = random.randint(-190, 190)
    artist.goto(x, y)
    artist.dot(6, "white")

screen.exitonclick()
```

REZULTAT



★★ Zadanie samodzielne

Twoje przypadkowe wędrówki

Zmień seed na inną liczbę i liczbę kroków – porównaj, jak różni się ścieżka.

Praktyka: Przypadkowy spacer i Pole Gwiezdne

Moduł turtle otwiera natywne okno Python – uruchamiane lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/10-19/index.html)

Siatka kształtów

Ten sam wzór wierszy × kolumn co w tablicy mnożenia, tylko że teraz rysuje zamiast drukować liczby.

Zagnieżdżone pętle jak prawdziwa grafika

Widzieliśmy już zagnieżdżone pętle na liczbach (tabliczki mnożenia) i na tekście (litery wewnątrz słowa). Ten sam wzorzec „wiersze × kolumny”

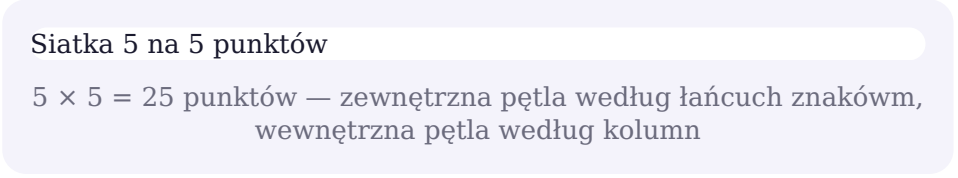
setka_figur.py

```
shag = 60
for row in range(5):
    for col in range(5):
        x = -140 + col * shag
        y = -140 + row * shag
        artist.goto(x, y)
        artist.dot(16, '#5B24F9')
```

setka_figur.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=380, height=380)
artist = turtle.Turtle()
artist.speed(0)
artist.hideturtle()
artist.penup()

shag = 60
for row in range(5):
    for col in range(5):
        x = -140 + col * shag
        y = -140 + row * shag
        artist.goto(x, y)
        artist.dot(16, "#5B24F9")

screen.exitonclick()
```

REZULTAT



Ten sam wzór co w tablicy mnożenia

row i col ponownie przebiegają cały zakres niezależnie od siebie — łączna liczba narysowanych punktów wynosi `5 * 5 = 25` , tak jak pomyśleliśmy o wzorze „wiersze × kolumny”

★★ Zadanie samodzielne

Siatka figur zamiast punktów

Zamień artist.dot(...) na mały kwadrat (kolejny, trzeci, zagnieżdżony pętla na 4 bokach) – otrzymasz siatkę małych kwadratów zamiast kropek.

Praktyka: siatka figur w zagnieżdżonych pętlach
Moduł turtle otwiera natywne okno Python – uruchamiane lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/10-20/index.html)

Mini-projekt – spirale łuków

Kończymy rozdział wzorcem, który zmienia się na każdym etapie cyklu, i podsumowujemy wyniki.

Mini-Projekt Ostatniego Rozdziału: Spirala Łuku – każdy wątek jest mniejszy poprzednie.

spirali_iz_dug.py

```
radius = 5

for _ in range(60):
    artist.circle(radius, 90)   # Łuk ćwierćkoła
    radius += 3                 # Za każdym razem trochę więcej
```

spirali_iz_dug.py

```python
import turtle

screen = turtle.Screen()
screen.setup(width=420, height=420)
artist = turtle.Turtle()
artist.speed(0)
artist.pencolor("#5B24F9")
artist.pensize(2)

radius = 5
for _ in range(60):
    artist.circle(radius, 90)
    radius += 3

screen.exitonclick()
```

REZULTAT

Spirala z rosnących łuków

60 łuków, promień rośnie z 5 o 3 przy każdym kroku — rozwijająca się spirala

Zmiana zmiennej w pętli — normalna sprawa

W przeciwieństwie do poprzednich przykładów, tutaj `radius` się zmienia *każdym kroku* cyklu powoduje to efekt rosnącej spirali, a nie powtarzającego się wzoru.

break w Turtle jest zatrzymanie się przed końcem

`break` świetnie działa w pętlach wewnętrznych, które rysują grafikę — na przykład, aby pozostać na ekranie:

break_v_turtle.py

```
dlina = 10
for _ in range(50):
    if artist.xcor() > 150:
        break
    artist.forward(dlina)
    artist.left(90)
    dlina += 6
```

break_v_turtle.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=420, height=420)
artist = turtle.Turtle()
artist.speed(0)
artist.pencolor("#5B24F9")
artist.pensize(3)

dlina = 10
for _ in range(50):
    if artist.xcor() > 150:
        break
    artist.forward(dlina)
    artist.left(90)
    dlina += 6

screen.exitonclick()
```

REZULTAT

Kwadratowa spirala, przycięta break

Kwadratowa spirala odrywa się w środku ostatniego zwrotu – gdy xcor() przekroczy 150, break zatrzymuje cykl

50 zaplanowano, ale wykonało się mniej

Pętla była gotowa do wykonania 50 razy, ale faktycznie ukończona wcześniej, gdy tylko spełniono warunki `artist.xcor() > 150` się sprawdziło. Dlatego ostatni segment na zdjęciu jest krótszy niż powinien był być pełny obrót — break przerywa rysunek dokładnie w środku kroku.

★★ Zadanie samodzielne

Spirala z kwadratów

Zastąp łuk małym kwadratem (pętlą czterostronną) wewnątrz zewnętrznej pętli, aby stworzyć spiralę z zmniejszającymi się lub rosnącymi kwadratami.

Praktyka: Spirale łuków

Moduł turtle otwiera natywne okno Python – uruchamiane lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/10-08/index.html)

Praktyka: break w Turtle jest kwadratową spiralą

Moduł turtle otwiera natywne okno Python – uruchamiane lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/10-21/index.html)

Podsumowanie rozdziału

Narzędzia pętli — na przyszłość

Rozdział 10 Podsumowanie Ściągawka

Liczba powtórzeń jest znana z góry	→ `for ... in range(n)`

Brutalna siła gotowej sekwencji (łańcuch znaków, lista)	→ `for element in sekwencję`
Potrzebujesz zarówno indeksu, jak i wartości	→ `for i, znaczenie in enumerate(...)`
Liczba powtórzeń nie jest znana z góry	→ `while stan`
Warunek zatrzymania wygodniej sprawdzić w środku ciała	→ `while True + break`
Musimy się dowiedzieć, czy był jakiś break	→ `for/while … else`
Musisz pominąć jeden krok, nie wychodząc z pętli	→ `continue`

Nie sztywne zasady – wytyczne, które stają się nawykiem z praktyką

Czego nauczyliśmy się w tym rozdziale

- Powtarzanie to trzecia struktura algorytmu po sekwencji i rozgałęzieniu (Rozdział 9).
- `for ... in range(n)` powtarza blok kodu określoną liczbę razy — pętla pyta "Czy jest następny przedmiot?", zamiast N wykonać ciało "magicznie".
- `range()` nie lista, lecz generator liczb; jego stop się nie włącza — ta sama logika co w liniach z rozdziału 8.
- `enumerate()` daje jednocześnie zarówno indeks, jak i wartość; licznik i akumulator to dwa podstawowe wzorce stanów w pętli.
- `while` powtarza akcję, dopóki warunek pozostaje prawdziwy — używane, gdy liczba powtórzeń jest nieznana.
- `break` przerywa pętlę całkowicie; `continue` pomija bieżący krok; opcjonalne `else` pętla działa tylko bez break.
- Cykle można zagnieżdżać w sobie — wtedy wewnętrzny cykl wykonuje się całkowicie przy każdym kroku zewnętrznego, a łączna liczba iteracji mnoży się.
- Większość błędów pętli mieści się w kilkunastu typowych scenariuszach — tabela śledzenia stworzona ręcznie pomaga znaleźć każdy z nich.
- Wszystkie ręcznie rysowane kształty z rozdziałów 6-7 można teraz zapisać w 2-4 linijkach w cyklu, a my widzieliśmy ich prawdziwe, naprawdę ukończone rezultaty.

co dalej

Widzieliśmy już pętle działające z listami liczb — `[4, 8, 15, 16, 23, 42]`. W następnym rozdziale szczegółowo przeanalizujemy, jak same zbiory danych są zorganizowane Listy, krotki, zbiory i słowniki to właśnie dla cykli istnieją w prawdziwych programach.

ROZDZIAŁ 11 · STR. 569

Dużo informacji!

Listy, krotki, zbiory i słowniki to cztery główne sposoby przechowywania dużej ilości danych jednocześnie. Nie lista metody, lecz analiza, dlaczego każda struktura jest potrzebna, jak są uporządkowane wewnętrznie (referencje, zmienność, kopiowanie) oraz jak wybierać między nimi.

Dlaczego przechowywać wiele wartości

Zanim lista metod — pytanie „po co”: jedno zmienna przechowuje jedną wartość, a kolekcja przechowuje ich wiele pod jedną nazwą.

Jeden zmienna to jedna wartość. A co jeśli jest pięćset?

Do tej pory każdy zmienna przechowywał jedną wartość. Wyobraź sobie, że zapisujesz wyniki Pięciu uczniów:

```
pyat_peremennyh.py
```

```
score_1 = 95
score_2 = 82
score_3 = 91
score_4 = 77
score_5 = 88
```

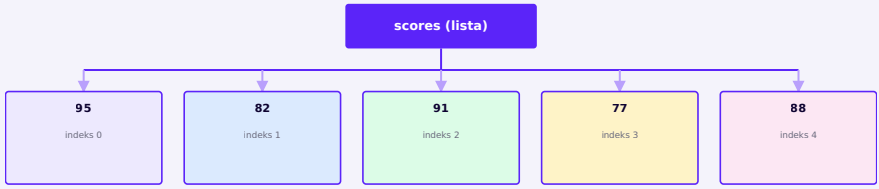
Działa. Teraz wyobraź sobie, że nie jest pięciu uczniów, ale **pięćset**. Zakładać `score_1` … `score_500` ręcznie nie jest łatwe przez długi czas, ale w rzeczywistości nie da się tego utrzymać: nie można ich przechodzić w cyklu, nie można obliczać Średnia jednolinijkowa, nie możesz dodać oceny ucznia 501st bez edycji kodu.

Kolekcja: Jedno imię, wiele znaczeń

Za to Python **kolekcje** są typy danych, które są przechowywane natychmiast Wiele znaczeń pod jednym nazwiskiem:

```
odin_spisok.py
```

```
scores = [95, 82, 91, 77, 88]
print(scores)
```



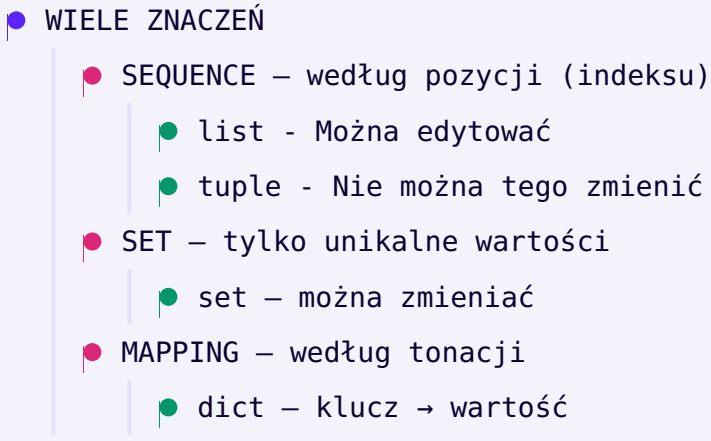
Nazwa scores wskazuje na JEDEN obiekt kolekcji, który przechowuje odwołania do pięciu wartości

Kolekcja organizuje odniesienia do obiektów, a nie „wkłada przedmioty do pudełka”

W rozdziale 3 już widzieliśmy, że zmienna to nazwa wskazująca na obiekt. Kolekcja jest zorganizowana podobnie: `scores` wskazuje na pojedynczy obiekt lista i lista przechowuje odniesienia do kilku oddzielnych wartości. Nie potrzebujemy teraz szczegółów dotyczących wewnętrznej struktury CPython — ten obraz wystarczy.

Python Mapa kolekcji

W Python cztery główne wbudowane kolekcje. Dzielą się na trzy rodziny według tego, **jak** dostęp do wartości jest organizowany:



Trzy rodziny kolekcji Python: sekwencję (według pozycji), zbiór (jednoznaczność), wyświetlanie (według klucza)

Cztery wbudowane kolekcje w skrócie

LIST

opływowy
zmieniamy

powtórki są dozwolone

`[1, 2, 2]`

TUPLE

opływowy
niezmienne
powtórki są dozwolone

`(1, 2, 2)`

SET

bez indeksów
zmieniamy
tylko unikalne

`{1, 2}`

DICT

według klucza
zmieniamy
klawisze są unikalne

`{"a": 1}`

Jak wybrać? Na razie — wskazówka, nie prawo

Pełna analiza wyboru struktury zostanie przedstawiona w §11.22, ale już przydatne jest posiadanie ogólnych wytycznych:

Którą kolekcję wybrać? (pierwszy punkt orientacyjny)

Czy potrzebujesz pary klucz→wartość?	→ **dict**
Potrzebne są tylko unikalne wartości, kolejność nie jest ważna?	→ **set**
Potrzebny jest porządek i trzeba będzie często zmieniać zawartość?	→ **list**
Potrzebny porządek, ale wartości ustalone raz na zawsze?	→ **tuple**

To zasada ogólna, a nie prawo matematyczne — będą od niej wyjątki

Praktyka: zamieniamy pięć zmiennych na jeden lista
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/11-11/index.html)

Przechowujemy więcej niż jedną wartość

Listy — najbardziej uniwersalna kolekcja Python: uporządkowany zestaw wartości w jednej zmiennej.

Listy

Lista (`list`) to uporządkowana kolekcja wartości w nawiasach kwadratowych, oddzielone przecinkami:

spiski.py

```
fruits = [„jabłko”, „banan”, „wiśnia”]
print(fruits)
print(type(fruits))
```

Lista może przechowywać wartości różnych typów jednocześnie, choć w praktyce częściej przechowuje wartości jednego rodzaju — tak kod jest łatwiejszy do czytania:

smeshannyj_spisok.py

```
smeshannyj = ["Cartesian", 5, 3.14, True]
print(smeshannyj)
```

Lista jakiegokolwiek rodzaju to nie błąd, lecz wybór
Python nie zabrania mieszania typów w tej samej liście. Ale jeśli elementy listy oznaczają różne rzeczy (nazwa, numer, flaga), prawie zawsze łatwiej jest użyć słownik (§11.17) — lista wygodniej, gdy wszystkie elementy mają to samo znaczenie: nazwy, oceny, współrzędne.

Długość listy: len()

Podobnie jak wiersze (rozdział 8), lista ma długość:

dlina_spiska.py

```
numbers = [10, 20, 30]
print(len(numbers))   # 3
```

len(numbers) = 3 → dopuszczalne indeksy: od 0 do len-1 = 2

Dostęp do wartości listy według indeksu

Tak jak w przypadku łańcuch znaków, elementy listy mają indeksy, zaczynając od zera — z tym samym regułą ujemnych indeksów „od końca”:

names = ["Anna", "Oleg", "Maria", "Leo"]

dostup_k_spisku.py

```
names = ["Anna", "Oleg", "Maria", "Leo"]
print(names[0])    # Anna
print(names[-1])   # Leo — ostatni element
```

Praktyka: Tworzenie list i dostęp do indeksu
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/11-01/index.html)

Robię kawałek listy!

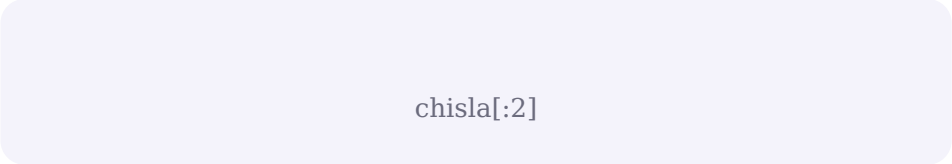
List slicery działają dokładnie tak samo jak sekacze wierszy — znana logika w nowym kontekście.

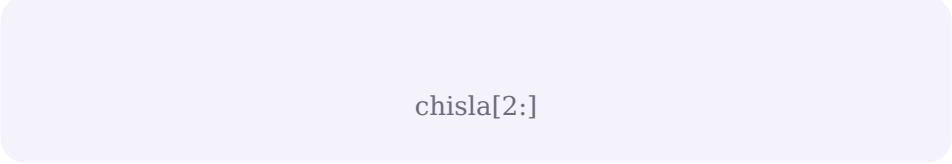
List slicery działają dokładnie tak samo jak linie liniowe z rozdziału 8: `start` wchodzi w wycinek, `stop` — nie.

chisla[1:3] → [20, 30] - Granica 1 jest włączona, granica 3 nie

srezy_spiskov.py

```
chisla = [10, 20, 30, 40, 50]
print(chisla[1:3])    # [20, 30]
print(chisla[:2])     # [10, 20]
print(chisla[2:])     # [30, 40, 50]
print(chisla[::-1])   # [50, 40, 30, 20, 10] — rozszerzony
lista
```





List slicer to nowy lista

Podobnie jak slicer, list slicer zwraca **nowe** lista bez zmiany oryginalnej. Warto o tym pamiętać — przyda się to w §11.10 „Listy kopiowania”

Praktyka: wycinki list

interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/11-02/index.html)

Zmiana lista

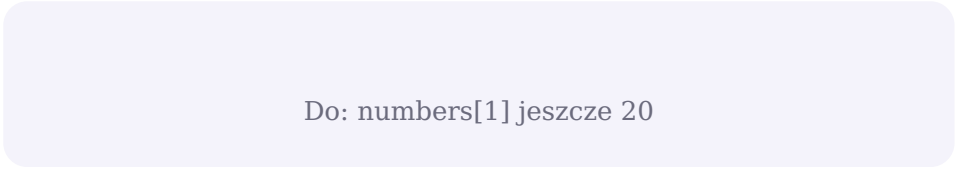
Pierwszy prawdziwie zmienny typ danych: element listy można zastąpić bez tworzenia nowego obiektu.

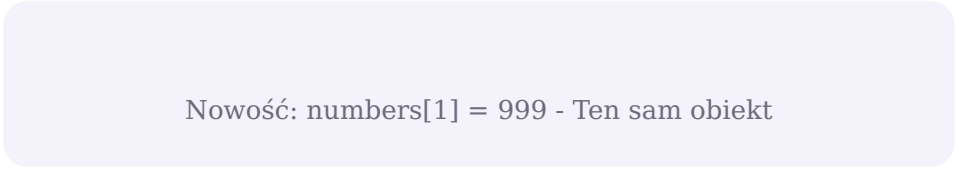
Listę można zmienić „na miejscu”

To fundamentalna różnica między listą a łańcuch znaków. Przypiszemy nową wartość elementowi według indeksu:

izmenenie_elementa.py

```
numbers = [10, 20, 30]
numbers[1] = 999
print(numbers)   # [10, 999, 30]
```





To ten sam obiekt, nie nowy lista
Po `numbers[1] = 999` lista nie jest tworzony na nowo — zmienia się zawartość tego samego obiektu. To stanie się ważne w §11.9, kiedy jedna lista będzie miała od razu dwie zmienne-referencje.

Mutable vs immutable

Widzieliśmy już niezmienne typy — liczby i łańcuch znaków. Lista jest pierwszą rzeczywistą **modyfikowalne (mutable)** typ, który badamy:

Typ	Czy można zastąpić część znaczenia „na miejscu”
`str`	Żadnych — `name[0] = "P"` powoduje TypeError, potrzebna jest nowa łańcuch znaków
`tuple`	Nie – krotki są również niezmienne (§11.13)
`list`	Tak — `numbers[1] = 999` zmienia ten sam obiekt

Zmienność jest właściwością typu, a nie konkretną wartością
Nie ma "mutowalnych łańcuch znaków" ani "list niezmiennych" w zależności od zawartości – niezmienność jest całkowicie określona **typem** obiektu: wszystko `list` mutowalne, wszystkie `tuple` oraz `str` — nie.

Praktyka: Zmień pozycję listy na miejscu
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji
`Otwórz praktykę →` (https://www.cartesianschool.org/pl/practice/11-12/index.html)

append, extend, insert

Trzy sposoby, by dodać coś do lista – i jedna z najczęstszych pułapek początkujących: append() i extend() robią zupełnie różne rzeczy.

append() - Dodaj jeden element

Wcześniej: numbers.append(3)

Nowość: [1, 2, 3]

append.py

```
numbers = [1, 2]
numbers.append(3)
print(numbers)   # [1, 2, 3]
```

`append()` zawsze dodaje 16 **dokładnie jedną** nowy element — nawet jeśli ten element sam jest listą.

append() vs extend() jest ważną pułapką

append_vs_extend.py

```
a = [1, 2]
a.append([3, 4])
print(a)   # [1, 2, [3, 4]] — lista na liście!

b = [1, 2]
b.extend([3, 4])
print(b)   # [1, 2, 3, 4]
```

append([3, 4]) → jeden NOWY element-lista

extend([3, 4]) → dwa punkty Z listy

append() dodaje jeden obiekt, extend() — elementy z iterowalnego

`append(x)` kładzie `x` do środka jako JEDEN nowy element, jaki by nie był. `extend(iterable)` rozwija przekazany lista (lub każdy inny obiekt iterowalny) i dodaje każdy jego element osobno. Pomylenie ich to klasyczny błąd nowicjusza.

insert() - Wklej według indeksu

insert.py

```
fruits = [„jabłko”, „banan”, „wiśnia”]
fruits.insert(1, „mango”)
print(fruits)   # ["jabłko", "mango", "banan", "wiśni"]
```

Wcześniej: insert(1, 'mango')

Nowość: Pozostałe elementy przesunęły się w prawo

Elementy zaczynające się od określonego indeksu zdają się „oddalać”

Praktyka: append, extend, insert
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/11-13/index.html)

remove, pop, clear, del

Cztery sposoby usunięcia czegoś z listy – każdy ma swoją rolę i nie warto ich mylić.

remove() — usuń według wartości

remove.py

```
names = ["Anna", "Oleg", "Maria"]
names.remove("Oleg")
print(names)   # ["Anna", "Maria"]
```

pop() — usuń według pozycji i zwróć wartość

pop.py

```
names = ["Anna", "Oleg", "Maria"]
removed = names.pop(1)
print(names, removed)   # ["Anna", "Maria"] Oleg
```

	remove(value)	pop(index)
Usunięte przez	znaczenie	pozycje (według indeksu)
Co wraca	nic (`None`)	wartość zdalna
Jeśli argument nie pasuje	`ValueError` , jeśli wartości nie ma	`IndexError` jeśli nie ma indeksu

pop() bez dyskusji – usuwa ostatni element
`names.pop()` bez indeksu usuwa i zwraca **ostatni** punkt z listy. To przydatne, jeśli używasz lista jako stosu — więcej o stosach w kolejnych rozdziałach.

clear() i del to dwa różne sposoby „zniszczenia”

clear_del.py

```
items = [1, 2, 3]
items.clear()
print(items)      # [] - ten sam lista, ale pusty

other = [1, 2, 3]
del other[1]
print(other)       # [1, 3] - Usunięto jeden element na indeks

third = [1, 2, 3]
del third           # usunięto samą zmienną
```

Komenda	Co się dzieje
`items.clear()`	ten sam obiekt-lista pozostaje, ale staje się pusty
`del items[i]`	usuwa element z indeksem i (lista sam pozostaje)
`del items`	usuwa samą zmienną nazwy (Rozdział 3) — po tym `items` nie istnieje

Po del items nazwa items już nie istnieje

`print(items)` po `del items` spowoduje `NameError` to samo zachowanie `del`, które widzieliśmy w rozdziale 3 dla zmiennych wspólnych, dopiero teraz zastosowano do listy.

Praktyka: remove, pop, clear, del

interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/11-14/index.html)

Potężne operacje listy!

Listy są modyfikowalne – mają cały zestaw metod dodawania, usuwania, wyszukiwania i sortowania.

Listy (w przeciwieństwie do łańcuch znaków) mają metody, które **zmienić sam lista** — listy, w przeciwieństwie do łańcuch znaków, są modyfikowalne (§11.4).

Kopiowanie i dodawanie

kopirovanie.py

```
original = [1, 2, 3]
kopiya = original.copy()
kopiya.append(4)
print(original)  # [1, 2, 3] — nie zmieniło się
print(kopiya)    # [1, 2, 3, 4]
```

kopiya = original — to nie jest kopia!
`kopiya = original` tworzy drugą zmienną, która określa **to samo** lista — zmiana przez jedną zmienną jest również widoczna przez inną. Nazywa się to **aliasing** zostanie szczegółowo omówione w §11.9. Aby uzyskać prawdziwą, niezależną kopię, musisz `.copy()` (lub wycinek `original[:]`) — szczegóły i ważny niuans z listami zagnieżdżonymi w §11.10.

Liczenie i sprzątanie

podschet.py

```
chisla = [1, 2, 2, 3, 2]
print(chisla.count(2))   #3 — Ile razy pojawia się 2
chisla.clear()
print(chisla)             # [] - lista pusty
```

Konkatenacja

konkatenaciya_spiskov.py

```
a = [1, 2]
b = [3, 4]
print(a + b)   # [1, 2, 3, 4]
```

Wyszukiwanie i kontrola dostępności: in, index(), count()

poisk.py

```
fruits = [„jabłko”, „banan”, „wiśnia”]
print(„banan” in fruits)          # True
print(fruits.index(„wiśnia”))      # 2 — indeks elementu
```

Weryfikacja `in` jest tym samym wyrażeniem logicznym co w rozdziale 9, tylko że Teraz po lewej stronie nie jest liczba, lecz zbiór:

membership_primer.py

```
allowed_users = ["anna", "oleg", "maria"]
username = "oleg"
if username in allowed_users:
    print(„Dostęp dozwolony”)
```

index() powoduje ValueError, jeśli nie ma wartości
Bezpieczny porządek – najpierw sprawdź `in`, a potem poszukaj indeksu: `if value in items: position = items.index(value)`.

Dodawanie i usuwanie przedmiotów

Krótko – szczegółowa analiza z diagramami wizualnymi w §11.5 i §11.6:

dobavlenie_udalenie.py

```
fruits = [„jabłko”, „banan”]
fruits.append(„wiśnia”)        # dodaj do końca
fruits.insert(0, „mango”)     # wklej według indeksu
print(fruits)

fruits.remove(„banan”)        # usuń według wartości
last = fruits.pop()            # usuń i cofnij ostatni element
print(fruits, last)
```

sort() mutuje i wraca None to klasyczna pułapka

razvorot_sortirovka.py

```
chisla = [3, 1, 4, 1, 5]
chisla.sort()
print(chisla)      # [1, 1, 3, 4, 5]
chisla.reverse()
print(chisla)      # [5, 4, 3, 1, 1]
```

sort_none_lovushka.py

```
chisla = [3, 1, 2]
chisla = chisla.sort()
print(chisla)   # None — a nie posortowany lista!
```

Nigdy nie pisz x = x.sort()
`.sort()` sortuje lista **na miejscu** i powroty `None` — to popularny sposób Python pokazać „zmieniłem obiekt, a nie utworzyłem nowy” (ta sama zasada, co w przypadku `.append()`). Przypisanie wyniku z powrotem do zmiennej usuwa lista, zastępując go słowem `None`.

sorted() to samo zadanie, ale bez mutacji

	list.sort()	sorted(list)
Czy zmienia to oryginalny lista?	Tak – mutuje	Nie, to cię nie dotyka
Co zwraca?	`None`	nowy posortowany lista

sorted_primer.py

```
chisla = [3, 1, 4, 1, 5]
novyj = sorted(chisla)
print(chisla)   # [3, 1, 4, 1, 5] - Bez zmian
print(novyj)     # [1, 1, 3, 4, 5]
```

`sorted()` mogą i `key` — przez który Własność porównywania elementów:

sorted_key.py

```
names = ["Bob", "Alexandra", "Li"]
print(sorted(names, key=len))   # ['Li', 'Bob', 'Alexandra']
```

Praktyka: operacje na listach
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji
Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/11-03/index.html)

Listy i pętle

Rozdział 10 w końcu się opłaca: przechodzimy przez cykl lista, zamiast ręcznie pracować z indeksami.

Iteracja listy pętlą for

Rozdział 10 w końcu się opłaca — możesz iterować przez lista pętlę, przechodząc od razu do wartości, Bez ręcznej pracy z indeksami:

perebor_spiska.py

```
fruits = [„jabłko”, „banan”, „wiśnia”]
for fruit in fruits:
    print(fruit)
```

for w pętli samodzielnie wyciąga kolejny element — nie ma potrzeby ręcznego odczytywania indeksów

enumerate() jest sytuacja, gdy potrzebujesz zarówno indeksu, jak i wartości

enumerate_spiski.py

```
names = ["Anna", "Oleg", "Maria"]
for index, name in enumerate(names):
    print(index, name)
```

index	name
0	Anna
1	Oleg
2	Maria

Kolekcja + Pętla + Warunek

Jednym z najczęstszych wzorców w programowaniu jest przejrzenie kolekcji i zrobienie czegoś Tylko przy dopasowaniu wartości:

spisok_cikl_uslovie.py

```
scores = [95, 82, 91, 58, 77]
for score in scores:
    if score >= 90:
        print(score, „Świetnie!”)
```

Kolekcja + Pętla + Warunek – Wiązka fundamentalna
Ten wzorzec — „przejdź przez kolekcję i filtruj według warunku”

Praktyka: Listy, enumerate() i filtrowanie pętli
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/11-15/index.html)

Kompendium, aliasing, == i is

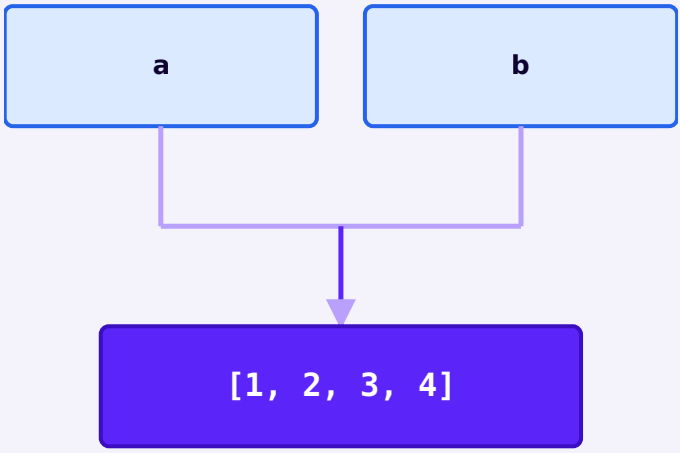
Dwie nazwy mogą wskazywać na ten sam lista — i to nie jest błąd, lecz fundamentalna cecha tego, jak Python działa z obiektami zmiennymi.

Aliasing: dwa imiona to ten sam obiekt

W §11.7 już widzieliśmy ostrzeżenie: `b = a` nie tworzy kopii listy. Przyjrzyjmy się bliżej, co naprawdę się dzieje.

aliasing.py

```
a = [1, 2, 3]
b = a
b.append(4)
print(a)   # [1, 2, 3, 4] — też się zmieniło!
print(b)   # [1, 2, 3, 4]
```



a i b to DWIE NAZWY tego samego obiektu listy, a nie dwie listy

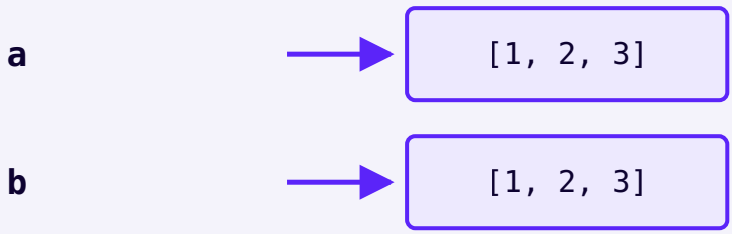
Nie stworzyliśmy drugiego lista — stworzyliśmy drugie imię
`b = a` nie kopiuje danych. Python po prostu mówi: "Teraz imię `b` wskazuje na to samo miejsce co `a`". Formalny termin tego to **aliasing** (współdzielone odniesienie). Zmiana przez którąkolwiek z nazw jest widoczna w obu przypadkach, ponieważ obiekt lista jest w rzeczywistości taki sam.

== porównuje treść is porównuje obiekt

Świetna okazja, by skonsolidować materiał z rozdziału 9 na nowym materiale:

eq_vs_is.py

```
a = [1, 2, 3]
b = [1, 2, 3]
print(a == b)   #True – Ta sama zawartość
print(a is b)   #False to dwie RÓŻNE byty

c = a
print(a is c)   #True to c ta sama nazwa – alias a
```



a i b — dwa różne obiekty o tej samej zawartości: a == b, ale nie a is b

Operator	co to sprawdza?	a = [1,2,3]; b = [1,2,3]
`==`	czy zawartość jest taka sama	`True`
`is`	ten sam obiekt w pamięci	`False`

★★ Zadanie samodzielne
Przewidywać wynik
Określone: `x = [10, 20]`, `y = x`, `y.append(30)`. Bez uruchamiania kodu odpowiedz: co jest równe `x`? A jeśli zamiast `y = x` było `y = x.copy()`?

Praktyka: aliasing, == i is
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji
Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/11-16/index.html)

Listy kopiowania

Kopiowanie zewnętrznego lista jest łatwe, ale listy zagnieżdżone w nim pozostają domyślnie współdzielone. Przyjrzyjmy się, dlaczego i co z tym zrobić.

Trzy Bezpieczne Sposoby Kopiowania lista

sposoby_kopirovaniya.py

```
original = [1, 2, 3]

kopiya_1 = original.copy()
kopiya_2 = list(original)
kopiya_3 = original[:]

kopiya_1.append(4)
print(original)   # [1, 2, 3] — nie zmieniło się
print(kopiya_1)   # [1, 2, 3, 4]
```

Wszystkie trzy sposoby tworzą nową ZEWNĘTRZNĄ lista. Ale co jeśli same elementy są Listy?

Powierzchowna kopia (shallow copy) to ważny niuans

shallow_copy_trap.py

```
original = [["Anna", 10], ["Bob", 20]]
kopiya = original.copy()

kopiya[0][1] = 999
print(original)   # [['Anna', 999], ['Bob', 20]] — również się zmieniło!
```

Dwie RÓŻNE listy zewnętrzne, ale ich elementy to TE same listy wewnętrzne

. copy() kopiuje tylko jeden poziom gnieżdżenia
`.copy()` (oraz `list(...)`, i cięcie `[:]`) tworzą nowy zewnętrzny lista, ale **nie** kopiować zagnieżdżone listy w jego obrębie — pozostają one współdzielone. To się nazywa **powierzchowna kopia (shallow copy)**. Modyfikacja podlisty poprzez kopię jest również widoczna w oryginale.

copy.deepcopy() — gdy potrzebna jest całkowicie niezależna kopia

deepcopy.py

```
import copy

original = [["Anna", 10], ["Bob", 20]]
polnaya_kopiya = copy.deepcopy(original)
polnaya_kopiya[0][1] = 999
print(original)   # [['Anna', 10], ['Bob', 20]] — nie zmieniło się
```

deepcopy nie zawsze jest poprawną odpowiedzią z definicji
`copy.deepcopy()` rekurencyjnie próbuje skopiować całą zagnieżdżoną strukturę w całości. Rozwiązuje to powyższy problem, ale dla dużych lub skomplikowanych struktur głębokie kopiowanie może być zauważalnie wolniejsze i nie zawsze jest potrzebne — często wystarczy skopiować tylko poziom, który faktycznie będzie się zmieniał.

Praktyka: kopiowanie list i powierzchowne kopiowanie
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji
Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/11-17/index.html)

Jeszcze ciekawsze rzeczy z listami!

Przydatne wbudowane funkcje, listy zagnieżdżone, znana pułapka mnożenia list zagnieżdżonych – oraz bardziej zwięzły sposób tworzenia nowych list.

Kilka dodatkowych sztuczek, które często przydają się przy pracy z listami.

len(), min(), max(), sum()

vstroennye_funkcii.py

```
chisla = [4, 8, 15, 16, 23, 42]
print(len(chisla))   # 6 — liczba elementów
print(min(chisla))   # 4
print(max(chisla))   # 42
print(sum(chisla))   # 108
```

Listy Listy (Podlisty) - Macierz

Elementem listy może być inny lista — tak powstają tabele (macierze):

vlozhennye_spiski.py

```
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
print(matrix[0])      # [1, 2, 3] — pierwszy łańcuch znaków
print(matrix[0][1])   #2 jest drugim elementem pierwszej
linijki
print(matrix[1][2])   #6 — łańcuch znaków 1, kolumna 2
```

	kolumna 0	kolumna 1	kolumna 2
łańcuch znaków 0	1	2	3
łańcuch znaków 1	4	5	6
łańcuch znaków 2	7	8	9

matrix[1][2] → 6 – najpierw łańcuch znaków, potem kolumna

perebor_matricy.py

```
for row in matrix:
    for value in row:
        print(value, end=' ')
    print()
```

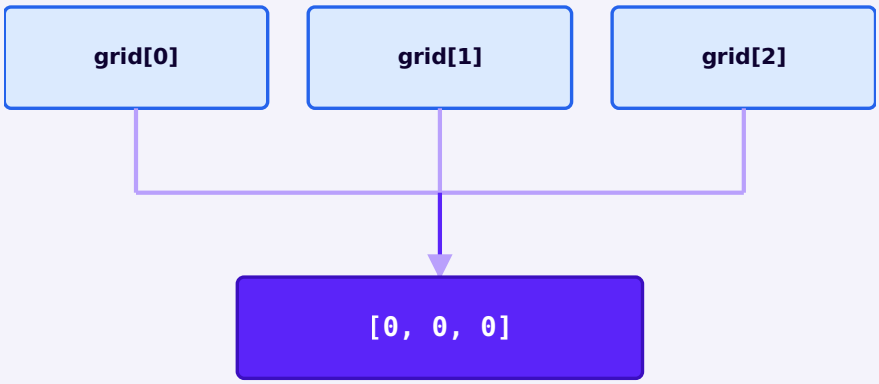
Zagnieżdżony lista + zagnieżdżona pętla — bezpośredni przeniesienie z rozdziału 10
Zewnętrzna pętla idzie po łańcuch znaków, wewnętrzna — po wartościach w wierszu. To dokładnie ten sam wzorzec zagnieżdżonych pętli, co tabliczka mnożenia w rozdziale 10, tylko teraz dane są już gotowe w postaci macierzy, a nie obliczane na bieżąco.

Pułapka: [[0] * 3] * 3

chcę stworzyć siatkę 3×3 z zer w jednej linii — ale to częsty i podstępny błąd:

lovushka_umnozheniya.py

```
grid = [[0] * 3] * 3
grid[0][0] = 1
print(grid)   # [[1, 0, 0], [1, 0, 0], [1, 0, 0]] — WSZYSTKIE linie się zmieniły!
```



[[0]*3]*3 tworzy JEDEN wewnętrzny lista i kopiuje do niego odnośnik trzy razy

***3 powtarza odwołanie, zamiast tworzyć trzy niezależne listy**

`[значение] * n` dla wartości niezmiennych (liczb, łańcuch znaków) działa bezpiecznie. Ale gdy `значение` sam jest listą, `* n` po prostu kopiuje odniesienie do tych samych wewnętrznych lista n czasów — dokładnie ta sama zasada aliasing z §11.9. Poprawnym sposobem utworzenia trzech NIEZALEŻNYCH wierszy jest comprehension:

pravilnaya_setka.py

```
grid = [[0] * 3 for _ in range(3)]
grid[0][0] = 1
print(grid)   # [[1, 0, 0], [0, 0, 0], [0, 0, 0]] - tylko pierwszy łańcuch znaków
```

Iteracja listy pętlą for

perebor_spiska_v4.py

```
fruits = [„jabłko”, „banan”, „wiśnia”]
for fruit in fruits:
    print(fruit)
```

Budowanie listy: Pętla z generatorem append() → List

Klasyczne podejście

```
kvadraty = []
for n in range(1, 6):
    kvadraty.append(n ** 2)
print(kvadraty)
```

Nowoczesny Python (list comprehension)

```
kvadraty = [n ** 2 for n in range(1, 6)]
print(kvadraty)
```

Co używać dzisiaj: list comprehension (generator list) dla prostych przypadków — istnieje w Python od wersji 2.0, więc to nie kwestia nowości, a kwestia stylu: krócej i, przy pewnej praktyce, czyta się jak jedno zdanie („kwadrat n dla każdego n z zakresu"). Dla bardziej skomplikowanej logiki (z kilkoma warunkami lub efektami ubocznymi) pętla z `append()` często pozostaje jaśniejsze — nie zawsze jest „gorzej"

Praktyka: len/min/max/sum, listy zagnieżdżone, list comprehension

interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/11-04/index.html)

Mini-projekt — automatyczna kolorowa gwiazda

Lista kolorów + cykl + Turtle to gwiazda, w której każdy promień ma swój własny kolor.

Połącz listy z Turtle (Rozdziały 6-7): narysuj gwiazdę, gdzie każdy promień ma inny kolor z predefiniowanej listy.

raznocvetnaya_zvezda.py

```
cveta = ["red", "orange", "yellow", "green", "blue"]

for cvet in cveta:
    artist.pencolor(cvet)
    artist.forward(150)
    artist.right(144)  # Kąt Pięcioramiennej Gwiazdy z rozdziału 6
```

Lista dowolnej długości — ten sam kod

Dodaj do `cveta` jeszcze kilka wartości — pętla `for cvet in cveta` dostosowuje się do nowej długości listy, nie zmieniając reszty kodu.

★★ Zadanie samodzielne

Losowe kolory

Zamień lista stałych kolorów na random.choice() z listy — aby kolor każdego promienia był wybierany losowo.

Praktyka: Kolorowa Gwiazda (Listy + Turtle)

Moduł turtle otwiera natywne okno Python – uruchamiane lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/11-05/index.html)

Krotki

Prawie lista, ale niezmienny – i dlatego czasem jest lepszym wyborem.

Poprowadzenie (`tuple`) nie jest „lista w kółko nawiasy”

kortezhi.py

```
point = (10, 20)
print(point)
print(point[0])   #10 — indeksowanie działa jak listy
```

kortezh_nelzya_menyat.py

```
point = (10, 20)
point[0] = 99   # TypeError: 'tuple' object does not support
item assignment
```

Krotka — sekwencja, tak jak lista

`len`, indeksowanie, cięcia, siła brute, `in` — wszystko to działa dla krotki tak samo jak dla listy (§11.1–§11.2). Jest tylko jedna różnica, ale Zasada: Krotki nie można zmienić.

	list	tuple
Składnia	`[1, 2, 3]`	`(1, 2, 3)`
Czy oszukujemy?	Tak	nie
Indeksowanie, wycinki, len, in	Tak	Tak

Pojedyncza krotka – potrzebny jest przecinek

odinochnyj_kortezh.py

```
not_a_tuple = (42)
print(type(not_a_tuple))   # <class 'int'>

one = (42,)
print(type(one))            # <class 'tuple'>
```

Krotka powstaje przez przecinek, a nie nawiasy

`(42)` — to po prostu liczba 42 w nawiasach, jak w matematyce. Krotką czyni przecinek: `(42,)`. Bez przecinka Python nie zorientuje się, że chcesz mieć krotkę, i cicho utworzy liczbę regularną — bez błędu, co jest szczególnie podstępne.

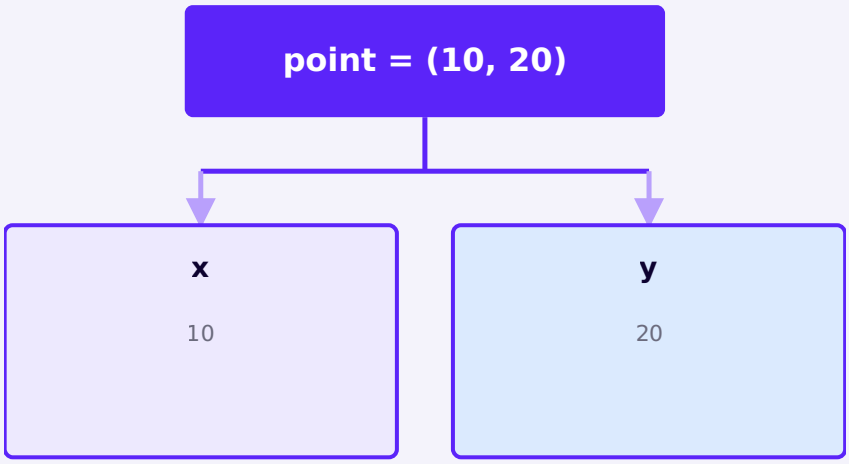
Packing i unpacking

packing.py

```
point = 10, 20   nawiasy # są opcjonalne – Python „pakuje”
print(point, type(point))
```

raspakovka.py

```
point = (10, 20)
x, y = point
print(x, y)   # 10 20
```



x, y = point - Rozpakowanie: Liczba nazw musi odpowiadać liczbie wartości

Już użyliśmy rozpakowywania

`artist.position()` w module turtle zwraca dokładnie taki krotkę `(x, y)` możesz od razu rozłożyć go na dwie zmienne.

Klasyczny trik: Wymiana wartości

swap.py

```
a = 1
b = 2
a, b = b, a
print(a, b)   # 2 1
```

Działa dzięki tej samej parze „packing + unpacking” `(b, a)` to jest rozpakowane w `a, b` — bez zmiennej pośredniej.

Star Unboxing

star_unpacking.py

```
first, *middle, last = [1, 2, 3, 4, 5]
print(first)    # 1
print(middle)   # [2, 3, 4]
print(last)     # 5
```

Star zbiera „wszystko inne”

`*middle` bierze wszystkie wartości, które nie są parsowane innymi nazwami, i pakuje je do zwykłego lista — nawet jeśli krotka jest rozpakowana.

Funkcje mogą zwracać wiele wartości jednocześnie

divmod_primer.py

```
quotient, remainder = divmod(17, 5)
print(quotient, remainder)   # 3 2
```

`divmod()` z rozdziału 4 w rzeczywistości zwraca jedną krotkę `(3, 2)` — rozkładamy to na dwie zmienne jednocześnie.

Niuanse: Krotka może zawierać obiekty zmienne

tuple_s_listom.py

```
t = ([1, 2], [3, 4])
t[0].append(99)
print(t)   # ([1, 2, 99], [3, 4]) — udało się!
```

Niezmienność krotki dotyczy tylko samych „komórek”

`t[0] = [...]` (zamienić link na inny lista) — zabronione. A oto `t[0].append(99)` (zmiana samego lista, do którego odnosi się krotka) jest dozwolona: krotka nie pozwala na przeorganizowanie TEGO, co przechowuje dla każdej pozycji, ale nie zabrania zmianie obiektów zmiennych wewnątrz.

Praktyka: krotki i rozpakowywanie
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/11-06/index.html)

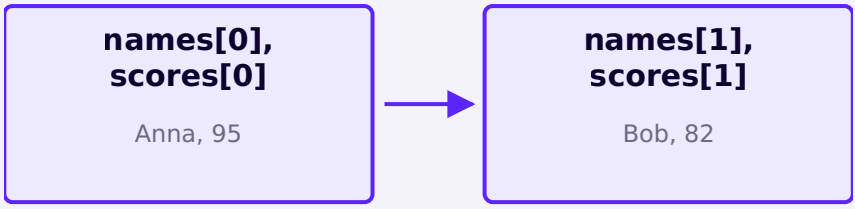
zip() i rozpakowywanie

Jedno narzędzie, które często idzie w parze z krotkami i rozpakowywaniem: łączenie dwóch list element po elemencie.

zip() — połączyć obie listy parami

zip_primer.py

```
names = ["Anna", "Bob"]
scores = [95, 82]

for name, score in zip(names, scores):
    print(name, "—", score)
```



zip() bierze po jednym punkcie z każdej listy na tym samym miejscu

zip() zatrzymuje się na najkrótszym
Jeśli listy mają różną długość, `zip()` domyślnie kończy się zanim krótszy — „dodatkowe”

zip() + dict — zbieraj słownik z dwóch list

zip_v_dict.py

```
names = ["Anna", "Bob", "Maria"]
scores = [95, 82, 91]

result = dict(zip(names, scores))
print(result)   # {'Anna': 95, 'Bob': 82, 'Maria': 91}
```

Rozpakowywanie w pętlach

Korzystaliśmy już z tego dla słowników (§11.17) — ta sama zasada, co rozpakowanie krotki z §11.13, tylko zastosowana wewnątrz pętli:

raspakovka_v_cikle.py

```
pary = [("Anna", 95), ("Bob", 82)]
for name, score in pary:
    print(f"{name}: {score}")
```

Praktyka: zip() i rozpakowywanie w pętlach
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/11-18/index.html)

Zestawy

Kolekcja bez duplikatów i pozycji to szybki sposób na usunięcie duplikatów i porównanie zestawów wartości.

Wielu (`set`) to zbiór figurek nawiasy, które mają dwie kluczowe różnice względem listy: elementy listy nie są powtarzane, a trafienia to Według indeksu numerycznego, nie, zbiór nie jest sekwencją pozycyjną.

mnozhestva.py

```
chisla = {1, 2, 2, 3, 3, 3}
print(chisla)   # {1, 2, 3} — powtórzenia zniknęły same z siebie
```

Po co są zbiory?

Dwa najczęstsze przypadki to szybkie usunięcie powtórzeń z listy i szybkie sprawdzenie, czy w ogóle są wartość kolekcji.

primenenie_mnozhestv.py

```
spisok_s_povtorami = [1, 2, 2, 3, 1, 4]
unikalnye = set(spisok_s_povtorami)
print(unikalnye)   # {1, 2, 3, 4}
```

chlenstvo_mnozhestva.py

```
allowed = {"admin", "editor", "viewer"}
role = "editor"
if role in allowed:
    print(„Dostęp dozwolony”)
```

Zbiór nie jest „przypadkowo uporządkowany” — po prostu nie jest pozycyjny

Nie traktuj set jak „losowego lista” **nie** stanowiska ogólnie to pisanie `my_set[0]` nie może (TypeError), a kod nigdy nie powinien polegać na kolejności zbiór pętli.

Pusty zbiór — częsta pułapka

pustoe_mnozhestvo.py

```
print(type({}))        # <class 'dict'> — to puste słownik!
print(type(set()))     # <class 'set'> to pusta zbiór
```

{} jest pustym słownik, a nie pustym zbiór

Nawiasy klamrowe bez zawartości Python domyślnie uznaje za puste `dict` (§11.17), ponieważ słowniki pojawiły się w języku jako bardziej podstawowy przypadek użycia `{}` . Aby stworzyć pusty zbiór, musisz wyraźnie napisać `set()` .

add, remove, discard, pop, clear

metody_mnozhestv.py

```
tags = {"python", "beginner"}
tags.add("tutorial")
tags.discard("missing")   # bez błędu, nawet jeśli przedmiot
nie jest obecny
print(tags)
```

Metoda	Jeśli elementu nie ma
`.remove(x)`	`KeyError`
`.discard(x)`	nic się nie dzieje, nie ma żadnego błędu

pop() usuwa element ARBITRARY ze zbioru
W przeciwieństwie do `list.pop()` zbiór nie zawiera pojęcia „ostatniego elementu” `set.pop()` usuwa i zwraca jeden element, ale nie wiadomo wcześniej, który to jest.

Operacje na zestawach

operacii_mnozhestv.py

```
a = {1, 2, 3}
b = {2, 3, 4}

print(a | b)   # union: {1, 2, 3, 4}
print(a & b)   # przecięcie: {2, 3}
print(a - b)   # Różnica: {1}
```

Pełna algebra zbiorów — w §11.16
To tylko wstęp. Diagramy Venna dla wszystkich czterech operacji, podzbiorów i hashowalności zostaną omówione w następnej sekcji.

Praktyka: Scenografia i ich działanie
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/11-07/index.html)

Operacje setów i możliwość hasowania

Cztery operacje algebry zbiorów na diagramach Venna — i dlaczego lista nie można umieścić wewnątrz zbioru.

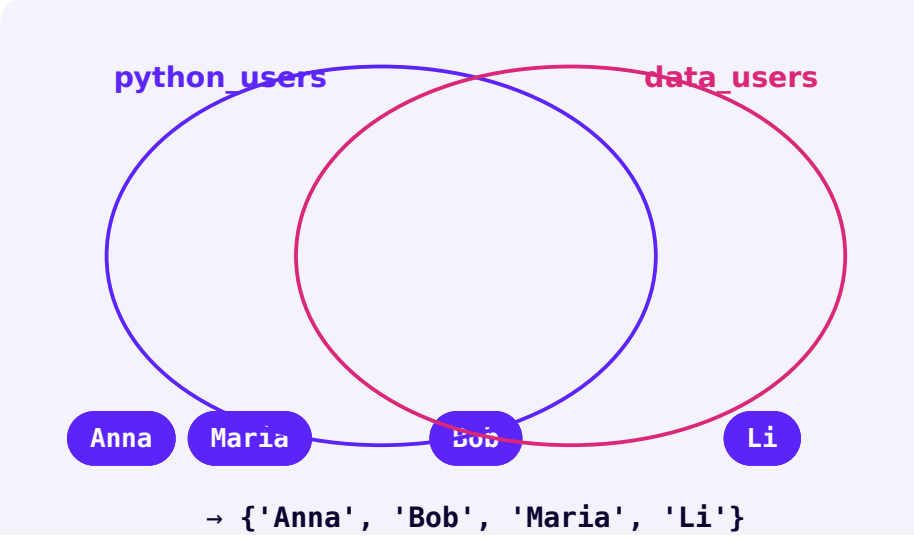
Zbiory jako okręgi na diagramie Venna

Dwie grupy uczestników kursu są wygodnym przykładem, który pokazuje wszystkie cztery operacje jednocześnie:

dve_gruppy.py

```
python_users = ['Anna', 'Bob', 'Maria']
data_users = ['Bob', 'Li']
```

Zjednoczenie – union, |

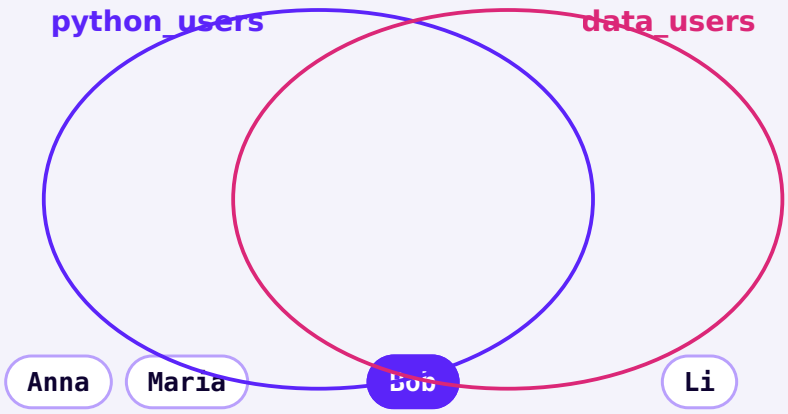


python_users | data_users — każdy, kto jest przynajmniej w jednej grupie

union.py

```
print(set(python_users) | set(data_users))
```

Przecięcie — intersection, &



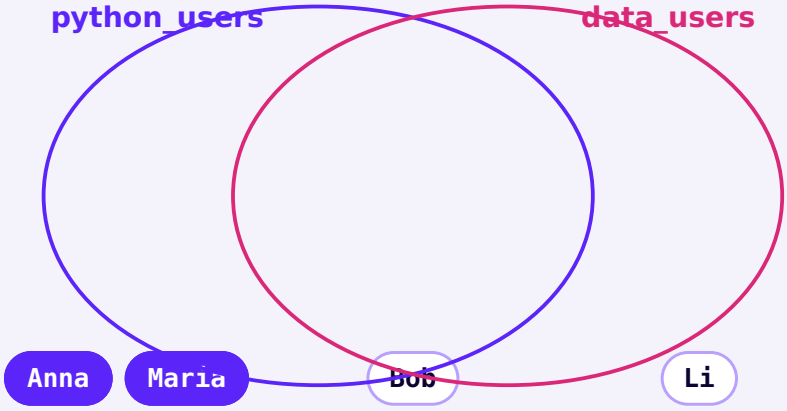
→ {'Bob'}

python_users & data_users – która należy do OBU grup

intersection.py

```
print(set(python_users) & set(data_users))
```

Różnica – difference, -



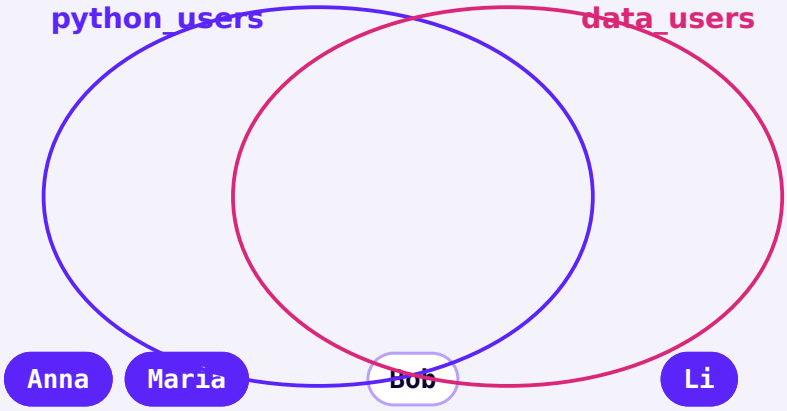

```
→ {'Anna', 'Maria'}
```

python_users - data_users — tylko w pierwszej grupie

difference.py

```
print(set(python_users) - set(data_users))
```

Różnica symetryczna — symmetric_difference, ^




```
→ {'Anna', 'Maria', 'Li'}
```

python_users^data_users – w tej samej grupie, ale nie w obu jednocześnie

symmetric_difference.py

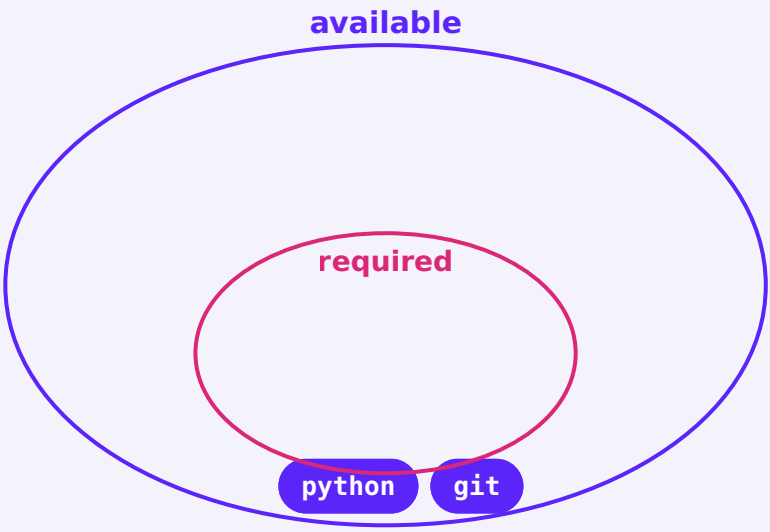
```
print(set(python_users) ^ set(data_users))
```

Chirurgia	Operator	Metoda
zjednoczenie	`a | b`	`a.union(b)`
skrzyżowanie	`a & b`	`a.intersection(b)`
różnica	`a - b`	`a.difference(b)`
różnica symetryczna	`a ^ b`	`a.symme-tric_difference(b)`

Pod zbiór i nad zbiór

podmnozhestvo.py

```
required = {"python", "git"}
available = {"python", "git", "docker", "linux"}

print(required <= available)     #True — required jest
CAŁKOWICIE zawarty w available
print(required.issubset(available))   # tak samo
```



required < = available — required (wewnętrzne koło) leży całkowicie wewnątrz available (zewnętrznego koła)

Operator / metoda	Znaczenie
`a <= b` / `a.issubset(b)`	wszystkie elementy a znajdują się w b
`a >= b` / `a.issuperset(b)`	wszystkie elementy b znajdują się w a
`a.isdisjoint(b)`	a i b nie mają w ogóle wspólnych elementów

Hashowalność: dlaczego niemożliwe jest przechowywanie lista w zbiorze

hashability_error.py

```
bad = {[1, 2], [3, 4]}
# TypeError: cannot use 'list' as a set element (unhashable
type: 'list')
```

Elementy zbioru i klucze słownika powinny być wystarczająco „stabilne”
Python musi być pewny, że wartość wewnątrz zbioru (lub klucz słownika) nie zmieni się niezauważalnie, dopóki tam jest. Formalny termin na taką stabilność — **hashowalność (hashable)**. Typy zmienne (lista, słownik, zbiór) nie są z tego powodu haszalne.

Co można umieścić w zbiór / używać jako klucz słownikowy

ZWYKLE HASZOWALNE

int, float, str
bool, bytes
tuple jeśli wszystkie elementy są również hashowalne
frozenset

NIE SĄ HASHOWANE

list
dict
set

frozenset jest niezmiennym zbiór

frozenset_primer.py

```
zamorozhennoe = frozenset({"python", "git"})
print(zamorozhennoe)
# zamorozhennoe.add(„docker”
```

Kiedy przyda się frozenset

`frozenset` jest potrzebny, gdy zbiór sam musi być hashowalny, na przykład, aby użyć go jako elementu innego zbioru lub jako klucza słownikowego, co ma miejsce w przypadku klucza regularnego `set` niemożliwe.

Praktyka: algebra zbiorów, podzbiory, hasowalność
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/11-19/index.html)

Słowniki

Najbardziej elastyczna wbudowana kolekcja Python to dostęp według klucza zamiast indeksu numerycznego.

Od pozycji do klucza

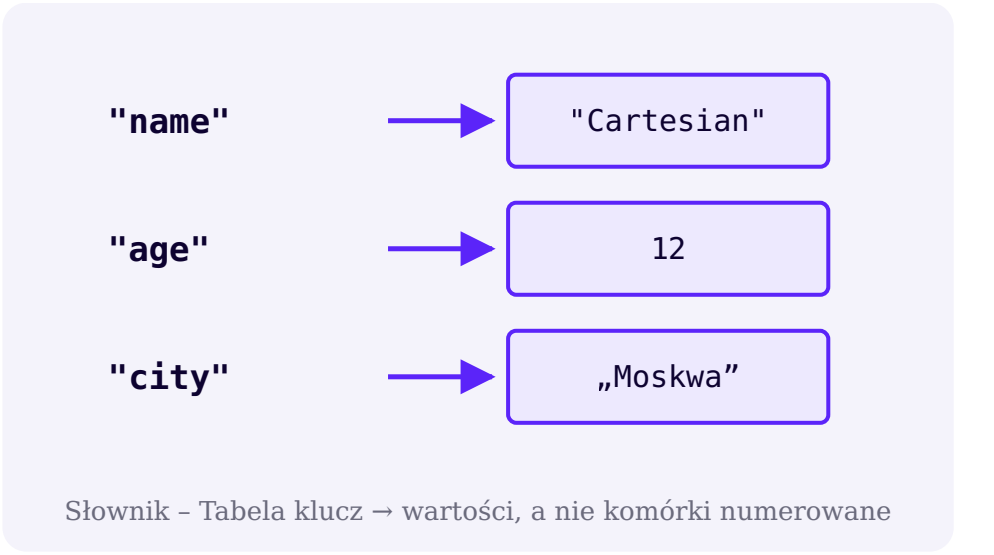
Dla listy dostęp odbywa się według pozycji numerycznej. Dla słownika dostęp jest dostępny przez **kluczu**, zazwyczaj Linia:

	list	dict
Model dostępu	pozycja → znaczenie	kluczowa → wartość
Przykład	`names[0]`	`student["name"]`

Słownik (`dict`) to najbardziej elastyczny zbiór: przechowuje pary klucz-wartość zamiast prostych wartości w kolejności.

slovari.py

```
student = {
    "name": "Cartesian",
    "age": 12,
    "city": „Moskwa”,
}
print(student["name"])   # Cartesian
```



Słownik – Tabela klucz → wartości, a nie komórki numerowane

Powtarzanie klucza dosłownie – ostatnia wartość wygrywa
Jeśli zapiszesz ten sam klucz dwa razy w nawiasach kręconych, nie będzie pomyłki — pozostanie tylko wartość zapisana później.

Dodawanie i zmienianie wartości

izmenenie_slovarya.py

```
student["age"] = 13          # zmieniać istniejącą wartość
student["grade"] = „7. klasa” # dodać nowy klucz
print(student)
```

Dostęp do kluczy: [] i get()

dostup_get.py

```
print(student["name"])        # Cartesian
print(student.get("email"))    # None — brak klucza, ale też
brak błędów
print(student.get("email", „no email”))
# z wyraźnie określoną wartością domyślną
```

KeyError - Dostęp do nieistniejącego klucza
`student["phone"]`, jeśli takiego klucza nie ma, wywoła `KeyError`. Bezpieczniejszą drogą jest `student.get("phone")` to po prostu wróci `None` zamiast błędu. Nawiasy kwadratowe nie są „gorsze”

Wyszukiwanie słownika

perebor_slovarya.py

```
for key, value in student.items():
    print(f"{key}: {value}")
```

Kontrola dostępności kluczy

proverka_klyucha.py

```
print("name" in student)      # True
print("phone" in student)     # False
```

Praktyka: słowniki — tworzenie, zmiana, iteracja
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/11-08/index.html)

Metody słowników

pop, update, setdefault i reprezentacje keys/values/items to narzędzia, bez których żaden prawdziwy słownik nie może się obejść.

pop() i popitem()

dict_pop.py

```
student = {"name": "Cartesian", "age": 13, "grade": „7. klasa”}
age = student.pop("age")
print(student, age)   # {'name': 'Cartesian', 'grade': '7
klasa'} 13
```

dict_popitem.py

```
last_pair = student.popitem()
print(last_pair)   # ('grade', '7 klasa') — ostatnia dodana
para
```

pop() słownik — według tonacji, nie pozycji

W przeciwieństwie do `list.pop(index)`, w słowniku `.pop(key)` akceptuje KLUCZ, a nie pozycję numeryczną — słownik po prostu nie ma pozycji. Możesz przekazać wartość domyślną jako drugi argument, jeśli klucz może nie istnieć: `student.pop("phone", None)`.

update() — połączyć kilka par naraz

dict_update.py

```
student = {"name": "Cartesian", "age": 13}
student.update({"age": 14, "city": „Moskwa”})
print(student)   #age zastąpił city dodał
```

setdefault() - Dodaj wartość tylko wtedy, gdy klucz nie istnieje

setdefault.py

```
counts = {}
word = "python"
counts.setdefault(word, 0)
counts[word] += 1
print(counts)   # {'python': 1}
```

setdefault() = „jeśli nie masz klucza, stwórz go z wartością domyślną”

`counts.setdefault(word, 0)` nic nie robi, jeśli `word` już znajduje się w słowniku i tworzy rekord o wartości 0, jeśli nie było klucza — w obu przypadkach, po wywołaniu można bezpiecznie zapisać `counts[word] += 1`. Pełny algorytm obliczania częstotliwości słów tym podejściem znajduje się w §11.24.

keys(), values(), items() nie są zwykłymi listami

keys_values_items.py

```
student = {"name": "Cartesian", "age": 13}
print(student.keys())     # dict_keys(['name', 'age'])
print(student.values())   # dict_values(['Cartesian', 13])
print(student.items())
# dict_items([('name', 'Cartesian'), ('age', 13)])
```

To są „występy na żywo”

`.keys()`, `.values()` oraz `.items()` obiekty reprezentacji zwracającej (`dict_keys` i tak dalej) — odzwierciedlają słownik w czasie rzeczywistym. Do iteracji pętlą nie ma to znaczenia, ale jeśli potrzebny jest konkretnie lista, opakuj w `list(student.keys())`.

Kolejność słownika

Nowoczesne Python zachowuje kolejność wstawiania
Począwszy od Python 3.7, kolejność par w słowniku gwarantowana jest taka, jaką zostały dodane — jest to oficjalna gwarancja języka, a nie przypadkowa implementacja. Ale słownik pozostaje **wyświetlenie** (dostęp przez klucz) zamiast kolekcji pozycyjnej — nie pisz kodu opartego na pozycjach słownikowych liczb.

Sprawdzanie in - tylko klucze

membership_dict.py

```
student = {"name": "Cartesian", "age": 13}
print("name" in student)          #True – taki KLUCZ istnieje
print("Cartesian" in student)     # False – to wartość, nie
klucz!
print("Cartesian" in student.values())   # True – i tak
poprawnie sprawdzić wartość
```

in słownik sprawdza klucze, a nie wartości
Powszechny błąd początkujących: `value in my_dict` sprawdza, czy istnieje `value` wśród KLUCZY. Aby sprawdzić wartości, musisz wyraźnie zapisać `value in my_dict.values()`.

Klucze słownikowe muszą być również haszalne

hashable_keys.py

```
{["x", "y"]: 1}
# TypeError: cannot use 'list' as a dict key (unhashable type:
'list')
```

Ta sama zasada hashowalności z §11.16 – lista nie może być używana jako klucz słownikowy dla z tego samego powodu, dla którego nie można go umieścić w zbiór.

Praktyka: pop, update, setdefault, keys/values/items
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/11-20/index.html)

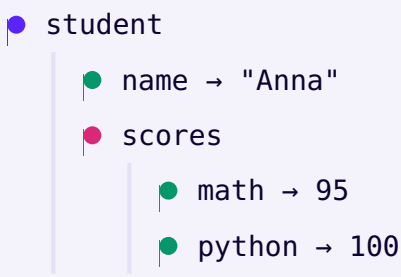
Zagnieżdżone struktury

Rzeczywiste dane rzadko są płaskie: listy słowników i słowników w słownikach są normą, a nie wyjątek.

Zagnieżdżone słownik

vlozhennyj_slovar.py

```
student = {
    "name": "Anna",
    "scores": {"math": 95, "python": 100},
}
print(student["scores"]["python"])   # 100
```

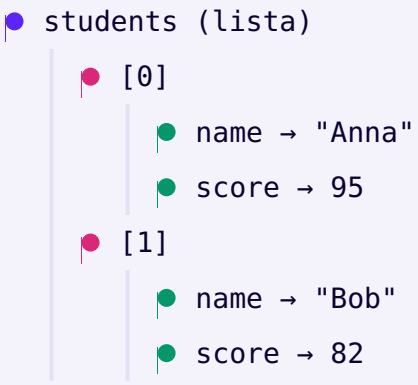


student['scores']['python'] - Najpierw klucz obcy, potem klucz wewnętrzny

Lista słowników jest najczęstszym rzeczywistym wzorcem

spisok_slovarej.py

```
students = [
    {"name": "Anna", "score": 95},
    {"name": "Bob", "score": 82},
]

for student in students:
    print(student["name"], student["score"])
```



Lista słowników — tak często wyglądają rzeczywiste dane (wpisy z bazy danych, odpowiedzi API)

Lista słowników jest prawie JSON

Dokładnie tak organizowane są dane pochodzące z wielu serwisów internetowych (JSON) szczegółowo JSON omówimy w jednym z następnych rozdziałów, ale struktura „lista rekordów, każdy rekord ma swoje pola”

Słownik list — odwrotna wersja

slovar_spiskov.py

```
classroom = {
    "students": ["Anna", "Bob"],
    "scores": [95, 82],
}
```

Oba przedstawienia są prawidłowe — które wybrać, zależy od tego, jak będą przetwarzane dane dalej: lista słowniki są wygodniejsze, jeśli potrzebny jest często JEDEN pełny wpis (znaleźć Bob i wszystkie jego dane); słownik listy — jeśli częściej potrzebna jest JEDNA cała kolumna (wszystkie imiona lub wszystkie oceny).

Praktyka: Zagnieżdżone słowniki i słowniki lista
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/11-21/index.html)

Przemiany i comprehensions

Od list()/tuple()/set() do zwartego sposobu budowania nowych kolekcji w jednej linii — ale dopiero po zrozumieniu zwykłego cyklu.

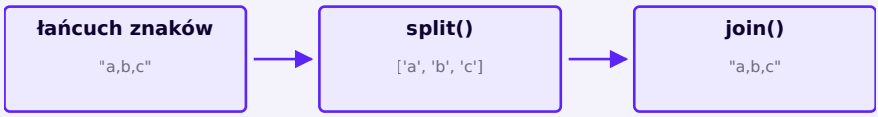
Konwersje między kolekcjami

preobrazovaniya.py

```
print(list("Python"))           # ['P', 'y', 't', 'h', 'o',
'n']
print(tuple([1, 2, 3]))         # (1, 2, 3)
print(set([1, 1, 2, 3]))         # {1, 2, 3}
```

dict() wymaga pary „klucz, wartość”
`dict(...)` przekształca w słownik nie każdą kolekcję, tylko taką, z której można utworzyć pary: na przykład, `dict(zip(a, b))` z §11.14. Proste `dict([1, 2, 3])` spowoduje błąd.

String lista ↔ to najczęstsza konwersja



text.split() → lista, ' '. join(lista) → łańcuch znaków z powrotem

split_join.py

```
text = "python git linux"
words = text.split()
print(words)          # ['python', 'git', 'linux']
print(" ".join(words))   # "python git linux"
```

sorted() dla różnych kolekcji

sorted_raznoe.py

```
print(sorted((3, 1, 2)))          # [1, 2, 3] - z krotki
print(sorted({3, 1, 2}))           # [1, 2, 3] – zestawu
print(sorted({"b": 1, "a": 2}))   # ['a', 'b'] – KLUCZE są
sortowane ze słownika
```

sorted() zawsze zwraca lista
Niezależnie od tego, co było przy wejściu – konwoj, zbiór czy słownik – `sorted()` zawsze daje to, co zwykle. `list` .

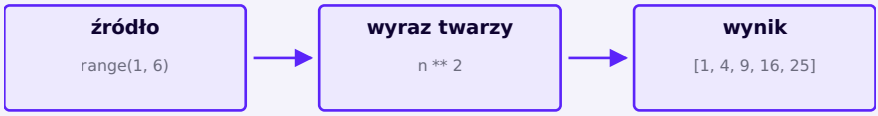
List comprehension już jest znajome, teraz krok po kroku

comprehension_klassika.py

```
kvadraty = []
for n in range(1, 6):
    kvadraty.append(n ** 2)
print(kvadraty)
```

comprehension_novyj.py

```
kvadraty = [n ** 2 for n in range(1, 6)]
print(kvadraty)
```



[wyrażenie for n in źródło] jest tym samym co pętla, ale w jednej linii

Comprehension z warunkiem

comprehension_uslovie.py

```
chetnye = [n for n in range(10) if n % 2 == 0]
print(chetnye)   # [0, 2, 4, 6, 8]
```

Końcowy warunek - Filtr: Element zostanie uwzględniony w wyniku tylko wtedy, gdy warunek True

Set i dict comprehension

set_comprehension.py

```
unikalnye_bukvy = {ch.lower() for ch in "Python"}
print(unikalnye_bukvy)
```

dict_comprehension.py

```
kvadraty_slovar = {n: n ** 2 for n in range(5)}
print(kvadraty_slovar)   # {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

Ta sama idea „źródło → wyrażenie → wynik”

Trochę głębiej — wyrażenia generatorowe

`(n ** 2 for n in range(5))` — wyrażenie generatorowe, podobne do comprehension, ale nie budujące lista całego na raz w pamięci, tylko zwracające wartości po jednej. Przyda się później, gdy będziemy mówić o iteratorach i pracy z dużymi zbiorami danych.

Budowanie listy: cykl vs comprehension

Cykl z append()

```
kuby = []
for n in range(1, 6):
    if n % 2 == 0:
        kuby.append(n ** 3)
```

List comprehension

```
kuby = [n ** 3 for n in
range(1, 6) if n % 2 == 0]
```

Co używać dzisiaj: comprehension prostych transformacji i filtrów — krótko mówiąc, i czyta się jak jedna myśl. Ale comprehension NIE jest uniwersalnym narzędziem do głębokiego kopiowania i nie zastępuje pętli, jeśli potrzebujesz efektów ubocznych w środku (print, zapis do pliku, kilka warunków z inną logiką) — wtedy zwykła pętla z append() jest łatwiejsza do odczytania.

Praktyka: transformacje i comprehensions
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/11-22/index.html)

Mini Project - Nieskończone Kolory

słownictwo jako „tłumacz”

Słownik świetnie sprawdza się w "tłumaczeniu" jednego znaczenia na inne, na przykład imienia kolory w jego kodzie. Narysujmy postać "pokolorowaną według słownika".

beskonechnye_cveta.py

```
cvetovaya_karta = {
    „ognia”: "red",
    „trawa”: "green",
    „niebo”: "blue",
}

zapros = „niebo”
artist.pencolor(cvetovaya_karta[zapros])
artist.circle(50)
```

★★ Zadanie samodzielne
Dodaj kolory
Dodaj 3-4 kolejne pary słów-kolorów do cvetovaya_karta i narysuj osobny kształt dla każdej z nich.

Praktyka: nieskończone kolory (słowniki + Turtle)
Moduł turtle otwiera natywne okno Python – uruchamiane lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/11-09/index.html)

Jak wybrać odpowiednią strukturę

Teraz, gdy wszystkie cztery kolekcje są znane osobno, — jak zdecydować, której użyć w konkretnym zadaniu.

Tabela zmienności

Typ	Zamówione?	Czy oszukujemy?	Powtórzenia?	Dostęp
`list`	Tak	Tak	Tak	według indeksu
`tuple`	Tak	nie	Tak	według indeksu
`set`	żadnych stanowisk	Tak	nie — tylko unikalne	(in)
`frozenset`	żadnych stanowisk	nie	nie — tylko unikalne	(in)
`dict`	zachowuje kolejność wstawiania	Tak	klawisze są unikalne	według klucza

Tabela skrótu

Co można umieścić w zbiór / używać jako klucz słownikowy

ZWYKLE HASZOWALNE

```
int, float, str
bool, bytes
tuple (jeśli elementy są haszowalne)
frozenset
```

NIE SĄ HASHOWANE

```
list
```

```
dict
set
```

Ostateczny punkt odniesienia do wyboru

Pełna wskazówka wyboru konstrukcji

Czy potrzebujesz pary klucz→wartość? → **dict**

W przeciwnym razie: potrzebne są tylko unikalne wartości, kolejność nie jest ważna? → **set**

Inaczej: potrzebny jest porządek, wartości dokładnie się nie zmienią? → **tuple**

W przeciwnym razie: potrzebujesz porządku i będziesz musiał zmienić treść? → **list**

Zasada – najpierw pomyśl o zadaniu, potem o składni

Prawdziwość kolekcji (truthiness)

Z rozdziału 9: Pusta kolekcja `False` w kontekście logicznym, niepusty — `True`, niezależnie od wartości zawartych w środku:

Pusty → False	Nie pusty → True
`[]`	`[0]`

Pusty → False	Nie pusty → True
`()`	`("",)`
`set()`	`{0}`
`{}`	`{"x": None}`

truthiness_kollekcij.py

```
items = []
if items:
    print(„Są elementy”)
else:
    print(„Lista jest pusta”)
```

if items: jest bardziej idiomatyczny niż if len(items) > 0:
Obie wersje działają tak samo, ale `if items:` bezpośrednio pyta „lista nie pusty?”

Zmiana kolekcji podczas brutalnej siły to kolejna pułapka

mutation_during_iteration.py

```
items = [2, 4, 6, 8]
for item in items:
    if item % 2 == 0:
        items.remove(item)
print(items)   # [4, 8] – nie [], chociaż warunek ten jest odpowiedni dla KAŻDEGO elementu!
```

Nie zmieniaj rozmiaru kolekcji, dopóki ją przeglądasz tym samym cyklem
Usuwanie elementów listy podczas wyliczania tej samej listy przesuwa indeksy tuż pod stopami pętli — niektóre elementy zostaną pomijane. Dla słownika lub zestawu Python wyrzuci `RuntimeError: dictionary changed size during iteration`. Bezpieczne sposoby: zbieraj wynik do NOWEJ kolekcji, używaj comprehension lub iteruj przez kopię jawną — `for item in items.copy():`.

Praktyka: Wybór struktury danych scenariusza
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/11-23/index.html)

Debugowanie kolekcji: 14 typowych błędów

IndexError, KeyError, append/extend zamieszania, aliasing, powierzchowne teksty i inne pułapki są w jednym miejscu, z przykładami i analizą.

14 typowych błędów przy pracy z listami, krotkami, zbiorami i słownikami — każdy z Przykład i wyjaśnienie. Pierwsze dwa będą analizowane osobno za pomocą diagramu, reszta będzie zwarta.

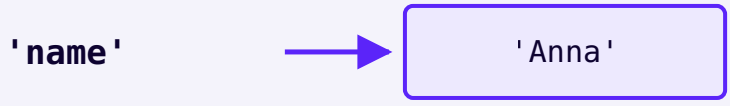
IndexError — wizualnie

items = ['A', 'B', 'C'] - Prawidłowe indeksy to tylko 0, 1, 2 (oraz -1, -2, -3)

indexerror.py

```
items = ['A', 'B', 'C']
print(items[3])
# IndexError: list index out of range
```

KeyError — wizualnie



student ma tylko klucz "name" — wywołanie klucza "email" niczego nie znajduje i powoduje KeyError

keyerror.py

```
student = {'name': 'Anna'}
print(student['email'])
# KeyError: 'email'
```

12 kolejnych powszechnych błędów

1 · IndexError

bug.py

```
items = ['A', 'B', 'C']
print(items[3])
```

Prawidłowe indeksy mieszczą się między 0 a len(items)-1 = 2. Indeks 3 nie istnieje.

2 · KeyError

bug.py

```
student = {'name': 'Anna'}
print(student['email'])
```

W słowniku nie ma takiego klucza. Safer: student.get('email').

3 · ValueError od index()/remove()

bug.py

```
fruits = ['jabłko', 'banan']
fruits.remove('wiśnia')
```

„wiśnia”

4 · TypeError: unhashable type

bug.py

```
bad = {[1, 2]: „znaczenie”}
```

Lista nie może być używana jako klucz słownikowy (ani element zbioru) — jest modyfikowalna, co oznacza, że nie jest haszalna (§11.16).

5 · append() zamiast extend()

bug.py

```
a = [1, 2]
a.append([3, 4])
print(a)   # [1, 2, [3, 4]] zamiast [1, 2, 3, 4]
```

append() zawsze dodaje jeden obiekt. Aby dodać elementy Z listy, musisz extend().

6 · x = x.sort() — utrata listy

bug.py

```
numbers = [3, 1, 2]
numbers = numbers.sort()
print(numbers)   # None
```

sort() mutuje lista na miejscu i zwraca None. Przypisanie wyniku z powrotem usuwa lista.

7 · Aliasing zamiast kopii

bug.py

```
a = [1, 2, 3]
b = a
b.append(4)
print(a)   # też [1, 2, 3, 4]!
```

b = a nie kopiuje lista — oba nazwy wskazują na ten sam obiekt (§11.9). Należy b = a.copy().

8 · Płytka kopia i zagnieżdżony lista

bug.py

```
original = [[1, 2]]
copy_ = original.copy()
copy_[0][0] = 99
print(original)   # [[99, 2]] — również się zmieniło!
```

. copy() kopiuje tylko zewnętrzną lista—listy zagnieżdżone pozostają współdzielone (§11.10). Potrzeba copy.deepcopy().

9 · [[0]*3]*3 — łączna łańcuch znaków

bug.py

```
grid = [[0] * 3] * 3
grid[0][0] = 1
print(grid)   # WSZYSTKIE linie się zmieniły
```

*3 kopiuje odnośnik do tego samego wewnętrznego lista trzy razy. Użyj [[0]*3 for _ in range(3)].

10 · Edytuj listę podczas wyszukiwania

bug.py

```
items = [1, 2, 3, 4]
for x in items:
    if x % 2 == 0:
        items.remove(x)
print(items)   # nie to, czego oczekiwano
```

Usunięcie elementu przesuwa indeksy w trakcie iteracji — część wartości zostaje pominięta. Twórz nowy lista lub iteruj po kopii.

11 · {} zamiast set()

bug.py

```
empty = {}
print(type(empty))   #<class 'dict'>, nie set!
```

Pusty słownik i pusty zbiór wyglądają inaczej tylko w stanie niepustym. Pusty zbiór — zawsze set().

12 · Krotka pojedynczoelementowa bez przecinka

bug.py

```
point = (42)
print(type(point))   # <class 'int'>
```

Krotka powstaje przez przecinek, a nie przez nawiasy. Jest to konieczne (42,).

13 · in w słowniku sprawdza klucze, nie wartości

bug.py

```
student = {'name': 'Anna'}
print('Anna' in student)   # False!
```

'Anna' to wartość, a nie klucz. Aby sprawdzić wartości: 'Anna' in student.values().

14 · Kolejność zagnieżdżonego indeksu jest pomieszana

bug.py

```
matrix = [[1, 2, 3], [4, 5, 6]]
print(matrix[2][0])   # IndexError
```

matrix mają tylko 2 wiersze (indeksy 0 i 1) po 3 kolumny – najpierw łańcuch znaków, potem kolumnę, a nie odwrotnie.

Metoda debugowania: Śledzenie krok po kroku

- Przed uruchomieniem przewidź, co będzie w zmiennej — potem porównaj z rzeczywistym wynikiem.
- W przypadku błędów indeksu/klucza wyraźnie określ ważny zakres lub lista kluczy.
- Jeśli wynik to „dziwny, ale bez błędów”
- print() po każdym kroku to najbardziej wiarygodny sposób, by znaleźć moment, w którym oczekiwania odbiegły od rzeczywistości.

Praktyka: znajdujemy i poprawiamy błędy w kolekcjach
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/11-24/index.html)

Mini-projekt — liczenie częstości słów

łańcuch znaków, pętle i licznik słownik w jednym klasycznym zadaniu — z ręcznym algorytmem i podglądem gotowego narzędzia ze standardowej biblioteki.

Klasyczne zadanie, które łączy prawie wszystko z tego rozdziału: łańcuchy (rozdział 8), pętle (rozdział 10) i słownik jako licznik.

Algorytm krok po kroku

chastota_slov.py

```
text = "python is great and python is fun"
words = text.lower().split()

counts = {}
for word in words:
    counts[word] = counts.get(word, 0) + 1

print(counts)
```

counts.get(word, 0) + 1 — jeśli słowa nie ma już tam, get() zwróci 0, a licznik zacznie od 1

Dlaczego właśnie .get(word, 0), a nie counts[word]

Na pierwszym spotkaniu słowa kluczowego `word` w `counts` jeszcze nie — `counts[word]` by spowodowało `KeyError`. `.get(word, 0)` zwraca 0 dla nowego słowa, a kod działa tak samo zarówno dla nowych, jak i istniejących słów.

Ten sam algorytm przez setdefault()

chastota_setdefault.py

```
counts = {}
for word in words:
    counts.setdefault(word, 0)
    counts[word] += 1
```

Trochę głębiej — standardowa biblioteka już rozwiązała ten problem

counter_preview.py

```
from collections import Counter

counts = Counter(words)
print(counts)              # Counter({'python': 2, 'is': 2,
'great': 1, 'and': 1, 'fun': 1})
print(counts.most_common(2))   # [('python', 2), ('is', 2)]
```

Algorytm ręczny jest ważniejszy niż gotowa funkcja

`Counter` z biblioteki standardowej robi to samo w jednej linii — a w prawdziwym kodzie często lepiej go używać. Ale algorytm na `.get(word, 0) + 1` warto było przejrzeć to rękami: pokazuje, jak liczenie działa w słownik ogólnie, a nie tylko że istnieje gotowe narzędzie do tego zadania.

Praktyka: Liczenie częstotliwości występowania słów w tekście
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/11-25/index.html)

Mini-projekty z kolekcjami

Cztery krótkie, ale realne zastosowania tego, czego się nauczyliśmy: Kontakty, Pole Gry, Klasa lista oraz Porównanie Zestawów.

Notatnik Kontaktowy (dict)

zapisnaya_knizhka.py

```
contacts = {
    "Anna": "anna@example.com",
    "Bob": "bob@example.com",
}

contacts["Maria"] = "maria@example.com"   # dodać
contacts["Anna"] = "anna.new@example.com"  # edycja
del contacts["Bob"]                          # usuń

for name, email in contacts.items():
    print(f"{name}: {email}")
```

Już mini-baza danych
Notatnik to w zasadzie mała baza danych w pamięci: słownik jako pamięć, klucz jako unikalny identyfikator dla rekordu.

Pole gry (zagnieżdżony lista)

igrovoe_pole.py

```
board = [
    [".", ".", "."],
    [".", "X", "."],
    [".", ".", "."],
]

for row in board:
    print(" ".join(row))
```

board[1][1] = 'X' to ten sam model macierzowy co w §11.11

Lista zajęć (lista słowników)

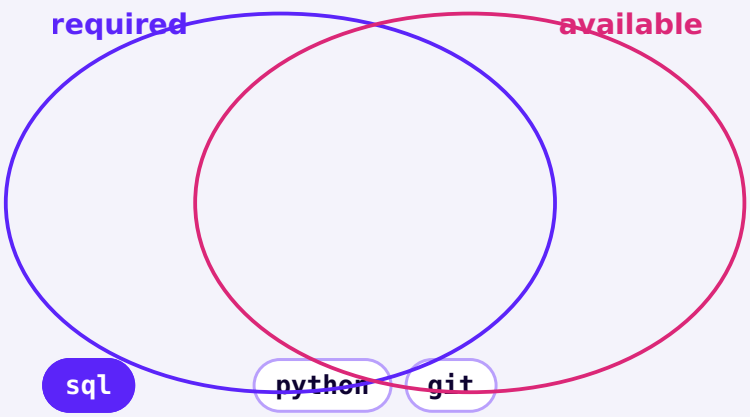
spisok_klassa.py

```
roster = [
    {"name": "Anna", "score": 95},
    {"name": "Bob", "score": 82},
    {"name": "Maria", "score": 91},
]

total = sum(student["score"] for student in roster)
average = total / len(roster)
print(f"Średnia ocena: {average:.1f}")
```

Porównanie zbiorów: czego brakuje

sravnenie_navykov.py

```
required = {"python", "git", "sql"}
available = {"python", "git"}

missing = required - available
common = required & available
print(„Zaginiony:”, missing)
print(„Już tam:”, common)
```



required - available - jakich umiejętności brakuje

Praktyka: mini-projekty z kolekcjami
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/11-26/index.html)

Mini-projekt – przekształcenie imienia i nazwiska

Zgromadzić split(), rozpakowania i wersy z rozdziału 8 w jednym krótkim, ale przydatnym programie — i podsumować cały rozdział.

Finalny mini-projekt rozdziału: podzielić pełne imię na części i złożyć ponownie w innej kolejności — „Nazwisko Imię” zamiast „Imię Nazwisko”, używając metod pracy ze stringami z rozdziału 8 i list z tego rozdziału.

perestanovka_imeni.py

```
full_name = input(„Wprowadź imię i nazwisko oddzielone
spację: ”)
parts = full_name.split()   #split() powraca lista
name, surname = parts        # rozpakowywanie jak krotki

print(f"{surname} {name}")
```

split() powraca lista
Widzieliśmy `.split()` w rozdziale 8, ale nie podkreślił, że wynik jest właśnie `list`. Teraz, znając listy, można go rozpakowywać tak samo, jak rozpakowywano krotki w §11.13.

Challenge
Trzy części nazwy
Rozwinąć sprawę z patronimikiem (trzy słowa): "Imię Nazwisko Nazwisko Nazwisko" → "Nazwisko pełne Imię" z inicjałami.

Praktyka: zamiana imienia i nazwiska
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji
Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/11-10/index.html)

Podsumowanie rozdziału

Zestaw podsumowujący rozdział 11

Czy potrzebujesz pary klucz→wartość? → **dict**

Potrzebne są tylko unikalne wartości, kolejność nie jest ważna? → **set**

Potrzebny jest porządek, wartości na pewno się nie zmienią?	→ `tuple`
Potrzebujesz zamówienia i będziesz musiał zmienić zawartość?	→ `list`
Potrzebujesz niezmiennego zestawu unikalnych wartości?	→ `frozenset`
Potrzebujesz pozycji i wartości razem, gdy używasz siły?	→ `enumerate(...)`
Czy musisz przeglądać kilka kolekcji parami?	→ `zip(...)`
Potrzebna nowa kolekcja przekształcona ze starej?	→ `comprehension`
Potrzebujesz niezależnej kopii zewnętrznej?	→ `.copy() / list(...) / [:]`
Potrzebna całkowicie niezależna kopia zagnieżdżonych struktur?	→ `copy.deepcopy(...) ostrożnie`

Pełną wersję tej Mapy Wyboru można znaleźć w §11.22

Czego nauczyliśmy się w tym rozdziale

- **Lista** (`list`) to uporządkowana, modyfikowalna kolekcja w `[]` : indeksy, wycinki, append/extend/insert, remove/pop/clear.
- **Poprowadzenie** (`tuple`) — jak lista, ale niezmienny, w `()` : packing/unpacking, rozpakowywanie gwiazdek, krotka jednego elementu wymaga przecinka.
- **Wielu** (`set`) — wartości unikalne bez pozycji, w `{}` (ale `{}` bez zawartości — to dict, a nie set!): union/intersection/difference/symmetric_difference.
- **Słownik** (`dict`) to para klucz:wartość, która zachowuje kolejność wstawiania, dostęp według klucza, a nie pozycji.
- `b = a` dla obiektów mutowalnych tworzy DRUGĄ NAZWĘ tego samego obiektu (aliasing), a nie kopię — dla kopii potrzebujesz `.copy()` , a dla struktur zagnieżdżonych `copy.deepcopy()` .
- Tylko niezmienne typy są haszalne (i użyteczne jako klucze słownikowe lub elementy zbioru): liczby, łańcuch znaków, krotki zahaszowanych elementów frozenset.
- Generator Listy `[выражение for элемент in коллекция]` — kompaktowa alternatywa dla pętli z `append()` , ale nie zastępuje pętli tam, gdzie potrzebna jest bardziej złożona logika.
- Wybór struktury danych to przede wszystkim pytanie o zadanie („czy potrzebny jest porządek?”, „czy potrzebne są unikalne wartości?”, „czy potrzebny jest klucz?”), a nie pytanie o składnię.

ROZDZIAŁ 12 · STR. 643

Mnóstwo fajnych mini-projektów!

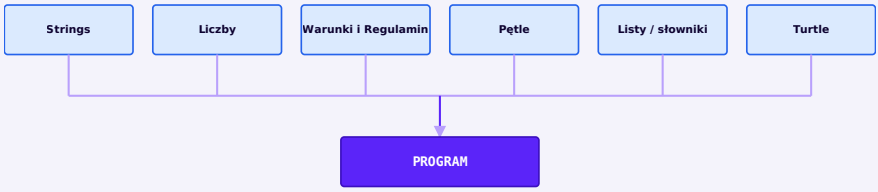
Pierwsze prawdziwe laboratorium projektowe kursu: 16 projektów, które zmuszają do współpracy wszystkiego, czego nauczyłeś się w rozdziałach 1-11 — warunki, pętle, łańcuch znaków, liczby, random, listy/krotki/zbiory/słowniki i Turtle. Nie nowa teoria, a sposób budowania programu z pomysłu: dekompozycja, algorytm do kodu, rozwój krokami, testy i debugowanie.

Czym jest projekt

Jak dotąd studiowaliśmy narzędzia indywidualnie. Teraz uczymy się, jak je połączyć w prawdziwy program, zaczynając od podejścia do wielkiego zadania w ogóle.

Od narzędzi do programów

Rozdziały 1-11 nauczył cię indywidualnie narzędzi: łańcuch znakówm, Liczby, Warunki, Pętle, Turtle, listy i słowniki. Ten rozdział nie dodaje żadnych nowych narzędzi — uczy **łączyć je razem**, aby powstał prawdziwy, działający program.



Rozdziały 1-11 — indywidualne narzędzia. Rozdział 12 — jak je połączyć w jeden program

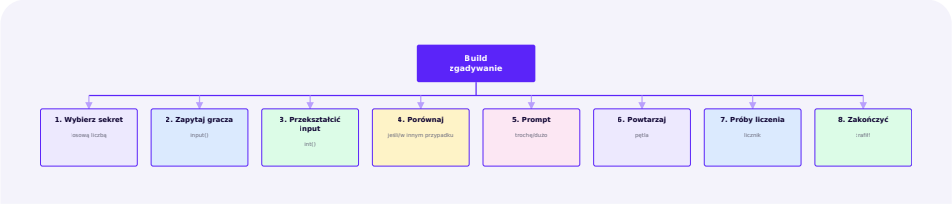
Jaka jest różnica między projektem a ćwiczeniem

	Praktyka	Projekt
Przykład	Obliczaj `7 % 3`	Zbuduj konwerter sekund na godziny/minuty/sekundy z walidacją danych wejściowych
Wkład	zazwyczaj nie	są, i trzeba je przemyśleć z wyprzedzeniem
Kroki	jeden	kilka ze sobą powiązanych
Rezultat	jedna wartość	sformalizowanym wnioskiem, reakcją na różne przypadki
Robaki	prawie niemożliwe	trzeba przemyśleć, co może pójść nie tak

Projekt to nie jest „większy kod”
Różnica nie polega na liczbie linii, ale na tym, że projekt posiada **cel**, **wkład**, **zasady** oraz **wynik** — i wszystko to musi zostać przemyślane, zanim zostanie napisany pierwszy łańcuch znaków kodu.

Dekompozycja: Rozkład dużego zadania na małe części

Zadanie „Zbuduj grę w zgadywanie” jest zbyt duże, aby rozwiązać je jednym kawałkiem kodu. Rozbijmy je na zrozumiałe kroki:



Rozkład: Jedno duże zadanie → ośmiu małych, zrozumiałych kroków

Rozkład

Rozkład to podzielić duże zadanie na małe, zrozumiałe części. To jedna z najważniejszych umiejętności programowania: znacznie łatwiej jest napisać (i sprawdzić) osiem małych kroków niż jeden złożony fragment kodu w stu linijkach.

Szablon projektu, którego użyjemy

Każdy duży projekt w tym rozdziale podąża tym samym schematem — niekoniecznie tymi z tymi samymi nagłówkami, ale w tym samym porządku myślenia:

Powtarzający się rytm każdego projektu rozdziału

1 · CO TWORZYMY

```
Cel projektu
Jak wygląda efekt końcowy
```

2 · DANE

```
Wkład
Jaka struktura je przechowuje
```

3 · ALGORYTM

```
Procedura
Schemat blokowy, jeśli jest to odpowiednie
```

4 · WDROŻENIE

Krok po kroku

Sprawdź po każdym etapie

5 · WERYFIKACJA

Typowe błędy

Debug Lab

Co można poprawić

Wymagania i „co oznacza 'zrobione'”

Przed napisaniem kodu warto jasno określić wymagania, jakie musi spełnić program by być uznanym za gotowego. Przykład przyszłej gry w zgadywanie:

Wymaganie	Weryfikacja
komputer wybiera liczbę całkowitą między 1 a 100	sekret — int w zakresie [1, 100]
gracz wpisuje liczby prób	input() + int()
program mówi "trochę" / "dużo"	if/elif/else
program kończy się poprawną odpowiedzią	pętla zatrzymuje się przy guess == secret
próby się liczą	counter wzrasta przy każdej próbie

Lista kontrolna odbioru

Ten lista nazywa się **kontrola przy odbiorze** — do kodu już wiemy, co dokładnie musi być poprawne, aby powiedzieć „projekt gotowy”. Nie trzeba formalnie go dokumentować za każdym razem — ale warto mieć to zawsze w głowie.

Rozgrzewka: statystyki trzech liczb

Zanim przejdziemy do dużych projektów, przećwiczmy sam proces — od zadania po kod — w programie wieloliniowym.

Zadanie: otrzymać trzy liczby i uzyskać minimalną, maksymalną ilość i drugorzędne.

statistika_treh_chisel.py

```
a = float(input(„Pierwszy numer: ”))
b = float(input(„Drugi numer: ”))
c = float(input(„Trzeci numer: ”))

chisla = [a, b, c]

print(f"Minimum: {min(chisla)}")
print(f"Maksimum: {max(chisla)}")
print(f"Suma: {sum(chisla)}")
print(f"Średniozaawansowany: {sum(chisla) / len(chisla):.2f}")
```

Nawet tak mały program to już projekt

Zawiera dane wejściowe (trzy liczby), strukturę do ich przechowywania (lista - Rozdział 11), regułę przetwarzania (`min/max/sum/len` — Rozdział 11) oraz sformatowany wynik (f-lines — Rozdział 8). Później w rozdziale — to samo, ale większe.

Praktyka: Statystyki trzech liczb — od zadania do kodu

interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

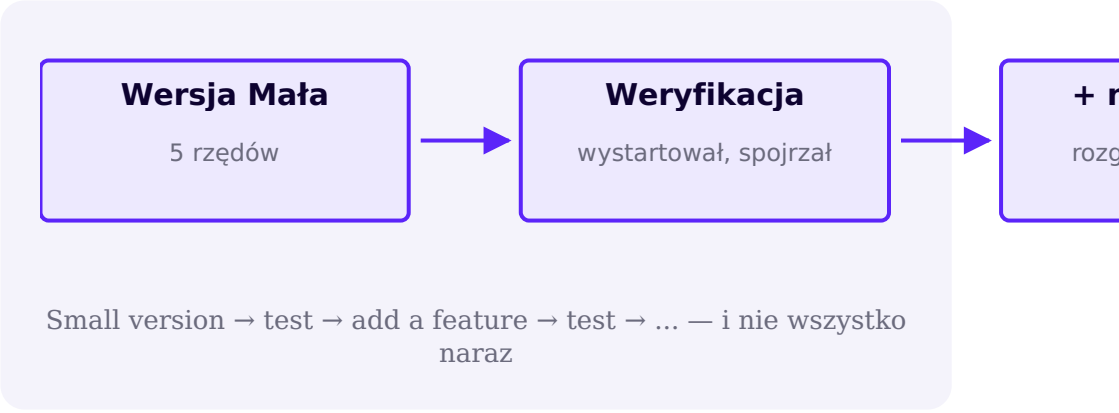
Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/12-07/index.html)

Budowanie projektu krok po kroku

Małymi, sprawdzalnymi krokami zamiast 80 linii naraz — i systemowy sposób znalezienia i naprawienia błędu, kiedy coś poszło nie tak.

Nie piszcie 80 linii, a potem wciśnijcie „Uruchom”

Profesjonalny nawyk, który powinien wypracować już od pierwszego dużego projektu: **stopniowy rozwój** małymi krokami, każdy z nich naraz jest sprawdzana.



Small version → test → add a feature → test → … — i nie wszystko naraz

Działająca mała wersja jest lepsza niż niedziałająca duża
Jeśli po każdym małym kroku program działa i robi to, czego się spodziewałeś, zawsze dokładnie wiesz, gdzie szukać przyczyny, jeśli coś się zepsuje w kolejnym kroku.

Wersje: najpierw działa, potem — lepiej

Wersja	Co w nim jest
V1	minimalna wersja działająca jest sednem pomysłu, bez dekoracji
V2	Bardziej niezawodna walidacja danych wejściowych, obsługa przypadków brzegowych
V3	dodatkowe funkcje, ulepszony output

Taką pierwszą działającą wersję czasami nazywa się **MVP** (minimal working version) brzmi: „najpierw osiągniemy małą, ale działającą wersję”

Testowanie projektu na przykładach

Dla każdego nietrywialnego projektu warto wcześniej przemyśleć kilka scenariuszy — co podać na wejście i co powinno wyjść:

Scenariusz	Loguj	Oczekiwany wynik
Secret 50, Mniej Prób	guess = 20	„za mało”
Secret 50, spróbuj więcej	guess = 70	„za dużo”
sekret 50, próba po-prawna	guess = 50	zwycięstwie cykl się kończy

Przypadki brzegowe

Zwykłe przykłady nie wystarczą — warto to sprawdzić wprost: pusty wpis, zero, liczba ujemna tam, gdzie nie jest oczekiwana, powtarzanie danych, pusty lista, nieznane polecenie, wartość dokładnie na granicy zakresu. Będziemy wracać do tego pomysłu w każdym głównym rozdziale roboczym.

Workflow debugowania

Oczekiwanie vs rzeczywistość → pierwsza rozbieżność → przyczyna → korekty

Krok po kroku:

Sześć Kroków Debugowania – Pracuj przy dowolnym projekcie w tym rozdziale

1

Odtwarzaj błąd
powtarzać to regularnie

2

Inspekcja wartości
print() właściwe zmienne

3

Znajdź pierwszy zły stan
gdzie oczekiwania różniły się od faktu

4

Izolować warunek/pętlę
który łańcuch znaków jest winny

5

Napraw jeden powód
nie przepisywać wszystkiego

```
6

Uciekaj jeszcze raz
upewnij się, że to naprawione
```

Debugowanie print()-

print_otladka.py

```
for word in words:
    print("DEBUG:", word, counts.get(word))   # Tymczasowa
diagnoza
    counts[word] = counts.get(word, 0) + 1
```

Usuń print() debugowania po znalezieniu przyczyny

Tymczasowe `print("DEBUG:", ...)` w pętli — najpewniejszy sposób, aby zobaczyć, co się naprawdę dzieje. Usuń takie linie, gdy błąd zostanie znaleziony i poprawiony — w przeciwnym razie zaśmiecają prawdziwe wyjście programu.

Debug Lab: znajdź błąd

W poniższym programie wynik powinien rosnąć przy każdej poprawnej odpowiedzi, ale z jakiegoś powodu zawsze Pozostałości `1`. Zanim czytasz dalej, spróbuj znaleźć powód siebie.

debug_lab_schet.py

```
answers = ["python", "git", "sql"]
user_answers = ["python", "python", "sql"]

for correct, given in zip(answers, user_answers):
    score = 0
    if given == correct:
        score += 1

print(„Wynik końcowy:”, score)
```

Przyczyna: Licznik resetuje się wewnątrz pętli

`score = 0` jest WEWNĄTRZ ciała pętli, oznacza, że jest odtwarzana przy każdej iteracji, a poprzedni postęp zostaje utracony. Licznik musi zostać zainicjowany RAZ, zanim pętla będzie tym samym błędem co „dlaczego cart total resetuje się przy każdej iteracji”

Lista kontrolna gotowości projektu

Zanim powiesz „projekt jest gotowy”

- Czy program spełnia wymagania (lista kontrolna przyjęć)?
- Czy normalne, „szczęśliwe”
- Sprawdzone przypadki graniczne (pusty, zero, powtórzenie, nieznana komenda)?
- Nazwy zmiennych są zrozumiałe bez dodatkowych wyjaśnień?
- Czy istnieje powtarzalny fragment kodu, który można by usunąć?
- Czy użyto odpowiedniej kolekcji (lista / zbiór / słownik)?
- Czy cykl naprawdę kończy się we właściwym momencie?
- Czy warunki sprawdzają dokładnie to, co trzeba?
- Wniosek jest jasny – czy użytkownik widzi, co się wydarzyło?
- Czy druga osoba będzie w stanie przeczytać i zrozumieć ten program?

Praktyka: Debug Lab - Znajdź i napraw błąd licznika
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/12-08/index.html)

Projekt: parzyste czy nieparzyste

Pierwszy szkic rozdziału ustala warunki i operator % z wcześniejszych rozdziałów — niewielki, ale z pełnym cyklem „algorytm → kodu → danych”

ZASTOSOWANIE

`input()` `int()` `%` `if/else` `list comprehension`

POZIOM

rozgrzewka

REZULTAT

program tekstowy

Co tworzymy

Pierwszy prawdziwy szkic rozdziału jest mały, ale z pełnym cyklem: dane → algorytm → kod → wynik.

Część 1 – Czy Twój numer jest parzysty czy nieparzysty?

chetnoe_ili_nechetnoe.py

```
number = int(input(„Wybierz numer: ”))

if number % 2 == 0:
    print(f"{number} jest równy.")
else:
    print(f"{number} – dziwne.")
```

```
Введите число: 17
17 — нечётное.
```

Przykład deterministycznego wykonania

Część 2 – Wyniki parzyste lub nieparzyste z zakresu

chetnye_iz_diapazona.py

```
nachalo = int(input(„Początek zakresu: ”))
konec = int(input(„Koniec zakresu: ”))

chetnye = [n for n in range(nachalo, konec + 1) if n % 2 == 0]
print(„Liczby parzyste:”, chetnye)
```

Jakie tematy tu się pojawiły
`input()` oraz `int()` — Rozdział 8; `%` — Rozdział 5; `if/else` — Rozdział 9; Generator list - Rozdział 11.

Praktyka: Parzyste lub nieparzyste
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/12-01/index.html)

Project: Czy twoja mama daje wystarczający napiwek?

Obliczaj procent napiwków z kwoty rachunku i oceń hojność w łańcuchu elif.

ZASTOSOWANIE

`input()` `float()` `арифметика` `if/elif/else` `f-строки`

POZIOM

rozgrzewka

REZULTAT

program tekstowy

Co tworzymy

Standardowy napiwek w restauracji to 15-20% rachunku. Sprawdźmy, czy konkretny Ilość napiwków w tym zakresie.

chaevye.py

```
schet = float(input(„Kwota faktury: ”))
chaevye = float(input(„Suma napiwków: ”))

procent = (chaevye / schet) * 100

if procent < 15:
    print(f"Za mało, po prostu {procent:.1f}%. Zazwyczaj
zostaje 15-20%.")
elif procent <= 20:
    print(f"W sam sam rys — {procent:.1f}%!")
else:
    print(f"Bardzo hojnie — całe {procent:.1f}%!")
```

```
Сумма счёта: 40
Сумма чаевых: 8
В самый раз — 20.0%!
```

Przykład deterministycznego wykonania

★★ Zadanie samodzielne
Własną granicę hojności
Dodaj czwartą kategorię — „fantasijnie hojny”

Praktyka: obliczamy i oceniamy procent napiwków
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji
Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/12-02/index.html)

Projekt: zgadnij liczbę, wersja 3

Pełny cykl rozwoju na znanej grze: wymagania → algorytm → schemat blokowy → tor stanu → kod.

ZASTOSOWANIE

`random` `while` `if/elif/else` `input()/int()` `счётчик`

POZIOM

★★ średni

REZULTAT

Gra tekstowa

Co tworzymy

Ostateczna, dopracowana wersja gry zgadywającej z rozdziałów 9-10 – z podsumowaniem prób i Fajny efekt.

```
Загадано число от 1 до 100. Попробуйте угадать!
Ваша попытка: 50
Слишком много!
Ваша попытка: 25
Слишком мало!
```

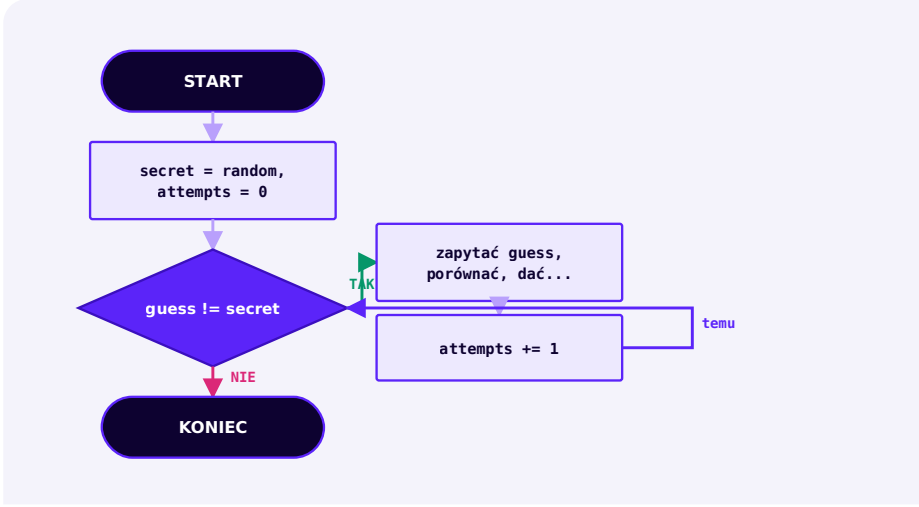
```
Ваша попытка: 37
Поздравляем! Загадано было 37. Попыток: 3
```

Przykład deterministycznego wykonania

Wymagania

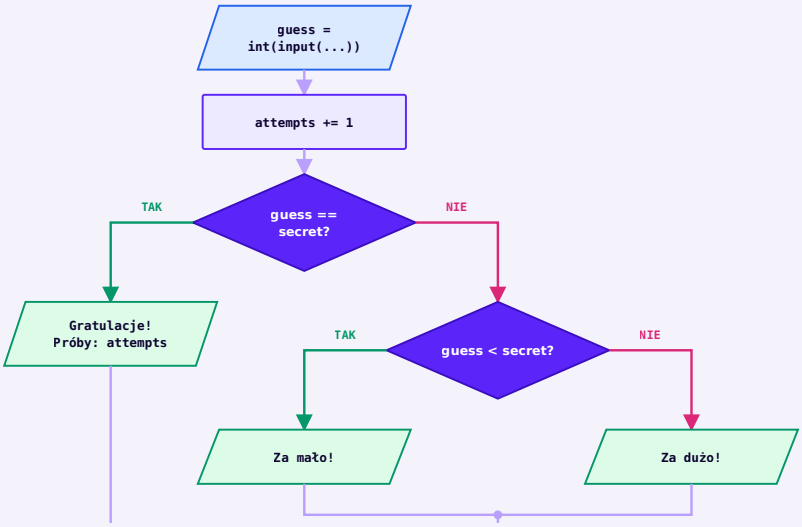
Wymaganie	Jak sprawdzić
komputer wybiera liczbę całkowitą między 1 a 100	`random.randint(1, 100)`
gracz próbuje, aż zgadnie	`while guess != secret`
po każdej próbie – podpowiedź niewiele/dużo	`if/elif/else`
próby się liczą	`attempts += 1`
gra kończy się poprawną odpowiedzią	cyklu kończy się wynik

Algorytm: Pętla zewnętrzna



Dopóki nie zgadniesz – poproś o próbę, porównaj, zasubstruuj, zwiększ licznik

Algorytm: co dzieje się podczas jednej próby



Jedna próba: porównać i podpowiedzieć — cały ten blok powtarza się w pętli while powyżej

Condition Track

Próba	guess	secret	Rezultat
1	50	37	za dużo
2	25	37	za mało
3	37	37	zgadywał, cykl się kończy

Ostateczny Kod

ugadaj_chislo_v3.py

```
import random

secret = random.randint(1, 100)
attempts = 0
guess = None

print(„Zgaduje się liczbę od 1 do 100. Spróbuj zgadnąć!”)

while guess != secret:
    guess = int(input(„Twoja próba: ”))
    attempts += 1
    if guess < secret:
        print(„Za mało!”)
    elif guess > secret:
        print(„Za dużo!”)
    else:
        print(f"Gratulacje! To było życzenie {secret}. Próby:
{attempts}")
```

W praktyce liczba jest stała, a nie losowa

Strona pokazuje prawdziwe `random.randint(1, 100)` — dokładnie tak powinna działać gra. W notatniku praktyki tajna liczba jest wcześniej ustalona, a wprowadzenie jest wstawiane automatycznie — w przeciwnym razie automatyczna weryfikacja wyniku byłaby niedeterministyczna.

Debug Lab

Co się stanie, jeśli sznurek `attempts = 0` przypadkowo przełożyłem wewnątrz cyklu `while`? Sprawdź zgadywanka przed czytaniem Następny.

attempts zawsze będzie 1
Dokładnie ten sam błąd, co w §12.2 Debug Lab: licznik, zerowany w obrębie pętli, nigdy nie kumuluje wartości — przy każdej iteracji jest tworzony od nowa.

Poziom 3: ograniczenie prób

Utrudnimy wymagania: gra kończy się porażką po 10 nieudanych próbach.

ugadaj_chislo_limit.py

```
max_attempts = 10

while guess != secret and attempts < max_attempts:
    ...

if guess == secret:
    print(f"Zwycięstwo dla {attempts} prób!")
else:
    print(f"Liczby się skończyły. Życzenie zostało złożone
{secret}.")
```

Praktyka: Zgadnij liczbę, wersja 3
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji
Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/12-09/index.html)

Praktyka (★★★): ograniczenie liczby prób
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji
Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/12-10/index.html)

Projekt: analizator tekstu i częstotliwość słów

Stringi, pętle, zbiory i słowniki — razem po raz pierwszy, w jednym programie, który mówi więcej o tekście, niż można zobaczyć na pierwszy rzut oka.

ZASTOSOWANIE

`strings` `циклы` `list` `set` `dict`

POZIOM

integracja

REZULTAT

raport tekstowy

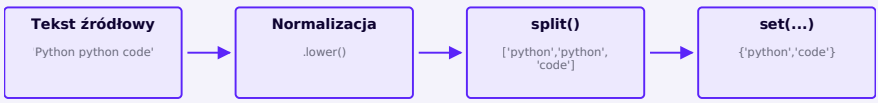
Co tworzymy

Program, który otrzymuje zdanie i liczy wszystko ciekawe: ile znaków, słów, unikatowych słów i które słowo pojawia się najczęściej.

```
Введите текст: Python is great and python is fun
Символов: 33
Слов: 7
Уникальных слов: 5
Самое частое слово: 'python' — 2 раз(а)
```

Przykład deterministycznego wykonania

Potok przetwarzania danych



Tekst → normalizacja → lista słów → zbiór unikalnych słów

Dlaczego set()
zbiory naturalnie eliminują powtarzalność (rozdział 11) – liczba unikalnych słów równa się po prostu `len(set(words))`, bez żadnej ręcznej kontroli.

Krok 1 – Podstawowe obliczenia

analizator_etap1.py

```
text = input(„Wprowadź tekst: ”)

words = text.split()

print(„Znaków:”, len(text))
print(„Słów:”, len(words))
```

Etap 2 — unikalne słowa

analizator_etap2.py

```
normalized = text.lower().split()
unikalnye = set(normalized)

print(„Unikalne słowa:”, len(unikalnye))
```

Bez . lower() 'Python' i 'python' to różne słowa
Struny są porównywane znak po znaku i na wielka litera (Rozdział 8) — `"Python" != "python"`. Jeśli zapomnieć `.lower()` zanim zaliczymy unikalne słowa, zostaną one zaliczone jako dwa różne słowa.

Etap 3 – Częstotliwość słów

Ten sam algorytm, który był szczegółowo omawiany w rozdziale 11 (§11.24) — teraz jako część większego programu:

chastota_slov.py

```
counts = {}
for word in normalized:
    counts[word] = counts.get(word, 0) + 1
```

word	stara wartość	nowa wartość
python	0 (get zwrócił 0)	1
is	0	1
great	0	1
and	0	1
python	1	2
is	1	2

Znajdź najczęstsze słowo

samoe_chastoe.py

```
samoe_chastoe = max(counts, key=counts.get)
print(f"Najczęściej używane słowo: {samoe_chastoe!r} —
{counts[samoe_chastoe]} raz(y)")
```

max() z key — nie tylko dla liczb

`max(counts, key=counts.get)` iteruje przez KLUCZE słownika (rozdział 11) i wybiera ten, dla którego `counts.get(ключ)` Maximum to kompaktowy sposób na znalezienie „najpopularniejszego”

Ostateczny Kod

analizator_teksta.py

```
text = input(„Wprowadź tekst: ”)
normalized = text.lower().split()

unikalnye = set(normalized)

counts = {}
for word in normalized:
    counts[word] = counts.get(word, 0) + 1
samoe_chastoe = max(counts, key=counts.get)

print(„Znaków:”, len(text))
print(„Słów:”, len(normalized))
print(„Unikalne słowa:”, len(unikalnye))
print(f"Najczęściej używane słowo: {samoe_chastoe!r} –
{counts[samoe_chastoe]} raz(y)")
```

Debug Lab

Jeśli się liczy. `len(unikalnye)` z słów BEZ `.lower()`, a częstotliwość słów jest już PO obniżeniu do niższej Rejestruj się, te dwie liczby mogą nie być ze sobą bi. Dlaczego?

Używaj wszędzie tego samego znormalizowanego lista

Jeśli `unikalnye` jest liczone z surowych słów, oraz `counts` — z `normalized`, wtedy "Python" i "python" będą się składać do `unikalnye` jako dwa różne słowa, a w `counts` — jak jedno. Zawsze licz od tej samej, już znormalizowanej listy.

Praktyka: Analizator tekstu – symbole, słowa, unikalne słowa
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/12-11/index.html)

Praktyka: częstość słów i najczęściej występujące słowo
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/12-12/index.html)

: Notes

Dictionary jako naturalne repozytorium „nazw → telefon”

ZASTOSOWANIE

`dict` `if/elif` `циклы` `f-строки`

POZIOM

★★ średni

REZULTAT

menu tekstowe

Co tworzymy

Kontakty w pamięci programu: pokaż wszystkie kontakty, dodaj, znajdź, Zmień, usuń.

Model danych

- `contacts (dict)`
  - `"Anna" → "+48 111 111 111"`
  - `"Bob" → "+48 222 222 222"`

Kluczem jest imię, wartość to telefon. Dlaczego dict? Bo szukamy kontaktu po imieniu, a nie po pozycji

Dlaczego słownik, a nie lista
Listę trzeba by przeszukać w całości w poszukiwaniu potrzebnej nazwy. Słownik od razu daje dostęp po kluczu (rozdział 11) — `contacts["Anna"]`, bez przesady.

Działania

zapisnaya_knizhka.py

```
contacts = {
    "Anna": "+48 111 111 111",
    "Bob": "+48 222 222 222",
}

# pokaż wszystkie kontakty
for name, phone in contacts.items():
    print(f"{name}: {phone}")

# dodaj kontakt
contacts["Maria"] = "+48 333 333 333"

# znajdź kontakt
zapros = "Anna"
if zapros in contacts:
    print(f"Znaleziono: {contacts[zapros]}")
else:
    print(„Kontakt nie znaleziony”)

# zmień kontakt
contacts["Anna"] = "+48 111 000 000"

# usuń kontakt
del contacts["Bob"]

print(„Kontakty zostawione:”, len(contacts))
```

Poziom 2: Interaktywne menu

To samo, ale sterowane poleceniami zamiast sztywno zapisanej sekwencji działań — zapowiedź konsoli poleceń z §12.10:

zapisnaya_knizhka_menu.py

```
contacts = {}

while True:
    command = input(„Drużyna (add/show/exit):”).strip().lower()
    if command == "exit":
        break
    elif command == "add":
        name = input(„Nazwa: ”)
        phone = input(„Telefon: ”)
        contacts[name] = phone
    elif command == "show":
        for name, phone in contacts.items():
            print(f"{name}: {phone}")
```

Gdy program się kończy, contacts zniknie

`contacts` istnieje tylko w pamięci, dopóki działa program — to zwykłe zmienna (rozdział 3). Aby dane były zachowane między uruchomieniami, będą potrzebne pliki — temat jednego z kolejnych rozdziałów.

Trochę głębiej – kontakt z kilkoma dziedzinami

Jeśli kontakt potrzebuje więcej niż jednej wartości (telefon I email), zewnętrzny słownik może przechowywać zagnieżdżone słowniki (rozdział 11, §11.19):

zapisnaya_knizhka_vlozhennaya.py

```
contacts = {
    "Anna": {"phone": "+48 111 111 111", "email":
"anna@example.com"},
}
print(contacts["Anna"]["email"])
```

Debug Lab

debug_lab_contacts.py

```
contacts = {"Anna": "+48 111 111 111"}
print("+48 111 111 111" in contacts)   # False!
```

in słownik sprawdza klucze, a nie wartości
Ta sama pułapka z rozdziału 11 (§11.19): `in` wyszukiwania w KLUCZACH. Aby sprawdzić, czy taki numer telefonu jest jednym ze znaczeń, musisz `"+48 111 111 111" in contacts.values()`.

Praktyka: notes — dodaj, znajdź, zmień, usuń
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/12-13/index.html)

Projekty: książka ocen i wózek na zakupy

Ten sam model danych — lista słowników — i ta sama metoda — akumulator, zastosowana w dwóch różnych zadaniach.

ZASTOSOWANIE

`list` `dict` `циклы` `накопитель`

POZIOM

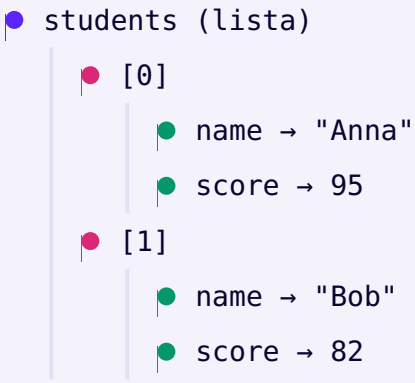
integracja

REZULTAT

raport / potwierdzenie

Dwa projekty na tej samej stronie celowo: oba mają ten sam model danych — " **lista słowniki** — i ten sam sposób — **przechowywanie** (Rozdział 10), stosowany tylko w inny sposób.

: Książka ocen



Lista słowników – dlaczego list? Ponieważ uczniowie tworzą uporządkowaną sekwencję rekordów

zhurnal_ocenok.py

```
students = [
    {"name": "Anna", "score": 95},
    {"name": "Bob", "score": 82},
    {"name": "Maria", "score": 91},
    {"name": "Leo", "score": 58},
]

for student in students:
    print(f"{student['name']}: {student['score']}")

scores = [student["score"] for student in students]
average = sum(scores) / len(scores)
print(f"Średnia ocena: {average:.1f}")
print(„Maximum:”, max(scores))
print(„Minimum:”, min(scores))

otlichniki = [s for s in students if s["score"] >= 90]
print(„Wybitni uczniowie:”, len(otlichniki))
```

Klasyfikacja przez if/elif

Jeśli potrzebujesz nie tylko filtrowania, ale oceny „doskonały/dobry/poprawka” `if/elif/else` z rozdziału 9 nie dotyczy prawdziwych kryteriów szkolnych, lecz trenowania algorytmu klasyfikacji.

Projekt: Wózek na zakupy

Każdy produkt podawa swoją własną ilość, akumulator sumuje je wszystkie razem

korzina_pokupok.py

```
cart = [
    {"name": „Mleko”, "price": 4.50, "qty": 2},
    {"name": „Chleb”, "price": 2.20, "qty": 1},
    {"name": „Cheese”, "price": 9.90, "qty": 1},
]

total = 0
for item in cart:
    line_total = item["price"] * item["qty"]
    total += line_total
    print(f"{item['name']:<10} {item['qty']} x
{item['price']:.2f} = {line_total:.2f}")

print(f"Łącznie: {total:.2f}")
```

```
Молоко     2 x 4.50 = 9.00
Хлеб       1 x 2.20 = 2.20
Сыр        1 x 9.90 = 9.90
Итого: 21.10
```

Check to przykład deterministycznego wykonania

To jest arytmetyka edukacyjna, nie Decimal

Do rzeczywistych obliczeń pieniężnych stosuje się bardziej precyzyjne typy (na przykład, `decimal.Decimal`), aby uniknąć nieścisłości float (Rozdział 5). Stosuje się tu zwykłą arytmetykę – nie zmienia to celu rozdziału 12: skupia się na strukturze programu, a nie na dokładności monetarnej.

Debug Lab

Dlaczego w tym kodzie `total` na końcu jest równe sumie tylko OSTATNI PRODUKT?

debug_lab_korzina.py

```
for item in cart:
    total = 0
    total += item["price"] * item["qty"]
```

Napęd jest zerowany wewnątrz pętli

Ten sam błąd, który już napotkano w §12.2 i §12.5: `total = 0` znajduje się wewnątrz ciała pętli, więc jest odtwarzany przy każdej iteracji. Napęd musi zostać zainicjowany RAZ, przed pętlą.

Praktyka: dziennik ocen — średnia, maksimum, minimum, najlepsi uczniowie

interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/12-14/index.html)

Praktyka: koszyk – paragon i całkowita kwota

interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/12-15/index.html)

Projekt: Quiz

Najważniejsza lekcja architektoniczna rozdziału: pytania to dane, a nie kod, i algorytm nie powinien się zmieniać, gdy zmieniają się dane.

ZASTOSOWANIE

`list` `dict` `for` `input` `if` `счётчик`

POZIOM

★★★★ challenge

REZULTAT

Quiz tekstowy

Co tworzymy

quiz składający się z kilku pytań: program zadaje pytanie, akceptuje odpowiedź, porównuje przy poprawnym zliczeniu liczy się punkt i pokazuje końcowy wynik.

```
Вопрос 1: Столица Франции?
Ваш ответ: paris
                                        Верно!
Вопрос 2: 7 * 8?
Ваш ответ: 54
```

```
                                    Неверно. Правильный ответ:

Счёт: 1 из 2
```

Przykład deterministycznego wykonania

Algorytm vs danych to główna lekcja tego projektu

Kusi, by napisać taki quiz:

Trzy pytania: logika zakodowana na stałe → dane + pętla

Pytania są „wbudowane”

```
question1 = „Stolica Francji?”
answer1 = "paris"
user1 = input(question1 + " ").strip().lower()
if user1 == answer1:
    score += 1

question2 = "7 * 8?"
answer2 = "56"
user2 = input(question2 + " ").strip().lower()
if user2 == answer2:
    score += 1

# ... I tak dalej przy każdym kolejnym pytaniu
```

Pytania – dane, algorytm jeden

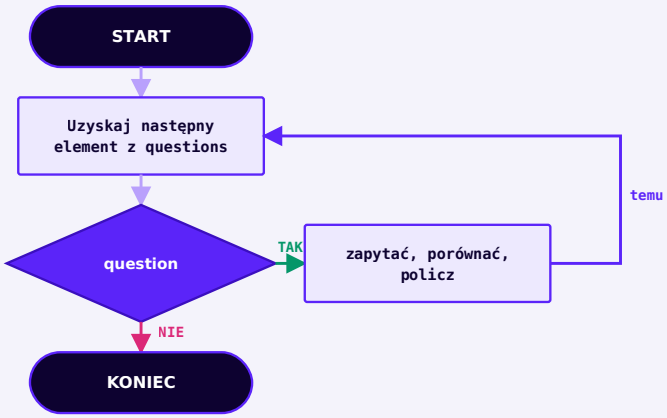
```
questions = [
    {"question": „Stolica Francji?”, "answer": "paris"},
    {"question": "7 * 8?", "answer": "56"},
]

score = 0
for q in questions:
    user_answer = input(q["question"] + " ").strip().lower()
    if user_answer == q["answer"]:
        score += 1
```

Co używać dzisiaj: przechowuje pytania jako DANE (lista słowników), a nie jako powtarzający się kod. Jeśli pytań będzie 20 zamiast 2, lewa wersja musiałaby być przepisana, a prawa — po prostu uzupełniona lista. **Algorytm się nie zmienia, gdy zmieniają się dane** — w tym rzecz *data-driven programming*.

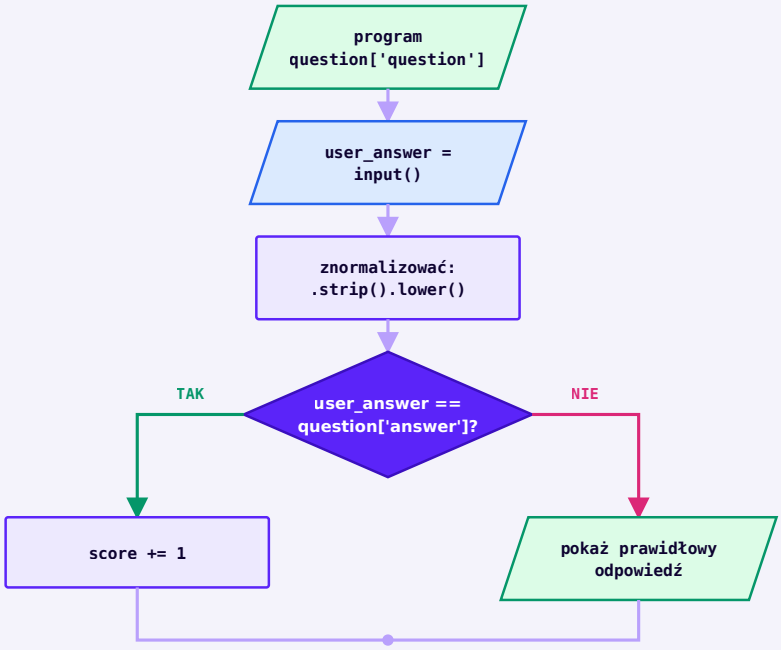
Refaktoryzacja

Przejście z lewej do prawej opcji nazywa się **refaktoryzacja** — poprawiamy wewnętrzną strukturę programu, nie zmieniając tego, co ma robić. Program z dwoma pytaniami zachowuje się tak samo w obu wersjach — prawa jest po prostu lepiej uporządkowana i łatwiej ją rozszerzać.

Algorytm



for question in questions — ten sam algorytm dla każdej liczby pytań



Jedna iteracja pętli – ten blok powtarza się przez for-loop od góry, dla każdego pytania

Ostateczny Kod

viktorina.py

```
questions = [
    {"question": „Stolica Francji?”, "answer": "paris"},
    {"question": "7 * 8?", "answer": "56"},
    {"question": „Języka, którego się uczymy?”, "answer":
"python"},
]

score = 0
for q in questions:
    user_answer = input(q["question"] + " ").strip().lower()
    if user_answer == q["answer"]:
        print(„   Poprawnie!”)
        score += 1
    else:
        print(f"   Błędnie. Poprawna odpowiedź: {q['answer']}")

print(f"Ocena: {score} z {len(questions)}")
```

Poziom 4: Wskaźnik poprawnych odpowiedzi

viktorina_procent.py

```
procent = score / len(questions) * 100
print(f"Wynik: {procent:.0f}%")
```

Debug Lab

Co się stanie, jeśli usuniesz `.strip().lower()` podczas normalizacji Odpowiedź?

„Paris” i „paris” — różne łańcuch znaków
Bez normalizacji przypadki i dodatkowe space sprawiają, że poprawna odpowiedź jest „nieprawidłowa”

Praktyka: Quiz z listy pytań
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/12-16/index.html)

Praktyka (★★★★): procent poprawnych odpowiedzi
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/12-17/index.html)

Projekty: Konsola poleceń i weryfikacja haseł

Dwa sposoby czytać dane wejściowe użytkownika w pętli: polecenie po poleceniu i znak po znaku, akumulując stan.

ZASTOSOWANIE

`while True` `if/elif/else` `break` `strings` `bool`

POZIOM

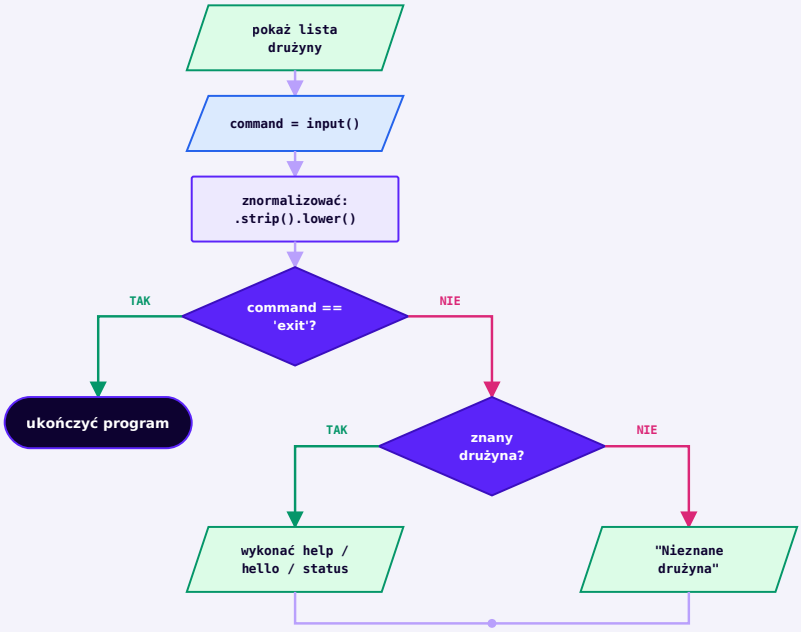
integracja

REZULTAT

Konsola tekstowa/Weryfikacja

Projekt: konsola poleceń

Prosta aplikacja tekstowa z poleceniami: `help`, `hello`, `status`, `exit`. Ten wzór — „niekończąca się pętla odczytu poleceń”



Do exit – przeczytaj polecenie, rozpoznaj je, wykonaj, powtórz

konsol_komand.py

```
history = []

while True:
    command = input("> ").strip().lower()
    history.append(command)

    if command == "exit":
        print(„Do zobaczenia!”)
        break
    elif command == "help":
        print(„Drużyny: help, hello, status, exit”)
    elif command == "hello":
        print(„Cześć!”)
    elif command == "status":
        print(f"Wykonanie komend: {len(history)}")
    else:
        print(f"Nieznane dowództwo: {command}")
```

```
> help
Команды: help, hello, status, exit
> hello
Привет!
> exit
До встречи!
```

Przykład deterministycznego wykonania

history jest regularnym lista

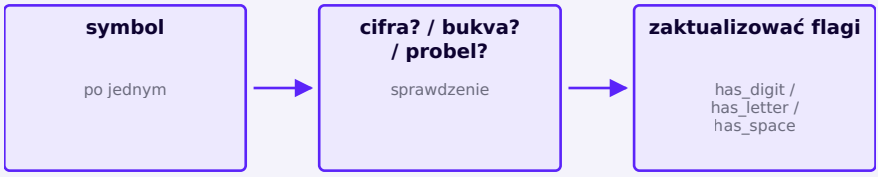
`history` gromadzi wszystkie polecenia w kolejności ich otrzymania, dokładnie dla czego lista został stworzony (Rozdział 11): uporządkowana, zmienna kolekcja.

Projekt: Weryfikacja nazwy użytkownika/hasła

To edukacyjny ćwiczenie po łańcuch znakówach, a nie ochrona danych

Poniżej przedstawiono praktykę pracy z flagami łańcuch znakówme i Boole'a, a NIE pełną polityką bezpieczeństwa haseł. Prawdziwa ochrona konta wymaga znacznie więcej (haszowanie, sól, menedżery haseł, limity prób) — na razie nie będziemy poruszać tego kursu.

Zasady (edukacyjne): co najmniej 8 znaków, jest co najmniej jedna liczba, co najmniej jedna litera, Nie ma żadnych luk.



Iteruj przez łańcuch znaków znak po znaku, gromadząc trzy flagi Boole'a

validator_parolya.py

```
password = input(„Wymyśl hasło: ”)

has_digit = False
has_letter = False
has_space = False

for ch in password:
    if ch.isdigit():
        has_digit = True
    elif ch.isalpha():
        has_letter = True
    elif ch == " ":
        has_space = True

dlinnyj_dostatochno = len(password) >= 8

if dlinnyj_dostatochno and has_digit and has_letter and not
```

```
has_space:
    print(„Hasło pasuje do zasad treningowych.”)
else:
    print(„Hasło nie pasuje:”)
    if not dlinnyj_dostatochno:
        print(„- za krótki (potrzeba 8+ znaków)”)
    if not has_digit:
        print(„- nie ma ani jednej liczby”)
    if not has_letter:
        print(„- nie ma ani jednej litery”)
    if has_space:
        print(„- zawiera spację”)
```

Gromadzenie bogactwa poprzez flagi boole'owskie

`has_digit`, `has_letter` oraz `has_space` są akumulatorami (Rozdział 10), tylko Booleany: każda postać może PRZEŁĄCZYĆ flagę `True`, i on pozostaje `True` do końca poszukiwań.

Debug Lab

Jeśli jest używany `elif` zamiast trzech oddzielnych `if` przy sprawdzaniu symbolu — niektóre symbole mogą być przetwarzane niepoprawnie. Na przykład, co jeśli reguła byłaby „jest znak specjalny”?

elif sprawdza tylko jedną gałąź na postać

Dopóki zasady są wzajemnie wykluczające się (liczba/litera/spacja – postać nie może być dwoma z nich jednocześnie), `elif` działa poprawnie. Ale jeśli dodasz czek na inną, NIEWYKLUCZAJĄCĄ się własność (na przykład „jest wielką literą” `if` zamiast gałęzi w tym samym łańcuchu `elif`.

Praktyka: Konsola poleceń z historią
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/12-18/index.html)

Praktyka: Weryfikacja hasła zgodnie z zasadami nauki
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/12-19/index.html)

Projekt: studio wielokątów

Ten sam wzór i ten sam cykl rysuje trójkąt, kwadrat, pięciokąt, sześciokąt lub ośmiokąt, w zależności od jednej liczby.

ZASTOSOWANIE

`Turtle` `input()/int()` `for` `формула`

POZIOM

integracja

REZULTAT

figura geometryczna

Co tworzymy

Program, który rysuje KAŻDY poprawny wielokąt — nie przez przepisanie kodu, lecz przez jego zmianę Jeden numer.

Wzór

Wielokąt regularny z `storony` imprezy zawsze skręca ten sam kąt między bokami:

formula_ugla.py

```
ugol = 360 / storony
```

storony	ugol
3	120°
4	90°
5	72°
6	60°
8	45°

Jeden algorytm — pięć wyników

studiya_mnogougolnikov.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=300, height=300)
artist = turtle.Turtle()
artist.speed(0)
artist.pencolor("#5B24F9")
artist.pensize(3)

storony = 5
dlina = 75
ugol = 360 / storony

for _ in range(storony):
    artist.forward(dlina)
    artist.right(ugol)

screen.exitonclick()
```

REZULTAT

Regularny Pentagon narysował Turtle

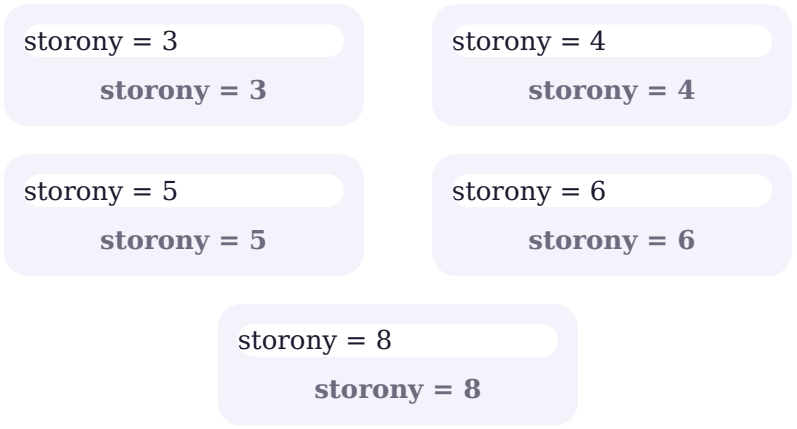
storony = 5 → regularnym Piątokątem

studiya_mnogougolnikov.py

```
storony = int(input(„Ile stron?”))
dlina = 300 / storony
ugol = 360 / storony

for _ in range(storony):
    artist.forward(dlina)
    artist.right(ugol)
```

Dane się zmieniają — algorytm nie

Ta sama koncepcja co w projekcie „Quiz” (§12.9): pętla `for _ in range(storony)` i formuła `ugol = 360 / storony` nie zmieniają się — zmienia się tylko wprowadzona liczba.



Ten sam kod, różna wartość storony — rzeczywiście wygenerowane wyniki

Debug Lab

Co się stanie, jeśli wejdziesz `storony = 1` lub `storony = 2`?

Wzór działa tylko dla storony >= 3
wielokąt z jedną lub dwiema bokami nie istnieje geometrycznie — kod nie zawiesi się z błędem (`360 / 1 = 360`), ale nie wyciągnie tego, czego się spodziewano. Pełna ochrona przed takim wpływem będzie wymagała weryfikacji `if storony < 3` przed rysowaniem.

Praktyka: Oblicz kąt obrotu dla wielokąta
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/12-20/index.html)

Praktyka: Narysuj wielokąt na wpisanej liczbie boków
Moduł turtle otwiera natywne okno Python – uruchamiane lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/12-21/index.html)

Projekt: choinka bożonarodzeniowa

Kilka trójkątnych warstw o coraz mniejszym rozmiarze, rysowanych w cyklu.

ZASTOSOWANIE

`Turtle` `for` `begin_fill/end_fill`

POZIOM

★★ średni

REZULTAT

Turtle graficznych

Co tworzymy

Narysuj choinkę z trójkątnych „poziomów”

elka.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=340, height=380)
artist = turtle.Turtle()
artist.speed(0)
artist.hideturtle()
artist.pencolor("green")
artist.fillcolor("green")

yarusy = 4
shirina = 120

artist.penup()
artist.goto(0, 100)
artist.pendown()

for yarus in range(yarusy):
    artist.begin_fill()
    artist.setheading(240)
    artist.forward(shirina)
    artist.setheading(0)
    artist.forward(shirina)
    artist.setheading(120)
    artist.forward(shirina)
    artist.end_fill()

    artist.penup()
    artist.setheading(270)
    artist.forward(30)
    artist.pendown()
    shirina -= 20

artist.pencolor("brown")
artist.fillcolor("brown")
artist.setheading(270)
artist.penup()
artist.forward(10)
artist.pendown()
artist.begin_fill()
for _ in range(2):
    artist.forward(40)
    artist.right(90)
```

```
    artist.forward(20)
    artist.right(90)
artist.end_fill()

screen.exitonclick()
```

REZULTAT

Choinka pomalowana Turtle o coraz mniejszych, trójkątnych piętrach

Cztery poziomy zmniejszających się rozmiarów + pień

Każdy poziom ma ten sam trójkąt co w rozdziale 6

Wzór na kąt obrotu (`360 / 3 = 120`) jest taki sam jak każdy regularny wielokąt z rozdziału 6 (oraz z Polygon Studio Project, §12.11). Choinka to po prostu seria trójkątów o malejących rozmiarach, narysowanych pod sobą w pętli.

Praktyka: Rysowanie choinki

Moduł turtle otwiera natywne okno Python – uruchamiane lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/12-03/index.html)

Projekt: spirale!

Ta sama zasada — pięć różnych wzorów, w zależności od kąta obrotu.

ZASTOSOWANIE

`Turtle` `for` `random (одна из версий)`

POZIOM

integracja

REZULTAT

grafika Turtle · sztuka generatywna

Co tworzymy

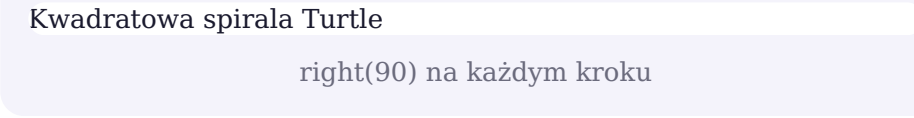
Pięć wariantów tego samego pomysłu – figurka, której każdy krok jest nieco większy od poprzedniego. Jeden Zasada z rozdziału 10 („Krok + obrót + zwiększaj zmienną”

Kwadratowa Spirala

kvadratnaya_spiral.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=480, height=480)
artist = turtle.Turtle()
artist.speed(0)
artist.pencolor("#5B24F9")
artist.pensize(2)

dlina = 5
for _ in range(60):
    artist.forward(dlina)
    artist.right(90)
    dlina += 3

screen.exitonclick()
```

REZULTAT



Losowa spirala

sluchaynaya_spiral.py

```
import turtle
import random

random.seed(4)
screen = turtle.Screen()
screen.setup(width=480, height=480)
artist = turtle.Turtle()
artist.speed(0)
artist.pencolor("#DB2777")
artist.pensize(2)

dlina = 5
for _ in range(60):
    artist.forward(dlina)
    artist.right(random.randint(80, 100))
    dlina += 3

screen.exitonclick()
```

REZULTAT

Spirala z okazjonalnym drgającym kątem

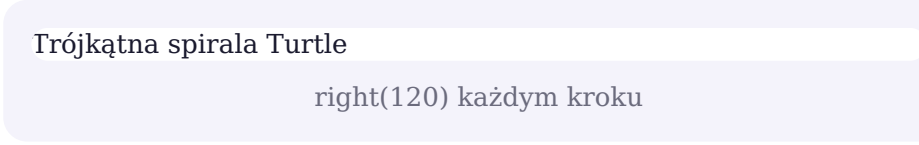
Kąt „drży”

Trójkątna spirala

treugolnaya_spiral.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=480, height=480)
artist = turtle.Turtle()
artist.speed(0)
artist.pencolor("#059669")
artist.pensize(2)

dlina = 5
for _ in range(60):
    artist.forward(dlina)
    artist.right(120)
    dlina += 3

screen.exitonclick()
```

REZULTAT



Star Spiral

zvezdnaya_spiral.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=480, height=480)
artist = turtle.Turtle()
artist.speed(0)
artist.pencolor("#5B24F9")
artist.pensize(2)

dlina = 5
for _ in range(100):
    artist.forward(dlina)
    artist.right(144)
    dlina += 2

screen.exitonclick()
```

REZULTAT

Spirala Gwiezdna Turtle

right(144) jest kątem gwiazdy pięcioramiennej

Spirala Okrągła

krugovaya_spiral.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=480, height=480)
artist = turtle.Turtle()
artist.speed(0)
artist.pencolor("#DB2777")
artist.pensize(2)

radius = 5
for _ in range(60):
    artist.circle(radius, 90)
    radius += 3

screen.exitonclick()
```

REZULTAT

Spirala łuków kołowych Turtle

circle(radius, 90) o rosnącym promieniu

Jedna ogólna zasada

Wszystkie pięć spiral — ta sama idea z rozdziału 10 („krok + obrót + zwiększenie zmiennej") z różnym kątem obrotu. Opanowując jedną wersję, faktycznie opanowałeś wszystkie pięć.

Praktyka: Spirale

Moduł turtle otwiera natywne okno Python – uruchamiane lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/12-04/index.html)

: Złożona mandala – w pełni zautomatyzowana

Ostateczna wersja mandali z rozdziałów 6 i 10 — z losowymi kolorami i kółkami na końcach promieni.

ZASTOSOWANIE

`Turtle` `for` `random.choice`

POZIOM

★★★★ challenge

REZULTAT

grafika Turtle · sztuka generatywna

Co tworzymy

Ostateczna ewolucja mandali z rozdziałów 6 i 10 – dodaj losowy kolor do każdej wiązki i Zrobimy to w pełni automatycznym, bez jednej „magicznej liczby”

Jak zbudowany jest wzór: od jednego promienia do kompletnej mandali

Ta sama zasada konstrukcji krok po kroku co w rozdziale 10: najpierw jeden element, potem Kilka, potem pełny wzór.

odin_luch.py

```
artist.circle(20)
artist.forward(150)
artist.forward(-150)
```

slozhnaya_mandala.py

```
import turtle
import random

random.seed(11)
screen = turtle.Screen()
screen.setup(width=420, height=420)
artist = turtle.Turtle()
artist.speed(0)

luchi = 36
shag_ugla = 360 / luchi
cveta = ["red", "orange", "purple", "blue", "green"]

for i in range(luchi):
    artist.pencolor(random.choice(cveta))
    artist.setheading(i * shag_ugla)
    artist.forward(150)
    artist.circle(20)
    artist.forward(-150)

screen.exitonclick()
```

REZULTAT

36 promieni, losowy kolor na każdym (seed poprawione dla dokumentacji)

slozhnaya_mandala.py

```
import random

luchi = 36
shag_ugla = 360 / luchi
cveta = ["red", "orange", "purple", "blue", "green"]

for i in range(luchi):
    artist.pencolor(random.choice(cveta))
    artist.setheading(i * shag_ugla)
    artist.forward(150)
    artist.circle(20)
    artist.forward(-150)
```

luchi decyduje o wszystkim innym

Tylko zmiana `luchi` to kąt kroku, liczba powtórzeń cyklu, a nawet gęstość wzoru zostaną automatycznie przeliczone, nic więcej nie wymaga zmian. To nazywa się „w pełni zautomatyzowanym”

Praktyka: Złożona mandala z przypadkowymi kwiatami

Moduł turtle otwiera natywne okno Python – uruchamiane lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/12-05/index.html)

Projekt: Grafika pikselowa na siatce

algorytm → danych → graf: zagnieżdżony lista decyduje, co rysować, a zagnieżdżona pętla decyduje jak.

ZASTOSOWANIE

`nested list` `nested for` `Turtle`

POZIOM

★★★★ challenge

REZULTAT

pixel art

Co tworzymy

Most między danymi a grafiką: macierz z zer i jedynek — oraz program, który rysuje według niej obraz. `0` — pusto, `1` — wypełniony kwadrat.

	0	1	2	3	4
0	0	1	0	1	0
1	1	1	1	1	1
2	1	1	1	1	1
3	0	1	1	1	0
4	0	0	1	0	0

picture jest zagnieżdżony lista: 0 = pusty, 1 = wypełniony kwadrat

setka_piksel_art.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=360, height=360)
artist = turtle.Turtle()
artist.speed(0)
artist.hideturtle()
artist.penup()

picture = [
    [0, 1, 0, 1, 0],
    [1, 1, 1, 1, 1],
    [1, 1, 1, 1, 1],
    [0, 1, 1, 1, 0],
    [0, 0, 1, 0, 0],
]

razmer = 40
for row_index, row in enumerate(picture):
    for col_index, value in enumerate(row):
        if value == 1:
            x = -100 + col_index * razmer
            y = 100 - row_index * razmer
            artist.goto(x, y)
            artist.pendown()
            artist.fillcolor("#DB2777")
            artist.begin_fill()
            for _ in range(4):
                artist.forward(razmer)
                artist.right(90)
            artist.end_fill()
            artist.penup()

screen.exitonclick()
```

REZULTAT

Serce z wypełnionych kwadratów, narysowane według macierzy zer i jedynek

Ta sama macierz rysowana przez zagnieżdżoną pętlę

Algorytm

setka_piksel_art.py

```
picture = [
    [0, 1, 0, 1, 0],
    [1, 1, 1, 1, 1],
    [1, 1, 1, 1, 1],
    [0, 1, 1, 1, 0],
    [0, 0, 1, 0, 0],
]

razmer = 40
for row_index, row in enumerate(picture):
    for col_index, value in enumerate(row):
        if value == 1:
            x = -100 + col_index * razmer
            y = 100 - row_index * razmer
            artist.goto(x, y)
            artist.pendown()
            artist.begin_fill()
            for _ in range(4):
                artist.forward(razmer)
                artist.right(90)
            artist.end_fill()
            artist.penup()
```

Zagnieżdżona pętla – łańcuch znaków wiersz, kolumna po kolumnie

Zewnętrzna pętla (rozdział 10) przechodzi po łańcuch znaków macierzy, wewnętrzna — po wartościach w wierszu; ta sama schematyczna jak w macierzy z rozdziału 11 (§11.11), tylko teraz dla każdej jednostki rysowany jest kwadrat, a nie tylko drukowana liczba.

Challenge

Zmiana JEDNA: Twoje zdjęcie

Zastąp picture własną macierzą zer i jedynek o rozmiarze 5×5 (lub innym rozmiarze) – litery, emotikony, dowolny wzór. Algorytm nie wymaga zmian.

Debug Lab

Co się stanie, jeśli pomieszasz miejsca w wzorze współrzędnych `row_index` oraz `col_index`?

debug_lab_grid.py

```
x = -100 + row_index * razmer   # było col_index
y = 100 - col_index * razmer    # było row_index
```

Obraz okaże się odbity lustrzanie (transponowany)

Wiersze i kolumny zamieniają się — tam, gdzie powinna być szeroka pozioma część serca, pojawi się wydłużona pionowa część. Wzory współrzędnych muszą być bardzo starannie sprawdzane względem znaczenia każdego indeksu.

Praktyka: pixel art w swojej matrycy

Moduł turtle otwiera natywne okno Python – uruchamiane lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/12-22/index.html)

Projekt: wyścig Turtle z użyciem pętli

Kilka żółwi na jednym ekranie jednocześnie — i podsumowujące wyniki całego laboratorium projektowego.

ZASTOSOWANIE

`Turtle` `list` `while` `random`

POZIOM

★★★★ challenge

REZULTAT

Turtle graficznych

Co tworzymy

Pamiętaj z rozdziału 6, że ekran i żółw to różne obiekty, a żółwie mogą być kilka? Czas go użyć: rasa kilku żółwi z losowym krok.

gonka_turtle.py

```
import turtle
import random

random.seed(2)
screen = turtle.Screen()
screen.setup(500, 400)

cveta = ["red", "blue", "green", "orange"]
uchastniki = []

for i, cvet in enumerate(cveta):
    t = turtle.Turtle()
    t.shape("turtle")
    t.color(cvet)
    t.penup()
    t.goto(-200, i * 40 - 60)
    uchastniki.append(t)

finish_line = 200
pobeditel = None

while pobeditel is None:
    for t in uchastniki:
        t.forward(random.randint(1, 10))
        if t.xcor() >= finish_line:
            pobeditel = t.pencolor()
            break

screen.exitonclick()
```

REZULTAT

Na końcu wyścigu Turtle cztery kolorowe żółwie na różnych pozycjach

Cztery Żółwie na końcu wyścigu (seed zarejestrowane do dokumentacji)

gonka_turtle.py

```
import random

screen = turtle.Screen()
screen.setup(500, 400)

cveta = ["red", "blue", "green", "orange"]
uchastniki = []

for i, cvet in enumerate(cveta):
    t = turtle.Turtle()
    t.shape("turtle")
    t.color(cvet)
    t.penup()
    t.goto(-200, i * 40 - 60)
    uchastniki.append(t)

finish_line = 200
pobeditel = None

while pobeditel is None:
    for t in uchastniki:
        t.forward(random.randint(1, 10))
        if t.xcor() >= finish_line:
            pobeditel = t.pencolor()
            break

print(f"Żółw kolorowy wygrał {pobeditel}!")
```

Lista żółwi — ten sam lista z rozdziału 11

`uchastniki` jest zwykłym lista Python (Rozdział 11), który po prostu przechowuje obiekty zamiast liczb `turtle.Turtle()`. Pętla `for t in uchastniki` iteruje przez to w taki sam sposób, jak przechodziłby przez lista liczby czy łańcuch znaków.

Challenge

Taśma wykończeniowa

Rysuj pionową linię na mecie (x=200) przed startem wyścigu, używając osobnego „sędziego”

Praktyka: wyścig Turtle
Moduł turtle otwiera natywne okno Python – uruchamiane lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/12-06/index.html)

Podsumowanie rozdziału

Co teraz umiemy budować

- Co teraz umiemy budować
  - Programy tekstowe
    - Analizator tekstu i częstotliwość słów
    - Quiz
    - Konsola Komend
    - Notes
    - Zgadnij liczbę, wersja 3
  - Programy z danymi
    - Książka ocen
    - Koszyk sklepowy
    - Weryfikacja hasła
  - Grafika
    - Studio wielokątów
    - Grafika pikselowa w siatce
    - Spirale i złożona mandala
    - Choinka
    - Race Turtle

16 projektów, wszystkie działające na tych samych narzędziach w rozdziałach 1-11

Wszystko to jest napędzane tymi samymi narzędziami, co rozdziały 1-11

STRINGS

`input(), split(), .lower()`

`formatowanie za pomocą f-stringów`

LICZBY

`arytmetyka, zaokrąglanie`

`random`

WARUNKI I REGULAMIN

`if / elif / else`

`Flagi Boole'a`

PĘTLE

`for, while, break`

`liczniki, magazyny`

KOLEKCJE

`list, dict, set`

```
lista słowniki
```

TURTLE

```
formuły toczenia
zagnieżdżonych pętli → grafów
```

Jak daleko zaszliśmy

Rozdział 1	Rozdział 12
`print("Hello")`	interaktywna gra, analizator tekstu, quiz, notatnik, dane strukturalne, generator graficzny

Lista kontrolna gotowości projektu

Pełna lista kontrolna znajduje się w §12.2 „Budowanie projektu krok po kroku”

Co dalej: Dlaczego funkcje są potrzebne

Przyjrzyj się bliżej projektom w tym rozdziale: Normalizacja wejścia (`.strip().lower()`) powtarza się w analizatorze tekstu, w konsole zespołu i w quizie. Napęd, który trzeba uruchomić przed cyklem, jest taki sam błąd był korygowany trzy razy w różnych Debug Lab. Wzór na kąt obrotu pojawia się Turtle w Choinka, w mandali i w pracowni wielokątów.

Wkrótce to ułatwimy

Jeśli potrzebujesz tego samego bloku kodu w więcej niż jednym miejscu, musisz go skopiować. W następnym rozdziale, **«Automatyzacja za pomocą funkcji»** nauczymy się, jak nazywać grupę poleceń i używać jej wielokrotnie, bez konieczności ponownego kopiowania kodu. Nic z rzeczy z rozdziałów 1-12 nie będzie przestarzałe — funkcje dadzą nam jedynie ładniejszy sposób na organizację tego, co już potrafimy napisać.

Co skonsolidowaliśmy w tym rozdziale

- **Rozkład** to rozbicie dużego zadania na małe, zrozumiałe kroki.
- **Rozwój stopniowy** — w małych krokach każdy jest sprawdzany jednocześnie, a nie 80 linijek naraz.
- **Dane oddzielne od algorytmu** — quiz i studio polygon pokazały, że ta sama logika działa dla wszystkich danych wejściowych.
- **Refaktoryzacja** — ulepszenie struktury programu bez zmiany jego działania.
- Warunki, pętle, łańcuch znaków, liczby, `random`, listy/zbiory/słowniki i Turtle — pracowały razem po raz pierwszy, w jednym programie, a nie osobno.
- Systemowy workflow debugowania: oczekiwanie vs rzeczywistość → pierwsza rozbieżność → przyczyna → poprawka.
- Złożone wyniki (quiz, analizator tekstu, mandala) — zawsze kombinacja kilku prostych, już znanych technik.

ROZDZIAŁ 13 · STR. 709

Automatyzacja przy użyciu funkcji

Funkcje · Dekompozycja · Parametry · return · zakresy. Nauczymy się, jak przekształcać duże programy w zrozumiałe części: nadać algorytmom nazwy, przekazywać im dane, uzyskiwać wyniki, zarządzać zakresem i ponownie używać kodu bez kopiowania — pomysł, który można przełożyć na każdy inny język programowania.

Dlaczego program potrzebuje funkcji

Pętle automatyzują powtarzanie w wątku wykonywania. Funkcje automatyzują ponowne wykorzystanie zachowań — i to nie to samo.

Problem, którego pętle nie rozwiązują

W rozdziale 12 nasze projekty stawały się coraz większe, a zaczęły się powtarzać Logiczne blokady. Na przykład w quizie ten sam zestaw działań to „zadaj pytanie, normalizuj odpowiedź, porównaj, obliczaj” **podczas gdy powtarzanie następuje jedno z rzędu, to w jednym Lokalizacja**.

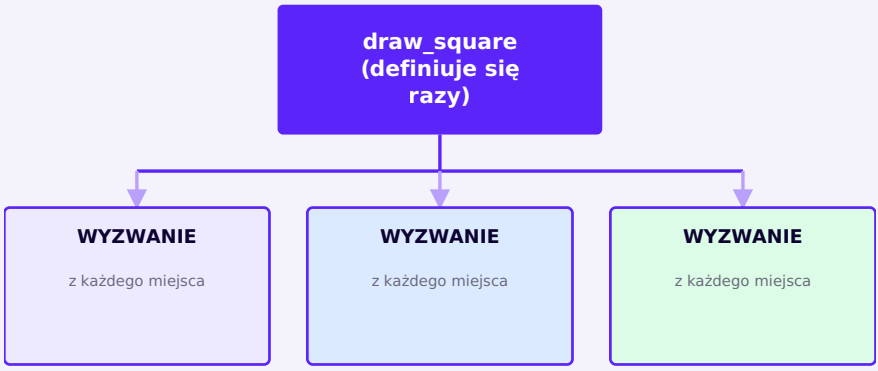
Ale co jeśli potrzebne są te same działania:

- w **różnych lokalizacji** programów — nie kolejno;
- z **różne dane** za każdym razem;
- z **różnych projektach** w ogóle;
- **na żądanie**, a nie według harmonogramu cyklu?

Cykl tutaj nie pomoże — on potrafi powtarzać tylko to, co znajduje się bezpośrednio w jego ciele. Potrzebne jest inne narzędzie: metoda **nadanie tej akcji nazwy** i powoduje to Z każdego miejsca.

Funkcja vs cyklu

Loop: Jeden wątek powtarzalnego wykonania



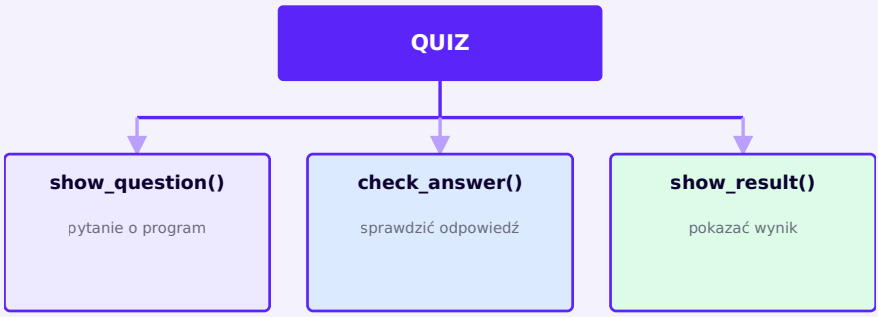
Funkcja: Definiowana raz, wywoływana na żądanie

Nie alternatywy, lecz wzajemne uzupełnianie się

Pętle automatyzują powtarzanie się w przepływie wykonania. Funkcje automatyzują ponowne wykorzystanie zachowań. Wkrótce zobaczymy funkcję o nazwie INSIDE the loop — działają one idealnie razem, a nie zamiast siebie.

Funkcje są rozkładane

W rozdziale 12 już rozbiliśmy wielki problem na małe kroki — tak się to nazywało **rozkład**. Funkcja to sposób nie tylko na mentalne wybranie kroku, ale Dosłownie nadaj temu nazwę w kodzie:

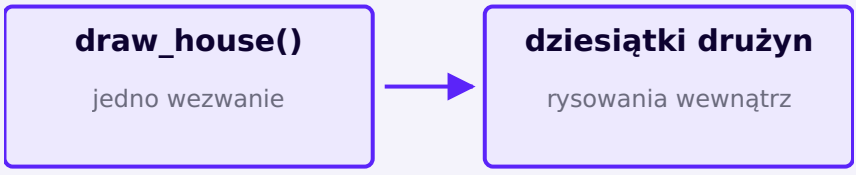


Duże zadanie → części zrozumiałe nazwami — to jest dekompozycja w kodzie

Każda funkcja — to znacząca nazwa dla jednej części większego algorytmu.

Abstrakcja: przydatny poziom szczegółowości

Już używasz `print(...)` bez zastanowienia się, jak dokładnie Tekst pojawia się na ekranie. Podobnie wyzwanie `draw_house()` mógł ukryć dziesiątki komend do narysowania dachu, ścian i okien:



Rozmowa ukrywa szczegóły wdrożenia za przyjazną nazwą

Abstrakcja nie polega na „ukryciu wszystkiego”

Abstrakcja oznacza, że możemy używać znaczącej operacji bez konieczności myślenia o każdym szczegółzie jej wdrożenia za każdym razem. To nie znaczy, że szczegóły nie są dostępne lub nieistotne — po prostu w większości zadań wystarczy wiedzieć, CO `draw_house()`, nie JAK dokładnie.

Już wywoływałeś funkcje — od samego pierwszego rozdziału

`print()`, `input()`, `len()`, `type()`, `int()`, `range()`, `sorted()`, `min()`, `max()` — wszystkie one to po prostu funkcje **wbudowane** w Python. Przez cały ten czas je PRZYWOŁYWAŁEŚ. Teraz dowiedzmy się, jak je TWORZYĆ:

Funkcja wbudowana	Twoja przyszła funkcja
`len("Python")`	`calculate_area(10, 5)`

Ogólny model jest taki sam dla obu:

Nazwa funkcji + argumenty → wywołanie → wynik lub efekt – zarówno dla wbudowanych funkcji, jak i dla twoich własnych

Praktyka: Pętla lub funkcja – co rozwiązuje problem
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/13-09/index.html)

Prawdziwa automatyzacja

Funkcje wykorzystują zestaw działań tyle razy, ile to konieczne, bez kopiowania kodu.

Prawdziwa automatyzacja

Pętle w rozdziale 10 automatyzowały powtarzanie tego samego kodu. Ale co jeśli jeden i To samo *zestaw działań* jest potrzebny w różnych miejscach programu — nie po kolei, lecz od czasu do czasu czasu? Kopiowanie kodu za każdym razem to zły pomysł: każda poprawka będzie musiała zostać wprowadzona we wszystkich kopiach. **Cechy** rozwiązać ten problem raz na zawsze.

Nasz pierwszy pełnometrażowy film

pervaya_funkciya.py

```
def privetstvie():
    print(„Cześć, Python!”)

privetstvie()
privetstvie()
privetstvie()
```

Anatomia def

Część	Znaczenie
`def`	słowo kluczowe: „dalej następuje definicja funkcji”
`privetstvie`	nazwa funkcji — po niej funkcja będzie wywoływana
`()`	lista parametrów (tutaj pusto — funkcja nie przyjmuje danych)
`:`	początek ciała funkcji
`print(...)`	ciało — kod z wcięciem, który wykona się przy wywołaniu

`def` **determinuje** funkcję — zapisuje jej kod, ale jeszcze go nie wykonuje. Funkcja wykonuje się dopiero wtedy, gdy ją **wywołujesz** — z imienia i nazwiska w nawiasach: `privetstvie()`.

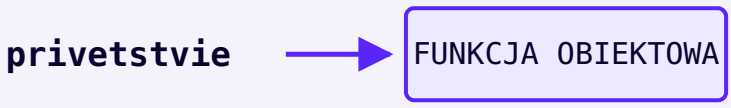
> **Definicja nie jest wyzwaniem**
> Jeśli tylko piszesz `def privetstvie(): ...` nie dodawać `privetstvie()` poniżej, program nic nie wyświetla – Python tylko zapamięta funkcję, ale jej nie uruchomi.

Determinacja jest wykonywana raz, w szczególny sposób

Gdy Python osiąga granicę `def privetstvie():`, ciało Cechy **niespełnione** jak zwykłe sekwencyjne polecenia. Zamiast tego tworzony jest obiekt funkcji powiązany z nazwą `privetstvie` — i wykonanie programu kontynuuje BEZPOŚREDNIO PO definicji. Ciało uruchomi się dopiero później, gdy nastąpi wywołanie.

Funkcja jako obiekt

Znamy już model „imię wskazuje na obiekt”



Po def nazwa privetstvie wskazuje na obiekt funkcji

imya_bez_skobok.py

```
print(privetstvie)   # <function privetstvie at 0x...> — sam obiekt funkcji
privetstvie()          # Witaj Python! — a to jest WYZWANIE
```

privetstvie i privetstvie() to nie to samo
Bez nawiasów — odwołanie do samego obiektu funkcji (można go przekazać, zapisać, porównać). Z nawiasami — polecenie „wykonać ją teraz”. To rozróżnienie stanie się szczególnie ważne w §13.17, gdy funkcje zaczną być przekazywane jako zwykłe wartości.

Praktyka: Zdefiniuj i wywołaj pierwszą funkcję
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/13-01/index.html)

Co się dzieje podczas wywołania

Wywołanie funkcji to przejście sterowania do ciała funkcji i obowiązkowy powrót do miejsca wywołania, a nie „wykonanie gdzieś na boku”.

Wyzwanie to nie „robienie czegoś na uboczu”

Przyjrzyjmy się krok po kroku, co naprawdę się dzieje, gdy program jest `greet()`:

poryadok_vypolneniya.py

```
def greet():
    print(„Cześć”)

print(„Before”)
greet()
print(„Po”)
```

```
До
Привет
После
```

Kolejność wyświetlania — od góry do dołu, tak jak kolejność wykonywania wierszy

Kontrola przepływu

Wywołanie przenosi sterowanie do ciała funkcji; po zakończeniu funkcji sterowanie wraca do dokładnie następnej linii

Uczeń musi dosłownie **zobaczyć**: Zadzwoń → wejdź do funkcji → wykonaj ciało → powrót do lokalizacji połączenia → kontynuacja od następnej linii. Funkcja nie jest wykonywany „gdzieś na uboczu”

Lokalizacja połączenia

Linia, gdzie jest `greet()`, nazywa się **miejsce wywołania** (call site). To tutaj wraca sterowanie, gdy funkcja się kończy:

mesto-vyzova.py

```
result = calculate(2, 3)
# ↑ to jest miejsce połączenia — po calculate() wykonanie
zostanie kontynuowane NATYCHMIAST TUTAJ
```

Praktyka: Przewidywanie kolejności wycofania
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/13-10/index.html)

Dlaczego potrzebuję funkcji?

Parameter to nazwa w definicji. Argument to wartość przekazywana podczas wywołania. Różnica między nimi jest fundamentem wszystkiego, co następuje dalej.

Dlaczego potrzebuję funkcji?

- **Kod nie jest powtarzany** — poprawka jest dokonywana w jednym miejscu.
- **Łatwiejsze do czytania** — nazwa funkcji wyjaśnia, co się dzieje, nie zmuszając do wgłębiania się w szczegóły implementacji.

- **Łatwiej szukać błędów** – jeśli coś jest zepsute w powitaniu, na pewno się to zrobi Wiedz, gdzie szukać: Do środka `privetstvie()`.

Za każdym razem, gdy robimy coś nowego!

Funkcja bez wejścia zawsze robi to samo. Aby funkcja zachowywała się inaczej W zależności od sytuacji otrzymuje **argumenty** są wartościami w nawiasach w miejscu Rozmow:

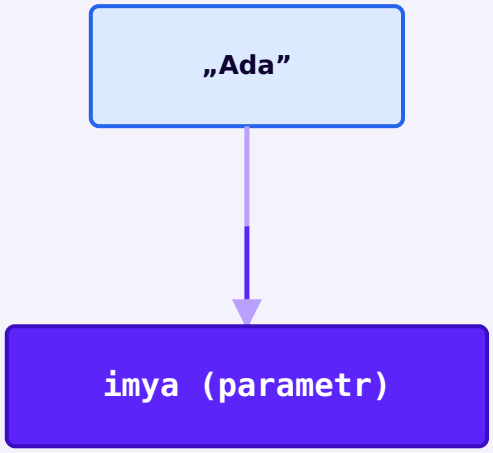
argumenty.py

```
def privetstvie(imya):
    print(f"Cześć, {imya}!")

privetstvie(„Ada”)
privetstvie("Cartesian")
```

Parametr vs argument — dokładna różnica

Te dwa słowa łatwo pomylić, ale mają różne role:

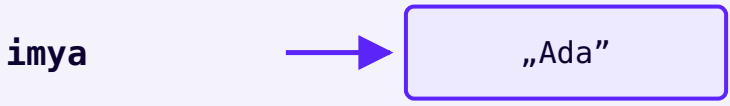
	Gdzie się pojawia	Co to jest
Parametr	w definicji: `def privetstvie(imya):`	lokalna nazwa wewnątrz funkcji
Argumentacja	w rozmowie: `privetstvie("Ада")`	obiekt/wartość przekazana, gdy



Argument z miejsca wywołania staje się wartością parametru wewnątrz funkcji

Parametry to nazwy lokalne

Kontynuujmy znany model „nazwij wskazuje na obiekt”



Inside the privetstvie(Ada" wywołanie), nazwa imya wskazuje na przekazany obiekt

"Transfer by meaning" i "by reference" są mylące
Nie traktuj Python jako języka, w którym "wszystko przekazywane jest przez odniesienie" lub "wszystko przekazywane jest przez znaczenie" — oba sformułowania są tu zbyt nieprecyzyjne i zbędne. Dokładniej, w ten sposób: **przy wywołaniu funkcji jej parametry są powiązane z tymi samymi obiektami, co przekazane argumenty** to ten sam model „nazw → obiekt” *call by sharing*, ale specjalnego terminu nie trzeba zapamiętywać — ważny jest sam model.

Bez kłótni?

Funkcja bez parametrów w nawiasie – na przykład, `privetstvie()` z §13.2 — też całkowicie normalne: nie każda funkcja potrzebuje danych wejściowych.

Praktyka: argument parametru vs
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji
Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/13-02/index.html)

Argumenty zmienne i niezmienne

Jedna z najważniejszych lekcji rozdziału: przypisanie parametru i mutacja obiektu, na który on wskazuje, to dwie zupełnie różne czynności.

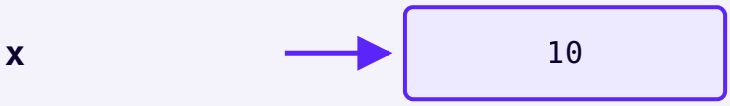
Niezmienny argument: rebinding nie wychodzi

immutable_argument.py

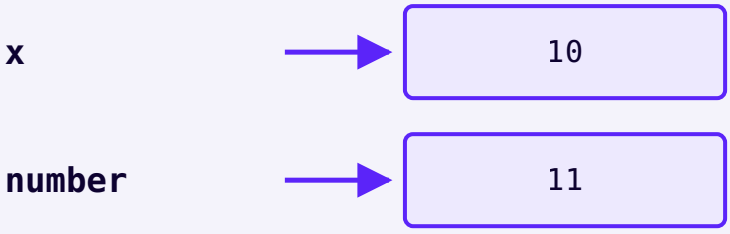
```
def add_one(number):
    number += 1
    print(number)

x = 10
add_one(x)
print(x)
```

```
11
10
```

add_one(x) drukuje 11, ale x na zewnątrz to 10



Przed zgłoszeniem: 10 x →



Wewnątrz add_one: number += 1 wiąże number z obiektem NEW 11 – x nie wpływa to

Nie „funkcja otrzymuje kopię x”
Dokładniej: `number` pierwsze wskazuje na to samo miejsce co `x` (o 10). Ale `number += 1` jest `number = number + 1` : Tworzony jest NOWY obiekt `11` , i `number` zaczyna wskazywać na niego. `x` nadal wskazuje na stary obiekt `10` - rozstali się.

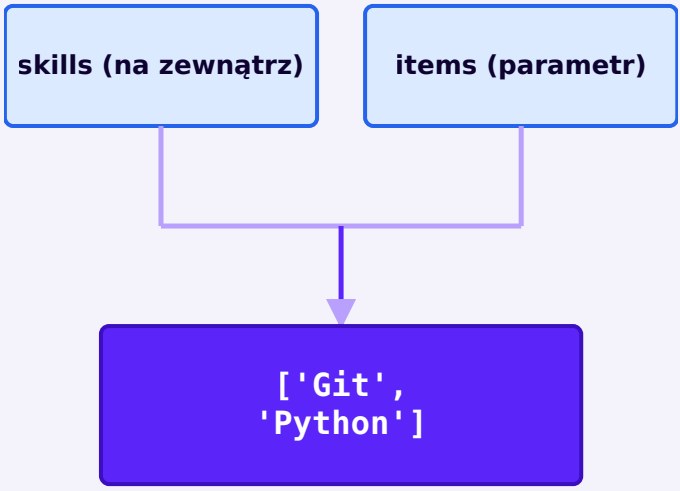
Argument modyfikowalny: mutacja widoczna z zewnątrz

mutable_argument.py

```
def add_item(items):
    items.append("Python")

skills = ["Git"]
add_item(skills)
print(skills)
```

```
['Git', 'Python']
```

skills się zmienił, chociaż wywołaliśmy funkcję, a nie przypisaliśmy skills bezpośrednio



skills i items — dwa imiona JEDNEJ listy; items.append(...) zmienia ją dla obu

To nie magia — to ten sam aliasing z rozdziału 11

`items` wewnątrz funkcji wskazuje na TEN sam lista obiekt co `skills` na zewnątrz. `.append(...)` mutuje ten współdzielony obiekt — więc zmiana jest widoczna przez oba nazwy. To dokładnie ta sama zasada aliasing, którą omawialiśmy w §11.9, tylko że teraz jedna z „dwóch nazw”

Rebinding vs mutation — porównujemy bezpośrednio

rebinding_vs_mutation.py

```
def replace(items):
    items = ["new"]      # rebinding: items teraz wskazuje na
INNY lista

def modify(items):
    items.append("new")  #mutation: zmienia się TAK SAMO lista
```

	replace(items)	modify(items)
Co się dzieje	ponowne przypisanie lokalnej nazwy `items`	zmiana obiektu, do którego `items` wskazuje
Czy widzisz to z zewnątrz po rozmowie?"	nie – wszystko jest takie samo jak na zewnątrz	tak – ten sam obiekt się zmienił

Główna zasada praktyczna

Przypisanie parametrowi (`items = ...`) usuwa lokalną nazwę z argumentu — nic się nie zmienia poza argumentem. Nazywanie metody mutacji (`.append()` , `.sort()` , `[i] = ...`) zmienia sam obiekt — a jest to widoczne z zewnątrz, jeśli obiekt jest zmienny.

Praktyka: rebinding vs mutation

interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/13-11/index.html)

Argumenty pozycyjne i nazwane

Dwa sposoby, by powiedzieć funkcji, który argument należy do którego parametru — każdy ma swoje zalety.

Kilka parametrów

neskolko_parametrov.py

```python
def rectangle_area(width, height):
    return width * height

print(rectangle_area(10, 5))   # 50
```

Argumentacja	Parametr
10	width
5	height

Argumenty pozycyjne

Domyślnie Python dopasowuje argumenty do parametrów **kolejność** — pierwszy argument należy do pierwszego parametru i tak dalej:

pozicionnye.py

```python
create_user("Anna", 25)
# position 1 → name = "Anna"
# position 2 → age  = 25
```

Pomieszanie porządku nie zawsze jest błędem Python, lecz zawsze błędem logicznym

`create_user(25, "Anna")` może nie wywołać żadnego błędu typów — Python nie zabrania przekazania liczby tam, gdzie zwykle przekazuje się łańcuch znaków. Ale wynik będzie logicznie nieprawidłowy: `name` otrzyma `25`, a `age` — `"Anna"`. Argumenty pozycyjne są wygodne, ale wymagają zapamiętania dokładnej kolejności.

Nazwane argumenty

Argument można przekazać za pomocą nazwy parametru, więc kolejność nie jest już istotna:

imennye.py

```
create_user(
    name="Anna",
    age=25,
)
```

Dlaczego nazwane argumenty

CZYTELNOŚĆ

możesz zobaczyć, co jest jak, prosto w rozmowie

PORZĄDEK NIE JEST WAŻNY

age=25, name=„Anna”

WYGODA API

szczególnie przy wielu parametrach

Mieszanie pozycjonalne i nazwane

smeshannye.py

```
def function(width, height, color):
    ...

function(10, height=20, color="red")   # możesz
```

Reguła porządku
Argumenty pozycyjne zawsze pojawiają się PRZED nazwanymi argumentami w rozmowie — `function(width=10, 20, "red")` spowoduje `SyntaxError`. I nie można uzyskać tego samego parametru dwa razy — `function(10, 20, width=5)` spowoduje `TypeError`, ponieważ `width` już otrzymał wartość pozycyjnie.

Praktyka: argumenty pozycyjne i nazwane
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/13-12/index.html)

Brak argumentów? Za dużo argumentów!

Domyślne wartości czynią argument opcjonalnym — ale zmienna wartość domyślna skrywa jedną z najbardziej znanych pułapek Python.

Brak argumentów? Co robić!

Jeśli wywołać funkcję bez obowiązkowego argumentu, Python zgłosi błąd. Aby argument można było pominąć, nadaje mu się **domyślny**:

znachenie_po_umolchaniyu.py

```
def privetstvie(imya=„przyjaciel”):
    print(f"Cześć, {imya}!")

privetstvie()          #Hello przyjacielu!
privetstvie(„Ada”)     # Cześć, Ada!
```

Podpis	Rola
`imya`	obowiązkowy parametr — jeśli nie było ciszy
`imya="друг"`	parametr opcjonalny z wartością domyślną

Pułapka zmiennej domyślnej mutowalnej

To jedna z najsłynniejszych niespodzianek Python – i trzeba ją powoli rozwiązywać.

mutable_default_trap.py

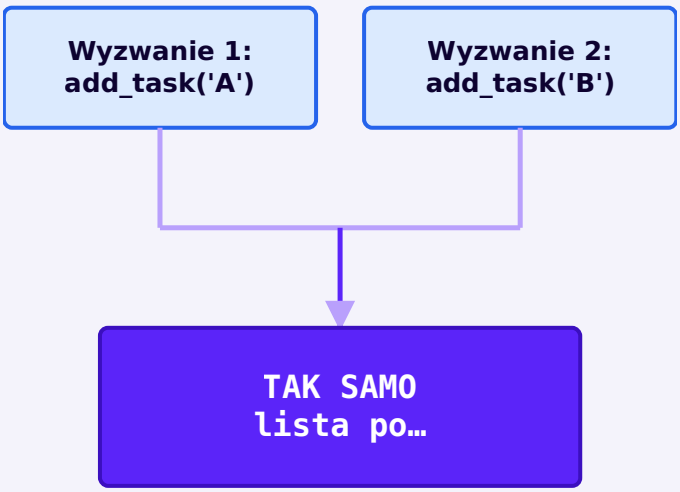
```
def add_task(task, tasks=[]):
    tasks.append(task)
    return tasks

print(add_task("A"))
print(add_task("B"))
```

```
['A']
['A', 'B']
```

Drugie wywołanie nie zaczyna od pustej listy — B dodał się do tej samej listy co A!

Domyślna lista jest tworzona RAZ – w momencie wykonania def
`tasks=[]` oblicza się w momencie **definicje** funkcje, a nie od nowa przy każdym wywołaniu. Powstaje ten sam obiekt domyślny lista, wspólny dla wszystkich wywołań, gdzie argument nie został przekazany jawnie.

Domyślna lista jest tworzona raz, gdy Python sam wykonuje wiersz def



Oba wywołania bez wyraźnego tasks używają tej samej wspólnej lista

Poprawny wzór: None jako znak „nie transmitowany”

mutable_default_fix.py

```
def add_task(task, tasks=None):
    if tasks is None:
        tasks = []

    tasks.append(task)
    return tasks
```

None jest domyślnym argumentem bezpiecznego sygnału
`None` jest niezmiennym pojedynczym obiektem, więc można go bezpiecznie używać jako domyślnego. Walidacja `if tasks is None` (Rozdział 9: `is None`) tworzy NOWY pusty lista przy każdym wywołaniu bez argumentu — dokładnie to zachowanie, którego intuicyjnie oczekiwano. To nie jest jedyny sposób rozwiązania problemu, ale najczęściej spotykany.

Za dużo kłótni!

Czasem nie wiadomo, ile argumentów trzeba przegłosować z wyprzedzeniem. Symbol `*` przed nazwą parametru zbiera **żadnych** liczba argumentów w jednej krotki (Rozdział 11):

args.py

```
def summa_vseh(*chisla):
    itog = 0
    for n in chisla:
        itog += n
    return itog

print(summa_vseh(1, 2))           # 3
print(summa_vseh(1, 2, 3, 4, 5))  #15 — Tyle argumentów, ile
chcesz
```

Argumenty nazwane – przypomnienie
Argumenty można przekazywać również po nazwie, a nie tylko po kolejności: `privetstvie(imya="Ада")` – więcej na ten temat w §13.6.

Praktyka: Domyślne i Pułapka Domyślna Zmienna
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/13-04/index.html)

*args, **kwargs i rozpakowywanie

*args zbiera argumenty pozycyjne do krotki, **kwargs — argumentów dodatkowych nazw w słownik. Te same postacie w momencie rozmowy rozkładają zbiór z powrotem na argumenty.

*args to dowolna liczba argumentów pozycyjnych

args_povtorenie.py

```
def total(*numbers):
    return sum(numbers)

total(1, 2, 3, 4)
```

Funkcja wewnętrzna numbers jest krotką regularną (Rozdział 11)

args to po prostu popularne imię
To symbol działa `*` przed parametrem jest nazwa `args` jedynie powszechnie przyjęta konwencja. `def total(*numbers)` działa dokładnie tak samo jak `def total(*args)` .

**kwargs to dowolna liczba nazwanych argumentów

kwargs.py

```
def show_profile(**fields):
    for key, value in fields.items():
        print(f"{key}: {value}")

show_profile(
    name="Anna",
    city="Warsaw",
)
```

Funkcja wewnętrzna fields jest regularnym słownik (Rozdział 11)

**** zbiera dodatkowe nazwane argumenty w słownik**
Wszystkie argumenty nazwane, dla których nie ma osobnego parametru, trafiają do słownik `fields` — klucz - nazwa argumentu, wartość - przekazana wartość.

*args vs **kwargs

vyzov_s_oboimi.py

```
func(10, 20, color="red", size=3)
```

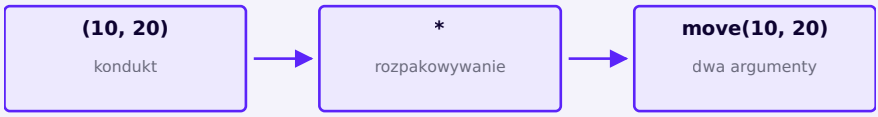
	*args	**kwargs
Zbiera	niepotrzebne argumenty pozycyjne	dodatkowe argumenty nazwane
Wpis w funkcji	`tuple`	`dict`
W powyższym przykładzie	`(10, 20)`	`{'color': 'red', 'size': 3}`

Rozpakowywanie argumentów w miejscu wywołania

Ten sam symbol `*` / `**`, ale teraz po stronie WYWOŁANIA, a nie definicji — łączymy z rozpakowaniem z rozdziału 11:

raspakovka_pri_vyzove.py

```
point = (10, 20)
move(*point)   # jest równoważne move(10, 20)
```



* przed argumentem, gdy wywołany, rozkłada krotkę/lista na oddzielne argumenty pozycyjne

raspakovka_slovarya.py

```
options = {
    "color": "red",
    "size": 3,
}
draw(**options)   # odpowiada draw(color=„red”
```

* i ** występują w kilku różnych kontekstach

Kontekst	Przykład	Co robi
Definicja funkcji	`def f(*args, **kwargs):`	zbiera niepotrzebne argumenty
Wywołanie funkcji	`f(*values, **mapping)`	rozkłada zbiór na argumenty
Kolekcja Dosłowna (Rozdział 11)	`[*a, *b]`	łączy kolekcje

Jeden znak, różne znaczenie w zależności od kontekstu
`*` oraz `**` nie oznacza dosłownie tej samej operacji wszędzie — ich rola zależy od tego, czy są w definicji funkcji, w wywołaniu czy w kolekcji dosłownie.

Praktyka: *args, **kwargs i rozpakowywanie na wezwanie
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji
Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/13-13/index.html)

Positional-only i keyword-only

Czasami chcesz wymusić, by argument był przekazywany tylko przez nazwę (lub tylko pozycję) — sygnatura funkcji może tego wyraźnie wymagać.

Keyword-only parametry — praktyczne już teraz

Są parametry, które CHCESZ wymusić, by przekazywały się tylko z nazwy – zazwyczaj są to opcjonalne flagi lub ustawienia, gdzie pozycja „goła”

keyword_only.py

```
def draw_rectangle(width, height, *, color="blue",
filled=False):
    ...

draw_rectangle(100, 50, color="red")   # możesz
```

Single * na liście opcji
Wszystko, co stoi PO samotności `*` w definicji funkcji musi być przekazywane nazwą przy wywołaniu — parametry pozycyjne nie mogą być przekazane.

bez_keyword_only.py

```
draw_rectangle(100, 50, "red", True)
# co tu jest True? filled? Nieoczywiste.
```

s_keyword_only.py

```
draw_rectangle(
    100,
    50,
    color="red",
    filled=True,
)
# od razu widać, co jest czym
```

Druga opcja jest znacznie szybsza do czytania i zrozumienia — dlatego istnieją keyword-only parametrów.

Trochę głębiej – positional-only parametry

Rzadziej, ale też występuje odwrotna sytuacja: parametr, który MOŻE być przekazany tylko pozycjonalnie, bez nazwy.

positional_only.py

```
def function(x, /):
    ...
```

Samotny / w liście parametrów
Wszystko, co warte PRZED `/`, można przekazać tylko pozycyjnie — nazwy parametru nie można używać podczas wywołania. Tak robi się, gdy autor funkcji nie chce, aby nazwa parametru stała się częścią „kontraktu” — będzie można ją swobodnie zmieniać w przyszłości, bez łamania kodu, który ją wywołuje.

Anatomia pełnej sygnatury

signatura.py

```
def draw(
    x, y, /,
    width, height,
    *,
    color="blue",
    filled=False,
):
    ...
```

Jeden podpis, trzy strefy

X, Y

tylko pozycyjnie
do /

WIDTH, HEIGHT

pozycjonalnie LUB po imieniu
pomiędzy / a *

COLOR, FILLED

```
tylko z imienia
po *
```

Nie musisz tego zapamiętywać za pierwszym razem
To zaawansowana, ale cenna umiejętność czytania API — szczególnie przydatna przy czytaniu dokumentacji zewnętrznych bibliotek. Dla własnych małych funkcji zwykle wystarczą zwykłe parametry bez `/` i samotny `*` — używaj ich świadomie, gdy naprawdę poprawia czytelność wywołania.

Praktyka: keyword-only parametry
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji
Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/13-14/index.html)

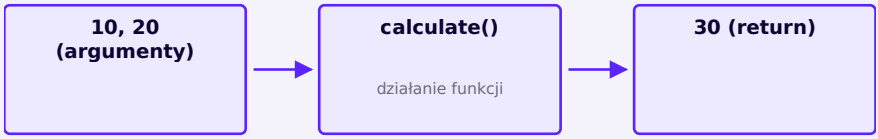
Odpowiedz

return przekazuje wynik funkcji z powrotem tam, z którego została wywołana — a funkcja bez return nadal zwraca coś: None.

Jak dotąd nasze funkcje drukowały tylko tekst. Ale często nie trzeba drukować wyniku, aby ekran, oraz **powrót** go wykorzystać dalej w programie. Słowo kluczowe `return` robi dokładnie to:

return.py

```
def summa(a, b):
    return a + b

result = summa(5, 7)
print(result)         # 12
print(summa(2, 3) * 10)
# wynik funkcji można użyć natychmiast — 50
```



Funkcja → wejście→ wynik wraca do lokalizacji połączenia

print() wewnątrz funkcji — to nie to samo co return
Funkcja, która tylko drukuje wynik, nic nie *zwroty* — próba zapisania wyniku do zmiennej skutkuje `None`:

print_vs_return.py

```
def summa_pechataet(a, b):
    print(a + b)   # wyświetla się, ale nie wraca

x = summa_pechataet(5, 7)  #12 pojawi się na ekranie
print(x)                    # Ale x jest None!
```

	print(...) wewnątrz funkcji	return ... wewnątrz funkcji
Rezultat	tekst na ekranie	obiekt przechodzi w miejsce wywołania
x = f(...)	x stanie się None	x stanie się wartością obliczoną

Ukryte None

Jeśli funkcja osiąga koniec ciała, nie wykonując jej nigdy `return`, Python sam ją odwzajemnia `None`:

implicit_none.py

```
def hello():
    print("Hello")

result = hello()
print(result)
```

```
Hello
None
```

hello() nic nie return- robi — Python sam wstawia None

None jest rzeczywistym obiektem, a nie „pustką”
`None` jest konkretnym dopełnieniem Python oznaczającym brak znaczenia w danym kontekście (Rozdział 9). Funkcja bez `return` nie „niczego nie zwraca” w sensie całkowitego braku wyniku — zwraca właściwie obiekt `None` .

Nagi return

`return` bez wyrażenia po tym, jak również zwraca `None` — i natychmiast kończy funkcję. Przydatne na początku Efekt:

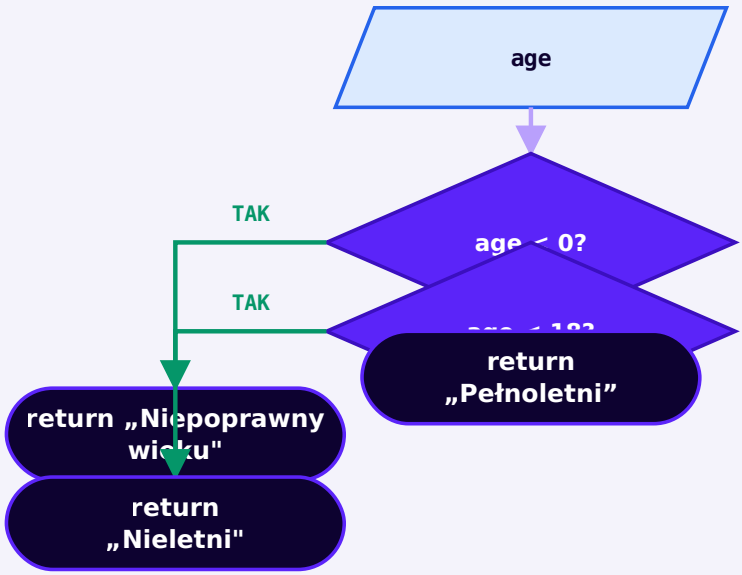
bare_return.py

```
def process(value):
    if value is None:
        return
    print(value)
```

Wczesne return

early_return.py

```
def classify_age(age):
    if age < 0:
        return „Nieprawidłowy wiek”

    if age < 18:
        return „Minor”

    return „Dla dorosłych”
```



Każdy return natychmiast kończy funkcję — kod następujący po niej w tym wywołaniu nie wykona się

return natychmiast kończy funkcję
Jak tylko zostanie wykonane `return` funkcja kończy się natychmiast — kod po niej wewnątrz funkcji nie zostanie wykonany.

Zwracaj wiele wartości

neskolko_znachenij.py

```
def min_max(numbers):
    return min(numbers), max(numbers)

low, high = min_max([3, 7, 1, 9])
print(low, high)   # 1 9
```

W rzeczywistości zwraca jedną krotkę
`return a, b` na poziomie Python tworzy i zwraca JEDEN obiekt — krotkę `(a, b)` (rozdział 11). Funkcja nie „zwraca dwóch obiektów jednocześnie” — zwraca jeden krotkę, którą następnie można rozpakować na dwie zmienne podczas wywołania, dokładnie jak każdą inną krotkę.

Praktyka: return, implicit None, wczesne return
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/13-03/index.html)

Wejście, Operacja, Wyjście: Funkcje czyste

Nie każda funkcja musi zwracać coś, ale warto jasno zrozumieć, do czego każda konkretna funkcja jest napisana: dla dobra wyniku czy dla samego efektu.

Ogólny Model Cech

Parametry → ciało funkcji → return – modelu stosowalnego do dowolnej funkcji z wynikiem

Przykład	Loguj	Wyjście
`convert_temperature`	stopni Celsjusza	stopni Fahrenheita
`normalize`	surowy tekst	wyczyszczony tekst
`calculate_average`	lista liczby	singiel

Dwa legalne rodzaje funkcji

Nie każda funkcja musi zwracać wartość — funkcje mają dwie legalne role:

Funkcja-Polecenie/Efekt	Funkcja obliczeniowa
`show_menu()`, `draw_square()`, `print_report()`	`area(...)`, `normalize(...)`, `average(...)`
główny cel — efekt (wyświetlanie, rysowanie)	głównym celem jest zwrot wartości

Oba rodzaje są legalne
Nie każda funkcja jest zobowiązana zwracać coś znaczącego — `draw_square()` jest całkiem normalne, nawet jeśli nie jest return-ith: jego działanie to efekt na ekranie, a nie wartość.

Skutki uboczne

Skutki uboczne — to gdy funkcja zmienia coś POZA swoim wynikiem zwracanym: drukuje na ekran, rysuje Turtle, mutuje przekazany lista, później — zapisuje do pliku.

storonnij_effekt.py

```
def add_item(items):
    items.append("Python")   #side efekt: mutuje przekazywany
lista
```

Funkcje czyste

Czysta funkcja - wynik zależy wyłącznie od argumentów, a celowo nie zmienia niczego poza sobą:

chistaya_funkciya.py

```
def rectangle_area(width, height):
    return width * height
```

Funkcja czysta: tylko wejście → wynik

Funkcja skutku ubocznego: OR/I Efekt Zewnętrzny

dlaczego czyste funkcje są wygodne

ŁATWIEJSZE DO ZROZUMIENIA

wynik zależy wyłącznie od wejścia

ŁATWIEJSZE DO PRZETESTOWANIA

nie potrzeba ekranu/pliku/sieci – wystarczy połączenie

PRZEWIDYWALNOŚĆ

`to samo wejście → zawsze to samo wyjście`

Nie każda funkcja MUSI być czysta
Grafika, I/O i obsługa interfejsu to z natury skutki uboczne. Czystość jest przydatna tam, gdzie jest naturalna, a nie uniwersalna zasada, której każda funkcja programu musi przestrzegać.

Praktyka: czyste funkcje vs funkcje z efektem ubocznym
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/13-15/index.html)

Zmienne globalne i lokalne

Gdzie „mieszka” zmienna i w jakiej kolejności Python jej szuka — model LEGB uwolni wielu dziwnych błędów.

Zmienne wewnątrz funkcji

Zmienna utworzona wewnątrz funkcji nazywana jest **lokalny** — ona istnieje tylko wtedy, gdy funkcja jest uruchomiona i nie jest dostępna z zewnątrz:

lokalnye_peremennye.py

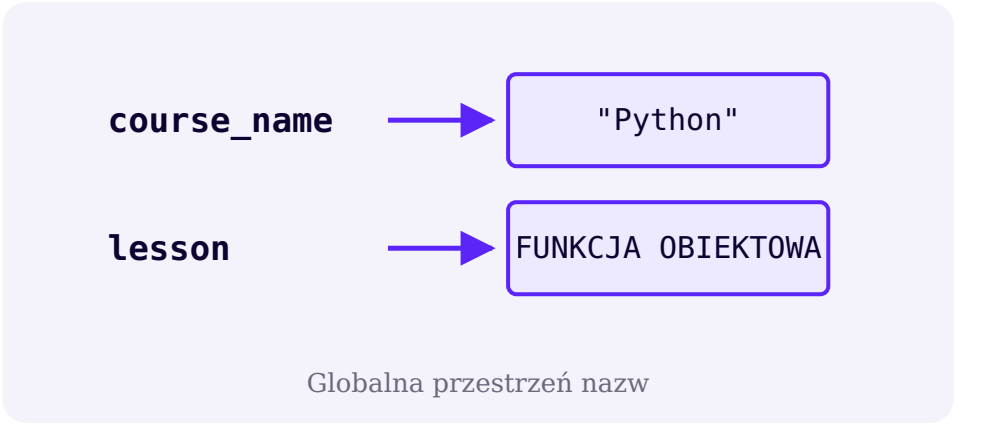
```
def moya_funkciya():
    message = „Mieszkam tylko wewnątrz funkcji”
    print(message)

moya_funkciya()
print(message)   #NameError: message tutaj nie jest
zdefiniowane
```

Zmienne globalne

Global zmienna jest deklarowany poza wszystkimi cechami — i jest czytelny W każdym z nich:

globalnye_peremennye.py

```
course_name = "Python"

def lesson():
    topic = „Funkcje”
    print(course_name)   # Czytanie globalnie – Praca bez
żadnych zastrzeżeń
    print(topic)

lesson()
```



Globalna przestrzeń nazw

Lokalna przestrzeń nazw lesson() podczas wywołania

Funkcja może patrzeć na zewnątrz, ale nie odwrotnie
Funkcja może ODCZYTYWAĆ nazwy globalne, ale zmienne utworzone w niej nie są automatycznie widoczne z zewnątrz.

Cieniowanie (shadowing)

shadowing.py

```
name = "GLOBAL"

def demo():
    name = "LOCAL"
    print(name)

demo()
print(name)
```

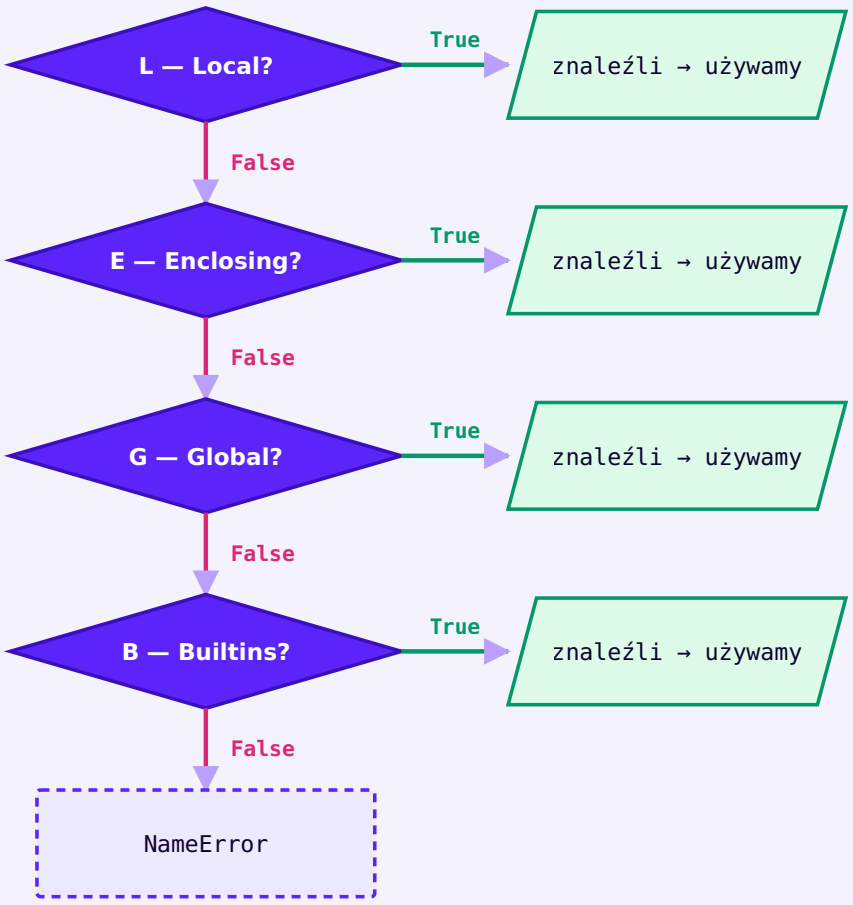
```
LOCAL
GLOBAL
```

Lokalna nazwa name istnieje tylko w obrębie demo() i nie wpływa na globalną

Lokalna nazwa przychodzi do globalnej nazwy o tej samej nazwie
Wewnątrz `demo()` TWOIM tworzymy osobną lokalną nazwę `name` tymczasowo „przyćmiewa” `name` kod wewnątrz funkcji, ale nie zastępuje ani nie usuwa go.

LEGB: Python jak szuka imienia

Gdy Python napotyka nazwę, szuka jej w jasnej kolejności — ten model nazywa się **LEGB**:



Local → Enclosing → Global → Builtins - Poszukiwania zatrzymują się przy pierwszym dopasowaniu

Litera	Co to jest	Przykład
L — Local	nazwy wewnątrz bieżącej funkcji	`topic` w środku `lesson()`

Litera	Co to jest	Przykład
E — Enclosing	nazwy funkcji zewnętrznej dla funkcji zagnieżdżonej (§13.13)	spójrzmy dalej
G — Global	nazwy na poziomie modułu	`course_name`
B — Builtins	wbudowane nazwy Python	`len`, `print`

UnboundLocalError — dlaczego to NIE jest przypadek

unbound_local.py

```
count = 10

def increment():
    count += 1

increment()
```

Przypisanie w dowolnym miejscu ciała sprawia, że nazwa jest lokalna W CAŁYM tkanku funkcji

Python widzi `count += 1` (to `count = count + 1`) i z góry decyduje: `count` jest przypisana wewnątrz funkcji, więc jest to nazwa LOKALNA — w całym ciele funkcji, nie tylko po przypisaniu. Ale po prawej stronie `count + 1` próbuje CZYTAĆ `count` ZANIM pojawiło się lokalne znaczenie — stąd stąd `UnboundLocalError`. To nie jest zachowanie losowe, lecz bezpośrednia konsekwencja sposobu, w jaki Python wstępnie oznacza nazwy funkcji lokalnych.

global — jeśli naprawdę trzeba zmienić zmienną globalną

global_keyword.py

```
count = 10

def increment():
    global count
    count += 1

increment()
print(count)   # 11
```

global jest potrzebne tylko do zadania, nie do czytania
Możesz po prostu ODCZYTAĆ zmienną globalną bez `global` (robiliśmy to już z `course_name` powyżej). `global` jest potrzebne tylko wtedy, gdy funkcja ma zamiar ZMIENIĆ (przydzielić) nazwę globalną.

global działa, ale częściej istnieje lepszy sposób
`global` nie jest zakazaną, ale często nie najlepszą praktyką: funkcje odczytujące parametry i zwracające wynik przez `return` łatwiej testować i ponownie używać, ponieważ ich zachowanie nie zależy od ukrytego stanu zewnętrznego. To nie jest absolutna zasada „zmienne globalne są zawsze złe”

Praktyka: zmienne lokalne, globalne i LEGB
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji
Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/13-05/index.html)

Zagnieżdżone funkcje i nonlocal

Funkcję można zdefiniować bezpośrednio wewnątrz innej funkcji — i nonlocal pozwala jej zmieniać nazwę funkcji obejmującej, a nie tylko ją odczytywać.

Funkcja w obrębie funkcji

Funkcję można zdefiniować bezpośrednio wewnątrz ciała innej funkcji:

vlozhennaya_funkciya.py

```
def outer():
    message = "Hello"

    def inner():
        print(message)

    inner()

outer()
```

inner widzi message jest Enclosing LEGB
`inner()` nie ma własnego `message`, ale znajduje go w funkcji obejmującej `outer()` — to litera „E” (Enclosing) z drabinki LEGB (§13.12).

Wgłębienie nie jest ozdobą, lecz deklaracją gniazdowania
Gdy przypadkowo przesuniesz `def inner():` poziom wcięcia jest mniejszy niż konieczne, nie będzie już zagnieżdżony w `outer` — i odwrotnie. Wcięcie dosłownie określa, która funkcja jest „wewnątrz”

nonlocal

Podobnie jak w przypadku zmiennych globalnych, można swobodnie odczytać nazwę z funkcji obejmującej, oraz oto ZMIANA (przypisuj) – musisz wyraźnie zapytać:

nonlocal.py

```
def outer():
    count = 0

    def inner():
        nonlocal count
        count += 1

    inner()
    inner()
    print(count)

outer()
```

```
2
```

nonlocal pozwalało inner() modyfikować count z outer() zamiast tworzyć lokalną kopię

nonlocal — to NIE global

`nonlocal` szuka nazwy w NAJBLIŻSZEJ odpowiedniej funkcji obejmującej — nie na poziomie modułu. Jeśli usunąć `nonlocal count` otrzymasz ten sam błąd `UnboundLocalError` jak w §13.12 — z tego samego powodu: przypisanie w ramach `inner` by się dobrze `count` lokalnej nazwy `inner`.

Trochę głębiej — zamknięcie (closure)

Funkcja zagnieżdżona potrafi „zapamiętać” wartości z funkcji otaczającej nawet po tym, jak wywołanie zewnętrzne już się zakończyło:

closure.py

```
def make_greeter(name):
    def greet():
        print(f"Cześć, {name}!")
    return greet

greet_anna = make_greeter("Anna")
greet_anna()   #Hey Anna! — chociaż make_greeter już skończył pracę
```

Zamknięcie to temat na przyszłość

Tak się nazywa **zamknięcie** — `greet` dostęp „zabiera ze sobą” `name` . Wystarczy tu zauważyć, że w ogóle tak się dzieje — szczegółowa analiza zamknięć i ich praktycznych zastosowań (dekoratorów i nie tylko) nie została uwzględniona w tym rozdziale.

Praktyka: funkcje zagnieżdżone i nonlocal

interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/13-16/index.html)

Stos wywołań i traceback

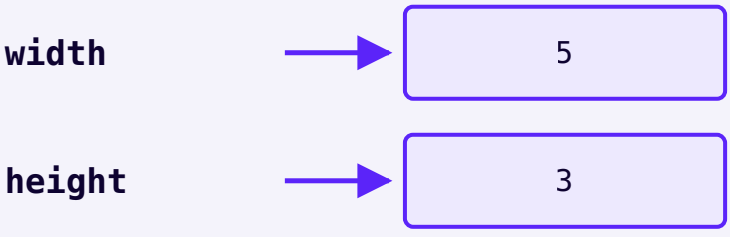
Funkcje wywołujące funkcje tworzą stos oczekujących wywołań — a traceback z rozdziału 3 faktycznie pokazuje ten stos w momencie wystąpienia błędu.

Ramka wywoławcza (call frame)

Gdy funkcja jest wywoływana, Python tworzy tymczasowy „kontener”

call_frame.py

```
def area(width, height):
    return width * height

area(5, 3)
```



Ramka wywołania area(5, 3): Parametry istnieją do momentu zakończenia danego wywołania

Funkcje mogą wywoływać inne funkcje

vlozhennye_vyzovy.py

```
def calculate_tax(amount):
    return amount * 0.2

def calculate_invoice(amount):
    tax = calculate_tax(amount)
    return amount + tax

calculate_invoice(100)
```

Pa `calculate_tax()` wykonuje się, wywołanie `calculate_invoice()` się nie kończy — „czeka” **stos wywołań**.

Stos rośnie i kurczy się

Krok 1 – Wykonanie głównego programu

Krok 2 – main spowodował calculate_invoice()

Krok 3 — calculate_invoice wywołała calculate_tax(); stos na szczycie

Krok 4 – calculate_tax() zwrócił wynik i zniknął ze stosu

Ostatni wszedł — pierwszy wyszedł
Stos wywołań rośnie wraz z każdym nowym wywołaniem i maleje przy każdym zakończeniu – najnowsze wywołanie zawsze znajduje się na górze i kończy się pierwsze. Nazywa się to zasadą LIFO (Last In, First Out).

Traceback jest migawką stosu wywołań

Przypomnijmy sobie debugowanie z rozdziału 3: gdy pojawi się błąd, Python drukuje traceback — i to jest Dosłownie pokazuje łańcuch połączeń, który doprowadził do błędu:

traceback_primer.py

```
def divide(a, b):
    return a / b

def calculate():
    return divide(10, 0)

def main():
    calculate()

main()
```

```
Traceback (most recent call last):
  File "example.py", line 8, in <module>
    main()
  File "example.py", line 6, in main
    calculate()
  File "example.py", line 4, in calculate
    return divide(10, 0)
  File "example.py", line 2, in divide
    return a / b
ZeroDivisionError: division by zero
```

Każdy łańcuch znaków traceback to jedna klatka od stosu wywołań w momencie błędu

Czytanie traceback od dołu do góry
Najniższy łańcuch znaków — sama pomyłka. Linia nad nią — gdzie dokładnie się ona zdarzyła (`divide`). Powyżej – kto to wywołał (`calculate`), i jeszcze wyżej — kto wywołał tego (`main`). Traceback to dokładnie stos wywołań, który właśnie sobie wyobraziliśmy, pokazany w momencie, gdy coś poszło nie tak.

Praktyka: czytamy stos wywołań i traceback
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/13-17/index.html)

Projektowanie dobrej funkcji

Zanim napiszesz implementację, warto zaprojektować funkcję: jeden jasny obowiązek, zrozumiała nazwa, wyraźny kontrakt.

Dobre imiona – czasowniki akcji

Dobrze	Zły
`calculate_total()`	`do_it()`
`normalize_text()`	`func1()`
`draw_polygon()`	`thing()`
`find_max_score()`	`test2()`

Funkcje zwykle przedstawiają DZIAŁANIE — nazwa powinna być czasownikiem lub zwrotem czasownikowym. Dla funkcji zwracających `True` / `False`, działają dobrze prefiksy `is_`, `has_`, `can_`, `contains_` są heteroseksualni Zgłasza się, że wartość boolowska wróci.

Jedna funkcja, jedna jasna odpowiedzialność

plohaya_dekompoziciya.py

```
def process_everything():
    # prosi o opinię
    # sprawdzone
    # liczy się
    # odbitki
    # Losowanie
    # aktualizuje pięć struktur danych
    ...
```

horoshaya_dekompoziciya.py

```
def read_answer(): ...
def normalize_answer(answer): ...
def check_answer(answer, correct): ...
def show_result(correct): ...
```

To jest wytyczne, a nie dogmat
Małe programy mogą rozsądnie łączyć kilka prostych działań w jedną funkcję — nie musisz sztucznie dzielić każdej linii na osobną funkcję (zobacz §13.19 dla przykładu CZEGO NIE robić). Celem jest jasność, a nie liczba funkcji.

Ile linii powinna zajmować funkcja?

Nie ma zasady „maksymalnie 10 linijek”

Pytania zamiast liczby linii

1

Czy ma jeden jasny cel?

2

Czy nazwa wyjaśnia, co ona robi?

3

Czy łatwo jest to przetestować osobno?

4

Czy w środku jest mocno powtarzająca się logika?

5

Czy wejścia i wyjścia są czytelnikowi jasne?

Kontrakt funkcji

Zanim napiszesz implementację, warto wyraźnie sformułować kontrakt — co funkcja przyjmuje, co zwraca i czy ma efekty uboczne:

Część kontraktu	calculate_discount(price, percent)
Loguj	price: liczba, percent: liczba
Wyjście	obniżona cena
Skutki uboczne	nie
Przykład	`calculate_discount(100, 10) → 90`

Warunki wstępne

Niektóre funkcje oczekują znaczących danych wejściowych, takich jak `draw_polygon(sides, length)` zakłada `sides >= 3` oraz `length > 0`. To **warunki wstępne** — wymagania dotyczące danych wejściowych, które warto sprawdzić zwykłymi warunkami:

```
predusloviya.py
```

```python
def polygon_angle(sides):
    if sides < 3:
        return None

    return 360 / sides
```

None nie jest jedyną opcją

Zwroty `None` przy niepoprawnym wejściu — prosty i działający wariant na tym etapie kursu. Bardziej rygorystyczne metody (na przykład zgłoszenie wyjątek) są omawiane w rozdziałach poświęconych obsłudze błędów.

Lista kontrolna projektowania cech

Zanim napiszesz def

- Jaką JEDNĄ pracę ona wykonuje?
- Jakie dane potrzebuje? To są parametry.
- Czy zwraca wynik — i jaki wynik?
- Czy to zmienia coś celowo (efekt uboczny)?
- Jak nazwać ją tak, aby nazwa wyjaśniała akcję?
- Na jakich danych wejściowych powinno być testowane?

Praktyka: projektowanie funkcji na podstawie kontraktu

interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/13-18/index.html)

Dockstringi i wskazówki typu

Profesjonalnym nawykiem, który należy wypracować wcześnie, jest dokumentowanie funkcji, aby można ją było zrozumieć bez czytania implementacji.

Docstring — dokumentacja funkcji

docstring.py

```
def rectangle_area(width, height):
    """Zwraca pole prostokąta."""
    return width * height
```

	Komentarz (# ...)	Dokstring ("""...""")
Wyjaśnia	dlaczego kod napisano dokładnie w ten sposób	co dokładnie robi ta funkcja – jak API
Widoczne z ze-wnątrz?	nie, tylko w źródle	tak, przez help() i __doc__

help() pokazuje docstring podczas działania programu

help_primer.py

```
help(rectangle_area)
print(rectangle_area.__doc__)
```

```
Help on function rectangle_area in module __main__:

rectangle_area(width, height)
    Возвращает площадь прямоугольника.
```

help() pobiera docstring bezpośrednio z obiektu funkcji — to ten sam pomysł, który już widzieliśmy w przypadku funkcji wbudowanych

Type hints - Wskazówki typu

type_hints.py

```
def rectangle_area(width: float, height: float) -> float:
    return width * height
```

Część	Znaczenie
`width: float`	oczekiwany typ parametru
`-> float`	oczekiwany typ zwrotu

Wskazówki typu nie są automatycznie sprawdzane w czasie działania
`def add(a: int, b: int) -> int:` NIE zaszkodzi zadzwonić `add("2", "3")` — Python nie będzie automatycznie zamieniać linii na liczby i samo z siebie nie wygeneruje błędu. Adnotacje typów to dokumentacja i wskazówka dla narzędzi programistycznych, a nie runtime-sprawdzenie.

Adnotacje do kolekcji

annotacii_kollekcij.py

```
def average(scores: list[float]) -> float:
    return sum(scores) / len(scores)

def count_words(text: str) -> dict[str, int]:
    ...
```

Wartość opcjonalna w adnotacji

optional_annotation.py

```
def find_user(name: str) -> str | None:
    ...
```

str | None czyta się jako „łańcuch znaków lub None”
Taka adnotacja mówi: funkcja albo zwróci łańcuch znaków, albo `None` jeśli nic nie zostanie znalezione. To nie jest osobny temat, a jedynie sposób na wyraźne opisanie słowami tego, co już widzieliśmy w praktyce (np. `dict.get()`).

Adnotacje nie nadpisują domyślnej pułapki mutowalnej

annotacii_ne_spasayut.py

```
def add(item: str, items: list[str] = []):
    ...
    # wciąż ta sama pułapka z §13.7!
```

Adnotacje typów są dokumentacją, a nie ochroną przed błędami w czasie wykonywania
Nawet z pełnymi adnotacjami `items: list[str] = []` pozostaje tą samą zmienną wartością domyślną co wcześniej — adnotacja nie zmienia nic w rzeczywistym zachowaniu funkcji.

Praktyka: docstringi i wskazówki typów
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/13-19/index.html)

Funkcje jako obiekty

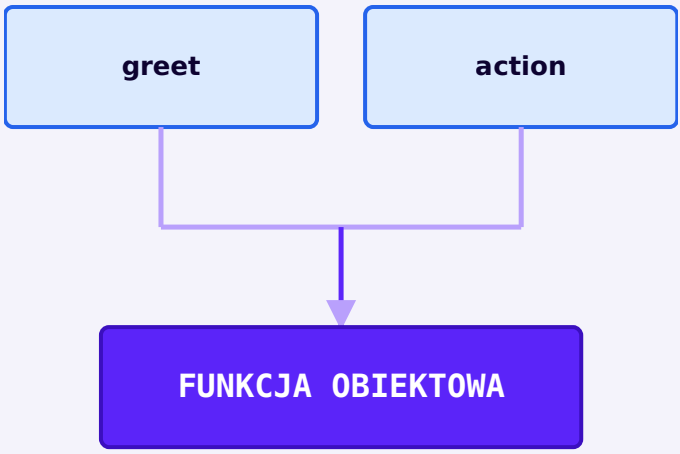
Funkcja — taki sam obiekt jak lista lub łańcuch znaków: można ją zapisać pod inną nazwą, umieścić w słownik, przekazać innej funkcji.

Funkcję można zapisać pod inną nazwą

Wróćmy do rozróżnienia z §13.2 — `greet` oraz `greet()` — i pójdźmy o krok dalej:

funkciya_kak_obekt.py

```
def greet(name):
    return f"Cześć, {name}"

action = greet

print(action("Anna"))   #Hello Anna
```



greet i action to dwie nazwy tego samego obiektu funkcji, tak jak w przypadku każdego innego obiektu (Rozdział 3)

To samo aliasing, co zawsze

`action = greet` nie kopiuje funkcji ani nie tworzy drugiej funkcji — obie nazwy wskazują na ten sam obiekt funkcji. Funkcje w Python to **obiekty pierwszej klasy**: Mogą być przechowywane w zmiennych, umieszczane w kolekcjach, przekazywane innym funkcjom — tak jak inne wartości.

Cechy w kolekcjach

funkcii_v_slovare.py

```
def double(x):
    return x * 2

def square(x):
    return x ** 2

operations = {
    "double": double,
    "square": square,
}

operation = operations["square"]
result = operation(5)
print(result)   # 25
```

Słownik + Rozdział 11

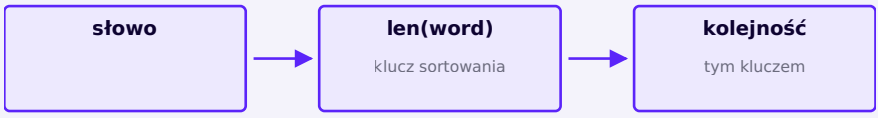
`operations` jest zwykłym słownik (Rozdział 11), który po prostu przechowuje nie liczby ani łańcuch znaków, lecz obiekty funkcji.

Przekazywanie funkcji jako argumentu

Już to zrobiłeś, nawet o tym nie myśląc:

sorted_key.py

```
words = ["python", „I”, „programowanie”]
print(sorted(words, key=len))
```



sorted() wywołuje funkcję przekazanej len dla każdego elementu, aby uzyskać klucz sortowania

len jest funkcją OBIEKTOWĄ, a nie wynikiem jej wywołania
`key=len` przekazuje samą funkcję `len` — BEZ nawiasów. `sorted()` sama to nazwie `len(...)` dla każdego elementu, gdy przyjdzie czas na porównanie. Jeśli piszesz `key=len()` byłaby próbą wywołania `len` teraz, bez argumentu, co spowoduje błąd.

Termin: funkcja wyższego rzędu

Funkcja, która przyjmuje inną funkcję jako argument (jak `sorted()`) lub zwraca funkcję (jako `make_greeter()` od §13.13), nazywa się **funkcję wyższego rzędu**. Specjalna teoria funkcjonalna Nie ma tu potrzeby programowania — wystarczy rozpoznać ten wzorzec, gdy się pojawi.

Praktyka: funkcje jako obiekty pierwszej klasy
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

`Otwórz praktykę →` (https://www.cartesianschool.org/pl/practice/13-20/index.html)

Funkcje lambda

Mały, nienazwany obiekt funkcji dla pojedynczego wyrażenia jest przydatny w wąskich przypadkach, a nie jako nowocześniejszy zamiennik def.

W §13.20 opublikowaliśmy `len` jako argument w `sorted()`. A co, jeśli potrzebna reguła sortowania nie istnieje jako gotowa funkcja — i zakładać osobną dla niej `def` wydaje się zbędne? Jest **lambda**.

lambda tworzy mały, nienazwany obiekt funkcji

lambda_sortirovka.py

```
students = [{"name": "Anna", "score": 82}, {"name": "Bob",
"score": 95}]
students.sort(key=lambda student: student["score"])
```

Nie „krótszy def”
Głównym zadaniem lambda jest przekazanie małej reguły/ transformacji do miejsca, gdzie pełnoprawna nazwana funkcja byłaby zbędna. To NIE jest nowsze zastępstwo `def` jest osobnym narzędziem dla wąskiego przypadku.

Anatomia lambda

lambda_anatomiya.py

```
lambda x: x ** 2
```

Część	Znaczenie
`lambda`	słowem kluczowym jest utworzenie funkcji wyrażenia
`x`	parametr (bez nawiasów)
`x ** 2`	pojedynczym wyrażeniu jego wynik jest automatycznie zwracany

lambda_vyzov.py

```
kvadrat = lambda x: x ** 2
print(kvadrat(5))   # 25
```

Brak wyraźnych informacji `return` i brak instrukcji blokowych w środku — Tylko jedno wyrażenie.

Ograniczenia lambda

Gdy lambda NIE JEST odpowiednia, należy def

TYLKO EKSPRESJA

żadnych instrukcji
ani przypisania, ani print

NIE DLA LOGIKI ZŁOŻONEJ

rozgałęzienia z efektami
długie obliczenia

BEZ NAME I DOCKSTRING

nie ma nic do pokazania w traceback
nic do dokumentowania

Jeśli logika zasługuje na nazwę — użyj def
Gdy reguła stanie się na tyle ważna, że warto ją ponownie stosować, testować osobno lub dokumentować, czas przekształcić lambda w stałą funkcję z `def`.

def → lambda: porównanie stylu, a nie „ulepszenie”

Prosta funkcja: def → lambda

Zwykła funkcja (def)

```
def kvadrat(x):
    return x ** 2

print(kvadrat(5))
```

Funkcja lambda

```
kvadrat = lambda x: x ** 2

print(kvadrat(5))
```

Co używać dzisiaj: zwykła funkcja z `def` jest czytelniejsza, ma zwykłą nazwę do debugowania i łatwo dodać kilka linii logiki lub komentarza. `lambda` wygodne tylko dla jednego krótkiego wiersza bez nazwy — na przykład jako argument funkcji `sorted()` lub `max()`. Przypisanie lambda imienia (`kvadrat = lambda x: ...`) nie jest bardziej „nowoczesny” `def` – w wielu prawdziwych projektach ten styl byłby uważany za mniej czytelny, a nie progresywny.

Opcjonalnie: map() i filter()

Skoro już znasz comprehensions (rozdział 11), warto zobaczyć ich alternatywę:

map_primer.py

```
numbers = [1, 2, 3, 4]
kvadraty = list(map(lambda n: n ** 2, numbers))
print(kvadraty)
```

comprehension_ekvivalent.py

```
numbers = [1, 2, 3, 4]
kvadraty = [n ** 2 for n in numbers]
print(kvadraty)
```

filter_primer.py

```
chetnye = list(filter(lambda n: n % 2 == 0, numbers))
```

comprehension_s_if.py

```
chetnye = [n for n in numbers if n % 2 == 0]
```

Python częściej woli comprehensions

Dla prostych transformacji i filtrów comprehension zwykle jest czytelniejszy niż `map()` / `filter()` z lambda. Znajomość obu stylów jest pomocna — oba znajdziesz w prawdziwym kodzie — ale domyślnie sięgnij po comprehension, jeśli zadanie jest proste.

Praktyka: funkcje lambda

interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/13-06/index.html)

Funkcje jak taśma produkcyjna

Kilka małych funkcji, każda z wyraźnym wejściem i wyjściem, może wykonać duże zadanie razem—tak dobre jak jeden długi skrypt, ale znacznie wyraźniej.

Wynikiem jednej funkcji jest dane wejściowe dla następnej

konvejer.py

```
clean = normalize_text(input_text)
words = split_words(clean)
counts = count_words(words)
```

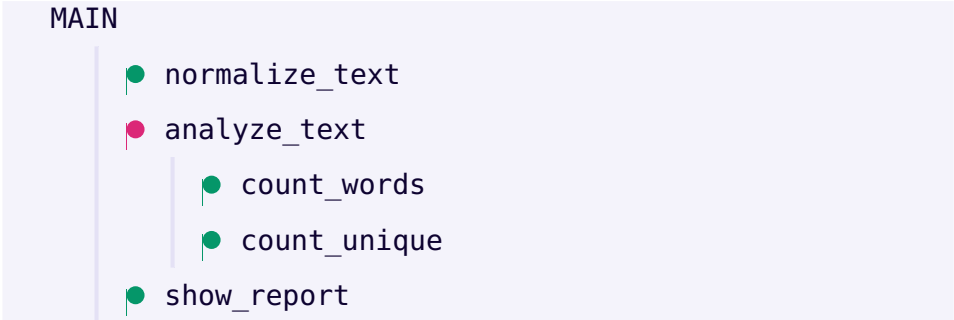
Każda funkcja jest jednym etapem zrozumiałym; wynik jednego staje się wejściem następnego

To technika architektoniczna, a nie tylko wygoda

Podział na takie etapy — potężny sposób na organizację programu: każda funkcja rozwiązuje jedno zrozumiałe zadanie, każda ma jasny wejście i wyjście, a etapy można testować oddzielnie (§13.24).

Graf wywołań (call graph)

Kiedy funkcje wywołują inne funkcje, powstałą strukturę można przedstawić jako drzewo — **wykres wywołań**:



Graf wywołań — która funkcja wywołuje kogo; pokazuje strukturę całego programu

Do tego pomysłu wrócimy w §13.22, gdy przerobimy rzeczywiste wersje rozdziału 12 w podobna struktura.

Praktyka: rurociąg funkcji
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/13-21/index.html)

Refaktoryzacja projektów rozdziału 12

Weźmy trzy ukończone wersje poprzedniego rozdziału i przekłóńmy je z jednego długiego scenariusza w zestaw zrozumiałych funkcji, a zachowanie na zewnątrz pozostanie takie samo.

Refaktoryzacja: zmiana struktury, a nie zachowanie

Refaktoryzacja

Refaktoryzacja polega na poprawie wewnętrznej struktury programu bez celowego zmieniania jego działań zewnętrznych. Już widzieliśmy ten termin w rozdziale 12 (§12.9, Quiz) — tutaj stosujemy go do kilku projektów z rzędu.

Quiz

viktorina_do.py

```
# DO - Jeden długi scenariusz (Rozdział 12, §12.9)
score = 0
for q in questions:
    user_answer = input(q["question"] + " ").strip().lower()
    if user_answer == q["answer"]:
        score += 1
```

viktorina_posle.py

```
def ask_question(question):
    return input(question["question"] + " ").strip().lower()

def check_answer(user_answer, question):
    return user_answer == question["answer"]

def run_quiz(questions):
    score = 0
    for q in questions:
        user_answer = ask_question(q)
        if check_answer(user_answer, q):
            score += 1
    return score
```

Analizator tekstu

analizator_posle.py

```
def normalize_text(text):
    return text.lower().split()

def count_words(words):
    return len(words)

def count_unique(words):
    return len(set(words))
```

Notes

zapisnaya_knizhka_posle.py

```
def add_contact(contacts, name, phone):
    contacts[name] = phone

def find_contact(contacts, name):
    return contacts.get(name)

def remove_contact(contacts, name):
    del contacts[name]
```

Główny program staje się czytelnym opisem algorytmu

do_glavnaya.py

```
text = input(„Tekst: ”)
text = text.lower()
words = text.split()
count = len(words)
unique = set(words)
unique_count = len(unique)
counts = {}
for word in words:
    counts[word] = counts.get(word, 0) + 1
# ... Jeszcze 40 linijek ...
```

posle_glavnaya.py

```
text = input(„Tekst: ”)
clean = normalize_text(text)
stats = analyze_text(clean)
show_report(stats)
```

Główną zaletą refaktoryzacji jest czytelność na najwyższym poziomie

Właściwa wersja brzmi niemal jak zwykłe zdanie: „pobierz tekst, normalizuj, analizuj, pokaż raport.”

Challenge

Turtle też jest refaktoryzowany

Weź kod choinki lub mandali z rozdziału 12 i wydziel osobną funkcję dla jednego poziomu/promienia — dokładnie tak, jak studio wielokątów w §13.26 wydziela draw_polygon().

Praktyka: refaktoryzacja projektu z rozdziału 12

interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/13-22/index.html)

Debug Lab: typowe błędy funkcji

Zapomniany return return wewnątrz pętli w pierwszej iteracji, losowo zmutowany argument — dziesięć błędów, które powtarzają się wielokrotnie w rzeczywistym kodzie.

Dziesięć typowych błędów związanych z funkcjami — każdy z przykładem i wyjaśnieniem. Pierwsze dwa były już omówione osobno (§13.1, §13.13) — tutaj dla kompletności listy, zwięźle; pozostałe osiem — szczegółowo.

1 · Losowe wywołanie przed definicją

bug.py

```
greet()

def greet():
    print('Cześć')
```

W momencie połączenia greet() nie jest jeszcze zdefiniowany — Python wykonuje kod od góry do dołu i dzieje się NameError zanim łańcuch znaków def zdąży go wykonać.

2 · Niewidoczny def w def

bug.py

```
def outer():
    ...

    def helper():
        ...
```

Wcięcie definiuje zagnieżdżanie. Jeśli helper jest wcięte wewnątrz outer, nie będzie widoczne z zewnątrz jako niezależna funkcja, tylko przez outer.

3 · Funkcja zawsze zwraca None

bug.py

```
def calculate(x):
    print(x * 2)

result = calculate(5) + 1
```

calculate się drukuje, ale nie wraca – result staje się None i None + 1 powoduje TypeError.

4 · Nie wszystkie ścieżki funkcji zwracają wartość

bug.py

```
def sign(number):
    if number > 0:
        return 'positive'
    # a co jeśli number <= 0?
```

Dla liczb ujemnych i zerowych funkcja osiąga koniec bez return — i bezszelestnie zwraca None.

5 · return w pętli przy pierwszej iteracji

bug.py

```
def contains_even(numbers):
    for n in numbers:
        if n % 2 == 0:
            return True
        else:
            return False
```

return w rozgałęzieniu else jest już wyzwalane na pierwszym elemencie — nie wszystkie liczby są zaznaczone. Poprawnie: return True wewnątrz if, a return False — po całej pętli, z tym samym wcięciem co for.

6 · return pomieszane z break

bug.py

```
def find_first_even(numbers):
    for n in numbers:
        if n % 2 == 0:
            break   # a nie return n!
```

break po prostu zatrzymuje cykl — n trzeba jeszcze jawnie zwrócić oddzielnym wierszem po cyklu. return natychmiast wychodzi z CAŁEJ funkcji, break — tylko z najbliższego cyklu.

7 · Niespodziewana mutacja przekazanej listy

bug.py

```
def sorted_names(names):
    names.sort()
    return names
```

names.sort() mutuje lista kodu wywołującego w miejscu. Jeśli nie jest to przewidziane w planie, bezpieczniej return sorted(names) — nowy lista, oryginał pozostaje nietknięty.

8 · Zapomnieli skopiować przed zmianą

bug.py

```
def normalize_items(items):
    items[0] = items[0].strip()
    return items
```

Jeśli dane źródłowe trzeba zachować niezmienione, zacznij od items = items.copy() — ale pamiętaj o powierzchowności kopii (rozdział 11, §11.10) dla struktur zagnieżdżonych.

9 · Stan globalny utrudnia testowanie

bug.py

```
score = 0

def add_point():
    global score
    score += 1
```

Działa, ale trudniej jest przetestować taką funkcję osobno — jej wynik zależy od ukrytej zmiennej globalnej. def add_point(score): return score + 1 jest testowany na jednej linii, bez żadnej konfiguracji.

10 · Parametr zacienia nazwę inline

bug.py

```
def total(list):
    return sum(list)
```

list — nazwa wbudowanego typu. Zastąpienie jej parametrem działa, ale myli czytelnika i łamie dostęp do prawdziwego list() wewnątrz tej funkcji. Lepiej: def total(numbers):

Metoda debugowania funkcji

- Drukuje funkcja, czy zwraca? Sprawdź na papierze, zanim przeczytasz kod.
- Sprawdzono WSZYSTKIE ścieżki wykonywania funkcji, w tym niejawny „koniec bez return"?
- return w pętli — czy na pewno powinien zadziałać w tej iteracji?
- Czy funkcja celowo mutuje argument przekazany? Jeśli nie, skopiuj go.
- Czy nazwa parametru jest mylona z wbudowaną nazwą Python?

Praktyka: znajdujemy i poprawiamy błędy w funkcjach
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/13-23/index.html)

Funkcje testowania

Funkcję można testować oddzielnie od całego programu, bez udziału użytkownika i bez uruchamiania całej aplikacji.

Główna praktyczna zaleta funkcji

testiruemaya_funkciya.py

```
def rectangle_area(w, h):
    return w * h
```

proverki.py

```
rectangle_area(2, 3) == 6
rectangle_area(0, 5) == 0
rectangle_area(10, 1) == 10
```

Nie trzeba wprowadzać użytkowników ani całej aplikacji
Funkcję można przetestować w izolacji — wywołać bezpośrednio z określonymi wartościami i porównać z oczekiwanym wynikiem. O to chodzi **testowanie jednostkowe** kod.

Stół Testowy

Loguj	Czekam	Fakt	Ukończono?
`classi-fy_score(95)`	„doskonały”	„doskonały”	

Loguj	Czekam	Fakt	Ukończono?
`classify_score(60)`	„dobrze”	„dobrze”	
`classify_score(0)`	„poprawka”	„poprawka”	
`classify_score(100)`	granicy z góry	?	sprawdzone

Przypadki Krawędziowe — z rozdziału 9
Tabela testowa — bezpośrednia kontynuacja testów granicznych warunków z rozdziału 9: sprawdzamy nie tylko „zwykłe” wartości, ale też granice zakresów.

assert jest łatwym testem tego założenia

assert_primer.py

```
assert rectangle_area(2, 3) == 6
assert rectangle_area(0, 5) == 0
print(„Wszystkie testy przeszły”)
```

```
assert_padaet.py
```

```
assert rectangle_area(2, 3) == 7
# AssertionError
```

assert jest narzędziem deweloperskim, nie weryfikującym danych użytkowników

`assert` sprawdza założenie i podnosi `AssertionError` jeśli jest fałszywy, jest przydatny do samodzielnego testowania podczas rozwoju i debugowania. To NIE jest mechanizm weryfikacji danych użytkownika ani ochrony bezpieczeństwa – stosuje się standardowe warunki (Rozdział 9) do weryfikacji danych otrzymanych od użytkownika, tak jak już zrobiliśmy.

Workflow: przykłady → kontraktów → przegląd wdrożenia →

Ten sposób myślenia — bezpośredni zwiastun kompletnego testowania unit-, które pojawi się w przyszłych rozdziałach

Praktyka: Testowanie funkcji względem tabeli
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/13-24/index.html)

Mini-projekt – odrabianie lekcji z matematyki z Python

Funkcje generowania i sprawdzania mnożenia przykładów – ćwiczenie wszystkich technik z rozdziału, z czystą logiką walidacyjną oddzieloną od I/O.

ZASTOSOWANIE

`def` `return`

`*args (опционально)` `random`

`циклы`

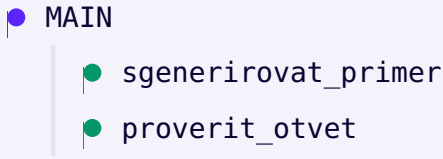
POZIOM

integracja

REZULTAT

Trenażer tekstowy

Zbieraj funkcje, argumenty i return w jednym użytecznym narzędziu: generatorze przykładów aby trenować tabliczkę mnożenia. Podzielmy problem na funkcje według znaczenia i nie zostawiajmy go w spokoju W częściach:



Graf wywołań: główny program opiera się na dwóch małych, niezależnie weryfikowanych funkcjach

domashnee_zadanie.py

```python
import random

def sgenerirovat_primer():
    a = random.randint(2, 9)
    b = random.randint(2, 9)
    return a, b, a * b

def proverit_otvet(pravilnyj_otvet, otvet_polzovatelya):
    return pravilnyj_otvet == otvet_polzovatelya

a, b, pravilnyj_otvet = sgenerirovat_primer()
otvet = int(input(f"Ile to będzie kosztować {a} x {b}? "))

if proverit_otvet(pravilnyj_otvet, otvet):
    print(„Zgadza się!”)
else:
    print(f"Poprawna odpowiedź jest poprawna:
{pravilnyj_otvet}")
```

Funkcja zwraca od razu trzy wartości

`return a, b, a * b` faktycznie zwraca jedną krotkę `(a, b, a * b)` — rozkładamy to na trzy zmienne jednocześnie, jak w §13.10 i rozdziale 11.

sgenerirovat_primer i proverit_otvet są testowane osobno

`proverit_otvet(56, 56)` oraz `proverit_otvet(56, 50)` można wywołać bezpośrednio i sprawdzić wynik — bez ani jednego `input()`. To właśnie idea z §13.24: czysta logika walidacji jest oddzielona od I/O.

★★ Zadanie samodzielne

Poprawny licznik odpowiedzi

Owiń generowanie przykładu w pętlę na 5 kolejnych pytań i policz, ile z nich rozwiązano poprawnie.

Praktyka: Generator przykładów matematycznych

interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/13-07/index.html)

Mini-projekt - Analizator tekstu, wersja 2 Projekt integracyjny

Ten sam parser tekstowy z rozdziału 12 — ale teraz jako czysty, testowalny pipeline funkcji.

ZASTOSOWANIE

`def` `return` `list` `set` `dict`

POZIOM

integracja

REZULTAT

raport tekstowy

Pełny refaktoring analizatora tekstu z rozdziału 12 (§12.6) w pipeline z czystych funkcji — bezpośrednie kontynuowanie idei §13.21.

- MAIN
  - normalize_text
  - split_words
  - word_frequency
  - build_summary

Każdy etap jest osobną, niezależnie testowaną funkcją

analizator_v2.py

```
def normalize_text(text):
    return text.lower()

def split_words(text):
    return text.split()

def word_frequency(words):
```

```
    counts = {}
    for word in words:
        counts[word] = counts.get(word, 0) + 1
    return counts

def build_summary(text):
    clean = normalize_text(text)
    words = split_words(clean)
    counts = word_frequency(words)
    return {
        "total_words": len(words),
        "unique_words": len(set(words)),
        "counts": counts,
    }

summary = build_summary("Python is great and python is fun")
print(summary)
```

build_summary jest potok, który wywołuje pozostałe trzy

Dokładnie wzorzec z §13.21: `build_summary` sam nic nie oblicza — przesyła dane wzdłuż łańcucha `normalize_text → split_words → word_frequency` i zbiera podsumowanie w słownik.

Praktyka: Parser tekstu jako potok funkcji

interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/13-25/index.html)

Mini-Projekty - Konwerter jednostkowy i narzędzia do zbierania

Dwa małe projekty składające się wyłącznie z czystych funkcji – wzorowe kandydaty do niezależnych testów.

ZASTOSOWANIE

`def` `return` `type hints` `assert`

POZIOM

★★ średni

REZULTAT

zbiór funkcji czystych

Konwerter jednostkowy

Mały projekt z całkowicie czystych funkcji — idealne kandydatury do testowania z §13.24:

konverter.py

```python
def celsius_to_fahrenheit(celsius: float) -> float:
    return celsius * 9 / 5 + 32

def fahrenheit_to_celsius(fahrenheit: float) -> float:
    return (fahrenheit - 32) * 5 / 9

def km_to_miles(km: float) -> float:
    return km * 0.621371

assert celsius_to_fahrenheit(0) == 32
assert celsius_to_fahrenheit(100) == 212
assert round(km_to_miles(10), 2) == 6.21
```

Ani jednej input(), ani jednej print() w samych funkcjach
Wszystkie trzy funkcje są czyste: wynik zależy całkowicie od argumentu, nie ma skutków ubocznych. Dlatego można je sprawdzać `assert` bez uruchomienia w ogóle pełnoprawnego programu.

Narzędzia do pracy z kolekcjami

Zestaw małych, skupionych funkcji na szczycie list i słowników z rozdziału 11:

kollekciya_utility.py

```python
def average(scores: list[float]) -> float:
    return sum(scores) / len(scores)

def count_above(scores: list[float], threshold: float) -> int:
    return len([s for s in scores if s > threshold])

def unique_words(text: str) -> set[str]:
    return set(text.lower().split())

def find_top_score(students: list[dict]) -> dict:
    return max(students, key=lambda s: s["score"])
```

find_top_score używa funkcji jako argumentu
`key=lambda s: s["score"]` to ta sama kombinacja `max()` / `sorted()` + lambda z §13.20 i §13.26, dopiero teraz na liście słowników (rozdział 11, §11.14).

★★ Zadanie samodzielne
Jeszcze jedno narzędzie
Write find_students_below(students threshold) to lista nazwiska uczniów z wynikiem niższym niż threshold. Sprawdź to na 2-3 przykładach w assert.

Praktyka: Konwerter jednostkowy
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji
Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/13-26/index.html)

Praktyka: Narzędzia do list i słowników
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji
Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/13-27/index.html)

Mini-projekt — Turtle Function Studio

Kształty z rozdziałów 6 i 10 stają się prawdziwą wielokrotnego użytku funkcją z wyraźnymi parametrami — i zamykamy cały rozdział funkcji.

ZASTOSOWANIE

`def` `параметры` `циклы` `Turtle`

POZIOM

★★★★ challenge

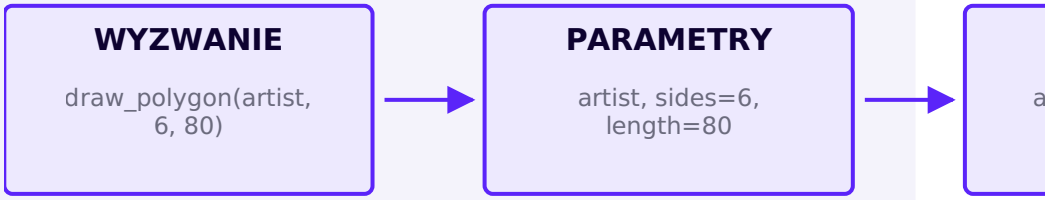
REZULTAT

Turtle graficznych

Co tworzymy

Projekt końcowy rozdziału: przekształcimy powtarzający się kod rysowania figur z rozdziałów 6 i 10 w prawdziwą, wielokrotnego użytku funkcję z parametrami.

Przepływ danych funkcji



Jeden wywołanie — parametry rozwijają się w obliczenie, pętlę i wynikowy rysunek

draw_polygon.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=300, height=300)
artist = turtle.Turtle()
artist.speed(0)
artist.pencolor("#5B24F9")
artist.pensize(3)

def draw_polygon(artist, sides, length):
    angle = 360 / sides
    for _ in range(sides):
        artist.forward(length)
        artist.right(angle)

draw_polygon(artist, 6, 80)

screen.exitonclick()
```

REZULTAT

Regularny sześciokąt rysowany przez funkcję draw_polygon

draw_polygon(artist, 6, 80) — prawidłowy sześciokąt

Dlaczego funkcja przyjmuje artist jawnie

figura_funkciya.py

```
def draw_polygon(artist, sides, length):
    angle = 360 / sides
    for _ in range(sides):
        artist.forward(length)
        artist.right(angle)

draw_polygon(artist, 4, 100)    # kwadrat
draw_polygon(artist, 3, 100)    # trójkąt, bez powtarzania
kodu!
draw_polygon(artist, 8, 60)     # ośmiokąt
```

Parametr artist jawny nie jest zbędną formalnością

Funkcja mogłaby polegać na zmiennej globalnej `artist`, jak w rozdziale 10. Ale parametr sprawia, że zależność jest WIDOCZNA (§13.15): od razu widać, że funkcja działa dokładnie z tą żółwikiem, którą jej przekazano — można ją ponownie wykorzystać z inną żółwikiem (na przykład podczas wyścigu Turtle z §12.6, gdzie jest kilka żółwi) bez żadnej zmiany kodu wewnątrz.

Funkcja wewnątrz pętli: to samo zachowanie, nowy parametr na każdym kroku

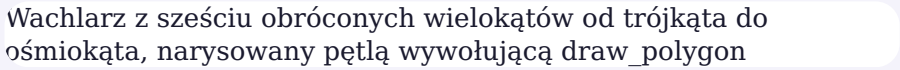
figury_v_cikle.py

```
cveta = ["#5B24F9", "#DB2777", "#059669"]
for i, sides in enumerate(range(3, 9)):
    artist.pencolor(cveta[i % 3])
    draw_polygon(artist, sides, 70)
    artist.right(15)
```

figury_v_cikle.py

```
import turtle

screen = turtle.Screen()
screen.setup(width=420, height=420)
artist = turtle.Turtle()
artist.speed(0)
artist.pensize(2)

def draw_polygon(artist, sides, length):
    angle = 360 / sides
    for _ in range(sides):
        artist.forward(length)
        artist.right(angle)

cveta = ["#5B24F9", "#DB2777", "#059669"]
for i, sides in enumerate(range(3, 9)):
    artist.pencolor(cveta[i % 3])
    draw_polygon(artist, sides, 70)
    artist.right(15)

screen.exitonclick()
```

REZULTAT



Pętla powoduje draw_polygon z nowym sides na każdym kroku — wachlarzem wielokątów

Pętla automatyzuje powtarzanie, funkcja automatyzuje zachowanie

Dokładnie idea §13.1: pętla decyduje ILE RAZY i z jakimi danymi wywołać `draw_polygon`; sama funkcja decyduje, JAK narysować wielokąt na podstawie tych danych. Żadne z tych dwóch narzędzi nie zastępuje drugiego.

Challenge

Funkcja z pozycją

Dodaj parametry x, y (z domyślnymi wartościami 0, 0) do funkcji, aby móc określić, od czego zacząć rysować kształt.

Praktyka: Turtle Function Studio

Moduł turtle otwiera natywne okno Python – uruchamiane lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/13-08/index.html)

Podsumowanie rozdziału

Zestaw narzędzi do projektowania funkcji

Ostatni zestaw narzędzi rozdziału 13

Potrzebujesz ponownie użyć tego samego zestawu akcji?	→ `wyróżnij funkcję`
Czy funkcja wymaga wejścia?	→ `parametry`
Trzeba zwrócić wynik do przyszłego użytku?	→ `return`
Czy liczba kłótni może być nieznana?	→ `*args / **kwargs`

Czy niektóre argumenty powinny być opcjonalne?	→ **domyślne ustawienia (niezmienialne!)**
Niektóre argumenty powinny być przekazywane tylko po nazwie?	→ **keyword-only, po ***
Czy funkcja powinna zachowywać się przewidywalnie i łatwo ją testować?	→ **czysta funkcja, bez efektów ubocznych**
Czy trzeba przekazać samą funkcję jako wartość?	→ **funkcja jest obiektem pierwszej klasy**
Potrzebna jest malutka zasada dla sorted()/max()?	→ **lambda**

Pełna lista kontrolna projektu w §13.18

Czego nauczyliśmy się w tym rozdziale

- Funkcja to nazwany, wielokrotnego użytku element zachowania: odbiera dane (parametry), pracuje z nimi, zwraca wynik (`return`).
- Wywołanie funkcji przekazuje sterowanie do jej ciała i koniecznie wraca do miejsca wywołania — funkcja nie wykonuje się „gdzieś obok”
- Parametr to nazwa w definicji, argument to wartość w wywołaniu. Przypisanie parametru i mutacja przekazanego obiektu to dwie różne akcje o różnych konsekwencjach na zewnątrz.
- Domyślna wartość zmienna jest tworzona RAZ przy definiowaniu funkcji — użyj `None` jako sygnał zamiast `[]` lub `{}` .
- `*args` oraz `**kwargs` zbierają zbędne argumenty w krotkę i słownik odpowiednio; to samo `*` / `**` rozpakowuje kolekcję z powrotem w momencie rozmowy.
- Funkcja bez `return` wciąż wraca `None` nie jest „niczym”
- Python szuka nazwy zgodnie z zasadą LEGB: Local → Enclosing → Global → Builtins.
- Funkcje wywołujące funkcje tworzą stos wywołań — a traceback z rozdziału 3 pokazuje ten stos w momencie wystąpienia błędu.
- Funkcje — obiekty pierwszej klasy: można je zapisać pod inną nazwą, umieścić w kolekcjach, przekazać innym funkcjom (`sorted(key=...)`).
- `lambda` — mała anonimowa funkcja dla jednego wyrażenia, nie więcej „nowoczesna" zamiana `def` .
- Refaktoryzacja dużego skryptu do zestawu funkcji sprawia, że główny program jest czytelnym opisem algorytmu, a każda część jest niezależnie testowalna.

co dalej

Nauczyliśmy się grupować ZACHOWANIE w funkcjach. Ale w wielu naszych projektach dane i zachowanie, które z nimi współdziała, naturalnie idą w parze — kontakt z własnym telefonem, uczeń ze swoją oceną, żółw ze swoim kolorem i pozycją. W następnym rozdziale — **„Tworzenie rzeczywistych obiektów”** – nauczymy się, jak łączyć dane i funkcje, które współtworzą jedną nazwaną strukturę.

ROZDZIAŁ 14 · STR. 797

Tworzymy obiekty rzeczywistego świata

Klasy, obiekty, atrybuty i metody to podstawy programowania obiektowego. Od znanych obiektów (żółw, łańcuch znaków, lista) po klasy własnościowe: stan i zachowanie, self i __init__, enkapsulację i property, składanie i dziedziczenie, super(), polimorfizm i duck typing, metody specjalne dataclasses — aż po pytanie ważniejsze niż jakakolwiek składnia: kiedy klasa w ogóle powinna pojawić się w programie.

Czym jest programowanie obiektowe?

Już korzystałeś z obiektów przez wiele rozdziałów — po prostu jeszcze ich tak nie nazywałeś.

Obiekt — to, co już jest znane, ale z nową nazwą

Przyjrzyj się kodowi z poprzednich rozdziałów: `artist.forward(100)`, `"Python".upper()`, `[1,2,3].append(4)` — we wszystkich trzech mamy jakiś rodzaj **cel** (żółw, łańcuch znaków, lista) oraz Komendę, którą wywołujemy z nim przez punkt. Przez cały ten czas już pracowałeś z przedmiotami "Ta sekcja daje znanej idei dokładną nazwę.

Każdy obiekt w Python ma dwie strony:

Strona obiektowa	Przykład w artist (żółwie)
Status są dane, które obiekt obecnie przechowuje	aktualnych współrzędnych, koloru, kierunku
Zachowanie — działania, które obiekt może wykonać	`forward()`, `right()`, `color()`

Programowanie obiektowe (OOP) to sposób organizacji kodu wokół Takie obiekty: dane i działania z nimi związane żyją **razem**, oraz nie są rozproszone między pojedynczymi zmiennymi i funkcjami.

Typ obiektu: `type()`

Każdy obiekt w Python ma typ, który może być rozpoznawany przez wbudowaną funkcję `type()` już znasz:

tip_obekta.py

```
artist = __import__("turtle").Turtle()
print(type(artist))     # <class 'turtle.Turtle'>
print(type("Python"))   # <class 'str'>
print(type([1, 2, 3]))  # <class 'list'>
```

Word `class` w każdej odpowiedzi — nie przypadek. Typ obiektu — to właśnie on **klasa**.

Udowodnijmy to!

Już korzystałeś z obiektów od samego rozdziału 6, nawet nie nazywając ich tak:

vy_uzhe_polzovalis_obektami.py

```
import turtle

artist = turtle.Turtle()   #artist jest obiektem klasowym
Turtle
artist.forward(100)         # forward() jest działaniem
(metodą) tego obiektu
artist.color("red")         # obiekt ma również własny stan –
na przykład kolor
```

`turtle.Turtle` jest **klasa**: rysunek, szablon, opisujące, jakie są żółwie i co potrafią. Każde wyzwanie `turtle.Turtle()` tworzy nową, niezależną **cel** (nazywane także „instancją klasy”

Klasa opisuje obiekty PRZYSZŁE, a nie jeden konkretny

`turtle.Turtle` sam nie ma pozycji na ekranie — tylko KAŻDY utworzony obiekt ma swoją pozycję `turtle.Turtle()` osobno. Klasa to opis tego, czym będą obiekty, a nie sam obiekt.

Praktyka: znajdujemy obiekty w już znanym kodzie

interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/14-01/index.html)

Klasy

Pisanie własnej pierwszej klasy — szablon, według którego Python będzie tworzyć obiekty.

Klasy

Zdefiniujmy własną klasę — rysunek obiektu „Dog”

klass_sobaka.py

```
class Sobaka:
    def __init__(self, klichka, vozrast):
        self.klichka = klichka
        self.vozrast = vozrast

rex = Sobaka(„Rex”, 3)
print(rex.klichka, rex.vozrast)
```

`__init__` jest specjalną metodą, która automatycznie jest wykonywany podczas tworzenia obiektu (`Sobaka("Рекс", 3)`) i dostosowuje w stanie początkowym. `self` — odwołanie do samego tworzonego obiektu; Python przekazuje je automatycznie, a pierwszym parametrem `__init__` (lub jakakolwiek inna metoda) powinna zawsze być obecna. Szczegółowo, co się tutaj dzieje „pod Hood”

Klasa vs obiekt — analogia do foremki do ciastek
Class jest jak forma do ciasteczek: ta sama forma może „pokroić”

Gdzie analogia się psuje
Forma ciasteczek to narzędzie pasywne: sama w sobie nic nie robi i nie przechowuje stanów. Klasa w Python nie jest biernym szablonem, lecz obiektem samym w sobie: możesz ją o to prosić `type(rex)` atrybuty mogą być przypisane do niego samemu, może być przekazywany zmiennej. Użyj tej analogii, by zapamiętać „jeden rysunek → wiele kopii”

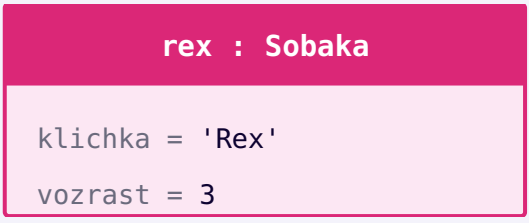
Klasa Sobaka — opisuje PRZYSZŁE atrybuty, ale jeszcze nie zawiera żadnej wartości

Obiekty ze swoimi wartościami

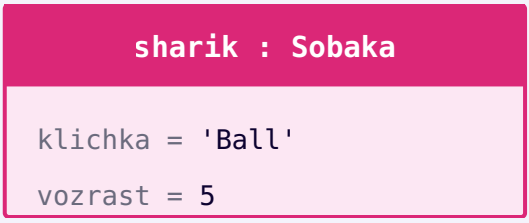
Każda cecha przechowuje własne wartości atrybutów, niezależne od innych cech tej samej klasy:

raznye_obekty.py

```
rex = Sobaka(„Rex”, 3)
sharik = Sobaka(„Ball”, 5)

print(rex.klichka, rex.vozrast)      # Rex 3
print(sharik.klichka, sharik.vozrast)  # Piłeczka 5 —
niezależnie od rex
```



Obiekt rex — konkretne wartości



Obiekt sharik — własne wartości, ta sama klasa

Jedna klasa — tyle obiektów, ile chcesz, a każdy ma swój własny, niezależny stan. Dokładnie tak To odróżnia obiekt od prostego słownika o podobnych kluczach: *należy do* swojemu klasie i uzyskuje przez nią dostęp do wspólnych metod (sekcja 14.3).

Praktyka: Tworzenie własnych klas
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/14-02/index.html)

Zarządzanie obiektami

Atrybuty można zmieniać po utworzeniu obiektu — a metody nadają obiektowi własne działania.

Zarządzanie obiektami

Wartości atrybutów można zmieniać już po utworzeniu obiektu — tak samo, jak zmienia się wartość zwykłej zmiennej:

menyaem_atributy.py

```
rex = Sobaka(„Rex”, 3)
rex.vozrast = 4   # Rex miał urodziny
print(rex.vozrast)
```

Obiekty wykonują działania

Oprócz danych (atrybutów), klasa może mieć własne działania — **metody**: funkcje normalne zdefiniowane w klasie, które mają dostęp do `self` a przez to także atrybuty obiektu:

metody.py

```
class Sobaka:
    def __init__(self, klichka, vozrast):
        self.klichka = klichka
        self.vozrast = vozrast

    def layat(self):
        print(f"{self.klichka} powiedział: „Hau-hau!”")

    def prazdnovat_den_rozhdeniya(self):
        self.vozrast += 1
        print(f"Teraz {self.klichka} się odwrócił
{self.vozrast}!")

rex = Sobaka(„Rex”, 3)
rex.layat()
rex.prazdnovat_den_rozhdeniya()
```

Znana notacja

`rex.layat()` zorganizowane dokładnie tak samo, jak `artist.forward(100)` lub `"текст".upper()` — metody obiektów, których już od dawna używasz, działają dokładnie tak samo jak metoda `layat()`, że właśnie sam napisałeś.

Klasa Sobaka — teraz z metodami

self nie jest słowem kluczowym

`self` — zwyczajowa nazwa parametru wybrana konwencją, a nie słowo zastrzeżone Python (w przeciwieństwie do `if` lub `class`). Technicznie rzecz biorąc, metoda będzie działać z nazwą `this` lub `obj` — ale użyj `self` zawsze: tak nazywa go cały kod Python, a każda inna nazwa może zdezorientować innych czytających twój kod. Szczegóły miejsca `self` pobiera się przy wywołaniu, omówimy w rozdziale 14.5.

Praktyka: metody i zmiana atrybutów
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/14-03/index.html)

Turtle Rasa Obiektów

Przerabianie rasy z rozdziału 12 z własną klasą i zobaczenie kompozycji w akcji po raz pierwszy.

Powrót do wyścigu żółwi z rozdziału 12, tym razem każdy uczestnik otula go własnym własna klasa, nie tylko `turtle.Turtle` bezpośrednio.

gonka_s_klassom.py

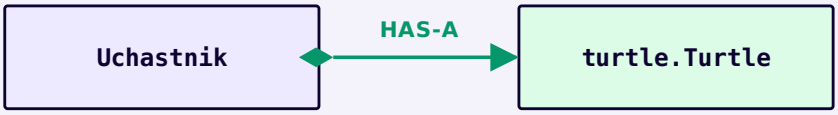
```
import turtle
import random

random.seed(5)
screen = turtle.Screen()
screen.setup(500, 400)

class Uchastnik:
    def __init__(self, cvet, startovyj_y):
        self.t = turtle.Turtle()
        self.t.shape("turtle")
        self.t.color(cvet)
        self.t.penup()
        self.t.goto(-200, startovyj_y)
        self.cvet = cvet

    def sdelat_shag(self):
        self.t.forward(random.randint(1, 10))

    def finishiroval(self, finish_line):
        return self.t.xcor() >= finish_line

cveta = ["red", "blue", "green", "orange"]
uchastniki = [Uchastnik(cvet, i * 40 - 60) for i, cvet in
enumerate(cveta)]

pobeditel = None
while pobeditel is None:
    for u in uchastniki:
        u.sdelat_shag()
        if u.finishiroval(200):
            pobeditel = u.cvet
            break

screen.exitonclick()
```

REZULTAT

Cztery żółwie różnych kolorów na mecie wyścigu, każdy będący osobnym obiektem klasy Uchastnik

Cztery obiekty Uchastnik — niezależny stan, wspólna klasa

Po co owijac Turtle w własną klasę?
W rozdziale 12 logika wyścigu (ruch, kontrola mety) została rozproszona po całym głównym kodzie. Teraz każdy uczestnik jest niezależnym obiektem `Uchastnik` kto wie, jak wykonać ruch i sprawdzić, czy skończył. Proszę zauważyć: `Uchastnik` nie jest żółwiem — przechowuje żółwia w atrybucie `self.t`. To nazywa się **skład** — formalnie omówimy ją w rozdziale 14.13.



Uchastnik PRZECHOWUJE obiekt turtle.Turtle — a nie jest nim

Challenge
Punktacja
Dodaj atrybut ochki do Uchastnik klasy oraz metodę zwiększającą wynik nabrat_ochki(n), aby przyznawać punkty zwycięzcy na koniec wyścigu.

Praktyka: wyścig Turtle z obiektami
Moduł turtle otwiera natywne okno Python – uruchamiane lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/14-04/index.html)

self i metody łączenia

rex.layat() i Sobaka.layat(rex) to to samo wyzwanie. Sprawdźmy, skąd self naprawdę się bierze.

Dwa równoważne nagrania tego samego wezwania

`rex.layat()` nie jest jedynym sposobem nazywania tej metody. To samo wywołanie można zapisać przez klasę, przekazując obiekt ręcznie pierwszym argumentem:

Ta sama akcja – dwa sposoby nagrywania

Przez Obiekt (Normalny wpis)	Przez zajęcia (co naprawdę się dzieje)
`rex.layat()`	`Sobaka.layat(rex)`

Co używać dzisiaj: Na piśmie zawsze używaj notacji przez dopełnienie — `rex.layat()`. Druga forma jest pokazana tylko po to, aby było widać, skąd się bierze `self`: To właśnie jest `rex` automatycznie przeszedł jako pierwszy argument.

co się dzieje, gdy dzwonisz `rex.layat()`

Wiązanie metody: obiekt po lewej stronie kropki staje się self w środku metody

self — znów nie jest słowem kluczowym

Jak już wspomniano w rozdziale 14.3: `self` — to po prostu nazwa pierwszego parametru metody, wybrana konwencją. Python podstawia w niego obiekt po lewej od kropki NIEZALEŻNIE od tego, jak go nazwiesz — ale nazywaj go `self` zawsze, to jedyna nazwa, której oczekuje każdy, kto czyta kod w Python.

DEBUG LAB 1: ZAPOMNIAŁEM SELF W DEFINICJI METODY

bez_self.py

```
class Sobaka:
    def __init__(self, klichka):
        self.klichka = klichka

    def layat():          # zapomniano self jako
pierwszy parametr
        print(„Hau-hau!”)

rex = Sobaka(„Rex”)
rex.layat()
```

```
Traceback (most recent call last):
  File "bez_self.py", line 9, in
    rex.layat()
TypeError: Sobaka.layat() takes 0 positional arguments but 1
```

Co widać na ekranie

Python i tak próbował podstawić `rex` pierwszą kłótnią podczas dzwonienia `rex.layat()` — wiązanie metody następuje ZAWSZE, a nie tylko wtedy, gdy się tego spodziewasz. Jeśli w definicji metody nie było parametru, żeby go przyjąć, — Python informuje, że przekazano o jeden argument więcej, niż metoda jest w stanie przyjąć.

POPRAWIONY KOD

s_self.py

```
class Sobaka:
    def __init__(self, klichka):
        self.klichka = klichka

    def layat(self):
        print(f"{self.klichka}: Hau-hau!")

rex = Sobaka(„Rex”)
rex.layat()
```

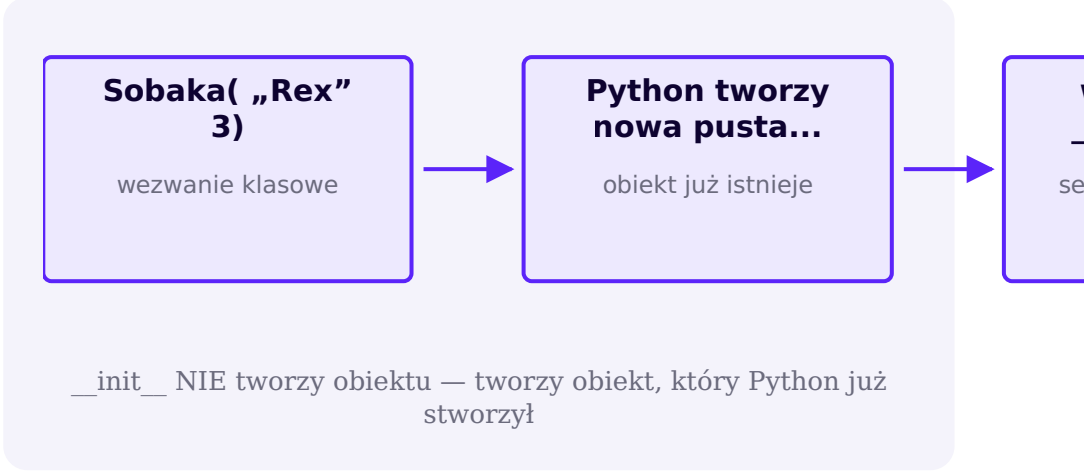
Praktyka: self i metody sprzężenia
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/14-05/index.html)

__init__ i tworzenie obiektu

__init__ nie tworzy obiektu — konfiguruje obiekt, który już istnieje.

Co tak naprawdę się dzieje, gdy `Sobaka("Pekc", 3)`



__init__ NIE tworzy obiektu — tworzy obiekt, który Python już stworzył

Częste nieporozumienie: „__init__ tworzy obiekt”
To nieprawda. W chwili wezwania `__init__` obiekt już istnieje, w przeciwnym razie nie Python można go przekazać pierwszym argumentem (`self`)! Rola `__init__` nie polega na stworzeniu obiektu, ale **dostosować jego początkowy stan**: nazwa byłaby bardziej uczciwa "inicjalizator" niż "konstruktor" – tak jest nazywana w dokumentacji Python.

Wartości domyślne w __init__

`__init__` jest funkcją regularną (rozdział 13 dotyczy w całości): Parametry mogą mieć wartości domyślne:

init_s_umolchaniem.py

```
class Sobaka:
    def __init__(self, klichka, vozrast=1):
        self.klichka = klichka
        self.vozrast = vozrast

shchenok = Sobaka(„Bim”)              # vozrast = 1 domyślnie
rex = Sobaka(„Rex”, vozrast=3)       # wyraźnie określone
```

__init__ może wywoływać inne metody

init_vyzyvaet_metod.py

```
class Sobaka:
    def __init__(self, klichka, vozrast=1):
        self.klichka = klichka
        self.vozrast = vozrast
        self.privet()
# może być już wywołana wewnątrz __init__

    def privet(self):
        print(f"Nowy pies na scenie: {self.klichka}!")

rex = Sobaka(„Rex”)   # drukuje powitanie od razu po utworzeniu
```

DEBUG LAB 2: SELF.X = X ZAPISALIŚMY PO PROSTU JAKO X = X

poteryannyj_atribut.py

```
class Sobaka:
    def __init__(self, klichka, vozrast):
        klichka = klichka      # to po prostu lokalny
zmienna!
        self.vozrast = vozrast

rex = Sobaka(„Rex”, 3)
print(rex.klichka)
```

```
Traceback (most recent call last):
  File "poteryannyj_atribut.py", line 7, in
    print(rex.klichka)
AttributeError: 'Sobaka' object has no attribute 'klichka'
```

Co widać na ekranie

String `klichka = klichka` bez `self.` po lewej tworzy regularną zmienną LOKALNĄ wewnątrz `__init__` znika, gdy tylko `__init__` końców — nigdy nie była przechowywana na obiekcie. Aby zachować wartość obiektu NA ZAWSZE, po lewej stronie `=` musi być obowiązkowo `self.имя_атрибута`.

POPRAWIONY KOD

atribut_ispravlen.py

```
class Sobaka:
    def __init__(self, klichka, vozrast):
        self.klichka = klichka
        self.vozrast = vozrast

rex = Sobaka(„Rex”, 3)
print(rex.klichka)
```

Praktyka: __init__ i tworzenie obiektu
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/14-06/index.html)

Atrybuty instancji i klas

Jedna cecha może istnieć w obiekcie, a inna w samej klasie, która jest wspólna dla wszystkich. To nie to samo.

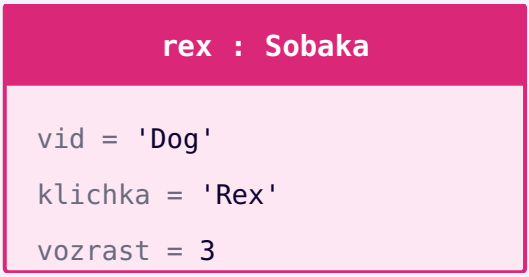
Dwa miejsca, gdzie atrybut może występować

Do tej pory wszystkie atrybuty były tworzone wewnątrz `__init__` przez `self.имя = ...` jest **atrybuty instancji**, u każdy obiekt ma swój własny. Ale atrybut może być również zadeklarowany bezpośrednio w ciele klasy, bez `self` — to tak jest **atrybut klasy**, jeden za wszystkich Obiekty jednocześnie:

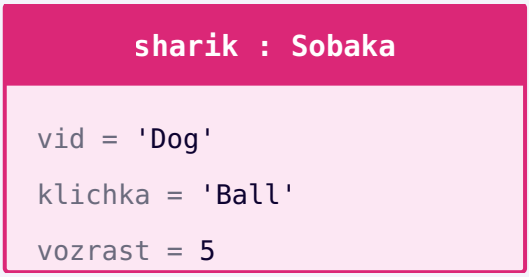
atribut_klassa.py

```
class Sobaka:
    vid = „Dog”          # atrybut KLASY – jeden za wszystkich

    def __init__(self, klichka, vozrast):
        self.klichka = klichka
# Atrybut INSTANCJI jest inny dla każdego
        self.vozrast = vozrast

rex = Sobaka(„Rex”, 3)
sharik = Sobaka(„Ball”, 5)
print(rex.vid, sharik.vid)   # Pies Pies to częsta cecha
```

Atrybut klasy jest deklarowany w treści klasy; Atrybuty instancji znajdują się w __init__



rex — vid odczytywane z klasy



sharik to ta sama vid, ale nie jest to jego własna kopia

Powszechne nieporozumienie: „atrybuty klasowe są kopiowane do każdego obiektu”

To nieprawda. `vid` przechowywane RAZ, w samej klasie `Sobaka`. Kiedy piszesz `rex.vid`, Python pierwsze spojrzenia `vid` wśród atrybutów INSTANCJI `rex` – nie znajduje jej – i dopiero potem patrzy na klasę `Sobaka`, gdzie znajduje wspólną wartość. Żaden bajt `vid` nie jest kopiowany do `rex`.

Jeśli ustawisz atrybut o tej samej nazwie na VIA OBJECT, Python utworzy nowy Atrybut INSTANCE, który teraz „okultyfikuje”

zatenenie.py

```
rex.vid = „Kundel”    # tworzy vid na rex bez dotykania klasy
print(rex.vid, sharik.vid)   # Kundel — sharik niezmieniony
```

Praktyka: Atrybuty instancji i klas
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/14-07/index.html)

Powszechna pułapka atrybutów zmiennych

One lista „for two”

Kiedy „wspólny atrybut” zamienia się w błąd

Atrybut klasy o NIEZMIENNEJ wartości (liczba, łańcuch znaków) jest bezpieczny – przypisanie przez obiekt po prostu tworzy nowy atrybut instancji (sekcja 14.7). Ale atrybut klasy z **ZMIENNE** znaczenie — lista lub słownik — jest ułożone inaczej: jeśli nie jest PRZYDZIEL i ZMIANA miejsca (`.append()` , `[key] = ...`), to wpłynie na obiekt klasy, którego używają WSZYSTKIE egzemplarze naraz.

DEBUG LAB 3: WSPÓLNY ZMIENNY LISTA JAKO ATRYBUT KLASY

obshchaya_korzina.py

```
class Korzina:
    tovary = []          # atrybut KLASY — jeden lista
dla wszystkich koszyków!

    def dobavit(self, tovar):
        self.tovary.append(tovar)   #. append() ZMIENIA
lista NA MIEJSCU

k1 = Korzina()
k2 = Korzina()
k1.dobavit(„Chleb”)
print(k2.tovary)
```

```
>>> print(k2.tovary)
['Хлеб']
```

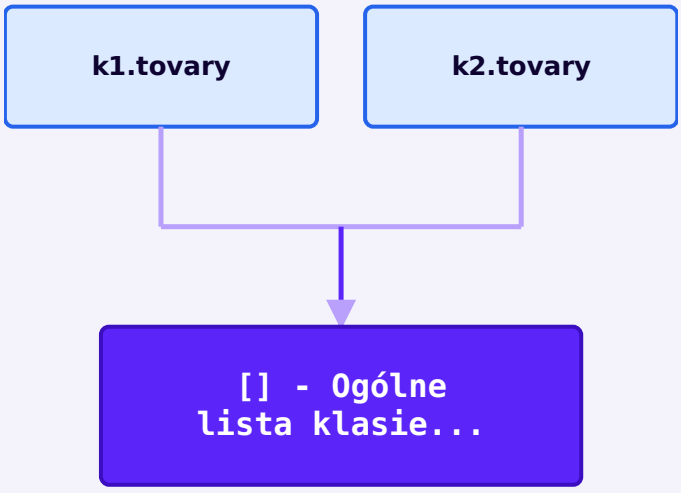
Co widać na ekranie

`k2` nigdy nie dzwonił `dobavit()`, ale „Chleb” jest już w jego koszyku! Powód: `self.tovary.append(...)` nie tworzy nowego lista — MODYFIKUJE istniejący, a ten lista jest jedynym, który odziedziczony są przez oba obiekty z `Korzina`.

POPRAWIONY KOD

svoya_korzina.py

```
class Korzina:
    def __init__(self):
        self.tovary = []    atrybut # INSTANCJA — każdy
kartridż ma własny lista

    def dobavit(self, tovar):
        self.tovary.append(tovar)

k1 = Korzina()
k2 = Korzina()
k1.dobavit(„Chleb”)
print(k2.tovary)   # [] - pusty, jak powinno być
```



Bez self.tovary = [] w __init__ oba odniesienia prowadzą do TEJ samej listy

Zasada: Wartości modyfikowalne są tylko w __init__

Jeśli atrybut potrzebuje wartości `list`, `dict` lub `set` – stwórz to w środku `__init__` przez `self.имя = []`, nie w treści klasy. To ta sama zasada co „pułapka zmiennej wartości domyślnej”

Praktyka: znajdujemy wspólny zmienny atrybut
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/14-08/index.html)

Mini-projekt: Player

Model Gracza z punktami zdrowia i wynikiem to pierwsza niezależna klasa rozdziału.

Mini-projekt: Player

Zbierzmy wszystko z sekcji 14.1-14.8 w jedną prawdziwą klasę, Model Gracza z Punktami Zdrowie i liczenie:

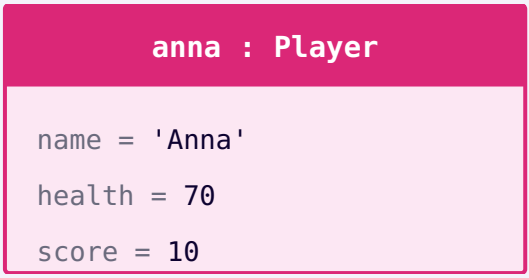
player.py

```
class Player:
    def __init__(self, name, health=100):
        self.name = name
        self.health = health
        self.score = 0

    def take_damage(self, amount):
        self.health -= amount
        if self.health < 0:
            self.health = 0

    def heal(self, amount):
        self.health += amount
        if self.health > 100:
            self.health = 100

    def add_score(self, points):
        self.score += points
```

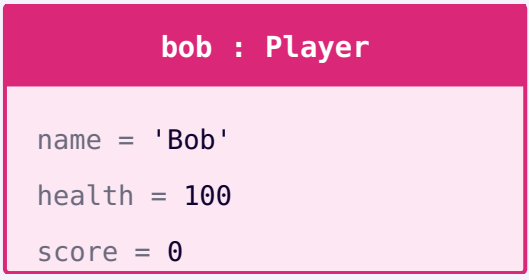
Klasa Player

dva_igroka.py

```
anna = Player("Anna")
bob = Player("Bob")
anna.take_damage(30)
anna.add_score(10)

print(anna.health, anna.score)   # 70 10
print(bob.health, bob.score)     # 100 0 — bob nie ucierpiał
```



anna — Twój stan



bob - Nie wpływają na działania anna

DEBUG LAB 4: == PORÓWNUJE NIE WARTOŚCI, LECZ SAME OBIEKTY

sravnenie_igrokov.py

```
a1 = Player("Anna")
a2 = Player("Anna")
# to samo imię, zdrowie i wynik co a1

print(a1 == a2)
```

```
>>> print(a1 == a2)
False
```

Co widać na ekranie

Chociaż `a1` oraz `a2` tych samych wartości WSZYSTKICH atrybutów, `==` domyślnie porównuje obiekty przez **tożsamość** — to dwa RÓŻNE obiekty w pamięci, dlatego wynik `False` nawet jeśli zawartość całkowicie się zgadza. To `==` porównywanych wartościach klasa wymaga specjalnej metody `__eq__` — do tego przejdziemy w sekcji 14.21.

POPRAWIONY KOD

sravnenie_cherez_pole.py

```
# Na razie porównujemy potrzebne pola ręcznie:
print(a1.name == a2.name and a1.health == a2.health)
```

★★ Zadanie samodzielne

is_alive

Dodaj metodę is_alive(self) do klasy Player, która zwraca True, jeśli health jest większa niż 0, a False w przeciwnym razie.

Praktyka: Klasa Player
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/14-09/index.html)

Enkapsulacja

Atrybuty są dostępne bezpośrednio — i to może zepsuć stan obiektu. Dowiadujemy się, jak tego uniknąć.

Problem: Atrybuty są dostępne bezpośrednio

Nic nie stoi na przeszkodzie w zmianie atrybutu obiektu, omijając wszystkie metody — bezpośrednio i zewnętrznie Klasa:

DEBUG LAB 5: BEZPOŚREDNIE PRZYPISANIE PSUJE STAN OBIEKTU

slomannoe_zdorove.py

```
anna = Player("Anna")
anna.health = -50    # bezpośrednio, omijając take_damage()
print(anna.health)
```

```
>>> print(anna.health)
-50
```

Co widać na ekranie

Metoda `take_damage()` niezgrabnie nie daje `health` zejść poniżej zera — ale NIC nie zmusza cię do stosowania tej konkretnej metody. Przydział `anna.health = -50` omija całkowicie logikę obronną bezpośrednio, a obiekt trafia w stan, który sama klasa uważała za niemożliwy.

POPRAWIONY KOD

cherez_metod.py

```
anna.take_damage(150)
#health poprawnie zatrzymuje się na 0
print(anna.health)
```

Enkapsulacja jest idea przechowywania stanu obiektu „wewnątrz”

Konwencja nazewnictwa: _internal i __name

Nagrania	Co oznacza według konwencji
`name`	atrybut publiczny – używaj go swobodnie poza
`_name`	„wewnętrzny”
`__name`	obejmuje podstawienie nazw (name mangling) – dostęp z zewnątrz jest trudny, ale nie znika

dvojnoe_podcherkivanie.py

```
class Konto:
    def __init__(self, balans):
        self.__balans = balans

schet = Konto(100)
print(schet._Konto__balans)   # 100 — dostęp JEST, po prostu nazwa została zmieniona
```

Częste nieporozumienie: «__private naprawdę prywatny»
To nieprawda – na poziomie języka w Python nie ma prawdziwej prywatności. `__balans` automatycznie zmienia nazwę na `_Konto__balans` (name mangling) – chroni to głównie przed PRZYPADKOWYM dziedziczeniem imion, a nie przed celowym uzyskaniem dostępu z zewnątrz. Enkapsulacja w Python to umowa i dyscyplina dowództwa, a nie zakaz ze strony tłumacza.

Praktyka: enkapsulacja
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji
Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/14-10/index.html)

property: obliczane atrybuty

@property pozwala wywołać metodę tak, jakby była regularnym atrybutem, z kontrolą przy przypisaniu i bez zmiany składni z zewnątrz.

property: write-on-the-job, bez zmiany składni odczytu

Dekorator `@property` pozwala wywoływać metodę TAK SAMO, jak zwykły atrybut — bez nawiasów:

property_getter.py

```
class Krug:
    def __init__(self, radius):
        self._radius = radius

    @property
    def ploshchad(self):
        return 3.14159 * self._radius ** 2

krug = Krug(10)
print(krug.ploshchad)   #314.159 — bez nawiasów, choć to metoda!
```

`ploshchad` nigdzie się nie przechowuje — jest obliczana na nowo przy każdorazowym wywołaniu. Na zewnątrz nie da się odróżnić tego od zwykłego atrybutu.

@x.setter: Sprawdzenie przydziału

property_setter.py

```
class Krug:
    def __init__(self, radius):
        self.radius = radius     # przechodzi przez setter
poniżej

    @property
    def radius(self):
        return self._radius

    @radius.setter
    def radius(self, value):
        if value <= 0:
            raise ValueError(„promień musi być dodatni”)
        self._radius = value

krug = Krug(10)
krug.radius = 5     # jest testowane
krug.radius = -1    #ValueError - Zadanie odrzucone
```

Główna korzyść property: kod zewnętrzny pozostaje niezmieniony

Jeśli dziś `radius` jest stałą cechą, a jutro musisz ją sprawdzić, możesz przerobić na property i cały kod, który napisał `krug.radius = 5`, nadal będzie działać bez żadnej poprawki. Dlatego w Python zwykle zaczyna się od zwykłych atrybutów i dodaje property dopiero wtedy, gdy sprawdzenie rzeczywiście jest potrzebne — a nie owija w property wszystkiego od samego początku.

DEBUG LAB 6: ZAPOMNIAŁEM O @RADIUS.SETTER DEKORATORZE

bez_setter.py

```
class Krug:
    def __init__(self, radius):
        self._radius = radius

    @property
    def radius(self):
        return self._radius

    def radius(self, value):    # zapomniałem
@radius.setter z góry!
        self._radius = value

krug = Krug(10)
krug.radius = 5
```

```
>>> krug.radius = 5
>>> print(krug.radius)
5
```

Co widać na ekranie

Nie będzie pomyłki – a to jest po prostu najbardziej podstępna rzecz. Bez dekoratora `@radius.setter` druga funkcja `radius` jest osobna, regularna metoda, która po prostu ZASTĄPIŁA property z góry (ta sama nazwa w grupie!). Po tym, `Krug.radius` nie jest już property, lecz funkcją zwykłą, oraz `krug.radius = 5` po prostu tworzy atrybut INSTANCE `radius`, cicho zasłaniając zarówno metodę, jak i cały czek z §14.11 — numer jest wydrukowany `5`, a nie wynik getter'a.

POPRAWIONY KOD

s_setter.py

```
    @radius.setter
    def radius(self, value):
        self._radius = value
```

Praktyka: property
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/14-11/index.html)

Mini-projekt: Rectangle

Zweryfikowane wymiary i obliczony obszar obwodu są property w prawdziwej klasie.

Mini-projekt: Rectangle

Klasa Prostokąta z potwierdzonymi wymiarami i obliczoną powierzchnią/obwodem:

rectangle.py

```
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height

    @property
    def width(self):
        return self._width

    @width.setter
    def width(self, value):
        if value <= 0:
            raise ValueError(„szerokość musi być dodatnia”)
        self._width = value

    @property
    def height(self):
        return self._height

    @height.setter
    def height(self, value):
        if value <= 0:
            raise ValueError(„wysokość musi być dodatnia”)
        self._height = value

    @property
    def area(self):
```

```
        return self.width * self.height

    @property
    def perimeter(self):
        return 2 * (self.width + self.height)
```

Rectangle — wszystkie cztery dostępne jako atrybuty, ale dwa z nich sprawdzają wartość, dwa są obliczane

ispolzovanie.py

```
r = Rectangle(10, 4)
print(r.area, r.perimeter)   # 40 28
r.width = 6
print(r.area)                 #24 — przeliczone samodzielnie
r.width = -1                  # ValueError
```

★★ Zadanie samodzielne

is_square

Dodaj Rectangle property is_square, który zwraca True, jeśli width jest height.

Praktyka: Klasa Rectangle

interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

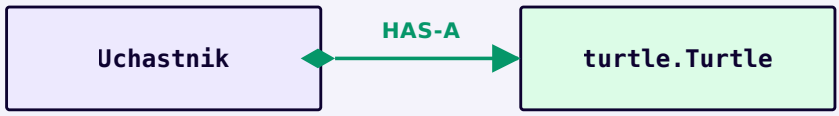
Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/14-12/index.html)

Kompozycja: Obiekty wewnątrz obiektów

Jeden obiekt może PRZECHOWYWAĆ inne obiekty, będąc za nie odpowiedzialnym – tak właśnie Uchastnik było uporządkowane w sekcji 14.4.

Cecha jako atrybut innej cechy

W rozdziale 14.4 klasa `Uchastnik` przechowywał obiekt `turtle.Turtle` w twoim atributie `self.t` – nie odziedziczyłem po nim, ale ZACHOWAŁEM to. Tak się nazywa **skład**: jeden obiekt jest budowany Z innych obiektów, odpowiadając za nie.



Skład: Uchastnik ZAPISUJE Turtle w swoich atrybutach self.t

Zasada nazewnictwa związku: HAS-A

Esej można łatwo sprawdzić, pytając "X is Y?" lub "X HAS Y?". `Uchastnik` nie jest żółwiem (HAS-A prawda, IS-A fałsz) — dlatego jest to kompozycja, a nie dziedziczenie (o nim — w rozdziale 14.15).

Graf obiektowy: kompozycja może iść w głąb

Obiekt może przechowywać nie tylko jeden obiekt, ale całe lista obiektów — i te własne Kolejka może przechowywać własne:

- Zakaz
  - `Tovar('Książka', 590)`
  - `Tovar('Rękawica', 90)`
  - `Tovar('Zeszyt', 120)`

Zakaz przechowuje LISTĘ obiektów Tovar jest kompozycją, w której właściciel przechowuje kilka obiektów jednocześnie

DEBUG LAB 7: POMIESZANIE POZIOMU ZAGNIEŻDŻANIA ATRYBUTU

propushchennyj_uroven.py

```
class Mashina:
    def __init__(self):
        self.dvigatel = Dvigatel()

mashina = Mashina()
mashina.start()     # i gdzie dokładnie jest
zdefiniowana start()?
```

```
Traceback (most recent call last):
  File "propushchennyj_uroven.py", line 5, in
    mashina.start()
AttributeError: 'Mashina' object has no attribute 'start'
```

Co widać na ekranie

`start()` jest określana w `Dvigatel`, nie `Mashina` - Podczas komponowania metody zagnieżdżonego obiektu NIE stają się automatycznie metodami właściciela. Musisz uzyskać do niego dostęp jawnie, poprzez atrybut, w którym obiekt jest przechowywany.

POPRAWIONY KOD

cherez_atribut.py

```
mashina.dvigatel.start()   # wyraźnie: przez
self.dvigatel
```

Praktyka: kompozycja
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/14-13/index.html)

Mini Projekt: Koszyk v2

Lista słowników z rozdziału 12 staje się listą przedmiotów Tovar w Korzina.

Mini Projekt: Koszyk v2

W rozdziale 12 koszyk zakupowy był przechowywany jako lista słowników. Teraz mamy narzędzia, aby go uczynić bardziej niezawodnym: klasa `Tovar` dla jednego towaru i klasy `Korzina`, który przechowuje LISTĘ obiektów `Tovar` jest składem z sekcji 14.13 w praktyce.

korzina_v2.py

```
class Tovar:
    def __init__(self, nazvanie, tsena):
        self.nazvanie = nazvanie
        self.tsena = tsena


class Korzina:
    def __init__(self):
        self.tovary = []      atrybut # INSTANCJA - Rozdział
14.8!

    def dobavit_tovar(self, tovar):
        self.tovary.append(tovar)

    def obshchaya_summa(self):
        return sum(t.tsena for t in self.tovary)

    def kolichestvo_tovarov(self):
        return len(self.tovary)
```

Korzina przechowuje lista przedmioty Tovar

ispolzovanie.py

```
korzina = Korzina()
korzina.dobavit_tovar(Tovar(„Book”, 590))
korzina.dobavit_tovar(Tovar(„Uchwyt”, 90))

print(korzina.kolichestvo_tovarov())   # 2
print(korzina.obshchaya_summa())        # 680
```

Dlaczego jest lepszy niż słownik

Blisko obiektu `Tovar` zawsze jest dokładnie `nazvanie` oraz `tsena` — literówka w kluczu słownika (`tovar["cena"]` zamiast `tovar["tsena"]`) zostanie wykryta dopiero w czasie wykonywania, a odwołanie do nieistniejącego atrybutu obiektu — również w czasie wykonywania, ale z klasą łatwiej dodać sprawdzenia i metody (liczenie sumy, rabat) w jednym zrozumiałym miejscu.

★★ Zadanie samodzielne

udalit_tovar

Dodaj Korzina udalit_tovar(self metodę nazvanie), która usuwa pierwszy produkt o tej nazwie z self.tovary.

Praktyka: Koszyk v2

interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/14-14/index.html)

Dziedzictwo

Kiedy jedna klasa jest bardziej szczególnym przypadkiem innej, dziedziczenie pozwala nie przepisywać ogólnego zachowania od nowa.

Zbuduj klasę na podstawie istniejącej

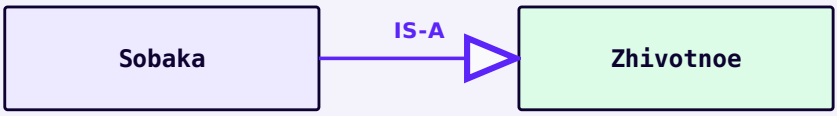
Jeśli dwie klasy mają wiele wspólnego, ale jedna jest pojęciem bardziej ogólnym, a druga jego szczególnym przypadkiem, relacja między nimi — **IS-A** („Kot to zwierzę” **dziedziczenie**:

nasledovanie.py

```
class Zhivotnoe:
    def __init__(self, klichka):
        self.klichka = klichka

    def predstavitsya(self):
        print(f"Ja {self.klichka}")


class Sobaka(Zhivotnoe):     # Sobaka — to Zhivotnoe
    def zvuk(self):
        return „Hau!”


class Koshka(Zhivotnoe):     #Koshka też jest Zhivotnoe
    def zvuk(self):
        return „Miau!”


rex = Sobaka(„Rex”)
rex.predstavitsya()    # odziedziczone po Zhivotnoe — działa
bez nadpisywania
print(rex.zvuk())       # Twoja własna, zdefiniowana w Sobaka
```

- Zhivotnoe
  - Sobaka
  - Koshka

Sobaka i Koshka dziedziczą po Zhivotnoe – powszechne zachowanie (predstavitsya) nie musi być zapisywane dwa razy



IS-A: otwarty trójkąt wskazuje rodzica

Dziedziczenie jest ponownym wykorzystaniem, a nie obowiązkiem

`Sobaka` dostaje `predstavitsya()` za DARMO, bez przepisywania czegokolwiek — o to właśnie chodzi. Ale dziedziczenie jest uzasadnione tylko wtedy, gdy relacja jest rzeczywista IS-A. Jeśli chcesz odziedziczyć klasę tylko po to, by „pożyczyć”

DEBUG LAB 8: ZAPOMNIAŁEM ZADZWONIĆ DO SUPER().__INIT__()

propushchennyj_super.py

```
class Zhivotnoe:
    def __init__(self, klichka):
        self.klichka = klichka


class Sobaka(Zhivotnoe):
    def __init__(self, klichka, poroda):
        self.poroda = poroda     #klichka Zhivotnoe nie
skonfigurowanych!

rex = Sobaka(„Rex”, „Kundel”)
print(rex.klichka)
```

```
Traceback (most recent call last):
  File "propushchennyj_super.py", line 9, in
    print(rex.klichka)
AttributeError: 'Sobaka' object has no attribute 'klichka'
```

Co widać na ekranie

Kiedy klasa potomna definiuje WŁASNĄ `__init__` całkowicie ZASTĘPUJE rodzicielskie `__init__` - Kod konfiguracji atrybutu nadrzędnego nie działa już samodzielnie. Aby uzyskać oba, dziecko `__init__` musi wyraźnie zadzwonić do rodzica — zobaczmy jak w sekcji 14.16.

POPRAWIONY KOD

s_super.py

```
class Sobaka(Zhivotnoe):
    def __init__(self, klichka, poroda):
        super().__init__(klichka)   # Najpierw
skonfiguruj część rodzica
        self.poroda = poroda
```

Praktyka: Dziedzictwo
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/14-15/index.html)

super() i procedura rozwiązywania rozwiązań

super() pozwala klasie podrzędnej korzystać z metody rodzica z samego nadpisywania — zamiast przepisywać ją od nowa.

super(): dzwonić do rodzica, a nie go zastępować

`super()` daje dostęp do metod klasy nadrzędnej OD WEWNĄTRZ metoda nadpisana — zwykle po to, by UZUPEŁNIĆ zachowanie rodzica, a nie je zastąpić Całkowicie:

super_init.py

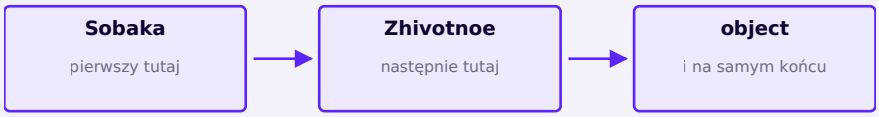
```
class Zhivotnoe:
    def __init__(self, klichka):
        self.klichka = klichka


class Sobaka(Zhivotnoe):
    def __init__(self, klichka, poroda):
        super().__init__(klichka)   # wykonuje
Zhivotnoe.__init__(self klichka)
        self.poroda = poroda         # następnie dodaje swoje

rex = Sobaka(„Rex”, „Kundel”)
print(rex.klichka, rex.poroda)   # Rex Mongrel — Oba atrybuty
są obecne
```

Powszechne nieporozumienie: „super() zwraca obiekt klasy nadrzędnej”

To nieprawda. `super()` zwraca specjalny pośredni obiekt (proxy), który potrafi znajdować metody RODZICA dla BIEŻĄCEGO `self` — `self` pozostaje obiektem `Sobaka` nie zamienia się w `Zhivotnoe`. Po prostu tym razem poszukiwanie metody się nie zaczyna `Sobaka`, ale o poziom wyżej.

Kolejność rozwiązywania metod (MRO)

W przypadku normalnego pojedynczego dziedziczenia procedura znalezienia metody jest prosta i przewidywalna: najpierw klasa obiektu, potem jej rodzic, potem rodzic rodzica — i tak dalej `object`, podstawowa klasa dla wszystkich klas w Python:



MRO dla pojedynczego dziedziczenia – przewidywalna kolejność wyszukiwania

DEBUG LAB 9: UŻYJ ATRYBUTU BEFORE, ABY WYWOŁAĆ SUPER().__INIT__()

nepravilnyj_poryadok.py

```
class Sobaka(Zhivotnoe):
    def __init__(self, klichka, poroda):
        self.poroda = poroda
        print(f"Rasa: {self.poroda}, przezwyk:
{self.klichka}")   # klichka jeszcze nie!
        super().__init__(klichka)
```

```
Traceback (most recent call last):
    print(f"Порода: {self.poroda}, кличка: {self.klichka}"
AttributeError: 'Sobaka' object has no attribute 'klichka'
```

Co widać na ekranie

`super().__init__(klichka)` powinno być PO ZA linijką, która już próbuje przeczytać `self.klichka` jest już rodzicem `__init__` jeszcze nie wykonano, atrybut jeszcze nie utworzono. Kolejność wywołań wewnątrz `__init__` wykonywana jest ściśle od góry do dołu, jak w każdej innej funkcji.

POPRAWIONY KOD

pravilnyj_poryadok.py

```
class Sobaka(Zhivotnoe):
    def __init__(self, klichka, poroda):
        super().__init__(klichka)   # Rodzic na
pierwszym miejscu
        self.poroda = poroda
        print(f"Rasa: {self.poroda}, przezwyk:
{self.klichka}")
```

Praktyka: super()
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/14-16/index.html)

Nadpisywanie metod

Klasa dziecięca może całkowicie zastąpić metodę rodzica — lub uzupełnić ją poprzez super().

Nadpisywanie: Własna wersja metody rodzica

Podklasa może zdefiniować metodę o TYM SAMYM imieniu, co u rodzica — to nazywa się **przejęcie** (override). Python przy wyszukiwaniu metody znajduje wersję klasy podrzędnej pierwszą (rozdział 14.16, MRO) i używa właśnie jej:

pereopredelenie.py

```
class Zhivotnoe:
    def zvuk(self):
        print(„Jakieś zwierzę wydaje dźwięk”)


class Sobaka(Zhivotnoe):
    def zvuk(self):     # całkowicie zastępuje wersję nadrzędną
        print(„Hau!”)

Sobaka().zvuk()   # Hau! — Zhivotnoe wersja nawet nie jest
nazywana
```

Czasami nie trzeba całkowicie zastępować zachowania rodzica, ale UZUPEŁNIĆ je — wtedy nadpisana metoda wywołuje `super().метод()` i dodaje coś własnego:

rasshirenie.py

```
class Zhivotnoe:
    def zvuk(self):
        print(„Zwierzę daje głos:”)


class Sobaka(Zhivotnoe):
    def zvuk(self):
        super().zvuk()     # Rodzic na pierwszym miejscu
        print(„Hau!”)      # potem własny

Sobaka().zvuk()
# Zwierzę daje głos:
#woof!
```

DEBUG LAB 10: ZDEFINIOWAŁEM NA NOWO METODĘ, ALE ZAPOMNIAŁEM ROZSZERZYĆ SUPER()

poteryannoe_povedenie.py

```
class Zhivotnoe:
    def zvuk(self):
        print(„Zwierzę daje głos:”)


class Sobaka(Zhivotnoe):
    def zvuk(self):
        print(„Hau!”)     # chcieli DODATKOWO, ale
przypadkowo zastąpili w całości

Sobaka().zvuk()
```

```
>>> Sobaka().zvuk()
Гав!
```

Co widać na ekranie

Linia „Animal Voices:” `super().zvuk()` muszą być jasno wyjaśnione: Python nigdy nie nazywa samej wersji nadrodzica, chyba że sam o to poprosiłeś.

POPRAWIONY KOD

s_rasshireniem.py

```
class Sobaka(Zhivotnoe):
    def zvuk(self):
        super().zvuk()
        print(„Hau!”)
```

Praktyka: nadpisywanie metod
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/14-17/index.html)

Polimorfizm

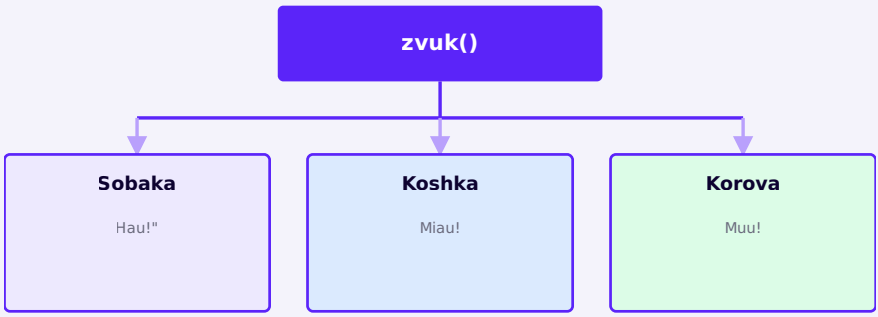
Ten sam łańcuch znaków kodu — .zvuk() — zachowuje się inaczej w zależności od tego, który dokładnie obiekt go wywołał.

Jedno wezwanie, różne wyniki

Polimorfizm („wiele form”

polimorfizm.py

```
class Sobaka:
    def zvuk(self):
        return „Hau!”


class Koshka:
    def zvuk(self):
        return „Miau!”


class Korova:
    def zvuk(self):
        return „Muu!”


zhivotnye = [Sobaka(), Koshka(), Korova()]
for zh in zhivotnye:
    print(zh.zvuk())
# INNA wersja zvuk() jest wywoływana za każdym razem
```



To samo wywołanie .zvuk() — różny rezultat dla każdej klasy

Pętla nie wie i nie powinna wiedzieć, z którą klasą pracuje
String `zh.zvuk()` wewnątrz pętli JEDEN — nie musi sprawdzać `if isinstance(zh, Sobaka): ...` jest taki sam dla każdej klasy. To jest wartość polimorfizmu: kod, który UŻYWA obiektów, pozostaje prosty i nie rośnie wraz z każdą nową klasą zwierząt.

Praktyka: polimorfizm
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

`Otwórz praktykę →` (`https://www.cartesianschool.org/pl/practice/14-18/index.html`)

Duck typing

Python nie sprawdza, z której klasy obiekt odziedziczył, tylko czy ma właściwą metodę.

„Jeśli wygląda jak kaczka i kwacze jak kaczka...”

W sekcji 14.18 wszystkie klasy nie zawsze miały wspólnego rodzica. W rzeczywistości Python w ogóle nie sprawdza, z której klasy obiekt jest dziedziczony, aby wywołać tylko jeśli obiekt MA właściwą metodę:

duck_typing.py

```
class Sobaka:                       # nie ma nic wspólnego z Robot
    def predstavitsya(self):
        print(„Hau! Jestem psem.”)


class Robot:                        # nie jest w żaden sposób
powiązane z Sobaka
    def predstavitsya(self):
        print(„BIP. Jestem robotem.”)


def poznakomit(obj):
    obj.predstavitsya()   # nieważne, jakiej klasy jest obj —
ważne, że metoda istnieje

poznakomit(Sobaka())
poznakomit(Robot())
```

Częsty błąd: „polimorfizm wymaga dziedziczenia”

To nieprawda, a duck typing — jest tego bezpośrednim dowodem. `Sobaka` oraz `Robot` NIE mają wspólnego rodzica (poza rodzicem domyślnym) `object`), ale `poznakomit()` działa równie dobrze z obiema metodami. W Python interfejs jest faktyczną obecnością pożądanej metody, a nie formalnym wpisem w drzewie dziedziczenia.

Nazwa pochodzi od wyrażenia „jeśli wygląda jak kaczka, pływa jak kaczka i kwacze jak kaczka — prawdopodobnie to jest kaczka”: nie ma znaczenia, jak obiekt został stworzony, ważne co POTRAFI.

DEBUG LAB 11: NIE WSZYSTKIE OBIEKTY NA LIŚCIE MAJĄ OCZEKIWANĄ METODĘ

nesovmestimyj_obekt.py

```
zhivotnye = [Sobaka(), Robot(), „po prostu łańcuch
znaków”]
for zh in zhivotnye:
    zh.predstavitsya()
```

```
Гав! Я собака.
БИП. Я робот.
Traceback (most recent call last):
AttributeError: 'str' object has no attribute 'predstavitsya'
```

Co widać na ekranie

Duck typing nie sprawdza wcześniej metody — Python po prostu próbuje wywołać `predstavitsya()` i zawiesza się w momencie wywołania, jeśli metody nie ma. Elastyczność duck typing nie zwalnia z odpowiedzialności: umieszczanie w jednym lista obiektów, które MUSZĄ wspierać wspólną metodę, wymaga świadomego działania.

POPRAWIONY KOD

s_proverkoj.py

```
for zh in zhivotnye:
    if hasattr(zh, "predstavitsya"):
        zh.predstavitsya()
```

Praktyka: duck typing
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/14-19/index.html)

Mini-projekt: kształty polimorficzne

Wspólny przodek określa interfejs, każda figura implementuje go na swój sposób — dziedziczenie i polimorfizm razem.

Mini-projekt: kształty polimorficzne

Wspólny przodek `Figura` przechowuje nazwę kształtu; każdy konkretny Nadpisania kształtu `ploshchad()` na swój sposób – dziedziczenie (14.15) i polimorfizm (14.18) razem:

figury.py

```
class Figura:
    def __init__(self, nazvanie):
        self.nazvanie = nazvanie

    def ploshchad(self):
        raise NotImplementedError


class Krug(Figura):
    def __init__(self, radius):
        super().__init__(„krąg”)
        self.radius = radius

    def ploshchad(self):
        return 3.14159 * self.radius ** 2


class Pryamougolnik(Figura):
    def __init__(self, width, height):
        super().__init__(„prostokąt”)
        self.width = width
        self.height = height

    def ploshchad(self):
        return self.width * self.height


class Treugolnik(Figura):
    def __init__(self, base, height):
        super().__init__(„trójkąt”)
        self.base = base
        self.height = height

    def ploshchad(self):
        return 0.5 * self.base * self.height
```

- Figura
  - Krug
  - Pryamougolnik
  - Treugolnik

Wszystkie trzy kształty są IS-A Figura, każdy redefiniuje ploshchad() na swój sposób

ispolzovanie.py

```
figury = [Krug(5), Pryamougolnik(4, 6), Treugolnik(8, 3)]
for f in figury:
    print(f"{f.nazvanie}: {f.ploshchad():.2f}")

# koło: 78,54
# prostokąt: 24.00
# trójkąt: 12.00
```

raise NotImplementedError – sygnał „zakończyć dziedzica”

`Figura.ploshchad()` nie oblicza celowo niczego użytecznego — istnieje jedynie, by ustawić WSPÓLNY INTERFEJS, który potomkowie muszą zaimplementować. Jeśli zapomnisz nadpisać `ploshchad()` w nowym kształcie, błąd zostanie wykryty natychmiast przy pierwszym wywołaniu i nie zostanie cicho podany w niewłaściwym obszarze.

★★ Zadanie samodzielne

perimetr

Dodaj Figura metodę-szkielet perimetr() (raise NotImplementedError) i zaimplementuj ją we wszystkich trzech figurach. Podpowiedź dla Treugolnik: base i height nie wystarcza do dokładnego obwodu — przyjmij go jako równoramienny i znajdź bok za pomocą twierdzenia Pitagorasa.

Praktyka: kształty polimorficzne
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/14-20/index.html)

Metody specjalne

__init__ nie jest jedyną metodą Dundera. __str__ i __eq__ łączą obiekt z print() i ==.

Jak obiekty łączą się z wbudowanymi operacjami

Kiedy piszesz `print(rex)` lub `len(text)` Python faktycznie wywołuje specjalną metodę na obiekcie o nazwie widoku `__имя__` („dander”, double underscore). Już korzystaliście z takiej metody od początku rozdziału — `__init__`.

To, co piszesz	Co Python wywołuje w rzeczywistości
`Sobaka("Рекс", 3)`	`__init__(self, ...)`
`print(rex)`	`__str__(self)`
`len(korzina)`	`__len__(self)`
`a + b`	`__add__(self, other)`
`a == b`	`__eq__(self, other)`

__str__ i __repr__

Bez specjalnej metody `print(объект)` usuwa bezużyteczność linijka jak `<__main__.Sobaka object at 0x7f...>`. `__str__` pozwala ustawić czytelny dla człowieka wygląd:

str_metod.py

```
class Sobaka:
    def __init__(self, klichka, vozrast):
        self.klichka = klichka
        self.vozrast = vozrast

    def __str__(self):
        return f"{self.klichka} ({self.vozrast} roku)"

rex = Sobaka(„Rex”, 3)
print(rex)   # Rex (3 lata) — nie jest adresem pamięci
```

__eq__: porównania przez znaczenie, a nie przez tożsamość

W rozdziale 14.9 `a1 == a2` dwa Player z tym samym wartości okazały się `False` bo `==` domyślnie porównuje obiekty jako różne lokalizacje pamięci. `__eq__` to zmienia:

eq_metod.py

```
class Sobaka:
    def __init__(self, klichka, vozrast):
        self.klichka = klichka
        self.vozrast = vozrast

    def __eq__(self, other):
        return self.klichka == other.klichka and self.vozrast
== other.vozrast

Sobaka(„Rex”, 3) == Sobaka(„Rex”, 3)   # teraz True
```

DEBUG LAB 12: __EQ__ ZOSTAŁ ZIDENTYFIKOWANY, A OBIEKT STAŁ SIĘ UNHASHABLE

unhashable_obekt.py

```
class Tochka:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __eq__(self, other):
        return self.x == other.x and self.y == other.y

tochki = {Tochka(1, 2), Tochka(3, 4)}   # umieść
przedmioty w zbiór
```

```
Traceback (most recent call last):
TypeError: unhashable type: 'Tochka'
```

Co widać na ekranie

Obiekt ma automatyczną `__hash__`, oparty na jego adresie w pamięci. Jak tylko określisz swój `__eq__`, Python USUWA AUTOMATYCZNIE `__hash__` — ponieważ jeśli obiekty są „równe” pod względem wartości, logiczne jest, że mają też taki sam hash, a stary adresowy hash już temu nie odpowiada. Obiektu bez hasha nie można umieścić w `set` lub użyć jako klucz `dict`.

POPRAWIONY KOD

s_hash.py

```
    def __eq__(self, other):
        return self.x == other.x and self.y == other.y

    def __hash__(self):
        return hash((self.x, self.y))
```

Praktyka: Metody specjalne
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/14-21/index.html)

Praktyka: Aplikuj __str__ i __eq__

Mała klasa Tochka to plac zabaw do utrwalania obu metod w Sekcji 14.21.

Praktyka: punkt na płaszczyźnie

Zastosuj obie metody z rozdziału 14.21 do małej klasy `Tochka` jest współrzędną `x`, `y`:

tochka.py

```
class Tochka:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __str__(self):
        return f"({self.x}, {self.y})"

    def __eq__(self, other):
        return self.x == other.x and self.y == other.y

a = Tochka(1, 2)
b = Tochka(1, 2)
c = Tochka(5, 5)

print(a)          # (1, 2)
print(a == b)      # True — współrzędne się zgadzają
print(a == c)      # False
```

f-string i __str__ współpracują
`f"Точка: {a}"` powoduje to samo `__str__` to i `print(a)` – każde miejsce, gdzie Python trzeba zamienić obiekt na łańcuch znaków dla człowieka, przechodzi przez tę metodę.

★★ Zadanie samodzielne
rasstoyanie
Dodaj Tochka metodę rasstoyanie_do(self other), która zwraca odległość euklidesową do innego punktu: ((self.x - other.x) ** 2 + (self.y - other.y) ** 2) ** 0,5.

Praktyka: __str__ i __eq__ w praktyce
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji
Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/14-22/index.html)

dataclasses

Klasy, które głównie przechowują dane, nie muszą zapisywać __init__ i __repr__ ręcznie.

Mniej szablonowego kodu dla klas danych

Zajęcia takie jak `Tochka` z sekcji 14.22 – głównie Kod szablonowy: `__init__` który po prostu kopiuje parametry do `self`, plus `__str__` oraz `__eq__` prawie zawsze wyglądają tak samo. Moduł `dataclasses` potrafi wygenerować to wszystko automatycznie:

Dane klasowe: ręcznie i @dataclass

Klasa Normalna

```
class Tochka:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __repr__(self):
        return
f"Tochka(x={self.x},
y={self.y})"

    def __eq__(self, other):
        return self.x ==
other.x and self.y == other.y
```

S @dataclass

```
from dataclasses import
dataclass

@dataclass
class Tochka:
    x: int
    y: int
#__init__, __repr__ i __eq__
są generowane automatycznie
```

Co używać dzisiaj: @dataclass — tam, gdzie klasa głównie PRZECHOWUJE dane i nie korzysta z ręcznej kontroli nad __init__/__repr__/__eq__. Wciąż jest to zwykła klasa — metody dodaje się do ciała dokładnie tak samo jak zawsze.

dataclass_s_metodom.py

```
from dataclasses import dataclass

@dataclass
class Tochka:
    x: int
    y: int

    def rasstoyanie_do(self, other):
        return ((self.x - other.x) ** 2 + (self.y - other.y)
** 2) ** 0.5

a = Tochka(0, 0)
b = Tochka(3, 4)
print(a)                    # Tochka(x=0, y=0) — __repr__ już
gotowy
print(a.rasstoyanie_do(b))   # 5.0
```

Powszechne nieporozumienie: "dataclass nie jest prawdziwą klasą" albo "nie da się pisać metod w dataclass"

Obie są błędne. `@dataclass` jest zwykłym dekoratorem klas: tylko DODAJE automatycznie generowane `__init__`, `__repr__`, `__eq__` na zwykłej klasie. Metody, property i wszystko, co można napisać w zwykłej klasie, jest napisane w ten sam sposób w dataclass.

DEBUG LAB 13: ZMIENNA DOMYŚLNOŚĆ W DATACLASS

izmenyaemoe_polei_umolchanie.py

```
from dataclasses import dataclass

@dataclass
class Korzina:
    tovary: list = []   # wygląda jak pułapka z sekcji
14.8!
```

```
Traceback (most recent call last):
ValueError: mutable default  for field tovary is not allowe
```

Co widać na ekranie

Python `dataclasses` CELOWO wyłącza domyślne prawo do zmiany w adnotacji – to takie samo ryzyko współdzielonej listy dla wszystkich obiektów jak w sekcji 14.8, z tą różnicą, że dataclass jest to wykrywane natychmiast, przez błąd definicji klasy, a nie cichy błąd w czasie działania.

POPRAWIONY KOD

s_default_factory.py

```python
from dataclasses import dataclass, field

@dataclass
class Korzina:
    tovary: list = field(default_factory=list)   # nowy
lista dla KAŻDEGO obiektu
```

Praktyka: dataclasses
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/14-23/index.html)

Praktyka: dataclass

Tovar Rozdział 14.14, ale krótko mówiąc, @dataclass przejmuje kod standardowy.

Praktyka: Tovar jako dataclass

Przeróbka `Tovar` od sekcji 14.14 do `@dataclass`:

tovar_dataclass.py

```python
from dataclasses import dataclass

@dataclass
class Tovar:
    nazvanie: str
    tsena: float

    def so_skidkoj(self, protsent):
        return self.tsena * (1 - protsent / 100)

knigi = Tovar(„Book”, 590)
print(knigi)                    # Tovar(nazvanie='Książka',
tsena=590)
print(knigi.so_skidkoj(10))      # 531.0
```

__eq__ jest darmowy
`Tovar("Книга", 590) == Tovar("Книга", 590)` — `True`, chociaż nie pisaliśmy `__eq__` ręcznie ani razu: dataclass porównuje wartości wszystkich pól automatycznie.

★★ Zadanie samodzielne
Zakaz jak dataclass
Zdefiniuj @dataclass Zakaz z polami tovar: Tovar i kolichestvo: int oraz metodą summa(self), która zwraca self.tovar.tsena * self.kolichestvo.

Praktyka: dataclass Tovar
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji
Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/14-24/index.html)

klasa czy nie? Projektowanie modeli

Nie każde zadanie wymaga klasy — a zły wybór nie jest od razu widoczny, lecz dopiero wtedy, gdy kod staje się trudniejszy do odczytania.

Nie każde zadanie wymaga zajęć

Po 24 rozdziałach o klasach łatwo dojść do wniosku, że klasą należy otaczać wszystko. To nie tak — klasa jest przydatna, gdy istnieje I stan, I zachowanie, które ma sens trzymać razem, i gdy będzie kilka obiektów tego rodzaju, z niezależną historią.

Zajęcia czy nie?

Trzeba zarówno przechowywać dane, jak i wykonywać na nich działania? → → pewnie klasa

Czy będzie kilka niezależnych egzemplarzy w różnych stanach? → →

Czy to jednorazowe obliczenie bez stanu zapisanego? → → Normalna funkcja

Dane bez zachowania, przeczytać raz? → → dict / tuple / dataclass bez metod

Potrzebna weryfikacja przy zmianie wartości? → → property (rozdział 14.11)

DEBUG LAB 14: KLASA DLA JEDNOOBLICZENIOWEGO DZIAŁANIA BEZ STANU

izlishnij_klass.py

```
class SredneeArifmeticheskoe:
    def __init__(self, chisla):
        self.chisla = chisla

    def vychislit(self):
        return sum(self.chisla) / len(self.chisla)

rezultat = SredneeArifmeticheskoe([4, 8,
15]).vychislit()
```

```
# Код работает правильно — ошибка не во время выполнения,
# а в самом ДИЗАЙНЕ: слишком много церемоний для одного вычис
```

Co widać na ekranie

Technicznie rzecz biorąc, to działający kod, ale nie ma stanu rzeczywistego (obiekt jest tworzony i używany raz) ani wielu instancji — tylko obliczenia udawające klasę. Zwykła funkcja jest łatwiejsza do odczytania i testowania: `srednee(chisla)`. Klasa jest uzasadniona, gdy obiekt żyje przez jakiś czas I zmienia stan jako `Player` z sekcji 14.9, nie gdy istnieje w jednej linii.

POPRAWIONY KOD

prostaya_funkciya.py

```
def srednee_arifmeticheskoe(chisla):
    return sum(chisla) / len(chisla)

rezultat = srednee_arifmeticheskoe([4, 8, 15])
```

Znaki, że klasa należy do niej

Obiekt zmienia swój stan PO utworzeniu kilka razy (nie tylko wtedy, gdy __init__); w programie będzie jednocześnie więcej niż jeden taki obiekt; dane i działania na nich są logicznie powiązane w takim stopniu, że ich rozdzielanie jest niewygodne. Jeśli żadna z cech nie jest spełniona, zacznij od funkcji lub słownika, klasa ta zawsze może zostać dodana później, gdy będzie naprawdę potrzebna.

★★ Zadanie samodzielne

Funkcja czy klasa?

Dla każdego scenariusza zdecyduj, co jest bardziej odpowiednie — funkcja czy klasa, i uzasadnij: (1) konwersja temperatury z Celsjusza na Fahrenheita; (2) konto bankowe z saldem i historią operacji; (3) sprawdzenie, czy liczba jest pierwsza.

Praktyka: klasa czy nie klasa

interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/14-25/index.html)

Mini-Projekt: Race Turtle v2

Sam wyścig staje się obiektem — kontroluje uczestników i nie powiela ich logiki.

Mini-Projekt: Race Turtle v2

Rozwińmy projekt z sekcji 14.4. Tym razem nie chodzi tylko o uczestników :(`Uchastnik`) — sama rasa staje się także obiektem: klasą `Gonka` przechowuje lista uczestników, rysuje linię mety i określa kolejność mety — kolejny przykład kompozycji (rozdział 14.13), ale już obiekt, który zarządza innymi obiektami.

gonka_v2.py

```
import turtle
import random

random.seed(7)
screen = turtle.Screen()
screen.setup(520, 420)

class Uchastnik:
    def __init__(self, cvet, startovyj_y):
        self.t = turtle.Turtle()
        self.t.shape("turtle")
        self.t.color(cvet)
        self.t.penup()
        self.t.goto(-220, startovyj_y)
        self.cvet = cvet
        self.finishiroval = False

    def sdelat_shag(self):
        if not self.finishiroval:
            self.t.forward(random.randint(1, 10))

    def proverit_finish(self, finish_x):
        if self.t.xcor() >= finish_x:
            self.finishiroval = True
        return self.finishiroval


class Gonka:
    def __init__(self, cveta, finish_x):
        self.finish_x = finish_x
        self.uchastniki = [
            Uchastnik(cvet, i * 50 - 75) for i, cvet in
enumerate(cveta)
        ]
        self.rezultaty = []

    def narisovat_finish(self):
        liniya = turtle.Turtle()
        liniya.hideturtle()
        liniya.penup()
        liniya.goto(self.finish_x, -120)
        liniya.pendown()
        liniya.pencolor("#0D0230")
```

```
        liniya.pensize(3)
        liniya.setheading(90)
        liniya.forward(240)

    def sygrat_do_kontsa(self):
        self.narisovat_finish()
        while len(self.rezultaty) < len(self.uchastniki):
            for u in self.uchastniki:
                u.sdelat_shag()
                if u.proverit_finish(self.finish_x) and u.cvet
not in self.rezultaty:
                    self.rezultaty.append(u.cvet)

gonka = Gonka(["red", "blue", "green"], 210)
gonka.sygrat_do_kontsa()

screen.exitonclick()
```

REZULTAT

Trzy żółwie na torach obok pionowej linii mety

Gonka trzyma trzy Uchastnik i sam rysuje linię mety

Gonka przechowuje lista obiektów Uchastnik i zarządza wyścigiem w całości

Obiekty współpracujące ze sobą

`Gonka` nie powtarza logiki sprawdzania kroku czy zakończenia — deleguje ją wszystkim 16 `Uchastnik` (`u.sdelat_shag()`, `u.proverit_finish(...)`) i tylko je koordynuje. To typowy obraz OWP: nie jedna wielka klasa, która potrafi wszystko, lecz kilka małych klas, z których każda odpowiada za swoje i współpracuje poprzez zrozumiałe metody ze sobą nawzajem.

Challenge

Tabela wyników

Dodaj Gonka tablitsa_rezultatov(self) metodę, która zwraca self.rezultaty jako tekst ponumerowany ("1. miejsce: red", "2. miejsce: blue", ...).

Praktyka: Wyścig Turtle v2

Moduł turtle otwiera natywne okno Python – uruchamiane lokalnie w VS Code, PyCharm lub Jupyter

`Otwórz praktykę →` (https://www.cartesianschool.org/pl/practice/14-26/index.html)

Podsumowanie rozdziału

Od pierwszego obiektu do modeli z wielu współpracujących klas. Łączenie wszystkiego.

Zestaw narzędzi OOP, który teraz masz

Podsumowanie Rozdziału 14

PODSTAWY

```
class, __init__, self
atrybuty i metody
instancja vs klasie (14.7)
```

ENKAPSULACJA

_internal, __name
property i @x.setter

RELACJE OBIEKTOWE

kompozycja — HAS-A (14.13)
dziedziczenie — IS-A (14.15)
super() (14.16)

ELASTYCZNOŚĆ ZACHOWAŃ

polimorfizm (14.18)
duck typing (14.19)

WSPÓŁCZESNE PYTHON

Metody specjalne (14.21)
@dataclass (14.23)

PROJEKT

zajęcia czy nie? (14.25)

Ścieżka, którą przeszliśmy

- Obiekt = Stan + Zachowanie
  - Klasa opisuje przyszłe obiekty
    - __init__ dostosowuje stan
    - self kojarzy metodę z obiektem
  - Relacje między obiektami
    - Skład: Obiekt PRZECHOWUJE obiekt
    - Dziedziczenie: Obiekt JEST obiektem
  - Elastyczność zachowań
    - Polimorfizm
    - Duck typing

Od pojedynczego obiektu do modeli z wielu współpracujących klas

Czego nauczyliśmy się w tym rozdziale

- OOP organizuje kod wokół **obiektów** są stanami i zachowaniami połączonymi razem.
- `class` definiuje rysunek; `__init__` dostosowuje stan już istniejącego obiektu, zamiast go tworzyć.
- `self` — nie słowo kluczowe, a obiekt po lewej od kropki, podstawiony automatycznie podczas wiązania metody.
- Atrybut instancji jest inny dla każdego obiektu; atrybut klasy jest taki sam dla wszystkich, a wartości, które można tam zmienić, są niebezpieczne.
- Encapsulation i property chronią państwo bez zmiany składni z zewnątrz.
- Kompozycja (HAS-A) i dziedziczenie (IS-A) to dwa różne sposoby relacji klas; wybór między nimi zależy od rzeczywistej relacji pojęć, a nie od wygody.
- Polimorfizm i duck typing pozwalają kodowi działać z różnymi obiektami w ten sam sposób, bez jawnego sprawdzania ich klasy.
- Specjalne metody łączą obiekt z print(), ==, len() oraz innymi wbudowanymi operacjami; @dataclass usuwa kod szablonowy tam, gdzie nie jest potrzebny.
- Nie każde zadanie wymaga klasy — to też część projektowania, a nie wyjątek reguł.

Następny rozdział 15: Python i akta

Wszystkie obiekty w tym rozdziale żyły tylko tak długo, jak program działał: zamykały Python — i `Player`, i `Korzina` zniknęły wraz z procesem. W następnym rozdziale nauczymy się, jak zapisywać dane na dysku — pliki tekstowe i CSV pliki — aby stan przetrwał restart programu, a nie był tracony za każdym razem.

ROZDZIAŁ 15 · STR. 863

Python i akta

Ścieżka danych od obiektu Python do pliku na dysku — i z powrotem. System plików, ścieżki przenośne przez pathlib, bezpieczne odczytywanie i zapisywanie tekstu oraz bajtów do UTF-8, JSON i CSV — oraz jak zapisać stan programu między uruchomieniami, nie tylko na czas trwania jednej sesji.

Dlaczego potrzebuję plików?

Zmienne znikają wraz z programem — pliki pozwalają na długotrwałe przechowywanie danych.

Gdzie znikają zmienne?

Mały program:

schet_igry.py

```
score = 1200
player = "Anna"

print(score)
```

Program się kończy. Gdzie teraz `score` oraz `player`? Uruchom program ponownie — zacznie on znowu od wartości `score = 1200` tak, jakby wcześniej nie było premiery.

Dokładne sformułowanie

Nie „wszystko w pamięci zostaje natychmiast wymazane” **następne uruchomienie programu nie dziedziczy automatycznie Python obiektów z poprzedniego procesu**. Podczas działania programu jego obiekty znajdują się w pamięci działającego procesu; gdy proces zostaje zakończony, ta pamięć jest uwalniana przez system operacyjny.

Bez zapisu do pliku stan programu nie przetrwa jego ukończenia.

Do danych **przetrwał ukończenie programu** — lista wyników gry, tekst, ustawienia — trzeba je zapisać w **plik** na dysku (lub inne trwałe przechowywanie) i odczyt przy kolejnym uruchomieniu.

Czym jest plik?

Plik to **nazwany zbiór danych**jest przechowywany w systemie plików.

Aby program mógł otworzyć wybrany plik, musi określić, gdzie plik się znajduje. Dla jest używany **ścieżka pliku** — zapis, który pokazuje jego położenie wśród folderów i plików. Na przykład:

primer_puti.py

```
# data/results.txt
```

Tutaj `data` to teczka, i `results.txt` jest plik w środku.

> **Ścieżka — to adres pliku**
> Na pierwsze poznanie ścieżkę można przedstawić jako adres pliku w systemie plików — tak samo jak adres domu pomaga go znaleźć wśród ulic i numerów. To tylko porównanie dla intuicji: ścieżka to zapis tekstowy, a nie fizyczny adres w pamięci komputera. Ścieżki absolutne i względne omówimy szczegółowo w rozdziale 15.6.

Ostatecznie zawartość pliku jest reprezentowana przez bajty. Jeśli plik zawiera tekst, Python musi wiedzieć, jak przekształcić te bajty na znaki i odwrotnie. Reguła takiej konwersji jest nazywana **kodowanie** – Porozmawiajmy o tym szczegółowo w sekcji 15.16 razem z różnicą `str` oraz `bytes`.

Otwieranie i odczytywanie istniejących plików

chtenie_fajla.py

```
file = open("privet.txt", "r", encoding="utf-8")   # „r”
content = file.read()
print(content)
file.close()   # koniecznie zamknij plik po pracy
```

Nie zapomnij o file.close()
Niezamknięty plik może nie zapisać zmian na dysku lub zablokować dostęp do niego dla innych programów. Zapomnij `close()` to bardzo częsty błąd popełniany przez początkujących.

Nowoczesny sposób: `with`

Ręczne zamknięcie → menedżer kontekstu with

Klasyczne podejście

```
file = open("privet.txt",
"r", encoding="utf-8")
content = file.read()
print(content)
file.close()   # łatwo
zapomnieć!
```

Nowoczesna Python (with)

```
with open("privet.txt", "r",
encoding="utf-8") as file:
    content = file.read()
    print(content)
# plik sam się zamknie, nawet
jeśli w bloku wystąpi błąd
```

Co używać dzisiaj: `with` — istnieje w Python od wersji 2.5, więc nie chodzi o nowość, lecz o niezawodność: plik zamknie się automatycznie po opuszczeniu bloku `with`, nawet jeśli coś pójdzie nie tak wewnątrz. Począwszy od tego rozdziału w książce używamy właśnie `with` i zawsze wskazywać `encoding="utf-8"` jasno (więcej szczegółów w 15.16).

Praktyka: Czytaj pliki przez with
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/15-01/index.html)

Linia po linijce

Często wygodniej jest przetwarzać plik linijka po linii niż cały plik.

Czytanie całego pliku naraz nie zawsze jest wygodne — szczególnie jeśli plik jest duży. Często konieczniejsze jest jego przetworzenie **wiersz po wierszu**:

stroka_za_strokoj.py

```
with open("spisok.txt", "r", encoding="utf-8") as file:
    for line in file:
        print(line.strip())
```

Co dokładnie usuwa strip()

Każdy łańcuch znaków pliku, może poza ostatnim, kończy się niewidzialnym sygnałem `\n`. Ale `.strip()` (Rozdział 8) Oczyszcza **wszystko** znaków białych spacji po obu stronach linii — spacji, tabulatorów i podziałów — nie tylko tego jednego przelewu linii. Jeśli dane w pliku mogą sensownie zaczynać się lub kończyć spacji, ślepa `.strip()` może je niezauważalnie uszkodzić. Jeśli celem jest usunięcie przewodu końcowego i nic więcej, użyj `line.rstrip("\n")`.

DEBUG LAB 1: STRIP() USUWA WIĘCEJ, NIŻ SIĘ WYDAJE

citaty.py

```
with open("citaty.txt", "w", encoding="utf-8") as file:
    file.write(„ważny cytat z odstępami wokół krawędzi
\n”)

with open("citaty.txt", "r", encoding="utf-8") as file:
    for line in file:
        print(f"[{line.strip()}]")
```

```
[важная цитата с пробелами по краям]
```

Co widać na ekranie

Program wyświetlił cytat bez początkowych i końcowych spacji — mimo że autor celowo je umieścił. `.strip()` usuwa spacje na brzegach razem z przejściem do nowej linii i nie rozróżnia „losowego śmiecia” od „zamierzonej spacji”: po prostu usuwa wszystkie białe znaki na obu końcach linii.

POPRAWIONY KOD

citaty_fixed.py

```
with open("citaty.txt", "r", encoding="utf-8") as file:
    for line in file:
        print(f"[{line.rstrip(chr(10))}]")   # Usuń tylko linię podania
```

Wszystkie wiersze naraz — w liście

readlines.py

```
with open("spisok.txt", "r", encoding="utf-8") as file:
    lines = file.readlines()

print(len(lines), „linie”)
print(lines[0])   # pierwszy łańcuch znaków - razem z twoim \n
```

Przerwy w liniach to nie tylko \n

W Python pliki tekstowe są odczytywane w trybie uniwersalnych podziałów linii, nawet jeśli linie są oddzielone sekwencją na dysku `\r\n` (jak historycznie akceptowano w Windows), podczas czytania w trybie tekstowym, Python daje ci `\n` . Szczegóły dotyczące trybów tekstowych i binarnych dostępne są w wersji 15.17.

Praktyka: Czytanie plików linia po linijce

interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/15-02/index.html)

Utwórz nowe pliki

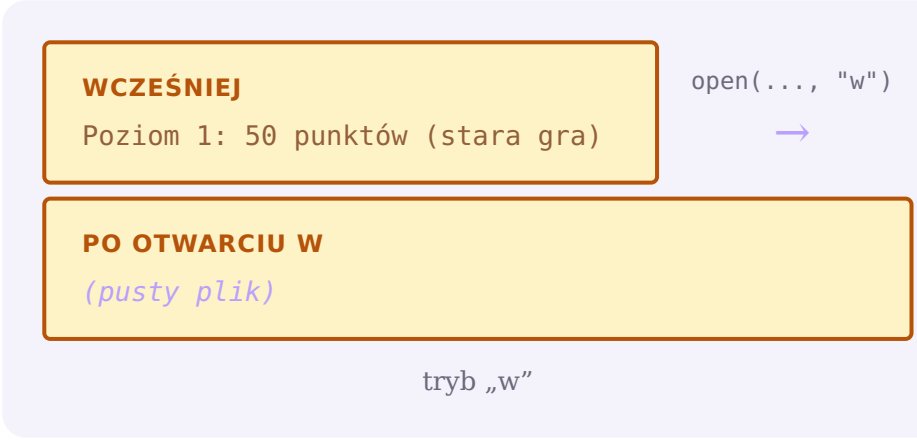
Zapisz dane na dysku i zapoznaj się z nowoczesnym sposobem pracy ze ścieżkami.

Utwórz nowe pliki

Aby coś nagrać do pliku, otwórz go w trybie nagrywania `"w"` (write) – jeśli plik nie istnieje, Python sam go utworzy; jeśli istnieje, jego wcześniejsza zawartość jest usuwana **już w momencie otwarcia**, Nawet przed pierwszym `write()`:

zapis_fajla.py

```
with open("rezultaty.txt", "w", encoding="utf-8") as file:
    file.write(„Poziom 1: 100 punktów\n”)
    file.write(„Poziom 2: 250 punktów\n”)
```



Praca z plikami

Trzy główne tryby otwierania plików:

- `"r"` - Odczyt (read) błąd, jeśli plik nie istnieje.
- `"w"` - Zapis (write); tworzy nowy plik lub całkowicie usuwa istniejący natychmiast po otwarciu.
- `"a"` jest dodatkiem (append); dodaje tekst na końcu pliku bez wymazując to, co już tam jest.

dozapis.py

```
with open("rezultaty.txt", "a", encoding="utf-8") as file:
    file.write(„Poziom 3: 400 punktów\n”)
```

"a" to nie jest "wstaw gdzie chcę"

Nadpisywanie zawsze dodaje tekst na koniec pliku. Nie może wstawiać tekstu w środku lub na początku istniejącej treści.

Ścieżki do plików: łańcuch znaków lub pathlib

Ścieżka do pliku: zwykła łańcuch znaków → pathlib

Klasyczne podejście

```
file_path = "data/
rezultaty.txt"
with open(file_path, "r",
encoding="utf-8") as f:
    print(f.read())
```

Nowoczesne Python (pathlib)

```
from pathlib import Path

file_path = Path("data") /
"rezultaty.txt"
print(file_path.exists())
#convenient metody, swoją
drogą
with file_path.open("r",
encoding="utf-8") as f:
    print(f.read())
```

Co używać dzisiaj: `pathlib` dla ścieżek plików — pojawiła się w Python 3.4 i od tego czasu jest uważana za preferowaną metodę. Ścieżki łańcuch znaków są nadal powszechne i występują w starszym kodzie, ale `pathlib` eliminuje konieczność ręcznego montażu gąsienic przez `+` i dodaje przydatne metody takie jak `.exists()` tuż obok toru. Przeanalizujmy to szczegółowo w sekcji 15.8.

Praktyka: nagrania, dodatkowe nagrania i pathlib

interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/15-03/index.html)

Mini-projekt: Dziennik notatek

Dziennik notatek zapisywany między uruchomieniami programu to pierwsza prawdziwa trwałość.

Zbierzmy czytanie i pisanie w jednym małym projekcie — dzienniku notatek, który utrzymuje się między uruchomieniami programu.

dnevnik_zametok.py

```
from pathlib import Path

fajl_zametok = Path("zametki.txt")

novaya_zametka = input(„Nowa notatka: ”)

with fajl_zametok.open("a", encoding="utf-8") as f:
    f.write(novaya_zametka + "\n")

print(„Wszystkie notatki:”)
with fajl_zametok.open("r", encoding="utf-8") as f:
    for line in f:
        print("-", line.strip())
```

Dlaczego nadpisać, a nie nadpisać

Mode `"a"` dodaje notatkę bez usuwania poprzednich — tak właśnie powinien zachowywać się prawdziwy dziennik: każde uruchomienie programu dodaje nowy wpis i nie zaczyna się od nowa.

★★ Zadanie samodzielne

Numeracja notatek

Dodaj liczbę do każdej nuty podczas czytania – na przykład "1. Pierwsza nuta", "2. Druga nuta".

Praktyka: Dziennik notatek

interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/15-04/index.html)

System plików, folderów i plików

Pliki przechowują dane, foldery organizują system plików w drzewo.

Plik i folder

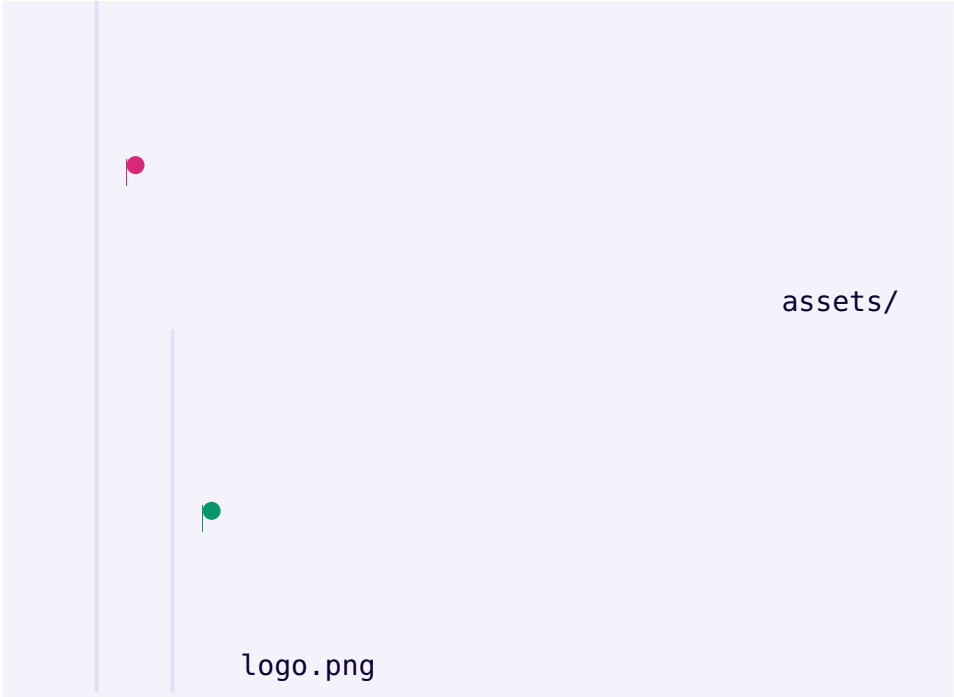
System plików rozróżnia dwa główne typy elementów:

- **Plik** przechowuje dane (tekst, kod, obrazki, cokolwiek).

- **Folder** (katalog) — nie przechowuje samych danych, lecz organizuje inne pliki oraz foldery zagnieżdżone w jego wnętrzu.

Ścieżka mówi, gdzie dokładnie w tej organizacji znajduje się konkretny element — szczegółowo o ścieżkach porozmawiamy w następnym rozdziale.

```
data/
    players.json
    scores.csv
    notes.txt
```



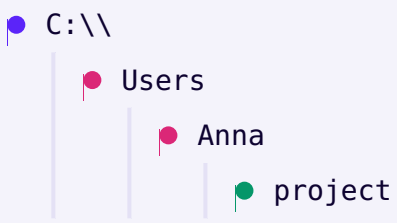
Pliki i foldery tworzą hierarchię — drzewo plików.

System plików – hierarchia od korzenia

Na Linux/macOS całe drzewo plików wyrasta z jednego korzenia — `/`:

POSIX: jednolity korzeń / — ścieżka do tego projektu.

W Windows każdy dysk ma swoje korzenie, na przykład `C:\`:



Windows: dysk jako root (np. C:\).

POSIX i Windows nie są rywalami

Nie trzeba spierać się, która opcja jest „właściwa” — są one zbudowane inaczej, a prawdziwym kodem z obiema opcjami pomoże pracować nie pamiętanie konkretnych separatorów, lecz `pathlib` (Rozdział 15.8), która tworzy przenośną ścieżkę dla systemu, w którym program działa.

Praktyka: Pliki, foldery i drzewo ścieżek

interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/15-05/index.html)

Ścieżki: bezwzględne i względne

Absolutna ścieżka jest zawsze jednoznaczna. Względna jest tylko względna względem czegoś.

Anatomia ścieżki

Path to adres elementu systemu plików, zapisywany jako sekwencja nazw folderów, oddzielone ukośniki, a (dla pliku) nazwa pliku na końcu:

```
/home/anna/project/data/scores.txt
```

Ścieżka składa się z łańcucha folderów oraz nazwy pliku – szczegółowej analizy nazwy pliku w wersji 15.9.

Ostateczna Ścieżka

Ścieżka absolutna określa lokalizację pliku, zaczynając od **korzeń** system plików (lub litera dysku na Windows) jest jednoznaczna, niezależnie od miejsca Odczytuje się ją przez:

absolyutnye_puti.py

```
# POSIX (Linux/macOS)
# /home/anna/project/data.txt

# Windows
# C:\\Users\\Anna\\project\\data.txt
```

Nie koduj na stałe ścieżek absolutnych
Takie ścieżki są tutaj pokazane jedynie jako ilustracja projektu. W obecnym kodzie kursu (i w przyszłych programach) absolutna ścieżka konkretnego komputera prawie nigdy nie jest odpowiednia — nie będzie istnieć na innym komputerze.

Ścieżka względna — względem czego?

Ścieżka gatunku:

otnositelny_put.py

```
# data/scores.txt
```

nie nadaje jednoznacznego miejsca w systemie plików, dopóki nie odpowiesz na główne pytanie: **o czym?** Odpowiedź — rozdział 15.7: ścieżka względna jest rozwiązywana w odniesieniu do **aktualny katalog roboczy** programy, jeśli API lub dokumentację Mówią inaczej.

Praktyka: Ścieżki absolutne i względne
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/15-06/index.html)

Obecny katalog roboczy jest CWD

Jednym z najczęstszych źródeł błędów plików: względna ścieżka nie prowadzi tam, gdzie myślałeś.

Bieżący katalog roboczy (CWD)

Sprawdź aktualny katalog roboczy programu:

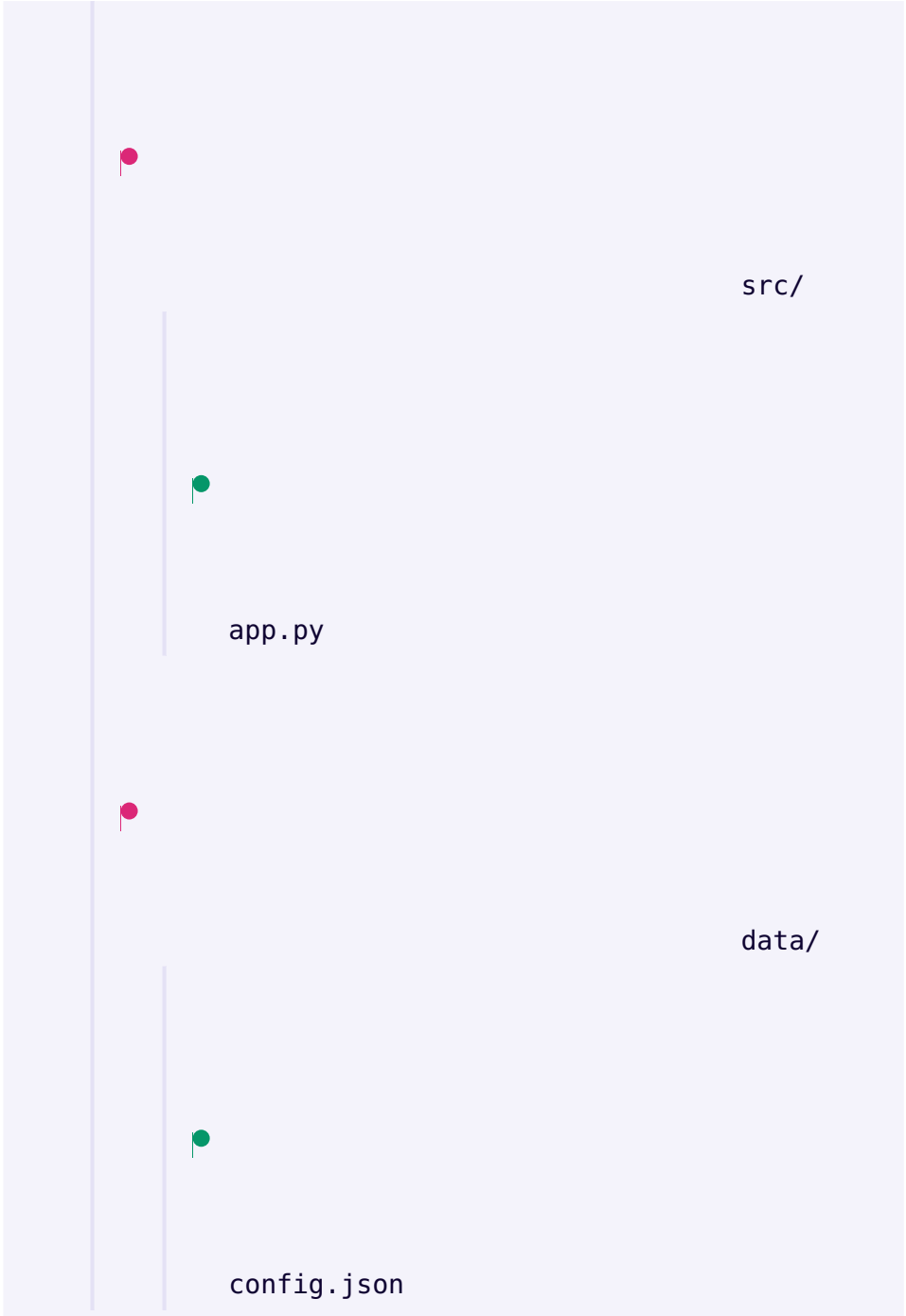
cwd.py

```
from pathlib import Path

print(Path.cwd())
```

Bieżący katalog roboczy (CWD) jest folderem względem którego, Zwykłe ścieżki względne w programie są interpretowane.

CWD NIE jest „folderem skryptów”
To jeden z najczęstszych błędów związanych ze ścieżkami w rzeczywistym kodzie. CWD oraz folder, w którym znajduje się plik `.py` , może się pokrywać lub nie. CWD jest determinowane przez fakt, że **skąd program został uruchomiony**, a nie tam, gdzie fizycznie znajduje się jej plik źródłowy.

project/ ← CWD (uruchomienie python src/app.py stąd)



Skrypt jest w src/app.py, ale CWD jest project/, jeśli uruchomisz program stąd.

app_py_fragment.py

```python
# w src/app.py jeśli program jest uruchamiany od project/:
from pathlib import Path

путь = Path("data/config.json")
# oznacza project/data/config.json — NIE project/src/data/
config.json
```

DEBUG LAB 2: CWD NIE JEST FOLDEREM SKRYPTÓW

app_wrong.py

```python
# plik jest w project/src/app.py
from pathlib import Path

config = Path("config.json")   # załóżmy: „obok
scenariusza”
print(config.read_text(encoding="utf-8"))
```

```
$ cd project
$ python src/app.py
Traceback (most recent call last):
FileNotFoundError: [Errno 2] No such file or directory: 'co
```

Co widać na ekranie

Plik `config.json` leży w `project/src/` obok skryptu. Ale program jest uruchamiany z `project/` oznacza, że CWD równa się `project/` oraz ścieżka względna `"config.json"` jest poszukiwane w `project/config.json` to nie istnieje.

POPRAWIONY KOD

app_fixed.py

```
# niezawodny sposób to ścieżka względem samego pliku, a
nie względem CWD
from pathlib import Path

BASE_DIR = Path(__file__).resolve().parent
config = BASE_DIR / "config.json"
print(config.read_text(encoding="utf-8"))
```

Folder skryptów: `__file__`

Dla zwykłych `.py`-plików, sprawdź folder, w którym był oryginał Możesz to zrobić tak:

script_dir.py

```
from pathlib import Path

BASE_DIR = Path(__file__).resolve().parent
```

__file__ nie zawsze jest gwarantowane

`__file__` jest zwykle definiowana w zwykłych skryptach i modułach, ale interaktywna powłoka Python i niektóre środowiska laptopowe mogą nie ustawiać tej zmiennej. Nie uważaj jej za uniwersalnie dostępną w żadnym kontekście wykonywania kodu.

Praktyka: CWD i folder skryptów na komputerze

Praktyka lokalna – uruchamiaj prawdziwy plik . py- z różnych folderów i obserwuj, co się zmienia

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/15-07/index.html)

pathlib.Path: ścieżki jako obiekty

Ścieżka to nie tylko tekst. To obiekt z własnymi atrybutami i metodami.

Dlaczego pathlib

Ścieżka nie jest tylko łańcuch znaków, ona jest **dziedzina przedmiotowa ze swoimi zasadami**: Ma rodzica, nazwę, rozszerzenie, może istnieć lub nie. Zamiast ręcznie złóż sznurek przez `+`:

Ścieżka jako łańcuch znaków → ścieżka jako obiekt

Klasyczne podejście

```
file_path = "data" + "/" +
"players" + "/" + "anna.json"
# na Windows musisz użyć „\\”
```

Nowoczesne Python (pathlib)

```
from pathlib import Path

file_path = Path("data") /
"players" / "anna.json"
# sam wstawi właściwy
separator dla bieżącego
systemu operacyjnego
```

Co używać dzisiaj: `pathlib` pojawił się w Python 3.4 i od tego czasu jest uważany za preferowany sposób pracy ze ścieżkami: eliminuje ręczne składanie strun, błędy ograniczników oraz dodaje wygodne metody tuż obok ścieżki.

Path nie jest zawartością pliku

Path(...) nie czyta pliku

`Path("notes.txt")` tworzy obiekt ścieżki — odniesienie do miejsca w systemie plików. Nie odczytuje ani nie otwiera niczego samodzielnie, dopóki nie wywołasz metody takiej jak `.open()` lub `.read_text()`.

Path jako obiekt: atrybuty i metody

To bezpośrednie powiązanie z Rozdziałem 14: `Path` — zwykły obiekt Python- z własnymi atrybutami i metodami.

pathlib.Path jest obiektem z metodami, a nie tylko tekstem ścieżki (pełną analizę atrybutów można znaleźć w 15.9).

Path.home()

path_home.py

```
from pathlib import Path

print(Path.home())   # tylko dla odniesienia
```

Nie zapisuj plików ćwiczeń do folderu domowego
`Path.home()` przydatny jako odniesienie, ale wszystkie ćwiczenia tego kursu zapisują pliki tylko w własnym katalogu roboczym — nigdy bezpośrednio w folderze domowym użytkownika.

Praktyka: Tworzenie ścieżek przez pathlib
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji
Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/15-08/index.html)

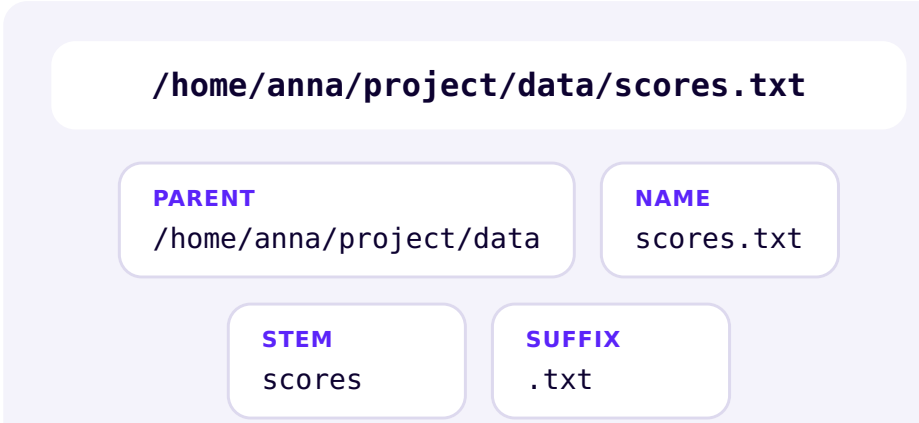
Rozkładanie ścieżki: name, stem, suffix, parent

Ścieżka, podobnie jak każdy obiekt, ma własne atrybuty — i odpowiadają one na konkretne pytania dotyczące pliku.

Dzielimy ścieżkę na części

razbor_puti.py

```
from pathlib import Path

path = Path("/home/anna/project/data/scores.txt")

print(path.name)     # scores.txt
print(path.stem)     # scores
print(path.suffix)   # .txt
print(path.parent)   # /home/anna/project/data
```



name = stem + suffix; parent jest folderem nadrzędnym.

Punkt i Dwa Punkty

- `.` — bieżący folder.
- `..` to folder nadrzędny.

dot_dotdot.py

```
from pathlib import Path

print(Path("."))
print(Path(".."))
```

Nie przesadzaj... /.. /.. /..

Długie łańcuchy `..` trudno odczytać i łatwo zepsuć podczas przenoszenia kodu gdzie indziej. Tam, gdzie to możliwe, lepiej używać wyraźnego folderu bazowego (jak `BASE_DIR` z sekcji 15.7) zamiast kilku `..` z rzędu.

Ekspansja to umowa, a nie gwarancja

Sufiks nie gwarantuje formatu zawartości

`.txt`, `.json`, `.jpg` są konwencjami nazw plików, a nie magiczną kontrolą tego, co jest w środku. Plik `report.txt` technicznie może zawierać wszystko. Nie sprawdzaj formatu treści tylko po rozszerzeniu pliku.

Praktyka: name, stem, suffix, parent
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji
Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/15-09/index.html)

Praktyka: Ścieżki i CWD

Zanim przejdziemy do samego pliku — utrwalmy pewność w obsłudze ścieżek.

Zbierzmy wszystko, co teraz wiemy o ścieżkach: ścieżkach absolutnych i względnych, CWD, `pathlib.Path` i rozkładanie ścieżki na części.

puteshestvie_po_putyam.py

```
from pathlib import Path

BASE_DIR = Path(__file__).resolve().parent if "__file__" in
globals() else Path.cwd()
data_dir = BASE_DIR / "data"
data_dir.mkdir(exist_ok=True)

file_path = data_dir / "otchet.txt"
print(„Ścieżka absolutna:”, file_path.resolve())
print(„Imię:”, file_path.name, „| Rozszerzenie:”,
file_path.suffix)
print(„Istnieje?”, file_path.exists())
```

Path.resolve()
`path.resolve()` tworzy absolutną, zezwoloną wersję ścieżki — przydatną do debugowania, by zobaczyć, dokąd faktycznie prowadzi ścieżka. To nie jest narzędzie bezpieczeństwa — nie polegaj na niej jako na „sanitizatorze”

Podstawowa praktyka
Ścieżka do siebie
Wyjście BASE_DIR, . parent z BASE_DIR i . parent.parent z BASE_DIR — zobacz, jak wspinać się po drzewie folderów.

★★ Zadanie samodzielne

Sprawdzanie przed czytaniem

Zbuduj ścieżkę do nieistniejącego pliku w data/ i, używając .exists(), wyświetl jedno z dwóch komunikatów: „plik znaleziony” lub „pliku nie ma — najpierw go utwórz”.

Praktyka: Ścieżki i CWD – Szkolenie końcowe

interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/15-10/index.html)

open() zwraca obiekt pliku

Plik w Python nie jest magią, lecz obiektem o własnym stanie i metodach.

open() zwraca obiekt

To kolejny bezpośredni link do rozdziału 14. Wynik `open(...)` — nie jest to „magiczny kanał”

file_object.py

```
with open("privet.txt", "r", encoding="utf-8") as file:
    print(type(file))
```

file — egzemplarz klasy odpowiedzialnej za czytanie/zapisywanie tekstu konkretnego otwartego pliku.

Zmienna `file` w naszych przykładach jest nazwa, którą sami sobie robimy wybierz; ważne jest, że odnosi się do obiektu o jego stanie (`.closed` , `.mode` , `.name`) i metody (`.read()` , `.write()` i innymi).

Nie pamiętam dokładnej hierarchii klas
Python mają oddzielne klasy wewnętrzne dla plików tekstowych i binarnych, ale do codziennej pracy wystarczy wiedzieć: `open()` zwraca obiekt pliku z przewidywalnym zestawem metod — dokładna hierarchia klas wejścia-wyjścia nie jest potrzebna do zapamiętania.

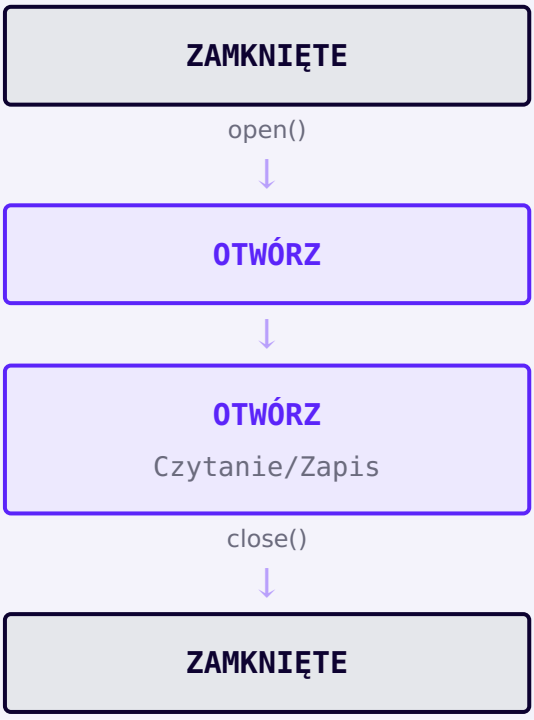
Praktyka: Obiekt pliku i jego atrybuty
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/15-11/index.html)

Cykl życia pliku i with

with nie jest tylko wygodną składnią, ale gwarantuje poprawne zamknięcie zasobu.

Cykl życia pliku



Cykl życia manualny: otwórz → pracę → pamiętaj, żeby zamknąć.

Z menedżerem kontekstu

with gwarantuje zamknięcie nawet jeśli wewnątrz bloku występuje błąd.

with_protokol.py

```
with open("privet.txt", "r", encoding="utf-8") as file:
    print(file.closed)   # False – plik otwarty wewnątrz bloku
print(file.closed)
#True – zamyka się podczas opuszczania bloku
```

with to nie tylko „modna składnia”

`with` pojawił się w Python 2.5 — nie chodzi tu o nowość, lecz o niezawodne zarządzanie zasobami: nawet jeśli wyjątek pojawi się wewnątrz bloku, protokół menedżera kontekstu i tak zamknie plik w sposób elegancki.

Co dzieje się pod maską

Jest częścią modelu obiektowego Python: obiektów zaangażowanych w `with`, wspierać specjalne zachowanie przy wchodzeniu i wyjściu z bloku - metody `__enter__` oraz `__exit__`. Nie musisz umieć Pisz własne takie obiekty — wystarczy wiedzieć, że pliki to tylko jedno z zastosowań tego a nie jakiejś funkcjonalności wymyślonej specjalnie dla plików.

Ręczne zamknięcie jest opcją historyczną

ruchnoe_zakrytie.py

```
file = open("privet.txt", "r", encoding="utf-8")
content = file.read()
file.close()   # nie zapomnieć — i wykonać nawet w przypadku wcześniejszego błędu
```

with jest nie tylko nowszy, ale i bardziej niezawodny

Ręczne otwieranie/zamykanie nadal działa i spotyka się w starym kodzie, ale `with` zalecane nie dlatego, że jest nowszy, ale dlatego, że zamknięcie jest gwarantowane, nawet jeśli wystąpi błąd.

Praktyka: cykl życia pliku

interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/15-12/index.html)

Kursor pliku: tell() i seek()

Otwarty plik ma swoją aktualną pozycję — porusza się do przodu, ale nie wraca do siebie.

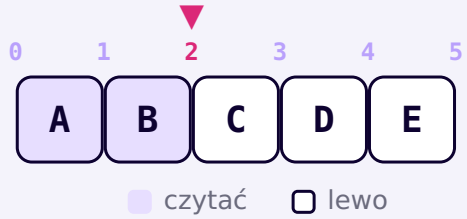
Kursor pliku

Otwarty plik zawiera aktualne **pozycja odczytu/zapisu** jest kursorem. On przesuwa się do przodu podczas odczytu lub zapisu:

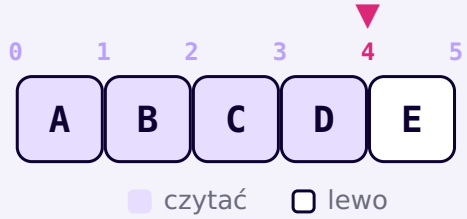
kursor.py

```
with open("alfavit.txt", "w", encoding="utf-8") as f:
    f.write("ABCDE")

with open("alfavit.txt", "r", encoding="utf-8") as f:
    print(f.read(2))   #AB — Kursor przesunął się 2
    print(f.read(2))   #CD — kursor przesunął się o kolejne 2
    print(f.read())    #E — do końca pliku
    print(repr(f.read()))   # '' — kursor już na końcu
```

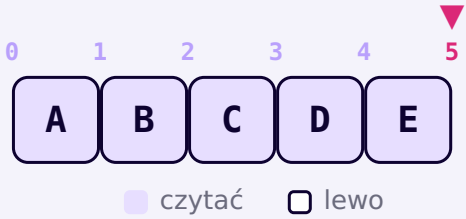
kursor przed pierwszym read(2): pozycję 0.



Po read(2): kursor na pozycji 2, odczytano «AB».



Po drugiej read(2): kursor na pozycji 4, odczytano «ABCD».



Po read(): kursorze na końcu pliku (EOF), pozycja 5.

DEBUG LAB 3: KURSOR JEST JUŻ NA KOŃCU PLIKU

posle_eof.py

```
with open("alfavit.txt", "r", encoding="utf-8") as f:
    print(f.read())
    print(f.read())   # oczekujemy zobaczyć to samo
jeszcze raz?
```

```
ABCDE
(пустая строка)
```

Co widać na ekranie

Drugi `f.read()` zwrócił pusty łańcuch znaków, a nie „ABCDE” ponownie. Kursor sam nie wraca na początek — po pierwszym `read()` pozostał na końcu pliku (EOF), i tam już nic do czytania.

POPRAWIONY KOD

vozvrat_v_nachalo.py

```
with open("alfavit.txt", "r", encoding="utf-8") as f:
    print(f.read())
    f.seek(0)          # ręcznie przywróć kursor do
początku
    print(f.read())    # teraz znowu ABCDE
```

tell() i seek()

tell_seek.py

```
with open("alfavit.txt", "r", encoding="utf-8") as f:
    f.read(2)
    print(f.tell())   #2 to aktualna pozycja kursora
    f.seek(0)         # Powrót na górę
    print(f.read())   # ABCDE — ponowne czytanie
```

seek() w trybie tekstowym nie jest „numerem znaku”

Dla prostych przykładów z ASCII-text, przesunięcie zwracające `tell()`, intuicyjnie odpowiada liczbie przeczytanych znaków. Ale w ogólnym przypadku dla tekstu w UTF-8 to nieprawda: jeden znak (szczególnie cyrylica lub emoji) może zajmować kilka bajtów (rozdział 15.16), a dowolne przesunięcia nie odpowiadają bezpośrednio indeksom znaków. Dla dowolnych przesunięć bajtowych jaśniej i bezpieczniej jest pracować w trybie binarnym (rozdział 15.17). Bezpieczne użycie `seek()` w trybie tekstowym — `seek(0)` zacząć od nowa.

Praktyka: Kursor plików, tell() i seek()

interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/15-13/index.html)

Przeczytaj akta: read(), readline(), readlines()

Ten sam plik można odczytać na różne sposoby – a wybór metody jest wyborem, a nie formalnością.

read() — cały plik

read_ves.py

```
with open("spisok.txt", "r", encoding="utf-8") as f:
    content = f.read()
```

read() nie zawsze jest zły
W przypadku małych plików czytanie wszystkiego naraz jest proste i wygodne. Dla bardzo dużych plików (gigabajty logów) może to zajmować dużo pamięci — wtedy lepiej czytać linię po linii lub w fragmentach (sekcja 15.23). Nie ma jednego rozwiązania, które zawsze jest właściwe — ważne jest, by wybrać dane zadanie.

readline() — linijka po linii

readline.py

```
with open("spisok.txt", "r", encoding="utf-8") as f:
    print(f.readline())   # pierwszy łańcuch znaków
    print(f.readline())   # drugi łańcuch znaków
    # ...
    print(repr(f.readline()))
# '' — kiedy łańcuch znaków już nie ma
```

DEBUG LAB 4: READ() ZAMIAST READLINE()

odna_stroka.py

```
with open("spisok.txt", "r", encoding="utf-8") as f:
    first_line = f.read()   # chciałem jednej linii
    second_line = f.read()
    print(„Pierwsza:”, first_line)
    print(„Drugi:”, repr(second_line))
```

```
Первая: яблоки\nхлеб\nмолоко\nсыр\n
Вторая: ''
```

Co widać na ekranie

Każde połączenie miało odpowiedzieć na jedną linię — ale `read()` czyta bez sprzeciwu **cały pozostały plik**, a nie jeden wiersz. Kursor od razu znalazł się na końcu, a drugie wywołanie zwróciło pusty łańcuch znaków.

POPRAWIONY KOD

odna_stroka_fixed.py

```
with open("spisok.txt", "r", encoding="utf-8") as f:
    first_line = f.readline()
    second_line = f.readline()
    print(„Pierwsza:”, first_line.strip())
    print(„Drugi:”, second_line.strip())
```

Pętla plików jest preferowaną metodą

cikl_po_fajlu.py

```
with open("spisok.txt", "r", encoding="utf-8") as f:
    for line in f:
        print(line.strip())
```

Obiekt pliku — iterowalny
Podobnie jak lista czy łańcuch znaków (Rozdział 10), obiekt pliku może być zapętlony `for` bezpośrednio Python czyta linię po linii, nie ładując całego pliku do pamięci.

readlines() — wszystkie linie w postaci listy

readlines_spisok.py

```
with open("spisok.txt", "r", encoding="utf-8") as f:
    lines = f.readlines()

print(len(lines), „linie”)
```

Metoda	Co wraca	Pamięć
`for line in file`	po jednej linii na raz	oszczędnie — nadaje się do dużych plików
`file.readlines()`	lista wszystkie linie na-raz	wszystkie lista linie w pamięci jednocześnie

Praktyka: read(), readline(), readlines()
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji
Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/15-14/index.html)

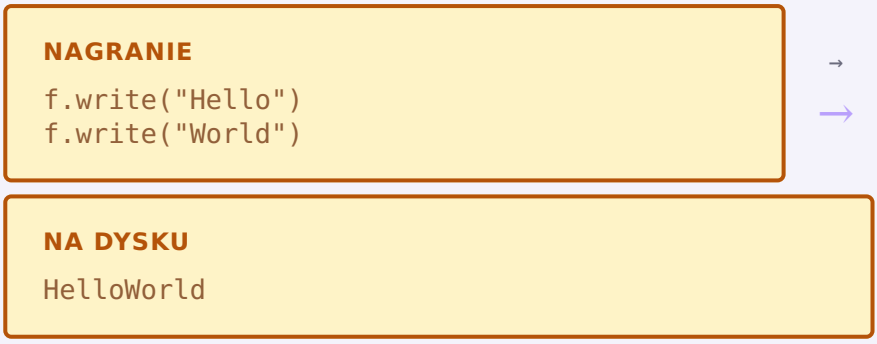
Zapis do plików: write() i tryby r/w/a/x

Każdy tryb otwarcia to świadomy wybór, a nie szczegół składni.

write() sam nie dodaje linii

write_bez_n.py

```
with open("log.txt", "w", encoding="utf-8") as f:
    f.write("Hello")
    f.write("World")
```



write() nie wstawia \n samego — jeśli potrzebujesz podziału linijki, zapisz go wyraźnie: f.write(„Hello\n”

Tryby otwarcia - Przegląd

Cztery podstawowe tryby

OPEN(PATH, „R”

Odczyt.

Błąd, jeśli plik nie istnieje.

OPEN(PATH, „W”

Zapis.
Tworzy nowy plik lub czyści istniejący
JUŻ PRZY OTWIERANIU.

OPEN(ŚCIEŻKA, „A”

Dodatkowe nagranie na końcu.
Tworzy plik, jeśli go nie ma.

OPEN(PATH, „X”

Tworzy nowy plik.
FileExistsError jeśli plik już istnieje.

Następnie dodane są t/b/+ modyfikatory (od sekcji 15.17).

DEBUG LAB 5: „W”

perezapis.py

```
#rezultaty.txt już zawiera ważne wyniki z poprzedniej
gry
with open("rezultaty.txt", "w", encoding="utf-8") as f:
    pass   # nawet nic nie nagrałem
```

```
(файл rezultaty.txt теперь пуст)
```

Co widać na ekranie

Plik stał się pusty, mimo że nie zapisaliśmy nic wewnątrz bloku. Otwieranie w trybie `"w"` kasuje wcześniejszą zawartość **w momencie otwarcia**, i to nie w momencie pierwszego `write()`.

POPRAWIONY KOD

dozapis_vmesto_w.py

```
# Jeśli celem jest dodawanie, a nie zastępowanie,
potrzebujesz trybu "a", a nie "w"
with open("rezultaty.txt", "a", encoding="utf-8") as f:
    f.write(„Nowy wynik\n”)
```

Tryb „x”

rezhim_x.py

```
try:
    with open("save.json", "x", encoding="utf-8") as f:
        f.write("{}")
except FileExistsError:
    print(„Plik zapisu już istnieje — nie nadpisuj go
przypadkiem”)
```

x - Ochrona przed przypadkowym nadpisaniem

Jeśli ważne jest, aby nie nadpisać istniejącego pliku przez pomyłkę – `"x"` będzie to wyraźnie komunikować za pomocą `FileExistsError`, zamiast cichego nadpisywania, które zrobiłby tryb `"w"`.

DEBUG LAB 6: WRITELINES() NIE DODAJE PRZERW MIĘDZY LINIAMI

writelines_lovushka.py

```
stroki = ["Anna", "Bob", "Carlos"]
with open("igroki.txt", "w", encoding="utf-8") as f:
    f.writelines(stroki)
```

```
(файл igroki.txt содержит одну строку: AnnaBobCarlos)
```

Co widać na ekranie

Oczekiwano trzech linii, ale okazało się, że była to jedna połączona łańcuch znaków. `writelines()` — mimo nazwy — **nie dodaje sama nowych liniipo** prostu zapisuje elementy pojedynczo. Jeśli potrzebne są oddzielne linie, przewod linii musi być uwzględniony w każdym elemencie wcześniej.

POPRAWIONY KOD

writelines_fixed.py

```
stroki = ["Anna", "Bob", "Carlos"]
with open("igroki.txt", "w", encoding="utf-8") as f:
    f.writelines(s + "\n" for s in stroki)
```

Trochę głębiej:

Modyfikatory +

`r+`, `w+`, `a+` od razu otworzyć plik do czytania i zapisu — ale podstawowa semantyka Obcinaj/nadpisuj każdy tryb (`w` myje, `a` dodaje na końcu) nadal obowiązuje tutaj. Jest to opcjonalne Materiał – Podstawowy `r` / `w` / `a` / `x` wystarczające dla większości programów.

Praktyka: Tryby write() i r/w/a/x
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/15-15/index.html)

Tekst, bytes i kodowanie UTF-8

Pliki przechowują bajty. Sposób, w jaki bajty są zamieniane na tekst, to kodowanie i nie jest to domyślnie uniwersalne.

Tekst i bytes to różne typy

str_vs_bytes.py

```
print(type("Python"))     # <class 'str'>
print(type(b"Python"))    # <class 'bytes'>
```

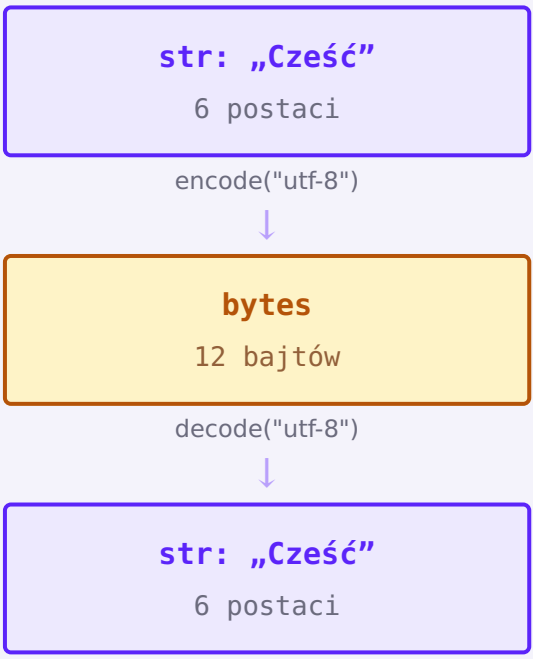
bytes nie jest „łańcuch znaków bez Unicode”
`bytes` jest łańcuch znaków liczb całkowitych od 0 do 255 (dane binarne), a nie skróconą wersją łańcuch znaków. Tekst łańcuch znaków (`str`) oraz dane binarne (`bytes`) — dwa różne typy o różnym przeznaczeniu.

Kodowanie i dekodowanie

Aby zapisać tekst w pliku, jego reprezentacja łańcuch znaków jest zamieniana na bajty — to jest jest nazywana **kodowaniem (encode)**. Podczas czytania dzieje się odwrotnie — **Dekodowanie (decode)**:

encode_decode.py

```
text = „Cześć”
b = text.encode("utf-8")
print(b)              # b'\\xd0\\x9f\\xd1\\x80\\xd0\\xb8\\xd0\
\xb2\\xd0\\xb5\\xd1\\x82'
print(len(text))      #6 – Sześć postaci
print(len(b))         #12 – dwanaście bajtów!

obratno = b.decode("utf-8")
print(obratno)        #Hello
```



Alfabet cyrylicy w UTF-8 zajmuje więcej bajtów niż znaki — to prawdziwy przykład.

Unicode ≠ UTF-8
Unicode jest systemem, który mapuje kody numeryczne na znaki (który symbol oznacza jaką liczbę). UTF-8 jest tylko jednym ze sposobów zapisu tych liczb w bajtach; istnieją też inne kodowania. W tym kursie zawsze wyraźnie wybieramy UTF-8 jako standard dla plików tekstowych — ale to wybór, nie jedyna opcja.

Gdy format tekstu jest znany wcześniej jako UTF-8 (co prawie zawsze ma miejsce na tym kursie), Wyraźne wskazanie `encoding="utf-8"` wpływa na zachowanie programu przewidywalny i przenośny – zamiast polegać na domyślnym kodowaniu, co jest determinowane przez ustawienia platformy i środowiska wykonawczego.

Dlaczego kodowanie jest ważne

DEBUG LAB 7: PLIK OTWARTY BEZ WSKAZANIA ENCODING

bez_encoding.py

```
with open("privet.txt", "w") as f:   # bez encoding!
    f.write(„Cześć!”)
```

```
# Работает по-разному на разных компьютерах и системах —
# кодировка по умолчанию НЕ гарантированно UTF-8 везде.
```

Co widać na ekranie

Bez jawnego `encoding="utf-8"` Python używa kodowania domyślnego, zależnego od platformy i ustawień środowiska — a nie gwarantowanie UTF-8. Plik zapisany na jednym komputerze może być błędnie odczytany na innym, jeśli ich domyślne kodowania się różnią.

POPRAWIONY KOD

s_encoding.py

```
with open("privet.txt", "w", encoding="utf-8") as f:
    f.write(„Cześć!”)   # kodowanie jest jawne i
przewidywalne przez cały czas
```

DEBUG LAB 8: UNICODEDECODEERROR: BŁĘDNE KODOWANIE PODCZAS ODCZYTU

nevernaya_kodirovka.py

```
data = „Cześć”.encode("utf-8")
with open("soobshenie.bin", "wb") as f:
    f.write(data)

# czytane jako tekst w INNYM kodowaniu
with open("soobshenie.bin", "r", encoding="ascii") as f:
    print(f.read())
```

```
Traceback (most recent call last):
UnicodeDecodeError: 'ascii' codec can't decode byte 0xd0 in p
```

Co widać na ekranie

bajty były kodowane w UTF-8 i odczytywane tak, jakby ASCII, kodowania nie pasowały i nie Python mógł przekształcić tych bajtów z powrotem w poprawne znaki. Rozwiązaniem jest odczytanie pliku w tym samym kodowaniu, w którym został zapisany.

POPRAWIONY KOD

pravilnaya_kodirovka.py

```
with open("soobshenie.bin", "r", encoding="utf-8") as f:
    print(f.read())   #Hello
```

Trochę głębiej:

parametr errors=

U `open()` jest parametrem `errors` (`"strict"` też domyślnie `"replace"`, `"ignore"`) w przypadkach niezgodności kodu.

errors=„ignore”
`errors="ignore"` cicho wyrzuca nierozpoznane bajty — może to niezauważalnie spowodować utratę części danych. Nie używaj tego jako sposobu na „usunięcie błędu”

Praktyka: tekst, bytes i UTF-8
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/15-16/index.html)

Binarki i podziały linii

Nie wszystko, co znajduje się w pliku, to tekst. A nawet tekst zajmuje na dysku więcej, niż się wydaje.

Tryb binarny

Dodaj `b` w tryb, aby pracować z plikiem jako czystymi bajtami, bez jakiegokolwiek kodowania tekstowego:

binarny_fajl.py

```
data = bytes([0, 1, 2, 3, 255])

with open("signal.bin", "wb") as f:
    f.write(data)

with open("signal.bin", "rb") as f:
    print(f.read())   # b'\\x00\\x01\\x02\\x03\\xff'
```

Mode	Co dostajemy w Python	Przykład
`"rt" / "r"`	str (tekst)	"Python"
`"rb"`	bytes (dane binarne)	b'Python'

Nie dekoduj dowolnych danych binarnych jako tekst
PNG, JPEG lub PDF nie będą czytelne z `open(..., encoding="utf-8")` — to nie są dane tekstowe i próba odczytu ich jako tekstu spowoduje błąd lub śmieci. Do takich formatów używaj trybu binarnego (`"rb"`) lub specjalistyczna biblioteka dla tego formatu.

DEBUG LAB 9: DANE BINARNE ODCZYTYWANE JAKO TEKST

bin_kak_text.py

```
data = bytes([0, 1, 2, 3, 255])
with open("signal.bin", "wb") as f:
    f.write(data)

with open("signal.bin", "r", encoding="utf-8") as f:
    print(f.read())
```

```
Traceback (most recent call last):
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xff in
```

Co widać na ekranie

bajtów `0xff` nie tworzy prawidłowego znaku w UTF-8 — te dane nigdy nie były tekstem. Plik musi być otwarty w trybie binarnym (`"rb"`), a nie w formie tekstowej.

POPRAWIONY KOD

bin_pravilno.py

```
with open("signal.bin", "rb") as f:
    print(f.read())   # b'\\x00\\x01\\x02\\x03\\xff'
```

Zawijanie wierszy i tryb tekstowy

\n wewnątrz Python jest uniwersalny kanał przewodowy

Na różnych systemach operacyjnych historycznie stosowano różne sekwencje do przechodzenia do nowej linii na dysku (np., `\r\n` w plikach utworzonych w Windows). trybie tekstowym Python odczytuje takie pliki w trybie uniwersalnych podziałów linii i przekazuje je Ci we wszystkich przypadkach `\n` — nie musicie przetwarzać obu wariantów ręcznie w zwykłym kodzie tekstowym.

Postacie kontra bajty – przykład z rzeczywistego świata

simvoly_vs_bajty.py

```
text = „Python  ”
print(len(text))                  # 6 — liczba znaków
print(len(text.encode("utf-8")))  #14 to liczba bajtów w UTF-8
```

Długość tekstu ≠ rozmiar pliku w bajtach
cyrylicy i emoji zajmują kilka bajtów na znak w UTF-8. `len(text)` liczy znaki, a nie bajty — rozmiar pliku na dysku może być znacznie większy niż liczba znaków w łańcuch znaków.

Praktyka: pliki binarne i przenoszenie linii
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/15-17/index.html)

pathlib: read_text, write_text, read_bytes, write_bytes

Dla "przeczytaj wszystko" i "zapisz wszystko" pathlib daje krótki zapis – ale nie anuluje open().

Wygodne metody pathlib dla całego pliku

read_write_text.py

```
from pathlib import Path

path = Path("nastroyki.txt")
path.write_text(„Cześć!”, encoding="utf-8")
print(path.read_text(encoding="utf-8"))   #Hello!
```

read_write_bytes.py

```
from pathlib import Path

path = Path("signal.bin")
path.write_bytes(bytes([0, 1, 2]))
print(path.read_bytes())   # b'\\x00\\x01\\x02'
```

write_text()/write_bytes() zastąpić całą zawartość
A także reżim `"w"` te metody całkowicie zastępują zawartość pliku — nie nadają się wtedy, gdy potrzebne jest dodatkowe zapisy.

open() vs wygodne metody pathlib – kiedy co

Co wybrać dla konkretnych zadań

Mały plik w całości, bez przetwarzania strumieniowego	→ `path.read_text()` / `write_text()`
Wymaga przetwarzania linia po linii lub dużego pliku	→ `with path.open(...)` jak w sekcjach 15.11
Całkowicie binarne dane bez strumienia	→ `path.read_bytes()` / `write_bytes()`
Potrzebny jest specjalny system, na przykład dodatkowe nagranie	→ `open(path, „a”`

pathlib nie zastępuje open() całkowicie

`Path` dodaje wygodne metody częstego czytania/zapisu całego pliku — ale przetwarzanie liniowe i specjalne tryby nadal zachowują `open()` / `path.open()` . pathlib nie „zastępuje open() i nie czyni go zbędnym” — rozwiązują one różne zadania.

Praktyka: read_text, write_text, read_bytes, write_bytes
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji
Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/15-18/index.html)

Foldery: exists(), is_file(), mkdir()

Zanim zaczniesz pracować ze ścieżką, często musisz zrozumieć, co jest przed tobą i utworzyć brakujące foldery.

Sprawdzamy, co mamy przed sobą

exists_is_file_is_dir.py

```
from pathlib import Path

p = Path("data")
print(p.exists())    # czy w ogóle istnieje jakaś ścieżka
print(p.is_file())   # to zwykły plik?
print(p.is_dir())    # czy to jest folder?
```

Trzy wyniki walidacji ścieżek

exists() → False → **ścieżka nie istnieje**

exists() → True, is_file() → True → **zwykłym plikiem**

exists() → True, is_dir() → True → **folder**

exists() nie gwarantuje przyszłości
Między kontrolami `if path.exists():` i kolejnej operacji system plików może się zmienić — plik może zostać usunięty przez inny proces, a uprawnienia mogą się różnić. Kontrola istnienia jest przydatna dla logiki programu, ale nie dowodzi, że następna operacja koniecznie zakończy się niepowodzeniem (wrócimy do tego w sekcji 15.22).

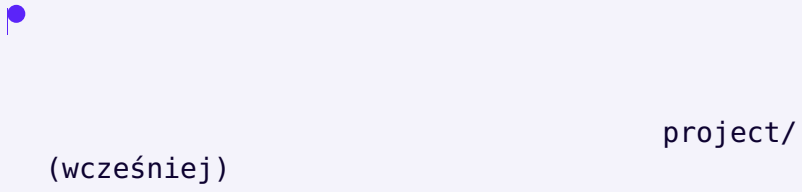
Tworzenie folderów

mkdir_prostoj.py

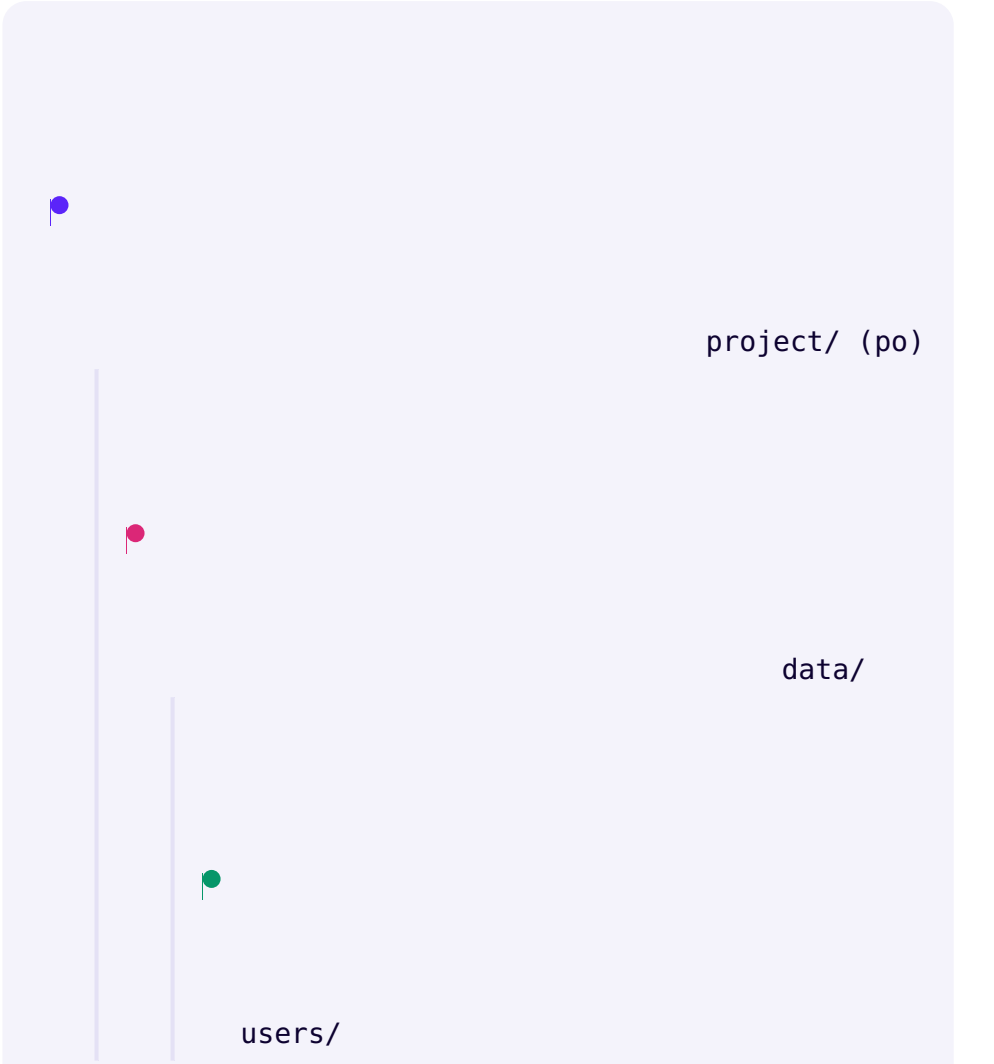
```
from pathlib import Path

Path("data").mkdir()
```

mkdir_nadezhny.py

```
from pathlib import Path

Path("data/users").mkdir(parents=True, exist_ok=True)
# parents=True – utwórz wszystkie pośrednie foldery, jeśli
jeszcze nie istnieją
# exist_ok=True – nie uważać za błąd, jeśli folder już istnieje
```



Wcześniej: project/ jest pustym projektem.



Po Path(data/users"). mkdir(parents=True, exist_ok=True): Oba foldery zostały utworzone podczas jednego połączenia.

DEBUG LAB 10: PISANIE DO NIEISTNIEJĄCEGO FOLDERU

propushena_papka.py

```
from pathlib import Path

path = Path("data/users/anna.json")
path.write_text("{}", encoding="utf-8")   # foldery
data/users jeszcze nie ma!
```

```
Traceback (most recent call last):
FileNotFoundError: [Errno 2] No such file or directory: 'da
```

Co widać na ekranie

Plik można utworzyć tylko w folderze, który już istnieje. `write_text()` / `open(..., "w")` tworzą plik, ale nie tworzą brakujących folderów nadrzędnych automatycznie.

POPRAWIONY KOD

papka_zagotovlena.py

```
from pathlib import Path

path = Path("data/users/anna.json")
path.parent.mkdir(parents=True, exist_ok=True)   #
Najpierw foldery
path.write_text("{}", encoding="utf-8")            #
potem — plik
```

Praktyka: exists(), is_file(), mkdir()
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/15-19/index.html)

Wyszukiwanie plików: iterdir() i glob()

Często nie musisz otwierać konkretnego pliku, ale najpierw zrozum, co znajduje się w folderze.

Lista zawartości folderów

iterdir.py

```
from pathlib import Path

for item in sorted(Path("data").iterdir()):
    print(item.name, "—", „folder” if item.is_dir() else
„plik”)
```

Porządek nie jest gwarantowany
`iterdir()` nie obiecuje żadnego konkretnego porządku elementów. Jeśli kolejność jest ważna dla wyjścia lub testowania, należy opakować `sorted(...)` jak w powyższym przykładzie.

Wyszukaj pliki według szablonu: glob()

glob_prosty.py

```
from pathlib import Path

for f in sorted(Path("data").glob("*.txt")):
    print(f.name)
```

Złóż data/	Pasuje pod „*.txt”
a.txt	Tak
b.csv	nie
c.txt	Tak
notes.md	nie

glob() jest jednym poziomem, rglob() jest rekurencyjny
`Path("data").glob("*.txt")` szuka tylko w samej folderze `data`. Aby przeszukiwać również wszystkie podfoldery, użyj `Path("data").rglob("*.txt")`. Nie uruchamiaj rekurencyjnego wyszukiwania na dużych folderach systemowych — to nie jest zabawka typu „skanuj cały dysk”

Mini-projekt: raport według folderów

Zbieraj lista zawartość folderu uczącego się: nazwę, typ (plik/folder), rozszerzenie i rozmiar (dla plików) — używając tylko `iterdir()` i atrybuty `Path` bez rekurencyjnego usuwania lub modyfikowania treści.

otchet_po_papke.py

```
from pathlib import Path

def otchet(papka: Path) -> list[str]:
    stroki = []
    for item in sorted(papka.iterdir()):
        vid = „folder” if item.is_dir() else „plik”
        razmer = f", {item.stat().st_size} bajtów" if
item.is_file() else ""
        stroki.append(f"{item.name} — {vid}{razmer}")
    return stroki

for stroka in otchet(Path("data")):
    print(stroka)
```

★★ Zadanie samodzielne
Tylko. json
Zmień raport, aby wyświetlał tylko pliki z . json rozszerzeniem używając glob(„*. json”

Praktyka: iterdir(), glob() i raport folderu
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/15-20/index.html)

Zmiana nazw, kopiowanie, usuwanie

Te operacje naprawdę zmieniają system plików — i nie zawsze są odwracalne.

Zmiana nazwy

rename.py

```
from pathlib import Path

old_path = Path("chernovik.txt")
old_path.rename("gotovo.txt")
```

Operacje z plikami mogą zawiódć
Zmiana nazw, kopiowanie i usuwanie to dostęp do systemu operacyjnego, a nie gwarantowane sukcesy Python. Mogą zawiódć z wielu powodów związanych z systemem plików lub uprawnieniami (Rozdział 15.22) — nie należy ich koniecznie traktować jako niepowodzenie.

replace() dla „Zamień plik docelowy”

replace.py

```
from pathlib import Path

novy_fajl = Path("save_new.json")
tselevoy_fajl = Path("save.json")
novy_fajl.replace(tselevoy_fajl)   # zastępuje plik docelowy, jeśli już istnieje
```

Wrócimy do tego w sekcji 15.28 — to sedno wzorca bezpiecznej trwałości „Najpierw tymczasowy plik, potem wymień główny.”

Kopiowanie jest shutil

shutil_copy.py

```
import shutil

shutil.copy("save.json", "save_backup.json")
```

pathlib i shutil się uzupełniają
`pathlib` świetnie działa z samymi ścieżkami, a wysokopoziomowe kopiowanie plików zazwyczaj przejmuje moduł standardowej biblioteki `shutil` (`shutil.copy`, `shutil.copy2`, `shutil.move`).

Usuwanie – z najwyższą starannością

Usunięcie to druzgocąca operacja
`path.unlink()` usuwa plik **bez możliwości cofnięcia** za pomocą narzędzi samego programu. `path.rmdir()` usuwa tylko **pusty** folder. W tym kursie nigdy nie używamy rekurencyjnego usuwania (`shutil.rmtree`) jako ćwiczenie samouka — a wszystkie przykłady usuwania w praktyce działają tylko z plikami utworzonymi przez praktykę w osobnym folderze roboczym, nigdy z dowolnymi ścieżkami użytkownika.

unlink_bezopasno.py

```
from pathlib import Path

vremenny_fajl = Path("chernovik_kopiya.txt")
if vremenny_fajl.exists():
    vremenny_fajl.unlink()
```

Praktyka: Przemianuj, kopiuj, usuń
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/15-21/index.html)

Błędy systemu plików

Operacje na plikach odwołują się do świata zewnętrznego — a świat zewnętrzny nie zawsze odpowiada tak, jakby się chciało.

Częste błędy systemu plików

Wyjątek	Kiedy powstaje
`FileNotFoundError`	ścieżka nie istnieje, a operacja czekała na istniejący plik
`FileExistsError`	tryb „x”
`PermissionError`	nie ma uprawnień do odczytu/zapisu na tej ścieżce
`IsADirectoryError`	ścieżka to folder, a operacja czekała na plik
`NotADirectoryError`	ścieżka to plik, a operacja czekała na folder
`UnicodeDecodeError`	bajtów nie można odkodować przy użyciu wybranego kodowania (15.16)

wspólnym przodkiem jest OSError
Większość wyjątków plików to warianty `OSError`. Nie jest konieczne znajomość całej hierarchii klas wyjątków — wystarczy pewnie rozpoznać powyższe klasy po nazwie i zrozumieć, kiedy każda z nich się pojawia.

Checklista: plik nie znaleziono

1. Co teraz jest CWD? (`Path.cwd()`, rozdział 15.7)
2. Jaka dokładnie ścieżka została zbudowana w kodzie?
3. Czy istnieje folder nadrzędny tej ścieżki?
4. Czy jest literówka w nazwie?
5. Czy ścieżka jest absolutna czy względna, i względna do czego?
6. Czy sprawa w nazwie pliku jest taka sama?

Sprawa dotycząca nazwy pliku to nie jest drobnostka

Zachowanie wrażliwości na wielkie litery różni się w zależności od systemu plików. Nie zakładaj, że `Data.txt` oraz `data.txt` musi być tym samym plikiem: napisz kod tak, aby nazwa była idealna.

DEBUG LAB 11: OTWÓRZ FOLDER JAKO PLIK

papka_kak_fajl.py

```
from pathlib import Path

Path("arhiv").mkdir(exist_ok=True)
with open("arhiv", "r", encoding="utf-8") as f:
#„arhiv”
    print(f.read())
```

```
Traceback (most recent call last):
IsADirectoryError: [Errno 21] Is a directory: 'arhiv'
```

Co widać na ekranie

`"arhiv"` — folder, a nie plik, i nie można go otworzyć jako pliku do odczytu tekstu. Przed otwarciem warto było sprawdzić `.is_file()`.

POPRAWIONY KOD

proverka_pered_otkrytiem.py

```
from pathlib import Path

path = Path("arhiv")
if path.is_file():
    with path.open("r", encoding="utf-8") as f:
        print(f.read())
else:
    print(f"{path} — nie zwykły plik")
```

Radzić sobie dokładnie z takim błędem, jakiego się spodziewasz

targeted_except.py

```
from pathlib import Path

path = Path("nastroyki.json")
try:
    text = path.read_text(encoding="utf-8")
except FileNotFoundError:
    print(„Nie znaleziono pliku ustawień — użyj wartości
domyślnych”)
    text = "{}"
```

Nigdy nie except: pass

Puste przechwytywanie wszystkich wyjątków z rzędu po cichu ukrywa prawdziwe problemy programu. Złap konkretny oczekiwany wyjątek — jak `FileNotFoundError` wyżej — i nie wszystko w rzędzie.

Trochę głębiej: EAFP i LBYL

Dwa style: **sprawdź wcześniej** (`if path.exists():`) lub **próbować poradzić sobie z błędem** (`try/except`). W Python operacje plikowe często zapisywane są w drugi sposób, właśnie dlatego, że stan System plików może zmieniać się między sprawdzaniem a samą operacją (Rozdział 15.19). Pamiętaj Same angielskie skróty nie są konieczne — idea jest ważna.

Praktyka: błędy systemu plików
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/15-22/index.html)

Duże pliki i przetwarzanie strumieni

Sposób odczytu pliku to rozwiązanie, które musi uwzględniać jego rozmiar.

Rozmiar pliku

razmer_fajla.py

```
from pathlib import Path

path = Path("spisok.txt")
print(path.stat().st_size, „bajt”)
```

Rozmiar w bajtach nie jest długością tekstu
`.stat().st_size` — to rozmiar w bajtach na dysku, a nie liczba znaków. Dla tekstu w UTF-8 z cyrylicą lub emoji te liczby, jak widzieliśmy w sekcji 15.17, zazwyczaj się nie zgadzają.

Mały plik vs duży plik

Wybór metody odczytu według rozmiaru pliku

plik jest
znany z
tego, że jest
mały
(ustawienia,

krótka notatka)	→	`read_text()` / `read()` w całości — proste i
Plik może być bardzo duży (log, log dla roku)	→	przetwarzanie linia po linii — cykl `for line i`

readlines() na naprawdę dużym pliku

Plik o rozmiarze 1 gigabajta logów to zły powód `readlines()`: całe lista linie muszą być przechowywane w pamięci jednocześnie. Pętla linia po linii `for line in file` przetwarza plik linijka po linii, nie gromadząc ich wszystkich naraz.

---

trochę głębiej:

Czytanie danych binarnych w fragmentach

Dla bardzo dużych plików binarnych czasem nie wszystkie są czytane naraz, lecz w fragmentach (*chunks*) Stały rozmiar:

chunked_read.py

```
with open("bolshoy_fajl.bin", "rb") as f:
    while True:
        chunk = f.read(8192)   # 8192 bajtów
        if not chunk:
            break
        # przetworzyć chunk tutaj
```

To jest opcjonalny, bardziej dogłębny materiał – dla większości edukacyjnych i małych tematów Jest wystarczająco dużo zadań praktycznych czytania tekstu linia po linijce.

Mini-projekt: analizator pliku tekstowego

Liczenie łańcuch znaków, słów i znaków pliku w strumieniu — bez ładowania całego pliku do pamięci natychmiast jako pojedyncza linia:

analizator_fajla.py

```
from pathlib import Path

def analiz_fajla(path: Path) -> dict[str, int]:
    stroki = 0
    slova = 0
    simvoly = 0
    with path.open("r", encoding="utf-8") as f:
        for line in f:
            stroki += 1
            slova += len(line.split())
            simvoly += len(line)
    return {„struny”: stroki, „słowa”: slova, „symbole”:
simvoly}

print(analiz_fajla(Path("spisok.txt")))
```

★★ Zadanie samodzielne

bajty osobno

Dodaj do raportu czwartą liczbę — rozmiar pliku w bajtach przez .stat().st_size — i porównaj go z liczbą znaków.

Praktyka: duże pliki i analizator tekstu
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/15-23/index.html)

Jak wybrać format przechowywania danych

Zanim zapiszesz dane, warto zapytać: w jakiej formie powinny być przechowywane?

To samo pytanie, tylko w różnych formatach

Od rozdziału 11 słownik znaki:

igrok_slovar.py

```
igrok = {
    "name": "Anna",
    "score": 1200,
    "skills": ["Python", "Git"],
}
```

Jak zapisać taką strukturę do pliku? `str(igrok)` technicznie zadziała, ale to nie jest prawdziwy format wymiany danych — trudno go i niebezpiecznie analizować z powrotem. Potrzebny jest **format pamięci**to ma jasne zasady Pisz i czytaj.

Serializacja

Serializacja i deserializacja
Serializacja to przekształcenie struktury Python (słownik, lista, obiekt) w reprezentację, którą można przechowywać lub przesyłać (tekst lub bajty). **Deserializacja** — przywracanie struktury danych z powrotem z tego przedstawienia.

Serializacja zamienia obiekty Python w przechowywalny tekst.

Jak wybrać format

Jaki format przechowywania wybrać.

Prosty, czytelny tekst — notatki, log	→ **zwykły plik tekstowy (rozdziały 15.1-15.4)**
Tabela z łańcuch znaków i kolumnami	→ **CSV (Rozdział 15.27)**

Zagnieżdżona struktura – słowniki, listy, ustawienia, zapisywanie gry	→ **JSON (Rozdział 15.25)**
Obraz, dźwięk, dowolne dane nietekstowe	→ **format binarny (Rozdział 15.17)**

Brak formatu zawsze jest najlepszy
Wybór formatu — to decyzja dostosowana do konkretnego zadania, a nie kwestia mody. Płaski tekst nadaje się do notatek, ale źle sprawdza się w przypadku zagnieżdżonych struktur; JSON doskonale zachowuje strukturę, ale źle nadaje się do tabel z tysiącami identycznych wierszy, gdzie znacznie wygodniejszy jest CSV.

Praktyka: wybieramy format przechowywania
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/15-24/index.html)

JSON: zapisujemy struktury danych

Słowniki i listy z rozdziału 11 wreszcie mają niezawodny sposób na przetrwanie restartu programu.

Moduł json

json_dump.py

```python
import json

data = {"name": "Anna", "score": 1200}

with open("igrok.json", "w", encoding="utf-8") as f:
    json.dump(data, f, ensure_ascii=False, indent=2)
```

```
$ cat igrok.json
{
  "name": "Anna",
  "score": 1200
}
```

Rzeczywiście zapisany plik — zwykły tekst, można go otworzyć w dowolnym edytorze.

json_load.py

```python
import json

with open("igrok.json", "r", encoding="utf-8") as f:
    data = json.load(f)

print(type(data))   # <class 'dict'>
print(data["name"])   # Anna
```

dump/load vs dumps/loads

Funkcja	Co robi
`json.dump(data, file)`	zapisuje JSON bezpośrednio w otwartym obiekcie pliku

Funkcja	Co robi
`json.dumps(data)`	zwraca JSON jako zwykły łańcuch znaków str
`json.load(file)`	czyta JSON z otwartego obiektu plikowego
`json.loads(text)`	analizuje JSON z gotowego łańcuch znaków str

Dopasowanie JSON Python ↔ typów

JSON	Python
object	dict
array	list
string	str
number	int / float
true / false	True / False
null	None

Czego JSON nie potrafi zrobić bezpośrednio

JSON nie przechowuje dowolnego obiektu Python sobie
`set` , instancja własnej klasy, `bytes` , `complex` — żaden z nich nie ma bezpośredniej reprezentacji w JSON. `json.dump()` nie może „magicznie”

Krotka wraca jako lista podczas czytania
Jeśli zapisujesz `tuple` jako JSON- array, po ponownym uruchomieniu otrzymasz normalny `list` , nie `tuple` – JSON nie rozróżnia tych dwóch typów Python, ma tylko jeden rodzaj „tablicy”

DEBUG LAB 12: NIEPRAWIDŁOWY JSON W PLIKU

slomanny_json.py

```
with open("nastroyki.json", "w", encoding="utf-8") as f:
    f.write('{"theme": "dark",}')   # dodatkowy
przecinek przed }

import json
with open("nastroyki.json", "r", encoding="utf-8") as f:
    data = json.load(f)
```

```
Traceback (most recent call last):
json.decoder.JSONDecodeError: Illegal trailing comma before e
```

Co widać na ekranie

Obecność pliku nie oznacza, że jego zawartość to poprawny JSON: plik mógł zostać uszkodzony, edytowany ręcznie lub zapisany błędnie w innym miejscu programu. `json.JSONDecodeError` — sygnał dokładnie o tym, a nie o problemie ze ścieżką lub uprawnieniami.

POPRAWIONY KOD

json_bez_zapyatoj.py

```
with open("nastroyki.json", "w", encoding="utf-8") as f:
    f.write('{"theme": "dark"}')
```

Mini Projekt: Menedżer ustawień

Ustawienia aplikacji, które będą potrzebne w aplikacji graficznej rozdziału 16 — z zrozumiałymi wartościami domyślnymi, jeśli plik jeszcze nie istnieje:

nastroyki_menedzher.py

```
import json
from pathlib import Path

DEFAULT_SETTINGS = {"theme": "light", "language": "ru",
"window_width": 900}

def load_settings(path: Path) -> dict:
    if not path.exists():
        return dict(DEFAULT_SETTINGS)
    with path.open("r", encoding="utf-8") as f:
        return json.load(f)

def save_settings(path: Path, settings: dict) -> None:
    with path.open("w", encoding="utf-8") as f:
        json.dump(settings, f, ensure_ascii=False, indent=2)

settings_path = Path("nastroyki.json")
settings = load_settings(settings_path)
settings["theme"] = "dark"
save_settings(settings_path, settings)
```

Przygotowanie gruntu pod rozdział 16

W ten sposób aplikacja graficzna na Tkinter (Rozdział 16) będzie mogła zapamiętywać ustawienia użytkownika między uruchomieniami — ładować je przy starcie i zapisywać po zmianie, bez żadnej linii kodu interfejsu w tej funkcji.

Praktyka: JSON i menedżer ustawień

interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/15-25/index.html)

Mini-projekt: ratowanie Player

Obiekt z rozdziału 14 w końcu wie, jak przetrwać restart programu.

Połącz rozdział 14 (dataclasses, obiekty) i rozdział 15 (JSON, pliki): zapisz i Prześlij prawdziwą postać z gry.

player_dataclass.py

```
from dataclasses import dataclass, asdict

@dataclass
class Player:
    name: str
    score: int
    inventory: list[str]
```

Zachowanie

save_player.py

```
import json
from dataclasses import asdict
from pathlib import Path

player = Player(name="Anna", score=1200, inventory=[„miecz”,
„tarcza”])

with Path("save.json").open("w", encoding="utf-8") as f:
    json.dump(asdict(player), f, ensure_ascii=False, indent=2)
```

asdict() jest skrótem od dataclass do słownika

`dataclasses.asdict()` (rozdział 14) zamienia instancję dataclass w zwykły słownik — jeśli wszystkie wartości wewnątrz są już kompatybilne z JSON- (łańcuch znaków, liczby, listy, słowniki), wynik można przekazać bezpośrednio `json.dump()`.

Ładowanie

load_player.py

```
import json
from pathlib import Path

with Path("save.json").open("r", encoding="utf-8") as f:
    data = json.load(f)

loaded_player = Player(**data)
print(loaded_player)
```

loaded_player — TO NIE ten sam obiekt co player:, to nowa instancja, przywrócona z pliku.

To nie jest „ten sam” obiekt

Po ponownym uruchomieniu programu `loaded_player` jest **nowe** egzemplarz `Player` zbudowany z danych pliku, a nie z obiektu, który istniał w poprzednim uruchomieniu programu. Po prostu ma ten sam stan — i to jest cel trwałości: nie po to, by zapisać sam obiekt Python, ale by przechować wystarczająco dużo danych, by przywrócić równoważny stan.

Challenge

Dalej samodzielnie: JSON książka kontaktowa

Weź książkę kontaktów z rozdziału 12 (słownik imię → numer telefonu) i dodaj do niej zapisy/ładowanie przez JSON, zgodnie z przykładem tej sekcji — wtedy kontakty przetrwają restart programu.

Praktyka: Zapisz i wczytaj Player

interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/15-26/index.html)

CSV: tabele w formie tekstowej

Dane tabelowe zasługują na format, który rozumie, czym jest pole — a nie tylko separator.

Tabela w formie tekstowej

rekordy.csv

```
name,score,level
Anna,1200,5
Bob,900,4
```

CSV (comma-separated values) to prosty format tekstowy dla tabel: wiersze to rekordy, Wartości w rzędzie są rozdzielone przecinkami. Odpowiednie do eksportu do tabel, raportów, zbiorów danych.

Dlaczego NIE split(NIE,")

DEBUG LAB 13: CSV ROZŁOŻONE PRZEZ SPLIT(„,”

csv_split_lovushka.py

```
stroka = 'Anna, „Świetnie, kontynuuj!”
chasti = stroka.split(",")
print(chasti)
```

```
['Anna', '"Отлично', ' продолжай!"']
```

Co widać na ekranie

Samo pole w cudzysłowie zawierało przecinek — w pełni pozwalający na to format CSV. `split(",")` nic nie wie o cudzysłowach w CSV i ślepo tnie po każdej przecinku, dzieląc jedno pole na dwa. XPath CSV należy analizować modułem `csv`, który poprawnie radzi sobie z cudzysłowami i przecinkami wewnątrz pól.

POPRAWIONY KOD

csv_modul_pravilno.py

```
import csv
import io

stroka = 'Anna, „Świetnie, kontynuuj!”
reader = csv.reader(io.StringIO(stroka))
print(next(reader))   # ['Anna', 'Super, tak dalej!']
```

csv.reader i csv.DictReader

csv_reader.py

```
import csv

with open("rekordy.csv", "r", encoding="utf-8", newline="") as
f:
    reader = csv.reader(f)
    for row in reader:
        print(row)
```

Dlaczego newline=„”
Dokumentacja modułu `csv` zaleca otwarcie pliku za pomocą `newline=""`, że sam moduł w pełni zarządza podziałami wierszy w polach — bez tego na niektórych platformach możliwe jest przetwarzanie podwójnej dzielnicy.

csv_dictreader.py

```
import csv

with open("rekordy.csv", "r", encoding="utf-8", newline="") as
f:
    reader = csv.DictReader(f)
    for row in reader:
        print(row["name"], int(row["score"]))
```

Wartości CSV — zawsze łańcuch znaków, dopóki ich nie przekształcisz
`row["score"]` — to łańcuch znaków `"1200"`, nie liczba `1200`, nawet jeśli wizualnie wygląda jak liczba. CSV nie przechowuje informacji o typie — rzutowanie (`int(...)`, `float(...)`) musisz zrobić to sam.

CSV nagrań

csv_writer.py

```
import csv

rows = [
    {"name": "Anna", "score": 1200},
    {"name": "Bob", "score": 900},
]

with open("novye_rekordy.csv", "w", encoding="utf-8",
newline="") as f:
    writer = csv.DictWriter(f, fieldnames=["name", "score"])
    writer.writeheader()
    writer.writerows(rows)
```

CSV vs JSON — i CSV — nie jest Excel

	CSV	JSON
Lepiej nadaje się dla	jednorodne tabele (wiersze/kolumny)	struktury zagnieżdżone
Typy danych	wszystko to linie tekstu	Znaki znaków, liczby, booleany, zagnieżdżanie

CSV nie jest plikiem Excel

`.csv` — zwykły format tekstowy, a nie format-książka `.xlsx` za pomocą arkuszy, wzorów i formatowania — robią to inne, wyspecjalizowane biblioteki.

Praktyka: CSV — csv.reader, DictReader, writer

interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/15-27/index.html)

Bezpieczne zarządzanie plikami

Zapisywanie do pliku to działanie z konsekwencjami. Rozsądna ostrożność jest warta kilku dodatkowych linijek kodu.

Przed nagraniem – zadaj sobie pytanie

1. Czy chcę zastąpić treść, czy nadpisać?
2. Albo utworzyć tylko, jeśli plik jeszcze nie istnieje?
3. Czy potrzebuję zapasowego przed zmianą?
4. Czy to dokładnie ta ścieżka, którą mam na myśli?"

Mode to świadomy wybór, a nie zalążek z przykładu
`"r"`, `"w"`, `"a"`, `"x"` - Każdy z nich ma inne konsekwencje dla istniejących treści (Rozdział 15.15). Kopiowanie trybu z przykładu bez pytania „czego dokładnie teraz potrzebuję?”

Praktyka w tym kursie są wyłącznie w piaskownicach

Obowiązkowa zasada bezpieczeństwa w praktykach
Wszystkie praktyki związane z plikami w tym kursie czytają, tworzą, modyfikują i usuwają pliki **tylko w swoim własnym folderze roboczym**. Nigdy — pliki repozytorium kursu, dokumenty, które już zapisałeś, folder domowy użytkownika ani dowolne ścieżki bezwzględne. Dotyczy to również Twoich własnych programów: nie wykonuj operacji destrukcyjnych na ścieżce, która przyszła z zewnątrz, bez jej sprawdzenia.

Wyjście z twojego folderu jest path traversal

put_naruzhu.py

```
imya_fajla = input(„Nazwa pliku do zapisu: ”)
# jeśli użytkownik wpisze „.. /.. /important.txt”
path = Path("data") / imya_fajla
```

Naiwne łączenie ścieżki wiąże się z ryzykiem wyjścia poza folder
Jeśli nazwa pliku pochodzi od użytkownika i zawiera `..`, samo połączenie z bazowym folderem może przenieść cię poza niego, do zupełnie innego miejsca w systemie plików. Nie budujemy tu pełnego filtra bezpieczeństwa — ważne jest po prostu, by być świadomym samego ryzyka i nie ufać nazwie pliku użytkownika jako bezpiecznej.

Bardziej profesjonalne: Bezpieczne oszczędzanie

Pomysł na zmniejszenie ryzyka pozostawienia głównego pliku w uszkodzonym stanie pośrednim: Najpierw zapisz w **tymczasowy** plik w tym samym folderze, a następnie zastąp główny plik w całości:

bezopasnoe_sohranenie.py

```
import json
from pathlib import Path

def bezopasno_sohranit(path: Path, data: dict) -> None:
    vremenny = path.with_suffix(path.suffix + ".tmp")
    with vremenny.open("w", encoding="utf-8") as f:
        json.dump(data, f, ensure_ascii=False, indent=2)
    vremenny.replace(path)   # zamień główny plik tylko wtedy,
gdy wszystko jest już zapisane
```

Nie absolutna gwarancja — a zmniejszone ryzyko
Zmniejsza ryzyko odejścia `path` w częściowo zapisanym stanie przy przerwaniu zapisu, ale nie jest absolutną gwarancją bezpieczeństwa danych na jakimkolwiek sprzęcie i dowolnym systemie plików.

Kopia zapasowa przed wymianą

backup_pered_zamenoj.py

```
import shutil
from pathlib import Path

path = Path("save.json")
if path.exists():
    shutil.copy(path, path.with_suffix(".json.bak"))
```

Jedna kopia zapasowa, nie nieskończony łańcuch

Jedna aktualna kopia zapasowa przed wymianą zwykle wystarcza do zadań edukacyjnych i małych zastosowań — nie trzeba tworzyć nieskończenie rosnącego łańcucha plików `.bak.bak.bak`.

Praktyka: bezpieczne zapisywanie i kopia zapasowa

interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/15-28/index.html)

Mini-projekt: tabela rekordów

Ładowanie → modyfikacja zapisu → to cykl zadań niemal każdego trwałego programu.

Zebranie plików, list, słowników, funkcji (Rozdział 13) i sortowania (Rozdział 11) w jednym Projekt praktyczny — tabela dokumentacji.

Load → add → sort → save to typowy trwały pipeline danych.

tablitsa_rekordov.py

```
import csv
from pathlib import Path

def load_rekordy(path: Path) -> list[dict]:
    if not path.exists():
```

```
        return []
    with path.open("r", encoding="utf-8", newline="") as f:
        reader = csv.DictReader(f)
        return [{"name": row["name"], "score":
int(row["score"])} for row in reader]

def save_rekordy(path: Path, rekordy: list[dict]) -> None:
    with path.open("w", encoding="utf-8", newline="") as f:
        writer = csv.DictWriter(f, fieldnames=["name",
"score"])
        writer.writeheader()
        writer.writerows(rekordy)

def top_n(rekordy: list[dict], n: int) -> list[dict]:
    return sorted(rekordy, key=lambda r: r["score"],
reverse=True)[:n]

path = Path("rekordy.csv")
rekordy = load_rekordy(path)
rekordy.append({"name": "Carlos", "score": 1500})
save_rekordy(path, rekordy)

for zapis in top_n(rekordy, 3):
    print(zapis["name"], zapis["score"])
```

★★ Zadanie samodzielne

No Name Repeats

Zmień load_rekordy/dodawanie tak, aby przy ponownym zapisaniu wyniku już istniejącego gracza aktualizowany był jego rekord, a nie dodawany drugi wpis o tej samej nazwie.

Challenge

Na JSON zamiast CSV

Przerób ten sam projekt, aby rekordy.csv stał się rekordy.json – porównaj to, co zmieniło się w load/save i co pozostało takie samo (sortowanie, top_n).

Praktyka Tabela wyników

interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/15-29/index.html)

Przeglądarka i dysk lokalny: Dwa systemy plików

"Utworzony plik" i "Plik pojawił się na moim komputerze" nie zawsze to samo.

Dwa różne systemy plików

Praktyki tego kursu są realizowane w przeglądarce przez Pyodide – prawdziwy Python, ale działając na stronie internetowej, a nie jak zwykły program na komputerze.

Pliki w praktyce przeglądarki nie są plikami na twoim komputerze

Podczas praktyki tego kursu występuje `Path("hello.txt").write_text(...)`, zachowanie `open` / `pathlib` prawdziwe — zapis, odczyt i dopisanie działają zgodnie z rzeczywistymi zasadami. Ale sam plik istnieje tylko w **tymczasowy system plików Python w tej karcie przeglądarki**. Nie pojawia się w `~/Downloads` nie jest widoczny w Eksploratorze/Finder i znika po kliknięciu Reset środowiska lub zamknięciu zakładki.

Praktyka przeglądarki zapisuje się na wirtualnym systemie plików — a nie na prawdziwym dysku komputerowym.

DEBUG LAB 14: PLIK UTWORZONY W PYODIDE, ALE NIE NA DYSKU

putanitsa_fs.py

```
# wykonane WEWNĄTRZ przeglądarkowej praktyki kursu
from pathlib import Path

Path("moy_fajl.txt").write_text(„cześć”,
encoding="utf-8")
# czekanie: „teraz ten plik jest na moim komputerze”
```

```
# Файл действительно создан и читаем внутри этой вкладки брау
# но его нет ни в проводнике/Finder, ни в реальной домашней п
```

Co widać na ekranie

Wirtualny system plików jest Pyodide całkowicie realny dla samego programu w zakładce — ale jest odizolowany od samego dysku komputerowego. Mylenie "pliku utworzonego przez program" z "plik pojawił się na moim komputerze" jest źródłem zamieszania w środowiskach Python. przeglądarkowych

POPRAWIONY KOD

kak-poluchit-nastoyaschiy-fajl.py

```
# dla pliku na prawdziwym dysku — uruchom ten sam kod
LOKALNIE
#(VS Code/PyCharm/Jupyter na komputerze, sekcja 15.30)
from pathlib import Path

Path("moy_fajl.txt").write_text(„cześć”,
encoding="utf-8")
```

Gdy prawdziwa trwałość na dysku jest ważna

W tej sekcji praktyka wymaga lokalnego uruchomienia pliku rzeczywistego na prawdziwym pliku Dysku, który przetrwa nawet zamknięcie edytora.

Praktyka: Prawdziwa wytrwałość na komputerze
Lokalna praktyka — pobierz .ipynb i uruchom go w VS Code/PyCharm/Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/15-30/index.html)

1. Pobierz laptopa z tej sekcji i otwórz go lokalnie (VS Code, PyCharm lub Jupyter).
2. Wykonaj komórkę, która tworzy folder `data/` i nagrywanie w UTF-8-file.
3. Zatrzymaj wykonanie (lub zamknij edytor).
4. Otwórz plik w Eksploratorze/Finder lub w samym edytorze, żeby upewnić się, że jest na miejscu.
5. uruchom laptopa ponownie i przeczytaj ten sam plik – dane z pierwszego uruchomienia na Bo to prawdziwy plik na prawdziwym dysku.

Podsumowanie rozdziału: Zestaw narzędzi do plików

Od „gdzie idą zmienne”

Zestaw narzędzi do plików

Co wybrać dla konkretnych zadań

Potrzebna jest ścieżka	→ `pathlib.Path`
Potrzebuję aktualnego katalogu roboczego	→ `Path.cwd()`
Potrzebujesz folderu obok twojego .py-pliku	→ `Path(__file__).resolve().parent`
Mały UTF-8-plik tekstowy w całości	→ `path.read_text() / write_text()`
Przetwarzanie wiersz po wierszu lub tryb specjalny	→ `with path.open( tryb, encoding=„utf-8”`
Dopisanie bez utraty starej zawartości	→ `open(path, „a”`

Tworzyć, ale nie zastępować istniejącego pliku	→ `open(path, „x”`
Dane binarne w całości	→ `path.read_bytes() / write_bytes()`
Lista zawartości folderów	→ `path.iterdir()`
Wyszukaj pliki według szablonu	→ `path.glob("*.txt")`
Zagnieżdżona struktura danych	→ `moduł json`
Tabela z łańcuch znaków i kolumnami	→ `moduł csv`
Wielu użytkowników / transakcji	→ `prawdopodobnie baza danych to nie jest`

Rozdział 15 całych

WYTRWAŁOŚĆ

```
pamięć procesu – tymczasowa
trwałe przechowywanie przetrwa proces
```

ładowanie ≠ ten sam obiekt

SYSTEM PLIKÓW

plik vs folderze
bezwzględna vs względna ścieżka
CWD ≠ folder skryptów

PATHLIB.PATH

ścieżka jako obiekt
name / stem / suffix / parent
read_text/write_text/read_bytes/write_bytes

PLIK I WITH

open() → obiekt pliku
with gwarantuje zamknięcie
kursor przesuwa się do przodu tell()/seek()

TEKST I BAJTY

str jest zakodowany w bytes
UTF-8 oczywiście jest wyraźnym wyborem
znaki ≠ bajty

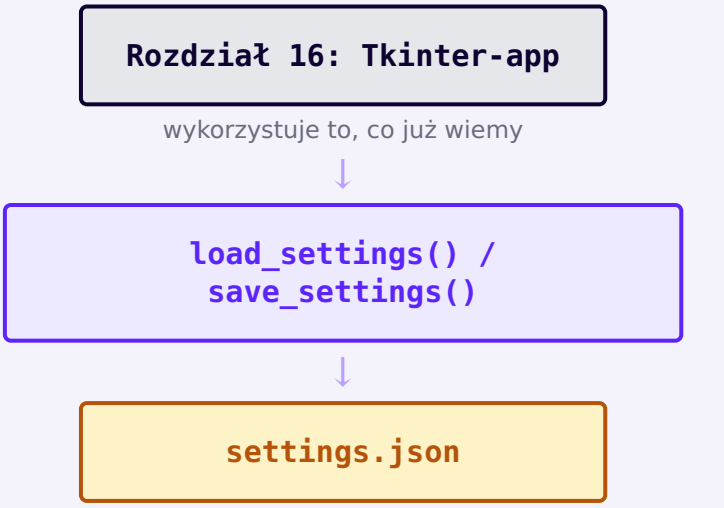
DANE STRUKTURALNE

```
JSON — struktury zagnieżdżone
CSV są tabelami, a nie split(„,”
bezpieczne oszczędzanie jest temp → replace
```

- Praca z plikami
  - Wytrwałość
    - pamięć jest tymczasowa
    - magazyn przeżywa zakończenie procesu
  - Ścieżki
    - absolutne i względne
    - CWD
    - pathlib.Path
  - Plik
    - open/with
    - kursor, tell/seek
    - r/w/a/x tryby
  - Tekst i bajty
    - str/bytes
    - UTF-8
  - Dane strukturalne
    - JSON
    - CSV

Mapa rozdziału 15 w całości.

co dalej



Trwałość opanowana w tym rozdziale staje się pamięcią przyszłej aplikacji graficznej.

W rozdziale 16 („Budowanie fajnych aplikacji z Tkinter” `load_settings()` / `save_settings()` z sekcji 15.25, bez żadnej nowej linijki o plikach.

Czego nauczyliśmy się w tym rozdziale

- Obiekty Python żyją w pamięci procesu i nie przeżywają jego zakończenia — dane w trwałym magazynie plików mogą przeżyć zakończenie procesu.
- Ścieżka — nie zawartość pliku: `Path(...)` tylko odnosi się do miejsca w systemie plików.
- Ścieżka względna jest rozwiązywana względem aktualnego katalogu roboczego (CWD) — a nie folderu skryptów.
- `pathlib.Path` jest ścieżką jako obiektem: name/stem/suffix/parent i wygodne metody zamiast ręcznego składania łańcuch znaków.
- `open()` zwraca obiekt pliku; `with` zapewnia, że zamknięcie się nawet w przypadku błędu.
- Plik posiada kursor — pozycję odczytu/zapisu, która przesuwa się do przodu i nie wraca sama; do tego jest `seek()`.
- Tryby `r/w/a/x` - Inne ryzyko niż istniejąca zawartość. `"w"` usuwa plik po otwarciu.
- Tekst jest kodowany w bajtach – zawsze wyraźnie `encoding="utf-8"`, nie polegaj na domyślnym kodowaniu.
- JSON przechowuje zagnieżdżone Python strukturach; CSV są tabelami; ani split(„,”
- Pliki ćwiczeń w przeglądarce (Pyodide) nie są plikami na twoim prawdziwym komputerze.

ROZDZIAŁ 16 · STR. 951

Twórz fajne aplikacje za pomocą Tkinter

Dowiedz się, jak budować rzeczywiste aplikacje okienkowe: analizuj model zdarzeń, drzewo widżetów, responsywny układ, formularze, menu, dialogi, timery oraz architekturę aplikacji oraz powiąż interfejs z funkcjami, obiektami i plikami z poprzednich rozdziałów.

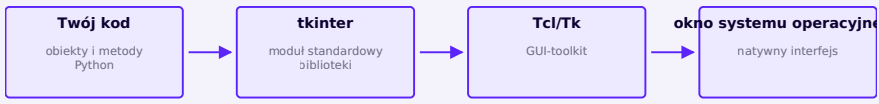
16.33 Podsumowanie rozdziału: Zestaw narzędzi Tkinter

Tkinter — wszystko poprawnie skonfigurowane!

Pierwsza prawdziwa aplikacja okienkowa — i przejście od sekwencyjnego wykonania do modelu opartego na zdarzeniach.

Dotychczas wszystkie programy albo wyświetlały tekst w terminalu (rozdziały 1–5, 8–15), albo rysowały na płótnie Turtle (rozdziały 6–7, 12). Program terminalowy wykonuje się **łańcuch znaków dla struna**: robi krok, czeka na wejścia przez `input()`, czy Kolejnym krokiem jest ciągłe sterowanie programem. Aplikacja okienkowa jest projektowana inaczej: buduje się interfejs raz, a potem **czeka, aż użytkownik podejmie działanie** i reaguje na nie w dowolnej kolejności — użytkownik może kliknąć dowolny przycisk, wpisać tekst w dowolne pole, zmienić rozmiar okna. To nazywa się **Programowanie zdarzane** (event-driven programming), a ten model omówimy szczegółowo w sekcjach 16.9–16.10.

Moduł dla aplikacji okienkowych nazywa się `tkinter` jest interfejs Python do biblioteki graficznej Tcl/Tk i lubi `turtle`, wchodzi w standardową dostawę Python:



tkinter jest pomostem między Python a środowiskiem systemowym Windows, a nie GUI od nowa wymyślonym.

Check: Czy Tkinter dostępny?

Nie każda instalacja Python musi mieć środowisko pracy Tk. Szybka diagnoza:

proverka_tkinter.py

```
python -m tkinter
```

Jeśli Tkinter i Tk są dostępne — pojawi się małe okno testowe. Jeśli pojawi się błąd importu lub okno się nie otwiera — z lokalną instalacją Python/Tk trzeba poradzić sobie osobno (w sprawdzonych instrukcjach dla twojego systemu operacyjnego), zanim przejdziesz dalej w tym rozdziale.

Pierwsze okno

pervoe_okno.py

```
import tkinter as tk

root = tk.Tk()
root.title(„Moja pierwsza aplikacja”)

root.mainloop()   # rozpoczyna pętlę zdarzeń
```

`tk.Tk()` tworzy główne okno aplikacji — coś w rodzaju `turtle.Screen()` stworzył płótno do malarstwa. To jest bezpośrednie powiązanie z Rozdział 14: Rezultat `tk.Tk()` jest regularnym obiektem Python- z własnym metodami.

root jest przykładem Tk z własnymi metodami; jest to OOP rozdziału 14 w prawdziwej bibliotece.

root — powszechne konwencje nazewnictwa
Główne okno jest zwykle nazywane `root` („rdzeń”

Zazwyczaj jeden root na aplikację
W zwykłym zastosowaniu, **jeden** cel `Tk()` na cały proces. Dla dodatkowych okien (ustawienia, dialogi) używa się `tk.Toplevel(...)` (sekcja 16.20), a nie drugi `Tk()`.

mainloop() nie jest „zamrożeniem”

`root.mainloop()` uruchamia cykl, który aktywnie obsługuje zdarzenia: kliknięcia, wprowadzanie tekstu, zmianę rozmiaru okna. Program nie „zawiesza się” i nie „zasypia” — czeka i reaguje, pozostając responsywny, aż zostanie zamknięty. Omówimy to szczegółowo w rozdziale 16.10.

> **Praktyka: Stwórz pierwsze okno**
> Moduł tkinter otwiera natywne okno Python — wykonaj lokalnie w VS Code, PyCharm lub Jupyter
>
> Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/16-01/index.html)

Etykiety, guziki i ich rozmieszczenie

Pierwsze interaktywne widgety – i jak Tkinter umieszcza je w oknie.

Widżet — wizualny obiekt interfejsu

Widżet (widget) — dowolny element interfejsu widzialnego/interaktywnego: etykieta, przycisk, pole wejściowe. Podobnie jak w rozdziale 14, każdy widżet jest instancją klasy z własnymi atrybutami i metodami. **Wytwórnia** (`Label`) pokazuje tekst, **przycisk** (`Button`) reaguje na kliknięcie.

metki_knopki.py

```
import tkinter as tk

root = tk.Tk()

label = tk.Label(root, text=„Cześć, Tkinter!”)
label.pack()

def na_knopku_nazhali():
    print(„Przycisk został naciśnięty!”)

button = tk.Button(root, text=„Click me”,
command=na_knopku_nazhali)
button.pack()

root.mainloop()
```

Wynik wykonania powyższego kodu — prawdziwe okno Tkinter. Przykład wyglądu w środowisku kursu. Na Twoim OS czcionki, rozmiary i motyw mogą się nieco różnić.

Tworzenie widżetu nie jest tym samym, co jego wyświetlanie

`tk.Button(root, ...)` tworzy tylko obiekt przycisku. Nie pojawi się on w oknie, dopóki go nie umieścisz **Menedżer Geometrii** — `.pack()` tutaj, albo `.grid()` (Rozdział 16.7), lub `.place()` (Rozdział 16.15).

command jest funkcją, a nie wywołaniem funkcji

Parametr `command` powiązanie przycisku z funkcją jest ważne aby przekazać samą funkcję, **bez nawiasów**: `command=na_knopku_nazhali`, a nie `command=na_knopku_nazhali()`.

Częsty błąd: nawiasy w command

`na_knopku_nazhali` — to sama funkcja jako obiekt (rozdział 13). `na_knopku_nazhali()` jest *wywołanie* funkcje obecnie. `command=na_knopku_nazhali()` zadzwoni **natychmiast**, raz, podczas tworzenia przycisku — i połączy przycisk z tym, co funkcja *powrócił* (zazwyczaj `None`), a nie z samą funkcją. Kliknięcia przycisku po tym niczego nie wywołają.

Przycisk normalny obok niedostępnego (przycisk disabled)

state=„disabled”

state="disabled"
Możesz tymczasowo uniemożliwić dostęp przycisku — `button.state(["disabled"])` z ttk-widgets lub `state="disabled"` przy utworzeniu. Przydatne, gdy dane nie są jeszcze gotowe do działania (Rozdział 16.23 - walidacja).

Szczegóły pack

`.pack()` najprostszy sposób na umieszczenie widżetu w oknie. Domyślnie widżety są ułożone pod sobą od góry do dołu, ale istnieją przydatne parametry:

pack_podrobno.py

```
label.pack(side="left", padx=10, pady=10)
# side: top (domyślnie), bottom, left, right
# padx/pady: zewnętrzny margines poziomy/pionowy
```

Więcej o `fill` / `expand` i grupowanie przez `Frame` — w sekcjach 16.13–16.14.

Praktyka: Label, Button i pack()
Moduł tkinter otwiera natywne okno Python — wykonaj lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/16-02/index.html)

Wiele pól wejściowych

Entry dla jednej linii, Text — dla kilku: dwa sposoby uzyskania tekstu od użytkownika.

Wiele pól wejściowych

pole wejściowe (`Entry`) to tekst jednolinijkowy Pole, w którym użytkownik może coś wpisać:

polya_vvoda.py

```
import tkinter as tk

root = tk.Tk()

label = tk.Label(root, text=„Wprowadź imię:”)
label.pack()

entry = tk.Entry(root)
entry.pack()

def pokazat_imya():
    imya = entry.get()   # wyjmij tekst z pola
    print(f"Cześć, {imya}!")

button = tk.Button(root, text=„Wyślij”, command=pokazat_imya)
button.pack()

root.mainloop()
```

`entry.get()` zwraca aktualny tekst pola — zwykły łańcuch znaków, z którym można pracować jak z każdym innym (rozdział 8). `entry.insert(0, "текст")` wstawia tekst programowo, `entry.delete(0, "end")` oczyszcza pole.

One łańcuch znaków SMS. Linijka po linijce

`Entry` nadaje się do krótkiego tekstu o jednej linijce – nazwy, numer, hasło. Do tekstu wielolinijowego użyj innego widżetu, `Text`:

Entry z tekstem „Cartesian” obok Text z trzech linii

Entry — zawsze jedna łańcuch znaków; Text — tyle linijek, ile chcesz. Przykład wyglądu w środowisku kursu. Na twoim systemie operacyjnym czcionki, rozmiary i motywy mogą się nieco różnić.

mnogostrochnyj_tekst.py

```
text_box = tk.Text(root, height=5, width=30)
text_box.pack()

def pokazat_tekst():
    content = text_box.get("1.0", "end-1c")   # od początku do
końca, bez końca \n
    print(repr(content))
```

Indeks „1.0”

U `Text` własny system indeksów: `"1.0"` oznacza „łańcuch znaków 1, symbol 0” `float` z rozdziału 4 — po prostu tekstowa etykieta pozycji, a zwykła indeksacja wierszy Python (`text[0]`) `Text` nie dotyczy.

„end” kontra „end-1c”

`Text` zawsze dodaje jeden ostatni link na końcu swojej zawartości. `.get("1.0", "end")` zawiera ten dodatek `\n`, że użytkownik nie napisał. Jeśli potrzebujesz tekstu widocznego dla użytkownika bez tego technicznego dodatku, użyj `.get("1.0", "end-1c")` („koniec minus jeden znak”

Praktyka: Entry i Text

Moduł tkinter otwiera natywne okno Python — wykonaj lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/16-03/index.html)

Zmienne Tkinter

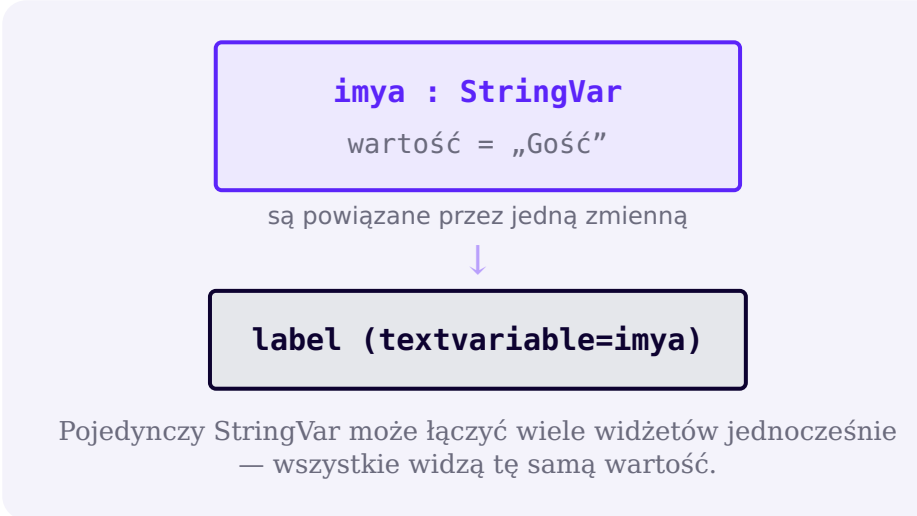
Specjalne zmienne, które same aktualizują powiązane z nimi widgety na ekranie.

Zmienne normalne Python nie „wiedzą” `StringVar`, `IntVar`, `BooleanVar`, `DoubleVar`.

peremennye_tkinter.py

```
import tkinter as tk

root = tk.Tk()

imya = tk.StringVar(value=„Gość”)

label = tk.Label(root, textvariable=imya)
# etykieta sama się zaktualizuje przy zmianie imya
label.pack()

def smenit_imya():
    imya.set("Cartesian")

button = tk.Button(root, text=„Zmień nazwę”,
command=smenit_imya)
button.pack()

root.mainloop()
```

Wartość jest uzyskiwana przez `.get()` i zmieniane przez metodę `.set()` — a widżety powiązane przez `textvariable` (lub `variable` u Radiobutton/Checkbutton), aktualizuje się automatycznie na ekranie.



Pojedynczy StringVar może łączyć wiele widżetów jednocześnie — wszystkie widzą tę samą wartość.

Zmienne Tk nie zastępują zwykłych zmiennych Pythona
`StringVar` i podobne są specjalnym narzędziem do komunikacji z widgetami, a nie zastępowaniem wszystkich zmiennych programowych universal-. Regularne `int`, `str`, `list` nadal świetnie sprawdzają się w przechowywaniu danych aplikacji, które nie pojawiają się bezpośrednio w widżecie.

Trochę głębiej:

trace_add

Można reagować na każdą zmianę Tk-zmiennej:

trace_add.py

```
def pri_izmenenii(*args):
    print(„Nowa wartość:”, imya.get())

imya.trace_add("write", pri_izmenenii)
```

Jest to wygodne dla interfejsów „reaktywnych”

Praktyka: StringVar i powiązane widgety
Moduł tkinter otwiera natywne okno Python — wykonaj lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/16-04/index.html)

Mnóstwo opcji!

Przyciski radiowe do wyboru jednej opcji oraz pola wyboru niezależnych rozwiązań tak/nie.

Do wyboru spośród kilku opcji Tkinter ma przełączniki (`Radiobutton` — wybierz jedną z kilku) oraz pola wyboru (`Checkbutton` – włączaj/wyłączaj każdy osobno).

Trzy przełączniki: Herbata, Kawa (wybrana), Woda

Jedną z wybranych opcji z grupy jest cieniowane koło. Przykład wyglądu w środowisku kursu. Czcionki, rozmiary i motyw mogą się nieco różnić w zależności od systemu operacyjnego.

radiobutton.py

```
import tkinter as tk

root = tk.Tk()
vybor = tk.StringVar(value="chay")

tk.Radiobutton(root, text=„Herbata”, variable=vybor,
value="chay").pack()
tk.Radiobutton(root, text=„Kawa”, variable=vybor,
value="kofe").pack()

def pokazat_vybor():
    print(f"Wybrałeś: {vybor.get()}")

tk.Button(root, text=„Gotowe”, command=pokazat_vybor).pack()
root.mainloop()
```

Grupa to wspólny zmienna

Wszystkie przełączniki jednej grupy muszą być używane **to samo** zmienna (`variable=vybor` oba) oraz **różne** wartości (`value="chay"` / `value="kofe"`). Tak Tkinter rozumie, że jest to jedna grupa wzajemnie wykluczających się wariantów.

DEBUG LAB 1: KAŻDY RADIOBUTTON — WŁASNY ZMIENNA

slomannaya_gruppa.py

```
chay_var = tk.StringVar()
kofe_var = tk.StringVar()

tk.Radiobutton(root, text=„Herbata”, variable=chay_var,
value="chay").pack()
tk.Radiobutton(root, text=„Kawa”, variable=kofe_var,
value="kofe").pack()
```

```
# Оба переключателя можно выбрать ОДНОВРЕМЕННО —
# это уже не взаимоисключающий выбор, а два независимых флажк
```

Co widać na ekranie

Każdy przycisk ma własne, odrębne zmienna, więc Tkinter nie widzi między nimi związku: wybór jednego nie wpływa na drugi. Zachowanie grupowe przełączników opiera się na wspólnej zmiennej, a nie na wizualnym podobieństwie przycisków.

POPRAWIONY KOD

gruppa_fixed.py

```
vybor = tk.StringVar(value="chay")

tk.Radiobutton(root, text=„Herbata”, variable=vybor,
value="chay").pack()
tk.Radiobutton(root, text=„Kawa”, variable=vybor,
value="kofe").pack()
```

Checkbutton jest niezależnym wyborem

Dwa znaczniki: zdjęty i ustawiony

Dwa stany tego samego widżetu – niedokończone/zainstalowane. Przykład wyglądu w środowisku kursu. Czcionki, rozmiary i motyw mogą się nieznacznie różnić w zależności od twojego systemu operacyjnego.

checkbutton.py

```
saharok = tk.BooleanVar(value=False)
tk.Checkbutton(root, text=„Z cukrem”, variable=saharok).pack()

def pokazat_flazhok():
    print(f"Z cukrem: {saharok.get()}")
```

Checkbutton nie jest Radiobutton

`Checkbutton` jest niezależnym rozwiązaniem tak/nie, które nie jest powiązane z innymi polami do zaznaczenia. Nie myl go z grupą `Radiobutton` gdy wybór jednej opcji wyklucza pozostałe.

Praktyka: Radiobutton i Checkbutton

Moduł tkinter otwiera natywne okno Python — wykonaj lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/16-05/index.html)

Menu

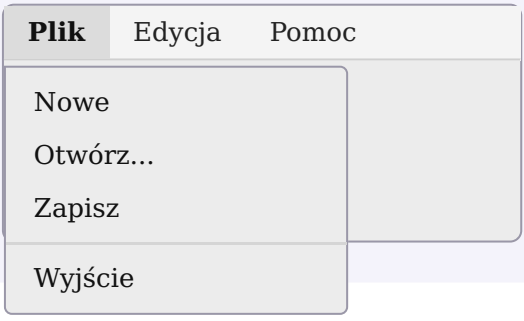
To prawdziwa aplikacja prawie zawsze zaczyna się od menu u góry okna.

Prawdziwa aplikacja rzadko jest kompletna bez menu na górze okna. W Tkinter menu jest zbudowany z obiektu `Menu`:

menu.py

```
import tkinter as tk

root = tk.Tk()

def novyj_fajl():
    print(„Utwórz nowy plik...”)

menu_bar = tk.Menu(root)
fajl_menu = tk.Menu(menu_bar, tearoff=0)
fajl_menu.add_command(label=„Nowy”, command=novyj_fajl)
fajl_menu.add_separator()
fajl_menu.add_command(label=„Wyjście”, command=root.destroy)
menu_bar.add_cascade(label=„Plik”, menu=fajl_menu)

root.config(menu=menu_bar)
root.mainloop()
```

SCHEMATYCZNY OBRAZ (NIE PRAWDZIWY ZRZUT EKRANU)



Wygląd menu różni się w zależności od systemu operacyjnego
Rzeczywisty wygląd paska menu i otwartego menu zależy od systemu operacyjnego, Tk wersji i motywu – trasa nie jest w stanie konsekwentnie odtworzyć zrzutu ekranu tego elementu headless-we wszystkich środowiskach, więc używa schematu z precyzyjną strukturą (łańcuch znaków menu → element → otwórz menu → przedmioty → separator), zamiast fotograficznego zrzutu ekranu.

tearoff=0
Domyślnie Tkinter dodaje na początku każdego menu linię przerywaną, pozwalającą „odłączyć” menu do osobnego okna. `tearoff=0` wyłącza właśnie tę możliwość odrywania menu — jeśli w twoim interfejsie nie jest ona potrzebna (a zwykle nie jest), jest to częsty i świadomy wybór, a nie uniwersalna zasada „zawsze tak rób”.

root.quit() kontra root.destroy()

Metoda	Co robi
`root.quit()`	zatrzymuje obecne mainloop() — ale okno i widgety pozostają
`root.destroy()`	niszczy okno i całe drzewo widżetów, a także zatrzymuje mainloop()

Nie są to synonimy wymienne
`root.quit()` pyta o prąd `mainloop()` zakończyć jest sam obiekt `root` i widgety nie są niszczone. `root.destroy()` idzie dalej: niszczy drzewo widgetów i samo okno, co również zatrzymuje `mainloop()` . Żaden z nich nie kończy samego procesu Python — po powrocie z `mainloop()` kod teoretycznie może być kontynuowany (np. coś zapisać przed wyjściem); proces kończy się później, naturalnie, jeśli po tym po prostu nie pozostanie kodu. Dla zwykłego punktu menu „Wyjście” w tym kursie jest to bardziej zrozumiałe i niezawodne `root.destroy` .

Akceleratory — to napis, nie przypisanie klawiszy

accelerator.py

```
fajl_menu.add_command(label=„Ratuj”, accelerator="Ctrl+S",
command=sohranit)
```

accelerator sam w sobie nie tworzy skrótu klawiszowego
`accelerator="Ctrl+S"` wyświetla tylko tekst tooltipu obok elementu menu. On **nie** automatycznie przypisuje skrót klawiaturowy do polecenia — wymaga to osobnego mechanizmu wiązania zdarzeń (`.bind(...)`), który zostanie szczegółowo omówiony w rozdziale 17.

Praktyka: Zbuduj menu aplikacji
Moduł tkinter otwiera natywne okno Python — wykonaj lokalnie w VS Code, PyCharm lub Jupyter
Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/16-06/index.html)

Wiersze i kolumny są grid

Dla formularzy wielokolumnowych grid() jest wygodniejsze niż pack().

`.pack()` jest świetna na proste okazje, ale tak jest Istnieje pewna granica: złożone kształty z wieloma kolumnami są trudne do złożenia. Dla tego istnieje `.grid()` — rozmieszczenie według łańcuch znakówm i kolumn, jak w tabeli:

grid.py

```
import tkinter as tk

root = tk.Tk()

tk.Label(root, text=„Imię:”).grid(row=0, column=0)
tk.Entry(root).grid(row=0, column=1)

tk.Label(root, text="Email:").grid(row=1, column=0)
tk.Entry(root).grid(row=1, column=1)

tk.Button(root, text=„Wyślij”).grid(row=2, column=0,
columnspan=2)

root.mainloop()
```

Pozycja	0	1
row=0	Label «Imię:»	Entry
row=1	Label «Email:»	Entry

Pozycja	0	1
row=2	Button (columnspan=2 — zajmuje obie kolumny)	

Nie mieszaj pack() i grid() w jednym kontenerze
Widżety w tym samym oknie nadrzędnym (lub ramce) muszą korzystać wyłącznie z jednego z tych elementów `pack()` lub tylko `grid()` — mieszanie powoduje błąd. Różne kontenery (np. zagnieżdżone `Frame`, rozdział 16.13) mogą używać różnych sposobów rozmieszczania niezależnie od siebie — szczegółowy schemat „dobrze/źle” będzie w rozdziale 16.15.

Jak sprawić, by taki formularz był responsywny na zmiany rozmiaru okien, znajduje się w sekcji 16.15 „Adaptacyjne grid”

Praktyka: Układ przez grid()
Moduł tkinter otwiera natywne okno Python — wykonaj lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/16-07/index.html)

Mini-projekt — kalkulator napiwków

Pierwsza pełnoprawna aplikacja książki — z polami wejściowymi, przyciskiem i wynikiem na ekranie.

Zebrajmy wszystko, czego nauczyliśmy się w rozdziale, w jednym prawdziwym aneksie: kalkulatorze napiwków z polem przycisku input oraz etykiety wyników. To są **fundacja** - W sekcji 16.31 my Wróć do niej i przekaż ją w pełnoprawne „Tip Calculator Pro”

kalkulyator_chaevyh.py

```
import tkinter as tk

root = tk.Tk()
root.title(„Kalkulator napiwków”)

tk.Label(root, text=„Suma konta:”).grid(row=0, column=0,
padx=10, pady=10)
schet_entry = tk.Entry(root)
```

```
schet_entry.grid(row=0, column=1)

tk.Label(root, text=„Procent napiwków:”).grid(row=1, column=0,
padx=10, pady=10)
procent_entry = tk.Entry(root)
procent_entry.grid(row=1, column=1)

rezultat_label = tk.Label(root, text="", font=("Arial", 14,
"bold"))
rezultat_label.grid(row=3, column=0, columnspan=2, pady=10)

def poschitat():
    schet = float(schet_entry.get())
    procent = float(procent_entry.get())
    chaevye = schet * procent / 100
    rezultat_label.config(text=f"Wskazówki: {chaevye:.2f}")

tk.Button(root, text=„Obliczaj”,
command=poschitat).grid(row=2, column=0, columnspan=2)

root.mainloop()
```

Jakie tematy tu się pojawiły

`grid()` — rozdział 16.7; `float()` - Rozdział 4 Formatowanie `:.2f` - Rozdział 8; sama formuła napiwków - Rozdział 12, Projekt 12-2.

Challenge

Sprawdzanie wejścia

Owinąć obliczenia w try/except (przeskakując trochę do przodu, więcej o tym w Rozdziale 21), aby aplikacja nie zawieszała się, jeśli użytkownik wpisze tekst zamiast liczby. W Sekcji 16.23 przyjrzymy się temu systematycznie.

Praktyka: Kalkulator napiwków

Moduł tkinter otwiera natywne okno Python — wykonaj lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/16-08/index.html)

Od terminala do modelu zdarzenia GUI:

Głównym intelektualnym przejściem tego rozdziału jest to, że zarządzanie nie płynie już w jednej ciągłej linii.

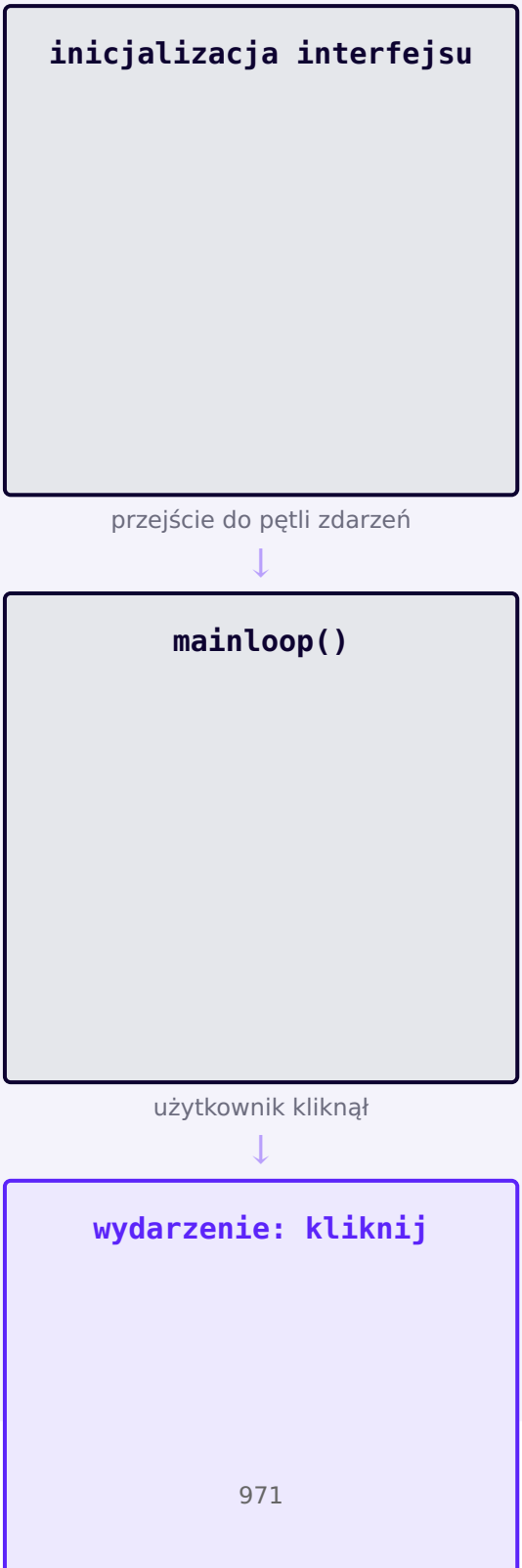
Terminal: Wykonanie sekwencyjne

Program terminala działa przewidywalnie, łańcuch znaków za linią sterowanie zawsze znajduje się na Sam program:

Terminal: program decyduje, kiedy i co zrobić dalej.

GUI: Realizacja oparta na zdarzeniach

Aplikacja okienna buduje interfejs raz, a następnie **czeka na wydarzenie** — i odpowiada na każdą osobną, wcześniej zarejestrowaną częścią kodu:


inicjalizacja interfejsu
przejście do pętli zdarzeń
mainloop()
użytkownik kliknął
wydarzenie: kliknij

Control nie pozostaje na jednej ciągłej sekwencji kodu — przekazuje się między handlerami.

Źródła zdarzeń

Skąd pochodzą zdarzenia

UŻYTKOWNIK

kliknięcie myszką
wprowadzanie tekstu
działanie okienne

SYSTEM

przemalowanie
zmiana rozmiaru okna

PROGRAM

```
Timer/after() — Rozdział 16.22
```

Wydarzenie, callback, command — nie są synonimami

Termin	Co to jest
Zdarzenie (event)	co się stało – klik, enter, timer
Callback	funkcja/wywołanie zarejestrowane wcześniej, aby zostać wywołane później
command	jeden ze sposobów, by Tkinter powiązać callback z konkretnym widżetem

Rejestracja callback jeszcze nie spełnia jego obowiązków:

registraciya_vs_vypolnenie.py

```
log = []

def on_click():
    log.append("clicked")

registered = on_click   # rejestracja: callback zapamiętana,
ale NIE wezwana
print(log)              # []

registered()            # Dyspozytornia: callback właśnie
działa
print(log)              # ['clicked']
```

To bezpośrednie powiązanie z rozdziałem 13

Funkcja jako obiekt, który można przekazać i wywołać później — dokładnie to samo „funkcja bez nawiasów”, o której była mowa w sekcji 16.2 o `command`. Model wydarzeń Tkinter całkowicie oparty na tym pomyśle.

Praktyka: rejestracja i wysyłka callback

Sprawdzane bez tkinter — czysta logika na funkcjach-obiektach

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/16-09/index.html)

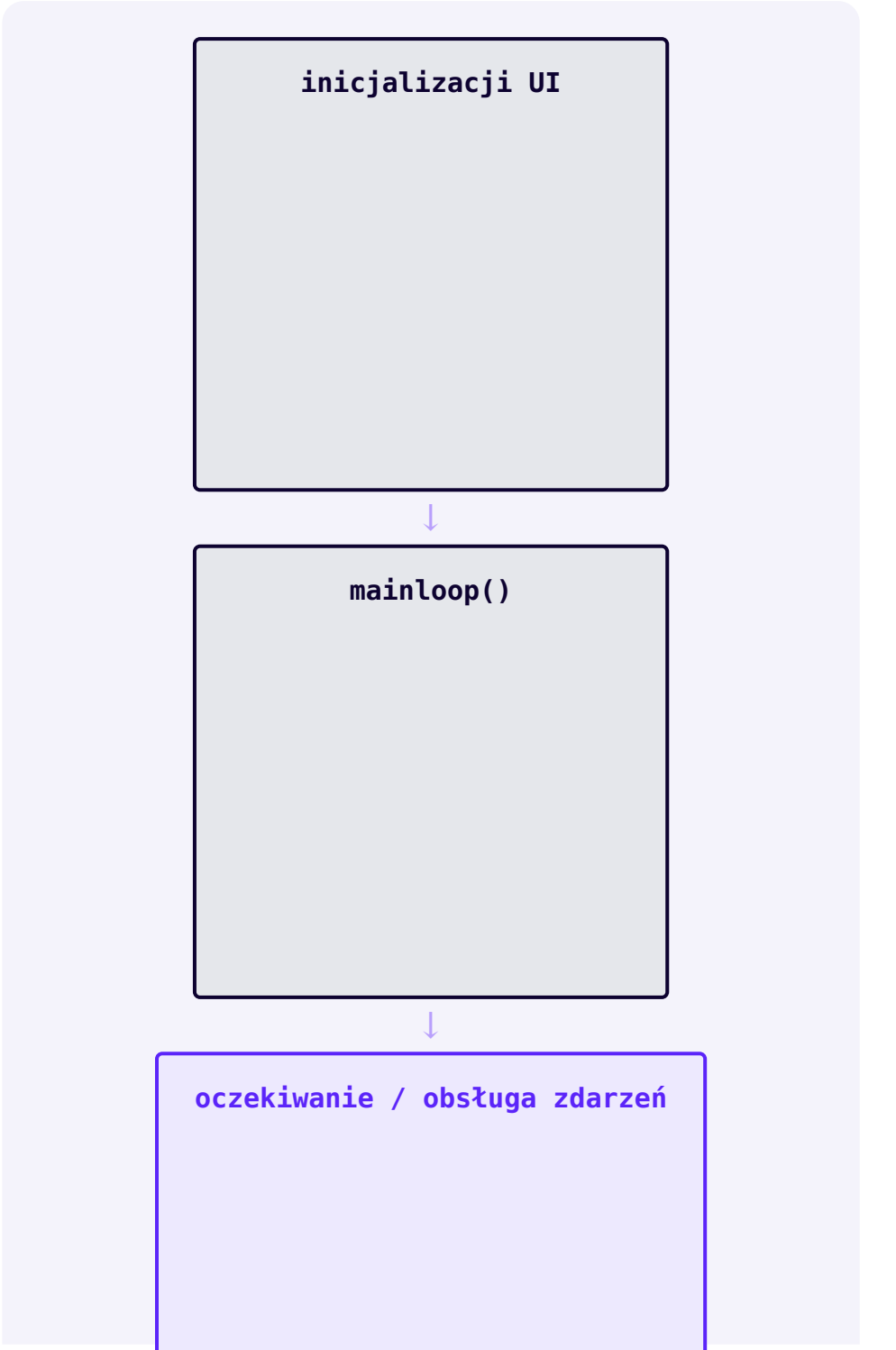
Jak działa pętla zdarzeń i mainloop

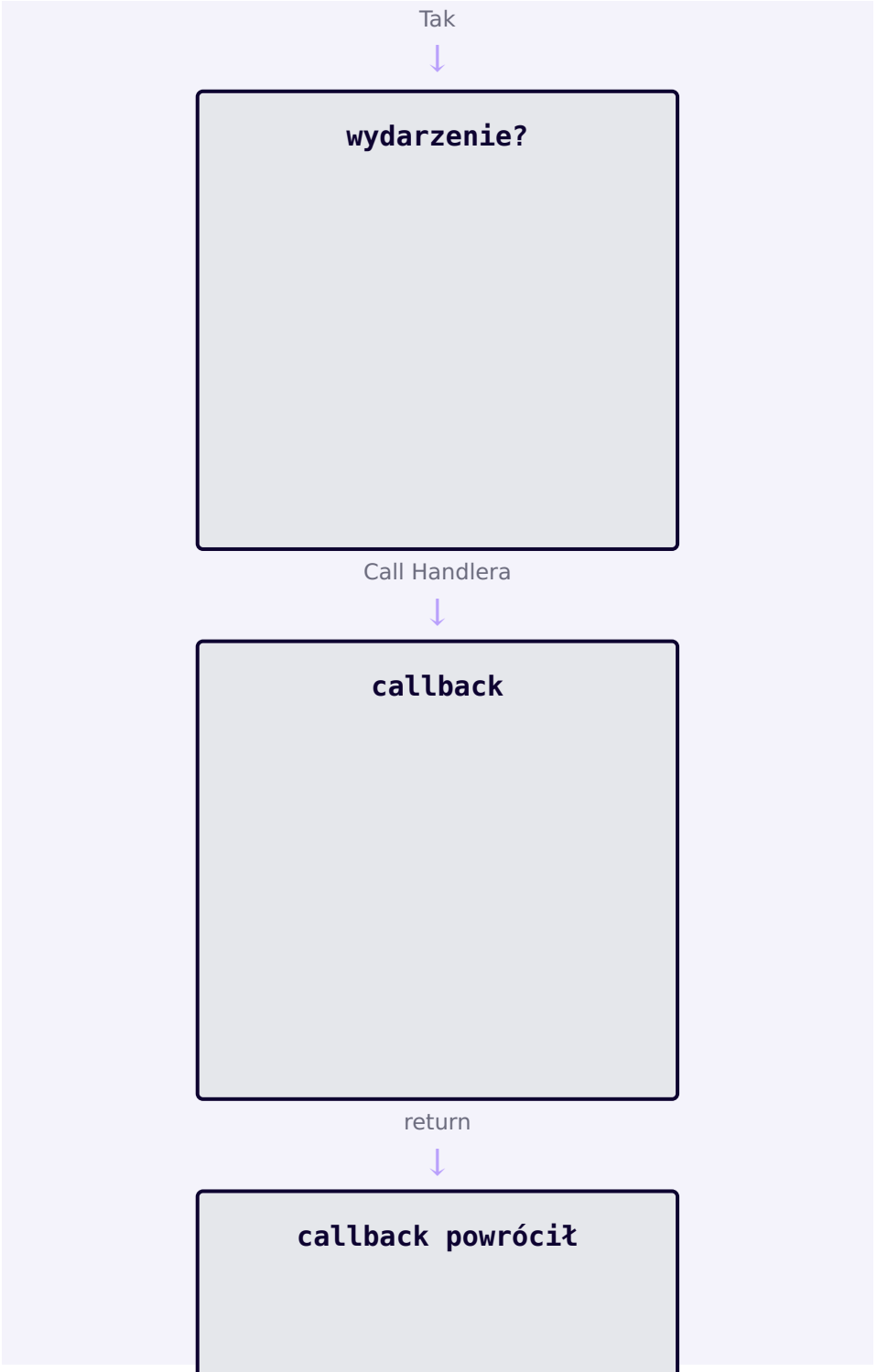
mainloop() nie jest czarną skrzynką ani „zamrożeniem”

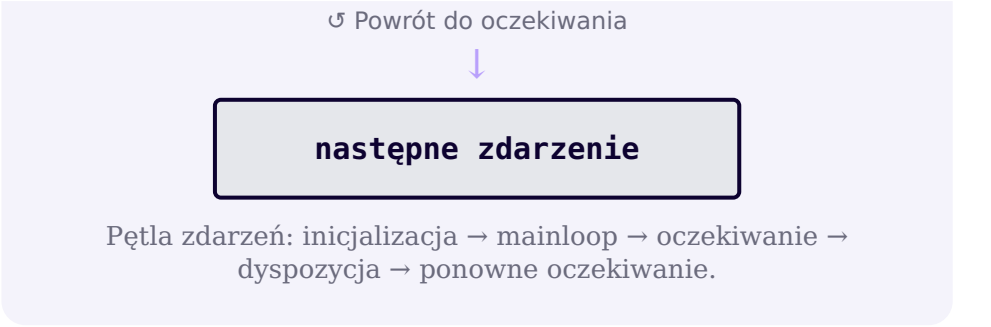
Pętla zdarzeń – nie Python while True

Co naprawdę dzieje się w mainloop()

Nie myśl o tym `mainloop()` jak o waszym własnym `while True: ...` jest wewnętrzną pętlą zdarzeń Tcl/Tk. Poniższy model ma charakter koncepcyjny, aby zrozumieć kolejność zdarzeń, a nie opisuje faktyczną implementację.


inicjalizacji UI
mainloop()
oczekiwanie / obsługa zdarzeń


Tak
wydarzenie?
Call Handlera
callback
return
callback powrócił



Pętla zdarzeń: inicjalizacja → mainloop → oczekiwanie → dyspozycja → ponowne oczekiwanie.

Kolejność zdarzeń — przewidywalna, ale nie sekwencyjna w kodzie

sobytijnyj_cikl.py

```
log = []

def on_click(): log.append("click")
def on_timer(): log.append("timer")
def on_type(): log.append("type")

handlers = {"click": on_click, "timer": on_timer, "type":
on_type}

def run_event_loop(queue):
    for event in queue:
        handlers[event]()

run_event_loop(["click", "timer", "type"])
print(log)   # ['click', 'timer', 'type']
```

Prawdziwe zdarzenia Tkinter pojawiają się, gdy użytkownik i system działają, ale idea jest ta sama To samo: kolejka zdarzeń jest przetwarzana pojedynczo i każda zaczyna własny callback.

Długi callback to osobny temat
Dopóki obsługiwacz działa, pętla zdarzeń czeka na jego zakończenie, zanim przejdzie do następnego zdarzenia. Co się dzieje, jeśli callback działa zbyt długo — szczegółowo w rozdziałach 16.22 i 16.24.

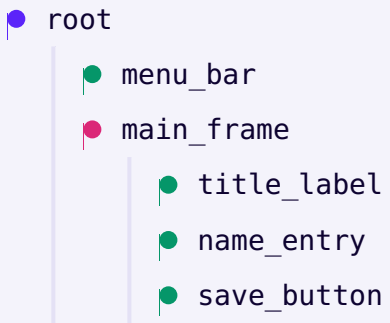
Praktyka: Kolejność obsługi zdarzeń
Inspekcja bez tkinter — symulacja kolejki zdarzeń na czystej Python
Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/16-10/index.html)

Widżet i drzewo interfejsu

Interfejs nie jest płaskim lista elementów, lecz drzewem zagnieżdżonych kontenerów.

Każdy widget ma rodzica

Widgety tworzą drzewo: każdy, oprócz samego korzenia, ma rodzica (master) definiuje kontekst umieszczenia i cyklu życia:



Typowe drzewo małej aplikacji: root → main_frame → oddzielne widżety formularza.

derevo_widgetov.py

```
import tkinter as tk

root = tk.Tk()
main_frame = tk.Frame(root)
main_frame.pack()

title_label = tk.Label(main_frame, text=„Forma”)
title_label.pack()
name_entry = tk.Entry(main_frame)
name_entry.pack()
```

Pierwszy argument prawie każdego widżetu — jego rodzic (`main_frame` powyżej). Rodzic określa, w którym kontenerze znajdzie się widżet — szczegóły o `Frame` w rozdziale 16.13.

Cykl życia widgetu

zniszczony

Tworzenie widgetu tworzy sam obiekt; Menedżer geometrii zawiera ją w układzie okien.

Tworzenie — to tworzenie obiektu, rozmieszczenie — to układ

Tworzenie widżetu (`tk.Button(...)`) tworzy sam Python- obiekt — tak jak tworzenie każdego innego obiektu (rozdział 14). Menedżer geometrii (`pack` , `grid` lub `place`) uwzględnia ten obiekt w układzie okna, aby użytkownik mógł go widzieć i z nim wchodzić w interakcję. Obiekt istnieje w obu przypadkach — jedyną różnicą jest to, czy uczestniczy w układzie widocznego okna.

destroy() nie jest jedynym sposobem na ukrycie widgetu

`.destroy()` niszczy widżet nieodwracalnie. Jeśli wystarczy tymczasowo usunąć widżet z ekranu, są łatwiejsze opcje (na przykład, `.grid_remove()`) — ale dla większości pierwszych aplikacji destrukcyjne usuwanie niepotrzebnych widżetów jest właściwym wyborem.

Praktyka: Drzewo widżetów jako dane

Sprawdza się bez tkinter — modelujemy drzewo przy użyciu zagnieżdżonych danych Python

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/16-11/index.html)

tk i ttk: klasyczne i tematyczne widżety

Zaczynając od tej sekcji, w nowych przykładach wolimy ttk, gdzie istnieje tematyczny odpowiednik — ale najpierw przyjrzyjmy się, jak widżety wyglądają razem.

Tkinter Prezentacja Widgetów

Zanim rozłożymy szczegóły każdego widżetu osobno (sekcje 16.13–16.22), oto jak wyglądają Spójrzcie razem, w jednym prawdziwym oknie:

Showcase: Label, Entry, Button, Checkbutton, Radiobutton, Combobox, Spinbox, Scale, Progressbar, Notebook w jednym oknie

Prawdziwe okno z reprezentatywnym zestawem widżetów rozdziałów. Przykład pojawienia się w środowisku kursu. Czcionki, rozmiary i motywy mogą się nieco różnić w zależności od twojego systemu operacyjnego.

Później w tym rozdziale każdy z tych widżetów otrzyma osobną sekcję z własną stany i przykłady kodeksów.

tk i ttk to dwa zestawy widgetów

	tk	ttk
Co to jest	klasyczne widgety Tkinter	tematycznie (themed) osadzone na Tk
Import	`import tkinter as tk`	`from tkinter import ttk`
Wygląd	Stały styl klasyczny	może dostosować się do tematu/platformy
Stylizacja	`fg=` / `bg=`	przez `ttk.Style()`

Ta sama forma (Label, Entry, Button) złożona dwukrotnie: w klasycznym tk i w tematyzowanym ttk

W tym temacie różnica jest ledwo zauważalna – powinna być: ttk nie gwarantuje „bardziej nowoczesnego”

tk_vs_ttk.py

```
import tkinter as tk
from tkinter import ttk

root = tk.Tk()

ttk.Label(root, text=„Nowoczesna ttk.Label”).pack()
ttk.Button(root, text="ttk.Button").pack()
ttk.Entry(root).pack()
```

W przypadku nowego kodu istnieje ttk, gdzie jest analogia

Od tej sekcji, w nowych przykładach, preferujemy `ttk.Label`, `ttk.Button`, `ttk.Entry` inne tematyczne widgety, gdzie istnieje tematyczny odpowiednik. Ale ttk **nie zastępuje** cały Tk w całości — widżety takie jak `tk.Text`, `tk.Canvas` oraz `tk.Menu` pozostają klasyczne, bo nie mają tematycznego odpowiednika.

ttk nie gwarantuje „bardziej nowoczesnego”

Motyw ttk zależy od systemu operacyjnego i zainstalowanych motywów Tk. Na niektórych platformach/motywach różnica między tk a ttk jest niemal niewidoczna wizualnie (jak na powyższym zrzucie ekranu) – powiedz "tematyczny" zamiast "nowoczesny" czy "ładniejszy".

Nie używaj fg/bg na widżetach ttk jak na klasycznych

Classic `tk` widżety rozumieją `fg=` / `bg=` bezpośrednio. Widgety `ttk` są stylizowane inaczej — poprzez obiekt `ttk.Style()` (następna sekcja). Nie przenosz dosłownie nawyków klasycznych Tk na ttk.

from tkinter import * — nie w stylu produkcyjnym

Import: gwiazdka → jawne nazwy

Nie polecam

```
from tkinter import *

root = Tk()
Button(root, text=„Guzik”)
# skąd Button pochodzi, nie
jest oczywiste
```

Polecam

```
import tkinter as tk
from tkinter import ttk

root = tk.Tk()
ttk.Button(root,
text=„Guzik”)   # źródło
nazwy jest od razu widoczne
```

Co używać dzisiaj: `from tkinter import *` działa, ale utrudnia czytanie kodu: nie jest jasne, do którego modułu należy każda nazwa, a łatwo przypadkowo nakładać się na wbudowane nazwy Python. W przykładach kursów, począwszy od tego rozdziału, pojawiają się tylko jawne importy.

ttk.Style() — stylizacja widgetów tematycznych

Zwykły przycisk obok przycisku stylizowanego przez Accent.TButton

Po lewej stronie jest styl domyślny, po prawej niestandardowy styl nazwany. Przykład wyglądu w środowisku pola. Czcionki, rozmiary i motywy mogą się nieco różnić w zależności od twojego systemu operacyjnego.

ttk_style.py

```
style = ttk.Style()
print(style.theme_use())   #active temat, np. "clam" lub
"default"

style.configure(
    "Accent.TButton",
    font=("TkDefaultFont", 11, "bold"),
    padding=8,
)

button = ttk.Button(
    root,
    text=„Ratuj”,
    style="Accent.TButton",
)
```

ttk.Style() nie jest CSS

`style.configure(...)` akceptuje tylko opcje, które dany widżet i aktywny motyw Tk obsługują, nie jest uniwersalnym językiem stylów CSS-, gdzie można zmienić dowolną właściwość dowolnego elementu. Styl nazwany (`"Accent.TButton"`) jest stosowane przez `style=` przy samym widżecie.

Cartesian Purple
#5B24F9

Cartesian Pink
#DB2777

Success Green
#059669

Warning Amber
#B45309

Kolor – najpierw widziany, potem numeryczny
Zanim napiszesz `#5B24F9` w kodzie, spójrz na rzeczywisty wzór kolorystyczny powyżej — łatwiej jest zapamiętać i odróżniać podobne odcienie niż na podstawie samych kodów szesnastkowych.

Canvas — rysowanie dowolnych kształtów

Canvas z linią, prostokątem, owalem i etykietą współrzędnych

Poniżej znajduje się wynik kodu. Przykład wyglądu w środowisku kursu. Czcionki, rozmiary i motywy mogą się nieco różnić w zależności od Twojego systemu operacyjnego.

canvas_basics.py

```
canvas = tk.Canvas(root, width=260, height=170,
background="white")
canvas.pack(padx=10, pady=10)

canvas.create_line(10, 10, 240, 10, fill="#5B24F9", width=2)
rect_id = canvas.create_rectangle(10, 30, 90, 90,
outline="#DB2777", width=2)
canvas.create_oval(130, 30, 210, 90, outline="#059669",
width=2)
canvas.create_text(130, 130, text=„(0,0) - Lewy górny róg”)
```

Oś Y rośnie w dół, a nie w górę
Pochodzenie `(0, 0)` jest w lewym górnym rogu `Canvas`. `x` rozrasta się w prawo, oraz `y` rośnie **na ziemię** — jest to przeciwieństwem zwykłego systemu współrzędnych matematycznych, w którym y rośnie w górę. Tak działa ekranowy system współrzędnych w większości bibliotek graficznych, nie tylko Tkinter.

create_* zwraca identyfikator elementu
`rect_id = canvas.create_rectangle(...)` — `Canvas` przechowuje wyrysowane elementy i zwraca identyfikator, dzięki któremu możesz później zmienić lub usunąć ten konkretny element (`canvas.itemconfig(rect_id, ...)`, `canvas.delete(rect_id)`) — przydatne w przyszłych interfejsach gier.

Canvas nie jest drugim Turtle

`Canvas` jest zwykłym Tkinter widżetem do rysowania dowolnych kształtów w oknie aplikacji, a nie osobnego modułu z własnym światem współrzędnych, takiego jak `turtle` (rozdziały 6-7). To krótkie wprowadzenie, a nie pełny kurs Canvas.

PhotoImage — obrazy w widżetach

Label z małym wygenerowanym obrazem szachownicy

Obraz jest generowany bezpośrednio w Python, bez pliku zewnętrznego. Przykład wyglądu w środowisku kursu. Na Twoim systemie operacyjnym czcionki, rozmiary i motyw mogą się nieco różnić.

photoimage.py

```
image = tk.PhotoImage(file="icon.png")
label = ttk.Label(root, image=image)
label.pack()
```

DEBUG LAB 2: OBRAZ ZNIKA, MIMO ŻE KOD JEST „POPRAWNY”

photoimage_propadaet.py

```
def build_ui():
    image = tk.PhotoImage(file="icon.png")   # funkcje
zmienna lokalne
    label = ttk.Label(root, image=image)
    label.pack()

build_ui()
```

```
# Окно открывается, но картинка не отображается
# на месте изображения пусто.
```

Co widać na ekranie

Po ukończeniu `build_ui()` lokalna nazwa `image` znika. Jeśli na obiekcie `PhotoImage` nie pozostaje już żadnych odwołań Pythona, obiekt może zostać zwolniony — widżet przechowuje odniesienie do niego tylko w Tcl, a nie w samym Pythonie. Dlatego aplikacja powinna przechowywać odniesienie do `PhotoImage` tyle czasu, ile obraz powinien być wyświetlany.

POPRAWIONY KOD

photoimage_fixed.py

```
class App:
    def __init__(self, root):
        self.logo_image =
tk.PhotoImage(file="icon.png")   # przechowywane w self
—
        self.logo_label = ttk.Label(root,
image=self.logo_image)   # żyje tak długo, jak self
        self.logo_label.pack()
```

Praktyka: zbieramy formularz z tk i ttk
Moduł tkinter otwiera natywne okno Python — wykonaj lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/16-12/index.html)

Frame i LabelFrame: organizują interfejs

Zanim umieścisz dziesiątki widżetów, podziel okno na zrozumiałe regiony.

Frame jest struktura przed układem

`ttk.Frame` jest kontenerem bez własnej widocznej zawartości, które grupują powiązane widgety w znaczące regiony:

- root
  - toolbar_frame
  - content_frame
  - status_frame

Podzielenie okna na nazwane regiony upraszcza zarówno kod, jak i myślenie o layout.

frame.py

```
import tkinter as tk
from tkinter import ttk

root = tk.Tk()

toolbar_frame = ttk.Frame(root)
content_frame = ttk.Frame(root)
status_frame = ttk.Frame(root)

toolbar_frame.pack(side="top", fill="x")
content_frame.pack(side="top", fill="both", expand=True)
status_frame.pack(side="bottom", fill="x")
```

W każdej klatce możesz używać niezależnie `pack()` lub `grid()` – zasada „nie mieszać”

LabelFrame jest grupą wizualnie podpisaną

Górny pasek z przyciskami Otwórz/Zapisz, poniżej — podpisana ramka „Ustawienia” z dwoma polami wyboru

Toolbar przez Frame (bez ramki) i LabelFrame z widocznym podpisem „Ustawienia”

labelframe.py

```
nastrojki = ttk.LabelFrame(root, text=„Ustawienia”)
nastrojki.pack(padx=10, pady=10, fill="x")

ttk.Checkbutton(nastrojki, text=„Mroczny
motyw”).pack(anchor="w")
```

LabelFrame nie jest na każdą drobnostkę

Zastosowanie `LabelFrame` dla naprawdę powiązanej grupy ustawień, a nie dla każdego pojedynczego widżetu — w przeciwnym razie interfejs zamienia się w drabinę klatek.

Praktyka: organizowanie interfejsu poprzez Frame

Moduł tkinter otwiera natywne okno Python — wykonaj lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/16-13/index.html)

pack szczegółowo: fill, expand i zagnieżdżone ramki

pack nie jest tylko z góry: ma kontrolę nad wolną przestrzenią.

pack: sposobu przydzielania przestrzeni

`side` decyduje, po której stronie zostanie położony następny widget, `fill` — rozciągać ją na całą dostępną szerokość/wysokość, a `expand` to, czy wykorzystać dodatkowe wolne miejsce:

Parametr	Znaczenie	Efekt
fill	"x"	rozciągnąć widget na szerokość przydzielonego mu paska
fill	"y"	rozciągnąć widżet na wysokość przydzielonego mu pasa
fill	"both"	rozciągają zarówno szerokość, jak i wysokość

Parametr	Znaczenie	Efekt
expand	True	dać widżetowi dodatkową wolną przestrzeń w kontenerze

pack_fill_expand.py

```
ttk.Label(root, text=„Top”,
background="#E7DEFF").pack(side="top", fill="x")
ttk.Label(root, text=„Center”,
background="#B9A0FC").pack(side="top", fill="both",
expand=True)
ttk.Label(root, text=„Bottom”,
background="#E7DEFF").pack(side="bottom", fill="x")
```

Wcięcie zewnętrzne i wewnętrzne to nie to samo
`padx` / `pady` u `.pack(...)` — to wcięcie **na zewnątrz** widżetu, między nim a sąsiadami. Odstęp **w środku** samego widżetu (pomiędzy ramką a treścią) może być konfigurowana osobno — na przykład przez `padding=` u `ttk.Frame`.

Zagnieżdżone ramki + pack

vlozhennye_frejmy.py

```
forma_frame = ttk.Frame(root, padding=10)
forma_frame.pack(fill="x")

ttk.Label(forma_frame, text=„Imię:”).pack(side="left")
ttk.Entry(forma_frame).pack(side="left", fill="x", expand=True)
```

Etykieta jest naciskana po lewej, a pole wejściowe zajmuje całą pozostałą przestrzeń poziomą to typowa kombinacja `side="left"` + `fill="x"` + `expand=True` dla formy Signature - Stretch Field.

Praktyka: pack z fill i expand
Moduł tkinter otwiera natywne okno Python — wykonaj lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/16-14/index.html)

Adaptacyjne grid: sticky i weight

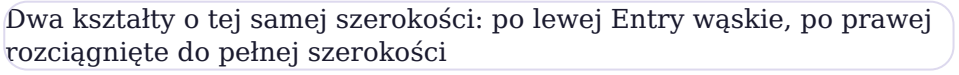
Mundur nie musi pozostawać w tym samym rozmiarze na zawsze – grid wie, jak go dostosować.

Adaptive grid: Kto dostaje dodatkowe miejsce

Domyślne wiersze i kolumny `grid()` nie rosną przy powiększaniu okna. Aby formularz dopasowywał się do rozmiaru okna, kolumnom/łańcuch znaków należy jawnie przydzielić **waga**:

adaptivny_grid.py

```
root.columnconfigure(1, weight=1)   # to kolumna 1 (z polami wejściowymi) jest rozciągana
root.rowconfigure(0, weight=1)

ttk.Label(root, text=„Imię:”).grid(row=0, column=0, sticky="w")
ttk.Entry(root).grid(row=0, column=1, sticky="ew")
# rozciąga się na całą szerokość komórki
```



Dwa kształty o tej samej szerokości: po lewej Entry wąskie, po prawej rozciągnięte do pełnej szerokości

Ta sama szerokość okna (340px) – tylko columnconfigure(weight=1 różnica) oraz sticky=„ew”

raspredelenie_vesa.py

```
def raspredelit_prostranstvo(weights, extra_space):
    total = sum(weights)
    if total == 0:
        return [0] * len(weights)
    return [extra_space * w / total for w in weights]

print(raspredelit_prostranstvo([1, 2], 300))   # [100.0, 200.0]
```

weight to współczynnik obrazu, a nie piksele

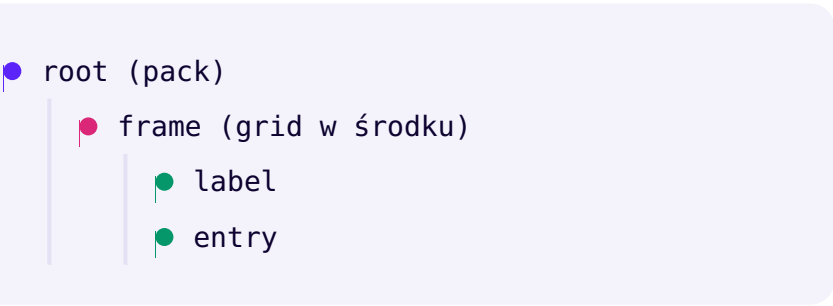
Waga określa **część** dodatkowej przestrzeni, a nie rozmiaru absolutnego. Kolumna o wadze 2 otrzyma dwa razy więcej miejsca niż kolumna o wadze 1 — ale obie nadal będą mieszczać swoją zwykłą minimalną zawartość.

sticky — do którego brzegu komórki przykleja się widżet

sticky	Efekt
"w" / "e" / "n" / "s"	przylega do jednej krawędzi komórki (zachód/wschód/północ/południe)
"ew"	rozciąga się na szerokość komórki
"nsew"	rozciąga się na całą komórkę — zarówno na szerokość, jak i na wysokość

Zasada jednego rodzica

✓ **Właściwie**



✗ **Błąd**

Mix `pack()` oraz `grid()` nie możesz widgety z **to samo** nadrzędny – ale inne pojemniki (root Zastosowania `pack()`, a zagnieżdżony `Frame` w środku — `grid()`) są całkowicie niezależne od siebie.

place() - Precyzyjne pozycjonowanie

place.py

```
label.place(x=20, y=10)
label2.place(relx=0.5, rely=0.5, anchor="center")   # środkowy pojemnik
```

place() nie jest pierwszym wyborem dla tradycyjnych form

`place()` jest przydatny do precyzyjnych nakładek i specjalnych okazji, ale nie dostosowuje się automatycznie do rozmiaru okna jak `grid()` z łuskami. Dla konwencjonalnych form adaptacyjnych, `grid()`.

Jak wybrać menedżera geometrii

pack vs grid vs place

Łatwy układ/regiony pionowe	→ **pack()**
Formularz/tabela/adaptacyjne kolumny	→ **grid()**
Precyzyjne lub specjalne pozycjonowanie, nakładki	→ **place() – świadomie**
Złożona przegroda	→ **Frame + różni menedżerowie u różnych rodziców**

Praktyka: rozkład wagi w adaptacyjnym grid
Sprawdza się bez tkinter — czysta arytmetyka rozdzielania przestrzeni
Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/16-15/index.html)

Widżety wyboru: Combobox, Listbox, Spinbox, Scale

Każdy z tych widżetów rozwiązuje inną wersję jednego zadania: wybiera wartość z ograniczonego zbioru.

Combobox — wybierz z listy

Zamknięty Combobox z wybraną wartością „Mały”

Stan zamknięty. Przykład wyglądu w środowisku kursu. Czcionki, rozmiary i motywy mogą się nieco różnić w zależności od twojego systemu operacyjnego.

combobox.py

```
variant = ttk.Combobox(root, values=[„Mały”, „Medium”, „Duży”], state="readonly")
variant.current(0)
variant.pack()
```

Otwarty rozwijany lista Combobox z trzema opcjami

Open State (rozwijane lista) to prawdziwy zrzut ekranu, a nie symulacja. Przykład pojawienia się w środowisku kursu. Na twoim systemie operacyjnym czcionki, rozmiary i motywy mogą się nieco różnić.

state="readonly"
Bez `state="readonly"` użytkownik będzie mógł wpisywać wolny tekst w polu, które nie jest częścią lista. Jeśli musisz wybrać spośród proponowanych opcji, użyj `"readonly"` .

Listbox jest lista do wyboru jednego/więcej elementów

Listbox z czterema elementami podkreślono „Chleb”

Jeden element jest podświetlony (widoczny przez podkreślenie). Przykład wyglądu w środowisku kursu. Czcionki, rozmiary i motywy mogą się nieco różnić w zależności od systemu operacyjnego.

listbox.py

```
spisok = tk.Listbox(root)
spisok.insert("end", „Mleko”)
spisok.insert("end", „Chleb”)
spisok.insert("end", „Apples”)
spisok.pack()

def pokazat_vybor():
    indeksy = spisok.curselection()
    print([spisok.get(i) for i in indeksy])
```

Spinbox jest polem z strzałkami do wyboru krok po kroku

Spinbox z wartością 3 i strzałkami w górę/dół

Strzałki po prawej zwiększają/zmniejszają wartość krokami. Przykład wyglądu w środowisku kursu. Na twoim systemie operacyjnym czcionki, rozmiary i motyw mogą się nieco różnić.

spinbox.py

```
kolichestvo = ttk.Spinbox(root, from_=1, to=10)
kolichestvo.pack()
```

from_/to nie jest ścisłą ochroną przed jakimkolwiek tekstem
`Spinbox` — to pole, podobne do `Entry`, z dodanymi strzałkami, które iterują wartości z określonego zakresu. Opcje `from_` / `to` określają zakres dla samych strzałek — nie stanowią uniwersalnej gwarancji, że użytkownik nie wpisze dowolnego tekstu bezpośrednio w pole. Jeśli wymagany jest ścisły wybór tylko z zakresu — konieczne będzie dodatkowe przemyślenie walidacji lub stanu podobnego do readonly-.

Scale — suwak dla wartości z zakresu

Scale jest poziomym suwakiem z wartością 50 na oznaczce obok

Suwak (track) i pokrętło (thumb); wartość pokazana w sąsiedniej etykiecie. Przykład wyglądu w środowisku kursu. Na twoim systemie operacyjnym czcionki, rozmiary i motyw mogą się nieco różnić.

scale.py

```
gromkost = ttk.Scale(root, from_=0, to=100,
orient="horizontal")
gromkost.set(50)
gromkost.pack()
```

Scale powraca float
`.get()` u `Scale` zwraca liczbę zmiennoprzecinkową, nawet jeśli wizualnie suwak stoi na pełnym podziale — zaokrąglaj jawnie (`round(...)`), jeśli potrzebujesz liczby całkowitej.

Praktyka: Combobox, Listbox, Spinbox, Scale
Moduł tkinter otwiera natywne okno Python — wykonaj lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/16-16/index.html)

Progressbar i tabletki Notebook

Postęp powinien być sprawiedliwy, a zakładki naturalnym sposobem na dużą ilość personalizacji.

Progressbar to postęp, a nie dekoracja

Cztery Progressbar przy 0%, 35%, 70% i 100%

Rzeczywiste stany tego samego widżetu o różnych wartościach value. Przykład wyglądu w środowisku kursu. Czcionki, rozmiary i motywy mogą się nieco różnić w zależności od systemu operacyjnego.

progressbar.py

```
progress = ttk.Progressbar(root, mode="determinate",
maximum=100, value=0)
progress.pack(fill="x")

def obnovit_progress(procent):
    progress["value"] = procent
```

Mode	Kiedy używać
determinate	zna dokładną liczbę etapów/procent ukończenia
indeterminate	prace są w toku, ale ich objętość nie jest znana z góry

Nie fałszuj postępu, którego nie znasz
Jeśli aplikacja nie jest w stanie oszacować procentu ukończenia, użyj `mode="indeterminate"` (pasek biegający) zamiast dowolnych liczb w `determinate`. Fałszywy postęp jest gorszy niż szczere „nie wiesz, ile jeszcze zostało”

progressbar_indeterminate.py

```
progress = ttk.Progressbar(root, mode="indeterminate")
progress.pack(fill="x")

progress.start()    # uruchamia animację taśmy biegnącej
# ... długa operacja ...
progress.stop()     # zatrzymuje animację
```

indeterminate nie animuje się samodzielnie
Tworzenie `Progressbar(mode="indeterminate")` nie uruchamia automatycznie animacji – ruch paska pełzającego zaczyna się dopiero po wyraźnym wywołaniu `.start()` i kończy się na `.stop()`.

Notebook — tabletki

Notebook z trzema zakładkami: Ogólne, Wygląd, Pliki — pierwsza jest aktywna

trzech zakładkach, aktywne jest "Ogólne" z polem "Autosave" w środku. Przykład pojawienia się w środowisku kursu. Na twoim systemie operacyjnym czcionki, rozmiary i motywy mogą się nieco różnić.

notebook.py

```
vkladki = ttk.Notebook(root)
vkladki.pack(fill="both", expand=True)

obshaya_vkladka = ttk.Frame(vkladki)
vneshnij_vid_vkladka = ttk.Frame(vkladki)
fajly_vkladka = ttk.Frame(vkladki)

vkladki.add(obshaya_vkladka, text=„Generał”)
vkladki.add(vneshnij_vid_vkladka, text=„Wygląd”)
vkladki.add(fajly_vkladka, text=„Pliki”)
```

Każda zakładka jest regularną `Frame`, wewnątrz którego można niezależnie używać `pack()` lub `grid()` — doskonale nadaje się do okna ustawień (rozdział 16.25). Na zrzucie ekranu powyżej widać dokładnie to, co opisuje termin „zakładka” — przełączalny obszar treści w jednym oknie, a nie kilka oddzielnych okien.

Praktyka: Progressbar i Notebook
Moduł tkinter otwiera natywne okno Python — wykonaj lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/16-17/index.html)

Messagebox i dialogi

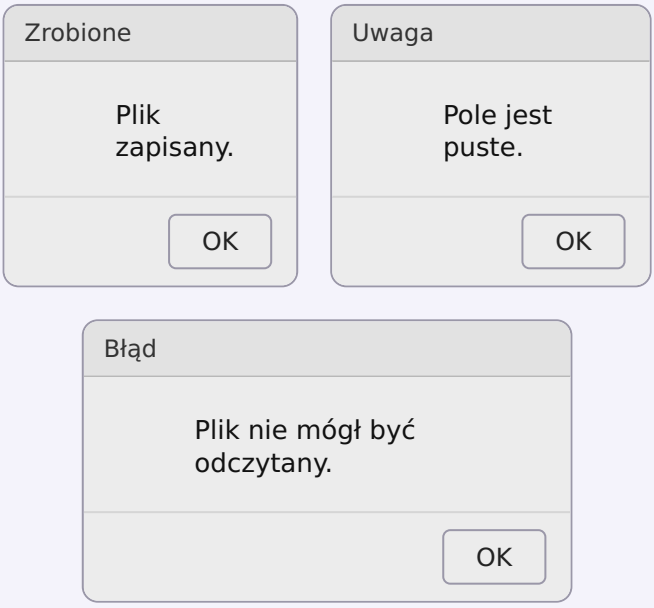
Użytkownik musi dowiedzieć się o sukcesie, ostrzeżeniu lub błędzie – nie z terminala.

messagebox — standardowe dialogi powiadomień

messagebox.py

```
from tkinter import messagebox

messagebox.showinfo(„Gotowe”, „Plik zapisany.”)
messagebox.showwarning(„Uwaga”, „Pole jest puste.”)
messagebox.showerror(„Błąd”, „Plik nie mógł zostać odczytany.”)
```

SCHEMATYCZNY OBRAZ (NIE PRAWDZIWY ZRZUT EKRANU)



Trzy rodzaje powiadomień — różne ikony i znaczenia, ta sama zasada dzwonienia.

Wygląd dialogów zależy od systemu operacyjnego
Prawdziwy wygląd tych dialogów zależy od systemu operacyjnego, wersji Tk i wybranej tematyki – dokładnego natywnego obrazu środowiska kursu headless- nie można stabilnie odtworzyć, dlatego tutaj użyto schematycznego rysunku, a nie zrzutu ekranu.

Dialogi z dwoma opcjami – odczytaj wartość zwrotną

messagebox_yesno.py

```
soglasen = messagebox.askyesno(„Potwierdzenie”, „Usunąć tę notatkę?”)
if soglasen:
    udalit_zametku()
```

Nie zgaduj wyniku po nazwie przycisku
Nie zakładaj z góry, co zwróci dialog — czytaj faktyczną wartość zwracaną `True` / `False` dla `askyesno`) i reagują na to, a nie na założenie, że „użytkownik prawdopodobnie kliknie tak”

Trzy rezultaty: Tak / Nie / Anulowanie

W przypadku scenariusza „wypożycz z niezapisanymi zmianami”

messagebox_yesnocancel.py

```
answer = messagebox.askyesnocancel(
    „Niezapisane zmiany”,
    „Zapisz zmiany przed wyjściem?”,
)

if answer is None:
    pass   # Anuluj — wcale nie odchodź, użytkownik zmienił zdanie
elif answer:
    save_document()
    root.destroy()
else:
    root.destroy()   # Nie — wychodzimy, nie zapisując
```

Wartość zwrotna	Co nacisnął użytkownik
`True`	Tak
`False`	Tak
`None`	Anuluj (lub zakończ dialog krzyżykiem)

Częstym błędem jest mylenie Tak/Nie z Zapisz/Zamknij
"Tak" odpowiada na pytanie "zapisz?", a nie "wyjść?". Nie pisz `if answer: save() else: root.destroy()` — wtedy „Nie" cicho zamknie okno, nawet nie pytając, a „Tak" zapisze, ale nie zamknie. Pełną działającą wersję dla prawdziwego edytora — uwzględniając oba działania i cofnięcie — zobacz w sekcji 16.30.

Modalność
Standardowe dialogi `messagebox` są modalne: blokują interakcję z resztą aplikacji, dopóki nie zostaną zamknięte. W przypadku natywnych modali są bardziej subtelne narzędzia (`transient` , `grab_set`) — rozdział 16.20, oznaczone jako zaawansowane.

Praktyka: dialogi messagebox i wybory
Moduł tkinter otwiera natywne okno Python — wykonaj lokalnie w VS Code, PyCharm lub Jupyter

`Otwórz praktykę →` (https://www.cartesianschool.org/pl/practice/16-18/index.html)

Otwieranie i zapisywanie plików: filedialog i pathlib

Okno wyboru pliku pokazuje ścieżkę, a potem wszystko, co już wiemy z rozdziału 15, działa.

Bezpośredni link do rozdziału 15: okno wyboru pliku pokazuje ścieżkę, a potem wszystko działa tak samo `pathlib` .

filedialog.py

```
from tkinter import filedialog
from pathlib import Path

filename = filedialog.askopenfilename(filetypes=[(„Pliki
tekstowe”, "*.txt")])
if not filename:
    pass   # użytkownik kliknął „Anuluj”
else:
    text = Path(filename).read_text(encoding="utf-8")
```

Nie zapomnij o anulowaniu

Jeśli użytkownik zamknął dialog bez wyboru pliku, `askopenfilename()` wróci **pusty wiersz**, nie `None` nie jest drogą. Próbuję od razu otworzyć `Path("")` jest częstym błędem. Zawsze sprawdź wynik przed użyciem: `if not filename: return`.

Zapisywanie pliku

filedialog_save.py

```
filename =
filedialog.asksaveasfilename(defaultextension=".txt")
if filename:
    Path(filename).write_text(text_widget.get("1.0",
"end-1c"), encoding="utf-8")
```

Prawidłowa kolejność: najpierw sprawdzenie anulowania, potem praca ze ścieżką.

Praktyka: filedialog + pathlib
Moduł tkinter otwiera natywne okno Python — wykonaj lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/16-19/index.html)

Toplevel: wiele okien

Prawdziwa aplikacja prawie zawsze potrzebuje więcej niż jednego okna, ale nie więcej niż jednego root.

Toplevel — dodatkowe okno

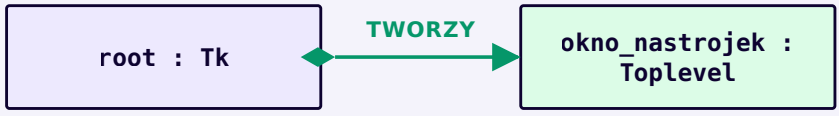
Główne okno aplikacji i oddzielne okno ustawień obok

Toplevel jest osobnym rzeczywistym oknem, a nie panelem wewnątrz głównego okna. Przykład wyglądu w środowisku kursu. Czcionki, rozmiary i motywy mogą się nieco różnić w zależności od twojego systemu operacyjnego.

Dla drugiego okna (ustawienia, „O mnie” `tk.Toplevel`:

toplevel.py

```
def otkryt_nastrojki():
    okno_nastrojek = tk.Toplevel(root)
    okno_nastrojek.title(„Ustawienia”)
    ttk.Label(okno_nastrojek, text=„Tutaj będą
ustawienia”).pack(padx=20, pady=20)

ttk.Button(root, text=„Ustawienia”,
command=otkryt_nastrojki).pack()
```



Jeden root, dodatkowe okna — przez Toplevel

Do regularnego zastosowania, **jeden** korzeń `Tk()`, a dodatkowe okna są tworzone przez `Toplevel`. Technicznie można stworzyć kilka niezależnych `Tk()` możliwe — każdy ma własny Tcl interpretera — ale dla zwykłej, pojedynczej aplikacji zwykle nie jest to to, czego chcesz i nie jest to zalecana struktura. Zasada ogólna: jedna aplikacja → jedna root → `Toplevel` na resztę okien.

Trochę głębiej: Modal

Niektóre dialogi muszą blokować interakcję z oknem nadrzędnym, dopóki nie będą zamknięte:

modalnoe_okno.py

```
okno = tk.Toplevel(root)
okno.transient(root)     # okno wizualnie należy do rodzica
okno.grab_set()          # przyjmuje wszystkie dane wejściowe
aż do zamknięcia
root.wait_window(okno)   # rodzic czeka na zamknięcie tego okna
```

Niekoniecznie na początku

Modale niestandardowe to zaawansowana technika. Dla większości zastosowań edukacyjnych i małych zastosowań stosuje się standardowe `Toplevel` bez `grab_set()` w zupełności wystarczy.

Praktyka: okno ustawień przez Toplevel

Moduł tkinter otwiera natywne okno Python — wykonaj lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/16-20/index.html)

Focus, klawiatura i podstawy dostępności

Nie każdy użytkownik ma mysz — a zrozumiała forma powinna działać również bez niej.

Skupienie na wejściu

Wejście z klawiatury zawsze trafia tylko do jednego widżetu — tego, który obecnie ma **skupienie**:

focus.py

```
entry.focus_set()      # przenieść fokus do tego pola programowo
print(root.focus_get())   # który widżet jest teraz w fokusu
```

> **Przygotowanie gruntu pod rozdział 17**
> Skupienie to ważna koncepcja dla dalszej pracy z wydarzeniami klawiaturowymi (`.bind("<Key>", ...)`), który szczegółowo przeanalizujemy w rozdziale 17.

Kolejność przechodzenia między polami (Tab)

Użytkownik musi umieć wypełniać formularz wyłącznie za pomocą klawiatury – nawigując między nimi pola z kluczem `Tab` w rozsądny sposób.

> **Kolejność Tab to nie tylko „w porządku stworzenia”**
> Przesuwanie przez klawiaturę opiera się na zasadach przesuwania fokusów Tk drzewie widżetów i ich pozycji oraz tego, czy dany widget w ogóle jest zaangażowany w uzyskanie fokusu (the `takefocus`). Kolejność tworzenia wpływa na naturalną strukturę prostej formy, ale nie jest pełną zasadą. Praktyczna rada beginner-: buduj elementy formy w logicznej kolejności, koniecznie **sprawdzone** ręcznie przechodzić przez Tab, unikać niepotrzebnie skupionych widgetów i korzystać z nich `takefocus` świadomie, gdzie chcesz wyraźnie uwzględnić lub wykluczyć widget z indeksowania.

Podstawy dostępności

Mały, ale prawdziwy lista

TAGI

dla każdego pola — widoczna etykieta
nie tylko tekst-zastępczy (placeholder)

STATUS

nie tylko kolorem — także tekstem/ikonką
disabled — gdy działanie jest niedostępne

KLAWIATURA

rozsądna kolejność Tab
nie tylko mysz jako jedyna metoda

WIADOMOŚCI

zrozumiały tekst błędu
co poszło nie tak i jak to naprawić

Tkinter nie decyduje o dostępności dla ciebie

Użycie Tkinter nie gwarantuje automatycznie dostępnego interfejsu na wszystkich platformach. Wymienione zasady — to, co zależy od decyzji w Twoim kodzie i oznaczeniach, a nie od samego faktu używania biblioteki GUI-.

proverka_dostupnosti.py

```
forma = [
    {"name": "name_entry", "has_label": True, "tab_index": 0},
    {"name": "email_entry", "has_label": True, "tab_index": 1},
    {"name": "submit_button", "has_label": True, "tab_index":
2},
]

def vse_imeyut_metku(widgets):
    return all(w["has_label"] for w in widgets)

def poryadok_posledovatelen(widgets):
    indeksy = [w["tab_index"] for w in widgets]
    return indeksy == sorted(indeksy)
```

Praktyka: Sprawdź dostępność formularza

Zweryfikowane bez tkinter – na modelu formularza jako lista danych

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/16-21/index.html)

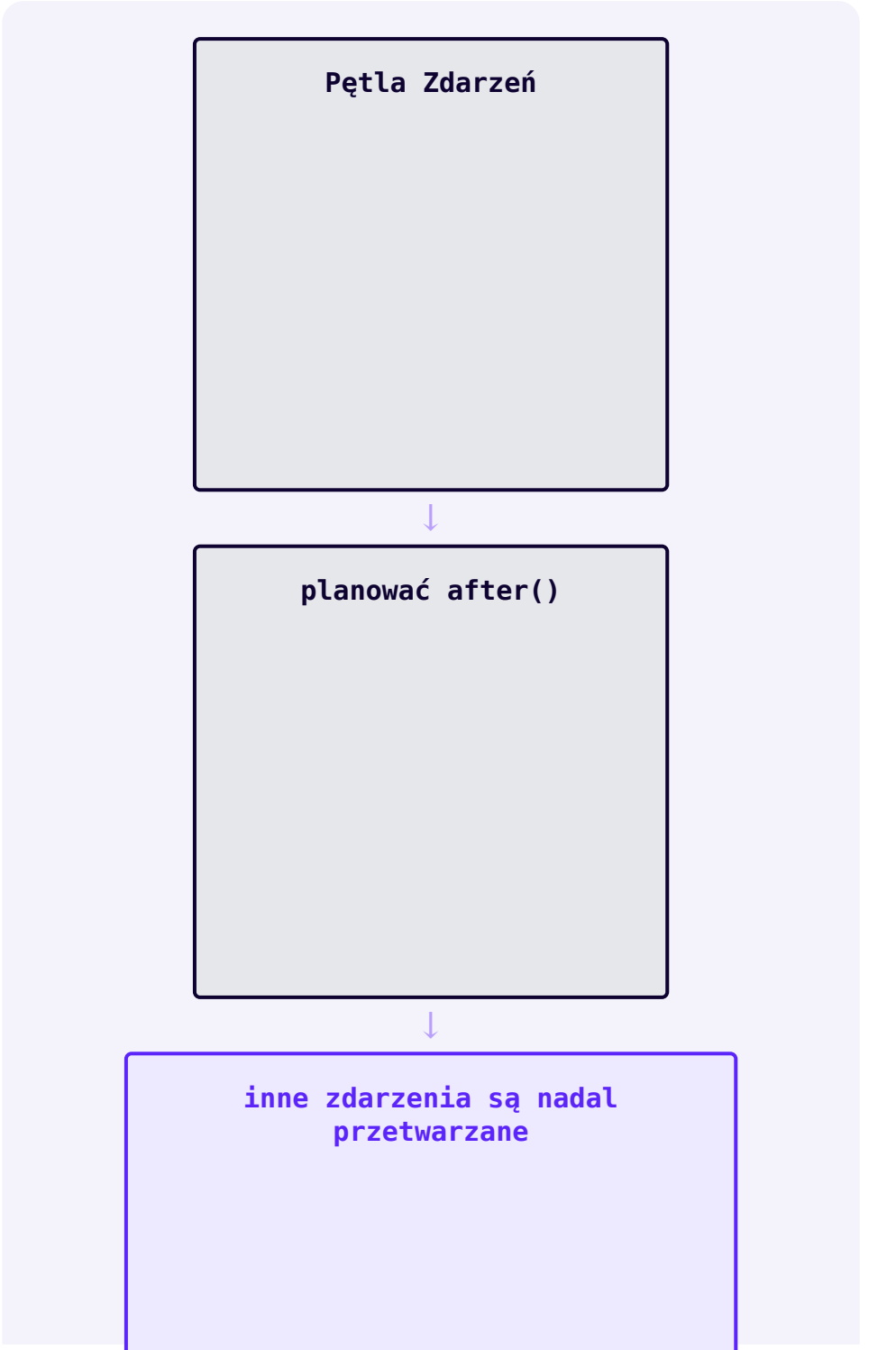
after(): timery bez blokowania

Czekanie nie oznacza zatrzymania całej aplikacji.

after() - Odroczone połączenie bez blokowania

after_prosto.py

```
after_id = root.after(1000, lambda: print(„Minęła sekunda”))
# program nadal przetwarza inne wydarzenia, czekając na tę
sekundę
```


Pętla Zdarzeń
planować after()
inne zdarzenia są nadal przetwarzane

„Za sekundę” — nie jest to stoper z dokładnością do milisekundy
`after(1000, callback)` oznacza „przygotuj callback do wykonania mniej więcej za 1000 ms, kiedy cykl zdarzeń będzie mógł go obsłużyć” — nie gwarantuje wykonania dokładnie w podanej milisekundzie. Jeśli w tym momencie cykl jest zajęty innym zdarzeniem, wywołanie nastąpi trochę później.

Powtarzający się timer (self-rescheduling)

tik.py

```
after_id = None

def tik():
    global after_id
    label.config(text=f"Sekundy: {schet.get()}")
    schet.set(schet.get() + 1)
    after_id = root.after(1000, tik)   # planuje się na nowo — to jest powtarzalność
```

Nie tylko for/while jest powtarzalną
Z rozdziału 10 znamy cykle `for` / `while`. Powtarzające się połączenia przez self-scheduling `after()` inny sposób organizacji powtarzania na podstawie zdarzeń, a nie sekwencyjnego bloku kodu.

Każdy nowy after() musi zaktualizować zapisane id

U `tik()` nowe wyzwanie `after()` przy każdym zaznaczeniu, a więc nowy identyfikator. Jeśli nie przydzielisz ponownie `after_id` na każdym połączeniu (jak wyżej) zapisany zmienna szybko staje się przestarzały — będzie wskazywał już wywołane połączenie, a nie faktycznie oczekujące, a próba zatrzymania timera na tym id niczego nie zatrzyma.

Zatrzymanie timera: after_cancel

after_cancel.py

```
def stop():
    global after_id
    if after_id is not None:
        root.after_cancel(after_id)
        after_id = None
```

Nie zgub dowodu tożsamości

Aby zatrzymać zaplanowaną rozmowę, musisz zapisać **istotne** identyfikator, który oddzwonił na ostatnie połączenie `after()`. Bez niego nie będziesz mógł anulować tego konkretnego zaplanowanego połączenia. Kompletny model trzech stanów (zatrzymany/w trakcie/zakończony) z ważnym `after_id` zobacz mini-projekt „Timer”

DEBUG LAB 3: PONOWNE URUCHOMIENIE TWORZY ZDUBLOWANE TIMERY

duplicate_after.py

```
def start():
    tik()   # każde kliknięcie na „Start”

ttk.Button(root, text=„Start”, command=start).pack()
```

```
# После нескольких нажатий «Старт» счётчик начинает прыгать
# работает уже несколько параллельных цепочек after().
```

Co widać na ekranie

Każde kliknięcie uruchamia nowy niezależny łańcuch samoplanowanych wywołań `tik()`, i one się nawzajem nie zastępują, lecz działają jednocześnie. Należy zablokować ponowne uruchomienie przez wspólny znacznik `running` — i ta sama flaga musi być sprawdzona samodzielnie `tik()`, w przeciwnym razie łańcuch nie zatrzyma się nawet po `stop()`.

POPRAWIONY KOD

duplicate_after_fixed.py

```
running = False
after_id = None

def tik():
    global after_id, running
    if not running:
        return   # stop() już opuścił flagę — nie
planujemy nowej sieci
    label.config(text=f"Sekundy: {schet.get()}")
    schet.set(schet.get() + 1)
    after_id = root.after(1000, tik)

def start():
    global running
    if running:
        return   # już działa, zignoruj drugie
kliknięcie
    running = True
    tik()

def stop():
    global running, after_id
    running = False
    if after_id is not None:
        root.after_cancel(after_id)
        after_id = None
```

Nigdy nie time.sleep() w callback
`time.sleep(...)` w obsłudze zatrzymuje całą pętlę zdarzeń na cały czas trwania snu, a interfejs przestaje odpowiadać. Szczegółowy podział można znaleźć w sekcji 16.32.

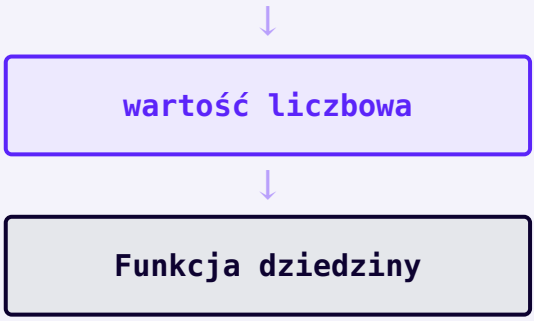
Praktyka: after() odliczanie
Moduł tkinter otwiera natywne okno Python — wykonaj lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/16-22/index.html)

Walidacja i informacja zwrotna

Surowy tekst użytkownika nie staje się liczbą sam w sobie — musi być między nimi sprawdzenie.

Walidacja – przed liczeniem



Wprowadzanie nie zamienia się samo w liczbę — między łańcuch znaków a obliczeniem jest sprawdzenie.

Podgląd try/except

To jest minimalna praktyczna zapowiedź `try/except`. Na razie użyjmy go jako gotowego szablonu do sprawdzania danych użytkowników. Wyjątki zostaną szczegółowo omówione w rozdziale 21.

parse_number.py

```
def parse_number(text):
    text = text.strip()
    if not text:
        return False, None, „Pole nie może być puste"
    try:
        return True, float(text), ""
    except ValueError:
        return False, None, „Wprowadź numer”

print(parse_number(""))      # (False, None, ...)
print(parse_number("abc"))   # (False, None, ...)
print(parse_number("-5"))
# (True, -5.0, '') — liczba ujemna to LICZBA!
print(parse_number("100"))   # (True, 100.0, '')
```

Nie każda liczba musi być dodatnia

`parse_number` sprawdza tylko „czy to w ogóle liczba?” — i celowo akceptuje wartości ujemne i zero: one też są liczbami. Na przykład temperatura w stopniach Celsjusza może być ujemna (rozdział 16.27) — wymaganie „dodatnie” jest specyficzne dla konkretnego zadania (sumy pieniędzy, liczba osób), a nie cechą liczb ogólnie.

Specjalistyczna walidacja na parse_number

validate_positive.py

```
def validate_positive_amount(text):
    ok, value, message = parse_number(text)
    if not ok:
        return False, message
    if value <= 0:
        return False, „Liczba musi być większa od zera"
    return True, ""

def validate_positive_int(text):
    ok, value, message = parse_number(text)
    if not ok:
        return False, message
    if value != int(value):
        return False, „Wprowadź liczbę całkowitą”
    if int(value) < 1:
        return False, „Liczba musi wynosić co najmniej 1”
    return True, ""
```

Różne zadania — różne funkcje

Kwota rachunku musi być dodatnia — `validate_positive_amount`. Liczba osób musi być dodatnia **całość** — `validate_positive_int` dalsze odrzucenia `2.5`. Obie funkcje ponownie wykorzystują `parse_number`, a nie powielają analizę tekstu.

Jak pokazać błąd użytkownikowi

status_label.py

```
ok, message = validate_positive_amount(schet_entry.get())
if not ok:
    status_label.config(text=message, foreground="#DB2777")
    return
```

Błąd — na ekranie, a nie tylko w terminalu
GUI- użytkownik zazwyczaj nie patrzy na terminal. Pokaż błąd za pomocą widżetu statusu lub `messagebox.showerror(...)` (Rozdział 16.18) - Pieczęć `print(...)` do terminala nie jest uznawane za informację zwrotną dla użytkownika w aplikacji okienkowej.

Trochę głębiej: validatecommand

Tkinter ma wbudowany mechanizm walidacji na poziomie widgetu (`validate=`, `validatecommand=`, `register(...)`) — jego składnia nie jest najbardziej intuicyjna, więc Dla większości formularzy w tym kursie wystarczy ręczne sprawdzenie przed obliczeniem, jak wyżej.

Praktyka: parse_number i specjalistyczna walidacja
Testowane bez tkinter – na czystych funkcjach

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/16-23/index.html)

Architektura aplikacji: logika osobno od widżetów

Najlepszy GUI-logic to ten, który można przetestować nawet bez otwierania okna.

Czysta logika oddzielna od widżetów

Bezpośrednie powiązanie z rozdziałem 13: funkcję domenową można napisać i przetestować **niezależnie** z interfejsu to po prostu wejście i wyjście, bez żadnego widżetu:

chistaya_logika.py

```
def calculate_tip(amount, percent, people):
    total_tip = amount * percent / 100
    return total_tip / people

print(calculate_tip(1000, 15, 2))   # 75.0
```

wyświetlić wynik w widżecie

Callback jest cienką warstwą pośrednią; cała logika obliczeniowa znajduje się poza interfejsem.

Callback nie powinien się rozrastać
Jeśli obsługa przycisków zajmuje sto wierszy zagnieżdżonych obliczeń, warto przenieść część tej logiki do osobnej funkcji, którą można zrozumieć i sprawdzić bez uruchamiania okna.

Trochę głębiej: Pułapka późnego wiązania w lambda

DEBUG LAB 4: WSZYSTKIE PRZYCISKI W PĘTLI „PAMIĘTAJĄ”

pozdnee_svyazyvanie.py

```
for i in range(3):
    ttk.Button(root, text=str(i), command=lambda:
print(i)).pack()
```

```
# Все три кнопки выведут одно и то же число — последнее зна
# а не 0, 1, 2 соответственно.
```

Co widać na ekranie

`lambda: print(i)` odwołuje się do zmiennej `i` z okolicznego pola widzenia **w momencie kliknięcia**zamiast momentu utworzenia przycisku (Rozdział 13, Zamknięcia). W momencie kliknięcia cykl już się zakończył, i `i` jest równe jej ostatniej wartości 2 dla wszystkich trzech przycisków jednocześnie.

POPRAWIONY KOD

pozdnee_svyazyvanie_fixed.py

```
for i in range(3):
    ttk.Button(root, text=str(i), command=lambda i=i:
print(i)).pack()
# i=i tworzy wartość domyślną, ustaloną WŁAŚNIE w
momencie stworzenia lambda
```

proverka_lambda_fix.py

```
callbacks = []
for i in range(3):
    callbacks.append(lambda i=i: i)

values = [cb() for cb in callbacks]
print(values)   # [0, 1, 2]
```

Praktyka: Czysta logika i pułapka lambda
Sprawdzane bez tkinter — na funkcjach i domknięciach

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/16-24/index.html)

Klasa aplikacji i ustawienia trwałe

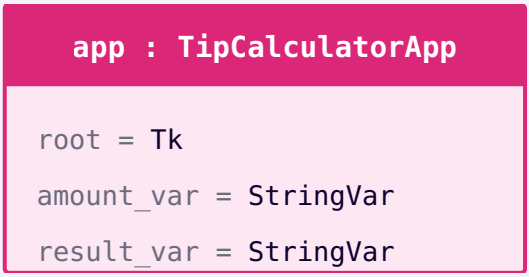
Aplikacja jest także obiektem: z własnym stanem, widgetami atrybutów i metodami obsługi użytkownika.

klasa aplikacji to kompozycja, a nie dziedziczenie Tk

Bezpośrednie powiązanie z rozdziałem 14. Dla aplikacji bardziej skomplikowanych niż para widżetów wygodnie jest zebrać wszystko w jednej klasie:

klass_prilozheniya.py

```
class TipCalculatorApp:
    def __init__(self, root):
        self.root = root
        self.amount_var = tk.StringVar()
        self.build_ui()

    def build_ui(self):
        ttk.Entry(self.root,
textvariable=self.amount_var).pack()
        ttk.Button(self.root, text=„Czytaj”,
command=self.on_calculate).pack()

    def on_calculate(self):
        ...

root = tk.Tk()
app = TipCalculatorApp(root)
root.mainloop()
```



Graf obiektów aplikacji: App HAS-A root, nie App IS-A Tk.

App zawiera root, a nie dziedziczy po Tk

`class App:` z `self.root = root` jest głównym modelem tego kursu: zależność od Tk jest wyraźnie widoczna w konstruktorze. Dziedziczenie `class App(tk.Tk):` jest również możliwe i występuje w praktyce, ale tutaj łatwiej zrozumieć skład, a nie "błędny" kontra "właściwy".

Nie każdy widget jest atrybutem self

Zapisz do `self.виджет` tylko to, do czego potrzebujesz uzyskać dostęp później z innej metody (pole wejściowe, etykieta wyniku). Tymczasowe widgety utworzone i umieszczone w jednym miejscu mogą pozostać jako zmienne lokalne `build_ui()`.

Ustawienia trwałe – Przejdź do rozdziału 15

nastroyki_gui.py

```
import json
from pathlib import Path

DEFAULT_SETTINGS = {"theme": "light", "window_width": 900}

def load_settings(path):
    if not path.exists():
        return dict(DEFAULT_SETTINGS)
    with path.open("r", encoding="utf-8") as f:
        return json.load(f)

def save_settings(path, settings):
    with path.open("w", encoding="utf-8") as f:
        json.dump(settings, f, ensure_ascii=False, indent=2)
```

Ani jedna linijka o JSON i plikach nie jest nowa — wszystko już było w rozdziale 15.

Ani jednej nowej idei dotyczącej plików

`load_settings()` / `save_settings()` korzystać z niczego poza tym, czego już się nauczyliśmy w rozdziale 15 — `json`, `pathlib`, bezpieczne domyślne ustawienia. GUI po prostu nazywa te funkcje przy starcie i przed zamknięciem.

Praktyka: load_settings/save_settings dla GUI

Sprawdza się bez tkinter — przenosimy funkcje persistence rozdziału 15

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/16-25/index.html)

Mini-projekt: licznik kliknięć

Mały projekt, ale prawdziwa interakcja: kliknij → callback → zmień stan → odśwież ekran.

Najmniejszy prawdziwy szkic rozdziału: liczba kliknięć. Dobra okazja, by zobaczyć oba sposób przechowywania numeru powiązanego z widżetem.

Kliknij okno licznika z wartością 3 i przyciskiem +1

Wynik wykonania kodu poniżej, po trzech kliknięciach. Przykład wyglądu w środowisku kursu. Na Twoim systemie operacyjnym czcionki, rozmiary i temat mogą się nieznacznie różnić.

schetchik_intvar.py

```
import tkinter as tk
from tkinter import ttk

root = tk.Tk()
root.title(„Kliknij licznik”)

schet = tk.IntVar(value=0)
ttk.Label(root, textvariable=schet, font=("Arial",
24)).pack(pady=20)

def na_klik():
    schet.set(schet.get() + 1)

ttk.Button(root, text="+1", command=na_klik).pack(pady=10)

root.mainloop()
```

Druga metoda to bez zmiennej Tk-

Ten sam miernik można zmontować z normalnym `int` i jawne `label.config(text=...)` po każdej zmianie – działa równie dobrze; `IntVar` jest przydatny, gdy wartość musi być synchronizowana z wieloma widgetami jednocześnie (Rozdział 16.4).

DEBUG LAB 5: ZAPOMNIAŁEM MAINLOOP() – OKNO NIE DZIAŁA TAK, JAK POWINNO

bez_mainloop.py

```
root = tk.Tk()
ttk.Button(root, text="+1").pack()
# zapomniano root.mainloop()!
```

```
# Скрипт завершается почти сразу — окно либо не появляется,
# либо мгновенно закрывается, не дожидаясь никаких действий г
```

Co widać na ekranie

Bez `root.mainloop()` program nie przechodzi do pętli obsługi zdarzeń (rozdział 16.10) — po prostu liniowo dochodzi do końca skryptu i kończy działanie jak zwykły program terminalowy. Widżety są stworzone i nawet umieszczone, ale nikt nie czeka na kliknięcia w nie.

POPRAWIONY KOD

s_mainloop.py

```
root = tk.Tk()
ttk.Button(root, text="+1").pack()
root.mainloop()   # bez tego przywoływania wydarzeń nie będzie żadnych wydarzeń
```

Praktyka: licznik kliknięć
Moduł tkinter otwiera natywne okno Python — wykonaj lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/16-26/index.html)

Mini Project: Konwerter temperatury

Prosta forma – ale z wzorową architekturą: czystą funkcją konwersji niezależną od interfejsu.

W praktyce forma „→ wejściowa sprawdzać → czystą funkcję → wyniku”

konverter_temperatur.py

```python
def celsius_v_farengejty(celsius):
    return celsius * 9 / 5 + 32

def farengejty_v_celsius(farengejty):
    return (farengejty - 32) * 5 / 9

print(round(celsius_v_farengejty(100), 1))    # 212.0
print(round(celsius_v_farengejty(-40), 1))    # -40.0
print(round(celsius_v_farengejty(0), 1))      # 32.0
```

Temperatura nie musi być dodatnia
0°C, −5°C, −40°C to wszystkie temperatury całkowicie normalne. Przetwornik temperatury nie powinien być nadużywany `validate_positive_amount` (Rozdział 16.23) odrzuciłby dane ujemne i zerowe jako "błąd". Potrzebne jest ogólne sprawdzenie "czy to liczba?" `parse_number`.

konverter_gui.py

```python
import tkinter as tk
from tkinter import ttk

def celsius_v_farengejty(celsius):
    return celsius * 9 / 5 + 32

def parse_number(text):
    text = text.strip()
    if not text:
        return False, None, „Pole nie może być puste"
    try:
        return True, float(text), ""
    except ValueError:
        return False, None, „Wprowadź numer”

root = tk.Tk()
celsius_var = tk.StringVar()
result_var = tk.StringVar()

def on_convert():
    ok, value, message = parse_number(celsius_var.get())
    if not ok:
        result_var.set(message)
        return
```

```
    farengejty = celsius_v_farengejty(value)
    result_var.set(f"{farengejty:.1f} °F")

ttk.Entry(root, textvariable=celsius_var).pack()
ttk.Button(root, text=„W Fahrenheita”,
command=on_convert).pack()
ttk.Label(root, textvariable=result_var).pack()
```

Konwerter temperatury: -40 wprowadzony, wynik -40,0° F

−40°C = −40°F — jedyny punkt, w którym skale Celsjusza i Fahrenheita pokrywają się. Wynik wykonania powyższego kodu. Przykład wyglądu w środowisku kursu. Na Twoim systemie operacyjnym czcionki, rozmiary i motyw mogą się nieco różnić.

parse_number już wiem

Funkcja `parse_number` jest taki sam, jak ten, który napisaliśmy w sekcji 16.23. Po napisaniu i przetestowaniu funkcja parsowania liczb przydaje się w kilku projektach — tutaj bez dodatkowego wymogu „pozytywnego”

★★ Zadanie samodzielne

Oba

Dodaj drugie pole oraz przycisk „W stopniach Celsjusza”

Praktyka: Przetwornik temperatury

Testowane bez tkinter – na czystych funkcjach konwersji, w tym wartości ujemnych

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/16-27/index.html)

Mini-Projekt: Odliczający Timer

after() w akcji: odliczanie bez żadnej akcji blokującej.

Countdown Timer – Wzorowe Demo `after()` i Wniosek stwierdza.

format_time.py

```python
def format_time(sekundy):
    minuty, ostatok = divmod(sekundy, 60)
    return f"{minuty:02d}:{ostatok:02d}"

print(format_time(65))    # 01:05
print(format_time(600))   # 10:00
```

Okno timera: 00:42 oraz przyciski Start, Stop, Reset

Wyświetlanie startowe — do pierwszego tykania. Przykład wyglądu w środowisku kursu. Na waszym systemie operacyjnym czcionki, rozmiary i motyw mogą się nieco różnić.

Trzy stany timera

STOPPED → RUNNING → FINISHED — i z powrotem do STOPPED przez Reset (nie pokazane osobną strzałką).

tajmer_gui.py

```
class TimerApp:
    def __init__(self, root, start_seconds=60):
        self.root = root
        self.start_seconds = start_seconds
        self.remaining = start_seconds
        self.running = False
        self.after_id = None
        self.label_var =
tk.StringVar(value=format_time(self.remaining))
        ttk.Label(root, textvariable=self.label_var).pack()
        ttk.Button(root, text=„Start”,
command=self.start).pack()
        ttk.Button(root, text=„Stop”, command=self.stop).pack()
        ttk.Button(root, text=„Reset”,
command=self.reset).pack()

    def start(self):
        if self.running or self.remaining <= 0:
            return   # już idzie, lub już FINISHED – najpierw
Reset
        self.running = True
        self._schedule_tick()

    def _schedule_tick(self):
        self.after_id = self.root.after(1000, self.tick)

    def tick(self):
        self.remaining -= 1
        self.label_var.set(format_time(self.remaining))
        if self.remaining <= 0:
            self.running = False
            self.after_id = None   #FINISHED – Nie ma już nic
do planowania
            return
        self._schedule_tick()

    def stop(self):
        self.running = False
        if self.after_id is not None:
            self.root.after_cancel(self.after_id)
```

```
            self.after_id = None

    def reset(self):
        self.stop()
        self.remaining = self.start_seconds
        self.label_var.set(format_time(self.remaining))
```

after(1000, ...) — „w ciągu około sekundy”, nie chronometr

`after(1000, callback)` oznacza „zrób callback gotowym do wykonania po około 1000 ms, gdy cykl zdarzeń będzie mógł go obsłużyć” — nie gwarantuje wykonania dokładnie w wskazaną milisekundę. Jeśli cykl w tym momencie jest zajęty innym zdarzeniem, wywołanie nastąpi nieco później.

DEBUG LAB 6: TIMER CIĄGLE TYKA PO „STOP”

tajmer_bez_proverki.py

```
def tick(self):
    self.remaining -= 1
    self.label_var.set(format_time(self.remaining))
    self.after_id = self.root.after(1000, self.tick)
# planuj ZAWSZE

def stop(self):
    self.running = False   # status został zmieniony,
ale tick() go nie sprawdza
```

```
# После нажатия «Стоп» таймер продолжает уменьшать remaining
# флаг running изменился, но ничто не проверяет его внутри ti
```

Co widać na ekranie

`stop()` zmienia tylko flagę `running`, ale sama metoda `tick()` nie testuje tego przed przełożeniem — łańcucha `after()` żyje niezależnie od flagi.

POPRAWIONY KOD

tajmer_s_proverkoj.py

```
def tick(self):
    if not self.running:   # sprawdź status PRZED
kontynuowaniem
        return
    self.remaining -= 1
    self.label_var.set(format_time(self.remaining))
    self.after_id = self.root.after(1000, self.tick)
```

Praktyka: Odliczanie

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/16-28/index.html)

Mini-projekt: lista zadania

Proste lista linii i cała mała aplikacja z zapisywaniem między przejściami.

Lista zadań — integracja rozdziału 11 (listy), rozdziału 15 (JSON) i Tkinter (Listbox).

todo.json

Dane, mapowanie i trwałość to trzy różne obowiązki jednej małej aplikacji.

todo_logika.py

```
import json
from pathlib import Path

def load_tasks(path):
    if not path.exists():
        return []
    with path.open("r", encoding="utf-8") as f:
        return json.load(f)

def save_tasks(path, tasks):
    with path.open("w", encoding="utf-8") as f:
        json.dump(tasks, f, ensure_ascii=False, indent=2)

def add_task(tasks, text):
    if not text.strip():
        return tasks
    return tasks + [text.strip()]

def remove_task(tasks, index):
    return tasks[:index] + tasks[index + 1:]
```

Lista zadań: trzy zadania, drugie jest podświetlone, pole wejściowe oraz przyciski + / −

Wynik wykonania kodu znajduje się poniżej. Przykład pojawienia się w środowisku kursu. Czcionki, rozmiary i motywy mogą się nieznacznie różnić w zależności od systemu operacyjnego.

Pojedyncze funkcje callback bez klasy są źródłem błędu
Callback trochę `on_add()`, zadeklarowany samodzielnie (nie w innej funkcji), nie można użyć `nonlocal tasks` — `nonlocal` wymaga zmiennej z funkcji obejmującej, a nie z zakresu globalnego. Bardziej odpowiednią klasą zastosowań tutaj (Rozdział 14) jest `self.tasks` rozwiązuje ten problem naturalnie.

todo_gui.py

```
class TodoApp:
    def __init__(self, root, path):
        self.root = root
        self.path = path
        self.tasks = load_tasks(path)
        self.new_task_var = tk.StringVar()
        self.listbox = tk.Listbox(root, height=6)
        self.listbox.pack()
        entry = ttk.Entry(root, textvariable=self.new_task_var)
        entry.pack(side="left", fill="x", expand=True)
        ttk.Button(root, text="+",
command=self.on_add).pack(side="left")
        ttk.Button(root, text="-",
command=self.on_delete).pack(side="left")
        self.refresh_listbox()

    def refresh_listbox(self):
        self.listbox.delete(0, "end")
        for task in self.tasks:
            self.listbox.insert("end", task)

    def on_add(self):
        self.tasks = add_task(self.tasks,
self.new_task_var.get())
        self.new_task_var.set("")
        self.refresh_listbox()
        save_tasks(self.path, self.tasks)

    def on_delete(self):
        selected = self.listbox.curselection()
        if not selected:
            return
        self.tasks = remove_task(self.tasks, selected[0])
        self.refresh_listbox()
        save_tasks(self.path, self.tasks)
```

Logika jest sprawdzalna bez żadnego widżetu

`add_task`, `remove_task`, `load_tasks`, `save_tasks` są zwykłe funkcje z listami i słownikami. To właśnie sprawdza praktyka tej sekcji — dokładnie ten podział omawiany w sekcji 16.24.

Praktyka: Logika listy zadań

Sprawdziłem bez tkinter – funkcje pracy z listą zadań

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/16-29/index.html)

Mini-Projekt: Edytor Notatek

Dziennik notatek z rozdziału 15 wreszcie dostaje prawdziwe okno i menu.

Bezpośrednia kontynuacja dziennika notatek z rozdziału 15 (15.4) – teraz z prawdziwym oknem, pole wieloliniowe oraz menu „Plik”

Okno edytora notatek z tekstem notatek

Wynik wykonania kodu znajduje się poniżej. Przykład pojawienia się w środowisku kursu. Czcionki, rozmiary i motywy mogą się nieznacznie różnić w zależności od systemu operacyjnego.

redaktor_zametok.py

```
class NotesEditorApp:
    def __init__(self, root):
        self.root = root
        self.current_path = None
        self.dirty = False
        self.text_widget = ScrolledText(root, wrap="word")
        self.text_widget.pack(fill="both", expand=True)
        self.text_widget.bind("<<Modified>>",
self.on_text_modified)
        self.build_menu()
        self.root.protocol("WM_DELETE_WINDOW", self.on_exit)

    def on_text_modified(self, event):
        self.dirty = True
        self.text_widget.edit_modified(False)   # zresetuj
wbudowaną flagę Tk

    def confirm_discard_if_dirty(self):
```

```
        """True – można kontynuować (New/Open/Exit). False –
użytkownik zmienił zdanie."
        if not self.dirty:
            return True
        answer = messagebox.askyesnocancel(
            „Niezapisane zmiany”,
            „Zapisz zmiany przed kontynuowaniem?”,
        )
        if answer is None:   #Cancel – Przerwij bieżącą
aktywność
            return False
        if answer:           #Yes – Kontynuuj tylko wtedy, gdy
zapis faktycznie się powiódł
            return self.save_as()
        return True          # No – kontynuuj, nie zapisując

    def on_exit(self):
        if self.confirm_discard_if_dirty():
            self.root.destroy()

    def save_as(self):
        """True - Plik został zapisany. False - Użytkownik
anulował okno "Zapisz jako".
        filename =
filedialog.asksaveasfilename(defaultextension=".txt")
        if not filename:
            return False
        self.current_path = Path(filename)

self.current_path.write_text(self.text_widget.get("1.0",
"end-1c"), encoding="utf-8")
        self.dirty = False
        return True
```

Trzy wyniki jednego pytania

`messagebox.askyesnocancel(...)` zwroty `True` (Tak – zapisz), `False` (Brak – Nie zapisuj) lub `None` (Anuluj/zamknij dialog — całkowicie przerwij bieżącą akcję). Normalne `askyesno` można zrobić tylko dwie pierwsze opcje — do „Wyjścia z niezapisanymi zmianami”

Yes – jeszcze nie został uratowany

Jeśli użytkownik wybrał „Tak”, ale następnie kliknął „Anuluj” w dialogu `asksaveasfilename`, plik nie został zapisany — i kontynuowanie działania (zamykanie okna, otwieranie innego pliku) jest niemożliwe, w przeciwnym razie niezapisany tekst zostanie utracony. Dlatego `confirm_discard_if_dirty()` zwraca wynik `save_as()`, nie `True` natychmiast po zadzwonieniu „Tak”

Nic fundamentalnie nowego w plikach

`filedialog` (16.19), `Path.read_text/write_text(encoding="utf-8")` (Rozdział 15), `ScrolledText` (gotowa kombinacja `Text` + `Scrollbar`) — wszystkie części są już znane, tutaj są zebrane w jednym zastosowaniu.

★★ Zadanie samodzielne

Nowy i Otwórz też pytają

Dodaj new_file() i open_file() metod, z których każda wywołuje confirm_discard_if_dirty() jako pierwsza — i kontynuuje tylko wtedy, gdy zwróciła True.

Praktyka: Edytor notatek

Moduł tkinter otwiera natywne okno Python — wykonaj lokalnie w VS Code, PyCharm lub Jupyter

`Otwórz praktykę →` (https://www.cartesianschool.org/pl/practice/16-30/index.html)

Tip Calculator Pro: wersja ostateczna

Ten sam projekt co w sekcji 16.8 — ale teraz, mając wszystko, czego się od tamtej pory nauczyliśmy.

Wróćmy do kalkulatora napiwków (sekcja 16.8) i zbierzmy wszystko, czego się nauczyliśmy w tym Rozdział — etapami, a nie jednym wielkim, ostatnim kawałkiem kodu.

Droga od fundacji do Pro

V1–V2 (16.8)

Entry + Button + grid()
czysta formuła napiwków

V3

Combobox z typowymi procentami zamiast Entry

V4

pole „liczba osób”

V5

validate_positive_amount/validate_positive_int (16.23)

V6

menu plików z Exit (16.6)

V7

load_settings/save_settings – zapamiętujemy ostatni procent (16.25)

V8

```
klasa TipCalculatorApp — wszystko w jednym
obiekcie (16.25)
```

Ustawienia są zapisywane względem aktualnego katalogu roboczego
`SETTINGS_PATH = Path("tip_calculator_settings.json")` jest ścieżka względna: plik pojawi się tam, gdzie program został uruchomiony (Rozdział 15, Rozdział 15.7 o CWD). Dla projektu uczącego się jest to świadomy, wyraźny wybór — nie zapomniana niepewność.

tip_calculator_pro.py

```
import json
import tkinter as tk
from tkinter import ttk
from pathlib import Path

SETTINGS_PATH = Path("tip_calculator_settings.json")
DEFAULT_SETTINGS = {"last_percent": "15"}

def load_settings():
    if not SETTINGS_PATH.exists():
        return dict(DEFAULT_SETTINGS)
    with SETTINGS_PATH.open("r", encoding="utf-8") as f:
        return json.load(f)

def save_settings(settings):
    with SETTINGS_PATH.open("w", encoding="utf-8") as f:
        json.dump(settings, f, ensure_ascii=False, indent=2)

def parse_number(text):
    text = text.strip()
    if not text:
        return False, None, „Pole nie może być puste"
    try:
        return True, float(text), ""
    except ValueError:
        return False, None, „Wprowadź numer”

def validate_positive_amount(text):
```

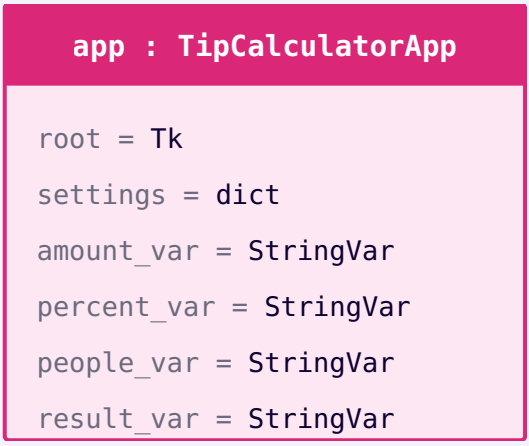
```
    ok, value, message = parse_number(text)
    if not ok:
        return False, message
    if value <= 0:
        return False, „Liczba musi być większa od zera"
    return True, ""

def validate_positive_int(text):
    ok, value, message = parse_number(text)
    if not ok:
        return False, message
    if value != int(value):
        return False, „Wprowadź liczbę całkowitą”
    if int(value) < 1:
        return False, „Liczba musi wynosić co najmniej 1”
    return True, ""

def calculate_tip(amount, percent, people):
    return (amount * percent / 100) / people

class TipCalculatorApp:
    def __init__(self, root):
        self.root = root
        self.root.title("Tip Calculator Pro")
        self.settings = load_settings()
        self.amount_var = tk.StringVar()
        self.percent_var =
tk.StringVar(value=self.settings["last_percent"])
        self.people_var = tk.StringVar(value="1")
        self.result_var = tk.StringVar()
        self.build_ui()
        self.root.protocol("WM_DELETE_WINDOW", self.on_close)

    def build_ui(self):
        frame = ttk.Frame(self.root, padding=12)
        frame.grid(row=0, column=0, sticky="nsew")
        self.root.columnconfigure(0, weight=1)

        ttk.Label(frame, text=„Suma konta:”).grid(row=0,
column=0, sticky="w")
        ttk.Entry(frame,
textvariable=self.amount_var).grid(row=0, column=1,
sticky="ew")

        ttk.Label(frame, text=„Procent napiwków:”).grid(row=1,
column=0, sticky="w")
        ttk.Combobox(frame, textvariable=self.percent_var,
values=["10", "15", "20"],
```

```
            state="readonly").grid(row=1, column=1,
sticky="ew")

        ttk.Label(frame, text=„Liczba osób:”).grid(row=2,
column=0, sticky="w")
        ttk.Entry(frame,
textvariable=self.people_var).grid(row=2, column=1,
sticky="ew")

        ttk.Button(frame, text=„Obliczaj”,
command=self.on_calculate).grid(
            row=3, column=0, columnspan=2, pady=8)
        ttk.Label(frame,
textvariable=self.result_var).grid(row=4, column=0,
columnspan=2)
        frame.columnconfigure(1, weight=1)

    def on_calculate(self):
        ok, message =
validate_positive_amount(self.amount_var.get())
        if not ok:
            self.result_var.set(message)
            return
        ok_people, message_people =
validate_positive_int(self.people_var.get())
        if not ok_people:
            self.result_var.set(„Liczba osób: ” +
message_people)
            return
        chaevye = calculate_tip(
            float(self.amount_var.get()),
            float(self.percent_var.get()),
            int(self.people_var.get()),
        )
        self.result_var.set(f"napiwki na osobę: {chaevye:.2f}")
        self.settings["last_percent"] = self.percent_var.get()

    def on_close(self):
        save_settings(self.settings)
        self.root.destroy()

root = tk.Tk()
app = TipCalculatorApp(root)
root.mainloop()
```

Liczba osób jest dodatnią liczbą całkowitą, a nie dowolną
nie ma 2,5 osoby. `validate_positive_int` (Rozdział 16.23) przyjmuje `1`, `2`, `5`, `10` — i odrzucone `0`, `-1`, `2.5`, `"abc"` i pusta linia z wyraźnym komunikatem o błędzie.

Okno Tip Calculator Pro z sumą rachunku 1000, procentem 15, liczbą osób 2 i wynikiem 75.00

Wyjściem powyższego kodu jest prawdziwe okno, a nie ilustracja. Przykład wyglądu w środowisku kursu. Czcionki, rozmiary i motywy mogą się nieco różnić w zależności od systemu operacyjnego.



Ostateczny graf obiektowy: jedna aplikacja, jawne zależności, brak zmiennych globalnych.

Praktyka: Tip Calculator Pro
Moduł tkinter otwiera natywne okno Python — wykonaj lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/16-31/index.html)

Debugowanie interfejsu i jakość GUI

Responsywność interfejsu nie jest automatyczną konsekwencją korzystania z Tkinter, lecz wynikiem konkretnych decyzji w kodzie.

Pętla zdarzeń nie może być wypożyczona przez długi czas

DEBUG LAB 7:TIME.SLEEP() WEWNĄTRZ CALLBACK ZAMRAŻA INTERFEJS

sleep_v_callback.py

```
import time

def on_click():
    time.sleep(5)   # „czekać 5 sekund”
    label.config(text=„Gotowe!”)
```

```
# На все 5 секунд окно перестаёт перерисовываться и реагирова
# событийный цикл заблокирован внутри time.sleep(), а не толь
```

Co widać na ekranie

`time.sleep(...)` zatrzymuje **cały proces**, w którym działa pętla zdarzeń — a Tkinter zwykle działa w jednym wątku. Gdy wątek śpi, pętla nie może obsłużyć w ogóle niczego: ani przerysowania, ani kliknięć, ani zamknięcia okna.

POPRAWIONY KOD

after_vmesto_sleep.py

```
def on_click():
    label.config(text=„Czekamy...”)
    root.after(5000, lambda:
label.config(text=„Gotowe!”))
    # pętla zdarzeń trwa przez te wszystkie 5 sekund
```

DEBUG LAB 8: WHILE TRUE ZAMIAST PĘTLI ZDARZEŃ

while_true_gui.py

```
while True:
    if button_nazhata():
        obrabotat_klik()
```

```
# Такого кода не должно быть в Tkinter-приложении вообще —
# у Tk уже есть собственный событийный цикл внутри mainloop
```

Co widać na ekranie

Tkinter już zapewnia pełnoprawną pętlę zdarzeń. Niestandardowe `while True` dla „pollowania”

POPRAWIONY KOD

mainloop_vmesto_while.py

```
button.config(command=obrabotat_klik)
root.mainloop()   # c obsługuje zdarzenia, w tym
kliknięcia na button
```

DEBUG LAB 9: DŁUGIE OBLICZENIA BEZPOŚREDNIO W CALLBACK

dolgoe_vychislenie.py

```
def on_click():
    result = obrabotat_million_zapisej()   # sekundy obliczeń wewnątrz callback
    label.config(text=str(result))
```

```
# Интерфейс "подвисает" на всё время вычисления — ровно так же,
# как и с time.sleep(), просто по другой причине.
```

Co widać na ekranie

Każda czasochłonna operacja w callback nie jest tylko `time.sleep()` — zajmuje cykl zdarzeń na cały ten czas. Dla naprawdę długich zadań: dzielić pracę na części, planowane przez `after()`, czyli przeniesienie go do osobnego wątku/procesu z bezpiecznym powrotem wyniku, to już zaawansowany temat, który nie wymaga pełnej implementacji w tym kursie.

POPRAWIONY KOD

razbivka_cherez_after.py

```
def obrabotat_chast(indeks):
    if indeks >= len(dannye):
        label.config(text=„Gotowe”)
        return
    obrabotat_odnu_zapis(dannye[indeks])
    root.after(0, obrabotat_chast, indeks + 1)   # następna część jest o kolejnym cyklu
```

Aktualizacja widżetu – ze strumienia pętli zdarzeń

Jeśli w twoim kodzie pojawi się podgląd z wątkiem roboczym, wynik powinien być bezpiecznie przekazany z powrotem i widżety aktualizowane z pętli zdarzeń Tkinter, a nie bezpośrednio z innego wątku. Pełna architektura wielowątkowa wykracza poza zakres tego rozdziału.

Testowanie tego, co można przetestować bez okna

test_domennoj_logiki.py

```
def test_calculate_tip():
    assert calculate_tip(100, 10, 2) == 5.0

test_calculate_tip()
print(„Logika domeny sprawdzona bez pojedynczego widżetu.”)
```

Lista kontrolna zanim uważysz GUI- projekt gotowy

- ☐ Okno się otwiera
- ☐ Wszystkie widżety są widoczne
- ☐ Kolejność Tab na klawiaturze jest rozsądna
- ☐ Przycisk wyświetla oczekiwany callback
- ☐ Nieprawidłowe wejście jest obsługiwane przez komunikat czystą
- ☐ Zmiana rozmiaru okna nie psuje układu
- ☐ Zamknięcie okna kończy proces w sposób elegancki
- ☐ Dane są przechowywane tam, gdzie są potrzebne

Praktyka: blokujący vs nieblokujący handler
Zweryfikowane bez tkinter - na modelu kolejki zdarzeń
Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/16-32/index.html)

Podsumowanie rozdziału: Zestaw narzędzi Tkinter

Od „program czeka na zdarzenia, nie w kolejności”

Wizualna lista kontaktowa widżetów

Zanim wybierzesz imię, pamiętaj, jak ono wygląda:

Rozdział 16 — lista kontaktów

Label / Entry / Button
tekst, pole wprowadzania, przycisk

witryna widgetów

Checkbutton
niezależne pole wyboru

usunięty/zainstalowany

Radiobutton
grupa wzajemnie wykluczających się wariantów

wybrano jeden z trzech

Combobox

wybór z listy rozwijanej

otwarty lista

Listbox

lista z doborem pierwiastków

wybrany przedmiot

Spinbox / Scale

liczba ze strzałkami / suwakiem

Spinbox

Scale

Progressbar

pasek postępu

0–100%

Notebook

przełączalne zakładki

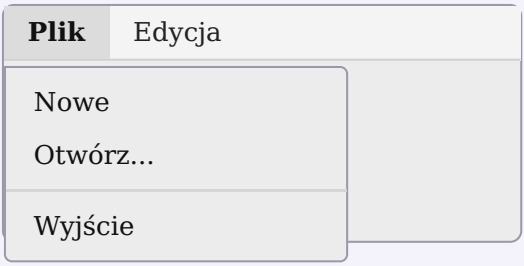
trzy zakładki

Toplevel

osobnym drugim oknem

główne + ustawienia

SCHEMATYCZNY OBRAZ (NIE PRAWDZIWY ZRZUT EKRANU)



Zestaw narzędzi Tkinter

Co wybrać dla konkretnych zadań

Potrzebne jest główne okno aplikacji	→ `tk.Tk() – jeden na każdy proces`
Potrzebujemy kolejnego okna	→ `tk.Toplevel(root)`
Potrzebny widżet formularza z motywem	→ `ttk.Label/Button/Entry/Combobox/...`
Potrzebny edytor wielolinijowy	→ `tk.Text / ScrolledText`

Łatwe ustawienie pionowe	→ `pack()`
Formularz/tabela/ adaptacyjne kolumny	→ `grid() + weight/sticky`
Precyzyjne pozycjonowanie	→ `place() – świadomie`
Ogólny status wielu widgetów	→ `StringVar/IntVar/BooleanVar/DoubleVar`
Powiadomienie użytkownika	→ `messagebox`
Wybór plików	→ `filedialog + pathlib`
Odłożone/powtarzające się wywołanie bez blokady	→ `root.after(...)`
Ustawienia trwałe	→ `JSON + pathlib (Rozdział 15)`
Duża struktura aplikacji	→ `klasa App + małe callback + czysta logika`

Bezpośrednio obsługa klawiatury/myszy	→	**rozdział 17 — .bind(...)**

Rozdział w całości 16

MODEL WYDARZENIA

mainloop() jest cyklem, a nie zamrożeniem
callback ≠ command ≠ event
function bez nawiasów

DRZEWO WIDŻETÓW

rodzic określa kontekst
create → configure → umiejscowienie → interaktywność

TK I TTK

ttk — motyw + nowoczesne widgety
styl ttk.Style(), nie fg/bg

UKŁAD

pack — proste regiony
grid + weight — formy adaptacyjne

nie mieszać jednego rodzica

INPUT I STATUS

Entry/Text, end-1c
Tk- zmienne ≠ zastąpić Python- zmienne

DIALOGI I OKNA

messagebox/filedialog — sprawdź wynik
Toplevel, nie drugi Tk()

RESPONSYWNOŚĆ

after() zamiast sleep()
długa praca nie jest w jednym callback

ARCHITEKTURA

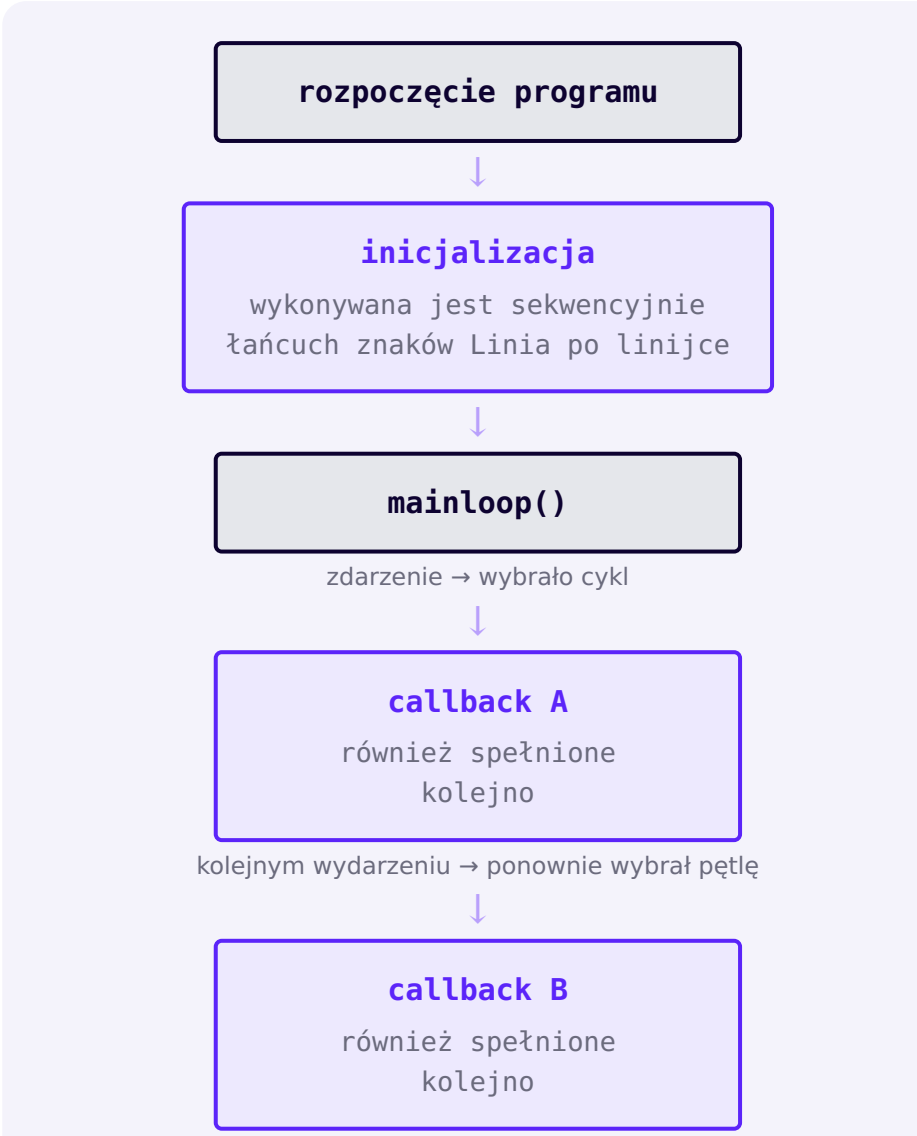
czysta logika oddzielna od widgetów
App HAS-A root
świat — przez rozdział 15 JSON

- Tkinter-app
  - Model wydarzenia
    - mainloop

Pełna mapa rozdziału 16.

Sekwencja w ramach wydarzenia

Precyzyjne sformułowanie jest ważniejsze niż skuteczne sformułowanie: kod inicjalizacyjny nadal działa kolejno łańcuch znaków linia po linii, a każdy pojedynczy callback również jest wykonywany kolejno, gdy zostaje wezwany. To nie to się zmienia, ale **ogólny model zarządzania**: to właśnie pętla zdarzeń decyduje, która callback zostanie wywołana jako następna, w odpowiedzi na zdarzenie — a nie sam kod, idący w jednej ciągłej linii od góry do dołu.



Sekwencja nie posunęła się donikąd — jest po prostu podzielona na kawałki, między którymi decyduje cykl wydarzeń.

co dalej

Teraz możemy budować root i drzewo widżetów, łączyć callback przez `command`, układać formy przez `pack` i adaptacyjny `grid`, przechowuj współdzielony stan w zmiennych Tk-, Pokazuj dialogi, otwieraj/zapisuj pliki, umawiaj opóźnione połączenia przez `after()` i złóż wszystko w klasę aplikacji z trwałością ustawienia.

W rozdziale 17 („Projekt: gra „Kółko-krzyżyk” z Tkinter”) zbudujemy pełną grę — i poznamy bardziej ogólny mechanizm wiązania zdarzeń, `.bind("<Button-1>", ...)`, który reaguje na kliknięcie w dowolnym miejscu widżetu (na przykład na konkretnej komórce płótna), a nie tylko na zdefiniowaną akcję taką jak `command` przycisk.

Czego nauczyliśmy się w tym rozdziale

- Inicjalizacja i każdy callback nadal wykonywane są kolejno — zmienia się ogólny model zarządzania: pętla zdarzeń decyduje, który callback wywołać następny w odpowiedzi na wydarzenia.
- `root.mainloop()` uruchamia pętlę zdarzeń zamiast „zawieszania”
- `command=функция` kojarzy callback z widżetem — bez nawiasów po nazwie funkcji.
- Każdy widget jest tworzony, konfigurowany i umieszczany osobno przez menedżera geometrii — `pack()`, `grid()` lub `place()`, ale bez mieszania pack/grid z jednym rodzicem.
- ttk daje tematyczne widżety i stylizację przez `ttk.Style()` — ale nie zastępuje tk.Text/Canvas/Menu.
- zmienne Tk-zmienne (StringVar i inne) wiążą wiele widżetów z jedną wartością — i nie zastępują zwykłych zmiennych Python.
- messagebox i filedialog zwracają rzeczywiste wartości, które należy sprawdzić (w tym anulowanie), a nie zakładać.
- `after()` planuje opóźnione lub powtarzające się połączenie bez blokowania — `time.sleep()` w callback zawiesza cały interfejs.
- Logikę domeny warto pisać jako czyste funkcje, sprawdzane bez żadnego widgetu — callback tylko czyta dane wejściowe, wywołuje logikę i aktualizuje ekran.
- GUI- wykorzystują te same funkcje JSON+pathlib co Rozdział 15 – nic nowego.

Project: Gra w kółko i krzyżyk z Tkinter

Zbudujmy pierwszą pełnoprawną GUI- grę jako prawdziwy mały projekt programistyczny: przeanalizujemy system zdarzeń Tkinter, oddzielimy stan gry od widżetów, zaimplementujemy zasady i testy, dodamy obsługę myszy i klawiatury, wizualną informację zwrotną, liczenie meczów i końcową architekturę aplikacji.

Event Binding – Dynamiczne tworzenie aplikacji!

command nie jest jedynym sposobem reagowania na działania użytkownika: bind() daje dostęp do wszelkiego rodzaju wydarzeń.

W rozdziale 16 przyciski reagowały na kliknięcie poprzez parametr `command` jest najczęstszym, ale nie jedynym sposobem powiązania kodu z akcją użytkownika. **Wiązanie zdarzeń** (`.bind()`) pozwala reagować na cokolwiek: naciśnięcie klawisza, ruch myszy, najechanie kursorem.

privyazka_sobytij.py

```
import tkinter as tk

root = tk.Tk()
label = tk.Label(root, text=„Naciśnij dowolny klawisz”,
font=("Arial", 14))
label.pack(padx=20, pady=20)

def na_klavishu(event):
    label.config(text=f"Kliknąłeś: {event.keysym}")

root.bind("<Key>", na_klavishu)
root.mainloop()
```

Gdy callback zarejestrowana przez `.bind()` , Tkinter przekazuje obiekt do niego `Event` szczegóły dotyczące wydarzenia: który klawisz jest naciśnięty (`event.keysym`), gdzie klikała mysz i tak dalej dalej. Ta zasada dotyczy `.bind()` : callback przekazany do `command=` (jak w rozdziale 16) zwykle nazywa się bez Event dopełnienia — Rozdział 17.8 szczegółowo wyjaśni różnicę.

Przyda się w rozdziale 19

Ta sama idea — powiązać klawisz z funkcją — powróci w rozdziale 19: wąż będzie reagować na naciśnięcia klawiszy ze strzałkami. W Turtle do tego jest własny, prostszy interfejs — `screen.listen()` oraz `screen.onkeypress()` , — ale zasada jest ta sama.

Praktyka: wiązanie zdarzeń klawiatury
Moduł tkinter otwiera natywne okno Python — wykonaj lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/17-01/index.html)

Wyjaśnienie gry kółko i krzyżyk

Przyjrzyjmy się sześciostopniowemu planowi, którego wykorzystamy do budowy gry — od pustego okna po gotowy program.

Wyjaśnienie gry kółko i krzyżyk

Zasady są znane niemal każdemu: pole 3×3, dwóch graczy na zmianę stawia "X" i "O", pierwsze, Ten, kto ustawi trzy swoje symbole w rzędzie – poziomo, pionowo lub ukośnie – wygrywa. Budujmy tę grę krok po kroku, tak jak zrobiłby ją deweloper, a nie jednym kawałkiem kodu Od razu:

1. Konfiguruj okno;
2. Tworzenie zmiennych stanu gry;
3. Utworzyć pole z dziewięcioma przyciskami;
4. Reaguj na kliknięcie — rysuj X lub O;
5. Sprawdzam po każdym ruchu, czy jest zwycięzca;
6. Dodaj przycisk „Nowa gra”

Puste okno z polem dziewięciu przycisków i statusem ruchu gracza: X

patrząc w przyszłość, tak właśnie będzie wyglądać okno pod koniec sekcji 17.6, po wszystkich sześciu krokach. Wtedy ta sama strona nie wykonuje jeszcze całego kodu, tylko konfiguruje okno (krok 1).

Konfigurujemy Tkinter

nastrojka_okna.py

```
import tkinter as tk

root = tk.Tk()
root.title(„Kółko-krzyżyk”)
```

Praktyka: Zaczynam budować grę
Moduł tkinter otwiera natywne okno Python — wykonaj lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/17-03/index.html)

Tworzenie zmiennych globalnych

gra potrzebuje pamięci między ruchami — ustaw zmienne stanu i zbudujmy pole przycisków w pętli.

Tworzenie zmiennych globalnych

grze musisz pamiętać o swoim stanie między ruchami – czyja jest kolej, czy gra się skończyła i Same przyciski pola zmieniają tekst:

globalnye_peremennye.py

```
tekuschij_igrok = "X"
polya = []              # lista z 9 przycisków pola
igra_okonchena = False
```

Tworzenie przycisków

Pole 3×3 to dziewięć przycisków rozmieszczonych przez `grid()` (rozdział 16). Stworzymy je pętlą, a nie dziewięcioma identycznymi łańcuch znaków kawałkami kodu ręcznie (rozdział 10):

sozdaem_knopki.py

```
pole_frame = tk.Frame(root)
pole_frame.grid(row=1, column=0, columnspan=3)

for indeks in range(9):
    knopka = tk.Button(
        pole_frame,
        text="",
        font=("Arial", 24, "bold"),
        width=3, height=1,
    )
    knopka.grid(row=indeks // 3, column=indeks % 3)
    polya.append(knopka)
```

indeks // 3 i indeks % 3 — współrzędne z problemu
Liczby od 0 do 8 powinny być przekształcone w wiersz i kolumnę w Tabeli 3×3. Dzielenie liczb całkowitoliczbowych `//` (rozdział 5) podaje numer linii (`0,0,0,1,1,1,2,2,2`), oraz pozostałe `%` to liczba kolumn (`0,1,2,0,1,2,0,1,2`).

Dziewięć pustych przycisków w siatce 3×3 i status Ruch gracza: X

Rzeczywiste okno pełnego pliku prototypu: dziewięć pustych przycisków rozłożonych tym cyklem.

Praktyka: zmienne stanowe i pole z przycisków
Moduł tkinter otwiera natywne okno Python — wykonaj lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/17-03/index.html)

Podczas naciśnięcia przycisku rysujemy na nim

Każdy przycisk powinien znać swój numer – jedna z najsłynniejszych subtelności Python. jest tu ukryta

Każdy przycisk potrzebuje własnej funkcji obsługi klików, która wie o tym *jaka dokładnie* przycisk jest naciskany. Sztuczka polega na podłączeniu `command` z numerem przycisku — do tego używamy funkcji lambda z rozdziału 13:

risuem_na_knopke.py

```
def na_knopku_nazhali(indeks):
    global tekuschij_igrok

    if polya[indeks]["text"] != "":
        return   # komórka już zajęta

    polya[indeks]["text"] = tekuschij_igrok
    tekuschij_igrok = "O" if tekuschij_igrok == "X" else "X"

# przy tworzeniu każdego przycisku:
knopka.config(command=lambda i=indeks: na_knopku_nazhali(i))
```

Dlaczego lambda i=indeks a nie tylko lambda: na_knopku_nazhali(indeks)?

Bez `i=indeks` wszystkie dziewięć przycisków pamiętało tę samą zmienną `indeks` — nie jego wartości w momencie tworzenia. W momencie kliknięcia cykl już dawno się zakończy, i `indeks` będzie 8 dla wszystkich przycisków jednocześnie. Domyślnie `i=indeks` „zamraża" aktualną wartość przy tworzeniu każdej lambdy — klasyczny i ważny niuans Python.

na przycisku pola 0 pojawił się X, status zmieniał się na Ruch Gracza: O

Rzeczywiste okno pełnego pliku prototypu: kliknięcie na pierwszą komórkę — na niej narysowano X, kolej przeszła do O.

Praktyka: Przetwarzanie kliknięcia na komórce pola

Moduł tkinter otwiera natywne okno Python — wykonaj lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/17-04/index.html)

Sprawdzaj po każdym ruchu, czy gracz wygrał

Osiem możliwych zwycięskich linii — trzy wiersze, trzy kolumny, dwie przekątne.

Po każdym ruchu musisz sprawdzić, czy w rzędzie są trzy identyczne symbole. W grze dostępnych jest łącznie osiem możliwych „linii wypłat"

proverka_pobedy.py

```
def proverit_pobedu():
    linii_pobedy = [
        (0, 1, 2), (3, 4, 5), (6, 7, 8),  # Linie
        (0, 3, 6), (1, 4, 7), (2, 5, 8),  # kolumny
        (0, 4, 8), (2, 4, 6),             # przekątne
    ]
    for a, b, c in linii_pobedy:
        znacheniya = (polya[a]["text"], polya[b]["text"],
polya[c]["text"])
        if znacheniya[0] != "" and znacheniya[0] ==
znacheniya[1] == znacheniya[2]:
            return znacheniya[0]   # 'X' lub 'O' – jest
zwycięzca
    return None   # jeszcze nie ma zwycięzcy
```

Lista krotek – Zwięzły sposób opisu 8 linii
Każda linia to krotka (rozdział 11) trzech indeksów pól. Lista ośmiu takich krotk zastępuje osiem oddzielnych kontrol `if` to musiało być napisane ręcznie.

Górny łańcuch znaków jest zajęty X, status mówi: Gracz wygrał X!

Okno plików prawdziwego prototypu: X zebrał najwyższą linię – proverit_pobedu() znalazł zwycięzcę.

Remis

Jeśli wszystkie dziewięć pól zostanie zapełnione i nie ma zwycięzcy, jest remis:

proverka_nichyej.py

```
def pole_zapolneno():
    return all(knopka["text"] != "" for knopka in polya)
```

Pole całkowicie wypełnione X i O bez zwycięskiej linii, status pokazuje Remis!

Rzeczywiste okno pełnego pliku prototypu: wszystkie dziewięć pól zajęte, ale nie ma zwycięskiej linii.

Praktyka: Sprawdzanie zwycięstwa i remisu
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/17-05/index.html)

Nowa gra i pierwsza pełna wersja

Finalny akcent pierwszego prototypu — reset gry bez ponownego uruchomienia programu. Ale rozdział dopiero się zaczyna.

Przycisk Nowej Gry

Aby nie uruchamiać programu od nowa po każdej grze, dodajemy przycisk resetu — przywraca wszystkie zmienne i wszystkie komórki do stanu początkowego:

novaya_igra.py

```
def novaya_igra():
    global tekuschij_igrok, igra_okonchena
    tekuschij_igrok = "X"
    igra_okonchena = False
    for knopka in polya:
        knopka.config(text="")
    status_label.config(text=f"tura gracza: {tekuschij_igrok}")
```

Pole znów jest puste, status Ruch gracza: X, po wygraniu X w poprzedniej grze

Pełne okno plików Real Prototype: „Nowa gra”

Pełny program

Zbioremy wszystkie części w jednym programie – został on już w pełni przetestowany i jest dostępny osobno Umieść w tej książce:

projects/tkinter/tic-tac-toe/tic_tac_toe_basic.py

Uruchom grę u siebie
Pobierz plik i uruchom go przez `python tic_tac_toe_basic.py` w terminalu (Rozdział 3) – albo przez Run w VS Code, albo PyCharm.

To PIERWSZY działający prototyp, a nie koniec rozdziału
Zbudowaliśmy prawdziwy działający program — z ruchami, zwycięstwem, remisem i resetem. To uczciwa, pełna, mała gra. Ale począwszy od następnego rozdziału będziemy przyglądać się jej dokładniej: skąd callback wie, że w ogóle ktoś go wywołał; dlaczego przechowywanie stanu w tekście przycisku nie jest najlepszym pomysłem na długo; jak zrobić interfejs, który reaguje na mysz i klawiaturę tym samym kodem. Pod koniec rozdziału (rozdział 17.31) ta sama gra stanie się `tic_tac_toe.py` — z osobnym modelem stanowym, oświetleniem linii zwycięzców, wynikami meczów i pełnymi testami.

★★ Zadanie samodzielne
Win Score
Dodaj etykiety wyniku zwycięstw X i O, zwiększaj odpowiedni licznik po każdej wygranej — licznik nie powinien zerować się przy przycisku „Nowa gra”.

Challenge
Wyróżnianie linii zwycięskiej
Gdy zostanie znaleziony zwycięzca, zmień kolor tła trzech zwycięskich przycisków — trzeba zwrócić z proverit_pobedu() nie tylko zwycięzcę, ale także indeksy linii.

Praktyka: Nowa gra i pierwsza pełna wersja
Moduł tkinter otwiera natywne okno Python — wykonaj lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/17-06/index.html)

Streszczenie sekcji

Czego się nauczyliśmy

- `.bind()` reaguje na dowolne zdarzenia — nie tylko kliknięcie przycisku.
- Duże projekty są budowane w etapach: interfejs → stan → obsługa zdarzeń → reguły → resetują.
- Lambda z domyślnym ustawieniem (`lambda i=indeks: ...`) „zamraża”
- Lista krotek to wygodny sposób na opisanie kilku podobnych kontrol (osiem linii wypłat) bez powtarzania kodu.
- Pełnoprawna gra prawie zawsze ma sposób, aby rozpocząć od nowa, bez ponownego uruchamiania programu.
- To dopiero początek rozdziału, ponieważ przyjrzymy się bliżej systemowi wydarzeń Tkinter i odbudujemy grę na bardziej rozbudowanej architekturze.

Event, callback, command i binding

„Tkinter przyciski to gra”

Kto wywołuje on_click()?

Porównajmy dwa znane sposoby uzyskania akcji od użytkownika.

posledovatelno.py

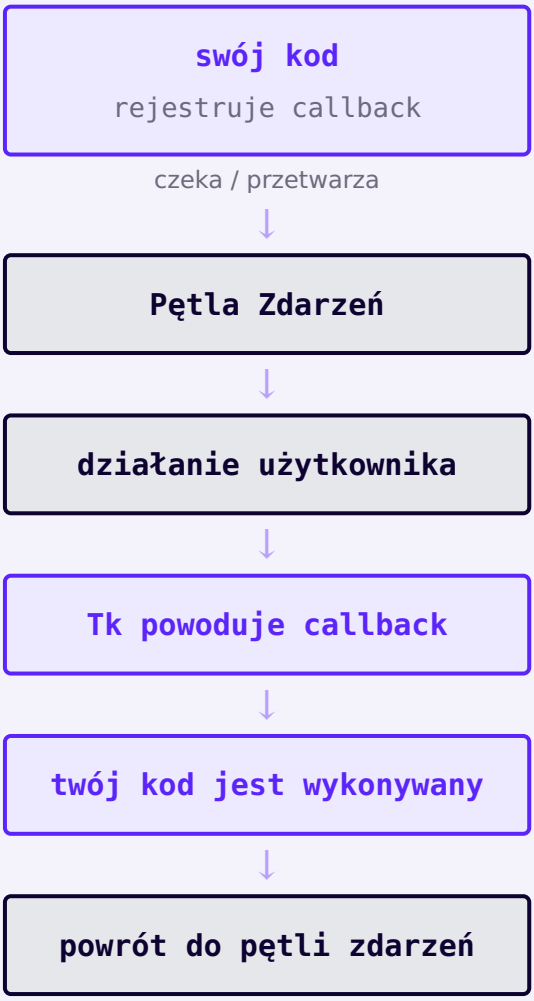
```
imya = input(„Jak masz na imię?”)
print(f"Cześć, {imya}!")
```

Wszystko jest tu proste: sam program, w dobrze znanym wierszu kodu, pyta i czeka na odpowiedź.

gui.py

```
def on_click():
    print(„Przycisk naciśnięty”)

button = ttk.Button(root, text="OK", command=on_click)
```

Kto dzwoni `on_click()` ? Nie my — nie w kodzie Smyczki `on_click()` . Funkcję można wykonać w sekundę, za godzinę lub nigdy, w zależności od tego, czy użytkownik naciśnie przycisk.



Callback — kod, którego wywołanie jest odkładane do momentu wystąpienia zdarzenia.

Odpowiedź
Callback przyczyny **Tk system wydarzeń** — w momencie, gdy następuje odpowiadająca mu akcja użytkownika (lub inne zdarzenie). Nie Twój kod, nie interpreter Python sam w sobie.

Event, callback, command binding to cztery różne słowa

Termin	Co to jest
EVENT (wydarzenie)	Fakt: coś się wydarzyło — klik, naciśnięcie klawisza, najechanie kursorem
CALLBACK (nawiązanie do powrotu)	Funkcja wywołana W ODPOWIEDZI na zdarzenie
COMMAND (drużyna)	Wysokopoziomowy hak dla konkretnego widżetu dla jego głównej akcji (np. kliknięcie Button)
BINDING (wiązanie)	Powiązanie między zdarzeniem (sekwencją) a callback-funkcją

Są ze sobą powiązane, ale nie wymienne
Wydarzenie to TO, co się wydarzyło. Callback to funkcja, która ZAREAGUJE. Command to przypadek szczególny, specyficzny dla konkretnego widżetu. Binding to mechanizm, który łączy wydarzenie z callbackiem. Mieszanie tych słów jest źródłem zamieszania w wyjaśnianiu własnego kodu.

Praktyka: klasyfikacja event/callback/command/binding
Automatyczna walidacja — kategoryzacja przykładów krótkich kodów według tych czterech terminów

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/17-07/index.html)

Obiekt Event

Callback z bind() otrzymuje Event; callback z command= zwykle jest wywoływany bez niego. Przyjrzyjmy się, co znajduje się wewnątrz tego obiektu.

Nie każdy callback dostaje event

Powszechnym błędnym przekonaniem jest to, że „obsługa zdarzeń zawsze przyjmuje event”

command= kontra bind()

command= — BEZ event

```
def new_game():
    ...

ttk.Button(root, text=„Nowa Gra”, command=new_game)
```

bind() — S event

```
def on_key(event):
    ...

root.bind("<KeyPress>", on_key)
```

Co używać dzisiaj: `command=` wywołuje funkcję BEZ argumentów. `.bind(sequence, callback)` wywołuje funkcję Z JEDNYM argumentem — obiektem `Event`. Pomyłka w podpisie to częsta przyczyna `TypeError`.

Przedmiot Event - Co jest w środku

event : Event

```
widget = widżet-źródło
type = typ wydarzenia
keysym = 'Return', 'a', ...
char = symbol tekstowy
keycode = kod numeryczny
x = współrzędne X wewnątrz widgetu
y = Y współrzędnych wewnątrz widżetu
```

Nie każde pole ma sens dla każdego wydarzenia – więcej szczegółów poniżej.

event.x nie ma sensu wciskać klawisza

`Event` jest jedną klasą dla wszystkich typów zdarzeń, więc ma wiele pól jednocześnie. Ale `event.x` / `event.y` są istotne dla zdarzeń myszy, nie dla klawiatur. `event.keysym` / `event.char` jest dla klawiatury, nie dla przesuwania myszką. Nie czytaj tego pola, chyba że jesteś pewien, że dotyczy tego typu zdarzenia.

event.widget

event_widget.py

```
def on_enter(event):
    print(event.widget)   # który dokładnie widget otrzymał
zdarzenie
```

`event.widget` jest widżetem, do którego należy to zdarzenie. To jest źródłem zdarzeń UI-, a nie stanem gry (Rozdział 17.18 to istotna różnica).

Praktyka: inspektor obiektu Event

Moduł tkinter otwiera natywne okno Python — wykonaj lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/17-08/index.html)

Składnia zdarzeń Tk

<modyfikator-typ-szczegół> — nie zawsze wszystkie trzy części są potrzebne jednocześnie.

Anatomia ciągu zdarzeń

Linia jak `"<Control-Button-1>"` — to nie magiczne zaklęcie, a struktura `<модификатор-тип-деталь>`, gdzie Niektóre elementy mogą być pominięte:

Nagrania	Co oznacza
`<KeyPress>`	nacisnąć dowolny klawisz (tak samo jak `<Key>`)
`<Return>`	konkretny kluczowy Enter

Nagrania	Co oznacza
`<Escape>`	konkretne Escape
`<Button-1>`	lewym przyciskiem myszy
`<Enter>`	kursor wszedł w granice widgetu
`<Leave>`	kursor opuścił granice widgetu
`<Control-r>`	Control modyfikator + klawisz r

Nie musisz zapamiętywać całej gramatyki Tk całego materiału
Tk dziesiątki obsługiwanych sekwencji zdarzeń. Dla rozdziału 17 wystarczy ten mały zestaw — pozostałe w razie potrzeby są wyszukiwane w oficjalnej dokumentacji Tcl/Tk i nie są uczeni na pamięć.

Button-1 — lewy przycisk myszy

button_1_demo.py

```
def on_raw_click(event):
    print(„Kliknięcie myszką w współrzędnych”, event.x,
event.y)

label.bind("<Button-1>", on_raw_click)
```

To nie jest główny sposób aktywacji komórek w grze
`<Button-1>` przydatne do demonstracji niskopoziomowego zdarzenia myszy — ale nadal użyjemy `command=` w Button (sekcja 17.10). Tutaj jest to po prostu ilustracja składni.

Praktyka: Składnia zdarzeń
Automatyczne sprawdzanie — dopasowujemy opis akcji do poprawnego łańcuch znaków łańcuch znaków zdarzeń
Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/17-09/index.html)

command vs bind — co wybrać

Nie „bind istnieje, więc używamy go wszędzie”

command= lub bind() — jak wybierać

Co wybrać?

Musisz wykonać główną akcję Button	→ `command=`
Potrzebuję współrzędnych myszy	→ `bind("<Motion>", ...)`
Trzeba wiedzieć, który dokładnie klawisz został naciśnięty	→ `bind("<Key>", ...)`
Potrzebna jest reakcja na najechanie kursorem (bez kliknięcia)	→ `bind("<Enter>"/"<Leave>", ...)`
Widżet nie dostarcza żadnych command (np. Label, Frame)	→ `bind()`

Domyślnie nie zastępuj Button.command bind(<Button-1>'', ...)
To jest zasada zawodowa. `command=` Button jest aktywacją semantyczną: opisuje główną akcję przycisku, a wbudowane zachowanie klasy Button decyduje, które metody aktywacji wywołują polecenie. Konkretne klucze mogą się różnić od jednego `tk.Button` oraz `ttk.Button`, a także między platformami i motywami. Gołe snapowanie `<Button-1>` opisuje tylko konkretne zdarzenie myszy i omija działanie semantyczne widżetu. Zastąpienie command bind „tylko dlatego, że możesz”

Jak będzie to rozmieszczone w grze

Trzy zadania — trzy różne narzędzia

RUCH PO POLU

```
command= w Button
główna akcja widgetu
```

HOVER - ZAPOWIEDŹ

```
bind(<Enter>/<Leave>)
nie ma gotowego command na to
```

KLAWISZE 1–9 / R

```
bind(<Key>) na root
nie jest przypisane do jednego widgetu
```

Praktyka: command vs bind – co wybrać
Automatyczne sprawdzanie – wybierz odpowiednie narzędzie do zestawu scenariuszy

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/17-10/index.html)

Focus i klawiatura

Istnieje okno, nie znaczy, że dostaje wszystkie naciśnięcia klawiszy. Potrzebujesz skupienia.

Wydarzenia klawiaturowe wymagają miejsca docelowego

Okno z przypisaniem klawiszy nie oznacza, że naciśnięcie DOWOLNEGO klawisza gdziekolwiek będzie należeć do tego handlera. Każde zdarzenie klawiatury najpierw trafia do widżetu, który Teraz **skupienie inputu** — a potem Tk sprawdza wiązania łańcucha *bindtags*: Sam widget → swoją klasę → okno najwyższego poziomu (root) → „all”

focus_demo.py

```
print(root.focus_get())   # który widżet jest teraz w fokusu (lub None)

entry.focus_set()         # jawnie przekazać fokus temu widżetowi
```

"Binding doesn't works" często oznacza "klasa widget przechwyciła zdarzenie jako pierwsza"
root.bind(<Key>„ `Entry` jako pierwszy przetwarza naciśnięcie liczby (wprowadza symbol do pola); wiąże się z `root` w tym przypadku zwykle NADAL wystrzeluje następny, chyba że handler wyraźnie wrócił `"break"` przez przerwanie łańcucha. Oznacza to, że symbol pojawi się w polu wejściowym, a jednocześnie trafi do logiki gry — rzadko co to, czego chciałeś. Zobacz Debug Lab 1 (sekcja 17.29).

W rozdziale 17 przypisania klawiatury są zawieszone `root` — ale, jak właśnie pokazano, skupienie na Entry lub Text samo w sobie NIE wyłącza powiązania z root (w naszej grze takich pól nie ma — wszystkie komórki to przyciski). W aplikacji z polami tekstowymi

ważne jest, aby nie „utrzymywać fokusu na root”, lecz świadomie zdecydować: czy skróty klawiszowe w grze powinny działać, gdy użytkownik wpisuje tekst? Jeśli nie — handler może sprawdzić `focus_get()` sam ignoruj prasę, gdy jest to skupienie w polu tekstowym.

Praktyka: focus_get() i focus_set()
Moduł tkinter otwiera natywne okno Python — wykonaj lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/17-11/index.html)

Mouse Enter/Leave i hover

Najazd kursorem — zdarzenie samo w sobie, bez kliknięcia. Idealne dla wizualnych podpowiedzi.

Enter / Leave – idealne do efektów wizualnych

Hover to nie kliknięcie: Żadne z tych zdarzeń nie powinno zmieniać stanu gry.

hover_bind.py

```
button.bind(
    "<Enter>",
    lambda _event, i=index: self.on_cell_enter(i),
)
```

_event jest celowo nieużywaną nazwą
Callback od `.bind()` jest zobowiązany zaakceptować ten argument `event`, nawet jeśli nie jest potrzebna w środku. Imię `_event` (podkreślone) to powszechny sygnał „parametr istnieje na żądanie sygnatury, ale nie jest świadomie używany”

Binding scope — gdzie więź jest ważna

Challenge	Zakres
`widget.bind(...)`	tylko dla tego konkretnego widgetu
`root.bind(...)`	dla głównego okna (np. globalne klawisze gry)

TROCHĘ GŁĘBIEJ — bind_class / bind_all
Tk również wspiera `bind_class(...)` (wszystkie widżety tej samej klasy) oraz `bind_all(...)` (dosłownie cała aplikacja). W rozdziale 17 nie są potrzebne — są wspomniane, żebyś wiedział, że istnieją, jeśli będziesz ich potrzebować w swoich projektach.

NIECO GŁĘBIEJ — add=«+» i unbind()

Domyślnie `bind(sequence, callback)` ZASTĘPUJE poprzedni uchwyt tej samej sekwencji na tym widżecie. Opcjonalny trzeci argument `add="+"` zamiast zamiany dodaje JESZCZE JEDEN uchwyt — oba zostaną wywołane. `widget.unbind(sequence)` całkowicie usuwa kotwicę. Dla rozdziału 17 nie są potrzebne — ale Debug Lab 17 (Rozdział 17.29) pokazuje, dlaczego nieostrożność `add="+"` może uruchomić ten sam kod dwa razy, nie zauważając tego.

Praktyka: hover aż do Enter/Leave

Moduł tkinter otwiera natywne okno Python — wykonaj lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/17-12/index.html)

Model stanu gry

Tkinter pokazuje grę i przekazuje dane wejściowe użytkownika. STAN GRY to osobna rzecz.

Co w ogóle powinna pamiętać program?

state : GameState

```
board = ['', '', '', '', '', '', '', '', '']
current_player = 'X'
game_over = False
winner = None
winning_line = None
score_x = 0
score_o = 0
draws = 0
```

To dane — ani jeden przycisk Tkinter tutaj.

Widgety – NIE jest kanonicznym stanem gry

Przyciski na ekranie WYŚWIETLAJĄ stan – nie są stanem – jeśli najechanie myszką tymczasowo narysuje „X”

Prototyp: state w zmiennych globalnych

Rozdział 17.3 zastosowała dokładnie takie podejście:

prototip_globals.py

```
tekuschij_igrok = "X"
polya = []
igra_okonchena = False
```

Prototyp: najpierw sprawiamy, że stan jest widoczny

Zmienne globalne — akceptowalny wybór dla pierwszego małego prototypu: łatwo je czytać i łatwo wyjaśnić. Problem pojawia się, gdy projekt rośnie — zmienne globalne tworzą ukryte powiązania między funkcjami, które niejawnie oczekują, że ktoś inny już je zmienił we właściwej kolejności. To nie znaczy „zmienne globalne są złe”: to oznacza, że mają kompromis między prostotą a utrzymaniem, na który warto zwrócić uwagę.

Ścieżka tego rozdziału

V3 — GAMESTATE (17.19)

```
@dataclass
maszynisty, czytelny, testowalny
```

Jeden stan — różne widoczne momenty gry

Puste pole i status Ruch gracza: X

Rzeczywisty stan kanonicznej aplikacji: początek rundy, ruch X.

Po pierwszej turze status X zmienił się na Turę Gracza: O

Rzeczywisty stan tej samej aplikacji po poprawnym ruchu X: modelu zmienił current_player na O.

Pięć oznaczeń X i O na planszy, partia trwa dalej

Rzeczywisty stan pośredni w aplikacji kanonicznej: board zapisuje status_var pokazuje aktualnego gracza.

Praktyka: Symulacja stanu gry
Automatyczna kontrola – buduj i sprawdzaj słownik stan bez Tkinter
Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/17-13/index.html)

Indeksy, wiersze i kolumny

Pole wygląda jak tabela 3×3 — ale w pamięci jest jednym lista z dziewięciu elementów.

Ciało to płaska lista dziewięciu elementów

Pole 0..8 indeksów komórek jest od lewej do prawej, od góry do dołu.

Chociaż wizualnie pole to Tabela 3×3, w pamięci jest to JEDEN płaski lista na dziewięć Elementy (`board[0]` .. `board[8]`), Zgodnie z artykułem 17.3.

→ łańcuch znaków Indeks i kolumna

indeks_v_koordinaty.py

```
row = index // 3
column = index % 3
```

index	row	column
0	0	0
1	0	1
2	0	2
3	1	0
4	1	1
5	1	2

index	row	column
6	2	0
7	2	1
8	2	2

Rewers: łańcuch znaków i indeks kolumny →

koordinaty_v_indeks.py

```
index = row * 3 + column
```

Przyda się nie tylko tutaj

Wzór `row * ширина + column` jest ogólną techniką dla dowolnej prostokątnej siatki przechowywanej na płaskiej liście (obrazy, mapy poziomów, arkusze kalkulacyjne).

Praktyka: indeks, łańcuch znaków, kolumna

Automatyczna kontrola – konwersja współrzędnych do przodu i do tyłu dla wszystkich 9 komórek

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/17-14/index.html)

Poprawny algorytm ruchu

Jeden ze sposobów wykonania ruchu – szach, ruch, zasady, zmiana gracza.

Jaki jest ważny ruch?

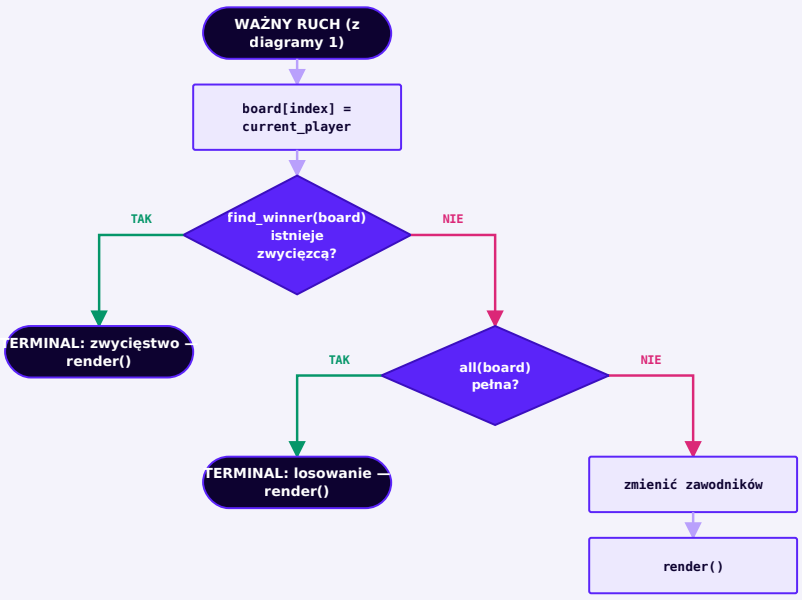
`attempt_move(index)` - Jedna ścieżka dla myszy i klawiatury (Rozdział 17.24). Przeanalizujmy to w dwóch etapach: najpierw filtr ważności, potem co się dzieje po tym, jak przeprowadzka już została podjęta.

Diagram 1 – Filtr Poprawności Tury

Oba „nie” muszą zostać wykonane, aby ruch dotarł do diagramu 2.

Nieprawidłowy ruch NIE zmienia gracza
To częsty błąd (Debug Lab 7, sekcja 17.29): jeśli kliknięcie zostanie zignorowane, tura musi pozostać taka sama. Gracz zmienia tylko PO udanym ruchu.

Diagram 2 – Co się dzieje po prawidłowym ruchu



Zobowierz się do modelu, sprawdź zasady (Rozdział 17.17) i dopiero wtedy zmień gracza.

Widoczny rezultat poprawnego ruchu

Puste pole przed pierwszym ruchem

Prawdziwe okno kanonicznej aplikacji przed ruchem: model board zawiera dziewięć pustych linii.

X zajął centralną klatkę, zwrot poszedł do O

To samo rzeczywiste zastosowanie po attempt_move(4): zapisanie znaku do modelu, a potem render() zaktualizowane pole i status.

Przełączanie gracza — trzy równoważne sposoby

smena_igroka_razvernuto.py

```
if current_player == "X":
    current_player = "O"
else:
    current_player = "X"
```

smena_igroka_kratko.py

```
current_player = "O" if current_player == "X" else "X"
```

Najpierw czytelność, potem zwięzłość

Obie wersje robią to samo. Rozszerzona wersja wyraźnie brzmi "if/otherwise" — łatwiej zacząć od niej. Krótka wersja (warunkowa, rozdział 9; często nazywana operatorem ternarnym) nie jest "mądrzejsza", po prostu bardziej zwarta, gdy czytasz ją już pewnie.

Praktyka: logika jednego ruchu

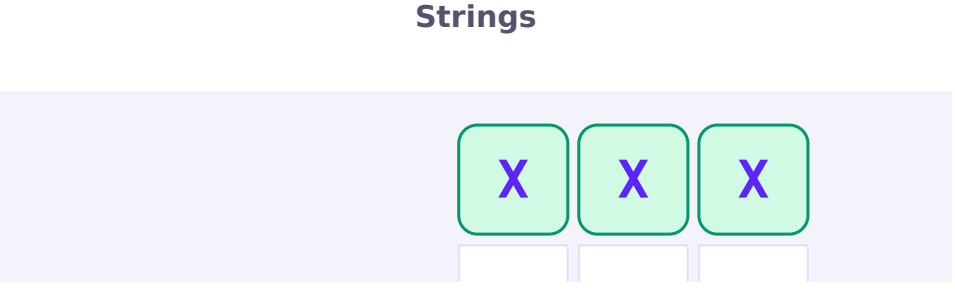
Automatyczny test – walidacja tur i przełączanie zawodnika na czyste funkcje

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/17-15/index.html)

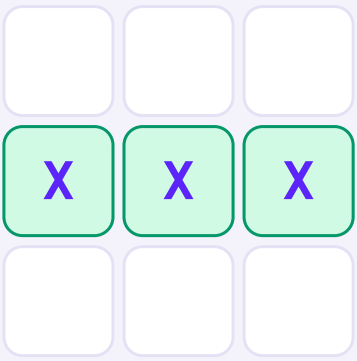
Osiem linii płatnych

Trzy wiersze, trzy kolumny, dwie przekątne i jedna czysta funkcja, która sprawdza je wszystkie.

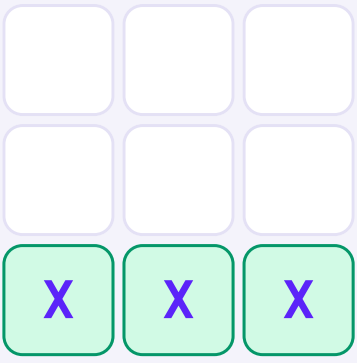
Wszystkie osiem linii płatnych



Linia 1

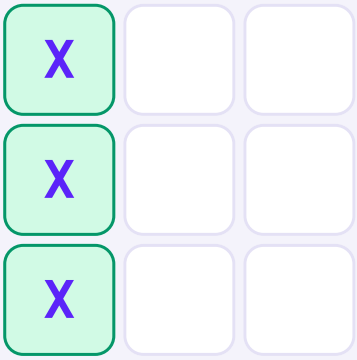


Linia 2

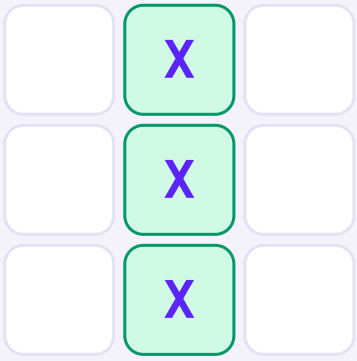


Linia 3

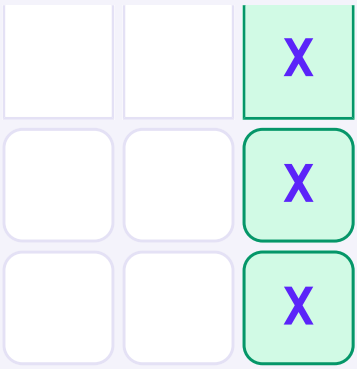
Kolumny



Kolumna 1

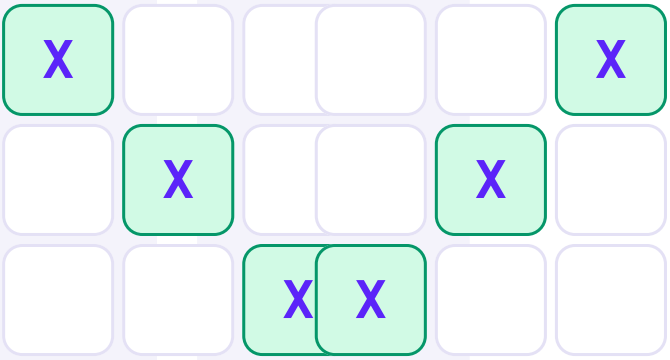


Kolumna 2



Kolumna 3

Diagonale



Diagonala 0-4-8

Przekątna 2-4-6

Ciągłe WINNING_LINES

winning_lines.py

```
WINNING_LINES = (
    (0, 1, 2), (3, 4, 5), (6, 7, 8),  # Linie
    (0, 3, 6), (1, 4, 7), (2, 5, 8),  # kolumny
    (0, 4, 8), (2, 4, 6),             # przekątne
)
```

WIELKĄ LITERĄ to dane reguły, a nie algorytm

`WINNING_LINES` jest zapisane wielkimi literami: tak nazywa się stałe modułu w Python (umowa PEP 8 to umowa między programistami, a nie zakaz zmiany przez interpretera). Są to dane dotyczące zasad gry, oddzielne od kodu, który je używa. `find_winner()` sam w sobie nie wie o rozmiarze 3×3 — po prostu przegląda gotowe linie. Ale stała nie rozwiązuje wszystkiego: arytmetyka indeksów (`// 3` , `% 3`), długość `[""] * 9` a sama siatka 3×3 widżetów też jest sztywno ustalona względem rozmiaru pola — rzeczywiste wsparcie innego rozmiaru wymagałoby zmiany całego tego układu, a nie tylko `WINNING_LINES` .

find_winner() jest czystą funkcją

find_winner.py

```
def find_winner(board):
    for a, b, c in WINNING_LINES:
        mark = board[a]
        if mark and mark == board[b] == board[c]:
            return mark, (a, b, c)
    return None, None
```

Returns Zarówno zwycięzca, jak i sama linia

Rozdział 17.5 zwracał tylko zwycięzcę. `find_winner()` odwzajemnia parę `(winner, winning_line)` — indeksy linii są potrzebne, aby ją podkreślić (sekcja 17.25) bez konieczności ponownego obliczania zwycięzcy.

Nic nie wie o Tkinter

`find_winner(board)` stosuje zwykłe lista — bez guzików, bez `root` . To właśnie sprawia, że cecha jest testowalna bez żadnego otwartego okna (Rozdział 17.28).

Praktyka: find_winner dla wszystkich ośmiu linijek

Automatyczny sprawdzenie – find_winner na każdej z ośmiu linii dla X i O

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/17-16/index.html)

Wygrać, zremisować i terminal state

Wypełnione pole nie zawsze kończy się remisem. Kolejność szachów jest częścią zasad gry.

is_draw() jest czystą funkcją

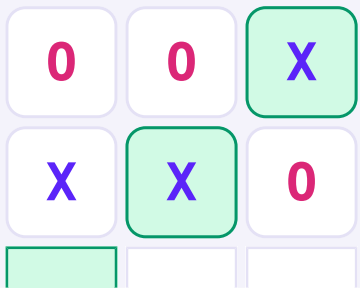
is_draw.py

```
def is_draw(board):
    winner, _ = find_winner(board)
    return winner is None and all(board)
```

„Po pełnym nastroju”

Samo wypełnione pole nie gwarantuje remisu – jeśli ostatni ruch wygrywa linię w tym samym czasie, jest to WYGRANA, a nie remis. Kolejność czeku decyduje o wszystkim.

Ostatni ruch: Wygrana pokonuje remis



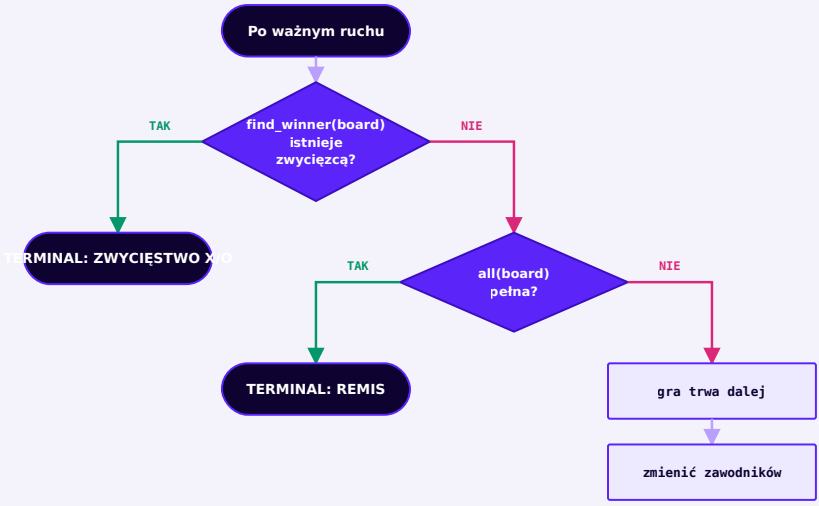
ruch 9 jednocześnie wypełnia pole I wygrywa przekątną 2-4-6.

Obowiązkowe przewidywanie ćwiczenie

Przed przeczytaniem kodu – przewidywać: jeśli dziewiąty ruch wypełni OSTATNIĄ pustą komórkę I zbuduje linię, co pokaże program? "Draw" czy "Win X"? Poprawna odpowiedź to "Win", jeśli `find_winner()` jest sprawdzane WCZEŚNIEJ `is_draw()`. Właśnie dlatego `is_draw` najpierw woła w sobie `find_winner` i dopiero potem patrzy na wypełnienie.

Terminal state

WYGRANA i REMISA to prawdziwe ślepe zaułki: gdy tylko gra tam dochodzi, droga powrotna do „Gra trwa dalej”



WYGRANA i REMISA to stany końcowe: gra się kończy i nie ma już ruchów.

Trzy terminale dają rzeczywiste okno

X wygrał górny wiersz

Rzeczywisty stan pełnej wersji wygląda następująco: X zwycięstwa, podświetlona linia zwycięska, pole jest zablokowane.

O wygrał środkową kolumnę

Prawdziwy stan pełnej wersji: ta sama logika find_winner() działa dla O.

Pełne pole bez linii zwycięskiej, status Losowanie

Rzeczywisty stan pełnej wersji: pole jest wypełnione, nie ma zwycięzcy, dodatkowe ruchy są blokowane.

Praktyka: Wygrana kontra remisy – procedura weryfikacji
Automatyczna weryfikacja — w tym test „ostatni ruch wygrywa, a nie kończy się remisem”

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/17-17/index.html)

Od widgets-as-state do board model

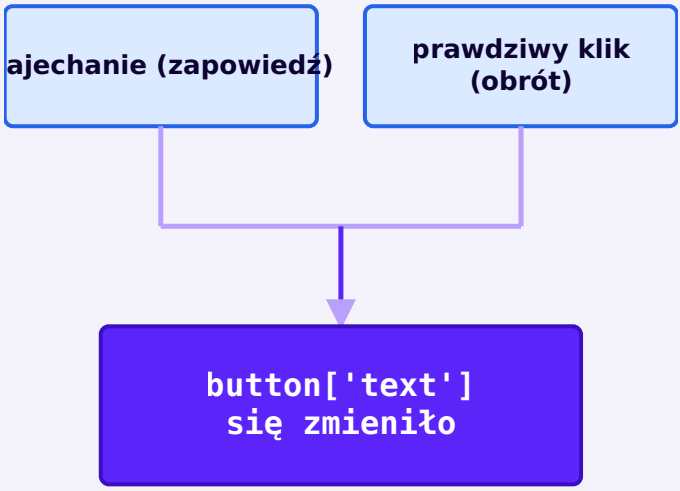
Tymczasowo narysuje X na pustym przycisku. Jeśli jest to „stan”

Rozdział 17.3 przechowywała stan W przycisku TEXT

widget_kak_istochnik.py

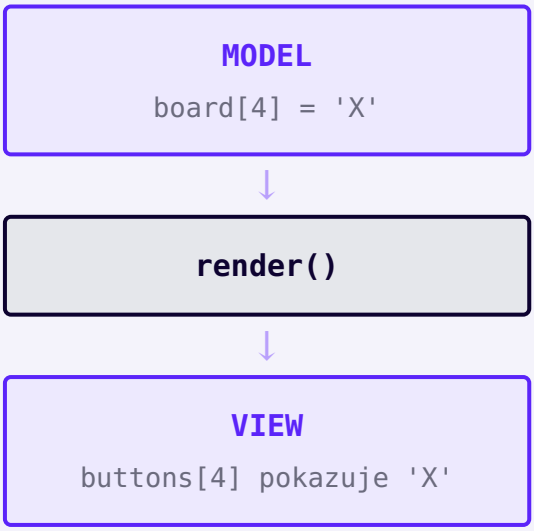
```
# text przycisków — jedyne miejsce, gdzie przechowuje się „czy
pole jest zajęte, czy nie”
if polya[indeks]["text"] != "":
    return
```

To zadziałało dla małego prototypu. Ale wyobraź sobie efekt zawisusu (Rozdział 17.23): tymczasowo wyświetla się "X" na przycisku pustym przed wykonaniem ruchu. Jeśli stan ŻYJE w Tekst przycisku — Program nie potrafi już rozróżnić "hover" od "real move".



Jeśli oba mają ten sam sygnał, jak je odróżnić?

Modeluj osobno, tylko przycisk



Dane płyną w jednym kierunku: model → render() → widżet. Nigdy nie jest odwrotnie.

Przycisk może TYMCZASOWO wyświetlać inny model
Podczas najechania `buttons[4]["text"] == "X"`, a `board[4] == ""` — i jest W PORZĄDKU, jeśli model pozostaje źródłem prawdy. Problem zaczyna się dopiero wtedy, gdy kod (np. winner check) odczytuje stan z tekstu przycisku zamiast z modelu — zobacz Debug Lab 3, sekcja 17.29.

board = lista liniach – model końcowy

board_model.py

```
board = ["", "", "", "", "", "", "", "", ""]
#board[4]=„X”
```

Praktyka: Model vs. Widget-as-State
Automatyczna kontrola – na modelu zabawki widgetu bez Tkinter pokazania, dlaczego widget-as-state psuje

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/17-18/index.html)

GameState z dataclass

Od prototypowych zmiennych globalnych do pojedynczego typowanego obiektu stanu.

GameState — @dataclass (rozdział 14)

game_state.py

```
from dataclasses import dataclass, field

@dataclass
class GameState:
    board: list[str] = field(default_factory=lambda: [""] * 9)
    current_player: str = "X"
    game_over: bool = False
    winner: str | None = None
    winning_line: tuple[int, int, int] | None = None
    score_x: int = 0
    score_o: int = 0
    draws: int = 0
```

NIE board: list[str] = [„”
Dla `@dataclass` to nawet nie jest cichy błąd: Python zauważa zmienną domyślną wartość (`list` , `dict` , `set`) właśnie przy ustalaniu klasy i natychmiast odmawia — `ValueError: mutable default <class 'list'> for field board is not allowed: use default_factory` . Klasa z takim łańcuch znaków w ogóle się nie utworzy, więc do „dzielenia między instancjami” (rozdział 14 omawiał tę pułapkę dla zwykłych domyślnych argumentów zmiennych) nie dojdzie. `field(default_factory=lambda: [""] * 9)` jest jedynym sposobem dataclass by za każdym razem, gdy tworzymy instancję, pole otrzymało nowy lista.

GameState przechowuje TYLKO dane partii — żadnego widżetu w środku.

Praktyka: GameState jak dataclass
Automatyczna kontrola — wartości domyślne, niezależność instancji, pola stanu

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/17-19/index.html)

Architektura TicTacToeApp

app.root, a nie class TicTacToeApp(tk.Tk), jest tym samym składem co w rozdziale 16.

app HAS-A root — to nie jest dziedziczenie po Tk

Składanie a dziedziczenie

Możliwe, ale niekonieczne

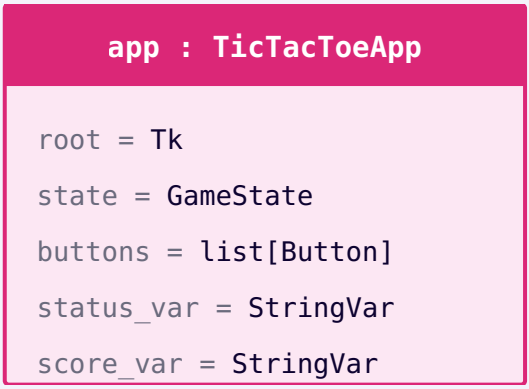
```
class TicTacToeApp(tk.Tk):
    def __init__(self):
        super().__init__()
        ...
```

Jak w rozdziale 16

```
class TicTacToeApp:
    def __init__(self, root):
        self.root = root
        ...
```

Co używać dzisiaj: Dziedzictwo od `tk.Tk` technicznie działa, ale nie jest konieczne i nie zgadza się ze sposobem konstrukcji załączników w rozdziale 16. `app.root` (kompozycja, „HAS-A”

Graf obiektowy aplikacji



app przechowuje widżety i LINK do stanu, a nie odwrotnie.

Domena kontra UI — podział odpowiedzialności

Czysta logika domenowa (brak Tkinter)	UI- logika (wewnątrz TicTacToeApp)
find_winner(board)	on_cell_click / attempt_move – wywołuje domenę, a potem render
is_draw(board)	on_cell_enter / on_cell_leave - Zapowiedź
index → row/column	render() - Rysuje widżety Z modelu (state)
	new_round() / new_match() – Cykl życia

Sprawdzanie „czy mogę testować bez okna?”
Dobry test architektury: jeśli do sprawdzenia funkcji konieczne jest otwarcie okna Tkinter — to prawdopodobnie UI- logika. Jeśli można ją wywołać z normalną listą/łańcuchem i uzyskać wynik — to logika domeny i MOŻNA byłoby ją wyodrębnić jako osobny moduł.

Praktyka: Złóż graf obiektowy TicTacToeApp
Moduł tkinter otwiera natywne okno Python — wykonaj lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/17-20/index.html)

Adaptacyjne pole gry

Gra nie powinna się rozpadać po zmianie rozmiaru okna — grid() i weight rozwiązują to tak samo jak w rozdziale 16.

Pole nie powinno stać się kruchą strukturą widżetów o stałym rozmiarze

adaptivnoe_pole.py

```
# outer — kontener nadrzędny (build_ui); łańcuch znaków 1 — to pola łańcuch znaków)
outer.rowconfigure(1, weight=1)

board_frame = ttk.Frame(outer)
board_frame.grid(row=1, column=0, columnspan=3, sticky="nsew")
for i in range(3):
    board_frame.rowconfigure(i, weight=1)
    board_frame.columnconfigure(i, weight=1)

for index in range(9):
    btn = tk.Button(board_frame, text="", font=("Arial", 28, "bold"), width=3, height=1)
    btn.grid(row=index // 3, column=index % 3, sticky="nsew", padx=3, pady=3)
```

sticky='nsew' + weight — na KAŻDYM poziomie zagnieżdżenia
Te same techniki adaptacyjne `grid()` jak w sekcji 16.15 — ale muszą być stosowane na obu poziomach jednocześnie. Wewnątrz `board_frame` już nie wystarczy nadać wagę wierszom/kolumnom, jeśli sam `board_frame` nie rozciąga się w swojej komórce `outer`: potrzeba I `outer.rowconfigure(1, weight=1)` na rodzica, I `sticky="nsew"` faktycznie `board_frame.grid()`. Pomiń którekolwiek z dwóch — a pole przestanie rosnąć wraz z oknem w pionie.

Nie obiecujemy idealnie kwadratowych kwadratów na każdej platformie

`width=3, height=1` u `tk.Button` ustawia rozmiar w jednostkach tekstowych (znaki/czcionki łańcuch znakówx) zamiast pikseli — dokładny stosunek obrazu zależy od czcionki i tematu danego systemu operacyjnego. Sprawiamy, że pole jest responsywne i uporządkowane — nie gwarantujemy matematycznie dokładnego kwadratu na wszystkich platformach jednocześnie.

Widoczne w prawdziwym oknie, nie tylko w kodzie

Po lewej — normalny rozmiar okna, po prawej — to samo okno po powiększeniu: ten sam stan gry, ale pola znacznie większe

Ta sama aplikacja i ten sam stan partii. Po lewej stronie jest oryginalny rozmiar okna, po prawej to samo okno po powiększeniu. Komórki zyskują dodatkową przestrzeń dzięki weight i sticky=„nsew”

Praktyka: Pole adaptacyjne przy zmianie rozmiaru okna

Moduł tkinter otwiera natywne okno Python — wykonaj lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/17-21/index.html)

X/O stylu wizualnego

X i O powinny się różnić na pierwszy rzut oka – i to nie tylko kolorem.

Paleta gier – najpierw widziane, potem numeryczne

X – fioletowy (`#5B24F9`), O — różowy (`#DB2777`) — te same korporacyjne kolory kursu co wszędzie indziej na stronie. Linia zwycięska jest podświetlona miękkim zielonym tłem, a na końcu gry znaki zmieniają się wyciszony — widget `disabled` sygnalizuje: „jest coś więcej nie możesz chodzić.”

Dlaczego tutaj używa się klasycznego tk.Button, a nie ttk

Świadomy wybór, a nie „zapominanie o ttk”
Rozdział 16 uczy cię preferować `ttk` tam, gdzie jest tematyczny odpowiednik (rozdział 16.12). Ale kolorowanie pojedynczej komórki w kolor gracza bezpośrednio przez `fg=` / `bg=` dokładnie to, co `tk.Button` sprawia, że jest proste i przewidywalne, a motyw przewodni `ttk` zazwyczaj nakłada takie kolory na widżecie przez cały czas `ttk.Style()` . Rama, status i muzyka są nadal budowane `ttk` — wybór widgetu był świadomy dla konkretnego zadania, a nie z przyzwyczajenia.

cvet_otmetki.py

```
MARK_COLOR = {"X": "#5B24F9", "O": "#DB2777"}

btn.config(text=mark, fg=MARK_COLOR.get(mark, "#0D0230"))
```

Nie tylko kolor, ale także litera i tekst statusu

Zwycięzca jest widoczny nie tylko po zielonym podświetleniu: litery „X”/„O” pozostają na miejscu, a status wyraźnie pisze „Wygrał gracz X!”. To ważna zasada dostępności: nie można polegać tylko na kolorze jako jedynym sygnale.

Praktyka: malarstwo X i O

Moduł tkinter otwiera natywne okno Python — wykonaj lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/17-22/index.html)

Hover preview przez bind()

Pokazanie możliwego ruchu nie oznacza jego dokonania. To właśnie ten efekt dowodzi, dlaczego model jest potrzebny.

Najazd — charakterystyczny efekt wizualny rozdziału

Najazd na pustą komórkę pokazuje blade podgląd X

Prawdziwe okno pełnej wersji: przebieg X, tymczasowy podgląd nie zmienia board. Fragment poniżej odpowiada jedynie za efekt zawisu.

Po X zakręcie najecham pokazuje blady podgląd O

Rzeczywiste okno tej samej wersji zmienia się wraz z current_player w miarę postępów O:, ale pusta komórka w modelu pozostaje pusta.

hover_preview.py

```
def on_cell_enter(self, index):
    state = self.state
    if state.game_over or state.board[index]:
        return   # nie pokazuj podglądu — koniec gry lub
komórka jest zajęta
    self.buttons[index].config(text=state.current_player,
fg=HOVER_COLOR)

def on_cell_leave(self, index):
    self.render()   # nie 'text=„”
```

on_cell_leave NIE powinno po prostu usuwać tekstu

Naiwna implementacja `on_cell_leave` typu `self.buttons[index].config(text="")` wymaże już WYKONANY ruch, jeśli gracz najecha myszką nad zajętą komórką w inny sposób – Debug Lab 13 (Rozdział 17.29) szczegółowo to wyjaśnia. Poprawny kod zawsze rysuje z modelu, zamiast zgadywać, co wymazać.

dowód: Model się nie zmienia

hover_ne_menyaet_model.py

```
assert app.state.board[4] == ""
app.on_cell_enter(4)
assert app.state.board[4] == ""     #AFTER zawisłem — nadal
pusty
app.on_cell_leave(4)
assert app.state.board[4] == ""     # i po opuszczeniu kursora
— też
```

Praktyka: hover preview na prawdziwym polu

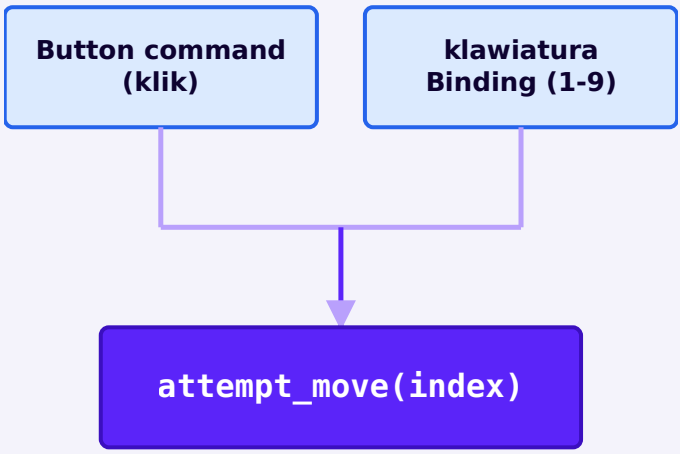
Moduł tkinter otwiera natywne okno Python — wykonaj lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/17-23/index.html)

Sterowanie klawiaturą

1-9 to te same ruchy co kliknięcie myszką, przez ten sam kod.

Mysz i klawiatura spotykają się w jednym punkcie



Obie ścieżki prowadzą do JEDNEJ I TEJ SAMEJ funkcji — zasady się nie dublują.

keyboard_control.py

```
def on_key(self, event):
    if event.keysym in ("r", "R"):
        self.new_round()
        return
    if event.char and event.char.isdigit():
        n = int(event.char)
        if 1 <= n <= 9:
            self.attempt_move(n - 1)   # klawisz 1 -> indeks
0, ... 9 -> indeks 8
```

Klawiatura nie otrzymuje swojej kopii zasad

Obsługa klawiatury TYLKO przekształca wciśnięty klawisz na numer pola i wywołuje `attempt_move()` jest tą samą funkcją, która powoduje kliknięcie myszką. Jeśli ścieżka klawiatury kopiowała sprawdzenie zwycięzcy osobno, oba zestawy zasad ostatecznie się rozdzieliły (Debug Lab 9, sekcja 17.29 – podobny błąd z literówką w linijce).

Klawisz	Indeks Komórki
1	0
2	1
3	2
4	3
5	4
6	5
7	6
8	7
9	8

keysym przeciwko char – char jest tu wybierany

Rozdział 17.8 pokazała różnicę: `event.char` to faktyczny wpisany znak. Dla klawiszy numerycznych w górnym rzędzie jest on przewidywalnie równy samej liczbie; z dodatkową klawiaturą numeryczną (NumPad) zachowanie jest zwykle takie samo, jeśli NumLock jest włączone, ale może się różnić w zależności od platformy, układu i stanu NumLock — jeśli klawiatura jest ważna dla Twojego projektu, przetestuj obie opcje na docelowym systemie. Tutaj `event.char` jest nadal wygodniejsze niż porównanie przez `event.keysym`.

Praktyka: Klucze 1-9 i R

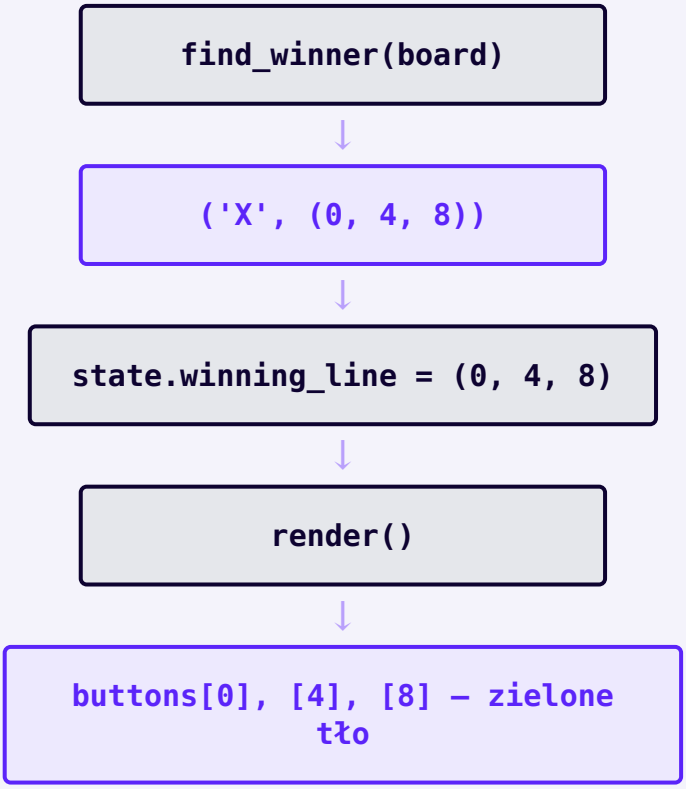
Moduł tkinter otwiera natywne okno Python — wykonaj lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/17-24/index.html)

Wyróżnianie linii zwycięskiej

Model już wie, które trzy komórki wygrały – render() po prostu je maluje.

Dane liniowe prowadzą do oświetlenia



Model określa, KTÓRE komórki wygrały – render() po prostu je maluje.

podsvetka_linii.py

```
for index, btn in enumerate(self.buttons):
    is_win_cell = state.winning_line is not None and index in
state.winning_line
    btn.config(bg=WIN_BG if is_win_cell else NEUTRAL_BG)
```

Podkreśl DOKŁADNIE linię — nie byle jakie pasujące symbole
Jeśli na polu jest na przykład cztery „X”, ale zwyciężyły tylko trzy z nich na diagonali — podświetlane jest ŚCISŁO `winning_line`, nie każda komórka oznaczona jako „X” `index in state.winning_line` to gwarantuje.

Przekątna 0-4-8 jest podświetlona na zielono, status to Player Won X

Rzeczywiste okno pełnej wersji gry: diagonalny zwycięstwa jest podświetlony, litery pozostają widoczne. Fragment powyżej to tylko kod samego podświetlenia, a nie cała droga do zwycięstwa.

Praktyka: Podkreśl zwycięską linię
Moduł tkinter otwiera natywne okno Python — wykonaj lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/17-25/index.html)

Wyniki meczów

Partia się kończy — mecz trwa dalej: wynik żyje dłużej niż jedna runda.

Konto znajduje się w GameState, a nie w losowej pojedynczej zmiennej

Wynik: X: 2, O: 1, Remisy: 1

Prawdziwe okno pełnej gry: wynik meczu poniżej pola gry. Poniższy fragment pokazuje, jak wartości punktowe mieszczą się w tej linii.

schet.py

```
if winner == "X":
    state.score_x += 1
else:
    state.score_o += 1

self.score_var.set(f"X: {state.score_x}  |  O:
{state.score_o}  | Losowanie: {state.draws}")
```

Uważaj na rzadki znak separatora w Tkinter- etykietach
Piękny symbol markera „•” `" | "` działa przewidywalnie wszędzie.

Wynik wzrasta dokładnie raz na mecz – wewnątrz `attempt_move()`, zaraz po znalezieniu zwycięzcy lub stwierdzeniu remisu, a nie gdzie indziej.

Praktyka: Rachunkowanie wyników meczów
Automatyczna weryfikacja – Czysta aktualizacja wyników na podstawie wyniku rundy

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/17-26/index.html)

New Round vs New Match

Czy „Nowa Gra”

Dwa różne „resety”

New Round (nowa runda)	New Match (nowy mecz)
Czyści pole bieżącego gracza, game_over, winner, winning_line	Robi dokładnie to samo, co New Round
ZAPISUJE wynik (score_x, score_o, draws)	I DODATKOWO zeruje wynik
Ogłoszony po każdym meczu	Wywoływane, gdy mecz jest całkowicie zakończony

new_round_vs_new_match.py

```python
def new_round(self):
    self.state.board = [""] * 9
    self.state.current_player = "X"
    self.state.game_over = False
    self.state.winner = None
    self.state.winning_line = None
    self.render()

def new_match(self):
    self.state.score_x = 0
    self.state.score_o = 0
    self.state.draws = 0
    self.new_round()   #New Match w pełni obejmuje New Round
```

Mylenie ich to prawdziwy błąd (Debug Lab 12, Rozdział 17.29)
Jeśli przycisk "Nowa gra" wywoła resetowanie wyniku, gracze tracą wynik turniejowy po każdej grze. Jeśli przycisk "Nowy mecz" nie zresetuje wyniku, stary wynik jest przeciągany do nowego turnieju. Nazwy funkcji powinny jednoznacznie odzwierciedlać tę różnicę.

Po New Round pole jest puste, a konto X:1 zapisane

Rzeczywiste okno pełnej wersji po new_round(): stan rundy został zresetowany, wynik meczu zapisany. Kod powyżej — fragment metod pełnej aplikacji.

Po New Match pole jest puste, a cały wynik wynosi zero

Prawdziwe okno pełnej wersji po new_match(): resetuje zarówno rundę, jak i wynik X/O/draws. Powyższy kod jest fragmentem pełnych metod aplikacji.

Opcjonalna ekspansja: zapisz wynik meczu w JSON

Poniżej znajduje się tylko fragment — ciało funkcji `save_scores()` z `tic_tac_toe.py`. Pełny plik już zawiera niezbędne `import json`, `from pathlib import Path` i definicja `GameState` – tutaj się nie powtarzają.

fragment save_scores() z tic_tac_toe.py

```
SCORES_PATH = Path("tic_tac_toe_scores.json")
# w stosunku do bieżącego katalogu roboczego

def save_scores(state):
    with SCORES_PATH.open("w", encoding="utf-8") as f:
        json.dump({"x": state.score_x, "o": state.score_o,
"draws": state.draws}, f,
                  ensure_ascii=False, indent=2)
```

Opcjonalne w pierwszej wersji gry

Trwałe Relacje – Miłe rozszerzenie (Rozdział 15: `pathlib` + JSON), ale NIE wymóg dla działającej gry. W `tic_tac_toe.py` jest to możliwe przez wyraźną flagę `persist_scores=True` — domyślnie wyłączone.

Praktyka: New Round, New Match i zapisywanie konta

Automatyczna weryfikacja — semantyka resetu rundy/meczu i zachowanie JSON w wirtualnym systemie plików przeglądarki

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/17-27/index.html)

Testowanie gry bez Tkinter

Czysta logika jest bezpośrednią nagrodą dla architektury: testujemy zasady gry bez ani jednego otwartego okna.

Dlaczego czysta logika jest łatwa do przetestowania

test_bez_okna.py

```
board = ["X", "X", "X", "", "", "", "", "", ""]
winner, line = find_winner(board)
assert winner == "X"
# Brak otwartego okna Tkinter – tylko lista linie i regularna
funkcja.
```

Bezpośrednia nagroda za architekturę sekcji 17.20

Jest to możliwe właśnie dlatego, że `find_winner` / `is_draw` nic o Tkinter. Jeśli zwycięzca był wyłoniony bezpośrednio w obsłudze kliknięć przycisków, trzeba by go przetestować przez prawdziwe okno i prawdziwe kliknięcia.

Pełna macierz testowa gry

Weryfikacja	Oczekiwany wynik
X wygrywa wszystkie 8 linii	find_winner zwraca ('X', linia) dla każdego
O wygrywa wszystkie 8 linii	find_winner zwraca ('O', linia) dla każdego
Puste/niekompletne pole	find_winner zwroty (None, None)
Znany losowanie	is_draw(board) is True
Ostatni ruch wygrywa jednocześnie	find_winner wygrywa, is_draw is False
Ruch na zajęte pole	attempt_move niczego nie zmienia, current_player to samo
Skręt za game_over	attempt_move niczego nie zmienia
new_round()	po opróżnieniu game_over=False konto jest ZAPISANE
new_match()	to samo, ale licznik ZEROWANY

Sprawdź wszystkie osiem linii, a nie tylko jedną losowo

Test tylko jednej linii nie wykryje literówki w opisie przekątnej – Debug Lab 9 (sekcja 17.29) jest zaprojektowana dokładnie tak, aby jedna łańcuch znaków testu jej nie zauważyła.

Praktyka: pełny zestaw testów bez Tkinter

Automatyczna weryfikacja — cała macierz regresji gry: 8 linii × 2 graczy, remis, ostatni ruch, nieprawidłowy ruch, reset

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/17-28/index.html)

Debug Labs — typowe błędy gry zdarzeniowej

Każdy błąd tutaj występuje w prawdziwych projektach studenckich — naucz się rozpoznawać objaw, zanim otworzysz debugator.

Niewielka kolekcja typowych błędów w grach opartych na wydarzeniach, każdy z objawem i Poprawka. Niektóre z nich już widzieliście wcześniej w rozdziale; Tutaj są one zebrane jako źródło referencyjne.

DEBUG LAB 1: NIESTANDARDOWE POWIĄZANIE WIDGETÓW Z BREAK WYCISZA KLAWISZE GRY

focus_problem.py

```
root.bind("<Key>", on_key)

entry = tk.Entry(root)
entry.pack()
entry.bind("<Key>", lambda e: "break")   # zatrzymuje łańcuch bindtags
entry.focus_set()
```

```
# Пока фокус в entry, нажатия цифр не долетают до on_key —
# entry.bind(..., 'break') обрывает цепочку прямо здесь.
```

Co widać na ekranie

Zdarzenie klawiatury najpierw trafia do skoncentrowanego widżetu, a następnie bindtags: sam widżet → jego → root → klasie „all” `"break"` - dalsze wiązania w łańcuchu (w tym `root.bind`) na to wydarzenie po prostu nie zostanie zwołane. Bez tego, `"break"` zdarzenie normalnie kontynuowałoby ścieżkę do `root` też — samo skupienie nie blokuje przechwytywania na najwyższym poziomie.

POPRAWIONY KOD

focus_fixed.py

```
root.bind("<Key>", on_key)

entry = tk.Entry(root)
entry.pack()
entry.focus_set()   # bez <Key>wiązania -binding,
zdarzenie osiąga root
```

DEBUG LAB 2: LAMBDA BEZ I=INDEKS — PÓŹNE PRZYPISANIE

lambda_pozdnyaya_privyazka.py

```
for indeks in range(9):
    btn = tk.Button(frame, command=lambda:
attempt_move(indeks))
    btn.grid(row=indeks // 3, column=indeks % 3)
```

```
# Клик по ЛЮБОЙ из девяти кнопок ставит отметку в клетку 8
# потому что все lambda читают ОДНУ И ТУ ЖЕ переменную inde
```

Co widać na ekranie

W momencie kliknięcia cykl dawno się zakończył, i `indeks` wynosi 8 dla wszystkich dziewięciu domknięć naraz — nie „pamiętały”

POPRAWIONY KOD

lambda_fixed.py

```
for indeks in range(9):
    btn = tk.Button(frame, command=lambda i=indeks:
attempt_move(i))
    btn.grid(row=indeks // 3, column=indeks % 3)
```

DEBUG LAB 3: NAJECHANIE MYLI ZAPOWIEDŹ Z PRAWDZIWYM RUCHEM

hover_kak_hod.py

```
def on_cell_enter(self, index):

self.buttons[index].config(text=self.state.current_player)

def find_winner_from_widgets(self):
    board = [b["text"] for b in self.buttons]   # czyta
TEKST PRZYCISKU
    return find_winner(board)
```

```
# Просто наведя курсор на три пустые клетки одной линии подря
# можно получить объявление победы без единого клика.
```

Co widać na ekranie

Jeśli czek zwycięzcy odczytuje status z `button["text"]`, a kursor tymczasowo zapisuje tam tekst — podgląd staje się nie do odróżnienia od rzeczywistego ruchu. Rozdział 17.18 wyjaśnia, dlaczego model musi być źródłem prawdy.

POPRAWIONY KOD

hover_ne_hod.py

```
def on_cell_enter(self, index):
    if self.state.game_over or self.state.board[index]:
        return

self.buttons[index].config(text=self.state.current_player,
fg=HOVER_COLOR)

def find_winner_call(self):
    return find_winner(self.state.board)   # odczytuje
MODEL, nie przyciski
```

DEBUG LAB 4: GRACZ PRZEŁĄCZA SIĘ PRZED SPRAWDZENIEM ZWYCIĘZCY

smena_do_proverki.py

```
state.board[index] = state.current_player
state.current_player = "O" if state.current_player ==
"X" else "X"

winner, line = find_winner(state.board)
if winner:
    status_var.set(f"Wygrana gracza {winner}!")   # ale
current_player już się zmieniło
```

```
# Статус верный ('Победил X'), но если где-то дальше по ко
# используется state.current_player как 'кто только что вы
```

Co widać na ekranie

Kolejność operacji jest ważna: zmiana gracza musi nastąpić PO weryfikacji zwycięzcy i TYLKO jeśli gra trwa dalej. W przeciwnym razie każdy kod, który patrzy na `current_player` zaraz po turze zobaczy następnego gracza, a nie autora zwycięskiego ruchu.

POPRAWIONY KOD

smena_posle_proverki.py

```
state.board[index] = state.current_player
winner, line = find_winner(state.board)
if winner:
    state.winner = winner
# 'kto wygrał' jest ustalone PRZED zmianą gracza
    state.game_over = True
elif not is_draw(state.board):
    state.current_player = "O" if state.current_player
== "X" else "X"
```

DEBUG LAB 5: REMIS SPRAWDZANY JEST PRZED ZWYCIĘSTWEM

nichya_do_pobedy.py

```
if all(state.board):
    status_var.set(„Draw!”)
elif find_winner(state.board)[0]:
    status_var.set(„Zwycięstwo!”)
```

```
# Девятый ход одновременно заполняет поле и выигрывает диагон
# программа объявляет 'Ничья!', хотя есть явный победитель.
```

Co widać na ekranie

Rozdział 17.17 pokazała dokładnie taki scenariusz. Sprawdzanie, czy pole jest wypełnione, musi zostać wykonane PO sprawdzeniu zwycięzcy, w przeciwnym razie ostatni ruch przegrywa przez fałszywe losowanie.

POPRAWIONY KOD

pobeda_do_nichyej.py

```
winner, line = find_winner(state.board)
if winner:
    status_var.set(f"Wygrana gracza {winner}!")
elif all(state.board):
    status_var.set(„Draw!”)
```

DEBUG LAB 6: RUCH NA ZAJĘTĄ KOMÓRKĘ JEST DOZWOLONY

zanyataya_kletka.py

```
def attempt_move(index):
    state.board[index] = state.current_player   #brak
weryfikacji!
```

```
# Повторный клик по уже занятой клетке перезаписывает X на
# (или наоборот) — чужой ход стирается.
```

Co widać na ekranie

Walidacja powinna być PIERWSZĄ linią, zanim zmieni stan – sekcja 17.15 określa to jako obowiązkowy pierwszy krok algorytmu ruchu.

POPRAWIONY KOD

zanyataya_kletka_fixed.py

```
def attempt_move(index):
    if state.game_over or state.board[index]:
        return
    state.board[index] = state.current_player
```

DEBUG LAB 7: GRACZ PRZEŁĄCZA SIĘ NAWET PO NIEPRAWIDŁOWYM KLIKNIĘCIU

smena_pri_nevalidnom_klike.py

```
def attempt_move(index):
    if not state.board[index]:
        state.board[index] = state.current_player
    state.current_player = "O" if state.current_player
== "X" else "X"   # na zewnątrz if!
```

```
# Клик по занятой клетке ничего не рисует —
# но ход всё равно передаётся сопернику. Игрок теряет очередь
```

Co widać na ekranie

Linia zmiany gracza znalazła się POZA blokiem `if` — wykonuje się niezależnie od tego, czy ruch był prawidłowy. Rozdział 17.15: nieprawidłowy ruch nie powinien zmieniać gracza.

POPRAWIONY KOD

smena_pri_nevalidnom_klike_fixed.py

```
def attempt_move(index):
    if state.board[index]:
        return
    state.board[index] = state.current_player
    state.current_player = "O" if state.current_player
== "X" else "X"
```

DEBUG LAB 8: BRAK GAME_OVER CHECK – GRA TRWA DALEJ PO WYGRANEJ

net_game_over.py

```
def attempt_move(index):
    if state.board[index]:
        return
    state.board[index] = state.current_player
    # ... sprawdzanie zwycięzcy, ale BRAK return/guard
na początku funkcji
```

```
# После объявления победы X можно продолжать кликать по пус
# и даже 'перезаписать' исход партии дополнительными ходами
```

Co widać na ekranie

Bez jawnej weryfikacji `state.game_over` na początku `attempt_move()` nic nie przeszkodzi ci kontynuować grę po jej zakończeniu.

POPRAWIONY KOD

game_over_guard.py

```
def attempt_move(index):
    if state.game_over or state.board[index]:
        return
    ...
```

DEBUG LAB 9: LITERÓWKA W WINNING_LINES

opechatka_v_linii.py

```
WINNING_LINES = (
    (0, 1, 2), (3, 4, 5), (6, 7, 8),
    (0, 3, 6), (1, 4, 7), (2, 5, 8),
    (0, 4, 6), (2, 4, 6),   # pierwsza przekątna musi
być (0, 4, 8)!
)
```

```
# X ставит 0, 4 и 8 — три подряд по настоящей диагонали —
# но игра не объявляет победителя, потому что такой кортеж
```

Co widać na ekranie

Literówka jest cicha: kod jest poprawny składniowo i przechodzi nawet powierzchowny test dla „pewnej”

POPRAWIONY KOD

linii_fixed.py

```
WINNING_LINES = (
    (0, 1, 2), (3, 4, 5), (6, 7, 8),
    (0, 3, 6), (1, 4, 7), (2, 5, 8),
    (0, 4, 8), (2, 4, 6),
)
```

DEBUG LAB 10: ZRESETUJ INTERFEJS, ALE NIE MODEL

reset_tolko_ui.py

```
def new_round(self):
    for btn in self.buttons:
        btn.config(text="")   # guziki są czyste...
    # ... Ale self.state.board wciąż trzyma stare X/O!
```

```
# Поле выглядит пустым, но следующий attempt_move()
# натыкается на 'клетка уже занята' там, где на экране пусто.
```

Co widać na ekranie

Reset wizualny nie jest tym samym co resetowanie modelu. Rozdział 17.18: Przyciski wyświetlają tylko stan; jeśli wyczyścisz tylko je, model nadal kłamie co do rzeczywistego stanu partii.

POPRAWIONY KOD

reset_model_i_ui.py

```
def new_round(self):
    self.state.board = [""] * 9
    self.state.current_player = "X"
    self.state.game_over = False
    self.render()   #UI jest aktualizowany OD modelu, a
nie osobno
```

DEBUG LAB 11: ZRESETOWAŁEM MODEL, ALE NIE ZADZWONIŁEM DO RENDER()

reset_tolko_model.py

```
def new_round(self):
    self.state.board = [""] * 9
    self.state.current_player = "X"
    self.state.game_over = False
    # zapomniano self.render() na końcu!
```

```
# Модель для нового раунда готова правильно —
# но на экране всё ещё видны старые X и O из прошлой партии
```

Co widać na ekranie

Druga strona tego samego błędu: brak połączenia `render()` zmiany w modelu nigdy nie pojawiają się na ekranie. Rozdział 17.20 stanowi tę zasadę: `render()` to miejsce, które rysuje widżety Z modelu (state) i musi być wywoływane jawnie po każdej zmianie tego modelu.

POPRAWIONY KOD

reset_s_render.py

```
def new_round(self):
    self.state.board = [""] * 9
    self.state.current_player = "X"
    self.state.game_over = False
    self.render()
```

DEBUG LAB 12: „NOWA GRA” LOSOWO ZERUJE WYNIK MECZU

novaya_igra_obnulyaet_schet.py

```
def new_round(self):
    self.state.score_x = 0   # powinno być tylko w new_match()!
    self.state.score_o = 0
    self.state.board = [""] * 9
```

```
# После каждой победы счёт тут же обнуляется —
# турнирный счёт невозможно накопить.
```

Co widać na ekranie

Rozdział 17.27: New Round i New Match to różne transakcje. Zeroowanie rachunku należy WYŁĄCZNIE `new_match()`.

POPRAWIONY KOD

new_round_bez_scheta.py

```
def new_round(self):
    self.state.board = [""] * 9
    # konto (score_x/score_o/draws) nie dotykaj tutaj
```

DEBUG LAB 13: ON_CELL_LEAVE WYMAZUJE JUŻ WYKONANY RUCH

leave_stiraet_hod.py

```
def on_cell_leave(self, index):
    self.buttons[index].config(text="")
#naively „usuń podglądy”
```

```
# После того как игрок делает настоящий ход и потом уводит
# отметка на клетке пропадает с экрана — хотя в модели ход
```

Co widać na ekranie

`on_cell_leave` nie wie, czy był zapowiedź, czy prawdziwy ruch na klatce — po prostu bezwarunkowo usuwa tekst. Właściwe podejście to nie zgadywać, lecz rysować na podstawie modelu, który zna prawdę dokładnie.

POPRAWIONY KOD

leave_fixed.py

```
def on_cell_leave(self, index):
    self.render()   # decyduje, co powinno być na
komórce
```

DEBUG LAB 14:EVENT.WIDGET UŻYWANY JAKO STAN GRY

widget_kak_state.py

```
def on_click(self, event):
    event.widget.config(text=self.state.current_player)

# gdzie jest wpis w board tutaj? Nie ma go — państwo
żyje TYLKO w widgetzie
```

```
# find_winner(self.state.board) никогда не видит эти ходы —
# победа не определяется, потому что модель не менялась.
```

Co widać na ekranie

`event.widget` — źródło UI- zdarzenia (rozdział 17.8), a nie domena gry. Zapis ruchu musi trafić do `state.board`, w przeciwnym razie cała logika domenowa (szukanie zwycięzcy, remis) działa na ślepo.

POPRAWIONY KOD

widget_i_model.py

```
def on_click(self, event, index):
    self.attempt_move(index)   # zmienia state.board,
potem render()
```

DEBUG LAB: NAKED <BUTTON-1> ZASTĘPUJE SEMANTYCZNE DZIAŁANIE BUTTON

goloj_button1.py

```
btn.bind("<Button-1>", lambda e, i=index:
attempt_move(i))
#command= wcale nie jest ustawiony
```

```
# Обработчик связан только с конкретным событием мыши.
# Основное действие Button больше не описано через command=
```

Co widać na ekranie

Rozdział 17.10: `command=` opisuje główną czynność Button, a wbudowane zachowanie klasy widgetu decyduje, które sposoby aktywacji go wywołują. Konkretne klawisze zależą od rodzaju Button, motywu i platformy. Same powiązania `<Button-1>` zna tylko kliknięcie myszką i omija tę semantykę — więc nie używaj go zamiast domyślnego command.

POPRAWIONY KOD

command_fixed.py

```
btn.config(command=lambda i=index: attempt_move(i))
```

DEBUG LAB: TIME.SLEEP() W ANIMACJI ZWYCIĘSTWA ZAWIESZA OKNO

sleep_v_animacii.py

```
import time

def pulse_winning_line(self):
    for _ in range(6):
        self.buttons[0].config(bg="green")
        time.sleep(0.15)
        self.buttons[0].config(bg="white")
        time.sleep(0.15)
```

```
# Окно перестаёт отвечать почти на две секунды анимации —
# та же ошибка, что и в главе 16 (раздел 16.32).
```

Co widać na ekranie

`time.sleep()` blokuje całą pętlę zdarzeń. Dla animacji nieblokujących musisz self-rescheduling przejść `after()` - Rozdział 17.30.

POPRAWIONY KOD

after_fixed.py

```
def pulse_winning_line(self, tick=0):
    color = PULSE_BG if tick == 0 else WIN_BG
    for i in self.state.winning_line:
        self.buttons[i].config(bg=color)
    if tick == 0:
        self.root.after(400, self.pulse_winning_line, 1)
```

DEBUG LAB 17: DODATKOWA BIND(..., ADD=„+”

dublirovannaya_privyazka.py

```
btn = ttk.Button(controls, text=„Nowa runda”,
command=self.new_round)
btn.bind("<Button-1>", lambda e: self.new_round(),
add="+")
# teraz jedno kliknięcie uruchamia new_round() PO
command I PO bind — dwa razy z rzędu
```

```
# Само по себе new_round() дважды подряд не лома
# но если бы это была, например, функция увеличе
```

Co widać na ekranie

`add="+"` dodaje JEDEN DODATKOWY handler bez zastępowania istniejącego (sekcja 17.12 określała to jako „trochę głębszy” `command=`, powtarzające wiązanie poprzez `bind(..., add="+")` uruchamia logikę ponownie na tym samym zdarzeniu `add="+"` świadomie tylko wtedy, gdy naprawdę potrzebujesz KILKU niezależnych opiekunów tego samego wydarzenia.

POPRAWIONY KOD

bez_dublirovaniya.py

```
btn = ttk.Button(controls, text=„Nowa runda”,
command=self.new_round)
# I to wszystko — jedno wiązanie wystarczy
```

Praktyka: wykrywanie wirusa na podstawie objawu
Automatyczna kontrola — dla zestawu opisanych objawów wybieramy właściwą przyczynę/naprawę

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/17-29/index.html)

Visual effects i after()

Jeden krótki, nieblokujący efekt – zwycięstwo powinno być zauważalne, a nie disco.

Jeden subtelny efekt — nie framework animacji

Po wygranej komórki linii zwycięskiej są raz delikatnie podświetlone i spokojnie wracają do stałego podświetlenia. W rzeczywistym oknie wygląda to jak krótka zmiana między accent-stanem a bazowym podświetleniem zwycięstwa; strona nie może pokazać animacji wideo, dlatego poniżej są prawdziwe kadry tej zmiany, a lokalna aplikacja przełącza je przez `after()`. Łącznie dwa rozróżnialne klatki: trzeci kadr pokrywa się z drugim nieprzypadkowo — efekt kończy się właśnie na stałym podświetleniu zwycięstwa i dalej nic się nie zmienia.

Rzeczywista pierwsza klatka: miękki akcent PULSE_BG.

Faktyczna druga klatka: stałe podświetlenie WIN_BG.

Stan ustalony: obraz pokrywa się z drugą klatką, ponieważ efekt się tam kończy.

Efekt jest celowo powściągliwy — brak ostrych, kontrastowych migotów.

pulse_winning_line.py

```
def pulse_winning_line(self, tick=0):
    if not self.state.winning_line:
        return
    color = PULSE_BG if tick == 0 else WIN_BG
    for index in self.state.winning_line:
        self.buttons[index].config(bg=color)
    if tick == 0:
        self._pulse_job = self.root.after(400,
self.pulse_winning_line, 1)
    else:
        self._pulse_job = None
```

Samoplanowanie after() — Znany trik z rozdziału 16

Ta sama technika self-rescheduling, co w rozdziale 16.22: każde wywołanie `pulse_winning_line` sam planuje kolejny tick through `after()` dopóki nie osiągnie limitu, pętla zdarzeń nie jest blokowana przez chwilę.

Odwołaj zaplanowany after() nowej rundy

Jeśli gracz kliknie „Nowa runda” `after()` już działa na pustym polu nowej rundy. `cancel_pulse()` przyczyny `after_cancel()` przed resetem obowiązuje ta sama dyscyplina trzech stanów co timer w sekcji 16.28.

cancel_pulse.py

```
def cancel_pulse(self):
    if self._pulse_job is not None:
        self.root.after_cancel(self._pulse_job)
        self._pulse_job = None
```

Żadnego time.sleep(): jeden spokojny przejście

Efekt nie miga w serii: linia otrzymuje miękki efekt accent-color i po 400 ms pozostaje w stałym podświetleniu zwycięstwa. Nawet krótkie efekty powinny być dyskretne i nie rozpraszać statusu „Player X/O won”

Praktyka: Dyskretna animacja zwycięstwa
Moduł tkinter otwiera natywne okno Python — wykonaj lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/17-30/index.html)

Tic-Tac-Toe Pro — pełny program i wyniki rozdziału

Ta sama gra co w rozdziale 17.6 — ale teraz z modelem, zasadami, testami i architekturą, które wytrzymają rozwój projektu.

Program końcowy

Końcowe okno Tic-Tac-Toe Pro z trwającą grą i wynikiem X:1 O:2 Remisy:1

Prawdziwe okno finalnej wersji to model, zasady, partytura, podświetlenie i klawiatura razem. Poniżej znajduje się struktura całego pliku, nie tylko jednego fragmentu.

Plik jest kompletny, samodzielny i nie posiada niewidzialnych zależności od innych lekcji:

projects/tkinter/tic-tac-toe/tic_tac_toe.py

tic_tac_toe.py — struktura

```
WINNING_LINES = (...)

def find_winner(board): ...
def is_draw(board): ...
def load_scores(): ...        # opcjonalne zapisywanie wyniku (17.27)
```

```
def save_scores(state): ...

@dataclass
class GameState:
    ...

class TicTacToeApp:
    def __init__(self, root, *, persist_scores=False): ...
    def build_ui(self): ...
    def build_board(self, outer): ...
    def attempt_move(self, index): ...
    def on_cell_enter(self, index): ...
    def on_cell_leave(self, index): ...
    def on_key(self, event): ...
    def render(self): ...
    def new_round(self): ...
    def new_match(self): ...
    def pulse_winning_line(self, tick=0): ...
    def cancel_pulse(self): ...      # odwołuje zaplanowany
after() (17:30)
    def on_close(self): ...

def main():
    root = tk.Tk()
    app = TicTacToeApp(root)
    root.mainloop()

if __name__ == "__main__":
    main()
```

Uruchom grę u siebie

`python tic_tac_toe.py` w terminalu — albo Run przycisku w VS Code, albo PyCharm. Porównaj z `tic_tac_toe_basic.py` (Rozdział 17.6) to ten sam wynik dla gracza, z zupełnie inną architekturą wewnętrzną.

Lista gotowych gier

MODEL

- pole (`board`) jest oddzielna od widgetów
- bieżący gracz — jawne pole stanu

- terminal state (game_over/winner) jest polem jawnym, a nie wyjściem z widgetów

RULES

- nieprawidłowe ruchy są odbite i nie zmieniają gracza
- wszystkie 8 linii wypłatnych testowanych na X i O
- losowanie jest sprawdzane PO wygranej

UI

- status tury jest zawsze widoczny
- zwycięzca jest widoczny nie tylko po kolorze
- pole reaguje na zmiany rozmiaru okien
- klawiatura działa

ARCHITEKTURA

- callback- i krótkie: walidacja → model → render
- czyste zasady są testowane bez okna
- render() ustala kanoniczny widok na podstawie modelu; Efekty tymczasowe są nałożone na to

Most do rozdziału 18

Dalej: GDZIE, nie tylko KTO i CO

W tej grze wydarzenia pomogły nam zrozumieć, KTO zareagował i CO powinno się wydarzyć. Ponadto wydarzenia z myszy podają nam WSPÓŁRZĘDNE — `event.x` oraz `event.y`. W następnym rozdziale współrzędne myszy staną się podstawą do rysowania na Canvas.

Praktyka: Tic-Tac-Toe Pro całość

Moduł tkinter otwiera natywne okno Python — wykonaj lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/17-31/index.html)

Streszczenie rozdziału 17

Co zbudowaliśmy i czego się nauczyliśmy

- Zdarzenie, callback, command i binding to cztery różne, powiązane pojęcia; callback command= zazwyczaj nie otrzymuje Event; callback zarejestrowany przez bind() go otrzymuje.
- command= opisuje główną akcję Button; bind() jest jawną odpowiedzią na konkretne zdarzenie myszy lub klawiatury. Wbudowane sposoby aktywacji Button zależą od klasy i platformy widgetu, więc command i bind nie powinny być traktowane jako zamienne.
- Stan gry — nie to samo co widżety, które go pokazują: najechanie myszą udowadnia to najlepiej.
- find_winner(board) i is_draw(board) — funkcje czyste, testowane bez żadnego otwartego okna; kolejność „najpierw zwycięstwo, potem remis” — część zasad, a nie szczegół implementacyjny.
- Mysz (command=) i klawiatura (bind na root) zbiegają się do tej samej funkcji attempt_move() – zasady gry istnieją tylko w jednym miejscu.
- render() ustawia kanoniczny wyświetlacz bazy Z modelu. Najechanie myszką i krótki akcent zwycięstwa tymczasowo zmieniają widok na górze bez zmiany GameState; następny render() ponownie całkowicie przywraca widok modelu.
- Subtelne efekty wizualne (podgląd po najechaniu, podświetlenie linii, puls zwycięstwa) opierają się na solidnej architekturze — są możliwe właśnie dlatego, że model jest oddzielony od widoku.

ROZDZIAŁ 18 · STR. 1132

Projekt: aplikacja do rysowania z Tkinter

Pełnoprawny edytor rysowania na Canvas: ołówek, figury, kolor, cofanie działań, zapisywanie i ładowanie — od pierwszego prototypu do architektury poziomu aplikacji.

Aplikacja do rysowania — wyjaśnienie

Tak będzie wyglądał gotowy projekt. Rozłożymy drogę do niego krok po kroku — od pustego płótna do architektury na poziomie aplikacji.

Gotowa aplikacja: Płótno z wieloma kształtami, pasek narzędzi z wybranym narzędziem, wybór kolorów, suwak grubości oraz łańcuch znaków stanu

Pod koniec tego rozdziału zbudujemy tę aplikację krok po kroku.

Aplikacja do rysowania — wyjaśnienie

Złóż własny mały „Paint”

1. Płótno do rysowania (widget `Canvas`);
2. Paska Pasków — przyciski wyboru kształtów;
3. Reakcja na ruchy myszy i kliknięcia;
4. Rysowanie linii, prostokątów, owalnych kształtów;
5. Wybór grubości i koloru;
6. Przycisk czyszczenia płótna i wolny los.

To da uczciwie działający program w kilku sekcjach – sekcja 18.7. A Następnie rozdział wraca i analizuje Canvas znacznie dokładniej: co jest naprawdę Przechowuje płótno, jak ułożone są współrzędne, co oznacza „cofnięcie akcji”

Zaczynamy

nachalo.py

```
import tkinter as tk

root = tk.Tk()
root.title(„Rysowanka”)
```

Praktyka: Zaczynam składać grę rysującą
Moduł tkinter otwiera natywne okno Python — wykonaj lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/18-02/index.html)

Dostosuj ekran. Stwórz płótno

Widżet Canvas — prostokątny obszar do rysowania w oknie Tkinter.

Personalizacja ekranu

nastrojka_ekrana.py

```
root.title(„Rysowanka”)
```

Stwórz płótno

Widżet `Canvas` jest prostokątnym obszarem, na którym możesz rysuj linie, kształty i tekst według współrzędnych, jak w Turtle (rozdziały 6-7), tylko wewnątrz okna Tkinter:

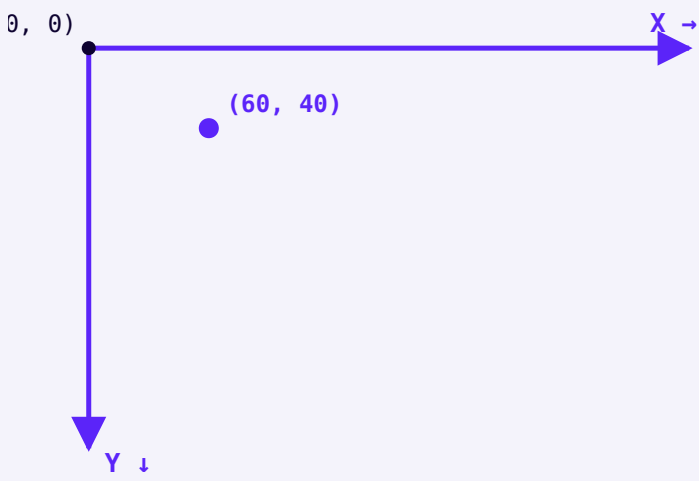
sozdaem_holst.py

```
canvas = tk.Canvas(root, width=600, height=400, bg="white")
canvas.pack()
```

Prawdziwe okno: 600×400 puste białe płótno wewnątrz okna Tkinter

Rzeczywiste okno: pusty obszar roboczy bezpośrednio po uruchomieniu — chwilowo nie ma czego rysować, ale obszar już istnieje.

Współrzędne Canvas są jak ekran, a nie jak Turtle
U `Canvas` punkt (0, 0) to lewy górny róg, a nie środek, jak na ekranie Turtle, a oś Y rośnie **na ziemię**, nie w górę. Rozdział 18.9 przyjrzy się temu bliżej i porówna oba układy współrzędnych bezpośrednio.



Point (60, 40): 60 px PRAWO i 40 px DÓŁ od lewego górnego rogu płótna.

Praktyka: Tworzenie Canvas
Moduł tkinter otwiera natywne okno Python — wykonaj lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/18-02/index.html)

Stwórz pierwsze menu (kształty)

Paska narzędzi z przyciskami wyboru kształtu — oraz zmiennymi zapisującymi aktualny stan.

Stwórz pierwsze menu (kształty)

Pasek narzędzi – normalny `Frame` (Rozdział 16) z guzikami, Każdy z nich wybiera własną pościąg:

menu_figur.py

```
toolbar = tk.Frame(root)
toolbar.pack(side="top", fill="x")

def vybrat_figuru(figura):
    global tekuschaya_figura
    tekuschaya_figura = figura

tk.Button(toolbar, text=„Linia”, command=lambda:
vybrat_figuru("linia")).pack(side="left")
tk.Button(toolbar, text=Prostokąt, command=lambda:
vybrat_figuru("pryamougolnik")).pack(side="left")
tk.Button(toolbar, text=„Oval”, command=lambda:
vybrat_figuru("oval")).pack(side="left")
```

Real Window: Pasek narzędzi z przyciskami Linia, Prostokąt, Owalny nad pustym płótnem

Prawdziwe okno: Paska taśm jest już widoczna, choć jeszcze nie wiemy, jak rysować kształty.

Zmuszamy parametry rysowania do działania!

Utwórzmy zmienne globalne dla bieżącego kształtu, koloru i grubości linii — będą je zmieniać przyciski paska narzędzi i czytać funkcje rysowania:

parametry.py

```
tekuschaya_figura = "linia"
tekuschij_cvet = "black"
tolschina = 3
```

Prototyp: zmienne globalne — nie jest to finalna architektura

Trzy zmienne globalne łatwo jest czytać i wyjaśniać, dopóki aplikacja jest mała — ten wybór już widzieliśmy w pierwszym prototypie kółko-krzyżyk (rozdział 17). Rozdział 18.13 pokaże, na co to się rozrasta, gdy przybywa narzędzi, działań i historii.

Praktyka: pasek narzędzi i opcje rysowania
Moduł tkinter otwiera natywne okno Python — wykonaj lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/18-03/index.html)

Pobieramy pozycję myszy

Wydarzenia Myszy pozwalają zamienić ruch kursora w prawdziwą linię na płótnie.

Pobieramy pozycję myszy

Podobnie jak w rozdziale 17, pomaga reagować na mysz `.bind()` — tylko zamiast zdarzeń klawiatury tutaj zdarzenia myszy: `<Button-1>` (naciska lewy przycisk), `<B1-Motion>` (ruch z użyciem zacisku lewym przyciskiem), `<ButtonRelease-1>` (przycisk został zwolniony).

pozyciya_myshi.py

```
def pokazat_pozyciyu(event):
    pozycia_label.config(text=f"x={event.x}, y={event.y}")

canvas.bind("<Motion>", pokazat_pozyciyu)
```

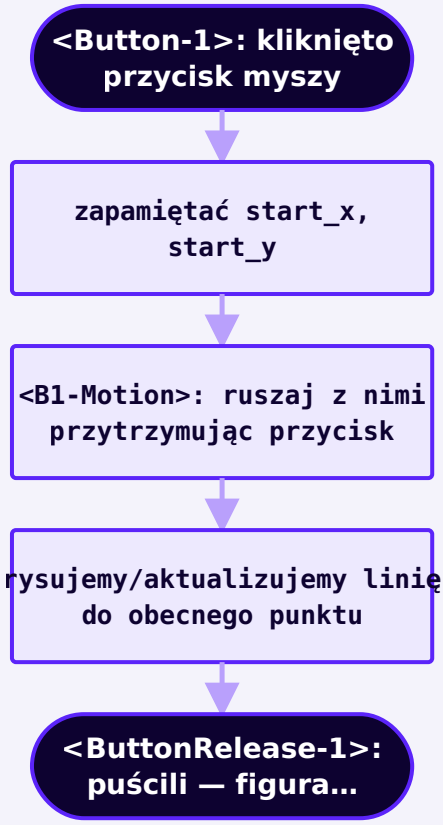
`event.x` oraz `event.y` — współrzędne kursorem względem płótna, w tych samych jednostkach co `Canvas`.

Rysujemy linie

Trzy zdarzenia razem tworzą efekt „rysowania od punktu do punktu” `<Button-1>`, narysuj linię do aktualnej pozycji na każdym z nich `<B1-Motion>`:

risuem_linii.py

```
start_x, start_y = None, None

def nachalo_risovaniya(event):
    global start_x, start_y
    start_x, start_y = event.x, event.y

def vo_vremya_risovaniya(event):
    canvas.create_line(start_x, start_y, event.x, event.y,
fill=tekuschij_cvet, width=tolschina)

canvas.bind("<Button-1>", nachalo_risovaniya)
canvas.bind("<B1-Motion>", vo_vremya_risovaniya)
```



Trzy wydarzenia razem tworzą gest „rysuj jednym ruchem”

Real Window: Kursor jest naciskany na początku linii, znak na płótnie ieszcze się nie pojawił

Real Window: Moment kliknięcia jest start_x/start_y pamiętany.

Rzeczywiste okno: gotowa linia prosta między punktem naciśnięcia a punktem puszczenia

Rzeczywiste okno: przycisk zwolniony — linia narysowana.

create_line — jak forward() w Turtle, tylko według współrzędnych

`canvas.create_line(x1, y1, x2, y2)` wyznacza granicę między dwoma punktami — koncepcyjnie tymi samymi jak `goto()` Turtle z rozdziału 6, tylko bez „żółwia”

Praktyka: pozycja myszy i rysowanie linii

Moduł tkinter otwiera natywne okno Python — wykonaj lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/18-04/index.html)

Kwadraty i prostokąty! Okręgi i owale!

Canvas potrafi rysować gotowe figury po dwóch kątach — nie tylko linie.

Prostokąty i owale rysowane są podobnie — Canvas potrafi budować je od razu z dwóch przeciwległych narożników, bez ręcznego obliczania boków:

pryamougolniki_ovaly.py

```
def vo_vremya_risovaniya(event):
    if tekuschaya_figura == "pryamougolnik":
        canvas.create_rectangle(
            start_x, start_y, event.x, event.y,
            outline=tekuschij_cvet, width=tolschina,
        )
    elif tekuschaya_figura == "oval":
        canvas.create_oval(
            start_x, start_y, event.x, event.y,
            outline=tekuschij_cvet, width=tolschina,
        )
```

Real Window: Prostokąt narysowany między punktem kliknięcia a punktem zwolnienia

Rzeczywiste okno: create_rectangle() w dwóch przeciwnych rogach.

Prawdziwe okno: Owal wyryty w obszarze między punktem ciśnienia a punktem uwolnienia

Prawdziwe okno: create_oval() na tych samych dwóch rogach – więcej o tym w sekcji 18.17.

Liczby pośrednie się mnożą

Jeśli rysujesz jak powyżej, każdy ruch myszy tworzy **nowe** kształt na poprzednim, zamiast taki, który rośnie wraz z ruchem. W pełnym programie (Rozdział 18.7) problem ten zostaje rozwiązany: poprzedni „przybliżony” `coords()` zamiast go odtwarzać.

Praktyka: prostokąty i owale

Moduł tkinter otwiera natywne okno Python — wykonaj lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/18-05/index.html)

Wybieramy rozmiar! Bardzo dużo kolorów!

Suwak grubości i wybór kolorów budowane przez pętlę na liście.

Wybierz rozmiar!

Widżet `Scale` — suwak do wyboru liczby w zakresie, świetnie nadaje się do grubości linii:

vybor_razmera.py

```
def vybrat_tolschinu(znachenie):
    global tolschina
    tolschina = int(znachenie)

tolschina_scale = tk.Scale(toolbar, from_=1, to=10,
orient="horizontal", command=vybrat_tolschinu)
tolschina_scale.set(3)
tolschina_scale.pack(side="left")
```

command w Scale otrzymuje wartość, a nie zdarzenie
W przeciwieństwie do `.bind()`, Handler `Scale` otrzymuje wartość końcowego łańcuch znaków suwaka bezpośrednio — więc `vybrat_tolschinu(znachenie)` natychmiast zamienia to w liczbę przez `int()` (Rozdział 4).

Prawdziwe płótno: Cztery linie tego samego koloru o grubości 1, 3, 8 i 16 pikseli, oznaczone cyframi

Rzeczywiste płótno: ta sama linia przy różnej grubości — liczby w kodzie zamieniają się w konkretną różnicę na ekranie.

Dużo kwiatów!

Paletę kolorów można łatwo zbudować pętlą (rozdział 10) po liście nazw (rozdział 11) — zamiast tworzyć każdy przycisk ręcznie:

vybor_cveta.py

```
cveta = ["black", "red", "blue", "green", "orange", "purple"]
for cvet in cveta:
    tk.Button(toolbar, bg=cvet, width=2,
command=lambda c=cvet: vybrat_cvet(c)).pack(side="left")
```

Real Window: Pasek narzędzi z serią kolorowych kwadratowych przycisków palety

Rzeczywiste okno: sześć przycisków-kwadratów, każdy w kolorze przycisku bg=cvet — paleta widoczna wcześniej niż efekt rysowania nią.

lambda c=cvet to ta sama subtelność co w rozdziale 17
Bez `c=cvet` wszystkie przyciski palety wybierały **ostatni** kolor z listy to ta sama pułapka z lambdami wewnątrz cyklu, którą już przeanalizowaliśmy w projekcie kółko-krzyżyk.

Praktyka: Scale i paleta kolorów
Moduł tkinter otwiera natywne okno Python — wykonaj lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/18-06/index.html)

Skończyłem rysować! Pierwszy kompletny program

Ostatnie szlify pierwszego prototypu — czyszczenie płótna i swobodne rysowanie — i działający program w całości.

Skończyłem rysować!

Pozostaje dodać przycisk czyszczenia płótna oraz tryb „wolnego rysowania”

ochistka_i_svobodno.py

```
def ochistit_holst():
    canvas.delete("all")

def vo_vremya_risovaniya(event):
    if tekuschaya_figura == "svobodno":
        canvas.create_oval(
            event.x - tolschina, event.y - tolschina,
            event.x + tolschina, event.y + tolschina,
            fill=tekuschij_cvet, outline=tekuschij_cvet,
        )
        return
    # ... pozostałe kształty jak w sekcji 18.5
```

Prawdziwe płótno: pociągnięcie pojedynczych kółek z wyraźnymi przerwami między nimi

Prawdziwe płótno: okrąg dla każdego ruchu myszy – ostrym gestem można zobaczyć przerwy między nimi.

Oddzielne okręgi nie są jedną linią

Taki ruch nie rysuje linii ciągłej, lecz rozpraszanie małych kółek-kropek, które czasem nawet się nie stykają, jeśli mysz szybko przesuwa się między dwoma zdarzeniami `<B1-Motion>`. Rozdział 18.14 pokazuje, jak uzyskać prawdziwy ciągły rys – łącząc każdą nową kropkę z poprzednią.

Pierwszy Kompletny Program

Pełna wersja pierwszego prototypu, w tym poprawione rysowanie „szkicowych” figur podczas przeciągania myszą, jest dostępna jako osobny plik:

projects/tkinter/paint-app/paint_app_basic.py

Prawdziwe okno: płótno z kilkoma liniami, prostokątem i owalnym kolorem – pierwszy prototyp gry rysującej

Rzeczywiste okno pierwszego prototypu: kilka figur w różnych kolorach na jednym płótnie.

Szczera, ale nie ostateczna wersja

Stworzyliśmy prawdziwy działający program — z płótnem, kształtami, kolorem i grubością. Ale stan tutaj nadal istnieje w zmiennych globalnych, nic nie da się "cofnąć", a "przybliżony" kształt jest po prostu usuwany i odtwarzany podczas przeciągania i upuszczania. Zaczynając od następnej sekcji, rozdział znacznie dokładniej przygląda Canvas się: co faktycznie przechowuje, jak cofnąć akcję na rzeczywiste podstawy oraz jak wyhodować aplikację poziomu z tego prototypu `PaintApp` (Rozdział 18.32).

★★ Zadanie samodzielne

Przycisk anulowania

Zapisz id każdy narysowany kształt (canvas.create_* zwraca id) do lista — i dodaj przycisk, który usuwa ostatni po canvas.delete(id). Rozdział 18.24 rozbije ten pomysł jako prawdziwą architekturę.

Challenge

Zapisywanie zdjęcia

Przestudiuj moduł tkinter.filedialog i spróbuj dodać przycisk „Zapisz” — przynajmniej zapisujący lista narysowane figury do pliku tekstowego (rozdział 15). Rozdział 18.30 pokaże pełne rozwiązanie przez JSON.

Praktyka: Pierwszy kompletny program

Moduł tkinter otwiera natywne okno Python — wykonaj lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/18-07/index.html)

Streszczenie sekcji

Czego się nauczyliśmy

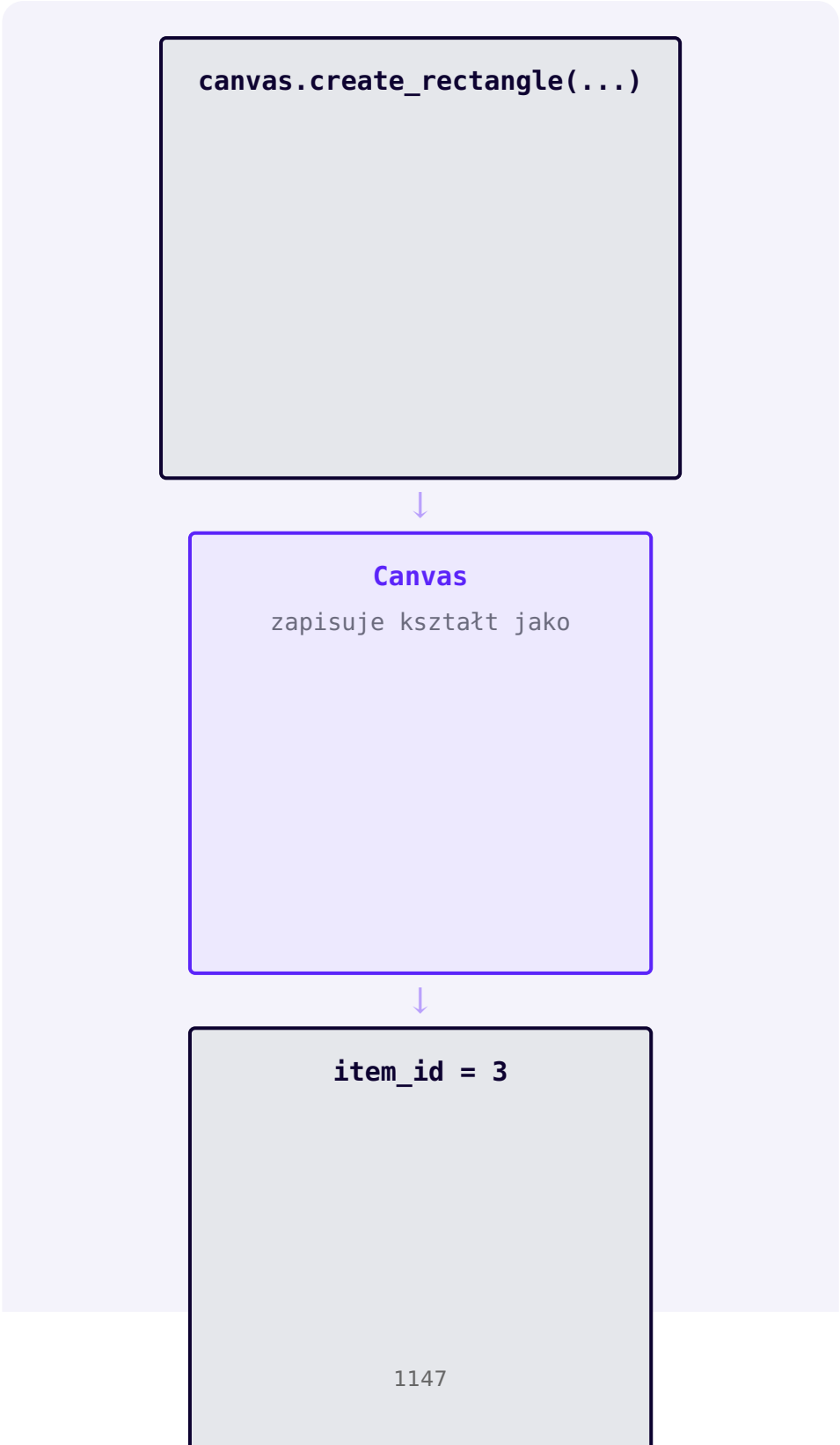
- `Canvas` — widżet Tkinter do rysowania linii i figur według współrzędnych; współrzędne rosną w prawo i w dół od lewego górnego rogu.
- `canvas.create_line/rectangle/oval(...)` rysują gotowe figury według współrzędnych.
- Wydarzenia Myszy (`<Button-1>` , `<B1-Motion>` , `<ButtonRelease-1>`) śledzić kliknięcie, przeciąganie i puszczanie przycisku myszy.
- `Scale` — suwak do wyboru liczby w zakresie.
- Paleta z kilkoma podobnymi przyciskami jest efektywniej tworzona w pętli niż ręcznie jeden po drugim.
- To dopiero pierwszy prototyp — wtedy rozdział znacznie dokładniej analizuje Canvas i architekturę aplikacji.

Canvas przechowuje obiekty, a nie piksele

create_line() nie tylko rysuje — dodaje element do lista, który Canvas pamięta siebie.

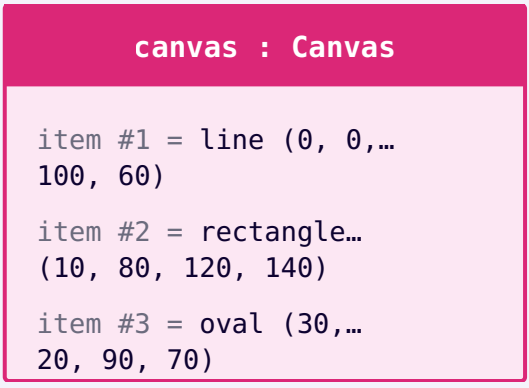
Canvas nie «maluje pikseli» — przechowuje obiekty

Pierwszy prototyp już rysuje, ale łatwo to sobie źle wyobrazić: jakby `create_line()` po prostu maluje piksele na zdjęciu jak pędzel w edytorze graficznym. W rzeczywistości Canvas działa inaczej — i prawie wszystko od tego zależy Reszta w tym rozdziale: cofanie akcji, edycja kształtów, kolejność stosowania.


canvas.create_rectangle(...)
Canvas
zapisuje kształt jako
item_id = 3
1147

create_* nie tylko rysuje — dodaje ELEMENT do lista, który Canvas sam się przechowuje.

Zwrócona liczba — **item ID**, identyfikator danego elementu wewnątrz Canvas (Rozdział 18.10). Dopóki figura nie zostanie wyraźnie usunięta, Canvas ją zapamięta i może Przerysuj, przesuwaj lub zmieniaj — nawet jeśli twój kod dawno zakończył pracę z tym systemem.



Canvas przechowuje LISTĘ elementów — mniej więcej tak, jakby miał swoje wewnętrzne „obiekty”

Rzeczywiste okno: płótno z trzema narysowanymi figurami — linią, prostokątem i owalu

To samo prawdziwe okno co Rozdział 18.7: trzy widoczne kształty – trzy przechowywane przedmioty.

Trzy różne warstwy – nie myl ich

Warstwa	Co to jest
Model aplikacji (Python)	Twoje zmienne i struktury danych to coś, co sam wymyśliłeś
Element Canvas (item)	Zapis wewnątrz Canvas z typem, współrzędnymi i parametrami — tworzony przez create_*
Piksele na ekranie	Ostatni obraz, który widzi użytkownik, jest wynikiem narysowania elementu

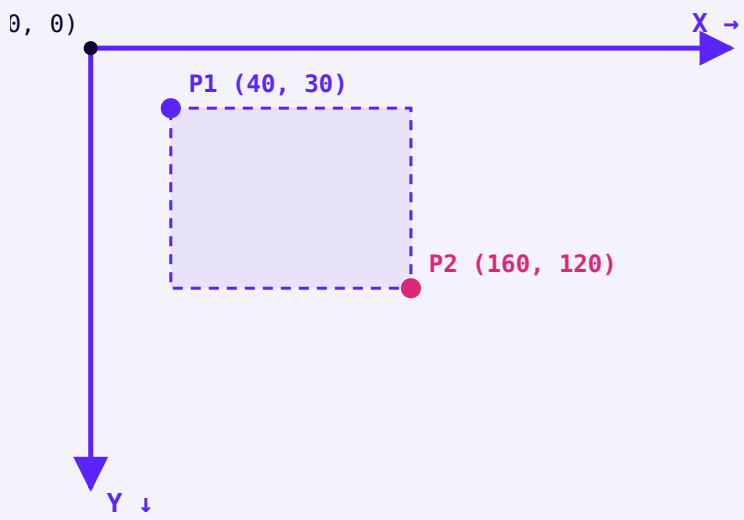
item ID NIE jest identyfikatorem obiektu Python- `item_id = canvas.create_rectangle(...)` zwraca regularną liczbę całkowitą, która jest wewnętrzną liczbą rekordu w obrębie Canvas, zamiast `id()` jakiś Python- obiekt, a nie indeks z listy, którą sam prowadzisz. Canvas daje liczby według własnych zasad (zwykle rosnących) — nie powinieneś polegać na konkretnych wartościach, a jedynie na fakcie, że każdy `item_id` unikalny dla jednego elementu.

Układ współrzędnych Canvas

Lewy górny róg to początek. X rośnie w prawo, Y schodzi w dół. Nie jak Turtle.

Współrzędne rosną w prawo i w dół

Canvas punkt `(0, 0)` to lewy górny róg. Pierwsza współrzędna (X) rośnie w prawo, drugi (Y) - w dół.



Prostokąt w dwóch przeciwległych narożnikach P1 i P2 dokładnie to, co create_rectangle(P1x robi, P1y, P2x, P2y).

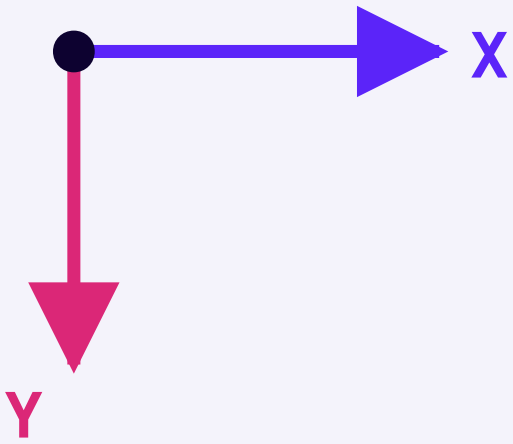
Prawdziwe okno: Płótno z oznaczonymi punktami i linią między nimi, pokazującą wznoszenie współrzędnych w prawo i w dół

Rzeczywiste okno: dwa punkty i linia między nimi — ruch myszy w prawo zwiększa x, ruch w dół zwiększa y.

Turtle vs Canvas — porównajmy to bezpośrednio.

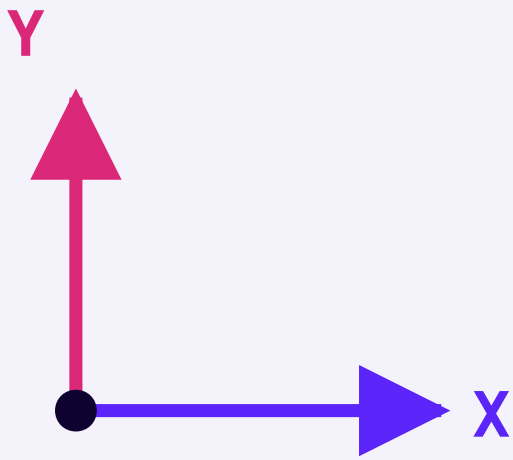
Jeśli przed tym rozdziałem narysowałeś Turtle (rozdziały 6-7), twoja intuicja może zawiodować: jest początek współrzędne są środkiem ekranu, a Y rośnie ku górze, jak w matematyce na papierze. Canvas ma inny kształt, oraz Nie jest to system "bardziej poprawny" czy "mniej poprawny", lecz po prostu inne porozumienie, które Tkinter odziedziczone po systemach okiennych, gdzie linie ekranu są numerowane od góry do dołu.

Canvas (Tkinter)



Origin to lewy górny róg, Y rozciąga się w dół

Turtle (rozdziały 6–7)



Początek współrzędnych — środek ekranu, Y rośnie W GÓRĘ

event.x / event.y — te same współrzędne, co sam Canvas
Współrzędne zdarzenia myszy (Rozdział 17 – `event.x`, `event.y`) oraz współrzędnymi, które `create_*` to ten sam system: piksele względem lewego górnego rogu płótna. Dlatego `canvas.create_line(start_x, start_y, event.x, event.y)` działa bez żadnych przeliczeń.

Praktyka: współrzędne Canvas
Automatyczna kontrola — określ kierunek ruchu kursora na podstawie współrzędnych Y

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/18-09/index.html)

Item ID: co wraca create_*

Liczba, którą create_* zwraca — to nie jest pusta formalność: jest to sposób, aby później wrócić do figury.

create_* zwraca identyfikator

Każde wyzwanie `canvas.create_line/rectangle/oval(...)` zwroty Liczba całkowita — **item ID** właśnie utworzonego elementu. Jeśli go nie zapiszesz, figura i tak się narysuje, ale później nie będzie można się do niej odwołać:

item_id.py

```
item_id = canvas.create_rectangle(10, 10, 100, 60,
outline="blue")
print(item_id)   #, na przykład 1 — konkretna liczba zależy od tego, ile przedmiotów zostało już utworzonych
```

Ta liczba przydaje się w trzech kwestiach, które omówimy później w rozdziale:

Po co pamiętać item_id

COORDS(ITEM_ID, ...)

zmienić współrzędne istniejącego przedmiotu

Rozdział 18.18–18.19

ITEMCONFIG(ITEM_ID, ...)

zmienić kolor/grubość bez ponownego tworzenia
Rozdział 18.19

DELETE(ITEM_ID)

usunąć ten konkretny element, nie wszystkie pozostałe
sekcje 18.19, 18.24

item_id nie jest indeksem listy ani id() obiektem Python
Pokusa, by sądzić, że `item_id` pozycja na liście ("pierwszy kształt to id 0", "drugi to id 1") jest wadliwy: Canvas daje liczby według własnych reguł i nie musi zaczynać od zera ani nie robić przerw po usunięciu elementów. Nie polegaj na konkretnych wartościach, tylko na tym, że każdy `item_id` unikalnie odnosi się do jednego elementu.

Praktyka: item ID
Automatyczna kontrola — przechowujemy figury po item_id jako klucz słownika, a nie według indeksu listy

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/18-10/index.html)

Tagi: Grupuj i wybieraj elementy

item_id wskazuje na jeden element. Znacznik może stać od razu na wielu — i działać z grupą jedną komendą.

Tag to nazwa, a nie liczba

`item_id` wskazuje na JEDEN konkretny przedmiot. Ale często musisz odwołuj się do kilku naraz, na przykład do wszystkich narysowanych kształtów naraz, bez dotykania tymczasowa „wstępna”

tegi.py

```
canvas.create_line(
    0, 0, 100, 60,
    fill="black", tags=("shape", "stroke"),
)
```

Jeden element może mieć jednocześnie kilka tagów. Ponadto tag może być nazywany zamiast `item_id` prawie wszędzie Canvas oczekuje adresu elementu:

operacii_s_tegami.py

```
canvas.delete("preview")           # usuń wszystkie elementy z tagiem „preview”
canvas.itemconfig("shape", width=2) # zmienia grubość WSZYSTKICH elementów oznaczonych jako „shape”
```

item_id kontra tag

ITEM_ID

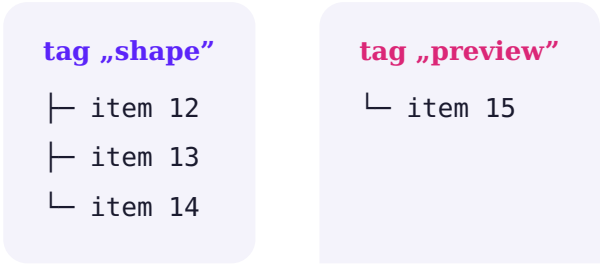
wskazuje na JEDEN konkretny element

liczba, którą zwraca create_*

TAG

może opierać się na wielu elementach jednocześnie

łańcuch znaków wymyślisz



W finalnej aplikacji tagi przydadzą się dwukrotnie

Rozdział 18.20 używa tagów do pokazywania kolejności, w jakiej kształty się nakładają. Element podglądu szkicu (Rozdział 18.18) otrzymuje własny tag `"preview"` tak, aby jedna drużyna mogła go usunąć, nie przechodząc przez wszystkie pozostałe elementy.

Praktyka: Tagi Canvas

Automatyczne sprawdzanie — zbieranie grup elementów według tagów, gdzie jeden tag mieści kilka id jednocześnie

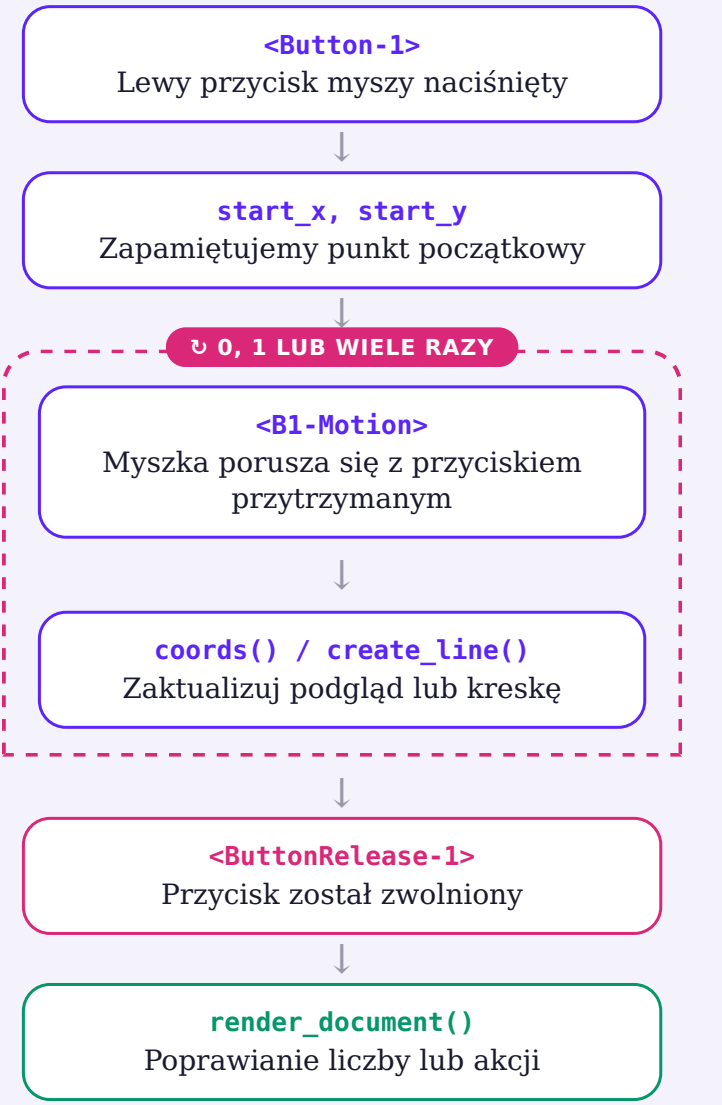
Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/18-11/index.html)

Cykl życia gestu myszy

Rysowanie jednej figury to sekwencja trzech różnych wydarzeń, a nie jedno.

Jeden gest, trzy wydarzenia

Narysowanie pojedynczego kształtu myszką to nie jedno zdarzenie, lecz ciąg trzech różnych Typy, które razem tworzą gest „naciągnij”



press → drag (0, 1 lub wiele razy) → release — ten sam gest dla ołówka, linii, prostokąta i owalu.

Wydarzenie	Kiedy się uruchamia
`<Button-1>`	dokładnie raz — w momencie naciśnięcia lewego przycisku
`<B1-Motion>`	wiele razy pod rząd, gdy przycisk jest przytrzymywany, a mysz porusza się
`<ButtonRelease-1>`	dokładnie raz, gdy przycisk zostanie zwolniony

<Motion> NIE jest <B1-Motion>
`<Motion>` wywołuje się przy JAKIMKOLWIEK ruchu myszy nad płótnem — nawet jeśli nie naciśnięto żadnego przycisku. `<B1-Motion>` — tylko przy ruchu z wciśniętym LEWYM przyciskiem. Pomylenie ich to częsta przyczyna, dla której rysowanie zaczyna się bez kliknięcia (rozdział 18.31, Debug Lab).

Wszystkie pożądane współrzędne przechodzą przez ten sam obiekt `Event` to a w rozdziale 17 — `event.x` , `event.y` . `event.widget` jest również dostępny, choć zwykle jest dostępny na pojedynczym płótnie Tak oczywiste.

Klik bez przeciągania — nie zapomnij o tym przypadku

Co jeśli użytkownik nacisnął i natychmiast puścił przycisk w pewnym momencie, nie poruszając myszką? `<B1-Motion>` w tym przypadku nie wydarzy się ani raz, wydarzenie uruchomi się tylko wtedy, gdy mysz faktycznie się poruszy. Ostateczne zastosowanie (Rozdział 18.32) Wyraźnie to sprawdza: jeśli punkt startowy i punkt zwolnienia są prawie takie same, to linia, prostokąt Albo po prostu nie powstaje owal – zamiast niewidzialnej figury o zerowym rozmiarze.

Praktyka: Gest myszy press → drag → release
Moduł tkinter otwiera natywne okno Python — wykonaj lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/18-12/index.html)

Stan rysowania

Narzędzie, kolor, grubość i punkt rozpoczęcia gestu — to parametry, a nie same figury.

Co program powinien zapamiętać między wydarzeniami

Trzy zmienne z Sekcji 18.3 (`tekuschaya_figura`, `tekuschij_cvet`, `tolschina`) — uczciwy prototyp, ale wraz z rozwojem aplikacji dodaje się punkt wyjścia gestu, Ostatni punkt ołówka, id przybliżony wykres podglądu, oraz zmienne globalne pochodzą na zbyt wielu, by pamiętać o ich więzi.

Jeden obiekt zamiast ośmiu osobnych zmiennych — te same dane, ale z czytelną granicą.

Nowość: Enum — wcześniej znany lista wariantów

Podczas gdy narzędzie jest przechowywane na sznurku, **żadnych** słowo wydaje się akceptowalne: `"pryamougolnik"`, `"pryamougolnick"`, `"квадрат"`. Python nie ma nic przeciwko — nie wie, jakie masz opcje Myślisz, że mają rację. Literówka pojawi się dużo później i w ogóle nie będzie obecna: znaczenie po prostu nie jest będzie się pokrywać z dowolną gałęzią `if` i klikanie myszką cicho się zatrzymuje żeby coś narysować.

Ale narzędzie ma ograniczoną, wcześniej znaną liczbę opcji — ołówek, linia, prostokąt, elipsa, gumka. Dla takich przypadków w Python jest **wyliczanie** (`Enum` z modułu `enum`): ty raz zapisz wszystkie dopuszczalne wartości, a następnie używaj tylko ich.

enum_vpervye.py

```python
from enum import Enum

class Tool(Enum):
    PENCIL = "pencil"
    LINE = "line"
    RECTANGLE = "rectangle"

print(Tool.RECTANGLE)          # Tool.RECTANGLE
print(Tool.RECTANGLE.value)    # rectangle
print(Tool.RECTANGLE is Tool.PENCIL)   # False
print(Tool.RECTANGL)
#AttributeError — literówka jest widoczna NATYCHMIAST
```

Trzy rzeczy, o których warto pamiętać `Enum`:

- `Tool.RECTANGLE` jest **członek** transferów, osobna wartość, nie łańcuch znaków. Łatwo jest ją przenosić i przechowywać;
- `.value` wyciąga linię, którą do niej przypiąłeś — to Przyda się, gdy trzeba zapisać postać w JSON (sekcja 18.30);
- terminy się porównoważają `is`, nie `==`: wyraz enumeracyjny istnieje dokładnie w jednym przypadku, Zatem `tool is Tool.ERASER` — najdokładniejsza i najszybsza weryfikacja.

Główna różnica w stosunku do łańcucha: `Tool.RECTANGL` jest `AttributeError` przy pierwszym wywołaniu, a `"pryamougolnick"` — całkowicie „normalny” łańcuch znaków, o którym Python milczy.

Zbieramy DrawingState

Teraz połączmy oba narzędzia: `Enum` dla narzędzia i `@dataclass` (Rozdział 14.23) dla samego obiektu państwowego — Usuwa to odręczne `__init__` i od razu daje zrozumiałe `repr`:

drawing_state.py

```python
from dataclasses import dataclass
from enum import Enum

class Tool(Enum):
    PENCIL = "pencil"
    LINE = "line"
    RECTANGLE = "rectangle"
    OVAL = "oval"
    ERASER = "eraser"

@dataclass
class DrawingState:
    tool: Tool = Tool.PENCIL
    color: str = "#111827"
    width: int = 4
    start_x: float | None = None
    start_y: float | None = None
```

Enum zamiast kwestii – gdy warto

`tekuschaya_figura = "pryamougolnik"` działa, ale nie zauważy Python literówki w linii, dopóki kod nie zostanie wykonany i kształt nie zostanie cicho „nieznany” `Tool.RECTANGLE` jest takie samo pod względem znaczenia, ale literówka zostanie wykryta przez redaktora lub `python -c "import module"` . Enum jest tu uzasadnione właśnie dlatego, że warianty instrumentu mają skończoną znaną liczbę. Jeśli istnieje tylko jeden wariant lub ich lista ciągle się zmienia, wystarczy regularna linia, a wyliczenie doda tylko ceremonię.

Prawdziwe okno: przycisk „Ołówek” na pasku narzędzi wygląda na wciśnięty (zanurzony), pozostałe — zwykłe

Prawdziwe okno: Ołówek jest wybrany — przycisk jest wizualnie „cofnięty”

Prawdziwe okno: Przycisk linii wygląda na naciśnięty (cofnięty), przycisk ołówka wraca do normalnego stanu

Prawdziwe okno: przełączono się na Linię — poprzedni przycisk wrócił do zwykłego widoku, nowy „zanurzony”.

Wybrane narzędzie jest widoczne, a nie tylko przechowywane w zmiennej

Stan — nie tylko to, co pamięta Python. Użytkownik powinien WIDZIEĆ, który narzędzie jest teraz wybrane, bez patrzenia w kod. W aplikacji końcowej `set_tool()` zmienia się jednocześnie `self.state.tool` I `relief` potrzebny przycisk — ta sama kombinacja «model → odzwierciedlenie na ekranie», co i `render()` w rozdziale 17.

Ścieżka tego rozdziału

Praktyka: Symulacja stanu rysowania
Automatyczne sprawdzanie – wybór narzędzi i parametrów bez Tkinter
Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/18-13/index.html)

Ołówek: ciągła kreska

Oddzielne okręgi tworzą przerwy. Połącz każdy punkt z poprzednim — a kreska staje się jedną linią.

Dlaczego kółka przy każdym ruchu — zły pomysł

Rozdział 18.7 wylosowała wolny styl z kółkami, po jednym dla każdej konkurencji `<B1-Motion>` Działa, ale to podejście ma trzy problemy:

- między kółkami pozostają przerwy, jeśli mysz poruszała się szybciej niż nadchodziły zdarzenia;
- sama linia wygląda „wyboiście”, a nie gładko;
- krąg nie wie nic o tym, gdzie przed chwilą był kursor, o ścieżce między nimi po prostu ginie przez sąsiednie zdarzenia.

Prawdziwe płótno: pociągnięcie pojedynczych kółek z wyraźnymi przerwami między nimi

To samo rzeczywiste okno z sekcji 18.7: rozpraszanie okręgów zamiast linii.

Najlepszy model: Pamiętaj o poprzednim punkcie

Zamiast okręgu dla każdego zdarzenia, połącz każdy nowy punkt z poprzednim segmentem prosta. Jest wiele odcinków, ale razem tworzą jedną ciągłą linię:

karandash.py

```
def on_drag(self, event):
    self.canvas.create_line(
        self.state.last_x, self.state.last_y, event.x, event.y,
        fill=self.state.color, width=self.state.width,
        capstyle=tk.ROUND,
    )
    self.state.last_x, self.state.last_y = event.x, event.y
```

Stąd kod — to metody klasy PaintApp

Począwszy od tej sekcji, przykłady są opisane jako metody: `def on_drag(self, event)`, `self.state`, `self.canvas`. Wszystkie należą do `PaintApp` w całości omówimy w sekcji 18.29 — do tego czasu wystarczy przeczytać `self.state.color` jako „aktualny kolor aplikacji”

Prawdziwe płótno: Ciągła, płynna linia bez przerw, która podąża za ruchem myszy

Prawdziwe okno ostatecznej wersji: ta sama ręka, ale linia bez przerw — ponieważ rysowane są segmenty, a nie pojedyncze punkty.

Parametr	Dlaczego
`capstyle=tk.ROUND`	zaokrągla końce każdego segmentu – połączenia między nimi nie wyglądają na „poszarpane”
`width`	grubości ruchu; Im większy, tym bardziej widoczne byłoby „poszarpanie”

A co z smooth=True? Tutaj nie zadziała

U `create_line()` jest parametrem `smooth=True` — Tk wygładza linię łamaną, prowadząc przez jej punkty krzywą. Ale wygładzać potrafi tylko **wewnętrzny** punkty jednej linii, a każdy z naszych odcinków składa się dokładnie z DWÓCH punktów — nie ma między nimi ani jednego rogu. Dodaj `smooth=True` w ten kod można, i Tk go milcząco zaakceptuje, ale na ekranie nie zmieni się dokładnie nic. Parametr byłby przydatny, gdyby cały kontur był JEDNĄ długą łamaną — taki wariant opisano poniżej.

Dokąd to można rozwinąć: jeden pociągnięcie — jedna łamana

Teraz każdy ruch myszy tworzy osobny element Canvas. Możesz zamiast tego utworzyć linię raz na `<Button-1>` i dodawaj nową parę współrzędnych przy każdym ruchu `coords()` (rozdział 18.18): wtedy długa kreska to jeden element zamiast setki, a `smooth=True` w końcu otrzymuje wewnętrzne punkty, które faktycznie można wygładzić. W tym rozdziale pozostała prostsza wersja z oddzielnymi odcinkami — jest krótsza i łatwiejsza do debugowania.

Praktyka: Ciągłe pociągnięcie ołówkiem

Moduł tkinter otwiera natywne okno Python — wykonaj lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/18-14/index.html)

Narzędzie liniowe

Kropka → szkic, który aktualizuje się na bieżąco → ostatnią pełną linią.

Linia to nie to samo co ołówek

Ołówek nie ma znanego końca: kreska kończy się tam, gdzie użytkownik ją zwalnia przycisk, a linia jest budowana z wielu małych segmentów na bieżąco. Narzędzie Linia ma wszystko w przeciwnym razie jest dokładnie JEDNA prosta linia od punktu nacisku do punktu uwolnienia i do samego końca gestu To tylko szkic, który można wielokrotnie zmieniać.

instrument_liniya.py

```
def on_press(self, event):
    self.state.start_x, self.state.start_y = event.x, event.y
    self.state.preview_id = self.canvas.create_line(
        event.x, event.y, event.x, event.y,
        fill=self.state.color, width=self.state.width,
dash=(4, 2),
    )

def on_drag(self, event):
    self.canvas.coords(
        self.state.preview_id,
        self.state.start_x, self.state.start_y, event.x,
event.y,
```

```
    )

def on_release(self, event):
    self.canvas.delete(self.state.preview_id)
    self.canvas.create_line(
        self.state.start_x, self.state.start_y, event.x,
event.y,
        fill=self.state.color, width=self.state.width,
    )
```

Prawdziwe okno: Przerywana linia podglądu rozciąga się od punktu kliknięcia do aktualnej pozycji kursora

Rzeczywiste okno: podczas przeciągania — przerywana linia robocza, jeszcze nie finalna.

Real Window: Solidna ostatnia linia zamiast poprzedniego przerywanego zapowiedzi

Prawdziwe okno: przycisk zostaje zwolniony — ciągła linia zastąpiła przerywany szkic.

dash=(4, 2) — sygnał wizualny „to jeszcze nie jest finał”
Kreskowanie w podglądzie — to nie wymóg techniczny Canvas, lecz świadomy wybór: użytkownik powinien gołym okiem odróżnić szkic od gotowego kształtu. Rozdział 18.18 uogólnia tę metodę „utwórz podgląd → zaktualizuj coords() → zatwierdź w release” dla prostokąta i owalu — z linią działa dokładnie tak samo.

Praktyka: Narzędzie linii z podglądem
Moduł tkinter otwiera natywne okno Python — wykonaj lokalnie w VS Code, PyCharm lub Jupyter

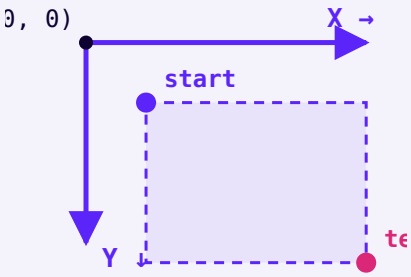
Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/18-15/index.html)

Geometria prostokąta

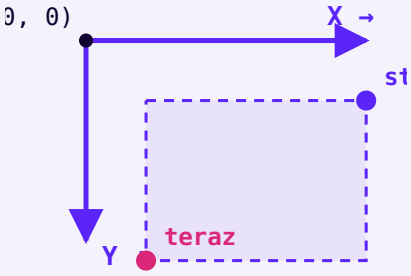
Myszką można przeciągać w dowolnym z czterech kierunków – prostokąt powinien być taki sam.

Dwa kąty, cztery możliwe kierunki

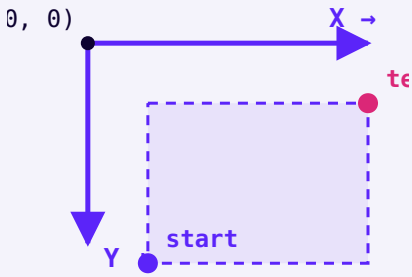
`canvas.create_rectangle(x1, y1, x2, y2)` rysuje prostokąt między dwoma punktami — ale użytkownik może przeciągnąć myszkę w dowolnym kierunku od punktu startowego: w dół-prawo, w dół-lewo, w górę-prawo, w górę-lewo. Prostokąt jest taki sam — Tk ustalić, która współrzędna jest większa:



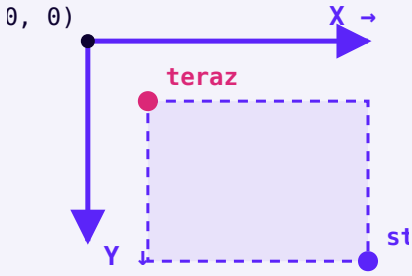
w dół-w prawo



w dół-w-lewo



w górę i prawo



w górę-w-lewo

Ale jeśli Twój kod potrzebuje też współrzędnych, na przykład do zapisu kształtu w dokumencie (Rozdział 18.13) z przewidywalnym porządkiem punktów, warto wyraźnie zredukować je do tej samej formy:

normalize_bounds.py

```
def normalize_bounds(x1, y1, x2, y2):

"""Przekształca dwa dowolne przeciwne punkty do (left, top,
right, bottom)."""
    return min(x1, x2), min(y1, y2), max(x1, x2), max(y1, y2)
```

Canvas robi to sam, a normalize_bounds() robi to dla twojego kodu

`create_rectangle()` idealnie rysuje prostokąt, nawet jeśli `x1 > x2` – sam Tk nie gubi się w kierunku. `normalize_bounds()` nie musisz Canvas, lecz swój model dokumentu: ten sam kształt, narysowany w dowolnym kierunku, jest zapisany w JSON (sekcja 18.30) w tej samej przewidywalnej formie.

Praktyka: normalize_bounds() dla wszystkich kierunków przeciągania i upuść

Automatyczna kontrola — przynieś współrzędne do (left, top, right, bottom) bez Tkinter

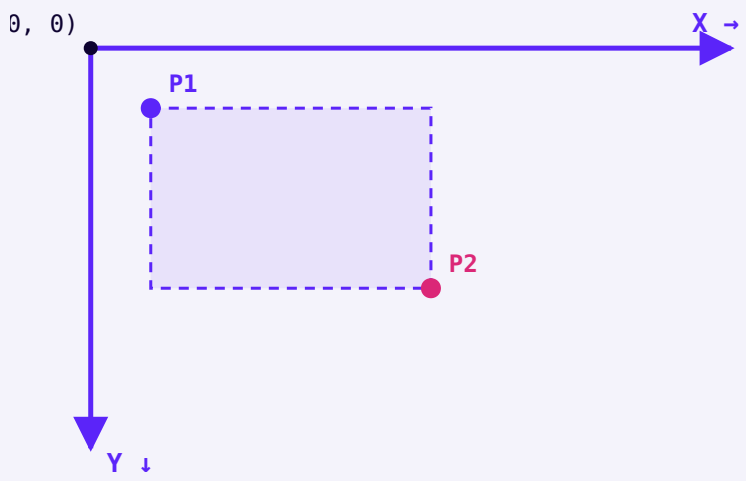
Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/18-16/index.html)

Geometria owalna

create_oval przyjmuje te same cztery współrzędne co prostokąt — owal jest wpisany w niewidzialne granice.

Owal — to prostokąt z zaokrąglonymi rogami do granic

`canvas.create_oval(x1, y1, x2, y2)` bierze dokładnie te same cztery liczby, że i `create_rectangle()` — ale nie rysuje prostokąta, ale elipsa, **wpisany** w to wciągnąć. Prostokąt nie jest narysowany — po prostu określa granice, w których zbudowany jest owal.



Przerywany prostokąt to te same współrzędne, które przekazałeś do create_oval(); sam owal jest wbudowany w jego wnętrzu.

Rzeczywiste okno: owal starannie wpisany w niewidoczny prostokątny obszar między dwoma punktami

Prawdziwe okno: create_oval(x1, y1, x2, y2) — owal dotyka wszystkich czterech stron niewidocznego prostokąta.

Nie środek + promień, lecz ramkę ograniczającą

Canvas nie ma osobnego `create_circle()` : koło — to po prostu elipsa, której ograniczający prostokąt jest kwadratem (`x2 - x1 == y2 - y1`). Jeśli aplikacja nadal wymaga „środek + promień” `x1, y1 = cx - r, cy - r` oraz `x2, y2 = cx + r, cy + r` .

Praktyka: Owalny prostokąt ograniczający

Automatic check — przetłumacz współrzędne między „centrum + promień”

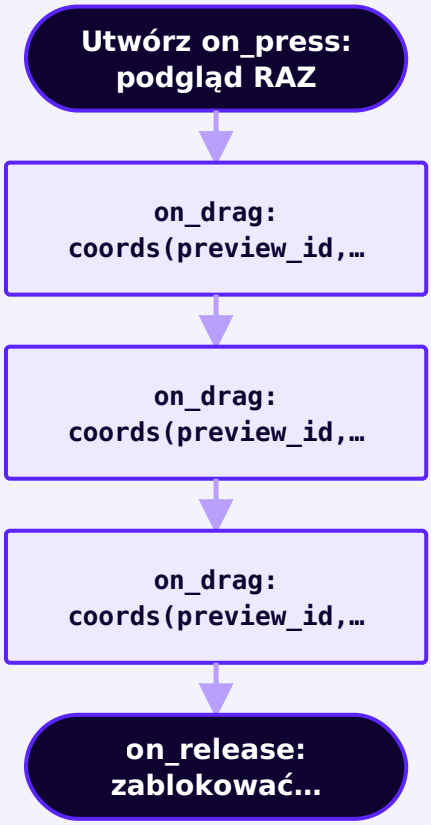
Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/18-17/index.html)

Zapowiedź na żywo: coords()

Pojedynczy element jest tworzony raz i aktualizowany wielokrotnie, zamiast strumienia jednorazowych kształtów.

Nie odtwarzaj kształtu dla każdego ruchu

Naiwne podejście do podglądu — na każdej `<B1-Motion>` usuń starą szkicową figurę i stwórz nową. Działa, ale zmusza Canvas do bezustannego wyrzucania i ponownego tworzenia elementów — a przy okazji traci wszelkie indywidualne ustawienia, które ustawiłeś może tylko raz.



Utwórz raz → aktualizuj wiele razy → zatwierdź (CREATE ONCE → UPDATE MANY TIMES → COMMIT) — ten sam element, a nie strumień jednorazowych.

zhivoe_prevyu.py

```
def on_press(self, event):
    self.state.preview_id = self.canvas.create_rectangle(
        event.x, event.y, event.x, event.y,
        outline=self.state.color, width=self.state.width,
dash=(4, 2),
    )

def on_drag(self, event):
    self.canvas.coords(
        self.state.preview_id,
        self.state.start_x, self.state.start_y, event.x,
event.y,
    )
```

Rzeczywiste okno: przerywany prostokąt-podgląd w połowie przeciągania

Prawdziwe okno: Ten sam element, preview_id co coords() zmienia rozmiar – nie jest odtwarzany.

Rzeczywiste okno: przerywana owalna podglądówka w połowie przeciągania

Prawdziwe okno: ten sam pomysł działa w przypadku owalnego – zmienia się tylko kształt, który rysuje create_oval().

coords() akceptuje tyle liczb, ile potrzebuje figura

Dla linii i prostokąta cztery liczby (dwa punkty). Dla wielopunktowej polilinii (na przykład rosnący kontur ołówkiem), tyle par chcesz `x, y`. Najważniejsze to sprawa `coords()` z tym samym `item_id`, który zwrócił oryginalny `create_*`.

Praktyka: Zapowiedź na żywo przez coords()

Moduł tkinter otwiera natywne okno Python — wykonaj lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/18-18/index.html)

itemconfig, move, delete

Element Canvas nie musi być odtwarzany — można go modyfikować na miejscu.

Cztery sposoby odwołania się do już narysowanego

Metoda	Co się zmienia
`canvas.coords(id, ...)`	współrzędne elementu — kształt i położenie
`canvas.itemconfig(id, ...)`	parametry wizualne — kolor, grubość, styl — bez zmiany kształtu
`canvas.move(id, dx, dy)`	przesuwa pierwiastek o dx dy — zamiast ustalać absolutną pozycję
`canvas.delete(id)`	trwałe usuwa pierwiastek

redaktirovanie.py

```
item_id = canvas.create_rectangle(10, 10, 80, 60,
outline="black", width=2)

canvas.itemconfig(item_id, outline="blue", width=4)
# ten sam prostokąt, inny widok
canvas.move(item_id, 30, 0)
# przesuń 30px w prawo
canvas.coords(item_id, 10, 10, 150, 100)                #
zmieniać same granice
canvas.delete(item_id)                                 #
całkowicie usuń
```

move(id, dx, dy) jest przesunięciem, a nie współrzędną

`canvas.move(item_id, 30, 0)` oznacza „przesuń o 30 pikseli w prawo od obecnej pozycji”, a nie „umieść na x=30”. Wywołać `move()` dwa razy z rzędu z tymi samymi argumentami przesuwa element o łącznie 60 pikseli zamiast pozostawiać go na miejscu.

Wszystkie trzy sposoby są `coords()`, `itemconfig()`, `move()` — pracy i `item_id`, oraz Tegu (Rozdział 18.11): `canvas.itemconfig("preview", ...)` się zmieni wszystkie elementy z tym tagiem jednocześnie.

Praktyka: coords, itemconfig, move, delete
Moduł tkinter otwiera natywne okno Python — wykonaj lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/18-19/index.html)

Nakładająca się kolejność

Później na szczycie trafia narysowany element – delikatnie Canvas mówiąc.

To, co jest rysowane później, jest na górze

Canvas elementy są przechowywane nie tylko jako lista, ale także w określonej KOLEJNOŚCI – tej, która w którym zostały stworzone. Jeśli dwa elementy nakładają się na siebie, to ten, który został stworzony, będzie górnym Później:

Prawdziwe okno: trzy nakładające się kształty – prostokąt, owal i linia – owal na górze prostokąta, linia na obu

Prawdziwe okno: kolejność tworzenia — prostokąt, potem owal, potem linia. Linia znalazła się na wierzchu.

Ten porządek można zmienić wyraźnie – podnieść element wyżej lub obniżyć pozostałe:

poryadok_sloev.py

```
canvas.tag_raise(rect_id)          # podnieść prostokąt NAD
wszystkimi innymi
canvas.tag_lower(rect_id)          # obniżyć prostokąt POD
wszystkie pozostałe
canvas.tag_raise(rect_id, oval_id)  # Podnieś prostokąt tylko
POWYŻEJ konkretnego owalu
```

Okno rzeczywiste: te same trzy figury, ale prostokąt teraz na owalu i linii

Prawdziwe okno: Po tag_raise(rect_id) prostokąt jest teraz wyższy niż pozostałe.

Mała wskazówka dotycząca warstw edytorów graficznych

Prawdziwi edytorzy graficzny budują na tym pomyśle cały system warstw — z ukrywaniem, blokowaniem i grupowaniem. Nie wdrażamy tu żadnej z tych zasad: `tag_raise` / `tag_lower` jest tylko dowodem, że kolejność nakładek istnieje i można nią kontrolować, jeśli zajdzie taka potrzeba.

Praktyka: Nakładająca się kolejność

Automatyczna weryfikacja — przewidujemy końcowy porządek po tag_raise/tag_lower

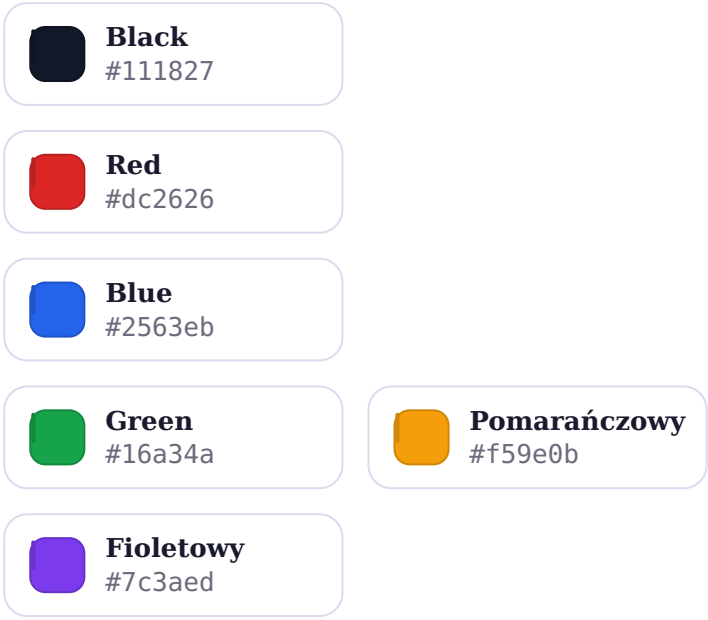
Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/18-20/index.html)

System Kolorów

Sześć szybkich kolorów i colorchooser dla każdego innego — kolor trzeba zobaczyć, a nie tylko czytać kod hex.

Kolor trzeba widzieć, a nie tylko czytać w kodzie

Sześć kolorów palety końcowej aplikacji to nie abstrakcyjne łańcuch znaków, lecz konkretne odcienie:



Okno rzeczywiste: rząd kolorowych przycisków palety nad płótnem

Prawdziwe okno: ta sama paleta — każdy przycisk jest pomalowany na własny bg=hex_color.

Własny kolor — przez colorchooser

Gotowa paleta — to tylko sześć wariantów. Moduł `tkinter.colorchooser` otwiera natywne okno dialogowe systemu do wyboru dowolnego koloru:

custom_color.py

```
from tkinter import colorchooser

def choose_custom_color(self):
    _rgb, hex_color =
colorchooser.askcolor(color=self.state.color, title=„Wybierz
kolor”)
    if hex_color is not None:
        self.set_color(hex_color)
```

KONSPEKT

DIALOGU (NIE PRAWDZIWY ZRZUT EKRANU)

colorchooser natywny schemat dialogowy jest headless Xvfb nie renderuje przewidywalnie bez prawdziwego kliknięcia użytkownika, więc jego struktura jest pokazana tutaj, a nie zrzut ekranu.

askcolor() może powrócić (None, None)

Jeśli użytkownik kliknie „Anuluj” `askcolor()` zwroty `(None, None)`, a nie jakiś domyślny kolor. Kod musi sprawdzić `hex_color is not None` przed użyciem — w przeciwnym razie następne wywołanie oczekujące łańcuch znaków jak `"#7c3aed"`, otrzyma `None` i pojawi się błąd.

Real Window: Obecny kolor na pasku został zmieniony na niestandardowy cień wybrany za pomocą

Realne okno: Efektem wyboru niestandardowego koloru jest to, że dialogi nie znajdują się w kadrze, ale efekt jest naprawdę widoczny.

Praktyka: Paleta i niestandardowy kolor
Moduł tkinter otwiera natywne okno Python — wykonaj lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/18-21/index.html)

Grubość pędzla

Liczba w kodzie i szerokość linii na ekranie — pokażemy różnicę na żywo, a nie tylko ją opisujemy.

Grubość w kodzie — oraz grubość na ekranie

Cztery linie tego samego koloru, narysowane o różnych grubościach – różnica jest widoczna od razu, bez potrzeba przedstawienia go w liczbie:

Prawdziwe płótno: Cztery poziome linie o grubości 1, 3, 8 i 16 pikseli, z oznaczonymi liczbami obok siebie

Prawdziwe płótno: width=1, 3, 8, 16 — ta sama linia, różna grubość.

tolschina_kisti.py

```
self.width_scale = tk.Scale(
    toolbar, from_=1, to=20, orient="horizontal",
command=self.set_width,
)
self.width_scale.set(4)

def set_width(self, value):
    self.state.width = int(value)
```

command w Scale otrzymuje łańcuch znaków, a nie Event
Zgodnie z Sekcją 18.6, `Scale` otrzymuje wartość gotowego suwaka bezpośrednio za pomocą łańcuch znaków – nie przez obiekt `Event` jak `.bind()`. Stąd oczywiste `int(value)`.

Grubość jest przechowywana w DrawingState, nie tylko w samym Scale

`self.state.width` jest źródłem prawdy, którą czytamy `on_press` / `on_drag`. `tk.Scale` to tylko przydatny widget do jego modyfikacji, a nie repozytorium samo w sobie: ta sama logika „widżet wyświetla stan, nie jest”

Praktyka: suwak grubości pędzla

Moduł tkinter otwiera natywne okno Python — wykonaj lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/18-22/index.html)

Gumka

Na Canvas „wymaż” — nie jest jedyną możliwą akcją. Omówimy, jaki sens wybrano tutaj i dlaczego.

Co właściwie oznacza „usunąć” na Canvas?

W edytorze graficznym z zwykłą bitmapą gumka jest oczywista: malowanie pikseli o kolorze tła. Ale Canvas przechowuje nie piksele, lecz elementy (sekcja 18.8) — i „wymazuje”

Trzy uczciwe znaczenia słowa „gumka"

Pomaluj obszar kolorem tła	→	**gumka = kolor tła ołówka**
Usuń cały dotknięty element	→	**potrzebują hit-test — find_overlapping()**

Usuń tylko część pociągnięcia pod kursorem → **trudne: trzeba by pociąć pierwiastek na c**

Na tym kursie wybrano pierwszą, najbardziej przewidywalną opcję: gumka to ten sam ołówek (Rozdział 18.14) tylko kolor pociągnięcia jest wymuszony z kolorem tła płótna:

lastik.py

```
def on_drag(self, event):
    tool = self.state.tool
    if tool in (Tool.PENCIL, Tool.ERASER):
        color = CANVAS_BG if tool is Tool.ERASER else
self.state.color
        self.canvas.create_line(
            self.state.last_x, self.state.last_y, event.x,
event.y,
            fill=color, width=self.state.width,
capstyle=tk.ROUND,
        )
        self.state.last_x, self.state.last_y = event.x, event.y
```

Prawdziwe okno: płótno z malowanymi kształtami przed nałożeniem gumki

Prawdziwe okno: płótno z figurami — do gumki.

Prawdziwe okno: to samo płótno z białym paskiem, gdzie pociągnięcie gumki znajduje się nad kształtami

Rzeczywiste okno: po gumce — biały pasek na części figur, a nie prawdziwe usunięcie dotkniętych elementów.

To NIE jest prawdziwe usuwanie dotkniętych elementów

Taki gumka rysuje NOWY element koloru tła na starych — same stare figury pozostają w dokumencie, po prostu są wizualnie przykryte. Jeśli tło płótna kiedykolwiek się zmieni (np. dodasz obraz tła), „wymazane" linie znowu staną się widoczne — to nie jest błąd, lecz bezpośrednia konsekwencja wybranego modelu.

Prawdziwe usunięcie jest opcjonalnym rozszerzeniem

Metoda `canvas.find_overlapping(x1, y1, x2, y2)` znajduje wszystkie elementy w prostokątnym obszarze pod kursorem gumki — można je naprawdę usunąć przez `canvas.delete(item_id)`. To uczciwa, ale znacznie bardziej złożona alternatywa: wymaga podjęcia decyzji, co zrobić, jeśli gumka dotyka tylko CZĘŚCI długiej figury. Nie jest to implementowane w tym rozdziale, ale jest wspomniane jako realny kierunek rozwoju.

Praktyka: gumka jako pociągnięcie koloru tła

Moduł tkinter otwiera natywne okno Python — wykonaj lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/18-23/index.html)

Cofnij działania (Undo)

Pojedynczy gest użytkownika może składać się z wielu elementów Canvas i musi być całkowicie anulowany.

Cofnij przez działanie, a nie przez Canvas przedmioty

Naiwną implementacją Undo jest po prostu usunięcie ostatniego elementu utworzonego Canvas. Problem: jeden kres ołówka składa się z WIELU pojedynczych segmentów (Rozdział 18.14) – „cofnij"

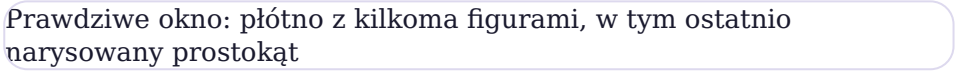
istoriya_deystviy.py

```
undo_stack = [
    [item_id_1, item_id_2, item_id_3],  # jeden pociągnięcie ołówka — trzy odcinki
    [item_id_4],                         # jeden prostokąt — jeden element
]
```

Ostateczna aplikacja rozwiązuje to poprzez dokument (sekcja 18.13, 18.29): każdy użytkownik Akcja — Jeden lub więcej obiektów `Shape` - W pełni dodane do Dokument można anulować w całości:

undo.py

```
def _commit_action(self, shapes):
    self.document.extend(shapes)
    self.undo_stack.append(shapes)   # JEDNA akcja – nawet
jeśli shapes z wielu segmentów
    self.redo_stack.clear()
    self.render_document()

def undo(self):
    if not self.undo_stack:
        return
    shapes = self.undo_stack.pop()
    del self.document[len(self.document) - len(shapes):]
    self.redo_stack.append(shapes)
    self.render_document()
```



Okno rzeczywiste: do Ctrl+Z — widoczny cały obraz, w tym ostatnia akcja.

Realne okno: po Ctrl+Z – ostatnia AKCJA jest całkowicie usuwana, poprzednie wartości nie są ruszane.

render_document() jest tym samym render() co w rozdziale 17

Po anulowaniu dokument zmienia się, a płótno po prostu PRZERYSOWUJE się z niego na nowo — nie jest to punktowe usuwanie konkretnych elementów Canvas. Ten sam zasad „model zmienia akcje, Canvas tylko wyświetla aktualny stan” `render()` w grze „Kółko-krzyżyk”

Praktyka: Cofnij stos według akcji
Automatyczne skanowanie – undo usuwa WSZYSTKIE przedmioty z jednej akcji, zamiast tylko jednej

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/18-24/index.html)

Powtarzanie czynności (Redo)

Redo jest drugim stosem obok Undo. Nowa akcja po anulowaniu powinna go opróżnić.

Redo — drugi stos, a nie „cofnij cofnięcie”

Rozdział 18.24 uchyla powództwo, usuwając je z `undo_stack`. Aby przywrócić go z powrotem, samo działanie trzeba tymczasowo gdzieś umieścić — na drugi stos, `redo_stack`:

redo.py

```
def redo(self):
    if not self.redo_stack:
        return
    shapes = self.redo_stack.pop()
    self.document.extend(shapes)
    self.undo_stack.append(shapes)
    self.render_document()
```

Real Window: Prostokąt wcześniej usunięty przez Undo ponownie jest widoczny na płótnie po Redo

Realne okno: Po Ctrl+Y - Decyzja cofnięta w sekcji 18.24 została przywrócona.

Nowe działanie musi zniszczyć redo_stack

Jeśli po Undo użytkownik narysuje coś NOWEGO, stara gałąź „to może być powtarzać” `_commit_action()` (Rozdział 18.24) zawsze oczyszcza `redo_stack`:

commit_ochischaet_redo.py

```
def _commit_action(self, shapes):
    self.document.extend(shapes)
    self.undo_stack.append(shapes)
    self.redo_stack.clear()   # nowa akcja sprawia, że stara
gałąź redo nieważna
    self.render_document()
```

Forgotten redo_stack.clear() to niezauważalny, ale realny błąd
Bez tej linii Redo mógłby "wskrzeszić" kształt, który nie ma nic wspólnego z aktualnym stanem dokumentu, ponieważ po Undo użytkownik zdołał narysować coś innego, a stara akcja cofnięta nadal znajduje się w kolejce "redo". Rozdział 18.31 traktuje ten przypadek jako osobny Debug Lab.

Praktyka: Stack powtórki i wyczyszczenie nowej akcji
Auto Check – redo przywraca akcję, nowa akcja resetuje redo

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/18-25/index.html)

Czyszczenie płótna

Nieodwracalna akcja wymaga potwierdzenia — i musi wyzerować historię cofania razem z dokumentem.

„Oczyszczenie”

Przycisk „Wyczyść płótno” usuwa CAŁY rysunek naraz. W przeciwieństwie do Undo, tutaj nie ma poprzedniego stanu, do którego można łatwo wrócić — jeśli historia cofania również jest czyszczona wraz z dokumentem. Dlatego przed nieodwracalnym działaniem warto poprosić o potwierdzenie, ale tylko jeśli jest co stracić:

ochistka_holsta.py

```
def clear_canvas(self):
    if self.document and not messagebox.askyesno(
        „Wyczyść płótno”, „Usunąć cały rysunek bez możliwości
cofnięcia?”
    ):
        return
    self.document.clear()
    self.undo_stack.clear()
    self.redo_stack.clear()
    self.render_document()
```

Sprzątanie płótna zapomniało usunąć historię

Jeśli usunąć tylko figury (`self.document.clear()`), ale nie dotykaj `undo_stack` , — `undo()` nie będzie robić NIC na zewnątrz po wyczyszczeniu (nie ma czego usuwać z pustego dokumentu), ale przechodzi przestarzała akcja do `redo_stack` . Następnie Ctrl+Y przywróci na płótno kształty, które użytkownik właśnie usunął „Czyść”

Nie zawsze pytamy o potwierdzenie

Jeśli `self.document` jest już pusty, pytanie „naprawdę wyczyścić?” nie ma sensu — zbędne okno dialogowe tylko denerwuje. Sprawdzenie `if self.document and ...` pomija potwierdzenie w tym konkretnym przypadku.

Praktyka: Czyść płótno bezpiecznie

Moduł tkinter otwiera natywne okno Python — wykonaj lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/18-26/index.html)

skróty klawiszowe

Ctrl+Z Ctrl+Y inne skróty są powiązane z root, nie tylko z płótnem.

Skróty klawiaturowe

Kombinacja	Akcja
`Ctrl+Z`	Undo - Cofnij ostatnią akcję

Kombinacja	Akcja
`Ctrl+Y`	Redo - Spróbuj ponownie cofnąć akcję
`Ctrl+N`	oczyścić płótno (po potwierdzeniu)
`Ctrl+S`	zachowaj rysunek do JSON
`Ctrl+O`	otworzyć zapisany rysunek

goryachie_klavishi.py

```
self.root.bind("<Control-z>", lambda _e: self.undo())
self.root.bind("<Control-y>", lambda _e: self.redo())
self.root.bind("<Control-s>", lambda _e: self.save_document())
self.root.bind("<Control-o>", lambda _e: self.load_document())
self.root.bind("<Control-n>", lambda _e: self.clear_canvas())
```

Skróty klawiszowe — nie są uniwersalne między systemami operacyjnymi

`Ctrl` jest standardem dla Windows i Linux. Na macOS użytkownicy nawykowo czekają na `Cmd`, nie `Ctrl` — Tkinter nie zastępuje jednego drugim automatycznie. Pełne wsparcie wieloplatformowe wymagałoby osobnego sprawdzania `sys.platform` i wiązać oba warianty — ten rozdział się tym nie zajmuje, ale ważne jest, aby nie obiecywać tego, czego kod nie robi.

Wiążące root, nie tylko Canvas — pamiętaj o Rozdziale 17

Skróty klawiszowe są zawieszone `self.root`, nie włączaj `self.canvas` — powinny działać niezależnie od tego, który widżet okna jest aktualnie w ostrości. Gdyby aplikacja miała pole tekstowe (na przykład do podpisów do obrazka), musiałbyś uwzględnić te same zasady fokusu i bindtags, jak omówione w sekcji 17.11 — skróty klawiszowe nie powinny przechwytywać tekstu wprowadzanego w polu, w którym użytkownik wpisuje.

Praktyka: Skróty klawiszowe aplikacji
Moduł tkinter otwiera natywne okno Python — wykonaj lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/18-27/index.html)

Pasek statusu

Narzędzie, współrzędne, kolor i grubość — od razu widoczne, bez potrzeby zaglądania do kodu.

Jeden łańcuch znaków, cztery fakty naraz

Real Window: łańcuch znaków stanu na dole okna pokazuje instrument, współrzędne kursora, aktualny kolor i grubość

Prawdziwe okno: łańcuch znaków stan pod płótnem – narzędzie, współrzędne, kolor, grubość.

stroka_sostoyaniya.py

```
def _update_status(self, x, y):
    coords = f"x={x} y={y}" if x is not None else "x=— y=—"
    self.status_var.set(
        f"Narzędzie: {TOOL_LABELS[self.state.tool]} | {coords}
| "
        f"Kolor: {self.state.color} | Grubość:
{self.state.width}"
    )
```

Linia jest aktualizowana w trzech różnych miejscach: gdy zmieniasz narzędzia, gdy poruszasz myszką (`<Motion>`) oraz w wyborze koloru lub grubości, gdziekolwiek co najmniej jedna z czterech wartości się zmienia.

Żywy dowód korzyści płynących z event.x / event.y
Współrzędne kursora — to nie abstrakcja „gdzieś w kodzie”: łańcuch znaków stany czynią je widocznymi w czasie rzeczywistym, dosłownie przy każdym ruchu myszy. To to samo `event.x` / `event.y` , którego używają `on_press` / `on_drag` dla samego rysunku.

Praktyka: łańcuch znaków stanu aplikacji
Moduł tkinter otwiera natywne okno Python — wykonaj lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/18-28/index.html)

PaintApp architektura

app.root, stan narzędzia, dokument kształtu i Canvas to trzy różne warstwy tej samej aplikacji.

app HAS-A root — ten sam wzorzec co w rozdziałach 16–17

Składanie a dziedziczenie

Możliwe, ale niekonieczne

```
class PaintApp(tk.Tk):
    def __init__(self):
        super().__init__()
        ...
```

Jak w rozdziałach 16–17

```
class PaintApp:
    def __init__(self, root):
        self.root = root
        ...
```

Co używać dzisiaj: Dziedziczenie od `tk.Tk` technicznie działa, ale nie zgadza się z strukturą załączników w tej książce od rozdziału 16. `app.root` jest tym samym wzorem co kalkulator końcówek i kółko i krzyżyk.

Trzy obowiązki, trzy grupy metod

app : PaintApp

```
root = Tk
state = DrawingState
document = list[Shape]
undo_stack = list[list[Shape]]
redo_stack = list[list[Shape]]
canvas = Canvas
```

app przechowuje widżety, parametry narzędzi i dokument — ale nie myli ich ze sobą.

Grupa metod	Za co odpowiada
`build_ui / build_toolbar`	budowania widgetów — raz, przy starcie
`on_press / on_drag / on_release`	zdarzenia myszy → zmiany state i document
`render_document`	dokument → Canvas (jedyne miejsce, które naprawdę rysuje)
`undo / redo / clear_canvas`	Zarządzanie historią dokumentów
`save_document / load_document`	plik dokumentu ↔ na dysku

zdarzenie myszy

↓

Zdarzenie najpierw zmienia stan i dokument, a dopiero potem Canvas jest przerysowane z dokumentu.

UI-status, dokument i display to trzy różne warstwy

`DrawingState` – co jest wybrane TERAZ (narzędzie, kolor). `document` — co jest już NARYSOWANE i co przeżywa zmianę narzędzia. `Canvas` — to, co widać na ekranie, jest pochodną dokumentu. Mieszanie pierwszego z drugim jest częstą przyczyną zamieszania: na przykład przechowywanie historii Undo wewnątrz `DrawingState` zamiast osobnej listy nie pozwoliłoby odwoływać się do działań wykonanych PRZED zmianą narzędzia przez użytkownika.

Praktyka: budujemy graf obiektowy PaintApp

Moduł tkinter otwiera natywne okno Python — wykonaj lokalnie w VS Code, PyCharm lub Jupyter

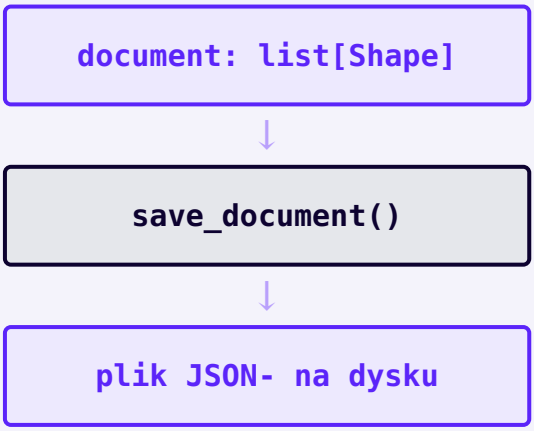
Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/18-29/index.html)

Zapisywanie i ładowanie JSON

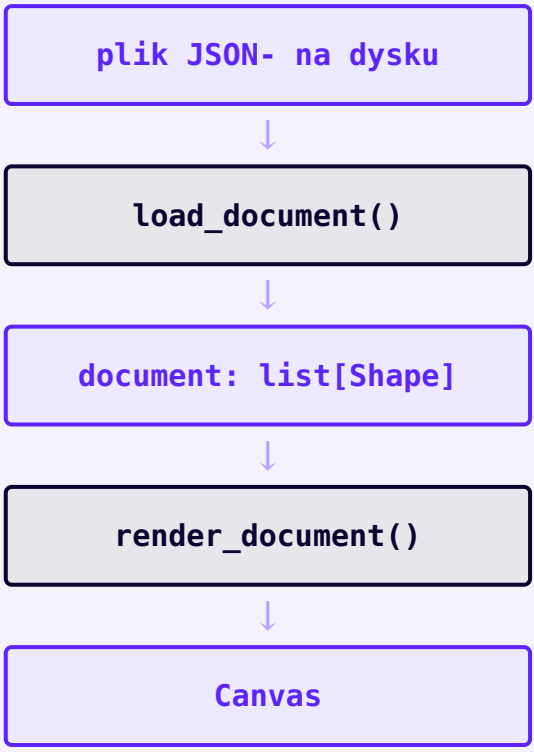
Zapisuje się nie obrazek, lecz dokument figur — Canvas potrafi odtworzyć z niego płótno w dowolnym momencie.

Canvas nie zapisuje zdjęcia jako pliku – dokument jest zapisywany

Ważne, szczere zastrzeżenie: `Canvas` nie potrafi zrobić tego samodzielnie Eksportuj się do zwykłych PNG lub JPEG – niezawodny, wieloplatformowy eksport rastrowy wymaga dodatkowych zależności i wykracza poza zakres tego rozdziału. Zamiast tego zapisujemy DOCUMENT to lista liczb, z których Canvas już zbudowany jest `render_document()` (Rozdział 18.29):



Ochrona środowiska.



Ładowanie – Zawsze kończy się ponownym rysowaniem, a nie tylko odczytem pliku.

drawing.json

```
{
  "version": 1,
  "canvas": {"background": "#ffffff"},
  "items": [
    {
      "kind": "line",
      "coords": [34, 52, 180, 90],
      "color": "#2563eb",
      "width": 4
    },
    {
      "kind": "rectangle",
      "coords": [80, 120, 220, 210],
      "color": "#dc2626",
```

```
      "width": 3
    }
  ]
}
```

sohranenie_json.py

```
def save_document(self):
    path_str = filedialog.asksaveasfilename(
        defaultextension=".json", filetypes=[(„Figura JSON”,
"*.json")]
    )
    if not path_str:  # użytkownik kliknął "Anuluj" – path_str
== ""
        return
    data = {
        "version": 1,
        "canvas": {"background": CANVAS_BG},
        "items": [shape.to_dict() for shape in self.document],
    }
    Path(path_str).write_text(json.dumps(data,
ensure_ascii=False, indent=2), encoding="utf-8")
```

Real Window: rysowanie z wieloma kształtami tuż przed zapisem

Prawdziwe okno: zdjęcie przed Ctrl+S — ten sam dokument będzie w pliku JSON-.

Rzeczywiste okno: ten sam obraz, odtworzony po załadowaniu JSON w nowej sesji

Realne okno: ten sam obraz po Ctrl+O w innym uruchomieniu programu — przywrócony z JSON, a nie z pamięci poprzedniego procesu.

Anulowanie dialogu nie jest błędem, lecz normalnym sposobem

I `asksaveasfilename()`, i `askopenfilename()` zwracają pusty łańcuch znaków `""` jeśli użytkownik kliknie „Anuluj” `None` i nie rzucaj wyjątek. Kod musi to zweryfikować PRZED przekazaniem ścieżki do `Path(...)`: `Path("")` jest ważnym, ale bezsensownym obiektem, który wskazuje na aktualny katalog, a nie na plik wybrany przez użytkownika.

A jeśli plik jest uszkodzony?

Użytkownik otwiera plik, co oznacza, że może wybrać dowolne: czyjąś JSON, obraz, plik niepełny w przypadku niepowodzenia, ręcznie edytowany rysunek z literówką. Rozdział 15 (Rozdział 15.25) już wykazała, że `json.load()` o tym informuje via `json.JSONDecodeError`. Tutaj jest dodawany do Wiele funkcji: Kluczowe `"items"` może w ogóle nie istnieć, kształt może brakować pola, a kolor może być nić, którego Tk nie zna.

Ważne jest, by nie tylko „nie upadać” **lokalny** lista i zastępuje dokument tylko wtedy, gdy analiza jest całkowicie skuteczna:

zagruzka_bezopasno.py

```
def load_from_path(self, path):
    try:
        data = json.loads(path.read_text(encoding="utf-8"))
        shapes = [Shape.from_dict(item) for item in
data["items"]]
        for shape in shapes:
            self.canvas.winfo_rgb(shape.color)   # kolor
istnieje?
    except json.JSONDecodeError as exc:
        messagebox.showerror(„Plik nie mógł zostać otwarty”,
f"To nie jest poprawna JSON.\n{exc}")
        return
    except (KeyError, TypeError, ValueError, tk.TclError) as
exc:
        messagebox.showerror(„Plik nie mógł zostać otwarty”,
f"To nie jest zapisany rysunek n{exc}")
        return


# Parsowanie zakończyło się sukcesem, tylko teraz zmieniamy
stan aplikacji.
    self.document = shapes
    self.undo_stack.clear()
    self.redo_stack.clear()
    self.render_document()
```

Najpierw rozmontować, potem wymienić — i nie odwrotnie

Pokusa pisania `self.document = [...]` od razu, ale `try` tylko się trzymać `json.loads()`. Wtedy plik o nieznanym kolorze zostanie rozłożony, dokument już zostanie zastąpiony — i upadnie `render_document()`. Podsumowując: stary rysunek zostaje zgubiony, historia cofnięć jest czysta, a każda kolejna próba narysowania czegoś znowu się zawiesza, bo przerysowanie zawsze kończy się tym samym złym kondem. Sprawdzenie PRZED zadaniem kosztuje jedną dodatkową zmienną i oszczędza pracę użytkownika.

Sprawdzanie koloru potrafi tylko płótno

`Shape` świadomie nic nie wie o Tkinter (Rozdział 18.29), więc nie może sprawdzić koloru za niego `"#2563eb"` oraz `"не-цвет"` są równie podobne do koloru. Ale `canvas.winfo_rgb(color)` sam pyta Tk i rzuca `TclError` jeśli takiego koloru nie ma. Podział obowiązków zostaje zachowany: model sprawdza strukturę, płótno sprawdza, co należy do rysunku.

Praktyka: Zapisz i wczytaj rysunek, aby JSON

Moduł tkinter otwiera natywne okno Python — wykonaj lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/18-30/index.html)

Debug Labs typowe błędy rysowania

Każdy błąd tutaj występuje w prawdziwych projektach studenckich — naucz się rozpoznawać objaw, zanim otworzysz debugator.

Mała kolekcja typowych błędów rysowników na Canvas — każdy z objawem i poprawką. Część z nich widzieliście już wcześniej w rozdziale; tutaj są zebrane jako podręcznik.

DEBUG LAB 1:Y WZNOSI SIĘ W GÓRĘ JAK TURTLE

turtle_privychka.py

```
def on_drag(event):
    # 'znana' logika: im wyżej na ekranie, tym WIĘCEJ Y
    if event.y > start_y:
        status.config(text=„Moving UP”)
# nieprawidłowe dla Canvas!
```

```
# Курсор двигается вниз по экрану, а статус говорит 'ВВЕРХ'
# логика была написана в терминах Turtle, а не Canvas.
```

Co widać na ekranie

Canvas Y rośnie W DÓŁ (Rozdział 18.9) — `event.y > start_y` oznacza „kursor jest niższy niż wcześniej”

POPRAWIONY KOD

canvas_napravlenie.py

```
def on_drag(event):
    if event.y > start_y:
        status.config(text=„Poruszamy się w DÓŁ”)  # Prawdziwe: Y Wznosi się w dół
```

DEBUG LAB 2: NIE PAMIĘTAŁEM PUNKTU POCZĄTKOWEGO GESTU

zabyli_start.py

```
def on_press(event):
    pass  # zapomnieli zapisać event.x, event.y

def on_drag(event):
    canvas.create_line(start_x, start_y, event.x,
event.y)  #start_x nieokreślone
```

```
# NameError: name 'start_x' is not defined —
# программа падает на первом же движении мыши после нажатия.
```

Co widać na ekranie

`on_press()` musi zachować punkt początkowy (Rozdział 18.12) bez niego `on_drag()` po prostu nie wiadomo, gdzie narysować linię lub podgląd kształtu.

POPRAWIONY KOD

sohranili_start.py

```
def on_press(event):
    global start_x, start_y
    start_x, start_y = event.x, event.y
```

DEBUG LAB 3: <MOTION> ZAMIAST <B1-MOTION>

lyuboe_dvizhenie.py

```
canvas.bind("<Motion>", on_drag)   # wyzwala się przy
KAŻDYM ruchu myszy
```

```
# Линия рисуется, даже если кнопка мыши вообще не нажата —
# достаточно просто провести курсором над холстом.
```

Co widać na ekranie

`<Motion>` reaguje na każdy ruch myszy nad widżetem. Aby rysować tylko podczas przeciągania, musisz `<B1-Motion>` – Rozdział 18.12 wyjaśnia różnicę.

POPRAWIONY KOD

b1_motion.py

```
canvas.bind("<B1-Motion>", on_drag)   # tylko przy wciśniętym lewym przycisku
```

DEBUG LAB 4: TYSIĄCE NIEPOŁĄCZONYCH PUNKTÓW ZAMIAST LINII

tochki_vmesto_linii.py

```
def on_drag(event):
    canvas.create_oval(
        event.x - 2, event.y - 2, event.x + 2, event.y + 2,
        fill="black",
    )
```

```
# Быстрое движение мыши оставляет заметные разрывы между кр
# штрих выглядит рваным, а не гладким.
```

Co widać na ekranie

Okrąg dla każdego zdarzenia indywidualnego nie uwzględnia ścieżki MIĘDZY dwoma kolejnymi punktami. Rozdział 18.14 pokazuje rozwiązanie połączenia nowego punktu z poprzednim przez `create_line()`.

POPRAWIONY KOD

linii_mezhdu_tochkami.py

```
def on_drag(event):
    canvas.create_line(last_x, last_y, event.x, event.y,
width=3, capstyle=tk.ROUND)
    # last_x, last_y aktualizowane po każdym odcinku
```

DEBUG LAB 5: PODGLĄD JEST ODTWARZANY PRZY KAŻDYM MOTION BEZ POTRZEBY

peresozdanie_prevyu.py

```
def on_drag(event):
    canvas.delete(preview_id)
    preview_id = canvas.create_rectangle(start_x,
start_y, event.x, event.y)
```

```
# Работает, но на каждое из десятков событий Motion в секунду
# Canvas выбрасывает и заново строит один и тот же элемент.
```

Co widać na ekranie

Usunąć i stworzyć od nowa — to nie błąd w sensie „zepsute”, ale zbędna praca: Canvas już ma `coords()` dokładnie do aktualizacji istniejącego elementu (sekcja 18.18) — bez ponownego tworzenia.

POPRAWIONY KOD

coords_vmesto_peresozdaniya.py

```
def on_drag(event):
    canvas.coords(preview_id, start_x, start_y, event.x,
event.y)
```

DEBUG LAB 6: FIGURKI ZAPOWIEDZI LOST ID

poteryan_id.py

```
def on_press(event):
    canvas.create_rectangle(event.x, event.y, event.x,
event.y, dash=(4, 2))
    # id nie zapisany — preview_id nigdzie się nie
pojawił

def on_drag(event):
    canvas.coords(preview_id, ...)  # NameError:
preview_id nie jest zdefiniowana
```

```
# NameError на первом же движении мыши —
# ссылаться не на что, id черновой фигуры потерян.
```

Co widać na ekranie

`create_*()` zwroty `item_id` (Rozdział 18.10), ale tylko jeśli zostanie ZAPISANE, w przeciwnym razie kształt zostanie narysowany, ale później nie będzie do czego się odwoływać.

POPRAWIONY KOD

sohranili_id.py

```
def on_press(event):
    global preview_id
    preview_id = canvas.create_rectangle(event.x,
event.y, event.x, event.y, dash=(4, 2))
```

DEBUG LAB 7: PROSTOKĄT „PRZEWRACA”

ne_normalizovano.py

```
def get_bounds():
    return start_x, start_y, current_x, current_y  # z
wyłączeniem kierunku

left, top, right, bottom = get_bounds()
if left > right:
    raise ValueError(„to niemożliwe”)  # ale
przeciąganie w górę-i-lewo właśnie to daje!
```

```
# Приложение падает с ValueError, если пользователь начал
# перетаскивание из правого нижнего угла будущей фигуры.
```

Co widać na ekranie

Myszką można przeciągać w dowolnym z czterech kierunków (sekcja 18.16) – oczekiwanie na kod `left <= right` bez wcześniejszej normalizacji działa tylko dla jednego z czterech kierunków.

POPRAWIONY KOD

normalizovano.py

```
left, top, right, bottom = normalize_bounds(start_x,
start_y, current_x, current_y)
# jest teraz left <= right oraz top <= bottom
gwarantowane, niezależnie od kierunku
```

DEBUG LAB 8: ITEM_ID UŻYWANY JAKO INDEKS LISTY

id_kak_indeks.py

```
shapes = []
item_id = canvas.create_rectangle(10, 10, 50, 50)
shapes.append("rectangle")
print(shapes[item_id])   #IndexError jeśli item_id
więcej niż len(shapes) - 1
```

```
# IndexError: list index out of range —
# item_id не совпадает с порядковым номером в собственном
```

Co widać na ekranie

`item_id` — wewnętrzny numer Canvas (rozdział 18.10), a nie pozycja na własnej liście figur. Używać go jako indeksu `shapes[item_id]` zdarzy się tylko przypadkowo.

POPRAWIONY KOD

id_kak_klyuch_slovarya.py

```
shapes_by_id = {}
item_id = canvas.create_rectangle(10, 10, 50, 50)
shapes_by_id[item_id] = "rectangle"   # słownik — id
jako KLUCZ, a nie indeks
```

DEBUG LAB 9: UNDO USUWA TYLKO OSTATNI SEGMENT KRESKI, A NIE CAŁĄ LINIĘ

chastichnaya_otmena.py

```
def undo(self):
    item_id = self.undo_stack.pop()
# zakłada JEDEN id na akcję
    self.canvas.delete(item_id)
```

```
# После Ctrl+Z длинный карандашный штрих теряет только
# последний маленький отрезок, а не исчезает целиком.
```

Co widać na ekranie

Jedno pociągnięcie ołówkiem to DUŻO segmentów `create_line()` (Rozdział 18.14), nie jeden element. Undo musi przechowywać cały lista id (lub liczby) dla KAŻDEJ akcji, a nie tylko jeden id (Rozdział 18.24).

POPRAWIONY KOD

otmena_celym_deystviem.py

```
def undo(self):
    shapes = self.undo_stack.pop()   # lista postacie
JEDNEGO działania
    del self.document[len(self.document) - len(shapes):]
    self.render_document()
```

DEBUG LAB 10:REDO WSKRZESZA NIEZWIĄZANE DZIAŁANIE

redo_bez_ochistki.py

```
def _commit_action(self, shapes):
    self.document.extend(shapes)
    self.undo_stack.append(shapes)
    # zapomniałem self.redo_stack.clear()
```

```
# Undo, затем новый рисунок, затем Redo —
# и на холсте внезапно появляется фигура, никак не связанна
```

Co widać na ekranie

Jeśli po Undo narysujesz coś nowego, stara gałąź redo już nie odpowiada aktualnemu dokumentowi. Rozdział 18.25 wymaga wyraźnego rozliczenia `redo_stack` przy każdej nowej akcji.

POPRAWIONY KOD

redo_s_ochistkoj.py

```
def _commit_action(self, shapes):
    self.document.extend(shapes)
    self.undo_stack.append(shapes)
    self.redo_stack.clear()
```

DEBUG LAB 11: OCZYŚCIŁEM PŁÓTNO, ALE NIE OPOWIEŚĆ

ochistka_bez_istorii.py

```
def clear_canvas(self):
    self.canvas.delete("all")
    self.document.clear()
    #undo_stack i redo_stack pozostają nietknięte!
```

```
# После 'Очистить' нажатие Ctrl+Z не делает ничего видимого –
# а следующий Ctrl+Y возвращает на холст фигуры, которые
# пользователь только что удалил кнопкой 'Очистить'.
```

Co widać na ekranie

Cichy błąd: `del document[len(document) - len(shapes):]` na pustym dokumencie — to jest `del document[-3:]`, że nic nie usuwa ani nie psuje. Ale anulowana „akcja” `redo_stack`, a następny Ctrl+Y wraca na płótno kształty wymazane przyciskiem Czyść. Historia opisuje dokument, który już nie istnieje: gdy `document` jest oczyszczony, `undo_stack` oraz `redo_stack` musi zostać oczyszczony razem z nim (Rozdział 18.26).

POPRAWIONY KOD

ochistka_s_istoriej.py

```
def clear_canvas(self):
    self.document.clear()
    self.undo_stack.clear()
    self.redo_stack.clear()
    self.render_document()
```

DEBUG LAB 12: ANULUJ COLORCHOOSER ZAPISUJE NONE JAKO KOLOR

otmena_colorchooser.py

```
def choose_custom_color(self):
    _rgb, hex_color = colorchooser.askcolor()
    self.set_color(hex_color)   # nie testowałem
hex_color na None!
```

```
# Пользователь нажал 'Отмена' в диалоге —
# следующая же попытка нарисовать линию падает: fill=None
```

Co widać na ekranie

`askcolor()` zwroty `(None, None)` przy anulowaniu (Rozdział 18.21) – Ustawienie tego jako aktualnego koloru bez zaznaczania oznacza przerwanie rysunku przy następnej próbie.

POPRAWIONY KOD

proverka_otmeny.py

```
def choose_custom_color(self):
    _rgb, hex_color = colorchooser.askcolor()
    if hex_color is not None:
        self.set_color(hex_color)
```

DEBUG LAB 13: ANULUJ DIALOG ZAPISU - PATH('') ZAMIAST WYCHODZIĆ

otmena_sohraneniya.py

```
def save_document(self):
    path_str = filedialog.asksaveasfilename()
    Path(path_str).write_text(...)   # nie sprawdziłem
pustego sznurka!
```

```
# Пользователь нажал 'Отмена' в диалоге сохранения —
# программа пытается записать файл по пути текущей директории
```

Co widać na ekranie

`asksaveasfilename()` / `askopenfilename()` zwracają pusty łańcuch znaków przy anulowaniu, a nie `None` , a nie wyjątek (Rozdział 18.30) — kod musi to zweryfikować przed użyciem ścieżki.

POPRAWIONY KOD

proverka_otmeny_faila.py

```
def save_document(self):
    path_str = filedialog.asksaveasfilename()
    if not path_str:
        return
    Path(path_str).write_text(...)
```

DEBUG LAB: JSON ZAŁADOWANA, ALE NIE PRZERYSOWANA CANVAS

zagruzka_bez_render.py

```
def load_from_path(self, path):
    data = json.loads(path.read_text(encoding="utf-8"))
    self.document = [Shape.from_dict(item) for item in
data["items"]]
    #forgot self.render_document()!
```

```
# self.document правильно заполнен новыми фигурами,
# но на экране всё ещё виден старый рисунок — или пустой хо
```

Co widać na ekranie

Tak samo jak w rozdziale 17: Zmiana modelu bez wywołania renderowania nie dociera do ekranu. `load_from_path()` musi się zakończyć `render_document()` (rozdział 18.30).

POPRAWIONY KOD

zagruzka_s_render.py

```
def load_from_path(self, path):
    data = json.loads(path.read_text(encoding="utf-8"))
    shapes = [Shape.from_dict(item) for item in
data["items"]]
    self.document = shapes
    self.undo_stack.clear()
    self.redo_stack.clear()
    self.render_document()   # Bez tej linii ekran nie
będzie wiedział o pobraniu
    # (pełna wersja z uszkodzonym przetwarzaniem plików
znajduje się w sekcji 18.30)
```

DEBUG LAB 16: USZKODZONY PLIK USUWA AKTUALNY RYSUNEK

zagruzka_bez_proverki.py

```
def load_from_path(self, path):
    data = json.loads(path.read_text(encoding="utf-8"))
    self.document = [Shape.from_dict(i) for i in
data["items"]]  # już zastąpiono!
    self.undo_stack.clear()
    self.render_document()
# ... I tylko tutaj wpadamy w zły kolor
```

```
# Открыли чужой JSON — рисунок, над которым работали час, исч
# Ctrl+Z его не возвращает, а каждая новая линия снова падает
# _tkinter.TclError: unknown color name "not-a-color"
```

Co widać na ekranie

Dokument został zastąpiony ZANIM wiadomo, że plik jest rysowany. Przeparsuj plik na zmienną lokalną i przypisz `self.document` dopiero po sukcesie (rozdział 18.30) — w przeciwnym razie jedna próba otwarcia niewłaściwego pliku zabiera zarówno rysunek, jak i historię oraz możliwość dalszego rysowania.

POPRAWIONY KOD

zagruzka_s_proverkoj.py

```
def load_from_path(self, path):
    try:
        data =
json.loads(path.read_text(encoding="utf-8"))
        shapes = [Shape.from_dict(i) for i in
data["items"]]
        for shape in shapes:
            self.canvas.winfo_rgb(shape.color)
    except (json.JSONDecodeError, KeyError, TypeError,
ValueError, tk.TclError) as exc:
        messagebox.showerror(„Plik nie mógł zostać
otwarty”, str(exc))
        return
    self.document = shapes
    self.render_document()
```

Realne okno: Ta sama aplikacja powiększona – płótno zajmuje więcej miejsca, ale wcześniej narysowane kształty pozostają tego samego rozmiaru przy tych samych współrzędnych absolutnych

Prawdziwe okno po przybliżeniu: Płótno się powiększyło, ale malowane kształty nie powiększyły się razem z nim – dokładnie to, co pokazuje Debug Lab 15 poniżej.

DEBUG LAB 15: ZMIANA ROZMIARU OKNA NIE POWODUJE SKALOWANIA OBRAZU

resize_kak_masshtab.py

```
#standby: jeśli rozciągniesz okno, stare kształty będą
się z nim zbliżać
canvas.grid(row=1, column=0, sticky="nsew")
# ... Ale nikt nie przelicza istniejących współrzędnych
create_line/rectangle/oval
```

```
# Сам виджет Canvas становится больше вместе с окном —
# но уже нарисованные фигуры остаются в прежних абсолютных ко
```

Co widać na ekranie

`sticky="nsew"` oraz wagi wierszy/kolumn (Rozdział 17) powodują, że sam widżet rośnie Canvas nie jest tym samym co automatyczne skalowanie współrzędnych już narysowanych kształtów. Rozdział 18.29 wyraźnie rozbija tę różnicę: większe płótno nie oznacza automatycznie większego rysunku.

POPRAWIONY KOD

resize_kak_bolshe_mesta.py

```
# Fair Model: Powiększone okno daje WIĘCEJ MIEJSCA na
nowe kształty,
# a nie rozciąga już istniejące — jeśli potrzebny
ostatni, współrzędne
# musiałaby być przeliczana ręcznie przy zmianie
rozmiaru.
```

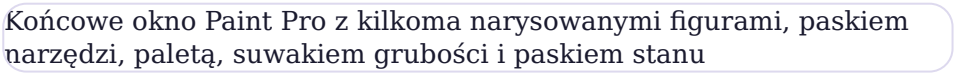
Praktyka: Odtworzenie i naprawa błędów rysunku
Automatyczna kontrola — Debug Lab 11 (czyszczenie i historia) oraz Debug Lab 10 (redo po nowym działaniu)

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/18-31/index.html)

Paint Pro — pełny program i wyniki rozdziału

Ten sam płótno co w rozdziale 18.7 — ale teraz z dokumentem, historią cofania i architekturą, która wytrzyma rozwój projektu.

Program końcowy



Właściwe okno w ostatecznej wersji to ten sam obraz, który otwierał rozdział w sekcji 18.1, ale teraz wiemy, z czego jest zrobiony.

Plik jest kompletny, samodzielny i nie posiada niewidzialnych zależności od innych lekcji:

projects/tkinter/paint-app/paint_app.py

paint_app.py — struktura

```
class Tool(Enum): ...

@dataclass
class Shape: ...

@dataclass
class DrawingState: ...

def normalize_bounds(x1, y1, x2, y2): ...

class PaintApp:
    def __init__(self, root): ...
    def build_ui(self): ...
    def set_tool(self, tool): ...
```

```
    def set_color(self, hex_color): ...
    def choose_custom_color(self): ...
    def set_width(self, value): ...
    def on_press(self, event): ...
    def on_drag(self, event): ...
    def on_release(self, event): ...
    def render_document(self): ...
    def undo(self): ...
    def redo(self): ...
    def clear_canvas(self): ...
    def save_document(self): ...
    def load_document(self): ...

def main():
    root = tk.Tk()
    app = PaintApp(root)
    root.mainloop()

if __name__ == "__main__":
    main()
```

Uruchom aplikację u siebie

`python paint_app.py` w terminalu — albo Run przycisku w VS Code, albo PyCharm. Porównaj z `paint_app_basic.py` (Rozdział 18.7) to to samo płótno i ta sama idea, ale z zupełnie inną architekturą wewnętrzną.

Lista kontrolna gotowej aplikacji

MODEL

- dokument (`document`) — lista figur, źródło prawdy
- DrawingState są parametrami narzędzi, oddzielnymi od samych kształtów
- Canvas - Tylko wyświetlanie, przerysowane z dokumentu

NARZĘDZIA

- ołówek ciągłą kreską

- linia, prostokąt, owal z podglądem na żywo
- gumkę jako odrobinę koloru tła
- wybór narzędzia jest widoczny na ekranie, nie tylko w pamięci

HISTORIA

- undo/redo przez działania, a nie przez pojedyncze elementy Canvas
- nowe działanie oczyszcza redo_stack
- wyczyszczenie canvasa resetuje historię wraz z dokumentem

ZAPIS

- zapis i ładowanie rysunku przez JSON
- odwołania dialogów zapisu/otwarcia/wyboru koloru obsłużone szczerze
- łańcuch znaków stany i skróty klawiszowe

Most do rozdziału 19

Następnie: Przesuwaj się i powtarzaj zamiast czekać na kliknięcie

Rysowanka jest całkowicie zbudowana wokół oczekiwania na działania użytkownika — program reaguje, ale sam nie „żyje”. W następnym rozdziale — gra „Wąż” na Turtle — pojawi się własna pętla gry, w której program sam porusza obiektami klatka po klatce, niezależnie od tego, czy użytkownik w danym momencie coś klika.

Praktyka: Paint Pro w całości
Moduł tkinter otwiera natywne okno Python — wykonaj lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/18-32/index.html)

Podsumowanie rozdziału 18

Co zbudowaliśmy i czego się nauczyliśmy

- Canvas przechowuje elementy (items), nie piksele— create_* dodaje do lista rekord, który sam widżet zapamiętuje i zwraca item_id.
- Współrzędne Canvas rosną w prawo i w dół od lewego górnego rogu — nie jak w Turtle, gdzie początek jest w centrum, a Y rośnie w górę.
- Rysowanie myszą — gest z trzech zdarzeń: press zapamiętuje początek, drag aktualizuje szkic, release utrwala wynik.
- Podgląd na żywo — to JEDEN element, który tworzy się raz i aktualizuje przez coords(), zamiast tworzyć ponownie przy każdym ruchu.
- Prostokąt i owal są określone przez dwa przeciwległe punkty; owal jest wpisany w powstały prostokąt, a nie określony przez środek i promień.
- Undo/redo działają według logicznych DZIAŁAŃ użytkownika, a nie według poszczególnych elementów Canvas — jeden pociągnięcie ołówkiem jest cofane w całości.
- Dokument (lista figur) — źródło prawdy; Canvas za każdym razem jest rysowany od nowa z niego przez render_document(), ta sama zasada, co render() w rozdziale 17.
- Zachowuje się nie obrazek, a dokument — jak JSON, który można wczytać i odbudować w dowolnym momencie.

ROZDZIAŁ 19 · STR. 1216

Project: Snake gra z Turtle

Od pierwszego działającego prototypu po model gry z tikiem, pauzą, restartem, punktacją i weryfikowalnymi zasadami, klasyczny „Snake”

Gra Snake

Klasyczna gra o ruchu, jedzeniu i kolizjach na znajomym module turtle.

Prawdziwe okno: wąż z kilkoma segmentami, jabłko, partytura i płyta na ciemnym polu

Pod koniec rozdziału ta gra będzie działać na własnym modelu stanu, cyklu gry i weryfikowanych regułach.

Gra Snake

„Wąż” — jedna z najbardziej rozpoznawalnych gier w historii programowania. Wąż porusza się nieprzerwanie po planszy, zjada pojawiające się jabłka i rośnie z każdym zjedzonym jabłkiem; gra kończy się przy uderzeniu w ścianę lub w własny ogon. Zbudujemy ją na już znanym module `turtle` (rozdziały 6–7, 12) — tylko tym razem Żółw stanie się nie artystą, lecz postacią w grze.

Jak w rozdziale 18, plan znajomy: najpierw uczciwie działający prototyp proceduralny (sekcje 19.1–19.8), potem dokładna analiza struktury gry — siatka, kierunek jako dane, tik gry, stan, architektura (sekcje 19.9–19.33).

Importuj niezbędne moduły

importy.py

```
import random
import turtle
```

`random` będzie potrzebne do losowej pozycji jabłka (Rozdział 5), `turtle` dla samej grafiki.

Praktyka: Zaczynam budować grę
Moduł turtle otwiera natywne okno Python – uruchamiane lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/19-02/index.html)

Personalizacja ekranu Turtle

Wyłączamy automatyczną aktualizację ekranu i tworzymy zmienne, które będą pamiętać stan gry.

Personalizacja ekranu Turtle

gra potrzebuje przewidywalnego, kontrolowanego ekranu — i, co ważne, ekranu dla niepełnosprawnych Automatyczna aktualizacja: Obraz będziemy aktualizować ręcznie, dokładnie raz na każdy krok gry, a nie kiedy Przy każdym drobnym ruchu:

nastrojka_ekrana.py

```
screen = turtle.Screen()
screen.title(„Wąż”)
screen.bgcolor("black")
screen.setup(width=600, height=600)
screen.tracer(0)   # wyłącz automatyczną aktualizację
```

Po co tracer(0)?

Bez `tracer(0)` Turtle przerysuje ekran po *wszystkich* osobny ruch – w grze, w której musisz poruszać głową i kilkoma segmentami ciała na każdym kroku, wyglądałoby to na migoczącą i powolną. `tracer(0)` + Manual `screen.update()` na końcu każdego kroku dają znacznie płynniejszą animację.

Utworzenie i inicjalizacja niezbędnych zmiennych

peremennye.py

```
RAZMER_SHAGA = 20
GRANICA = 280

napravlenie = "stop"
schet = 0
igra_okonchena = False
segmenty = []   #snake ciało
```

CAPITAL_LETTERS - Konwencja dotycząca stałych

`RAZMER_SHAGA` oraz `GRANICA` nie zmieniają się podczas gry — zgodnie z konwencją takie "stałe" zmienne nazywane są zazwyczaj wielkimi literami. Samo w sobie technicznie nic nie zmienia, ale sygnalizuje czytelnikowi: "ta wartość nie powinna się zmieniać."

Faktyczne okno: ekran jest ustawiony, a zmienne wprowadzone. Jabłko i tablica na zdjęciu pojawią się w kodzie dopiero w sekcjach 19.3 i 19.5 — tutaj ważny jest sam ustawiony ekran.

Praktyka: Ustawianie ekranu i zmiennych

Moduł turtle otwiera natywne okno Python – uruchamiane lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/19-02/index.html)

Rysuj głowę

Głowa i jabłko to powszechne przedmioty Turtle o różnych kształtach i kolorach.

Rysuj głowę

Snake Head to powszechny przedmiot `turtle.Turtle()`, znajomy Od rozdziału 6, tylko z kwadratowym kształtem i bez narysowanych linii (pióro uniesione):

golova.py

```
golova = turtle.Turtle()
golova.speed(0)
golova.shape("square")
golova.color("white")
golova.penup()
golova.goto(0, 0)
```

Dobierz pierwsze jabłko

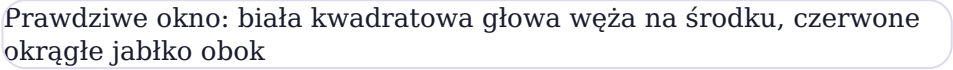
Jabłko to także żółw, tylko okrągły i czerwony, i pojawia się w losowym miejscu na polu, wyrównane do siatki schodkowej (`random.randrange()` z krokiem `RAZMER_SHAGA`, aby jabłko zawsze znajdowało się „w kratce”):

yabloko.py

```
yabloko = turtle.Turtle()
yabloko.speed(0)
yabloko.shape("circle")
yabloko.color("red")
yabloko.penup()

def novoe_yabloko():
    x = random.randrange(-GRANICA, GRANICA + RAZMER_SHAGA,
RAZMER_SHAGA)
    y = random.randrange(-GRANICA, GRANICA + RAZMER_SHAGA,
RAZMER_SHAGA)
    yabloko.goto(x, y)

novoe_yabloko()
```

Dlaczego GRANICA + RAZMER_SHAGA, a nie tylko GRANICA

U `randrange()` prawej granicy *jest wykluczony* — `randrange(-280, 280, 20)` daje wartości tylko do 260 włącznie. Jednocześnie wąż łatwo osiąga 280 (sekcja 19.19), więc bez `+ RAZMER_SHAGA` najbardziej zewnętrzny rząd komórek po prawej i górze byłby nieosiągalny dla jabłka, pole byłoby niezauważalnie asymetryczne.



Prawdziwe okno: głowa i jabłko — dwa niezależne obiekty Turtle, oba z podniesionym piórem.

Praktyka: Head and Apple

Moduł turtle otwiera natywne okno Python – uruchamiane lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/19-03/index.html)

Czy ekran rejestruje naciśnięcia strzałek na klawiszach?

Podłącz klawiaturę i poruszaj głową — chroniąc przed natychmiastowym obrotem o 180°.

Czy ekran rejestruje naciśnięcia strzałek na klawiszach?

Aby Turtle reagować na klawiaturę, musi ona mieć wyraźne pozwolenie na „słuchanie” `screen.listen()`) i przypisują konkretne klucze do funkcji za pomocą `onkeypress()` jest tym samym pomysłem co `.bind()` Tkinter w rozdziale 17, tylko z prostszym, bardziej wyspecjalizowanym interfejsem:

klavishi.py

```
def idti_vverh():
    global napravlenie
    if napravlenie != "down":   # nie może natychmiast obrócić
się o 180°
        napravlenie = "up"

def idti_vniz():
    global napravlenie
    if napravlenie != "up":
        napravlenie = "down"

#idti_vlevo() i idti_vpravo() są ułożone podobnie

screen.listen()
screen.onkeypress(idti_vverh, "Up")
screen.onkeypress(idti_vniz, "Down")
screen.onkeypress(idti_vlevo, "Left")
screen.onkeypress(idti_vpravo, "Right")
```

Dlaczego nie mogę po prostu obrócić się o 180°?
Jeśli wąż przesunie się w prawo, a gracz naciśnie "lewo", głowa natychmiast "przejedzie" po pierwszym segmentie jego ciała, co wyglądałoby jak natychmiastowa strata bez wyraźnego powodu. Weryfikacja `if napravlenie != "down":` zabrania obracania się na miejscu, pozwalając na obrót tylko o 90°.

Spraw, by głowa węża się ruszyła

dvizhenie_golovy.py

```
def dvigat_golovu():
    if napravlenie == "up":
        golova.sety(golova.ycor() + RAZMER_SHAGA)
    elif napravlenie == "down":
        golova.sety(golova.ycor() - RAZMER_SHAGA)
    elif napravlenie == "left":
        golova.setx(golova.xcor() - RAZMER_SHAGA)
    elif napravlenie == "right":
        golova.setx(golova.xcor() + RAZMER_SHAGA)
```

sety()/setx() — połowa od goto()
`sety(y)` zmienia jedynie współrzędną Y, pozostawiając X tym samym — wygodniej niż obliczanie obu współrzędnych przez `goto()` za każdym razem.

Rzeczywiste okno: głowa węża przesunęła się w prawo od środka pola o kilka kroków

Rzeczywiste okno: po kilku wywołaniach dvigat_golovu() z kierunkiem right.

Praktyka: Klucze i ruch głowy
Moduł turtle otwiera natywne okno Python – uruchamiane lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/19-04/index.html)

Strzel tablicę wyników

Osobny żółw bez kształtu, który pokazuje tylko tekst bieżącego rachunku.

Wygodnie jest pokazać partyturę jako osobnego żółwia bez formy (`hideturtle()`), który rysuje tylko tekst (`write()` z rozdziału 7):

tablo_scheta.py

```
tablo = turtle.Turtle()
tablo.speed(0)
tablo.color("white")
tablo.penup()
tablo.hideturtle()
tablo.goto(0, 260)

def obnovit_tablo():
    tablo.clear()   # usuwamy stary napis przed nowym
    tablo.write(f"Ocena: {schet}", align="center",
font=("Arial", 16, "normal"))

obnovit_tablo()
```

clear() przed write() — obowiązkowo
Bez `tablo.clear()` każde nowe liczenie nałożyło się na poprzednie, liczby szybko zamieniały się w nieczytelny bałagan.

Tablica wyników jest tuż na polu gry – na razie
`tablo.goto(0, 260)` umieszcza napis na współrzędnej `y = 260`, a to *legalna klatka*: wąż może przejechać przez tekst. W pierwszym prototypie jest to do przyjęcia, ale sekcja 19.32 przenosi tablicę wyników na osobny pas HUD nad boiskiem, gdzie wąż nie może przejechać zgodnie z zasadami.

Prawdziwe okno: Pole gry oznaczone „Count: 30”

Prawdziwe okno: obnovit_tablo() ponownie losuje licznik po każdym zjedzeniu jabłku.

Praktyka: Tablica wyników i jedzenie jabłek
Moduł turtle otwiera natywne okno Python – uruchamiane lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/19-06/index.html)

Nasz wąż je!

Zjedzone jabłko dodaje segment – a każdy segment musi prawidłowo podążać za poprzednim.

Nasz wąż je!

Sprawdź, czy główka jest wystarczająco blisko jabłka (metoda `.distance()` liczy odległość między dwoma żółwiami) — jeśli Tak, jabłko jest „zjadane”

eda.py

```
def proverit_edu():
    global schet
    if golova.distance(yabloko) < RAZMER_SHAGA:
        novoe_yabloko()
        dobavit_segment()
        schet += 10
        obnovit_tablo()
```

Zmuszamy całą węża do ruchu

Każdy segment ciała powinien zajmować miejsce *poprzednich* segment (a pierwszy to miejsce głów) — tworzy to efekt "wąż pełza jako całość" zamiast "części poruszają się niezależnie":

dobavit_segment.py

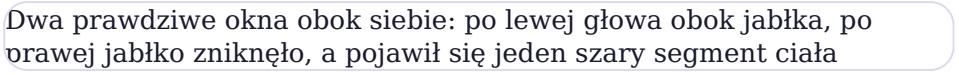
```
def dobavit_segment():
    novyj = turtle.Turtle()
    novyj.speed(0)
    novyj.shape("square")
    novyj.color("grey")
    novyj.penup()
    segmenty.append(novyj)
```

dvigat_telo.py

```
def dvigat_telo():
    # zacznij od ogona i przejdź do głowy, aby nie tracić
pozycji po drodze
    for indeks in range(len(segmenty) - 1, 0, -1):
        x = segmenty[indeks - 1].xcor()
        y = segmenty[indeks - 1].ycor()
        segmenty[indeks].goto(x, y)

    if segmenty:
        segmenty[0].goto(golova.xcor(), golova.ycor())
```

Kolejność wywołań decyduje o wszystkim

`dvigat_telo()` jest koniecznie nazywane **do** `dvigat_golovu()` — w przeciwnym razie pierwszy segment już „odziedziczy” *nowy* pozycję głowy zamiast starej, a ciało zlepi się z głową w jeden punkt.



Dwa prawdziwe okna obok siebie: po lewej głowa obok jabłka, po prawej jabłko zniknęło, a pojawił się jeden szary segment ciała

Prawdziwe okno przed i po proverit_edu(): zjedzeniu jabłka, wyniku wzrosł, a segment został dodany.

Praktyka: jedzenie jabłek i poruszanie się ciałem

Moduł turtle otwiera natywne okno Python – uruchamiane lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/19-06/index.html)

Kontrola kolizji

Gra kończy się, gdy zderzysz się ze ścianą lub własnym ogonem.

Gra kończy się przy jednym z dwóch kolizji: ze ścianą pola lub z własnym ciałem.

stolknoveniya.py

```
def proverit_stolknoveniya():
    global igra_okonchena

    # zderzenie ze ścianą
    if abs(golova.xcor()) > GRANICA or abs(golova.ycor()) >
GRANICA:
        igra_okonchena = True

    # zderzenie z własnym ciałem
    for segment in segmenty:
        if segment.distance(golova) < RAZMER_SHAGA / 2:
            igra_okonchena = True
```

abs() ponownie oszczędza kod

`abs(golova.xcor()) > GRANICA` sprawdza obie ściany (lewą i prawą) jednocześnie jednym porównaniem — zamiast `golova.xcor() > GRANICA or golova.xcor() < -GRANICA`. Sama wbudowana funkcja `abs()` jest znany z sekcji 5.4 — tutaj po raz pierwszy działa nie jako „moduł liczby”

Prawdziwe okno: Głowa Snake'a na prawym krańcu ciemnego pola, wynik pozostaje taki sam

Prawdziwe okno: głowa wychodziła poza GRANICA – proverit_stolknoveniya() igra_okonchena trafiła do True.

★★ Zadanie samodzielne

Przyspieszenie gry

Zwiększ prędkość ruchu (zmniejszaj opóźnienia między krokami) o każde 50 punktów – gra stopniowo stanie się trudniejsza.

Praktyka: Kolizje

Moduł turtle otwiera natywne okno Python – uruchamiane lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/19-07/index.html)

Pełny kod pierwszego prototypu

Złożenie wszystkich części w jedną pętlę gry — i przejście od szczerego prototypu do dokładnej analizy tego, jak gra naprawdę działa.

Zbieraj jeden krok gry ze wszystkich części rozdziału – ta cecha jest sercem gry, Powtarzane wielokrotnie, aż gra się zakończy:

igrovoj_shag.py

```
import time

ZADERZHKA_SEK = 0.12   #pause między krokami

def igrovoj_shag():
    if igra_okonchena:
        return False

    dvigat_telo()
    dvigat_golovu()
    proverit_edu()
    proverit_stolknoveniya()
    screen.update()
    return not igra_okonchena

def glavnyj_cikl():
    global napravlenie
    napravlenie = "right"
    while igrovoj_shag():
        screen.update()
        time.sleep(ZADERZHKA_SEK)
    tablo.goto(0, 0)
    tablo.write(f"Koniec gry! Wynik: {schet}", align="center",
font=("Arial", 20, "bold"))
```

Bez time.sleep() gra kończyłaby się, zanim zdążyłeś nacisnąć klawisz

Cykl `while` obraca się z prędkością Python, a nie ludzką: od środka do ściany tylko 14 kroków i bez pauzy przechodzą w około dwie tysięczne sekundy — okno zdążyłoby tylko mrugnąć napisem „Koniec gry! Wynik: 0”. `time.sleep(0.12)` wyznacza ludzkie tempo: około ośmiu kroków na sekundę. Tutaj `sleep()` jest uzasadniony właśnie dlatego, że pętla i tak jest blokująca — dopóki działa, w programie i tak nic innego się nie dzieje, a `screen.update()` na końcu każdego kroku udaje się uporządkować zgromadzone naciśnięcia klawiszy. W architekturze, `screen.ontimer()` (sekcja 19.13) ten sam `sleep()` już będzie błędem – Debug Lab 4 w sekcji 19.31 wyjaśnia dlaczego.

Real Window: Zmontowana gra – Ocena 30, Wąż z Trzech Segmentów, Jabłko na polu

Prawdziwe okno pierwszego prototypu: wszyscy razem – ruch, jedzenie, liczenie.

Kompletny, już złożony i przetestowany program pierwszego prototypu jest dostępny jako osobny plik:

projects/turtle/snake/snake_basic.py

Uruchom grę u siebie

`python snake_basic.py` w terminalu — sterowanie klawiszami strzałek.

Szczera, ale nie ostateczna wersja

Zbudowaliśmy prawdziwą działającą grę — z ruchem, jedzeniem, wynikiem i kolizjami. Ale `while igrovoj_shag(): ...` jest pętlą blokującą z stałym opóźnieniem: prędkości nie można zmieniać podczas gry, a pauza i restart musiałyby być wplecione bezpośrednio w ciało pętli. Cały stan znajduje się w globalnych zmiennych modułu. Zaczynając od następnej sekcji, rozdział przygląda się grze bliżej: czym jest siatka i odtyk gry, jak zorganizować pauzę i restart bez drugiego równoległego łańcucha timerów oraz jak rozwinąć architekturę poziomu na podstawie tego prototypu `SnakeApp` (Rozdział 19.32).

Challenge

Poziomy trudności

Dodaj wybór trudności przed grą (input() z rozdziału 8) – który wpływa na początkową szybkość przez opóźnienie między krokami.

Praktyka: Kompletna gra

Moduł turtle otwiera natywne okno Python – uruchamiane lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/19-08/index.html)

Wyniki pierwszego prototypu

Czego nauczyliśmy się w pierwszych ośmiu sekcjach

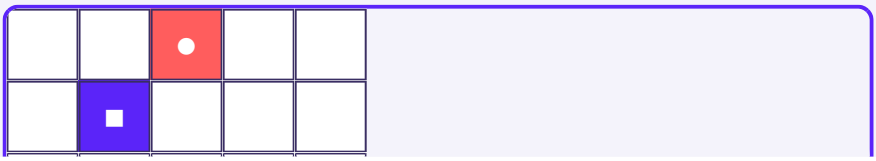
- `screen.tracer(0)` + Manual `screen.update()` — standardowy trik dla płynnej animacji w grach na Turtle.
- `screen.onkeypress()` łączy strzałki z funkcjami kierunkowymi; Zakaz zawracania o 180° zapobiega natychmiastowej kolizji samobójczej.
- Ciało węża — lista poszczególnych segmentów, z których każdy podąża za poprzednim; kolejność aktualizacji (najpierw ciało, potem głowa) jest krytycznie ważna.
- `.distance()` między dwoma żółwiami to wygodny sposób, by sprawdzić, czy są „wystarczająco blisko”
- Pętla gry to funkcja wywoływana wielokrotnie, aż do osiągnięcia warunku końcowego gry.
- To dopiero pierwszy prototyp – potem rozdział analizuje siatkę, odznaczenie, stan i architekturę znacznie dokładniej.

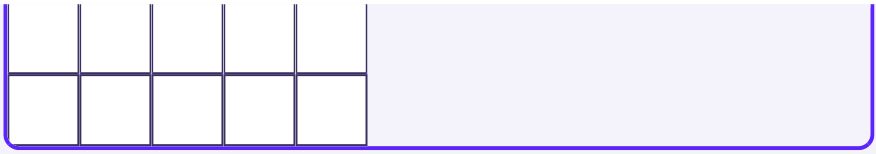
Świat gry jako siatka

Głowa, ciało i jabłko zawsze znajdują się w węźle sieci – ruch to przejście do sąsiedniego węzła, a nie poślizgnięcie.

Wąż żyje nie w dowolnych współrzędnych

Pierwszy prototyp już porusza się głową w 20px odstępach, ale łatwo przeoczyć dlaczego Dokładnie: wąż nie może być w punkcie (137, 52) — tylko w punktach będących wielokrotnościami `RAZMER_SHAGA`. Pole gry — to nie płótno, na którym można narysować linię pod dowolnym kątem (jak Canvas w rozdziale 18), a **Dyskretna siatka**: skończony zestaw pól, a każdy obiekt gry — głowa, segment ciała, jabłko — zawsze znajduje się dokładnie na jednym z nich. Obok słowa „siatka” dalej w rozdziale pojawi się też „kratka” — to to samo: węzły kratki to komórki siatki.





Legalne stanowiska to węzły siatki w RAZMER_SHAGA krokach, a nie w każdym punkcie na polu.

Przy `RAZMER_SHAGA = 20` współrzędne prawne na każdej osi — `..., -40, -20, 0, 20, 40, ...`. Posunięcie się o jeden krok to przejście do do sąsiedniego węzła kratowego, zamiast gładkiego przesuwania się na dowolnej odległości.

To ten sam pomysł co w rozdziale 18 — ale brak pikseli między komórkami

Canvas również działał w całych pikselach, ale figurę można było przesunąć o 1 piksel. Wąż ma stały krok: `RAZMER_SHAGA` — to nie „minimalna jednostka ekranu", lecz sama jednostka świata gry. To definiuje wszystko inne w rozdziale: jak rozmieszczana jest żywność, jak wykrywane jest zderzenie, jak wygląda ruch ciała.

Praktyka: Stanowiska prawne w siatce

Automatic check – funkcja is_on_grid():, czy punkt jest wielokrotnością kroku siatki

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/19-09/index.html)

Współrzędne komórek i piksele Turtle

Nie jest potrzebne osobne tłumaczenie „komórka → piksel"

Komórka siatki vs pikselowa Turtle

Turtle nie ma osobnego pojęcia „komórki" `(x, y)`, te same, co w rozdziałach 6–7. Można by wprowadzić osobny układ współrzędnych komórek — `col = x // STEP`, `row = y // STEP` — i przekształcać jedno w drugie przy każdym odwołaniu do ekranu.

pixel_to_cell.py

```
def cell_to_pixel(col, row, step=20):
    return col * step, row * step

def pixel_to_cell(x, y, step=20):
    return x // step, y // step
```

Dzielenie całkowite liczb ujemnych — to nie to samo co zaokrąglanie w stronę zera

`-10 // 20` w Python równa się `-1`, nie `0` jest Python zaokrągla dzielenie w dół (do minus nieskończoności) zamiast do zera. Wzór na konwersję pikseli na komórki, zapisany bez uwzględnienia tego, będzie nieprawidłowy dokładnie w połowie pola — tej o ujemnych współrzędnych.

Dla tego rozdziału wybrano prostsze rozwiązanie: logiczne położenie węża i jabłka to Są to współrzędne pikselowe Turtle, już wielokrotności `STEP`. Oddzielnie Nie ma w ogóle systemu „klatki (col, row)” `(x, y)` jednocześnie Co przechowujemy w modelu i co rozumie `turtle.goto()`.

logicheskaya_poziciya.py

```
STEP = 20
head = (0, 0)          # już pikseli Turtle — i już pozycja
komórki
head = (head[0] + STEP, head[1])
# krok w prawo — natychmiast w pikselach
```

Mniej kodu oznacza mniej miejsca na błędy

Tłumaczenie „cell-pixel ↔” `STEP` to tłumaczenie nie dodaje żadnych nowych informacji — tylko ryzyko błędu zaokrąglenia odwrotnego z powyższego przykładu.

Praktyka: Komórka i piksele

Automatic Check – funkcja pixel_to_cell(): zaokrąglenie współrzędnych na komórki o liczbach ujemnych

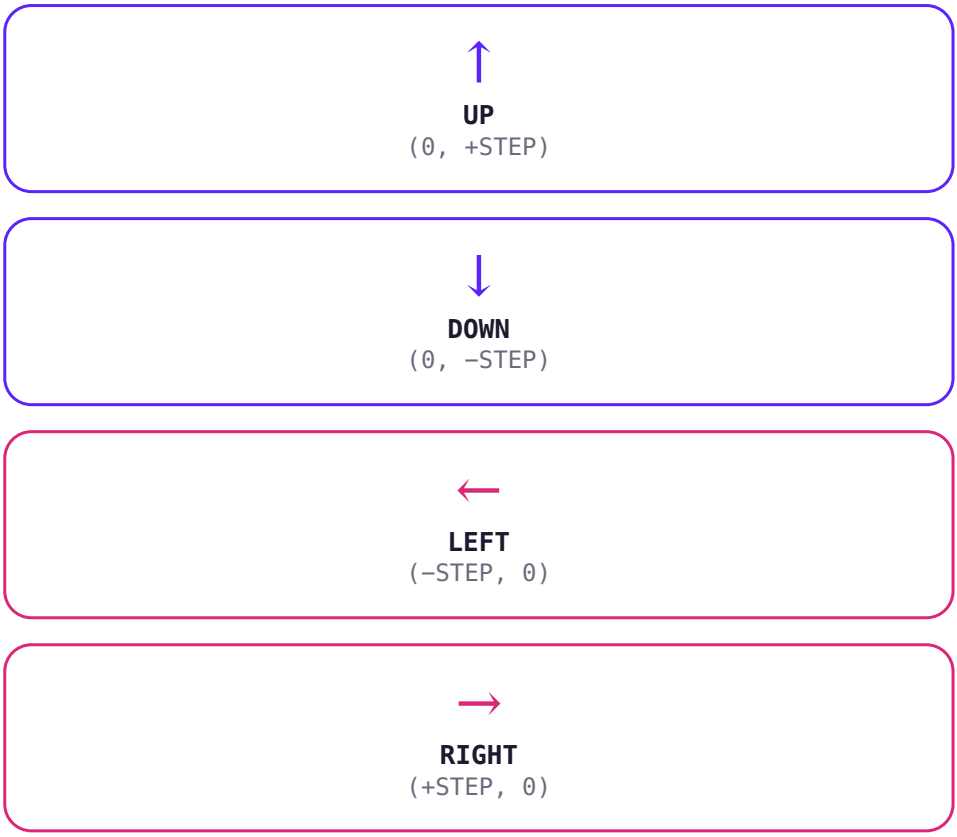
Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/19-10/index.html)

Kierunek jako wektor

Kierunek to nie łańcuch znaków porównywany w if/elif, lecz dane: para liczb (dx, dy).

Od czterech if/elif do jednego słownika

Pierwszy prototyp definiował ruch głowy za pomocą czterech gałęzi `if/elif` (Rozdział 19.4) — czytelna, ale gdy dodana jest nowa logika (na przykład ruch diagonalny w inna gra) musiałaby powtórzyć strukturę. Wskazówki można przedstawić inaczej — Jak to zrobić **wektor przemieszczenia**:



direction_vectors.py

```
DIRECTION_VECTORS = {
    Direction.UP: (0, STEP),
    Direction.DOWN: (0, -STEP),
    Direction.LEFT: (-STEP, 0),
    Direction.RIGHT: (STEP, 0),
}

def next_head(head, direction):
    dx, dy = DIRECTION_VECTORS[direction]
    return (head[0] + dx, head[1] + dy)
```

if/elif kontra słownik wektorów

Jawnie, ale rośnie wraz z liczbą kierunków

```
if napravlenie == "up":
    golova.sety(golova.ycor()
+ RAZMER_SHAGA)
elif napravlenie == "down":
    golova.sety(golova.ycor()
- RAZMER_SHAGA)
# ... I tak dla left, right
```

Jedna słownik, jedna formuła

```
dx, dy =
DIRECTION_VECTORS[direction]
new_x, new_y = x + dx, y + dy
```

Co używać dzisiaj: Obie opcje dają taki sam wynik dla czterech kierunków. Różnica pojawia się w tym, jak kod rośnie: słownik — to dane, a nie rozgałęzienie, więc testowanie i rozwijanie jest łatwiejsze, bez zmieniania samej formuły ruchu.

Praktyka: kierunek jako wektor
Automatyczna kontrola — funkcja next_head() dla wszystkich czterech kierunków

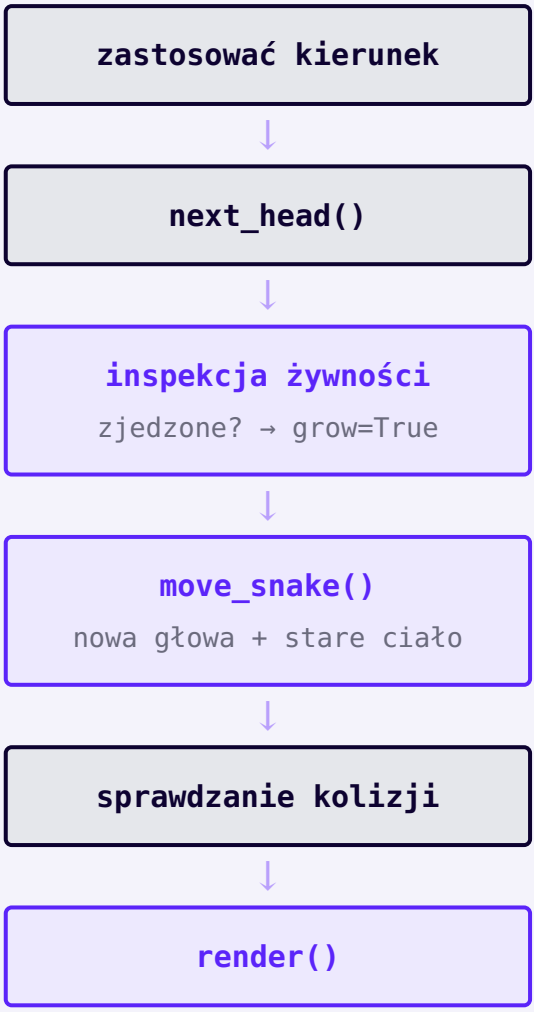
Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/19-11/index.html)

Jeden odznak gry

Takt — to cały ciąg reguł w całości, a nie tylko jeden ruch głowy po ekranie.

Tick gry – Jedna konkretna aktualizacja gry

Tick gry to nie „jedna klatka animacji”



Jeden tik — to cały ten ciąg w całości, a nie jeden ruch głowy.

Tick i render to różne pojęcia, nawet jeśli występują razem

W tej prostej grze każdy odstawk kończy się losowaniem — ale to decyzja, a nie obowiązkowa zasada. `game_tick()` odpowiada za ZASADY: co zmieniło się w modelu. `render()` odpowiada jedynie za wyświetlanie aktualnego modelu na ekranie. Rozdział 19.29 szczegółowo rozłoży tę granicę — stanie się ona ważna, gdy logika zostanie przetestowana bez Turtle (Rozdział 19.26).

Praktyka: Kolejność kroków o jednym ticku
Automatyczna kontrola – funkcja tick_order(): tego, co następuje najpierw, kontrola jedzenia lub składanie ciała

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/19-12/index.html)

Prawdziwa pętla rozgrywki

screen.ontimer() planuje kolejne tiknięcie i natychmiast przywraca kontrolę — zamiast blokować program w while.

while nie jest jedynym sposobem na powtarzanie tików

Pierwszy prototyp prowadzi grę przez cały proces `while igrovoj_shag(): ...` z `time.sleep()` w ciele to prosty i prosty cykl blokowania. Łatwo wyciągnąć tu błędne wnioski, więc warto opowiedzieć o tym osobno: `screen.update()` wewnątrz pętli FAKTYCZNIE przetwarza zgromadzone Tkinter zdarzeń, w tym naciśnięcia klawiszy, oznaczają, że handler jest związany przez `onkeypress()`, technicznie może zadziałać bezpośrednio podczas tego wywołania. Problem takiej pętli nie polega na tym, że klawiatura „nie nadąża” — ale na tym, że on *posiada* zarządzanie od początku do końca gry i nikogo o to nie pyta rozdzielczość. Prędkość w niej to jedna stała `ZADERZHKA_SEK`, którego nie można zmienić w trakcie gry; a pauza i restart musiałyby być ręcznie wplecione w ciało pętli z osobnymi flagami i kontrolami w każdej iteracji. I przez cały ten czas program nie może Nic więcej: ona siedzi w środku `sleep()`.

ontimer_cikl.py

```
def game_tick(self):
    if self.state.status is not GameStatus.RUNNING:
        return
    # ...zastosować kierunek, przesunąć głowę, sprawdzić
jedzenie i kolizje...
    self.render()
    self.screen.ontimer(self.game_tick, self.state.delay_ms)
```

screen.ontimer() planuje kolejny tick zamiast blokować program

`screen.ontimer(callback, delay_ms)` prosi cykl zdarzeń Tkinter (ten sam, co w rozdziale 17) o wywołanie `callback` około `delay_ms` — i natychmiast wraca do kontroli. Między taktami program jest całkowicie wolny: klawiatura, zmiany pauzy i prędkości są przetwarzane jako zwykłe zdarzenia i nie czekają na swoją kolej w pętli.

time.sleep() w pętli Tkinter/Turtle gry to zły pomysł

`time.sleep()` zatrzymuje *wszystkim* proces, włączając obsługę zdarzeń — okno przestanie odpowiadać na klawiaturę i zamykanie dokładnie na czas uśpienia. `screen.ontimer()` niczego nie blokuje: przekazuje zegar do pętli zdarzeń, która w tym czasie nadal obsługuje wszystko inne.

Każde wyzwanie `game_tick()` sam planuje następny przez `ontimer()` — powstaje łańcuch, a nie cykl w zwykłym znaczeniu. Tutaj jest subtelność: `ontimer()` planuje tylko przyszłe wyzwanie — Nie usuwa się samodzielnie, jeśli status gry się zmieni. Rozdział 19.22 wyjaśnia dlaczego Dlatego pauza nie może polegać wyłącznie na weryfikacji `status` wewnątrz ticku — i co się stanie, jeśli polegać tylko na niej.

Praktyka: Pętla gry na ontimer()

Moduł turtle otwiera natywne okno Python – uruchamiane lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/19-13/index.html)

Czas, prędkość i opóźnienie

delay_ms — nie dokładny czas, ale przybliżony odstęp między tyknięciami, który zmniejsza się wraz ze wzrostem wyniku.

---

delay_ms nie jest gwarancją, lecz prośbą

`screen.ontimer(game_tick, 120)` oznacza "wołanie `game_tick` po około 120 milisekundach od zdarzenia będzie w stanie to zrobić" — a nie "dokładnie w 120 000 ms." Jeśli pętla zdarzeń jest teraz zajęta coś innego (np. przerysowanie), połączenie będzie nieco opóźnione. Jak na mecz takiej skali Różnica jest zwykle niezauważalna, ale nie warto obiecywać dokładnej godziny.

delay_ms	Sensation
200	wolno — wygodne na pierwsze zapoznanie się z zarządzaniem
140	normalna prędkość to wartość domyślna (BASE_DELAY_MS)
120	szybkość po zdobyciu 100 punktów – gra jest już zauważalnie szybsza
70	szybki – wymaga precyzyjnych i szybkich reakcji

calculate_delay.py

```
def calculate_delay(score, *, base_ms=140, min_ms=60,
step_score=50, step_ms=10):
    steps = score // step_score
    return max(min_ms, base_ms - steps * step_ms)
```

max() — obowiązkowa ochrona przed zerem i ujemnym opóźnieniem

Bez `max(min_ms, ...)` wystarczająco wysoki wynik prędzej czy później doprowadziłby do opóźnienia równego zero lub liczby ujemnej. `screen.ontimer(callback, 0)` formalnie nie spadnie, ale gra stanie się niegrywalna szybko; wartość ujemna to zachowanie, którego nie powinno się testować w praktyce. `min_ms` to twarda dolna granica prędkości.

Praktyka: opóźnienie liczenia

Automatyczna kontrola – funkcja calculate_delay(): opóźnienie spada wraz z wynikiem i nie spada poniżej minimalnego poziomu

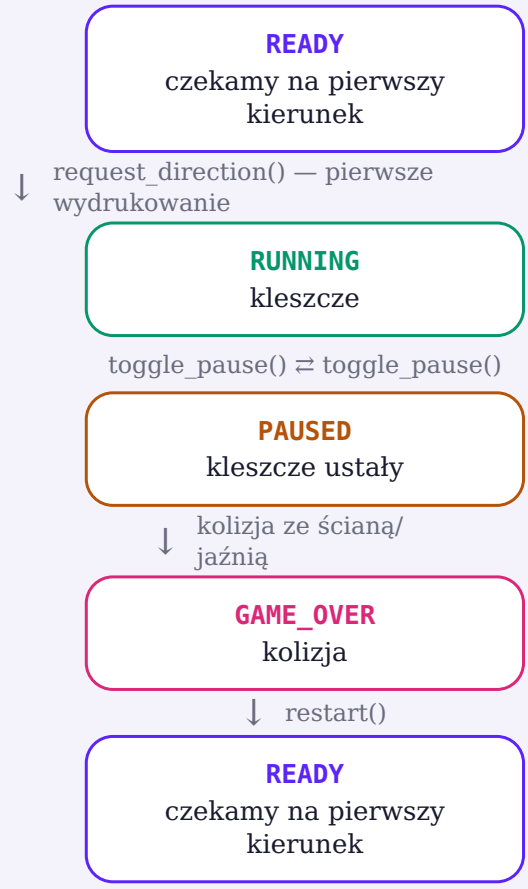
Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/19-14/index.html)

Stan gry

Nie tylko „gra trwa lub się skończyła”

Gra nie tylko "idzie" lub "nie idzie"

Pierwszy prototyp znał tylko `igra_okonchena` — Bulevo tak/nie. Ostateczna wersja rozróżnia cztery stany zwykłej gry, a przejścia między nimi są następujące: To nie jest przypadek, ale jasne zasady:



Status	Co się dzieje
READY	wąż jest na miejscu, poczekaj na pierwszy nacisk – kleszcze jeszcze nie odchodzą

Status	Co się dzieje
RUNNING	kleszcze są planowane jeden po drugim przez ontimer()
PAUSED	tiknięcia się zatrzymają, model się nie zmienia, ekran pokazuje „PAUSE”
GAME_OVER	doszło do kolizji; czekamy na restart()
WON	pole jest pełne, nie ma już pustej klatki na jedzenie - Rozdział 19.18

Piąty stan jest rzadki, ale prawdziwy
B `GameStatus` programie końcowym jest pięciu członków, a nie czterech: oprócz czterech stanów regularnej partii, są `WON` — wąż zajął dosłownie każdą komórkę pola. Jest to prawie nieosiągalne w prawdziwej grze, ale jest to pełny stan terminalny, a nie „prawie GAME_OVER”: rozdział 19.18 pokazuje, skąd pochodzi, a rozdział 19.21 — czym stany terminalne są podobne do siebie.

GameStatus jak Enum — ta sama idea, co Tool w rozdziale 18
Cztery stany to ostateczny znany lista, więc `Enum` tutaj jest tak samo uzasadnione jak `Tool` w przypadku narzędzi do rysowania: edytor wykryje literówkę w nazwie państwa, a nie cichy błąd w trakcie gry.

Praktyka: przejścia stanów gry
Automatyczne sprawdzanie – funkcja can_transition(): której dozwolone są przejścia między READY/RUNNING/PAUSED/GAME_OVER

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/19-15/index.html)

Głowa, ciało i model Snake

Lista współrzędnych zamiast listy obiektów ZHHK1ZKHK — snake[0] to głowa, reszta to ciało.

Wąż jako lista pozycji, a nie lista żółwi

Pierwszy prototyp przechowuje nadwozie jako lista *obiekty Turtle* jest stan gry, oraz Grafika jest ze sobą nierozerwalnie powiązana, rozdział 18 już rozwiązał ten sam problem w Canvas (Rozdział 18.8). Ostateczny model przechowuje węża jako regularną współrzędną lista:

model_snake.py

```
snake = [
    (0, 0),      # snake[0] — Głowa
    (-20, 0),
    (-40, 0),
]
```

snake[0] — głowa; snake[1:] — ciało. Żadnego obiektu Turtle w środku.

Renderowanie zamienia pozycje na Turtle- segmenty, a nie odwrotnie

Model nic nie wie o tym, jak wygląda wąż na ekranie. `render()` (Rozdział 19.29) brzmi `snake` i rozkłada współrzędne na już istniejące Turtle- segmenty — to samo rozdzielenie „model ≠ widżety”, co i `render_document()` w PaintApp rozdziału 18.

Praktyka: lista pozycje jako modelowy Snake

Automatyczna kontrola – działa head_of(), a body_of(): snake[0] jest głową, snake[1:] jest ciałem

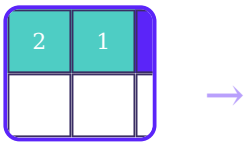
Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/19-16/index.html)

Dlaczego ciało porusza się od ogona

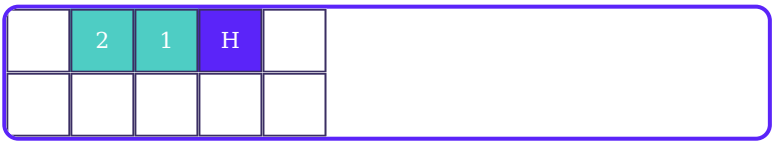
Aktualizacja w odwrotnej kolejności zwija wszystkie segmenty w jeden punkt — wyjaśnimy, dlaczego.

Kolejność aktualizacji segmentów nie jest szczegółem, lecz koniecznością

Rozdział 19.6 już pokazała zasadę: ciało jest aktualizowane od ogona, każdy segment zajmuje Lokalizacja *poprzednich*. Zobaczymy, co się stanie, jeśli zmienimy kolejność na przeciwną — od głowy do ogona:



był: H-S1-S2



stało się (poprawnie): każdy odziedziczył pozycję poprzedniego

ot_hvosta.py

```
def dvigat_telo():
    for indeks in range(len(segmenty) - 1, 0, -1):
        #segmenty[indeks] dostaje STARE stanowisko
segmenty[indeks - 1]
        x = segmenty[indeks - 1].xcor()
        y = segmenty[indeks - 1].ycor()
        segmenty[indeks].goto(x, y)
```

Od głowy do ogona — segmenty łączą się w jeden punkt

Jeśli pójdziemy w odwrotnej kolejności, `segmenty[0]` jako pierwszy otrzyma pozycję głowy. W następnej iteracji `segmenty[1]` odczyta współrzędne `segmenty[0]` — ale to jest NOWA, tylko nadpisana pozycja, a nie stara. Po kilku iteracjach wszystkie segmenty będą w tym samym punkcie — wąż wizualnie zsuwa się w kwadrat.

Rozdział 19.18 pokaże bardziej elegancką alternatywę dla tego cyklu, „nowa głowa + starego korpusu”

Praktyka: właściwa kolejność odnowy ciała
Automatyczna weryfikacja — funkcje move_from_tail() i move_from_head_BROKEN():, kolejność aktualizacji decyduje o wszystkim

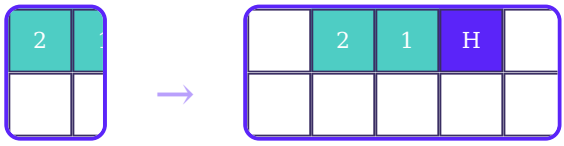
Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/19-17/index.html)

Food and Free Cell

Nowa głowa plus stare ciało budują ruch za jednym przejściem — a ta sama zasada wielu wolnych komórek decyduje, gdzie uczciwie umieścić jedzenie.

Elegancka alternatywa dla cyklu segmentowego

Zamiast przenosić każdy istniejący segment osobno (Rozdział 19.17), Można zbudować zupełnie nową lista – nową głowę przed sobą, stare ciało za nimi, bez Ostatni element (jeśli wąż nie urośnie):



snake to [nowy _head] + snake[:-1]

move_snake.py

```
def move_snake(snake, new_head, *, grow):
    if grow:
        return [new_head, *snake]
    return [new_head, *snake[:-1]]
```

Jednocześnie ogon naturalnie „opada” `snake[:-1]` odrzuca ostatni element. Brak pętli, brak ryzyka zamieszania kolejności aktualizacji: Efekt powstaje w jednym przejściu, zamiast zmieniać istniejące pozycje jedna po drugiej.

Jedzenie to losowa wolna komórka

W rozdziale 19.3 komórka na jabłko była wybierana bez sprawdzenia — przy dość długim wężu jabłko czasem znajdowało się bezpośrednio w ciele. Uczciwa wersja wybiera tylko spośród naprawdę wolnych komórek:

choose_food.py

```
def choose_food(snake, rng, *, half=280, step=20):
    occupied = set(snake)
    free = tuple(
        cell for cell in all_cells(half=half, step=step) if
cell not in occupied
    )
    if not free:
        return None
    return rng.choice(free)
```

rng jako parametr – ten sam pomysł przyda się w sekcji 19.30
`choose_food()` przyjmuje gotowy `random.Random` zamiast czytać globalne `random` moduł bezpośrednio. W prawdziwej grze, `random.Random()` jest prawdziwym wypadkiem; w testach — `random.Random(seed)` z ustalonym ziarnistością, aby wynik był przewidywalny i możliwy do zweryfikowania `assert` -om.

Niezmiennik: pokarm — tylko z komórek wolnych

Kontrakt `choose_food()` można zapisać wzorem: `food ∈ FREE_CELLS`, gdzie `FREE_CELLS = ALL_CELLS − SNAKE_CELLS`. Musi pozostać prawdziwe ZAWSZE, w tym skrajne przypadki, które rzadko są osiągane w praktyce, ale które Trzeba zapewnić: wąż urósł tak bardzo, że zajmuje dosłownie każdą komórkę pola. Potem `FREE_CELLS` pusto, i uczciwej odpowiedzi, gdzie położyć jedzenie, nie ma.

FREE_CELLS

`food ∈ FREE_CELLS`

Jeśli FREE_CELLS pusty, nie ma fizycznej wolnej klatki do jedzenia.

Nie możesz potajemnie zastąpić klatki dla węży None
Kuszącą, ale błędną decyzją jest na przykład powrót, `snake[0]`, gdy nie ma wolnych komórek: wtedy `food == snake[0]`, a niezmiennik „jedzenie nigdy nie jest w wężu” `None`: Nie ma faktycznie darmowej klatki na jedzenie, a osoba wywołująca kod (Rozdział 19.32) musi to wyraźnie załatwić, zamiast udawać, że jedzenie nadal istnieje.

B `SnakeApp.game_tick()` wygląda to tak: jeśli wąż zjadł jedzenie i `choose_food()` powrócił `None`, gra przechodzi w stan końcowy `GameStatus.WON` — Stawka jest całkowicie zajęta, nie ma dokąd się posunąć, a to jest zwycięstwo, a nie błąd. `state.food` w tym stanie również staje się `None` oraz rendering (sekcja 19.29) muszą to wyraźnie zweryfikować zanim narysuje jedzenie na ekranie.

Rzeczywiste okno: pole gry całkowicie wypełnione ciałem węża, na górze tablica wyników, na środku napis ZWYCIĘSTWO i końcowy wynik

Rzeczywiste okno po tym, jak wąż uczciwie zajął każdą legalną komórkę pola — GameStatus.WON, a nie jedzenie wewnątrz ciała.

Praktyka: jedzenie tylko na wolnym polu
Automatyczna kontrola – move_snake() i choose_food(): działa jedzenie nigdy nie znajduje się w wężu

`Otwórz praktykę →` (https://www.cartesianschool.org/pl/practice/19-18/index.html)

Zderzenie ze ścianą

GRANICA jest granicą współrzędnej środka głowy, a nie rozmiarem okna; ramka jest uwzględniona w obszarze prawnym.

Granica — współrzędna środka segmentu, a nie krawędzi ekranu

`GRANICA = 280` — to nie jest rozmiar okna (600×600), a limit dla współrzędnej *centrum* głowy. O grubości segmentu 20 px, głowica ze środkiem przy dokładnie 280 stopień nadal mieści się całkowicie na polu widzialnym; przy 300 stopniach już przekracza region.

is_wall_collision.py

```
def is_wall_collision(head, *, half=280):
    x, y = head
    return abs(x) > half or abs(y) > half
```

Pozycja Głowy	is_wall_collision()
(280, 0)	False jest granica prawna
(-280, -280)	False — legalna granica, róg pola
(300, 0)	True — już poza granicami

Ścisłe porównanie (>), nie (>=) - uwzględnione granice
Gdyby weryfikacja używała `>=`, sytuacja prawna `280` już byłoby uznane za kolizję – wąż nie byłby w stanie przejechać na sam skraj pola. Rozdział 19.30 bada te przypadki graniczne jako osobne testy, a nie tylko słowami.

Praktyka: Granica boiska
Automatyczna weryfikacja – is_wall_collision() funkcja na granicy i poza nią

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/19-19/index.html)

Zderzenie z samym sobą

Musisz sprawdzić ciało po przeprowadzce, a nie przed nim – w przeciwnym razie legalne wejście do klatki pustego ogona zostanie błędnie uznane za kolizję.

Wjechnięcie w klatkę tylną nie zawsze kończy się kolizją

Klasyczna zasada „wężyk”: jeśli wąż nie rośnie, ogon w tym samym takcie zwalnia swoją komórkę — jazda tam głową jest legalna. Jeśli rośnie — ogon nigdzie się nie rusza, i ta sama komórka jest już zajęta.

is_self_collision.py

```
def is_self_collision(new_head, body_after_move):
    # body_after_move – ciało PO move_snake(), bez samej głowy
    return new_head in body_after_move
```

Musisz sprawdzić ciało PO ruchu, nie przed nim

Jeśli sprawdzisz `new_head in snake[1:]` na liście OLD (do `move_snake()`), komórka starego ogona nadal będzie na liście — i legalny wjazd w nią błędnie zostanie policzony jako kolizja. Poprawne sprawdzenie odbywa się wobec `move_snake(...)[1:]` jest ciało takim, jakim stanie się PO tej turze, a nie taką, jaka była przed nim.

Sytuacja	is_self_collision()
Głowa wjeżdża do klatki starego ogona, wąż nie rośnie	False — ogon już zwolnił komórkę
Głowa wjeżdża w kratkę starego ogona, wąż rośnie	True — ogon nigdzie nie zniknął
Głowa wsuwa się do klatki na środku ciała	True prawdziwy konflikt

Praktyka: Zderzenie z własnym ciałem

Automatyczna kontrola – funkcja is_self_collision(): wjeżdżania w klatkę tylną, powiedzmy, na środku kadłuba – nie

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/19-20/index.html)

Game Over jako państwo

Kolizja nie tylko zatrzymuje ruch, ale przenosi grę do osobnego stanu GAME_OVER.

Game Over — stan, a nie tylko napis na ekranie

Prawdziwe okno: S--kształtny wąż, duży napis GAME OVER i licznik na wierzchu pola

Rzeczywiste okno: zderzenie z własnym ciałem przenosi grę do GAME_OVER — ta sama inskrypcja pojawiłaby się również przy zderzeniu ze ścianą.

Kiedy `is_wall_collision()` lub `is_self_collision()` True odliczanie tyka nie tylko zatrzymuje ruch — on tłumaczy `state.status` w `GAME_OVER` i rysuje nakładkę na ostatniej klatce gry:

game_over.py

```
if is_wall_collision(head):
    state.status = GameStatus.GAME_OVER
    self.render()
    self._show_overlay("GAME OVER", f"Ocena:
{state.score}  | R to nowa gra")
    return
```

GAME_OVER nie wywołuje kolejnych ticków
`game_tick()` sprawdza się na samym początku `if state.status is not RUNNING: return` — `GAME_OVER` dociera tutaj w taki sam sposób jak `PAUSED` . Jedyny sposób, by znów się ruszyć, to `restart()` (rozdział 19.23), który wyraźnie tworzy nowy stan.

`GAME_OVER` — to nie jest jedyny status terminalny: rozdział 19.18 pokazuje drugi, znacznie rzadszy przypadek — `GameStatus.WON` , gdy wąż zajmuje dosłownie całe pole. Oba zatrzymują tiki w ten sam sposób, różni się tylko powód i tekst nakładki.

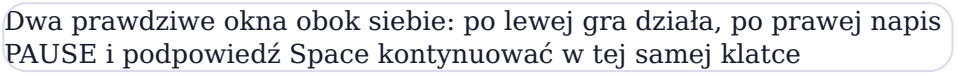
Praktyka: przejście do GAME_OVER
Automatyczne sprawdzanie – funkcja, resolve_tick(): której kolizje zamieniają grę w GAME_OVER

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/19-21/index.html)

Pause / Resume

Pauza zatrzymuje model w miejscu – konsumuje tę samą grę, nie rozpoczyna nowej.

Pauza zatrzymuje tykanie, a nie rysuje na wierzchu gry biegowej



Prawdziwe okno: toggle_pause() nie przesuwa węża dalej — ramka po prawej stronie jest taka sama jak po lewej, tylko z nakładką.

Klawisz `Space` przełącza grę między `RUNNING` oraz `PAUSED`:

toggle_pause.py

```
def toggle_pause(self):
    if self.state.status is GameStatus.RUNNING:
        self._generation += 1   # natychmiast unieruchamia już
zaplanowany tik
        self.state.status = GameStatus.PAUSED
        self._show_overlay(„PAUZA”, „Space kontynuować”)
    elif self.state.status is GameStatus.PAUSED:
        self._generation += 1   # Nowe pokolenie — dokładnie
jeden świeży łańcuch odkleń
        self.state.status = GameStatus.RUNNING
        self._clear_overlay()
        self._schedule_next_tick()
```

Zatrzymanie nie odtwarza węża

`toggle_pause()` nie dotyka `state.snake`, `state.score` lub `state.food` — tylko zmiany `status`. Restart kontynuuje tę samą grę z tego samego punktu zamiast zaczynać nową.

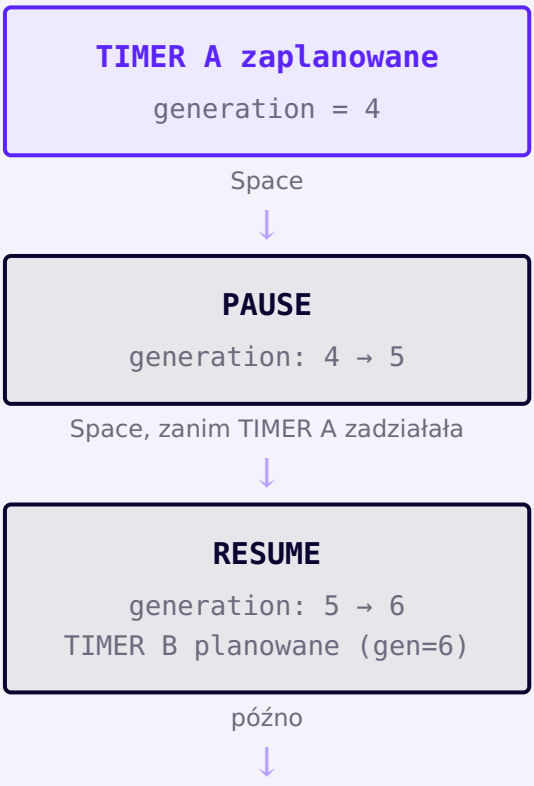
Dlaczego jedna kontrola status nie wystarcza

`screen.ontimer(callback, delay_ms)` tylko PLANUJE przyszłe wywołanie — sam fakt, że `status` zmienione na `PAUSED` nie usuwa już zaplanowanych `callback` z kolejki Tkinter: wciąż zostanie wywołany, po prostu nic pożytecznego nie zrobi, jeśli wewnątrz sprawdzić status. Prawie działa — ale nie do końca, a różnica ujawnia się tam, gdzie jest najtrudniej ją zauważyć: co jeśli wznowienie (`Resume`) stanie się PRZED wywołaniem połączenia Tik zaplanowany przed przerwą?

Prawdziwy błąd: Pause → Resume zanim stary tik się uruchomił

Bez niepełnosprawności pokolenia okazuje się, że to właśnie w momencie przerwy okazuje się: stary kleszcz nadal jest zaplanowany → Resume planuje SEKUNDĘ, jego własny tik → oba kiedyś zadziałają, każdy planuje kolejny dla siebie, → DWA niezależne łańcuchy tykają w grze jednocześnie, a wąż zaczyna poruszać się dwa razy częściej, niż powinien. Weryfikacja `status is RUNNING` w środku tyka nie łapie tego wyścigu: zanim stary tik w końcu działa, gra jest już RUNNING – test przechodzi, a tik planuje kontynuować, jakby nic się nie stało.

Dlatego `toggle_pause()` zwiększa `self._generation` przy KAŻDEJ zmianie — i przy ustawianiu na Pauza, a przy wznowieniu, nie tylko podczas restartu. Pauza jest paraliżująca zaplanowane tknięcie natychmiast, w momencie naciskania `Space`, nie Polega na tym, że rozpozna siebie, gdy w końcu zacznie pracować. Odnowienie otrzymuje posiadanie nowej generacji i planowanie dokładnie jednego nowego odszycia – niezależnie od sieci istniał wcześniej, już nie żyje.



Pauza i wznowienie rozpoczynają nowe pokolenie od razu, w momencie naciśnięcia Space — przeterminowane TIMER A dowiaduje się o tym przy pierwszym sprawdzeniu i nie planuje niczego więcej.

Praktyka: pause nie przesuwa tego stanu
Automatyczne sprawdzanie – game_tick(): funkcja tykania podczas PAUSED nie zmienia węża

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/19-22/index.html)

Restart / New Game

Nowa gra musi przerwać stary łańcuch ticków, inaczej przez chwilę będziesz mieć dwa niezależne łańcuchy ticków jednocześnie.

Restart musi przerwać stary łańcuch kleszczy

Prawdziwe okno: świeża gra z jedną głową na środku, tablica wyników
Wynik: 0 Rekord: 40

Prawdziwym oknem po restart(): świeżym wężem, wynik zostaje zerowy, rekord pozostaje.

Restart nie tylko przywraca węża na jego pierwotne miejsce — sekcja 19.22 ostrzega: jeśli stary łańcuch `ontimer()` wciąż tyka, gra trwa Za chwilę pojawiłyby się dwie niezależne serie aktualizacji jednocześnie (rozdział 16 już Analizowałem podobny problem z ponownym naciśnięciem przycisku).

restart.py

```
def restart(self):
    self._generation += 1   # przerywa wszelki wciąż tykający
łańcuch poprzedniej gry
    high_score = self.state.high_score
    self.state = new_game_state(self.rng,
high_score=high_score)
    self._clear_overlay()
    self.render()
```

generation_guard.py

```
def _schedule_next_tick(self):
    generation = self._generation
    self.screen.ontimer(lambda: self._on_timer(generation),
self.state.delay_ms)

def _on_timer(self, generation):
    if generation != self._generation:
        return   # wygasły łańcuch od gry do restart() — sam
się gasnie
    self.game_tick()
    if self.state.status is GameStatus.RUNNING:
        self._schedule_next_tick()
```

Generowanie (generation) — prosty licznik, nie skomplikowana architektura

Każdy `restart()` zwiększa `_generation` na jednostkę. Jakieś już zaplanowane `_on_timer()` zapamiętał znaczenie OLD – gdy w końcu działa, liczba się nie zgadza i cicho nic nie robi, zamiast przesuwać już nieistniejącego węża z poprzedniej gry.

`high_score` — jedyne pole, które restart przenosi ze starego stanu do nowego; wszystko pozostałe tworzone jest na nowo przez `new_game_state()` (Rozdział 19.25).

Praktyka: restart i przeterminowane kleszcze
Automatyczna Weryfikacja – Funkcje new_game_state() i restart(): rekord się restartuje, generacja rośnie

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/19-23/index.html)

Prędkość i wzrost złożoności

Każde jabłko nie tylko zwiększa wynik, ale też przelicza opóźnienie — gra stopniowo przyspiesza.

Gra przyspiesza wraz z wynikiem

Prawdziwe okno: wąż z pięcioma segmentami, wynik 450, jabłko na szczycie pola

Realne okno: przy wysokim wyniku calculate_delay() już zwrócił minimalną wartość – gra gra działa z maksymalną prędkością.

Rozdział 19.14 już pokazała formułę – tutaj łączy się ona z grą: po każdym Zjedzonego jabłka nie tylko przeliczany jest wynik, ale także opóźnienie kolejnego ticku:

speed_progression.py

```
if grow:
    state.score += FOOD_SCORE
    state.high_score = max(state.high_score, state.score)
    state.food = choose_food(state.snake, self.rng)
    state.delay_ms = calculate_delay(state.score)
```

Nowa wartość `delay_ms` wejdzie w życie przy następnym wezwaniu `_schedule_next_tick()` — poprzedni tick został już zaplanowany z wcześniejszym opóźnieniem i to jest w porządku: prędkość zmienia się płynnie, bez szarpnięć.

min_ms jest świadomym ograniczeniem, a nie przypadkiem
Rozdział 19.14 już została wyjaśniona `max(min_ms, ...)` — tutaj widać, dlaczego jest to potrzebne w praktyce: bez niego wystarczająco długa gra prędzej czy później doprowadzi opóźnienie do zera.

Praktyka: Szybkość rośnie wraz z liczeniem
Automatyczne sprawdzanie – funkcja eat_food(): każdym jabłkiem przelicza zarówno liczbę, jak i opóźnienie

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/19-24/index.html)

High Score

score i high_score to różne pola o różnych czasach życia: jedno jest resetowane po restarcie, drugie go przeżywa.

Rekord przetrwa restart — licznik nie

Realne okno: jedna głowa na środku boiska, tablica wyników powyżej pokazuje Wynik: 0 Rekord: 40

Rzeczywiste okno: po restart() wynik został zresetowany, a rekord pozostał taki sam.

`score` oraz `high_score` — dwa różne pola o różnym czasie życia. Rozdział 19.23 już pokazał, że `restart()` wyraźnie przekazuje `high_score` w Nowe państwo, zamiast tworzyć je od nowa:

high_score.py

```
def new_game_state(rng, *, high_score=0):
    snake = [(0, 0)]
    return GameState(
        snake=snake,
        # ...
        score=0,                # zawsze od zera
        high_score=high_score, # przekazane przez wywołujący
kod
        # ...
    )
```

Pole	Co się dzieje przy restart()
score	zawsze resetuje się do 0 — nowa gra zaczyna się od czystego wyniku

Pole	Co się dzieje przy restart()
high_score	przekazywana z przeszłości state.high_score doświadcza restartu

high_score jest aktualizowany w max() momencie, a nie na końcu gry
`state.high_score = max(state.high_score, state.score)` wywoływana od razu przy każdym zjedzonym jabłku (rozdział 19.24) — rekord nie wymaga „podsumowania" osobno na końcu: jest już dokładny w każdej chwili gry, w tym na ekranie Game Over.

Praktyka: album zaczyna się wznowiać
Automatyczne sprawdzanie — funkcja update_high_score(): rekord tylko rośnie

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/19-25/index.html)

Czysta logika gry bez Turtle

Zasady gry to zwykłe funkcje liczb i list; Turtle wystarczy je pokazać.

Żadna z funkcji reguł nie otwiera okna

Przyjrzyjmy się wszystkim omówionym funkcjom od sekcji 19.9: `next_head()`, `move_snake()`, `is_wall_collision()`, `is_self_collision()`, `choose_food()`, `calculate_delay()`. Żaden z nich nie tworzy `turtle.Turtle()`, nie czyta `golova.xcor()` w ogóle nie wie nic o ekranie:

Zasady gry – Wspólne funkcje liczb i list

NEXT_HEAD(HEAD, DIRECTION)

krotka → krotka

sekcja 19.11

MOVE_SNAKE(SNAKE, HEAD, GROW)

`lista → lista`

Rozdział 19.18

IS_WALL_COLLISION(HEAD)

`motorcade → bool`

Rozdział 19.19

IS_SELF_COLLISION(HEAD, BODY)

`konwoj lista → bool`

Rozdział 19.20

CHOOSE_FOOD(SNAKE, RNG)

`lista → kondukt`

Rozdział 19.18

CALCULATE_DELAY(SCORE)

`int → int`

Rozdział 19.14

To nie jest przypadek, lecz świadome rozdzielenie: zasady „Węża” *program*. Rozdział 19.30 wykorzystuje dokładnie tę właściwość, aby testować reguły bezpośrednio, bez żadnej Prawdziwe okno.

Ta sama zasada co w rozdziale 18

`normalize_bounds()` oraz `Shape` w PaintApp (rozdział 18) były dokładnie takie same — czyste funkcje i dane bez `tkinter` wewnątrz. Są z tego dwa korzyści: logikę łatwiej testować i łatwiej ją zrozumieć — czytając funkcję nie trzeba trzymać w głowie stanu całego okna.

Praktyka: Które funkcje są czystą logiką

Automatyczna walidacja – funkcja is_pure_logic(): rozróżnianie reguł bez Turtle od kodu wymagającego ekranu

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/19-26/index.html)

GameState z dataclass

Jedna struktura danych opisuje całą grę, bez żadnego obiektu Turtle środku.

Jedna struktura zamiast ośmiu oddzielnych zmiennych

Zebrać wszystkie pola podzielone na części — węża, kierunek, jedzenie, wynik, status, Prędkość to jeden. `@dataclass`, ten sam pomysł co `DrawingState` w rozdziale 18:

Żadnych obiektów Turtle — tylko dane w pełni opisujące aktualną grę.

gamestate.py

```
@dataclass
class GameState:
    snake: list[Position] = field(default_factory=lambda: [(0,
0)])
    direction: Direction = Direction.RIGHT
    next_direction: Direction = Direction.RIGHT
    food: Position | None = (0, 0)
    score: int = 0
    high_score: int = 0
    status: GameStatus = GameStatus.READY
    delay_ms: int = BASE_DELAY_MS
```

Dlaczego food w ogóle można None

Dziewięćdziesiąt dziewięć meczów na sto `food` jest zwykłą klatką. Ale sekcja 19.18 pokazuje szczery, skrajny przypadek: jeśli wąż zajmuje dosłownie całe pole, nie będzie wolnej klatki na nowe jedzenie. Rodzaj Model `Position | None` sprawia, że ten przypadek jest widoczny w samej sygnaturze, a nie ukryty — kod, który czyta `state.food` jest zobowiązany jasno zdecydować, co zrobić, jeśli nie ma jedzenia.

Pokusa, aby zapisać tutaj obiekt Turtle- — częsty błąd

Jeśli wpiszesz `head_turtle` w środku `GameState`, testy z rozdziału 19.30 przestaną działać bez prawdziwego okna, a „model” przestanie być modelem — znowu połączy się z wyświetlaniem, jak w pierwszym prototypie. Rozdział 19.31 omawia ten błąd jako osobny Debug Lab.

`next_direction` — oddzielny od `direction` pole z jakiegoś powodu: Rozdział 19.31 (Debug Labs 5-6) wyjaśnia, dlaczego klawiatura żąda kierunek i nie zmienia go bezpośrednio.

Praktyka: projektujemy i porównujemy GameState

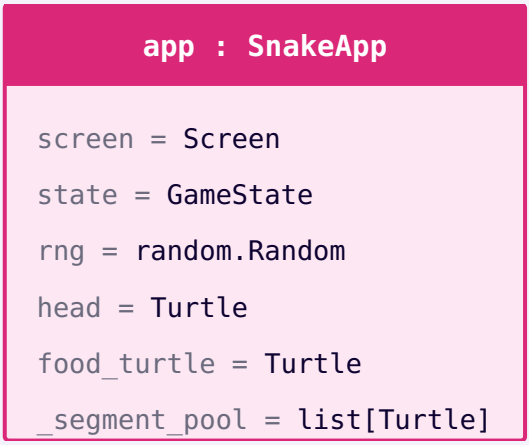
Automatyczna weryfikacja — klasa GameState: tworzenie, porównanie i dataclasses.replace()

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/19-27/index.html)

SnakeApp: oddzielenie modelu od wizualizacji

app.state jest model gry; app.screen i Turtle-obiekty to tylko to, co jest widoczne na ekranie.

app HAS-A screen jest taki sam jak w rozdziałach 16–18



app przechowuje zarówno widżety Turtle, jak i model (state) — ale ich nie myli ze sobą.

Grupa metod	Za co odpowiada
`bind_keys`	budować obsługi klawiatury — raz, przy starcie
`request_direction`	klucz → poproś o kierunek, teraz brak ruchu
`game_tick`	jeden odznak → zmianie state
`render`	state → Turtle (jedyne miejsce, które naprawdę rysuje)
`toggle_pause / restart`	Zarządzanie cyklem życia gry

SnakeApp nie dziedziczy po turtle.Screen
Na przykład `PaintApp(tk.Tk)` w rozdziale 18 dziedziczenie tutaj technicznie jest możliwe, ale nie wybierane: `self.screen` jest atrybutem indywidualnym, a nie samym obiektem `SnakeApp`. Kompozycja sprawia, że granica między „moją logiką" a „wnętrzem Turtle" jest oczywista.

Praktyka: Złóż graf obiektowy SnakeApp
Moduł turtle otwiera natywne okno Python – uruchamiane lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/19-28/index.html)

Render: model → Turtle

Ta sama pula segmentów Turtle-segment jest wykorzystywana ponownie przy każdym ticku – nowe powstają dopiero wtedy, gdy wąż urośnie.

render() nie tworzy nowych segmentów przy każdym ticku

Prawdziwe okno: cienka fioletowa siatka na boisku, oznaczona head cell i food cell

Prawdziwe okno: ta sama siatka z sekcji 19.9, ale teraz widać głowę i jedzenie — po prostu komórki, które render() przekształca w konkretne obiekty Turtle-.

Wąż rośnie i pojawia się proste rozwiązanie: stworzyć nowego `turtle.Turtle()` za każde zjedzone jabłko i dokładnie to robił pierwszy prototyp (Rozdział 19.6). Ale obiekty Turtle nie znikają: po `restart()` będzie ich tyle samo, co było w najdłuższym czasie Partii, a każda z nich będzie nadal obsługiwać ekran. Dlatego wchodzimy **basen Segmenty**: po stworzeniu żółwie są wykorzystywane ponownie z klatki do klatki, nowe Są dodawane tylko wtedy, gdy naprawdę jest ich za mało, a te dodatkowe są ukryte, a nie usuwane.

render.py

```
def render(self):
    self.head.goto(*self.state.snake[0])
```

```
    body = self.state.snake[1:]
    self._ensure_segment_pool(len(body))
    for i, segment in enumerate(self._segment_pool):
        if i < len(body):
            segment.showturtle()
            segment.goto(*body[i])
        else:
            segment.hideturtle()

    if self.state.food is not None:
        self.food_turtle.showturtle()
        self.food_turtle.goto(*self.state.food)
    else:
        self.food_turtle.hideturtle()   # WON — brak wolnych
komórek

    self._render_scoreboard()
    self.screen.update()
```

Sprawdzenie food is not None jest tutaj obowiązkowe

Rozdział 19.18 obiecywał, że rysowanie uczciwie obsłuży sytuację, gdy nie ma już wolnego pola na pożywienie — tutaj. Bez sprawdzenia `self.food_turtle.goto(*self.state.food)` at `food = None` spada z `TypeError: turtle.TNavigator.goto() argument after * must be an iterable, not NoneType` – dokładnie w momencie zwycięstwa, w najrzadszym i najbardziej ofensywnym stanie gry.

Pul rośnie, ale nigdy nie zmniejsza się sam

`_ensure_segment_pool()` dodaje nowe segmenty Turtle-, tylko jeśli jest ich za mało, istniejące są ponownie wykorzystywane. Gdy wąż staje się krótszy (po `restart()`), dodatkowe segmenty nie są usuwane, lecz po prostu ukryte przez `hideturtle()` — w następnej długiej grze będą znowu potrzebne i nie trzeba ich będzie tworzyć od nowa.

Praktyka: render() i pula odłamków

Moduł turtle otwiera natywne okno Python – uruchamiane lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/19-29/index.html)

Testowanie zasad gry

Ponieważ reguły są czystymi funkcjami, można je sprawdzać assert normalnie, bez okna i bez Xvfb.

Sprawdzamy zasady bezpośrednio, bez żadnego okna

Rozdział 19.26 pokazała, że zasady gry są czysto funkcjonalne. Można je więc sprawdzić zwyczajny `assert` jak w rozdziale 3 — bez Xvfb, bez `turtle.Screen()`, nie czekając na wyciągnięcie czegoś:

test_snake_logic.py

```
def test_move_snake_without_growth_drops_tail():
    snake = [(0, 0), (-20, 0), (-40, 0)]
    result = move_snake(snake, (20, 0), grow=False)
    assert result == [(20, 0), (0, 0), (-20, 0)]

def test_wall_collision_boundary_is_safe():
    assert is_wall_collision((280, 280)) is False

def test_wall_collision_beyond_boundary():
    assert is_wall_collision((300, 0)) is True
```

Przypadki borderline to indywidualne testy, a nie jeden ogólny
„Bezpiecznie dokładnie na granicy” i „zderzenie nieco za granicą” — to dwa różne testy, nie jeden: jeśli połączyć je w jedno sprawdzenie, test może pozostać zielony nawet przy błędzie dokładnie w tym punkcie (rozdział 19.19 wyjaśniał, dlaczego granica jest uwzględniona).

Testowanie scenariuszy, dla których ważny jest przypadek (wybór jedzenia), pomaga `random.Random(seed)` z ustalonym ziarnem – sekcja 19.18 już jest wyjaśniłem dlaczego `choose_food()` akceptuje gotowy generator zamiast Czyta Global `random` bezpośrednio:

test_food.py

```
def test_choose_food_never_lands_on_snake():
    rng = random.Random(1)
    snake = [(0, 0), (-20, 0), (-40, 0)]
    for _ in range(50):
        food = choose_food(snake, rng)
        assert food not in snake
```

Praktyka: Pisanie testu zasad gry
Automatyczny test – funkcja test_boundary_is_safe(): test musi wykryć uszkodzoną wersję

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/19-30/index.html)

Debug Labs są typowe błędy „Węża”

Każdy błąd tutaj występuje w prawdziwych projektach studenckich — naucz się rozpoznawać objaw, zanim otworzysz debugator.

Mała kolekcja typowych błędów „Wężyka” — każdy z objawem i poprawką. Część z nich widzieliście już wcześniej w rozdziale; tutaj są zebrane jako podręcznik.

DEBUG LAB 1: ZAPOMNIANO SCREEN.TRACER(0)

bez_tracer.py

```
screen = turtle.Screen()
screen.title(„Wąż”)
# tracer(0) nie wywołano
```

```
# Игра работает, но каждое движение головы и КАЖДОГО сегмен
# перерисовывается отдельно — заметное мерцание и заметная
```

Co widać na ekranie

Bez `tracer(0)` Turtle odświeża ekran po każdym pojedynczym połączeniu `goto()` — sekcja 19.2 już wyjaśniła, dlaczego należy wyłączyć automatyczne aktualizacje i robić to ręcznie, dokładnie raz na tick.

POPRAWIONY KOD

s_tracer.py

```
screen = turtle.Screen()
screen.title(„Wąż”)
screen.tracer(0)
```

DEBUG LAB 2: ZAPOMNIAŁEM SCREEN.UPDATE() NA KOŃCU RENDER()

bez_update.py

```
def render(self):
    self.head.goto(*self.state.snake[0])
    # ...pozostałe renderowanie...
    # self.screen.update() nie wywołany
```

```
# Модель меняется на каждом тике (можно проверить print()),
# но на экране змейка стоит неподвижно.
```

Co widać na ekranie

P `tracer(0)` instrukcja `screen.update()` jest jedynym momentem, w którym zmiany docierają na ekran. Ta sama zasada co w rozdziale 17: zmiana modelu bez wywołania renderowania jest niewidoczna dla użytkownika.

POPRAWIONY KOD

s_update.py

```
def render(self):
    self.head.goto(*self.state.snake[0])
    # ...pozostałe renderowanie...
    self.screen.update()
```

DEBUG LAB 3: PĘTLA BLOKUJĄCA UTRUDNIA PAUZĘ I PRĘDKOŚĆ

busy_loop.py

```
while igrovoj_shag():
    screen.update()
    time.sleep(ZADERZHKA_SEK)   # stała latencja
# pętla decyduje, kiedy dokładnie sprawdzić zdarzenia — pauzę, prędkość
# i restart musi być ręcznie wpleciony w jego ciało
```

```
# Игру нельзя поставить на паузу: событие обрабатывается,
# но цикл ничего не проверяет и продолжает крутиться дальше
# Скорость задана одной константой и по ходу партии не меня
```

Co widać na ekranie

Rozdział 19.13 szczegółowo poruszała ten temat: `screen.update()` w środku `while` OBSŁUGUJE Tkinter zdarzenia, w tym naciśnięcia klawiszy — technicznie rzecz biorąc, obsługa pauzy mogłaby działać. Ale sama pętla nie pyta, czy musi się zatrzymać: pauza, prędkość i restart musiałyby być ręcznie sprawdzane przy każdej iteracji. `screen.ontimer()` jest inna: każdy tick jest osobnym zaplanowanym wywołaniem, a nie iteracją jednej nieskończonej pętli, a decyzja o „wykonaniu następnego ticku lub nie”

POPRAWIONY KOD

ontimer_vmesto_busy.py

```
def game_tick(self):
    if self.state.status is not GameStatus.RUNNING:
        return
    # ...
    self.screen.ontimer(self.game_tick,
self.state.delay_ms)
```

DEBUG LAB 4: TIME.SLEEP() W CYKLU GRY

s_sleep.py

```
def game_tick(self):
    # ...
    time.sleep(self.state.delay_ms / 1000)
    self.game_tick()
```

```
# Окно замирает НАВСЕГДА, а не на delay_ms: тик зовёт сам себ
# без условия выхода, поэтому пауз становится бесконечно мног
# а примерно через 1000 вызовов программа падает с RecursionE
```

Co widać na ekranie

Błędy są tu dwa. Pierwszy: `time.sleep()` blokuje CAŁY proces, włącznie z cyklem zdarzeń Tkinter, na którym opiera się Turtle.. `game_tick()` wywołuje się bezpośrednio, bez żadnego warunku wyjścia, to po prostu nieskończona rekurencja, a stos się wyczerpie, zanim gracz zdąży uzyskać choćby jedną odpowiedź. `screen.ontimer()` rozwiązuje oba: nic nie blokuje i nie zwiększa stosu — rozdział 19.13 wyjaśnia różnicę szczegółowo.

POPRAWIONY KOD

bez_sleep.py

```
def game_tick(self):
    # ...
    self.screen.ontimer(self.game_tick,
self.state.delay_ms)
```

DEBUG LAB 5: DOZWOLONY OBRÓT O 180°

bez_is_reverse.py

```
def request_direction(self, direction):
    self.state.next_direction = direction   # bez
weryfikacji!
```

```
# Змейка едет вправо, игрок нажимает Left —
# голова немедленно врезается в первый сегмент собственного
```

Co widać na ekranie

Bez `is_reverse()` nic nie stoi na przeszkodzie, by zamienić węża w jego ciało jednym kliknięciem — sekcje 19.4 (i 19.11) wyjaśniały, dlaczego obrót o 180° powinien być zabroniony na poziomie wejścia, a nie tylko przypadkowo.

POPRAWIONY KOD

s_is_reverse.py

```
def request_direction(self, direction):
    if is_reverse(self.state.direction, direction):
        return
    self.state.next_direction = direction
```

DEBUG LAB 6: SZYBKA PARA KLUCZY WYKONUJE ZAWRACANIE OBOK CZEKU

proverka_ne_protiv_togo.py

```
def request_direction(self, direction):
    if is_reverse(self.state.next_direction,
direction):   # kontra next_direction!
        return
    self.state.next_direction = direction
```

```
# Змейка едет вправо. Игрок быстро нажимает Up, потом Left —
# Left проходит проверку, потому что next_direction уже стало
```

Co widać na ekranie

Pułapka jest cienka: weryfikacja musi iść przeciwko `state.direction` jest kierunek, który TERAZ jest stosowany na kleszczu – a nie przeciwko temu, który już został zmieniony `next_direction`, w przeciwnym razie drugi klawisz między tykaczami może przeciągnąć nielegalny zwrot w stronę (U-turn).

POPRAWIONY KOD

proverka_protiv_direction.py

```
def request_direction(self, direction):
    if is_reverse(self.state.direction, direction):
        return
    self.state.next_direction = direction
```

DEBUG LAB 7: JEDZENIE NIE JEST WYRÓWNANE W SIECI

neverno_vyrovnennaya_eda.py

```
def choose_food(snake, rng, *, half=280):
    x = rng.randint(-half, half)   # dowolna liczba
całkowita, a nie wielokrotność STEP!
    y = rng.randint(-half, half)
    return (x, y)
```

```
# Яблоко появляется в точке вроде (137, -52) —
# голова змейки никогда не попадёт туда ровно, съесть его н
```

Co widać na ekranie

Rozdział 19.9 wyjaśniona: pozycje prawne to węzły siatki z nachyleniem `STEP`. `choose_food()` musi wybrać z tego samego zestawu komórek co `next_head()`, w przeciwnym razie matematycznie niemożliwe jest dokładne wejście do klatki z jedzeniem.

POPRAWIONY KOD

vyrovnennaya_eda.py

```
def choose_food(snake, rng, *, half=280, step=20):
    free = tuple(c for c in all_cells(half=half,
step=step) if c not in set(snake))
    return rng.choice(free)
```

DEBUG LAB 8: JEDZENIE POJAWIA SIĘ WEWNĄTRZ CIAŁA WĘŻA

eda_bez_proverki.py

```
def choose_food(snake, rng, *, half=280, step=20):
    coords = range(-half, half + 1, step)
    return (rng.choice(coords), rng.choice(coords))   #
nie sprawdza, snake!
```

```
# При достаточно длинной змейке яблоко иногда появляется
# прямо внутри собственного тела — заведомо несъедобное.
```

Co widać na ekranie

Rozdział 19.18 omawiał dokładnie ten problem: bez wyjątku zajętych komórek `choose_food()` wybiera uczciwie losowo spośród WSZYSTKICH pól, w tym zajętych ciałem — co technicznie jest losowe, ale nieuczciwe wobec gracza.

POPRAWIONY KOD

eda_s_proverkoj.py

```
def choose_food(snake, rng, *, half=280, step=20):
    occupied = set(snake)
    free = tuple(c for c in all_cells(half=half,
step=step) if c not in occupied)
    return rng.choice(free)
```

DEBUG LAB 9: CIAŁO JEST ODNAWIANE OD GŁOWY PO OGON

ot_golovy_k_hvostu.py

```
def dvigat_telo():
    for indeks in range(len(segmenty) - 1):   #wrong
kierunek jazdy na rowerze!
        segmenty[indeks].goto(segmenty[indeks +
1].xcor(), segmenty[indeks + 1].ycor())
```

```
# Через несколько шагов все сегменты тела
# визуально схлопываются в одну точку.
```

Co widać na ekranie

Rozdział 19.17 szczegółowo omawiała ten temat: aktualizacja musi przejść od ogona do główka, w przeciwnym razie każdy kolejny segment odczytuje już nadpisane, a nie starą pozycję sąsiada.

POPRAWIONY KOD

s_hvosta_k_golove.py

```
def dvigat_telo():
    for indeks in range(len(segmenty) - 1, 0, -1):
        segmenty[indeks].goto(segmenty[indeks -
1].xcor(), segmenty[indeks - 1].ycor())
```

DEBUG LAB 10: TABLICA WYNIKÓW BEZ CLEAR()

tablo_bez_clear.py

```
def obnovit_tablo():
    tablo.write(f"Wynik: {{schet}}", align="center",
font=("Arial", 16, "normal"))
    #tablo.clear() nie wywołane
```

```
# После нескольких съеденных яблок надпись на табло
# превращается в нечитаемую кашу из наложенных друг на друга
```

Co widać na ekranie

Rozdział 19.5 wyjaśniona: `write()` nie zastępuje poprzedniego tekstu, lecz rysuje na nim. `clear()` jest obowiązkowe przed każdym nowym wpisem tego samego żółwia.

POPRAWIONY KOD

tablo_s_clear.py

```
def obnovit_tablo():
    tablo.clear()
    tablo.write(f"Wynik: {{schet}}", align="center",
font=("Arial", 16, "normal"))
```

DEBUG LAB 11: GRANICA NIE JEST ŚCIŚLE KONTROLOWANA – GŁOWA NIE SIĘGA KRAWĘDZI

granica_s_ravno.py

```python
def is_wall_collision(head, *, half=280):
    x, y = head
    return abs(x) >= half or abs(y) >= half   #>=, nie >
```

```
# Игра заканчивается на одну клетку раньше настоящей границы
# змейка никогда не может доехать до последнего легального
```

Co widać na ekranie

Rozdział 19.19 wyjaśniona: `GRANICA` to współrzędna środka SEGMENTU, który nadal mieści się całkowicie na polu. `>=` błędnie wyklucza samą granicę z legalnego obszaru.

POPRAWIONY KOD

granica_strogo.py

```python
def is_wall_collision(head, *, half=280):
    x, y = head
    return abs(x) > half or abs(y) > half
```

DEBUG LAB 12: SAMOZDERZENIE SPRAWDZONE DLA STAREGO CIAŁA

stolknovenie_po_staromu.py

```
grow = head == state.food
if is_self_collision(head, state.snake[1:]):   # PRZED
move_snake()!
    state.status = GameStatus.GAME_OVER
```

```
# Змейка не растёт, а игрок пытается заехать в клетку, котору
# хвост как раз освобождает в этом же тике — игра ошибочно за
```

Co widać na ekranie

Rozdział 19.20 dotyczyła właśnie tego: stara klatka ogonowa nadal jest w środku `state.snake` TO `move_snake()`. Musisz sprawdzić ciało PO przeprowadzce — `new_snake[1:]` — gdzie ogon już został uczciwie odcięty, jeśli wąż nie rośnie.

POPRAWIONY KOD

stolknovenie_po_novomu.py

```
new_snake = move_snake(state.snake, head, grow=grow)
if is_self_collision(head, new_snake[1:]):
    state.status = GameStatus.GAME_OVER
```

DEBUG LAB 13:RESTART ZOSTAWIA STARY ŁAŃCUCH KLESZCZOWY PRZY ŻYCIU

restart_bez_generation.py

```
def restart(self):
    self.state = new_game_state(self.rng)
    self.render()
    #generation nie powiększone — stary _on_timer()
wciąż jest planowany!
```

```
# Через мгновение после Restart на экране внезапно появляется
# фигура/движение от уже несуществующей прошлой игры.
```

Co widać na ekranie

Rozdział 19.23 zajmowała się tą sprawą, klasycznym błędem „dwóch równoległych łańcuchów timera” `_on_timer()` nie różni się od nowego – oba działają jednocześnie.

POPRAWIONY KOD

restart_s_generation.py

```
def restart(self):
    self._generation += 1
    self.state = new_game_state(self.rng,
high_score=self.state.high_score)
    self.render()
```

DEBUG LAB 14: ZATRZYMAJ EKRAN, ALE GRA NADAL SIĘ PORUSZA

pauza_tolko_vizualno.py

```
def toggle_pause(self):
    self._show_overlay(„PAUZA”, „Space kontynuować”)
    # state.status niezmienione — game_tick() nic nie
wie o pauzie!
```

```
# Оверлей «ПАУЗА» показан, но змейка под ним
# продолжает двигаться и может врезаться в стену.
```

Co widać na ekranie

Rozdział 19.22 wyjaśniona: nakładka to tylko coś, co jest WIDOCZNE. Prawdziwy stop wynika z czeku `status is not RUNNING` na samym początku `game_tick()` — bez zmiany `state.status` ten czek nigdy nie zadziała.

POPRAWIONY KOD

pauza_po_statusu.py

```
def toggle_pause(self):
    if self.state.status is GameStatus.RUNNING:
        self.state.status = GameStatus.PAUSED
        self._show_overlay(„PAUZA”, „Space kontynuować”)
```

DEBUG LAB 15: NOWA GRA NIE RESETUJE KIERUNKU

restart_bez_napravleniya.py

```
def restart(self):
    self._generation += 1
    self.state.snake = [(0, 0)]   # poprawia pole
punktowo, a nie tworzy state od nowa
    self.state.score = 0
    # direction/next_direction pozostały z poprzedniej
gry!
```

```
# Новая змейка стоит в центре, но при первом же движении
# уезжает в направлении, в котором закончилась ПРОШЛАЯ игра
```

Co widać na ekranie

Spot edytuje do istniejącego `state` łatwo zapomnieć o jednym z pól. `new_game_state()` (rozdział 19.25) tworzy całą strukturę od nowa — zapomnieć o oddzielnym polu w nowym obiekcie jest niemożliwe, albo jest w konstruktorze, albo kod się nie uruchomi.

POPRAWIONY KOD

restart_novym_state.py

```
def restart(self):
    self._generation += 1
    self.state = new_game_state(self.rng,
high_score=self.state.high_score)
```

DEBUG LAB: REKORD ZOSTAJE ZRESETOWANY PRZEZ WYNIK

restart_teryaet_rekord.py

```
def restart(self):
    self._generation += 1
    self.state = new_game_state(self.rng)
# high_score nie przekazano — znowu 0!
```

```
# Игрок набрал 90 очков, проиграл, нажал R —
# табло снова показывает «Рекорд: 0», как будто игра только ч
```

Co widać na ekranie

Rozdział 19.25 wyjaśniał: `high_score` musi być wyraźnie przekazany ze starego `state` w nową. `new_game_state()` bez argumentu `high_score` używa wartości domyślnej — zero.

POPRAWIONY KOD

restart_s_rekordom.py

```
def restart(self):
    self._generation += 1
    self.state = new_game_state(self.rng,
high_score=self.state.high_score)
```

DEBUG LAB 17: GAMESTATE PRZECHOWUJE OBIEKT TURTLE

gamestate_s_turtle.py

```
@dataclass
class GameState:
    snake: list[Position]
    head_turtle: turtle.Turtle   # obiekt Turtle
wewnątrz modelu!
```

```
# Тесты из раздела 19.30 падают с ошибкой создания Turtle,
# хотя проверяют только математику координат — окно им не н
```

Co widać na ekranie

Rozdział 19.27 wyraźnie ostrzegała: `GameState` jest domeną danych, a nie kontenerem na widgety. Gdy tylko `turtle.Turtle` staje się niemożliwe stworzenie stanu bez prawdziwego okna — cała użyteczność czystej logiki (Rozdział 19.26) znika.

POPRAWIONY KOD

gamestate_bez_turtle.py

```
@dataclass
class GameState:
    snake: list[Position]
    # obiekty Turtle żyć w SnakeApp, a nie w GameState
```

DEBUG LAB 18: TICK PLANUJE SIEBIE, NAWET GDY GRA JUŻ NIE JEST RUNNING

tik_planiruet_vsegda.py

```
def _on_timer(self, generation):
    if generation != self._generation:
        return
    self.game_tick()
    self._schedule_next_tick()   # brak sprawdzania
statusu!
```

```
# После Game Over или паузы змейка выглядит остановленной,
# но цепочка ontimer() продолжает тикать вхолостую в фоне.
```

Co widać na ekranie

`game_tick()` sam w sobie jest bezpieczny — po prostu nic nie robi, gdy `status != RUNNING`. Ale jeśli następny tik jest ZDECYDOWANIE zaplanowany, łańcuch nigdy nie zatrzyma się sam, marnując timer nawet po zakończeniu gry.

POPRAWIONY KOD

tik_planiruet_esli_running.py

```
def _on_timer(self, generation):
    if generation != self._generation:
        return
    self.game_tick()
    if self.state.status is GameStatus.RUNNING:
        self._schedule_next_tick()
```

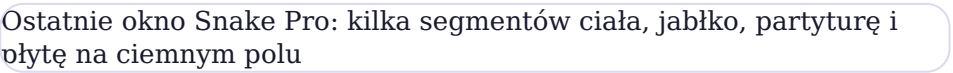
Praktyka: znalezienie błędu „Snake”
Automatyczna kontrola — funkcja diagnose(): pozwala według objawu ustalić przyczynę, również nieznany objaw jest obsługiwany

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/19-31/index.html)

Snake Pro - Finałowa gra

Ten sam prototyp z sekcji 19.1 — ale teraz z modelem stanu, tickiem gry i architekturą, która wytrzyma rozwój projektu.

Program końcowy



Właściwe okno w ostatecznej wersji to ten sam pomysł, który otwierał rozdział w sekcji 19.1, ale teraz wiemy, z czego jest zrobiony.

Plik jest kompletny, samodzielny i nie posiada niewidzialnych zależności od innych lekcji:

projects/turtle/snake/snake.py

snake.py — struktura

```
class Direction(Enum): ...
class GameStatus(Enum): ...

@dataclass
class GameState: ...

def next_head(head, direction): ...
def move_snake(snake, new_head, *, grow): ...
def is_wall_collision(head): ...
def is_self_collision(new_head, body_after_move): ...
def choose_food(snake, rng): ...
def calculate_delay(score): ...
def new_game_state(rng, *, high_score=0): ...

class SnakeApp:
    def __init__(self, *, rng=None): ...
```

```
    def bind_keys(self): ...
    def request_direction(self, direction): ...
    def game_tick(self): ...
    def toggle_pause(self): ...
    def restart(self): ...
    def render(self): ...
    def run(self): ...

def main():
    app = SnakeApp()
    app.run()

if __name__ == "__main__":
    main()
```

Uruchom grę u siebie

`python snake.py` w terminalu. Strzałki lub WASD — ruch, `Space` — pauza, `R` to nowa gra. Porównaj z `snake_basic.py` (Rozdział 19.8) to ten sam pomysł dla gry, ale z zupełnie inną architekturą wewnętrzną.

Lista kontrolna gotowej aplikacji

MODEL

- GameState — snake, direction, food: Position | None, score, status, delay_ms, bez Turtle w środku
- czyste funkcje reguł to next_head, move_snake, is_wall_collision, is_self_collision, choose_food, calculate_delay
- nowa głowa + stare ciało zamiast cyklu po segmentach

CYKL

- tik przez screen.ontimer(), bez cyklu blokującego i time.sleep()
- kierunek klawiatury żąda, kliknij go stosuje

- generowanie (generation) jest wyłączane natychmiast zarówno podczas przerwy, jak i wznowienia – nie tylko podczas restart()
- dokładnie jeden aktywny łańcuch tików w dowolnym momencie czasu

FUNKCJE GRY

- pause/resume bez ponownego tworzenia węża
- restart z zachowaniem zapisu
- prędkość rośnie wraz z wynikiem, nie spadając poniżej minimum
- zapełnione pole to wyraźny GameStatus.WON, a nie pożywienie wewnątrz węża
- HUD — osobny pas nad polem, a nie nad legalnymi kratkami

TESTY

- testowane zasady bez żadnego prawdziwego okna
- deterministyczne jedzenie przez random.Random(seed)
- render() ponownie wykorzysta pulę segmentów Turtle-0

Paleta Gier

Ostateczna wersja gry używa tylko pięciu kolorów – nie po to, by wyglądać skromnie, ale Ponieważ łatwiej jest rozróżnić głowę, ciało i jedzenie po porównaniu, a nie po liczbie odcieni:

Tło
#000000

Głowa
#FFFFFF

Korpus
#4ECDC4

Jedzenie
#FF5D5D

Tekst
#FFFFFF

Praktyka: Snake Pro w całości
Moduł turtle otwiera natywne okno Python – uruchamiane lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/19-32/index.html)

Podsumowanie rozdziału

Od blokującej pętli z globalnymi zmiennymi do modelu stanu z tikiem gry, pauzą, restartem i sprawdzanymi zasadami.

Podsumowanie rozdziału 19

Co zbudowaliśmy i czego się nauczyliśmy

- Realm – Dyskretna siatka z STEP przyrostami Pozycje prawne są zawsze wielokrotnością kroku i nie są arbitralne.
- Kierunek nie jest łańcuch znaków do porównania w if/elif, lecz wektor przemieszczenia (DIRECTION_VECTORS), a zakaz zawracania o 180° jest sprawdzany względem kierunku AKTUALNEGO, a nie już żądanego kolejnego kierunku.
- Tick gry to cały łańcuch zasad (kierunek → ruch → kolizje → jedzenie → renderowanie), a rysowanie to koncepcyjnie osobny etap.
- screen.ontimer() planuje następny tik i od razu zwraca sterowanie — zamiast blokować program w pętli while z time.sleep().
- Model węża — lista logicznych pozycji; nowa głowa plus stare ciało tworzy ruch w jednym przebiegu, bez pętli z ryzykiem pomylenia kolejności.
- Zderzenie z samym sobą jest sprawdzane z ciałem PO zakręcie – w przeciwnym razie legalne wejście do klatki pustego ogona jest błędnie uznawane za stratę.
- Jawne stany READY/RUNNING/PAUSED/GAME_OVER – pauza i restart nie tylko zmieniają obraz, ale też zmieniają stan, co decyduje, czy gra w ogóle tyka.
- Generowanie (generation) chroni restart() przed łańcuchami równoległymi ontimer() jest tym samym problemem co ponowne naciśnięcie przycisku w rozdziale 16.
- GameState nie przechowuje pojedynczego obiektu Turtle – zasady gry są testowane przez zwykłą assert, bez jednego prawdziwego okna.

Most do rozdziału 20

Następne: pełnoprawny silnik gier zamiast Turtle

Snake jest zbudowany na tej samej pętli zdarzeń Tkinter- co gra rysująca z rozdziału 18 — przydatne do samouczków, ale nigdy nie Turtle zaprojektowano jako silnik gry. Następny rozdział, Pygame, wprowadza prawdziwe sprite'y, zarządzanie kolizjami na poziomie biblioteki oraz pętlę klatek zaprojektowaną specjalnie do gier w czasie rzeczywistym.

ROZDZIAŁ 20 · STR. 1288

Tworzenie gier z Pygame

Wprowadzenie do tworzenia gier – gatunków, platform, rozwoju gier mobilnych – oraz Pygame jako narzędzia do ćwiczenia podstawowych zasad GameDev: pętli gry, czasu klatki, sprite'ów, kolizji i architektury gry.

Pygame i pygame-ce

Dedykowane narzędzie do gier – z precyzyjną kontrolą nad klatkami animacji niż Turtle i Tkinter.

Co to jest Pygame i pygame-ce

`Pygame` to biblioteka przeznaczona specjalnie do gier: daje znacznie więcej Większa kontrola nad klatkami animacji, kolizjami i dźwiękiem niż Turtle (Rozdziały 6-7, 12, 19) lub Tkinter (rozdziały 16–18). Dla kontrastu, `Pygame` nie uwzględnione Python w standardowej wersji – musi być montowany osobno.

Pygame ma dwie paczki o prawie tej samej nazwie i warto to od razu wyjaśnić. Klasyk `pygame` jest oryginalnym projektem. **pygame-ce** (Community Edition) jest jego aktywnie wspieraną kontynuacją: API jest w dużej mierze kompatybilna z klasyczny pygame, większość podstawowych przykładów pygame działa bez zmian, a import pozostaje takie samo — `import pygame`. Jednocześnie pygame-ce otrzymuje Nowe wersje i poprawki są instalowane szybciej i bardziej niezawodnie na najnowszych wersjach Python. W tej książce instalujemy i korzystamy z pygame-ce.

Instaluj i importuj Pygame

ustanovka.txt

```
pip install pygame-ce
```

Nazwa pakietu i modułu to różne rzeczy

Nazwa opakowania (co przeszedłeś przez pip) — `pygame-ce`.
Nazwa modułu (to, co importujesz w kodzie) nadal jest `pygame`, bez zmian: pygame-ce specjalnie zachował to imię, aby istniejący kod na pygame działał bez poprawek. Wszystkie kody w tej książce są pisane tak jak `import pygame`.

Nie trzymaj pygame i pygame-ce w tym samym środowisku o tej samej porze.

Oba pakiety są zainstalowane pod tą samą nazwą modułu `pygame`, więc jeśli w środowisku przypadkowo znajdą się oba, nieprzewidywalne, który z nich faktycznie zostanie zaimportowany. Jeśli wcześniej instalowałeś klasyczny `pygame`, usuń to najpierw: `pip uninstall pygame`, a dopiero wtedy wpis `pip install pygame-ce`. Nowe standardowe wirtualne środowisko (rozdział 15) pomaga izolować zależności projektu od pakietów globalnych. Ale nawet w nim nie instaluj jednocześnie pygame i pygame-ce.

import_pygame.py

```
import pygame

pygame.init()   # W projektach w tej książce zaczynamy
przygotowywać Pygame od tego wezwania
```

Praktyka: Instalacja i pierwszy import
Pygame otwiera natywne okno Python – uruchomione lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/20-02/index.html)

Tworzenie okna gry i pierwszej pętli gry

Pętla rozgrywki jest sercem każdej gry na Pygame: jest pisana ręcznie, klatka po klatce.

Tworzenie okna gry

Ekran Pygame — zwykła powierzchnia (`Surface`) dane Rozmiar:

igrovoj_ekran.py

```
import pygame

pygame.init()
screen = pygame.display.set_mode((600, 400))
pygame.display.set_caption(„Moja pierwsza gra”)
```

Prawdziwe okno Pygame: ciemnoniebieskim ekranie 600×400 zalanym kolorem tła

Prawdziwe okno: ekran jest tworzony i wypełniony kolorami – już teraz prawdziwa, choć pusta, gra.

Pętla rozgrywki

W przeciwieństwie do Tkinter z jego `mainloop()`, w Pygame pętla gry jest pisana ręcznie — daje to pełną kontrolę nad tym, co dzieje się w każdej klatce:

igrovoj_cikl.py

```
clock = pygame.time.Clock()
rabotaet = True

while rabotaet:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            rabotaet = False

    pygame.display.flip()   # pokaż narysowaną klatkę
    clock.tick(60)          # nie więcej niż 60 fps

pygame.quit()
```

Po co potrzebny jest pygame.QUIT?
Brak obsługi zdarzeń `pygame.QUIT` (klikając krzyżyk w oknie) program nie wie, że użytkownik chce zamknąć grę, a okno przestaje reagować. Obsługa zdarzeń na początku każdej klatki pętli jest obowiązkową częścią każdego programu na Pygame.

Tło i czyszczenie ramek

`screen.fill(цвет)` maluje cały ekran kolorami (jak RGB, Rozdział 11) jest zazwyczaj pierwszym poleceniem w każdej klatce, w przeciwnym razie „Ślady”

zalivka_fona.py

```
CVET_FONA = (20, 20, 40)

screen.fill(CVET_FONA)
```

Realne okno: Kwadrat przesuwa się w każdej klatce — ten sam ruch co w Sekcji 20.4, ale to sama pętla jest tu najważniejsza.

Praktyka: Pętla gry i wypełnienie tła
Pygame otwiera natywne okno Python – uruchomione lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/20-02/index.html)

Rysujemy pierwsze obiekty gry

Pierwsze obiekty gry — proste figury, rysowane od nowa w każdej klatce.

Obiekty gry w Pygame rysują bezpośrednio na ekranie co klatkę prostymi kształtami (kółka, prostokąty), jak w Turtle, czyli gotowych obrazach. Zacznijmy od Kształty – Nie wymagają zewnętrznych plików i świetnie sprawdzają się w pierwszych grach. Pełnoprawne Postacie z obrazami i animacjami zbiorzemy później (sekcje 20.19 i 20.23) — tutaj Sama zasada jest ważna: obiekt na ekranie to po prostu postać, którą na każdym z nich narysuje się na nowo Ramka na właściwym miejscu.

personazhi.py

```
CVET_IGROKA = (100, 200, 255)
CVET_VRAGA = (255, 80, 80)

x_igroka, y_igroka = 300, 350
x_vraga, y_vraga = 300, 50

# wewnątrz pętli gry, po screen.fill(...):
pygame.draw.rect(screen, CVET_IGROKA, (x_igroka, y_igroka, 40,
40))
pygame.draw.circle(screen, CVET_VRAGA, (x_vraga, y_vraga), 20)
```

Prawdziwe okno: niebieski kwadrat gracza na dole ekranu i czerwony okrągły wróg na górze na czarnym tle

Rzeczywiste okno: te same dwie figury, co w kodzie — prostokąt i koło, każdy w swoim kolorze.

`pygame.draw.rect()` przyjmuje krotkę `(x, y, ширина, высота)` — `(x, y)` to Lewy górny róg, jak `Canvas` w rozdziale 18, nie w środku, czyli w `circle()`.

Rect jest prostokątem jako obiekt
Dla bardziej złożonych gier Pygame oferuje klasę `pygame.Rect` - Przechowuje pozycję i rozmiar razem oraz może sprawdzać przejścia (`.colliderect()`) to to, czego potrzebujesz do kosmicznej strzelanki w rozdziale 21.

Rzeczywiste okno: trzy kolorowe koła w rzędzie — czerwone, zielone, niebieskie — na czarnym tle

Prawdziwe okno: trzech wrogów różnych kolorów w rzędzie, narysowanych w jednym cyklu for.

Praktyka: rysujemy pierwsze obiekty gry
Pygame otwiera natywne okno Python – uruchomione lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/20-03/index.html)

Sterujemy obiektem z klawiatury

Ruch to po prostu zmiana współrzędnych w każdej klatce; klawiaturę można przetestować na dwa różne sposoby.

Ruch obiektów

„Ruch”

peremeshenie.py

```
x_igroka += skorost_x
pygame.draw.rect(screen, CVET_IGROKA, (x_igroka, y_igroka, 40,
40))
```

Prawdziwe okno: to samo zmienna x_igroka w trzech kolejnych obrazach — ruch to zmiana współrzędnych.

Wydarzenia klawiszowe

Są dwa sposoby, aby dowiedzieć się o klawiaturze. Pierwszy — przez zdarzenia, jak w `pygame.event.get()` (wygodne przy jednorazowych akcjach, jak skok):

sobytiya_klavish.py

```
for event in pygame.event.get():
    if event.type == pygame.KEYDOWN:
        if event.key == pygame.K_SPACE:
            print(„Skacz!”)
```

Drugim sposobem jest sprawdzenie aktualnego stanu klawiszy w każdej klatce (wygodniej dla ciągły ruch, na przykład znak w lewo lub prawo):

sostoyanie_klavish.py

```
klavishi = pygame.key.get_pressed()
if klavishi[pygame.K_LEFT]:
    x_igroka -= 5
if klavishi[pygame.K_RIGHT]:
    x_igroka += 5
```

Którą metodę powinienem wybrać?
Wydarzenia (`event.type == pygame.KEYDOWN`) nadają się do działań „raz na naciśnięcie” — skok, strzał, pauza. `get_pressed()` nadaje się do ciągłego poruszania się, gdy klucz jest przytrzymany, co jest zgodne z kontrolą statku w rozdziale 21.

Praktyka: ruch i klawisze
Pygame otwiera natywne okno Python – uruchomione lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/20-04/index.html)

Mini-projekt: odbijająca się piłka

Pierwsza prawdziwa mini-gra na Pygame – ruch, odbicia i pętla rozgrywki w jednym krótkim projekcie.

Wszystko to połącz w jeden mini-projekt: piłkę, która odbija się od wszystkich czterech ścian ekranu.

prygayushij_myach.py

```
import pygame

SHIRINA, VYSOTA = 600, 400
RADIUS = 20

pygame.init()
screen = pygame.display.set_mode((SHIRINA, VYSOTA))
clock = pygame.time.Clock()

x, y = SHIRINA // 2, VYSOTA // 2
dx, dy = 4, 3

rabotaet = True
while rabotaet:
```

```
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            rabotaet = False

    x += dx
    y += dy

    if x - RADIUS < 0 or x + RADIUS > SHIRINA:
        dx = -dx
    if y - RADIUS < 0 or y + RADIUS > VYSOTA:
        dy = -dy

    screen.fill((20, 20, 40))
    pygame.draw.circle(screen, (255, 100, 100), (int(x),
int(y)), RADIUS)
    pygame.display.flip()
    clock.tick(60)

pygame.quit()
```

Trzy prawdziwe okna obok siebie: czerwona kula w różnych miejscach na ciemnoniebieskim polu

Realne okno: Ta sama piłka z projects/pygame/bouncing-ball/bouncing_ball_basic.py na trzech kolejnych pozycjach trajektorii.

dx = -dx — odbicie w jednej linii

Zmiana znaku prędkości na przeciwny — standardowy sposób odbicia od ściany: ruch trwa z tą samą prędkością, tylko w przeciwnym kierunku, bez dodatkowych kontroli kierunku.

Kompletny, już zeskanowany plik – osobno:

projects/pygame/bouncing-ball/bouncing_ball_basic.py

★★ Zadanie samodzielne

Zmieniamy kolor przy odbiciu

Za każdym razem, gdy odbijesz się od ściany, wybierz nowy losowy kolor kuli (random.choice() z rozdziałów 5/11).

Challenge

Dwie Kule

Dodaj drugą piłkę własnym x, y, dx dy – obie powinny się poruszać i odbijać niezależnie.

Praktyka: odbijająca się piłka

Pygame otwiera natywne okno Python – uruchomione lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/20-05/index.html)

Ograniczenia tej wersji

Piłka już się odbija, ale ta wersja ma słabości: prędkość odbicia zależy od liczby klatek na daną komputer, a cały kod leży w zmiennych globalnych bez żadnej struktury. Później w tym rozdziale przyjrzymy się, skąd biorą się oba ograniczenia i jak rozwinąć ten sam projekt w profesjonalnie zorganizowaną wersję z klasą `Game` , z pauzą i ruchem, który nie zależy od liczby klatek.

Jak działa tworzenie gier

Zanim przejdziemy do Pygame, przyjrzyjmy się, czym jest tworzenie gier jako rzemiosło: role, skala i droga od pomysłu do wydania.

Piłka z poprzedniej sekcji już podskakuje — najwyższy czas cofnąć się o krok i zobaczyć, w jaki świat właśnie weszliśmy. Tworzenie gier (game development, gamedev) — to nie jedna umiejętność, lecz rzemiosło na styku programowania, designu i sztuki. Niewielką grę całkiem może stworzyć jedna osoba — właśnie to zrobiłeś. W miarę rozwoju projektu zazwyczaj rośnie też liczba wyspecjalizowanych ról, a duże komercyjne gry tworzą duże zespoły.

Role w tworzeniu gier

W dużym studiu za każdą z tych ról odpowiada osobna osoba lub cały zespół; w projekcie indie ta sama osoba często łączy trzy-cztery role jednocześnie — a pracując nad mini-projektami tego rozdziału, już byliście jednocześnie programistą, projektantem i testerem.

Kto tworzy grę

PROJEKTANT GIER

Wymyśla zasady gry
Równoważy złożoność
Pisze dokument projektowania gry

PROGRAMISTA

Implementuje reguły w kodzie
Pisze pętlę gry
Optymalizacja wydajności

ARTYSTA / ANIMATOR

Rysuje sprite'y i tła
Przygotowuje animacje postaci
Ustala styl wizualny

INŻYNIER DŹWIĘKU

Pisze muzykę
Nagrywa efekty dźwiękowe
Redukuje dźwięk dla zdarzeń gry

PRODUCENT

Planowanie terminów

```
Zarządza zespołem
Pilnuje budżetu
```

QA-TESTER

```
Szuka błędów
Kontroluje równowagę
Gra w grę setki razy przed premierą
```

Indie i AAA

Gry indie zazwyczaj tworzą niezależni deweloperzy lub małe zespoły, często bez dużego wydawcy. «AAA» — oznaczenie dużych projektów komercyjnych z dużymi zespołami, znacznymi budżetami i długim cyklem produkcyjnym. To nie są ścisłe kategorie z wyraźną granicą: rzeczywiste projekty znacznie różnią się skalą, a wiele zespołów znajduje się gdzieś pośrodku.

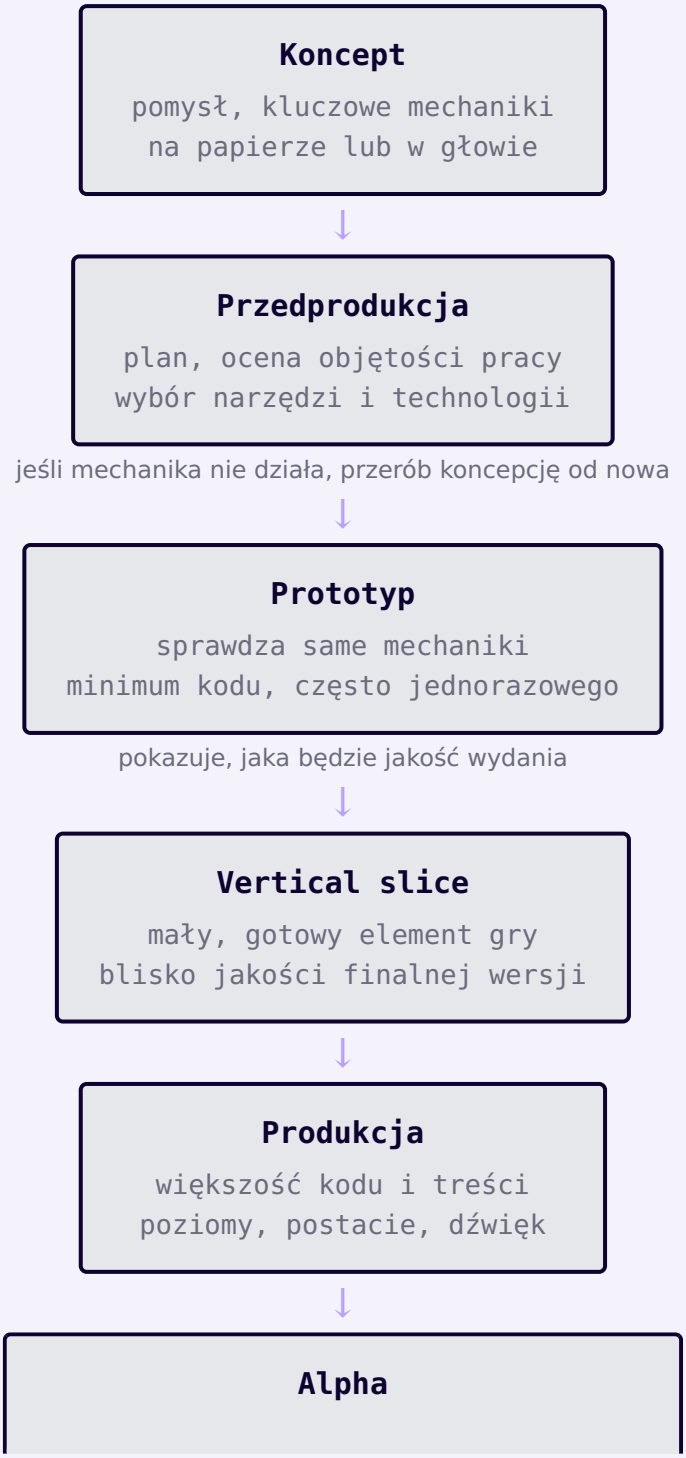
Charakterystyka	Projekt indie	Duży projekt AAA-
Komenda	Od pojedynczego dewelopera do małego lub średniego zespołu	Zazwyczaj duży zespół specjalistów z różnych dziedzin
Okres rozwoju	Od kilku miesięcy do kilku lat	Zazwyczaj kilka lat
Podejmowanie decyzji	Decyzje często podejmowane są szybciej i przez mniej osób	Zmiany mogą wymagać koordynacji między wieloma zespołami
Narzędzia	Gotowe do użycia silniki, biblioteki i dostępne narzędzia programistyczne	Gotowe lub własne technologie i wewnętrzne narzędzia

Charakterystyka	Projekt indie	Duży projekt AAA-
Siła	Elastyczność i swoboda eksperymentowania	Skala produkcji i duże zasoby

Pygame dla pierwszych prototypów indie
Projekty edukacyjne tego rozdziału są napisane w tym samym duchu, co wiele gier indie: mały zespół (w twoim przypadku — jedna osoba), ogólnodostępne narzędzie, szybkie eksperymenty. Różnica między projektem edukacyjnym a prawdziwym wydaniem indie zwykle nie tkwi w narzędziu, lecz w tym, ile iteracji dzieli prototyp od gotowej gry.

Droga od pomysłu do premiery

Żadna gra nie pojawia się gotowa od razu. Proces ma typowe etapy — w konkretnym projekcie ich nazwy i granice mogą się różnić, ale ogólna logika jest podobna zarówno w AAA-studiach, jak i u jednej osoby z laptopem:


Koncept
pomysł, kluczowe mechaniki
na papierze lub w głowie
Przedprodukcja
plan, ocena objętości pracy
wybór narzędzi i technologii
jeśli mechanika nie działa, przerób koncepcję od nowa
Prototyp
sprawdza same mechaniki
minimum kodu, często jednorazowego
pokazuje, jaka będzie jakość wydania
Vertical slice
mały, gotowy element gry
blisko jakości finalnej wersji
Produkcja
większość kodu i treści
poziomy, postacie, dźwięk
Alpha

w konkretnym projekcie kryteria mogą się różnić
zazwyczaj: główna funkcjonalność jest już dostępna i testowana w zespole

↓

Beta

zwykle: zawartość prawie się nie zmienia
testowanie na zewnątrz, polowanie na owady

↓

Premiera

gra należy do graczy

życie gry trwa dalej po premierze

↓

Wsparcie po wydaniu

łatki, balans, nowa zawartość

Piłka odbijająca się od ścian to już działający prototyp: minimalna mechanika, jaką można wypróbować. To jedna z najczęstszych opcji drogi od pomysłu do premiery — nie jedyny poprawny schemat.

Prototypowanie i vertical slice to różne zadania

Prototyp odpowiada na pytanie "czy sam pomysł działa" — kod w nim jest prawie zawsze jednorazowy, a jakość obrazu i tekstu nie ma znaczenia. Vertical slice odpowiada na inne pytanie: "jak będzie wyglądać i jak będzie wyglądać gotowa gra" — to już niewielka, ale pełnoprawna część zbliżona do docelowej jakości, często jednocześnie sprawdza pipeline produkcyjny projektu. W praktyce zespół może zbudować architekturę techniczną (sekcja 20.26) wcześniej, nawet na etapie prototypu, a zadania prototypu i vertical slice się pokrywają: różne projekty organizują te etapy na swój sposób.

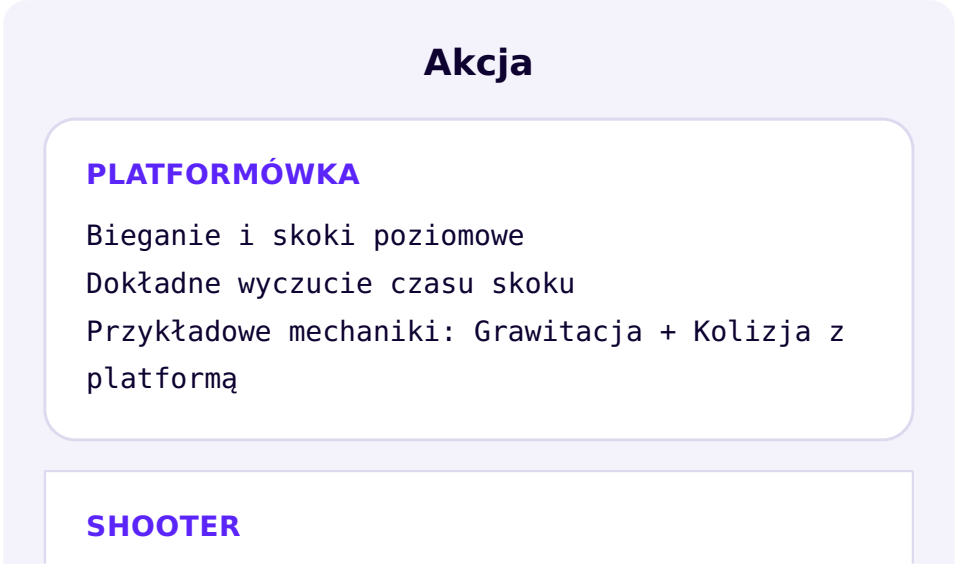
Gatunki gier i mechaniki gry

Gatunek — to przede wszystkim mechanika: co dokładnie gracz robi od klatki do klatki.

Gry zwykle dzielą się na gatunki — nie jest to ścisła klasyfikacja naukowa, lecz wygodny język wspólny: Gdy gracz słyszy "platformówkę" lub "zagadkę", już ma oczekiwanie, że Wewnątrz. Gatunek jest przede wszystkim o **mechanika** (to, co robi gracz), nie o fabule lub obrazie.

Akcja

Gry akcji opierają się na szybkiej reakcji i koordynacji — to raczej duża rodzina niż jeden gatunek:



Strzelanie do celów
Szybka odpowiedź
Przykład: pocisk – to kolejny poruszający się obiekt

GRA BIJATYKI

Pojedynek jeden na jednego lub drużynowy pojedynek
Dokładne czasy uderzeń i bloków
Przykład: automat stanów postaci

BEAT 'EM UP

Walcz z kilkoma przeciwnikami jednocześnie
Przechodzisz przez poziom z falami wrogów
Przykład: lista aktywnych przeciwników, sprawdzając, czy spotkania z każdym z nich

Przygodówki i gry fabularne

Przygody i RPG

PRZYGODA (ADVENTURE)

Eksploracja świata i rozwiązywanie problemów zgodnie z fabułą

Reakcja zwykle nie jest krytyczna
Przykład: Stan gry jako automat stanów

POINT-AND-CLICK QUEST

Interakcja z obiektami sceny za pomocą kliknięcia
Łamigłówki wplecione w fabułę
Przykład: lista interaktywne obiekty na ekranie

ACTION RPG

Rozwój postaci oraz walka w czasie rzeczywistym
Cechy i postęp
Przykład: Inwentarz – Struktura danych rozdziału 11

KROK PO KROKU RPG

Rozwój postaci oraz turowa walka
Mniej refleksu, więcej taktyki
Przykład: kolejka do tury postaci

RPG TAKTYCZNE

Walka na szachownicy
Pozycja i kolejność ruchów decydują o wyniku

Przykład: plansza gry – siatka pól z zajętością

Strategia i symulacje

Strategia i symulacje

RTS (STRATEGIA W CZASIE RZECZYWISTYM)

Zarządzanie zasobami i jednostkami bez zatrzymywania gry
Planowanie jednocześnie z reakcją
Przykład: Jednostki to lista obiekty posiadające stan

STRATEGIA TUROWA

Planowanie wielu ruchów z wyprzedzeniem
Nie ma limitu czasu między turami
Przykład: kolejka tur jak w turze RPG

4X

Rozwój terytorialny, rozwój, dyplomacja, wojna
Zwykle długie partie na dużej mapie
Nazwa pochodzi od czterech goli: eXplore, eXpand, eXploit, eXterminate

TOWER DEFENSE

Ustawianie obiektów obronnych na drodze wrogów
Fale przeciwników według harmonogramu
Przykład: lista wieże, lista wrogów, kontrola zasięgu

SYMULATOR TRANSPORTU

Sterowanie jednym samochodem, samolotem i podobnym
Fizyka ruchu – mechanika centralna
Przykład: przyspieszanie, hamowanie, bezwładność

SYMULATOR ŻYCIA

Modelowanie życia lub relacji postaci
Postęp mierzy się metrykami, a nie walką
Przykład: Zestaw cech liczbowych znaku

ZARZĄDZANIE / CITY BUILDER

Budowa i zarządzanie systemem (miasto, farma, baza)
Równowaga zasobów jest ważniejsza niż reakcja
Przykład: Model numeryczny + wizualizacja

Kolejne gatunki, które warto znać

Więcej gatunków

GRY SPORTOWE

Symulacja prawdziwego sportu
Zasady ustala sama dyscyplina sportowa

WYŚCIGI

Zarządzanie transportem na trasie
Konkurs na czas
Przykład: ruch z przyspieszeniem i tarciem

PUZZLE

Logika jest ważniejsza niż reakcja
Jasne zasady zwycięstwa
Przykład: pole gry – siatka stanów

ARCADE

Proste zasady, rosnąca złożoność
Wyniki i rekordy
Przykład: nasza skacząca piłka

SURVIVAL

Przetrwanie w wrogim środowisku
Zarządzanie zasobami: zdrowie, głód, schronienie

SANDBOX

Wolność działania jest ważniejsza niż sam cel
Gracz sam ustala zadania

ROGUELIKE / ROGUELITE

Losowo generowane poziomy
Śmierć zwykle zaczyna grę od nowa
Przykład: random z rozdziału 5, ale na poziomie całej serii

CYFROWE GRY KARCIANE I PLANSZOWE

Cyfrowa wersja mechaniki kart lub planszy
Przykład: Talia i ręka – listy kart

MMO / ONLINE WIELOOSOBOWE

Wiele graczy jednocześnie w ogólnym świecie
Wymaga interakcji sieciowej – temat osobnej książki

IDLE / INCREMENTAL

Postęp trwa nawet bez aktywnych akcji gracza
Przykład: Liczby rosną według wzoru ograniczonego czasowo

CASUAL I HYPERCASUAL

Bardzo proste zasady, krótka sesja
Często jedna mechanika całej gry jest jak nasza odbijająca się piłka

Gatunki mieszają się

W praktyce niewiele gier to tylko jeden gatunek z podręcznika: strzelanka z elementami RPG (levelowanie broni), platformówka logiczna (poziom rozwiązuje się przez skoki w odpowiedniej kolejności), Strategia czasu rzeczywistego z elementami symulatora. Granice między sąsiadującymi gatunkami w powyższa lista często się zaciera — Roguelike przecina się z RPG, tower defense — ze strategią, idle z symulatorem. Klasyfikacja jest narzędziem komunikacji, a nie klatką, której jest zobowiązana by pasować do tej idei.

Od czego zacząć pierwszy projekt?

Uwielbiam precyzyjne wyczucie czasu i reakcje → **Platformówka / Strzelanka / Arcade**

lubię liczyć z wyprzedzeniem, a nie reagować	→ **Strategia / zagadka**
Chcę opowiadać historię przez grę	→ **Przygoda / RPG**
chcę zasymulować coś z rzeczywistości	→ **Symulacja**

Gatunek i pierwszy projekt

Skacząca piłka z rozdziału 20.5 — edukacyjna arcade: prosta zasada (odbicie od ścian) plus liczenie punktów. Projekt następnego rozdziału — kosmiczny shooter — jest zbudowany w tym samym duchu, ale dodaje strzelanie i kolizje z wrogami. Rozumienie gatunku wcześniej pomaga przewidzieć, jakie mechaniki (rozdziały 20.21–20.22) będą potrzebne wcześniej niż pozostałe.

Biblioteka, framework lub silnik: gdzie znajduje się Pygame

Pygame to biblioteka, a nie silnik: dostarcza cegiełki, a strukturę gry budujesz samodzielnie.

„Game Development”

Trzy poziomy narzędzi

Od cegieł do gotowych warsztatów

BIBLIOTEKA

Daje pojedyncze „cegły”
Sam budujesz strukturę aplikacji
Pygame jest biblioteka

RAMY

Ustala ogólną strukturę aplikacji
Wypełniasz przewidziane miejsca swoim kodem
Przykład ducha podejścia: Flask z rozdziału 22 dla internetu

SILNIK

Pełne środowisko: Edytor wizualny, Fizyka, Światło, Dźwięk od razu po wyjęciu z pudełka
Piszesz mniej kodu, konfigurujesz więcej komponentów
Przykłady: uniwersalne silniki gier z edytorem scen

Pygame jest biblioteką: zapewnia okno, powierzchnię do rysowania, obsługę kluczy i dźwięku, ale pętla gry, kolizje i struktura całego projektu są napisane przez ciebie, z twoją własną ręce. To świadomy kompromis między poziomem kontroli a szybkością startu:

	Pygame (biblioteka)	Typowy silnik gry
Krzywa uczenia się	Musisz zrozumieć cykl, ramy, współrzędne	Wiele można zrobić bez kodu

	Pygame (biblioteka)	Typowy silnik gry
Kontrola nad szczegółami	Bardziej bezpośrednia kontrola nad cyklem i decyzje na niskim szczeblu	Bardziej gotowa konstrukcja; stopień sterowania zależy od silnika
Edytor scen wizualnych	Brak – scena jest budowana przez kod	Zwykle jest
Wbudowana fizyka/światło	Nie – sam to implementujesz lub łączysz osobno	Zwykle jest gotowa od razu
Co otrzymujesz w zamian	Głębokie zrozumienie, jak gra działa od środka na zewnątrz	Szybkość montażu dużego projektu

Pygame i SDL

Sam Pygame nie rysuje pikseli i nie komunikuje się bezpośrednio z kartą dźwiękową — pod maską opiera się na bibliotece **SDL** (Simple DirectMedia Layer) napisane w C: Odpowiada za pracę niskopoziomową z oknami, wejściem, grafiką i dźwiękiem na różnych systemów operacyjnych. Pygame nadaje mu wygodną, przypominającą Pythona otoczenie otoczeniem.

Po co uczyć się na bibliotece, jeśli są silniki

Koncepcje, których poznajesz w tym rozdziale — pętla gry, klatka, czas między klatkami, układ współrzędnych, kolizja prostokąta — są przenoszone między narzędziami do tworzenia gier w ich podstawowych ideach, w tym dużymi silnikami wizualnymi. Jednak konkretna architektura, API, timestep fizyczny, model sceny i system kolizji mogą być bardzo różne, a silnik często ukrywa te koncepcje za interfejsem przycisków i suwaków. Ucząc się ich bezpośrednio przez kod, lepiej zrozumiesz, co dzieje się za tymi przyciskami — nie tylko jak je klikać.

Platformy gamingowe: komputery, konsole i urządzenia przenośne

Gra została napisana na konkretne urządzenie — i jest skierowana głównie Pygame na komputer stacjonarny.

Gra prawie nigdy nie jest pisana „ogólnie”, lecz pod konkretną platformę. Od platformy zależy także wprowadzanie (klawiatura czy ekran dotykowy?), oraz dystrybucja (plik do pobrania czy sklep z aplikacjami?), oraz ograniczenia techniczne (ile pamięci jest dostępne?). Warto od razu rozróżnić dwa różne pytania: na jakiej **urządzenie** gra się uruchamia — i jakim **sposób** dotrze do tego urządzenia. Pierwszym jest pulpit, urządzenia mobilne, konsole, urządzenia przenośne XR; drugi (np. gry w chmurze od sekcja 20.10) to bardziej sposób na dostarczenie gry niż osobny typ urządzenia: oparty na chmurze W rezultacie usługa nadal uruchamia grę na prawdziwym komputerze lub serwerze.

Urządzenia docelowe

Urządzenia docelowe

PULPIT

Windows, macOS, Linux
Klawiatura, mysz, gamepad
Bezpośrednie udostępnienie pliku lub sklep jak Steam

KONSOLE

PlayStation, Xbox, Nintendo i podobne
Oficjalny zestaw dla deweloperów (SDK) i certyfikacja
edycja wymaga umowy z producentem konsoli

URZĄDZENIA PRZENOŚNE

Przenośna konsola lub laptop PC producenta z pełnym systemem operacyjnym
Ekran dotykowy, przyciski fizyczne lub jedno i drugie
Często bardziej ograniczone pod względem baterii i pamięci niż urządzenie stacjonarne

MOBILE

Android, iOS
Ekran dotykowy, akcelerometr
Sklepy z aplikacjami – Rozdział 20.11 – 20.13

SIEĆ

Uruchamia się bezpośrednio w przeglądarce
Nie jest wymagana instalacja
WebAssembly - Rozdział 20.30

XR (VR/AR)

Zestawy VR i rozszerzonej rzeczywistości
Renderowanie stereo, śledzenie głowy i rąk
Oddzielny, wysoce wyspecjalizowany rozwój

Urządzenie przenośne — nie zawsze konsola do gier

"urządzenie przenośne" może oznaczać dwie różne rzeczy: przenośną konsolę producenta (z własnym zamkniętym ekosystemem i SDK jak duże konsole) lub przenośny komputer z regularnym systemem operacyjnym desktop, który może uruchomić dowolny program desktopowy, w tym napisany w Pygame. Różnica ta jest fundamentalna dla dystrybucji gry i powinna być wyjaśniona dla konkretnego urządzenia, a nie traktowana jako "przenośna" jako jedna jednorodna kategoria.

Desktop to natywna platforma Pygame

Pygame jest pierwotnie i przede wszystkim narzędziem do gier desktopowych na Windows, macOS i Linux. Wszystko, co napisałeś w tym rozdziale, działa jak zwykły program: kliknij dwa razy Według pliku (po opakowaniu, sekcja 20.29) lub przez `python имя_файла.py`.

Konsole i konsole przenośne

Tu potrzebna jest szczerość: Pygame nie ma oficjalnego wsparcia dla konsol do gier. Wydanie gry na konsolę wymaga oficjalnego zestawu narzędzi developerskich (SDK) od producenta samej konsoli, podpisania umowy developerskiej i przejścia certyfikacji — to osobny, zamknięty proces, który nie ma żadnego związku z Pygame ani Python.")

Konsole — nie droga Pygame

Jeśli ostatecznym celem jest wydanie gry na konsolę, Pygame nie jest właściwym narzędziem na rozpoczęcie tej konkretnej ścieżki: tworzenie konsol prawie zawsze odbywa się przez oficjalne silniki i SDK dostawców. Ale umiejętności zdobyte tutaj — pętlowanie gry, praca nad kadrami, architektura — są przekazywane bezpośrednio do tych narzędzi, bez strat.

Gry internetowe, XR i gry w chmurze

Trzy sposoby dotarcia do gracza, zorganizowane zupełnie inaczej niż zwykłe uruchomienie na desktopie.

Web: gra bezpośrednio w przeglądarce

Gra internetowa nie wymaga instalacji – gracz otwiera link i gra bezpośrednio w przeglądarce. Pomiędzy Twój Python- kod na Pygame i przeglądarce ma kilka warstw, a nie jedną magiczną Zmiana:

WebAssembly — to format bajt-kodu, który wykonuje przeglądarka; sam w sobie nie uruchamia zwykłego desktopowego Python — do tego potrzebny jest osobno zbudowany runtime i narzędzie typu pygbag.

XR: wirtualna i rozszerzona rzeczywistość

XR (extended reality) integruje rzeczywistość wirtualną (VR, całkowicie cyfrową) środowiska w gogle) oraz rozszerzonej rzeczywistości (AR, cyfrowych obiektów na prawdziwych obiektach świat). Rozwój XR wymaga renderowania stereo (osobny obraz dla każdego oka), śledzenie pozycji głowy i dłoni oraz ścisłe wymagania dotyczące liczby klatek na sekundę, aby nie powodować Gracz cierpi na chorobę lokomocyjną. To osobny, wysoce wyspecjalizowany obszar rozwoju — Pygame nie Zaprojektowane dla XR.

Gry w chmurze to sposób dostarczania, a nie osobne urządzenie

W przypadku gier w chmurze gra działa na serwerze w centrum danych — a właściwie na zwykłym serwerze Sprzęt stacjonarny lub serwerowy, ale na urządzeniu gracza w czasie rzeczywistym przesyłany jest tylko strumień wideo, jak podczas rozmowy wideo z grą po drugiej stronie. Dlatego chmura Bardziej szczerze mówiąc, można opisać gry nie jako kolejną platformę na poziomie desktopowym czy mobilnym, ale Jako sposób *dostawy* już istniejącej wersji desktopowej (lub konsolowej) gry do ekranu, na którym fizycznie jej nie uruchomiono. To usuwa wymagania dotyczące mocy urządzenia gracza, ale wymaga stabilnego i szybkiego połączenia internetowego.

Podsumowanie: Gdzie Pygame pasuje bezpośrednio

Kuda	Typowe dane wejściowe	Jak gra trafia do gracza	Pygame jest właściwym narzędziem?
Pulpit	Klawiatura, mysz, gamepad	Pobieranie pliku lub sklepu, np. Steam	Tak – natywna platforma
Konsole	Producent padów do gry	Oficjalny sklep konsoli, przez SDK	Nie — potrzebny jest oficjalny SDK
Konsole przenośne	Ekran dotykowy lub przyciski	Sklep z urządzeniami	Brak bezpośrednich - Zobacz Sekcję 20.13

Kuda	Typowe dane wejściowe	Jak gra trafia do gracza	Pygame jest właściwym narzędziem?
Mobile	Ekran dotykowy, akcelerometr	App Store, Google Play	Tylko za pomocą narzędzi społeczności — sekcja 20.31
Sieć	Klawiatura, mysz, touch	Link w przeglądarce	Tak, przez pygbag — rozdział 20.30
XR	Kontrolery, śledzenie ręki	Sklep ze słuchawkami	Nie – XR- potrzebny silnik

Cloud gaming nie został celowo uwzględniony w tej tabeli: nie zmienia odpowiedzi na pytanie „czy Pygame pasuje”, a jedynie dodaje kolejny sposób dostarczenia gotowej kompilacji desktopowej na ekran gracza.

Gdzie Pygame jest najsilniejsze

Spośród docelowych urządzeń Pygame najbardziej pewny siebie na komputerze stacjonarnym i potrafi, z rezerwacją oraz dzięki narzędziom społecznościowym, docierać do urządzeń internetowych i mobilnych (sekcje 20.11–20.13, 20.30–20.31). Znajomość tych granic z wyprzedzeniem pozwala uniknąć frustracji na późnym etapie projektu — znacznie lepiej wybrać docelowe urządzenie niż natknąć się na narzędzie, które nie pasuje do już wybranego urządzenia.

Gry mobilne: sterowanie dotykowe i wirtualne

Na urządzeniu mobilnym podstawowy sposób wprowadzania — ekran dotykowy, i to zmienia zasady sterowania grą.

Gry mobilne docierają do ogromnej publiczności, a telefony i tablety mają inne podejście Główną metodą wprowadzania jest wbudowany ekran dotykowy zamiast klawiatury i myszy. Zewnętrzne Klawiatura, mysz lub gamepad również mogą być podłączone do telefonu lub tabletu, ale Nie powinieneś polegać na nich jako na głównej metodzie sterowania — gra powinna być w pełni Pracuj tylko na ekranie dotykowym. Zacznijmy od tego, jak to działa.

Ekran dotykowy i multitouch

Dotknięcie ekranu informuje grę w podobny sposób jak Pygame raporty kluczowe, zdarzenie z współrzędnymi. Kluczowa różnica polega na tym, że ekran może natychmiast **kilka** jednoczesne punkty dotyku (multitouch) — na przykład jeden palec przesuwa postać, a drugi w tym samym czasie naciska przycisk strzału. W desktopowym Pygame zdarzenie myszy zawsze ma jeden punkt i jeden przycisk; w tworzeniu aplikacji mobilnych trzeba od samego początku projektować pod kilka niezależnych punktów dotyku jednocześnie.

Wirtualne sterowanie

Bez fizycznej klawiatury przyciski ruchu, skoku i strzału muszą być rysowane bezpośrednio na ekran — wirtualny joystick, wirtualne przyciski. To nie jest drobiazg, lecz osobna część Projekt gry: Wirtualne sterowanie powinno być na tyle duże, by pomieścić palec (palec jest znacznie bardziej niedokładny niż wskaźnik myszy), nie nakładają się na ważną część ekranu i pozostają Responsywny nawet przy szybkich ruchach.

Real Window: Okrągły wirtualny joystick po lewej i dwa okrągłe wirtualne przyciski po prawej na ciemnym tle

Rzeczywiste okno: makieta wirtualnych elementów sterujących, narysowana zwykłymi pygame.draw.circle() — duże strefy pod palec, a nie pod kursor myszy.

	Klawiatura + mysz (komputer stacjonarny)	Ekran dotykowy (telefon komórkowy)
Dokładność wskazań	Wysoka — piksel pod kursorem	Poniżej — palec szerszy niż kursor
Liczba jednoczesnych wejść	Ograniczenie do liczby kluczy	Wielokrotne niezależne dotyki (multi-touch)
Sprzężenie zwrotne dotykowe	Fizyczne naciśnięcie klawisza po naciśnięciu	Ekran jest płaski, ale urządzenie może reagować wibracjami oprogramowania

	Klawiatura + mysz (komputer stacjonarny)	Ekran dotykowy (telefon komórkowy)
Miejsce na ekranie	Nie zajmuje pola gry	Wirtualne przyciski nakładają się na część ekranu

Akcja w grze jako abstrakcja

Aby ta sama logika gry działała na klawiaturze, padzie i ekranie dotykowym, Warto je oddzielić *akcja w grze* ("przesuń się w prawo", "skocz") z konkretnego fizyczne wejście, które go wywołuje. Wtedy "skok" może oznaczać naciśnięcie spacji na Na komputerze stacjonarnym, A przycisk na padzie lub stuknięcie wirtualnego przycisku na telefonie — oraz kod do gry, która reaguje na "skok", w ogóle się nie zmienia:

input_action.py

```
# Schematycznie: to samo wywołanie nie zależy od źródła wejściowego
def obrabotat_dejstviya(igra):
    if igra.dejstvie_nazhato("prygok"):
        igrok.prygnut()

# dejstvie_nazhato() wewnątrz sprawdza albo klawisz, albo gamepad,
# albo trafienie tapem w obszar wirtualnego przycisku — gracz tego nie widzi
```

Abstrakcji danych wejściowych – nie tylko na urządzeniach mobilnych

To rozdzielenie jest przydatne nawet w czysto desktopowych grach: pozwala później dodać wsparcie dla gamepada lub konfigurowalne klawisze bez konieczności przepisywania logiki gry. Architektura klas `Game` w sekcji 20.26 dokładnie od samego początku określa ten podział.

Ekran gry mobilnej: orientacja, proporcje obrazu i High-DPI

Ekran mobilny nie przypomina okna desktopowego: obraca się, nie ma standardowego stosunku boków i często wysoką gęstość pikseli.

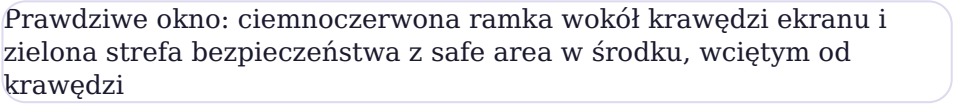
Na komputerze gra prawie zawsze otrzymuje okno o stałym lub dowolnie zmiennym rozmiarze, które wybiera sam gracz. Ekran mobilny jest inaczej skonstruowany — i ma od razu kilka własnych źródeł trudności: orientacja, proporcje, bezpieczny obszar i wysoka gęstość pikseli.

Orientacja ekranu

Telefon można trzymać pionowo (portrait) lub poziomo (landscape)), a odtwarzacz ma prawo obracać urządzenie w dowolnym momencie. Gra musi albo wyraźnie ograniczyć się do jednego orientacja (częsty wybór w salonach gier i zagadek), albo szczerze mówiąc, przeliczenie układu ekran podczas obracania — czyli ponowne zdecydowanie, gdzie umieścić pole gry i wirtualne Przyciski sterujące z sekcji 20.11.

Proporcje i bezpieczny obszar

Ekrany różnych telefonów różnią się bardzo różnymi proporcjami obrazu – w przeciwieństwie do komputerów stacjonarnych monitorów, gdzie prawie zawsze występuje prostokąt szerokoekranowy, tutaj zakres jest znacznie szerszy, od prawie kwadratowe do mocno wydłużonych. Dodatkowo wiele nowoczesnych ekranów ma wycięcia pod kamerą i zaokrąglonymi rogami — obszar ekranu widoczny fizycznie dla użytkownika, ale niebezpieczne dla ważnych elementów interfejsu. Deweloperzy uważają ją za odpowiednią do tworzenia treści Część ekranu **safe area** (bezpieczny obszar) — licznik i ważne przyciski umieszczają się w jego wnętrzu, a nie przylegają do samej krawędzi ekranu.



Prawdziwe okno: strefa bezpieczeństwa to prostokąt z wgłębieniem na każdej krawędzi, gdzie umieszczone są wyniki i ważne przyciski.

Ta sama zasada znajduje się na stronie internetowej książki

Pomysł safe area znany ci nie tylko z rozwoju mobilnego: strona tej książki jest też projektowana z myślą o małych ekranach — treść nie przylega do samej krawędzi na małych telefonach, lecz pozostawia marginesy właśnie z tego powodu.

Rozdzielczość i gęstość pikseli to różne rzeczy

Rozwiązanie — to po prostu liczba pikseli ekranu, na przykład 2400×1080. **Gęstość pikseli** (PPI/DPI, pixels per inch) jak ciasno są te Piksele są umieszczone na rzeczywistym fizycznym rozmiarze ekranu. Nowoczesne telefony mają fizyczne Ekran jest mały i często ma więcej pikseli niż wyraźnie większy pulpit monitor — to znaczy, gęstość jest wyższa przy porównywalnej lub nawet niższej rozdzielczości. Ekran z Wysoka gęstość nazywana jest high-DPI.

Stąd trzecia koncepcja — **logiczny rozmiar**: to, w jakich jednostkach twój kod i interfejs myślą o ekranie, w przeciwieństwie do tego, ile faktycznie ma fizycznych pikseli:

Obraz narysowany „jeden logiczny piksel, jeden fizyczny”

Cechy ekranu	Pulpit	Mobile
Orientacja	Zazwyczaj stałe, nie zmienia się podczas gry	Mogą ulec zmianie w dowolnym momencie

Cechy ekranu	Pulpit	Mobile
Proporcje	Standardowy wąski zakres	Szerokie rozmieszczenie między urządzeniami
Wycięcia i zaokrąglone rogi	W praktyce prawie się nie spotyka	To powszechne, że potrzebuje się safe area
Gęstość pikseli	Zazwyczaj niższy	Często zauważalnie wyższe — potrzebujesz wagi do high-DPI

Ograniczenia gier mobilnych: bateria, pamięć i cykl życia

Bateria, Przegrzewanie, Pamięć, Cykl życia aplikacji – oraz prawdziwa odpowiedź na pytanie o wydanie Pygame- gry w App Store i Google Play.

Przeanalizujmy ograniczenia rozwoju urządzeń mobilnych, które nie są widoczne na pierwszy rzut oka, oraz pytanie, która prędzej czy później pojawia się u każdego, kto dotarł do tego miejsca: czy to możliwe Wziąć i opublikować projekt Pygame- do App Store i Google Play?

Bateria i przegrzewanie się

Telefon działa na ograniczonej baterii i ma ograniczony budżet cieplny — zauważalnie bardziej rygorystyczny niż w laptopie, który często jest podłączony do gniazdka lub ma baterię o większej pojemności. Cykl gry, który utrzymuje procesor w pełnym obciążeniu bez przerwy (zob. rozdział 20.15 o `clock.tick()`), długoterminowe duże obciążenie CPU/GPU przyspiesza rozładowywanie baterii i może prowadzić do thermal throttling – systemowego spadku Częstotliwość CPU, aby chronić przed przegrzaniem, które powoduje spowolnienie gry, nie z powodu Zły kod, ale właśnie przez tę ochronę.

Pamięć

Urządzenia mobilne zwykle mają zauważalnie mniej pamięci RAM niż desktop, a system operacyjny jest znacznie bardziej agresywny w monitorowaniu aplikacji, które zużywają go za dużo.

Cykl życia aplikacji

Aplikacje desktopowe też nie działają w próżni: mogą tracić koncentrację, zasnąć razem z komputerem lub z powodu awarii. Ale na telefonie system operacyjny steruje Cykl życia aplikacji jest zauważalnie bardziej aktywny i częstszy: może przesłać grę w dowolnym momencie w tle (połączenie przyszło, gracz minimalizował aplikację) i później lub odpowiedział dokładnie z tego samego miejsca miejsce, a jeśli nie ma wystarczająco dużo pamięci, całkowicie ją wyładować z pamięci i przy ponownym uruchomieniu Gra zaczyna się od nowa. Gra mobilna powinna prawidłowo doświadczać przejść między foreground a background. Jeśli utrata aktualnego stanu jest ważna dla gracza, aplikacja musi zapisać niezbędny postęp lub checkpoint do możliwego zakończenia procesu.

Sklepy z aplikacjami, reklamy i zakupy w grze

Publikacja na telefon prawie zawsze oznacza publikację za pośrednictwem sklepu z aplikacjami — Google Play dla Android lub App Store dla iOS. Oba mają proces weryfikacji przed publikacją, zasady tworzenia strony aplikacji i (dla wielu kategorii gier) typowy model biznesowy poprzez wyświetlanie reklam lub zakupy w grze (in-app purchases, IAP) — to osobny duży temat biznesu gier mobilnych, wykraczający poza ramy tej książki, ale warto wiedzieć, że istnieje i wpływa na to, jak skonstruowane są wiele gier mobilnych „od środka”.

Publikacja gry Pygame- na Android i iOS

Pygame nie ma wbudowanych eksportów do Android i iOS

Pygame pygame-ce nie ma oficjalnego, wbudowanego polecenia "zbuduj APK" lub "zbuduj aplikację dla iOS". W momencie sprawdzania główne repozytorium narzędzia firm trzecich `python-for-android` nie ma akceptowanej oficjalnej receptury na pygame-ce — istnieją tylko otwarte community pull request i niestandardowe przepisy, więc eksperymentalne buildy na Android są możliwe, ale nie można tego uznać za standardowy ani gwarantowany sposób wspierania. Publikowanie do iOS jest możliwe za pomocą osobnego narzędzia społecznościowego (szablon projektu plus wbudowany Xcode), którego autor sam opisuje jako niewystarczająco przetestowane i niegotowe do produkcji, a także nie testował osobiście wersji dla App Store. Obie ścieżki wymagają znacznie więcej kroków niż `pip install`, a wynik należy sprawdzić na rzeczywistym urządzeniu, a nie tylko zakładać na podstawie dokumentacji narzędzia.

To nie oznacza, że projekt mobilnej Pygame- jest niemożliwy – sekcja 20.31 dzieli Obie ścieżki (dla Android i dla iOS) oraz jak wyglądają w praktyce. Jest to zawarte w planie projektu z wyprzedzeniem, a nie w ostatniej chwili: jeśli ostatecznym celem są magazyny Ograniczenia aplikacji, ograniczenia tej sekcji (zużycie energii, pamięć, cykl życia, budowanie poprzez narzędzia społeczności) powinny być rozpatrywane od samego początku projektu, a nie Spróbuj je naprawić po fakcie, gdy gra jest już napisana pod kątem założeń desktopowych.

Ograniczenia	Pulpit	Mobile
Źródło zasilania	Często gniazdo lub bateria zaprojektowana tak, by trwała	Zawsze ograniczona bateria
Reakcja na obciążenie	Rzadko staje się problemem	Przegrzewanie zmniejsza wydajność
Pamięć operacyjna	Zwykle dużo	Znacznie mniej, surowsze limity
Zarządzanie cyklem życia systemu operacyjnego	Mniej aktywny	Aktywne — tło, wyładunek, ponowne uruchomienie
Ścieżka publikacji	Bezpośrednia dystrybucja pliku lub Steam	App Store oraz tworzenie narzędzi firm trzecich

Jak działa pętla gry: wejście, aktualizacja i renderowanie

Podstawowym modelem pętli gry w tej książce są trzy powtarzalne fazy: proces wprowadzania, stan aktualizacji, rysowanie klatki.

W sekcji 20.2 już napisałeś pętlę gry — ale teraz, znając kontekst całego rozdziału, powinieneś Nazywaj jego części dokładnymi nazwami. W naszych Pygame- gier używamy podstawowego modelu: dowolnych Cykl w tej książce, od najprostszej piłki do końcowego projektu, składa się z trzech powtarzalnych fazy.

Input → Update → Render jest podstawowym modelem, na którym opierają się wszystkie projekty w tym rozdziale.

Kolejność faz jest ważna: wprowadzanie jest przetwarzane wcześniej niż aktualizacja stanu (w przeciwnym razie postać reaguje na klawisz z opóźnieniem jednej klatki), a aktualizacja — wcześniej niż rysowanie (w przeciwnym razie na ekranie pojawi się stara, jeszcze nie zaktualizowana klatka). W rozdziale 20.26 te trzy fazy staną się trzema metodami klasy `Game`.

Duże silniki mają bardziej złożony schemat

Input → Update → Render solidna podstawa dla projektów edukacyjnych i małych, ale nie jedyny możliwy schemat. Duże silniki gier często stosują stały krok fizyczny oddzielny od liczby klatek na ekran, kilka różnych faz odświeżania (np. logikę i animację osobno), interpolację między klatkami dla płynniejszego obrazu, kolejki zdarzeń priorytetowych oraz czasem wiele wątków wykonania. Te komplikacje są tu celowo zbędne — trzy fazy wystarczą, by zrozumieć zasadę, a bardziej złożone schematy można wrócić później, gdy projekt naprawdę tego wymaga.

Zapomniałem pygame.init()

`pygame.init()` nie jest jedynym sposobem na inicjalizację Pygame —możesz inicjować potrzebne moduły pojedynczo—ale w projektach w tej książce zawsze zaczynamy od niego: to wygodny sposób na inicjalizację wszystkich modułów, które wymagają inicjalizacji jednocześnie. Jeśli go pominiesz, niektóre moduły (głównie audio) nie będą gotowe, a komunikaty o błędach czasem wskazują na złe miejsce `pygame.init()`, ale na zupełnie innej linii — gdzie najpierw potrzebował Pygame nieinicjalizowanego modułu.

Kolejka zdarzeń nie jest przetwarzana – okno „zawiesza się”

System operacyjny uznaje okno za zawieszone, jeśli nie zajmuje mu dużo czasu pobranie zdarzeń z kolejki — `pygame.event.get()` powinien być wywoływany w każdej klatce, nawet jeśli wewnątrz pętli `for event in ...` nic nie robisz w większości zdarzeń. Przegapienie tego połączenia na kilka sekund to częsty powód oznaczenia „Nie odpowiada”

Praktyka: trzy fazy cyklu gry

Pygame otwiera natywne okno Python – uruchomione lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/20-14/index.html)

Klatki na sekundę i czas klatki

FPS i czas klatki — dwie strony jednej wielkości, a clock.tick() tylko trzyma cykl, ale go nie przyspiesza.

FPS (frames per second, klatki na sekundę) – ile razy na sekundę gra udaje mu się przejść przez cały cykl Input → Update → Render i pokazać graczowi nową klatkę. Inne Jeśli system wyświetlania i wyjścia

są w stanie wyświetlić dodatkowe klatki, tym wyższe Płynność animacji może poprawić wizualną płynność ruchu – ale płynność już Naturalne fizyczne ograniczenie związane z ekranem i czasem, jaki zajmuje każdy kod Rama.

Czas klatki

Czas klatki (frame time) to liczba milisekund, jakie trwała pojedyncza pętla. FPS oraz Czas ramki to dwie strony tej samej wartości:

vremya_kadra.py

```
#frame_time trwa sekundy, fps klatki na sekundę
frame_time = 1 / 60          # 60 klatek na sekundę -> sekund
na klatkę
fps = 1 / frame_time         # powrót: sekundy na klatkę -> fps

print(frame_time)   # 0,016666... sekund, czyli około 16,7 ms
na klatkę
print(fps)          # 60.0
```

`clock.tick(60)` z sekcji 20.2 nie przyspiesza gry – wręcz przeciwnie, Jest sztucznie *przytrzymuje* pętlę, jeśli ta próbuje wykonać się szybciej niż 60 razy w sekundę, i nic nie może zrobić, jeśli kod jednej klatki sam zajmuje więcej czasu, niż przewidziany budżet. Rzeczywisty, zmierzony FPS można poznać poprzez `clock.get_fps()`, ale nie od razu: średnio wypada na ostatnie dziesięć Wyzwania `clock.tick()` szczerze wraca do dziesiątej klatki `0.0`. Zero w pierwszych klatkach nie jest zepsutym licznikiem, ale jeszcze nie Zebrane statystyki.

Brak limitu częstotliwości – cykl zużywa 100% CPU

Jeśli nic nie ogranicza tempa cykli, staje się to prostym busy loop i działa tak szybko, jak CPU – często tysiące klatek na sekundę, bez korzyści dla gracza, ale z pełnym wykorzystaniem rdzenia CPU i zwiększonym zużyciem baterii w laptopie lub telefonie (Rozdział 20.13). W cyklu gry tej książki rozwiązanie jest następujące `clock.tick(FPS)` na końcu każdej klatki inne silniki i gry mogą inaczej kontrolować tempo: za pomocą vsync, stałego timestep-schedulera lub własnego systemu czasowania silnika.

Praktyka: FPS i budżet

Zeskanowanie bezpośrednio w przeglądarce – nie wymaga instalacji Pygame

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/20-15/index.html)

Delta time: ruch niezależnie od FPS

Prędkość w "pikselach na klatkę" zależy od czyjegoś sprzętu. Prędkość w "pikselach na sekundę" pomnożona przez delta time już nie.

W mini-projekcie sekcji 20.5 piłka poruszała się tak: `x += dx` Każda klatka. Działa — ale tylko tak długo, jak FPS jest stałe. To podejście ma ukrytą Problem, który prędzej czy później ujawnia się na prawdziwych urządzeniach.

Problem: ruch ograniczony klatką

Jeśli `dx` to „piksele na klatkę” *drugi* bezpośrednio zależy od tego, ile klatek faktycznie zdąży przejść na sekundę, a nie od tego, o co prosiliście `clock.tick()` . `clock.tick(60)` z sekcji 20.5 określa jedynie *góra Limit* – 60 FPS, nie więcej – ale nie można nic zrobić, jeśli urządzenie Nie radzi sobie z tym również. Na laptopie gamingowym dewelopera gra konsekwentnie mieści wszystkie 60 FPS; Na słabym lub przegrzanym urządzeniu (Rozdział 20.13) kod jednej klatki może nie być a rzeczywista częstotliwość spada do, powiedzmy, 30 FPS. Z kodem "piksele na jedno" frame" oznacza, że kula takiego gracza porusza się dwa razy wolniej niż piłka wytwórcy, Z dokładnie tym samym kodem gra, która dobrze się czuje na tym samym urządzeniu, może Bądź zauważalnie wolniejszy lub szybszy na innym.

Rozwiązanie: delta time

Delta time (w skrócie `dt`) — czas, prawdziwy minęło od poprzedniej klatki w sekundach. `clock.tick(FPS)` nie Ogranicza liczbę klatek na klatkę, ale także *zwroty* liczbę milisekund, które minęły od tego czasu poprzednie wezwanie `clock.tick()` — czyli całkowity czas klatki, W tym czekanie, jeśli w ogóle występuje:

delta_time.py

```
clock = pygame.time.Clock()

while rabotaet:
    dt_ms = clock.tick(FPS)     # ile milisekund minęło od
ostatniej klatki
    dt = dt_ms / 1000           # przelicz na sekundy

    x += skorost_x * dt
#skorost_x teraz w pikselach na SEKUNDĘ, a nie na klatkę
```

Teraz `skorost_x` oznacza „piksele na sekundę”

Ważne jest, że `tick()` raporty dokładnie *pełne* czasie klatki, oraz Nie o to, jak długo czekało. Jeśli urządzenie nie poradzi sobie, nie musi wcale czekać — i potem `tick(60)` zwróci nie 16, ale, powiedzmy, 30 milisekund: dokładnie tyle, ile zajęło wykonanie klatki. Dlatego właśnie delta time ratuje na słabym urządzeniu: im wolniejsza klatka, tym więcej `dt` i im większy stopień. (Osobno, czas poświęcony wyłącznie użytecznej pracy, bez oczekiwania, jest dostępny przez `clock.get_rawtime()` .)

Zdezorientowane milisekundy i sekundy

`clock.tick(FPS)` zwraca czas w **milisekund**, nie w sekundy. Jeśli zapomnisz podzielić przez 1000, prędkość w kodzie zostanie przeszacowana dokładnie tysiąckrotnie — postać wylatuje z ekranu w jednej klatce. To jeden z najczęstszych błędów przy pierwszym przejściu na delta time, a objaw zawsze jest ten sam: ruch nagle staje się „teleportacją”

Ruch diagonalny szybszy niż ruch osiowy

Jeśli postać porusza się zarówno X, jak i Y z pełną prędkością i obiema prędkościami (`x += v * dt` oraz `y += v * dt` jednocześnie), a następnie faktycznie porusza się po przekątnej $\sqrt{2} \approx 1{,}41$ razy szybciej niż dokładnie jedna oś — oba składniki sumują się geometrycznie. Poprawnym rozwiązaniem jest normalizacja wektora kierunku (zmniejszenie jego długości do 1) przed pomnożeniem przez prędkość, tak aby całkowita prędkość pozostała taka sama w obu kierunkach.

Praktyka: ruch niezależny od FPS
Zeskanowanie bezpośrednio w przeglądarce – nie wymaga instalacji Pygame
Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/20-16/index.html)

Surface: powierzchnia do obrazowania i ramki

Ekran — to Surface, można tworzyć inne Surface w pamięci, a potem nakładać je na siebie przez blit().

`screen` dostajesz od `pygame.display.set_mode()` jest obiektem typu `pygame.Surface` : Prostokątny obszar pikseli, na którym możesz rysuj. Ekran nie jest jedynym Surface w grze: możesz tworzyć dodatkowe wypływa w pamięci i pracuje z nimi w ten sam sposób.

sozdanie_surface.py

```
# Dodatkowa powierzchnia 100x100, oddzielna od ekranu
sprajt = pygame.Surface((100, 100))
sprajt.fill((255, 100, 100))

# Nakładamy (blit) zawartość sprajt na ekran w punkcie (50, 50)
screen.blit(sprajt, (50, 50))
```

Rzeczywiste okno: różowy kwadrat 100×100 naniesiony na ciemnoniebieskie tło w punkcie (50, 50)

Okno rzeczywiste: Wynik screen.blit(sprajt, (50, 50)) - Drugi Surface wylosowany dokładnie w wyznaczonym punkcie.

`blit()` — główna operacja kompozycji w Pygame: „narysować zawartość jednej Surface na innej w wybranym punkcie”. Załadowany obraz (`pygame.image.load()` , sekcja 20.19) — to też Surface, i na ekran trafia tym samym wywołaniem `blit()` .

Przezroczystość

Surface mają dwa różne sposoby ustalania przejrzystości. Pierwszy to **colorkey**: jeden konkretny kolor jest deklarowany jako „przezroczysty” **per-pixel alpha**: y każdy piksel ma własny stopień przezroczystości,

co daje miękkie, przezroczyste krawędzie, ale wymaga obrazy w formacie alfa (zwykle PNG) i powierzchniach tworzonych flagą `pygame.SRCALPHA`.

Rzeczywiste okno: półprzezroczysta czerwona warstwa nad czarnym tłem wygląda na ciemno-bordową, a nie na czystą czerwień

Real Window: per-pixel alpha w akcji – ostateczny kolor to mieszanka tła i przezroczystej warstwy, a nie czysty kolor warstwy.

Ekran nie jest zalany – są „ślady”

Jeśli pominąć `screen.fill(...)` na początku klatki, nowe renderowanie po prostu nakłada się na poprzednie — poruszający się obiekt pozostawia za sobą ślad starych pozycji zamiast czystej klatki. Odwrotny błąd — wywołaj `fill()`, ale zapomnieć `pygame.display.flip()`: wtedy narysowana ramka pozostanie niewidoczna, a okno pokaże jednolity czarny (lub wcześniej wypełniony) kolor.

Prawdziwe okno: Łańcuch czerwonych kółek pozostaje na ekranie, ponieważ tło między ujęciami nigdy nie zostało ponownie zalane

Rzeczywiste okno: celowo odtworzony błąd „zapomniano screen.fill()” — każdy nowy okrąg pozostaje nad poprzednimi.

Przezroczystość „zniknęła” po konwersji obrazu

`pygame.image.load()` sam zapisuje kanał alfa przesłanego zdjęcia. Przezroczystość jest naprawdę tracona, jeśli dzwonisz `.convert()` zamiast `.convert_alpha()`: `.convert()` tworzy kopię w formacie pikselowym ekranu, ale bez per-pixel alpha, a krawędzie sprite'a będą rysowane z jednolitym kolorem tła zamiast miękkiego przejścia. Ten sam efekt dotyczy ręcznego utworzenia nowej powierzchni (`pygame.Surface(size)`) bez flagi `pygame.SRCALPHA`, gdy potrzebny jest dokładnie per-pixel alpha.

Praktyka: Surface, blit i przejrzystość

Pygame otwiera natywne okno Python – uruchomione lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/20-17/index.html)

Rect: pozycja, rozmiar i współrzędne

pygame.Rect utrzymuje pozycję i rozmiar razem, a oś Y na ekranie rośnie w dół — w przeciwieństwie do Turtle i matematyki na papierze.

Już wspomnieliśmy `pygame.Rect` w sekcji 20.3 jako sposób przechowywania pozycji i rozmiaru razem. Czas poznać go naprawdę — i przy okazji zrozumieć, jak w Pygame działa układ współrzędnych.

pygame.Rect

rect_osnovy.py

```
igrok_rect = pygame.Rect(100, 200, 40, 40)
# x, y, szerokość, wysokość

print(igrok_rect.x, igrok_rect.y)       # lewy górny róg
print(igrok_rect.center)                # środek — krotka (x,
y)
print(igrok_rect.right, igrok_rect.bottom)   # prawy i dolny
krawędź

igrok_rect.x += 5        # przesunąć cały prostokąt w prawo
igrok_rect.center = (300, 300)
# lub natychmiast ustawić nowy środek
```

`Rect` jest wygodna właśnie dlatego, że nie zmusza do ręcznego zapamiętywania cztery oddzielne zmienne (`x` , `y` , `ширина` , `высота`) i przeliczaj granice obiektu samodzielnie — za to odpowiadają gotowe atrybuty, takie jak `.right` , `.center` , `.bottom` .

System współrzędnych: świat i ekran

W Pygame punkt `(0, 0)` to lewy górny róg ekranu, a oś jest Y rośnie **na ziemię**, nie w górę. Warto o tym pamiętać, zwłaszcza po Turtle (Rozdziały 6–7, 12 i 19), gdzie oś Y rośnie w górę, jak w matematyce na papierze:

	Matematyka / Turtle	Pygame
Pochodzenie (0, 0)	Ekran centralny	Lewy górny róg
Kierunek wzrostu Y	Up	Down

	Matematyka / Turtle	Pygame
„Skok w górę”	y rośnie	y się zmniejsza

Stąd początkowe zamieszanie: kod „skoku” `y -= skorost_pryzhka`, nie `y +=` — Bo "wyżej na ekranie" oznacza "niższą wartość Y".

Współrzędne świata kontra ekran

W grach z przewijaniem (przewijany poziom, kamera podążająca za bohaterem) warto rozróżnić między *świat* współrzędne obiektu (jego położenie w całym świecie gry) oraz *na ekranie* współrzędne (gdzie dokładnie są narysowane na aktualnie widocznej części ekranu) są takie same tylko tak długo, jak kamera się nie porusza. W projektach tej książki kamera jest nieruchoma, więc współrzędne świata i ekranu są tym samym, ale w dużych grach to rozdzielenie jest standardową częścią architektury.

Praktyka: Rect i współrzędne

Zeskanowanie bezpośrednio w przeglądarce – nie wymaga instalacji Pygame

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/20-18/index.html)

Sprite'y, obrazy i zasoby gry

Gotowe obrazy są ładowane na jednej linii — ale ścieżka do nich powinna być liczona od lokalizacji kodu, a nie od miejsca jego uruchomienia.

Kształty rysowane `pygame.draw`, świetnie do Ale prawdziwe gry prawie zawsze wykorzystują gotowe obrazy — zasoby. Są ładowane w jednej linii:

zagruzka_izobrazheniya.py

```
igrok_kartinka = pygame.image.load("assets/
igrok.png").convert_alpha()
screen.blit(igrok_kartinka, (x, y))
```

`pygame.image.load()` już całkowicie ładuje obraz — `.convert_alpha()` po tym, jak technicznie nie jest to konieczne, ale dla obrazy z przezroczystością, zdecydowanie zaleca się wywołanie go natychmiast po inicjalizacji ekran: tworzy kopię obrazu w formacie pikselowym optymalnym do ponownego rysowania na tym ekranie i

zapisuje per-pixel alpha (Rozdział 20.17). Bez niego Obraz z przezroczystością zwykle nadal działa, ale można go wyraźnie wyrenderować wolniejsze przy częstych połączeniach `blit()`.

„Sprite”
W tym rozdziale **sprite** oznacza po prostu obraz obiektu gry: ładujemy go i rysujemy `blit()`. Pygame ma również klasę o tej samej nazwie `pygame.sprite.Sprite` z grupami `pygame.sprite.Group`, które mogą aktualizować i sprawdzać kolizje dla wielu obiektów jednocześnie, ale tutaj nie są potrzebne, a wrócimy do nich w rozdziale 21, gdzie zauważalny wzrost liczby obiektów.

Skąd gra wie, gdzie są pliki

Ścieżka `"assets/igrok.png"` — **względny**: to liczy się od bieżącego katalogu roboczego procesu (CWD), a nie od lokalizacji samego pliku z kodem. Dopóki uruchamiasz grę z tego samego folderu, w którym leży kod, różnicy nie widać — ale warto uruchomić ją z innego miejsca (na przykład przez skrót lub przez `python папка/igra.py` z sąsiedniego katalogu), oraz ładowania Spadki z `FileNotFoundError`. Niezawodne rozwiązanie – moduł `pathlib` (Rozdział 15): buduj ścieżkę od lokalizacji samego pliku za pomocą Kod, a nie na podstawie miejsca, z którego został wystrzelony.

nadezhnyj_put.py

```
from pathlib import Path

BASE_DIR = Path(__file__).resolve().parent
igrok_kartinka = pygame.image.load(BASE_DIR / "assets" /
"igrok.png").convert_alpha()
```

Droga do zasobu zależy od aktualnego katalogu startowego
"Wszystko działa dla mnie" i "FileNotFoundError na innym komputerze" to klasyczny objaw względnej ścieżki zbudowanej bez `Path(__file__)`. Szczególnie łatwo to przeoczyć przy pakowaniu do pliku wykonywalnego (rozdział 20.29), gdzie katalog roboczy procesu może w ogóle nie zgadzać się z folderem, z którego uruchomiono plik wykonywalny.

Organizacja folderu projektu

Dobrze ugruntowaną konwencją jest utrzymywanie kodu i zasobów oddzielnie, grupując zasoby według typu:

struktura_proekta.txt

```
moya_igra/
  igra.py
  assets/
    images/
      igrok.png
      vrag.png
    sounds/
      prygok.wav
```

Praktyka: ładowanie sprite'ów i niezawodne ścieżki
Pygame otwiera natywne okno Python – uruchomione lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/20-19/index.html)

Obsługa wejścia: zdarzenia i przytrzymanie klawiszy

Jednorazowa akcja — zdarzenie. Ciągła akcja — sprawdzanie stanu. Pomieszanie ich — częste źródło błędów w zarządzaniu.

W sekcji 20.4 widziałeś już oba sposoby testowania klawiatury. A dokładniej, o Do jakiego rodzaju działania każdy z nich się nadaje i co się dzieje, jeśli się pomylą.

Wydarzenia czy status?

Akcja
następuje raz
na każde
stuknięcie

(skok, pauza, strzał)	→	**`pygame.event.get() + KEYDOWN`**
Akcja trwa przez cały czas trwania klawisza (bieg w lewo/prawo)	→	**`pygame.key.get_pressed()`**

KEYDOWN ciągłego ruchu – postać drga

Wydarzenie `KEYDOWN` uruchamia się raz w momencie naciśnięcia, a nie podczas trzymania klawisza. Jeśli powiązasz z nim ruch postaci, porusza się dokładnie o jeden krok na naciśnięcie i zawiesza się, nawet jeśli gracz nadal trzyma klawisz – aby poruszać się nieprzerwanie, musisz to zrobić `pygame.key.get_pressed()`, sprawdzany w każdej klatce.

get_pressed() jednorazowej akcji – broń strzela seriami

Błąd wsteczny: jeśli sprawdzisz `klavishi[pygame.K_SPACE]` każdy kadr i strzelać przy każdym prawdziwym wartości, przy 60 FPS jedno przytrzymanie spacji na sekundę spowoduje około 60 strzałów — klawisz pozostaje wciśnięty cały czas. Dla jednorazowego działania przy wciśnięciu potrzebne jest właśnie zdarzenie `KEYDOWN` zamiast sprawdzać obecny stan.

Mysz także jest podwójna

Mouse ma ten sam wybór: zdarzenie `pygame.MOUSEBUTTONDOWN` — dla pojedynczego kliknięcia, a `pygame.mouse.get_pressed()` — dla weryfikacji, Czy przycisk jest teraz przytrzymywany (przydatne na przykład do rysowania linii podczas trzymania przycisku myszy jest trzymane). Aktualne współrzędne kursora są dostępne w dowolnym momencie za pomocą `pygame.mouse.get_pos()`.

Bridge do gamepada i ekranu dotykowego

Abstrakcja akcji gry z sekcji 20.11 (`"prygok"` zamiast konkretnego klucza) opłaca się szczególnie tutaj: pod nim mogą równie dobrze ukryć i `KEYDOWN` z odstępem, oraz przycisk pada `pygame.JOYBUTTONDOWN`), i stuknąć w wirtualny przycisk na Ekran dotykowy — Kod gry nie zmienia się ani o jedną linię.

> **Praktyka: Wydarzenia i status klawiatury**
> Pygame otwiera natywne okno Python – uruchomione lokalnie w VS Code, PyCharm lub Jupyter
>
> Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/20-20/index.html)

Kolizje i hitboxy

collideret() pod maską — cztery porównania liczb, a hitbox nie musi pasować do widocznego obrazu pixel w pixel.

Kolizja to moment, gdy dwa obiekty gry przecinają się w przestrzeni, a gra musi na to zareagować: piłka odbija się od ściany, postać podnosi monetę, pocisk pokonuje wroga. U `pygame.Rect` istnieje gotowa metoda na to:

collideret.py

```
if igrok_rect.collideret(vrag_rect):
    print(„Kolizja!”)
```

Jak to jest naprawdę zorganizowane

`collideret()` sprawdza przecięcie dwóch prostokątów, Wyrównany do osi (AABB, axis-aligned bounding box) jest metodą redukcją pod maską do czterech prostych porównań liczbowych:

aabb_vruchnuyu.py

```
def pryamougolniki_peresekayutsya(a, b):
    ax1, ay1, aw, ah = a
    bx1, by1, bw, bh = b
    ax2, ay2 = ax1 + aw, ay1 + ah
    bx2, by2 = bx1 + bw, by1 + bh
    return ax1 < bx2 and ax2 > bx1 and ay1 < by2 and ay2 > by1
```

Dwa prostokąty **nie** przecinają się, jeśli jest się całkowicie po lewej, po prawej, Wyższe lub niższe niż pozostałe, powyższa kontrola ma na celu tylko upewnić się, że żadna z tych czterech Warunki nie są spełnione.

Prawdziwe okno: nakładające się prostokąty – drugie jest zacienione na żółto, bo colliderect() wróciło True.

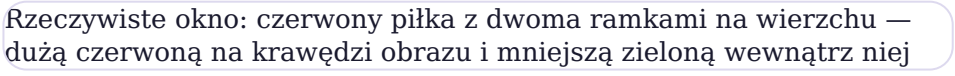
Faktyczne okno: prostokąty się nie przecinają — drugi pozostał czerwony.

Hitbox nie zawsze jest tym samym co rysowanie

"Hitbox" to prostokąt (lub inny kształt), używany specjalnie do sprawdzania kolizji, i nie musi odpowiadać rozmiarowi widocznego rysunku znaku w Jeden. Okrągła kula z przezroczystymi krawędziami zwykle ma kwadratowy obraz — jeśli liczyć kolizja na pełnym kwadracie obrazu, gracz zobaczy niesprawiedliwe "trafienia" chybił", gdzie nic nie dotykało oka.

Hitbox nie pasuje do obrazu – kolizje wydają się „niesprawiedliwe”

Częsta przyczyna: `Rect` pobiera się bezpośrednio z rozmiaru całego załadowanego obrazu, w tym przezroczystych pól wokół samej postaci. Rozwiązanie — zmniejszyć hitbox względem widocznego obrazu przez `rect.inflate(-20, -20)`: `inflate()` zmiany *ogólnie* wymiary prostokąta, zachowując środek, dlatego `-20` według szerokości i wysokości usuwa dokładnie 10 pikseli z każdej strony, a nie 20. Konkretne marginesy dobiera się wzrokowo dla każdego sprite'a.



Rzeczywiste okno: czerwona ramka — pełny rozmiar obrazu, zielona — zmniejszony hitbox, bliżej do rzeczywistych widocznych krawędzi piłki.

Praktyka: sprawdzanie przecięcia prostokątów
Zeskanowanie bezpośrednio w przeglądarce – nie wymaga instalacji Pygame
Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/20-21/index.html)

Podstawowa fizyka gry: prędkość, grawitacja i odbicie

Grawitacja jest prędkością skumulowaną, a nie współrzędną skoku natychmiastowego; odbicie i tarcie są dodawane tą samą logiką przez delta time.

Piłka z sekcji 20.5 zmienia kierunek na przeciwną stronę, gdy uderza w ścianę – to już jest Podstawowa fizyka. Dodajmy do tego jeszcze dwa pomysły, które najczęściej są potrzebne w grach arcade: grawitacja i tarcie.

Jednostki

W formułach w tej sekcji znajdują się trzy różne wartości i łatwo je pomylić. Pozycja (`x` , `y`) mierzone w pikselach. Prędkość — w pikselach na sekundę (px/s): dokładnie w tych jednostkach, jak już wiesz z rozdziału 20.16, ustalają `skorost_x` / `skorost_y` podczas ruchu przez delta time. Przyspieszenie — w pikselach na sekundę na sekundę, czyli (px/s)/s, co zwykle zapisuje się jako px/s²: to, jak szybkość sama się zmienia każdą sekundę. Grawitacja poniżej — właśnie przykład przyspieszenia.

Grawitacja jako akumulacja prędkości

Grawitacja — to nie „dodać do Y dużą liczbę raz”, a stałe, małe przyspieszenie, które w każdej klatce nieco zwiększa prędkość pionową. Sama pozycja zmienia się już za pomocą tej prędkości — z użyciem delta time z sekcji 20.16:

gravitaciya.py

```
GRAVITACIYA = 900     # pikseli na sekundę na sekundę
(akceleracja)

skorost_y += GRAVITACIYA * dt   # prędkość wzrasta z każdą
klatką
y += skorost_y * dt             # pozycja zmienia się w wyniku
skumulowanej prędkości
```

Właśnie dlatego w prawdziwym życiu (i w grach z uczciwą grawitacją) upadek zaczyna się powoli i przyspiesza — jest to bezpośredni skutek stałego narastania prędkości, a nie ustawienia „prędkość upadku” jedną liczbą.

Trzy prawdziwe okna obok siebie: pomarańczowa kula spada coraz szybciej od góry do dołu

Prawdziwe okno: trzy momenty tego samego upadku – odległość między pozycjami rośnie, ponieważ prędkość się kumuluje.

odbicie z utratą energii

Prawdziwa piłka nie zawsze odbija się z tej samej wysokości – część energii jest tracona, gdy uderzenie. W kodzie jest to jeszcze jeden dodatkowy łańcuch znaków po odwróceniu prędkości:

otskok_s_zatuhaniem.py

```
KOEFF_OTSKOKA = 0.8   # piłka zachowuje 80% prędkości po
odbiciu

if y + radius > VYSOTA:
    y = VYSOTA - radius
    skorost_y = -skorost_y * KOEFF_OTSKOKA
```

Brak mnożnika zaniku `KOEFF_OTSKOKA` piłka odbija się do tej samej wysokości w nieskończoność — wraz z nią wysokość odbicia naturalnie maleje od z każdym uderzeniem piłka prawie się zatrzymała. Taki czynnik tłumienia prędkości od uderzenia do jest dobrze ugruntowaną techniką w grach i nazywa się **restitution** (współczynnik) retracement, dosłownie „współczynnik odbicia” `1.0` odpowiada

idealizowanemu bezstratnemu odbiciu przy pierwotnej prędkości, a dowolna wartość między 0 a 1 — stopniowe odbicie z stopniową utratą energii.

Tarcie

Tarcie działa podobnie w przypadku ruchu poziomego – prędkość jest stopniowa zmniejsza się modulo (ale nie zmienia znaku), aż wychodzi do zera. Ważne jest, aby tarcie podobnie jak grawitacja, otrzymała przyspieszenie px/s² i była stosowana przez delta time — innymi słowy, Jeśli pomnożysz prędkość przez stały czynnik każdej klatki, hamowanie będzie zależało od FPS dokładnie taki sam jak ruch nieskorygowany w sekcji 20.16: na 120 FPS Mnożenie działa dwa razy częściej niż na 60 FPS, a obiekt zatrzyma się zauważalnie szybciej:

trenie.py

```
TRENIE = 300   # pikseli na sekundę w ciągu sekundy — przyspieszenie hamowania

if skorost_x > 0:
    skorost_x = max(0.0, skorost_x - TRENIE * dt)
elif skorost_x < 0:
    skorost_x = min(0.0, skorost_x + TRENIE * dt)
```

`max()` / `min()` nie jest tutaj podana hamowanie, by „ślizgać się”

To nie jest cały symulator fizyki

Prawdziwe silniki fizyczne (Box2D i podobne) rozwiązują znacznie bardziej ogólne zadania — obrót, masę, tarcie między dowolnymi kształtami, kolizje wielu ciał naraz. Wzory z tego działu — uproszczony, ale całkowicie działający model właśnie dla gier arcade, takich jak te, które tworzysz w tym rozdziale, a ich zrozumienie — dobra baza do przejścia do prawdziwego silnika fizycznego później, jeśli zajdzie taka potrzeba.

Praktyka: Grawitacja i rozpad odbijają się

Zeskanowanie bezpośrednio w przeglądarce – nie wymaga instalacji Pygame

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/20-22/index.html)

Animacja sprite'ów: Klatki i lista sprite'ów

Animacja — to szybka zmiana klatek jednego obrazu według własnego zegara, niezgodnego z częstotliwością klatek gry.

Do tej pory postacie były statycznymi figurami lub jednym nieruchomym obrazkiem. Animacja — to szybka zmiana kilku klatek rysunku, tworząca iluzję ruchu, — ta sama zasada, co w kreskówce lub flipbooku.

Lista sprite'ów

Lista sprite'ów (sprite sheet) to pojedynczy obraz, w którym klatki animacji są ułożone w rzędzie. Aby pokazać pożądaną klatkę, po zdjęciu wycina się prostokątny obszar `subsurface()`:

kadr_iz_lista.py

```
SHIRINA_KADRA, VYSOTA_KADRA = 32, 32
sprajt_list = pygame.image.load("assets/
hodba.png").convert_alpha()

def poluchit_kadr(nomer):
    x = nomer * SHIRINA_KADRA
    return sprajt_list.subsurface((x, 0, SHIRINA_KADRA,
VYSOTA_KADRA))
```

Timer animacji

Klatki animacji nie powinny zmieniać każdej klatki w pętli gry – jest za szybka (przy 60 FPS jest to 60 zmian wzoru na sekundę). Zamiast tego czas gromadzi się w urządzeniu pamięci masowej (baterii), oraz Klatka animacji przełącza się na własny, wolniejszy timer:

tajmer_animacii.py

```
INTERVAL_KADRA = 0.12   # sekund na klatkę animacji
tekushij_kadr = 0
vremya_animacii = 0.0

vremya_animacii += dt
while vremya_animacii >= INTERVAL_KADRA:
    vremya_animacii -= INTERVAL_KADRA
    tekushij_kadr = (tekushij_kadr + 1) % KOLICHESTVO_KADROV
```

`% KOLICHESTVO_KADROV` (pozostałość podziału, rozdział 4) Pętla licznika klatek: Po ostatniej klatce animacja zaczyna się od zera. Ważne jest właśnie to, co jest tutaj ważne `while`, nie `if`: jeśli jedna klatka gry trwa dłużej niż kilka odstępów naraz z powodu FPS drawdownu animacje, `while` przełączy klatkę animacji potrzebną liczbę razy z rzędu i zapisze resztę czasu w `vremya_animacii` na następny przejście. `if` zamiast tego przesuwa animację tylko o jeden krok do przodu i po cichu stracił dodatkowy zgromadzony czas.

Cztery prawdziwe okna obok siebie: noga niebieskiej postaci przesuwa się w prawo z klatki na ramę, mierząc stopień

Rzeczywiste okno: cztery klatki tej samej prostej animacji chodzenia, ręcznie narysowane dla tego przykładu — ta sama zasada co w prawdziwym arkuszu sprite'ów.

Aktualizacja po renderowaniu – animacja opóźnia się o klatkę

Jeśli kod się zmienia `tekushij_kadr`, warto **po** rysowanie bieżącej klatki, gracz na ekranie zawsze widzi poprzedni stan animacji, a nie tylko właśnie obliczony. Kolejność Update → Render z rozdziału 20.14 istnieje właśnie po to, aby uniknąć takiej jednoklatkowej niesynchronizacji.

Praktyka: lista sprajtów i timer animacji

Pygame otwiera natywne okno Python – uruchomione lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/20-23/index.html)

Efekty dźwiękowe i muzyka

Krótkie efekty i muzyka tła w Pygame to dwa różne instrumenty z inną mechaniką odtwarzania wewnętrznego.

Za dźwięk w Pygame odpowiada osobny podmoduł `pygame.mixer`. Podobnie jak reszta Pygame, ma własną obowiązkową inicjałację — zazwyczaj już uwzględniony w ogólnym `pygame.init()`, ale z problemami z dźwiękiem Warto to jasno podkreślić `pygame.mixer.init()`.

Krótkie dźwięki i muzyka — to nie to samo

Pygame mają dwa różne instrumenty dźwiękowe i nie są one wymienne:

	pygame.mixer.Sound	pygame.mixer.music
Dlaczego	Krótkie efekty: skok, strzał, moneta	Muzyka tła
Ładowanie	Całkowicie zapamiętana od razu	Streamowane, nie w pełni załadowane
Dźwięki jednoczesne	Kilka naraz, na różnych kanałach	Jeden strumień muzyki naraz
Typowe wyzwanie	sound.play()	pygame.mixer.music.play(loops=-1)

zvuk_i_muzyka.py

```
zvuk_pryzhka = pygame.mixer.Sound("assets/sounds/prygok.wav")

pygame.mixer.music.load("assets/sounds/fon.mp3")
pygame.mixer.music.play(loops=-1)   # -1 oznacza „pętlę bez końca”

# wewnątrz pętli gry, w momencie skoku:
zvuk_pryzhka.play()
```

Dźwięk ładuje się co klatkę – zauważalne zacięcia

`pygame.mixer.Sound(...)` odczytuje plik z dysku, co jest zauważalnie wolniejsze niż odtwarzanie już załadowanego dźwięku. Wszystkie dźwięki muszą zostać załadowane raz, przed rozpoczęciem pętli gry (podobnie jak obrazy z sekcji 20.19), i zapisane w zmiennych lub słowniku, a w pętli tylko wywołane `.play()` dla już załadowanego obiektu. Ładowanie zasobu w pętli jest jedną z najczęstszych przyczyn niewytłumaczalnych spadków klatek w pierwszych projektach na Pygame.

Praktyka: Efekty dźwiękowe i muzyka w tle

Pygame otwiera natywne okno Python – uruchomione lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/20-24/index.html)

Stany gry

Menu, rozgrywka, pauza i game over — to różne stany gry i kod musi wyraźnie wiedzieć, w którym z nich jest w danej chwili.

Już teraz Twoje projekty mają przynajmniej jeden niejawny stan — «gra trwa». W praktyce prawie każda gra przechodzi przez kilka wyraźnie rozróżnialnych stanów i w każdym z nich zachowuje się inaczej.

Typowe stany gry

MENU

```
Pokazuje nagłówek i przycisk „Graj"
Nie przesuwa żadnych obiektów gry
Czeka, aż zostanie nacisnięty lub naciśnięty
klawisz
```

GRA TRWA DALEJ

```
Aktualizuje pozycje, sprawdza kolizje
```

Reaguje na sterowanie graczem
Podstawowy stan gry

PAUZA

Nie aktualizuje pozycji funkcji
Kontynuuje renderowanie ostatniej klatki
Czeka na polecenie wznowienia

GAME OVER

Pokazuje końcowy wynik
Nie akceptuje sterowania
Proponuje zacząć od nowa

Państwo jako wyliczenie

Znany z rozdziału 19 sposób — przechowywać bieżący stan jako wartość `enum.Enum` (Rozdział 18), nie jako kilku niezależnych Boolean flagi takie jak `na_pauze` oraz `igra_okonchena` oddzielnie. Enum nie może znajdować się w dwóch stanach jednocześnie — u poszczególnych flag boolean taka pomyłka jest całkowicie możliwa przypadkowo:

sostoyaniya_enum.py

```
from enum import Enum, auto

class SostoyanieIgry(Enum):
    MENU = auto()
    IGRA = auto()
    PAUZA = auto()
    GAME_OVER = auto()

tekushee_sostoyanie = SostoyanieIgry.MENU
```

Przejścia między stanami

Stany tej mini-gry tworzą nie linię prostą, lecz graf: z `PAUZA` gra wraca z powrotem do `IGRA` i nie idzie dalej do przodu, podobnie jak powrót z `GAME_OVER` w `MENU`. Bezpieczniej jest narysować wyraźne lista dozwolone przejścia Tabela „od → do”

Spoza stanu	W stanie	Kiedy
MENU	IGRA	naciśnięto „Graj”
IGRA	PAUZA	nacisnął pauzę
IGRA	GAME_OVER	kolizja / porażka
PAUZA	IGRA	ponownie nacisnąć pauzę (odpauzować)
GAME_OVER	MENU	kliknął „Jeszcze raz”

Ważne, że nie każdy ruch jest dozwolony: z `MENU` nie możesz bezpośredni dostęp do `GAME_OVER`, omijając samą grę, i z `PAUZA` nie można trafić do `GAME_OVER` bezpośrednio — najpierw gra musi wrócić do `IGRA`. Kod, który obsługuje wprowadzanie danych i aktualizacje, musi wcześniej wyraźnie sprawdzać aktualny stan Wykonywanie czynności dozwolonych tylko w nim jest dokładnie takie samo jak w rozdziale 19 Potrzebne było oczyszczenie `GameStatus` dla węża.

Cztery rzeczywiste okna obok siebie: menu z podpisem i podpowieścią, rozgrywka z piłką i punktem, piłka z żółtą nakładką PAUSE na środku oraz GAME OVER ekran z końcowym wynikiem

Real Window: Cztery stany mini-gry – Menu, Gra, Pauza game over. To cztery oddzielne strzały, a nie jeden ciągły przebieg, więc liczenie nie powtarza się nawzajem.

Praktyka: stany gry i przejścia
Zeskanowanie bezpośrednio w przeglądarce – nie wymaga instalacji Pygame
Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/20-25/index.html)

Struktura Małych Gier: Klasa Game

Trzy fazy cyklu gry stają się trzema metodami tej samej klasy — tymi samymi ramami co ostateczny szkic tego rozdziału i następnego.

Do tej pory cały kod żył w zmiennych globalnych i jednej płaskiej pętli — normalnie dla edukacyjnego mini-projektu, ale niewygodne, gdy gra zaczyna mieć stany (rozdział 20.25), kilka obiektów i jakąkolwiek pauzę. Zbierzemy wszystko, czego nauczyłeś się w tym rozdziale, w jedną klasę — ten sam szkielet architektoniczny, na którym oparty jest finalny projekt rozdziału 20.33 i kosmicznego strzelanka rozdziału 21.

Każda metoda odpowiada za dokładnie jedną fazę pętli gry w sekcji 20.14 i nie więcej.

To szkic szkieletu, a nie pełny plik: pomija import `pygame`, definicja `SHIRINA` / `VYSOTA`, Klasa `SostoyanieIgry` z sekcji 20.25 i Treści Metody `toggle_pause()` — wszystko to w prawdziwym projekcie, oczywiście, powinno być zdefiniowany. Kompletnym, naprawdę możliwym do uruchomienia przykładem takiej klasy jest projekt końcowy Rozdział 20.33.

fragment_class_game.py

```
class Game:
    def __init__(self):
        pygame.init()
        self.screen = pygame.display.set_mode((SHIRINA,
VYSOTA))
```

```
        self.clock = pygame.time.Clock()
        self.state = SostoyanieIgry.MENU
        self.running = True

    def handle_events(self):
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                self.running = False
            elif event.type == pygame.KEYDOWN and event.key ==
pygame.K_p:
                self.toggle_pause()

    def update(self, dt):
        if self.state is not SostoyanieIgry.IGRA:
            return   # zatrzymaj się albo nie ruszaj niczego w
menu
        # tutaj — aktualizacja pozycji, kolizji, wynik

    def render(self):
        self.screen.fill((20, 20, 40))
        # tutaj jest renderowanie wszystkich obiektów w
aktualnym stanie
        pygame.display.flip()

    def run(self):
        while self.running:
            dt = self.clock.tick(60) / 1000
            self.handle_events()
            self.update(dt)
            self.render()
        pygame.quit()
```

`run()` — jedyne miejsce, gdzie widać cały cykl w całości, i powtarza dosłownie trzy fazy z rozdziału 20.14: `handle_events()` to Input, `update(dt)` tego Update, `render()` to Render. Tak właśnie kończy się `bouncing_ball.py` (rozdział 20.33) — i w tym samym stylu w rozdziale 21 napisano kosmiczny shooter.

Pauza nie zatrzymuje sterowania

Jeśli czek `self.state is not SostoyanieIgry.IGRA` włożone tylko `render()`, nie w `update()` ekran wizualnie zawiesza się poprawnie podczas pauzy — ale obiekty będą się dalej poruszać w pamięci, a po usunięciu pauzy gra „teleportuje się” `update()`, nie tylko w renderingu.

Praktyka: Składanie klasy Game
Pygame otwiera natywne okno Python – uruchomione lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/20-26/index.html)

Wydajność: FPS i budżet

Każda klatka ma trudny budżet czasu w milisekundach i prawie zawsze bardziej przydatne jest mierzenie niż zgadywanie, co jest wolne.

Rozdział 20.15 już wykazała, że 60 FPS oznacza budżet około 16,7 milisekundy na klatkę Cały proces, w tym przetwarzanie wejściowe, aktualizację i renderowanie. Jeśli kod pojedynczej klatki zajmuje czas Więcej niż ten budżet, personel jest tracony, bez względu na to, co zostanie ujawnione w `clock.tick()`.

byudzhet_kadra.py

```
# Budżet klatek dla różnych FPS
for fps in (30, 60, 120):
    byudzhet_ms = 1000 / fps
    print(f"{{fps}} FPS -> {{byudzhet_ms:.2f}} ms na klatkę")
```

Gdzie zwykle trafia budżet na personel?

Częsta przyczyna spadku	Dlaczego to jest wolne	Co zrobić zamiast tego
Pobieranie pliku w pętli	Odczyty dysku są tysiące razy wolniejsze niż operacje z pamięcią	Załaduj wszystkie zasoby raz przed pętlą (Sekcje 20.19, 20.24)
Stwórz nową Surface każdej klatce	Przydział pamięci pod obraz — to nie jest darmowa operacja	Twórz powierzchnie raz i używaj ponownie
Renderowanie obiektów poza ekranem	Każde wywołanie blit() wykonuje pracę nad kompozycją klatki niezależnie od tego, czy jej wynik jest widoczny	Pomijaj rysowanie obiektów, których nie widać

Częsta przyczyna spadku	Dlaczego to jest wolne	Co zrobić zamiast tego
Sprawdzanie kolizji „każdy z każdym”	Liczba checków rośnie kwadratowo wraz z liczbą obiektów	Zmniejszenie liczby testów (np. podzielenie ekranu na strefy)

Jak mierzyć, a nie zgadywać

`clock.get_fps()` z sekcji 20.15 — najprostszy sposób aby zobaczyć rzeczywistą wydajność w trakcie gry. Dla dokładniejszej analizy, która dokładnie część kodu jest wolna, w Python jest wbudowany profiler `cProfile` — pokazuje, ile czasu zostało spędzone w każdej funkcji, a nie tylko końcowy FPS.

Najpierw zmierz, potem optymalizuj
Pierwszą reakcją na FPS spadku poziomu jest przepisanie „na oko” (`get_fps()` lub `cProfile`), dokąd dokładnie zmierza czas: intuicja dotycząca „wolnego kodu”

Praktyka: Obliczanie budżetu personelu
Zeskanowanie bezpośrednio w przeglądarce – nie wymaga instalacji Pygame

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/20-27/index.html)

Jak debugować gry

Kompendium według objawu: większość błędów Pygame-projektów nie powoduje wyjątek — gra po prostu zachowuje się nie tak, jak oczekiwano.

W trakcie rozdziału natknąłeś się już na ponad tuzin typowych błędów Pygame-projects. jeden lub dwa dla każdego nowego tematu. Tutaj są zebrane w jednej tabeli – jako książka referencyjna, co jest wygodne do powrotu, gdy coś w swoim projekcie zachowuje się niewytłumaczalnie.

Objaw	Zwykła przyczyna	Gdzie dowiedzieć się więcej
Okno oznaczone jako „Nie odpowiada”	pygame.event.get() nie jest nazywana w każdej klatce	20.14

Objaw	Zwykła przyczyna	Gdzie dowiedzieć się więcej
Pygame moduły od początku nie działają prawidłowo	Zapomniane pygame.init()	20.14
Procesor obciążony w 100% bez powodu	Zapomniano clock.tick(FPS)	20.15
Postać teleportuje się na cały ekran	dt nie przelicza się z milisekund na sekundy	20.16
Ukośnie szybciej niż prosta	Wektor prędkości nie jest znormalizowany	20.16
„Ślady”	Zapomniane screen.fill() na początku każdej klatki	20.17
Okno pokazuje tylko ekran	Zapomniano pygame.display.flip()	20.17
Przezroczystość obrazu zniknęła	Przywołany przez . convert() zamiast . convert_alpha(), czyli Surface stworzony bez pygame.SRCALPHA	20.17
FileNotFoundError na cudzym komputerze	Droga do aktywa nie jest budowana przez pathlib	20.19
Postać drga zamiast płynnie	KEYDOWN stosowane do działania ciągłego	20.20
Seria ognia z broni wybucha jednym stuknięciem	get_pressed() używane do jednorazowej akcji	20.20
Zderzenia wydają się niesprawiedliwe	Hitbox nie jest zredukowany z obrazu	20.21
Animacja wizualnie opóźnia się o jedną klatkę	Aktualizacja klatki animacji następuje po rysowaniu	20.23

Objaw	Zwykła przyczyna	Gdzie dowiedzieć się więcej
Zauważalne spowolnienia przy każdym dźwięku	Sound() jest odtwarzana w każdej klatce, a nie raz	20.24
Gra „teleportuje się” po zdjęciu pauzy	Pauza jest sprawdzana tylko w render(), nie w update()	20.26

Ogólna metoda debugowania gier

Prawie wszystkie robaki z tej tabeli mają jedną wspólną cechę: nie porzucają wyjątek i nie porzucają Pokazuj tradycyjny komunikat o błędzie – gra po prostu nie działa zgodnie z oczekiwaniami. Dla tej klasy błędów szczególnie przydatne są następujące elementy:

otladka_pechatyu.py

```
#Temporary print() najszybszy sposób, by zobaczyć rzeczywiste wartości
print(f"dt={{dt:.4f}} x={{x:.1f}} skorost_x={{skorost_x:.1f}}")

# Tymczasowo renderuj hitbox na szczycie sprite'a – zobacz jego granice oczami
pygame.draw.rect(screen, (255, 0, 0), hitbox, width=2)
```

Debugowanie rysowania — nie zapomnij usunąć

Tymczasowe `pygame.draw.rect(..., width=2)` hitboxa to jeden z najskuteczniejszych trików debugowania kolizji, ale łatwo o nim zapomnieć w gotowym projekcie. Przydatnym nawykiem jest trzymanie takich insertów pod jednym wspólnym flagiem, jak `OTLADKA = False` włączać i wyłączać je jednocześnie, zamiast przeglądać cały plik.

Praktyka: diagnoza na podstawie objawu

Zeskanowanie bezpośrednio w przeglądarce – nie wymaga instalacji Pygame

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/20-28/index.html)

Składanie wersji stołowej gry

PyInstaller buduje grę jako samodzielny plik wykonywalny, ale tylko dla systemu operacyjnego, na którym działa.

Gracz, dla którego tworzysz grę, prawie nigdy nie zainstaluje Python i `pip install pygame-ce` tylko po to, by ją zacząć. Gra musi być pakuj go do pliku wykonywalnego, który działa samodzielnie.

PyInstaller

Jednym z najczęściej używanych narzędzi na komputerze stacjonarnym jest `PyInstaller`: analizuje twój program, znajduje wszystko, co potrzebne (w tym sam Pygame) i składa pojedynczy plik wykonywalny wraz z interpreterem Python w środku – można grać w taką grę bez instalowania Python na komputerze gracza.

pyinstaller.txt

```
pip install pyinstaller
pyinstaller --onefile --add-data "assets:assets" igra.py
```

`--onefile` zbiera wszystko w jeden plik zamiast folderu z zależnościami. `--add-data` wyraźnie określa, aby dołączyć ten folder `assets` w zgromadzeniu – bez tej flagi PyInstaller tylko się zapakuje kod, a obrazy i dźwięki pozostaną poza gotowym plikiem.

Colon w --add-data

W obecnych wersjach PyInstaller źródło i miejsce docelowe są oddzielone dwukropkiem (`"assets:assets"`) na wszystkich systemach operacyjnych, w tym Windows. W wersjach sprzed 6.0 separator był specyficzny dla systemu i Windows musiał być zapisywany średnikiem: `"assets;assets"` . Jeśli pracujesz ze starą, przypiętą wersją i build nie znajduje zasobów, najpierw sprawdź ten symbol.

Platforma poprzeczna – z pewnym zastrzeżeniem

PyInstaller nie kompiluje krzyżowo
PyInstaller tworzy plik wykonywalny dla systemu operacyjnego, na którym działa – kompilacja dla Windows odbywa się na Windows, dla macOS – na macOS, dla Linux – na Linux. Nie wie, jak je zbudować `.exe` Windows podczas pracy na Linux i odwrotnie. Aby naprawdę wieloplatformowe wydanie było potrzebne dostęp do każdego z docelowych systemów operacyjnych, bezpośrednio lub przez maszynę wirtualną lub usługę chmurową.

Ukryte importy

Pygame zaprojektowany tak, że niektóre moduły są połączone w nietypowy sposób dla Python W ten sposób czasem PyInstaller nie znajduje ich automatycznie i pomija je w budowli. Jeśli gotowy plik wykonywalny zawiesza się z `ModuleNotFoundError`, Chociaż gra działa poprawnie z kodu źródłowego, zwykle pomaga jednoznaczne określenie takiego modułu via `--hidden-import`.

Sprawdź plik zbudowany, nie tylko kod źródłowy
Uruchamianie gry przy starcie przez `python igra.py` i działający build wykonywalny nie gwarantują, że to to samo: różnica w ścieżkach zasobów (sekcja 20.19) i ukrytych importach jest widoczna już na etapie budowania. Zawsze warto uruchomić gotowy plik raz na czystym komputerze (lub przynajmniej z innego folderu), zanim uznamy go za gotowy do wydania.

Uruchom Pygame- grę w przeglądarce

pygbag — aktywnie wspierane narzędzie społeczności do tworzenia Pygame- gier w WebAssembly, nieoficjalna część pygame-ce.

Rozdział 20.10 już wspomniała: gra internetowa na Pygame trafia do przeglądarki przez WebAssembly. Narzędzie, które w praktyce tworzy ten zespół, nazywa się **pygbag** jest aktywnie wspieranym projektem społecznościowym, a nie oficjalną częścią pygame-ce.

pygbag.txt

```
pip install pygbag
pygbag moya_igra/
```

Polecenie uruchamia lokalny serwer WWW i otwiera grę bezpośrednio w przeglądarce — wygodne do sprawdzenia przed publikacją na stałym hostingu. Zwróć uwagę: argument to **folder** projektu, a nie osobny plik.

Plik z pętlą gry musi nazywać się main.py
pygbag przeszukuje w folderze projektu wyłącznie dla `main.py` — ta nazwa jest wbudowana w zespół i nie można jej przemianować. Trudne jest to, że przy innej nazwie (`igra.py` , `game.py`) budowa się zakończy *pomyślnie* i bez żadnego ostrzeżenia, ale w przeglądarce gra po prostu się nie uruchomi: loader będzie szukał `main.py` tego nie ma w archiwum zbudowanym. Jeśli strona otwiera się po kompilacji, ale pozostaje pusta, pierwszą rzeczą jest sprawdzenie nazwy pliku.

pygbag jest narzędziem społecznościowym, a nie częścią pygame-ce
Tak jak w przypadku mobilnych kompilacji (sekcja 20.13, sekcja 20.31), ważne jest rozumienie granicy odpowiedzialności: pygbag jest rozwijany i utrzymywany osobno od pygame-ce.. Jest aktywnie rozwijany i w praktyce dobrze radzi sobie z typowymi 2D- grami na Pygame, ale przy aktualizacji wersji Pygame należy ponownie sprawdzić kompatybilność z aktualną wersją pygbag, a nie zakładać, że jest gwarantowana na zawsze.

Co działa inaczej w sieci

Wersja przeglądarkowa gry nie działa dokładnie tak jak zwykłe uruchomienie na komputerze stacjonarnym — kod wykonywany jest wewnątrz piaskownicy przeglądarki, z której:

	Normalne uruchamianie pulpitu	Zgromadzenie do pygbag
Dostęp do systemu plików	Pełne	Ograniczone do piaskownicy przeglądarki

	Normalne uruchamianie pulpitu	Zgromadzenie do pygbag
Główna Pętla	while True w twoim kodzie	Asynchroniczny, z await asyncio.sleep(0) na każdej klatce
Instalacja	wymaga Python i pip install	Nie jest wymagane – otwórz link i zagraj
Występy	Natywna prędkość CPU	Może różnić się od desktopu – warto mierzyć na docelowym urządzeniu

Główna praktyczna różnica w kodzie — pętla gry musi stać się asynchroniczna (`async def main()` z `await asyncio.sleep(0)` na końcu każdej klatki), aby nie blokować wątku przeglądarki: pygbag oczekuje dokładnie taka struktura programu. Ważne jest, aby `await` tutaj *dodane W pobliżu* z `clock.tick()` i nie zastępuje go w przykładach pygbag limit liczby klatek pozostaje niezmienny, a łańcuch znaków `await` następuje zaraz po tym.

Jak wolniejsza jest budowa przeglądarki niż na komputerze, nie da się powiedzieć z góry: zależy od przeglądarka, urządzenie gracza oraz sama gra. Wydajność powinna być mierzona na podstawie celu i nie ustalać określonego procentu strat z góry.

Co naprawdę jest możliwe w Android, iOS i konsolach

Android jest dostępny wyłącznie poprzez eksperymentalną integrację społecznościową, nie jest akceptowany do głównego repozytorium python-for-android. iOS – przez narzędzie, które sam autor nazywa eksperymentalnym.

Rozdział 20.13 już udzieliła odpowiedzi w ogólnych kategoriach: Pygame nie jest publikowana na platformach mobilnych jako zespół. Tutaj możesz przeczytać więcej o tym, jak obie rzeczywiste ścieżki wyglądają w praktyce, a jak nie tylko że istnieją.

Android – python-for-android, Buildozer i eksperymentalne przepisy

`python-for-android` — to infrastruktura do pakowania Python-aplikacji pod Android; na niej zwykle używa się `Buildozer` — wyższej klasy narzędzie, które pobiera ustawienia z prostego pliku konfiguracyjnego i uruchamia kompilację `.apk` za ciebie, podłączając potrzebne przepisy. W momencie sprawdzania w głównym repozytorium `python-for-android` nie ma przyjętego oficjalnego przepisu pygame-ce — są tylko otwarte community pull request, dodające taki przepis, i przepisy użytkowników, które można podłączyć ręcznie. Dlatego zwykła instalacja Buildozer sama w sobie nie daje gotowego wsparcia pygame-ce:, zbudowanie APK z pygame-ce jest możliwe, podłączając jeden z tych przepisów ręcznie (na przykład przez fork lub lokalny przepis custom-), ale jest to eksperymentalna, a nie standardowa lub gwarantowanie wspierana integracja — kompatybilność należy sprawdzać ponownie w momencie kompilacji i obowiązkowo testować na rzeczywistym urządzeniu. Proces w każdym przypadku jest zauważalnie dłuższy niż pakowanie pod desktop (sekcja 20.29): potrzebny jest zestaw narzędzi Android (Android SDK/NDK)), kompilacja zajmuje znacznie więcej czasu, a końcowy rozmiar pliku jest wyraźnie większy niż sam kod gry — do środka APK zapakowane jest całe środowisko uruchomieniowe Python.

iOS – poprzez pygame-ios, narzędzie eksperymentalne

> **pygame-ios jest projektem pilotażowym, który nie jest gotowy do publikacji**
> Dla iOS istnieje narzędzie społecznościowe pygame-ios — szablon projektu, który osadza grę Pygame- w standardowej Xcode. Własna dokumentacja wyraźnie opisuje ją jako niedoprzetestowaną i niegotową do produkcji: autor narzędzia informuje, że wersja do prawdziwej publikacji w App Store nie była osobiście przez niego testowana. To nie jest „jedna z dwóch równych ścieżek”

Oprócz statusu samego narzędzia, publikacja na iOS wymaga komputerowego Mac (Xcode jest dostępny tylko na macOS), buildach i podpisach poprzez sam Xcode i, jak w każdej iOS- aplikacji, Płatne konto deweloperskie Apple. Narzędzie nie obsługuje każdej wersji pygame-ce — Specyficzna zgodność powinna być sprawdzana w dokumentacji narzędzi w momencie montażu, a nie Polegaj na pamięci z poprzedniego projektu.

	Android	iOS
Tool	python-for-android + Buildozer (społeczność)	pygame-ios (społeczność, eksperymentalny)
Oficjalne wsparcie pygame-ce	Nie — przepis nie został przyjęty do głównego repozytorium	Tak
Gotowy do publikacji	Eksperymentalne składanie za pomocą community pull request lub custom-recept; wymaga sprawdzenia aktualnego statusu receptury i narzędzi	Autor opisuje jako niegotowy do produkcji
Odpowiedni system montażu	Windows, macOS czy Linux	Tylko macOS (potrzeba Xcode)
Format końcowy	.apk / .aab	Project Xcode →. ipa
Publikacja	Google Play	App Store potrzebujesz konta deweloperskiego Apple

Konsole

Dla konsol do gier (sekcja 20.9) zauważalny, wspierany przez społeczność łańcuch pośredni taki jak `python-for-android` nie istnieje. Publikacja na konsoli pozostaje oddzielnym, zamkniętym procesem przez oficjalny SDK producenta — nie jest to związane z wyborem Pygame konkretnie, tak samo zorganizowana jest publikacja gier napisanych w dowolnym narzędziu.

Zaplanuj swój mobilny build z wyprzedzeniem, a nie na końcu
Obie mobilne ścieżki dodają rzeczywistą pracę inżynierską na gotowej grze desktopowej — to nie jest finalny eksport jednym kliknięciem, lecz oddzielny etap projektu z własnymi wymaganiami dotyczącymi środowiska budowy, przy czym gotowość tych ścieżek znacznie różni się między Android i iOS.. Decyzję o wydaniu gry również na telefony należy podjąć na początku pracy nad grą (biorąc pod uwagę ograniczenia rozdziału 20.13 — bateria, pamięć, cykl życia, wprowadzanie), a nie próbować dopasować go po tym, jak cały kod został już napisany pod założenia desktopowe.

Zawody w tworzeniu gier i portfolio

tworzenie gier to rodzina powiązanych zawodów, a ukończone projekty w tej dziedzinie są praktycznym sposobem na pokazanie prawdziwych umiejętności.

Tworzenie gier to nie jeden zawód, lecz cała rodzina powiązanych specjalności (sekcja 20.6 już wymieniło główne role). Jeśli spodobał Ci się ten mini-projekt, oto kilka z nich obszary, w których możesz się rozwijać.

Kierunki w tworzeniu gier

GAMEPLAY-PROGRAMISTA

Implementuje mechanikę gry

Zacznij: Ten projekt → własne, małe gry → bardziej złożonym silniku

PROJEKTANT GIER

Projektuje reguły i balans

Zacząć: analizować, co sprawia, że istniejące gry są wciągające

ARTYSTA TECHNICZNY

Pomost między grafiką a kodem

Rozpoczęcie: shadery, sprite'y, animacja - Rozdział 20.23

INŻYNIER SILNIKA

Narzędzia budujące dla innych programistów
Rozpoczęcie: Zrozumienie, jak Pygame działa od środka

QA / TESTER

Wykrywa błędy przed wydaniem
Zacząć: rozdział 20.28 — debugowanie systemu w cudzym kodzie również rozwija tę umiejętność

Dlaczego twórca gier potrzebuje portfolio

Dla wielu junior- stanowisk w tworzeniu gier ukończone projekty są praktycznym dowodem Prawdziwa umiejętność: pracodawca może odkryć i zobaczyć, co zbudowałeś i jak to wygląda pracuje, a nie tylko czytam lista z kursów. Jednocześnie wymagania są bardzo silne Różni się między firmami i stanowiskami: gdzieś formalna edukacja jest obowiązkowa, gdzieś Doświadczenie jest ważniejsze, a portfolio nie zawsze zastępuje dyplom — to kolejny sposób na pokazanie praktyczny poziom i nie gwarantująca przepustka do zawodu. Odbicia piłki i finał Projekt tego rozdziału (Rozdział 20.33) to mały, ale realny pierwszy punkt takiego portfolio: Doprowadzony do końca, a nie porzucony w środku.

Kolejny krok jest już w tej książce
Rozdział 21 — kosmiczny shooter na Pygame — używa dokładnie tej samej architektury klasy `Game` (Rozdział 20.26) jako ostateczną wersję tego rozdziału, ale dodaje wrogów, pociski i bardziej złożone starcia. To naturalna kontynuacja tego samego portfolio, a nie osobny projekt od podstaw.

Projekt końcowy: „Skacząca Piłka”

Ta sama piłka co w sekcji 20.5 — ale teraz z architekturą, która przetrwa rozwój projektu do ponad jednego pliku.

Wróćmy ostatni raz do odbijającej się piłki z sekcji 20.5, ale teraz ją złóżmy na nowo Ponowne wykorzystanie wszystkiego, co rozdział dodał po drodze: architektury klas `Game` (Rozdział 20.26), ruch przez delta time (Rozdział 20.16) oraz jawne stany gry (Rozdział 20.25).

myach : BouncingBallGame

```
state = SostoyanieIgry.IGRA

x, y = 38.7, 328.8

vx, vy = -182.5,…
-122,9 # pikseli w
drugi

otskokov = 7

schet = 70
```

Rzeczywisty stan działającej gry w konkretnym momencie czasu — to, co faktycznie jest przechowywane w pamięci między klatkami. Zwróć uwagę na dwie prawidłowości: wynik zawsze jest równy liczbie odbić pomnożonej przez 10, a długość wektora prędkości pozostaje równa 220 pikseli na sekundę — odbicie zmienia kierunek, ale nie zmienia wielkości.

Co się zmieniło w porównaniu z wersją 20.5

	bouncing_ball_basic.py (20.5)	bouncing_ball.py (ta wersja, Pro)
Organizacja Kodeksu	Zmienne globalne, funkcje płaskie	Game klas z handle_events/update/render/run metodami

	bouncing_ball_basic.py (20.5)	bouncing_ball.py (ta wersja, Pro)
Wniosek	x += dx na klatkę – zależy od FPS	x += vx * dt — nie zależy od FPS (rozdział 20.16)
Pauza	Tak	Zjada, zatrzymuje update(), nie tylko renderowanie
Restart	Nie, musisz zrestartować program	Klucz, bez restartowania procesu
Ocena	Tak	Liczba punktów i licznik odbić
Interakcja myszą	Tak	Kliknięcie myszką — opcjonalny obrót piłki w stronę kliknięcia

bouncing_ball_pro_fragment.py

```
class BouncingBallGame:
    def __init__(self):
        pygame.init()
        self.screen = pygame.display.set_mode((SHIRINA,
VYSOTA))
        self.clock = pygame.time.Clock()
        self.state = SostoyanieIgry.IGRA
        self.running = True
        self._reset_myach()

    def update(self, dt):
        if self.state is not SostoyanieIgry.IGRA:
            return
        self.x += self.vx * dt
        self.y += self.vy * dt
        if self.x - RADIUS < 0 or self.x + RADIUS > SHIRINA:
            self.vx = -self.vx
            self.otskokov += 1
        if self.y - RADIUS < 0 or self.y + RADIUS > VYSOTA:
            self.vy = -self.vy
            self.otskokov += 1
```

Real Window: Czerwona piłka na górnej krawędzi ciemnoniebieskiego boiska, tekst w lewym górnym rogu „Score: 10 Zbiórki: 1”

Rzeczywiste okno: finalna Pro-wersja podczas gry — wynik i liczba odbić rosną razem, jak powinny zgodnie z regułą punktacji: jedno odbicie to dziesięć punktów.

Prawdziwe okno: ta sama piłka i wynik, z żółtym napisem „PAUSE – P to continue”

Realne okno: stan PAUZY – nakład na zamrożoną klatkę, dokładnie w tej samej pozycji co przed naciskiem pauzy.

Kompletny, już zeskanowany plik – osobno:

projects/pygame/bouncing-ball/bouncing_ball.py

★★ Zadanie samodzielne

Zderzenie piłka z piłką

Dodaj drugą kulę i zderzenie między tymi dwoma kulami na colliderect() (Rozdział 20.21), nie tylko ścianami.

Challenge

Menu i Game Over

Dodaj stany MENU i GAME_OVER (sekcja 20.25): gra zaczyna się w MENU po naciśnięciu klawisza, a po określonej liczbie odbić przechodzi do GAME_OVER z końcowym wynikiem na ekranie.

Praktyka: Składanie wersji Pro-

Pygame otwiera natywne okno Python – uruchomione lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/20-33/index.html)

Podsumowanie rozdziału

Czego nauczyliśmy się w tym rozdziale

- Pygame jest biblioteką, a nie silnikiem: zapewnia okno, renderowanie, wejście i dźwięk, a strukturę gry budujesz samodzielnie (Rozdział 20.8).
- Tworzenie gier — cały świat ról, gatunków i platform, a Pygame pewnie obsługuje przede wszystkim desktop, a web i mobile — przez oddzielne narzędzia społeczności (rozdziały 20.6–20.13, 20.29–20.31).
- W tym rozdziale zbudowaliśmy pętlę gry według podstawowego wzoru Input → Update → Render, powtarzanego dziesiątki razy na sekundę (Rozdział 20.14).
- Ruch piksel na klatkę zależy od czyjegoś sprzętu; ruch przez delta time nie jest (Rozdział 20.16).
- pygame.Rect i colliderect() — podstawa kolizji, a hitbox nie musi pokrywać się z obrazkiem (rozdział 20.21).
- Jawne stany gry (enum) oraz separacja handle_events(), update(dt), render() i run() to prosta i użyteczna architektura dla projektów Pygame- oraz dobra podstawa do kolejnego rozdziału (sekcje 20.25–20.26).

ROZDZIAŁ 21 · STR. 1369

Project: kosmiczna strzelanka z Pygame

Największy projekt książki — statek, wrogowie, strzelanie, wynik, życie, trudność i pełny koniec gry, zebrane w architekturę opartą na klasach i czasie delta.

Plan gry i przygotowanie projektów

Największa gra książki — wszystkie części są już znane z rozdziału 20, tutaj działają razem po raz pierwszy.

Ten projekt to największa gra w książce: statek gracza, wrogowie schodzący z góry, strzelanie, niszczenie wrogów, zdobywanie punktów, życia, rosnący poziom trudności i pełne zakończenie gry z restartem.

Wszystkie elementy są już znane z rozdziału 20 — `Rect`, `colliderect()`, sprite'y, stany gry delta time — oto są pracuj razem po raz pierwszy, w tym samym projekcie edukacyjnym.

Prawdziwe ujęcie ukończonej gry: niebieski statek poniżej, kilku wrogów różnych typów i pociski na polu gry, HUD z punktami i życiami na wierzchu

Prawdziwe ujęcie finalnej wersji gry to efekt tego rozdziału.

Importuj niezbędne moduły

importy.py

```
import random
from dataclasses import dataclass
from enum import Enum, auto
from pathlib import Path

import pygame
```

`dataclasses` (rozdział 14) i `enum` (rozdział 18) tutaj nie są przypadkowe: opis klasy typu wroga i jawne stany gry — ten sam zabieg, co `SostoyanieIgry` w sekcji 20.25.

Inicjalizuj wszystko, czego potrzebujesz

inicializaciya.py

```
SHIRINA, VYSOTA = 480, 720
FPS = 60

pygame.init()
screen = pygame.display.set_mode((SHIRINA, VYSOTA))
pygame.display.set_caption(„Space Shooter”)
clock = pygame.time.Clock()
shrift = pygame.font.SysFont(None, 28)
```

pygame.font — ten sam moduł, co w rozdziale 20
`pygame.font.SysFont(None, 28)` tworzy domyślny rozmiar czcionki systemowej 28 — będziesz go potrzebować do tablicy wyników w sekcji 21.5. Pełna wersja końcowa używa jednocześnie trzech rozmiarów czcionek — zwykłej, dużej (tytuły ekranu) i małych (napisy).

Plan projektu

Zanim napiszesz kod, warto zobaczyć mapę tego, co ostatecznie otrzymujesz. Wersja ostateczna (Rozdział 21.25) składa się z pięciu klas o jasnych obowiązkach:

Pięć Klas Końcowych

GAME

```
Kontroluje cały stan gry
Pętla Gry: Zdarzenia → Aktualizacje →
Aktualizacje → Renderowanie
Przełączniki MENU/PLAYING/PAUSED/GAME_OVER
stany
```

PLAYER

```
Pozycja jak Vector2 (rozdział 21.11)
Ruch ograniczony przez pole gry
Interwał między strzałami i licznik
nieśmiertelności
```

BULLET

```
Leci w górę z stałą prędkością
```

Usuwa się poza polem gry

ENEMY

Lecą w dół z indywidualną prędkością

Punkty do zniszczenia

EXPLOSION

Animacja wieloklatkowa

Usuwa się po ostatniej klatce

Ta sekcja oraz sekcje 21.2–21.8 powtarzają historyczny porządek książki papierowej I celowo zaczynają się prościej niż w ostatecznej wersji — z typowymi funkcjami i listami, jak w rozdziale 20. W sekcji 21.9 książka wyjaśnia, jak i dlaczego ten kod rozwija się w architekturze klas.

Praktyka: import i inicjalizacja
Pygame otwiera natywne okno Python – uruchomione lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/21-01/index.html)

Pętla rozgrywki i statek gracza

Plan tego, co powinno dziać się na każdej klatce — i pierwszy bohater gry.

Pętla rozgrywki

Podobnie jak w rozdziale 20, pętla jest sercem gry. Tym razem odświeża się jednocześnie na każdej klatce kilka rzeczy: pozycja statku, wrogowie, pociski – oraz sprawdzanie kolizji między nimi. S Z pierwszego kadłu liczenie cyklu delta time (w skrócie dt) — sekcja

20.16 już wyjaśniła, Po co: Prędkość określana w pikselach na sekundę i stosowana w dt jest niezależna od częstotliwości Klatki konkretnego komputera:

igrovoj_cikl_plan.py

```
while rabotaet:
    dt = clock.tick(FPS) / 1000.0   # sekund od ostatniej
klatki

    # 1. obsłużyć zdarzenia (wyjście, pauza, strzał po
naciśnięciu)
    #2. Przetwarzaj zaciskane klucze (ruch statku, utrzymanie
ognia)
    #3. Aktualizuj pozycje wrogów i pocisków o dt, sprawdź
kolizje
    # 4. narysować wszystko od nowa
```

dt tu od początku, nie dodaje się później

Mini-projekt z piłką odbijającą się (Rozdział 20.5) zaczął się od ruchu pikseli na klatkę i przeszedł do delta time dopiero w Sekcji 20.16. Nie zrobimy tego tutaj: ponieważ rozdział 20 już wyjaśnił, dlaczego ruch pikseli na klatkę zależy od czyjegoś sprzętu, space shooter oblicza prędkość w pikselach na sekundę (px/s) od pierwszego ruchomego obiektu i stosuje ją przez dt.

Tworzenie statku kosmicznego

Zamiast oddzielnych zmiennych x, y statku wygodnie jest używać `pygame.Rect` — od razu przechowuje pozycję i rozmiar razem i będzie potrzebny do sprawdzania kolizji:

korabl.py

```
KORABL_SHIRINA, KORABL_VYSOTA = 44, 44

korabl = pygame.Rect(
    SHIRINA // 2 - KORABL_SHIRINA // 2,
    VYSOTA - KORABL_VYSOTA - 20,
    KORABL_SHIRINA,
    KORABL_VYSOTA,
)

# w pętli gry:
pygame.draw.rect(screen, (80, 220, 120), korabl)
```

Real Window: Zbliżenie niebieskiego trójkątnego statku z pomarańczowym blaskiem silnika poniżej

Ostateczna wersja rysuje statek z własnym sprite'em (sekcja 21.10) zamiast prostokąta, ale Rect pozostaje jego hitboxem.

Rect przechowuje cztery liczby — oraz wiele użytecznych właściwości

`Rect(x, y, ширина, высота)` dodatkowo oblicza `.centerx`, `.bottom`, `.top` inne wygodne współrzędne — przydadzą się w następnej części.

Praktyka: Rect i pierwszy statek

Pygame otwiera natywne okno Python – uruchomione lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/21-02/index.html)

Ruch statków i wrogowie

od początku prędkość pikseli na sekundę – i wrogów pojawiających się z góry z czasem.

Przesuwamy statek kosmiczny

Sterowanie – przez `get_pressed()` z rozdziału 20 (ruch ciągły, dopóki klawisz jest wciśnięty). Prędkość od razu podana w pikselach na sekundę (px/s), a pozycja statku przechowywana jest jako osobna

liczba ułamkowa — `Rect` rozumie tylko całkowite współrzędne, a aktualizowanie go bezpośrednio po troszeczku w każdej klatce oznaczałoby utratę części ułamkowej ruchu:

dvizhenie_korablya.py

```
KORABL_SKOROST = 260.0   # px/s

korabl_x = float(korabl.x)   # ułamkowa kopia pozycji — sam
Rect przechowuje jedynie liczby całkowite

def obrabotat_klavishi(korabl_x, klavishi, dt):
    napravlenie = klavishi[pygame.K_RIGHT] -
klavishi[pygame.K_LEFT]
    korabl_x += napravlenie * KORABL_SKOROST * dt
    return max(0.0, min(korabl_x, SHIRINA - KORABL_SHIRINA))

# w pętli gry:
korabl_x = obrabotat_klavishi(korabl_x, klavishi, dt)
korabl.x = round(korabl_x)
```

max/min jest tą samą metodą powstrzymania co w rozdziale 20
`max(0.0, min(korabl_x, SHIRINA - KORABL_SHIRINA))` gwarantuje, że `korabl_x` nigdy nie przekroczy granic `[0, SHIRINA - KORABL_SHIRINA]` jest ta sama technika „ściskania”

napravlenie to liczba -1, 0 lub 1
`klavishi[pygame.K_RIGHT] - klavishi[pygame.K_LEFT]` są wartościami boolowskimi `True` / `False` zachowują się jak 1/0 (Rozdział 4), więc różnica daje dokładnie to, czego chcesz: 1, jeśli trzymasz tylko strzałkę w prawo, −1, jeśli trzymasz tylko strzałkę w lewo, i 0, jeśli obie lub żadne. Rozdział 21.11 podsumowuje tę samą technikę w dwóch wymiarach jednocześnie `Vector2`.

Tworzenie i przesuwanie wrogów

Wróg pojawi się później jako osobna klasa sprite'ów (sekcja 21.9), ale na razie – słownik z `Rect` i współrzędną ułamkową Y, z tej samej przyczyny co statek. Będzie wielu wrogów i pojawiają się z czasem — więc potrzebny jest lista (rozdział 11) i timer w sekundach do kolejnego pojawienia się:

vragi.py

```
VRAG_SHIRINA, VRAG_VYSOTA = 32, 28
VRAG_SKOROST = 150.0            # px/s
INTERVAL_POYAVLENIYA_VRAGA = 0.75   # sekund między nowymi
wrogami

vragi = []
vremya_do_vraga = INTERVAL_POYAVLENIYA_VRAGA

def sozdat_vraga():
    x = random.randint(0, SHIRINA - VRAG_SHIRINA)
    return {{
        "rect": pygame.Rect(x, -VRAG_VYSOTA, VRAG_SHIRINA,
VRAG_VYSOTA),
        "y": float(-VRAG_VYSOTA),
    }}

# w pętli gry, każda klatka:
vremya_do_vraga -= dt
if vremya_do_vraga <= 0.0:
    vragi.append(sozdat_vraga())
    vremya_do_vraga += INTERVAL_POYAVLENIYA_VRAGA

for vrag in vragi:
    vrag["y"] += VRAG_SKOROST * dt
    vrag["rect"].y = round(vrag["y"])
```

Prawdziwe okno: mały pomarańczowy trójkątny wróg schodzi z góry w stronę niebieskiego statku gracza

Prawdziwym Oknem jest Mały Zwiadowca (scout), jeden z dwóch typów wrogów w ostatecznej wersji (Rozdział 21.17).

y = -VRAG_VYSOTA — wróg pojawia się nieco powyżej ekranu
Ujemna początkowa współrzędna Y oznacza, że wróg rodzi się nieco powyżej widocznego obszaru i płynnie „wjeżdża” do kadru od góry — a nie pojawia się nagle pośrodku ekranu.

+= interval, a nie = interval — to właśnie zachowuje resztę

Gdy pojawia się wróg, dodajemy ten przedział do aktualnej wartości timera zamiast przypisywać go ponownie. W związku z tym nie tracimy niewielkiego przekroczenia czasu: jeśli `vremya_do_vraga` zeszedł do −0.01, następny odczyt rozpocznie się właśnie od tej małej zaległości, a nie od czystego interwału. W rozdziale 21.14 ta sama zasada zostanie uogólniona za pomocą `while` w przypadku, gdy jeden przetworzony krok może obejmować kilka przedziałów jednocześnie.

Praktyka: ruch statku i wrogów

Pygame otwiera natywne okno Python – uruchomione lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/21-03/index.html)

Strzelanie: Tworzenie i przesuwanie pocisków

Spacja tworzy kulę na dziobie statku – leci w górę, aż opuści ekran.

Pociski - słownik `Rect` i współrzędna ułamkowa Y występuje w dziób statku, gdy naciska się spację, wznosi się z prędkością w pikselach na sekundę:

strelba.py

```
PULYA_SHIRINA, PULYA_VYSOTA = 6, 18
PULYA_SKOROST = 560.0   # px/s

puli = []

def vystrelit():
    pulya_rect = pygame.Rect(
        korabl.centerx - PULYA_SHIRINA // 2,
        korabl.top,
        PULYA_SHIRINA,
        PULYA_VYSOTA,
    )
    puli.append({{"rect": pulya_rect, "y":
float(pulya_rect.y)}})

# w obsłudze zdarzeń:
if event.type == pygame.KEYDOWN and event.key ==
pygame.K_SPACE:
```

```
    vystrelit()

# w aktualizacji ramek:
for pulya in puli:
    pulya["y"] -= PULYA_SKOROST * dt
    pulya["rect"].y = round(pulya["y"])
puli = [p for p in puli if p["rect"].bottom > 0]   # usuwamy
te, które wyleciały poza ekran
```

Rzeczywiste okno — pocisk pojawia się dokładnie przy nosie statku i leci do góry.

Lista przez generator list — czyszczenie „śmieci”

`[p for p in puli if p["rect"].bottom > 0]` (generator list z rozdziału 11) pozostawia tylko te pociski, które są widoczne na ekranie – bez tej linii lista pociski rosłyby w nieskończoność, co coraz bardziej spowalniałby grę.

KEYDOWN — jedno zdarzenie na jedno naciśnięcie

Wydarzenie `pygame.KEYDOWN` następuje raz w momencie fizycznego naciśnięcia klawisza. Automatyczne powtarzanie klawisza w Pygame jest domyślnie wyłączone (można go włączyć tylko wyraźnie, przez `pygame.key.set_repeat(...)` i nie mamy) oznacza, że trzymanie luki nie generuje nowej `KEYDOWN` na każdej klatce. Tutaj daje dokładnie jeden strzał jednym kliknięciem: wygodne do pojedynczego strzału, ale nieodpowiednie do trzymania ognia – zostanie to omówione w sekcji 21.13.

Praktyka: Strzelanie i ruch pocisków

Pygame otwiera natywne okno Python – uruchomione lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/21-04/index.html)

Ocena i informacje na ekranie

Tekst w Pygame jest rysowany w dwóch etapach: najpierw render(), potem blit() na ekranie.

Faktura jest wyświetlana gotową czcionką Pygame (`pygame.font` , sekcja 21.1) — w przeciwieństwie do Turtle, tutaj tekst najpierw jest „narysowany” `render()`) i nakłada na ekran (`blit()`):

tablo_scheta.py

```
schet = 0

# w renderowaniu klatek:
tablo = shrift.render(f"Ocena: {schet}", True, (255, 255, 255))
screen.blit(tablo, (10, 10))
```

render() + blit() to ta sama zasada co write() Turtle

`render(текст, сглаживание, цвет)` tworzy obraz tekstu; `screen.blit(изображение, позиция)` nakłada go na ekran — koncepcyjnie to samo, co `artist.write()` Turtle od rozdziału 7, tylko w dwóch wyraźnych etapach zamiast jednego.

Real Window: Górny pasek interfejsu z wynikiem 100 po lewej i liczbą żyć po prawej na ciemnym tle

Ostateczna wersja wyróżnia wynik i życia na oddzielnym pasku HUD u góry (rozdział 21.12) — to ta sama tablica, po prostu oddzielona od pola gry.

Praktyka: Obejmuje wypłaty z konta

Pygame otwiera natywne okno Python – uruchomione lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/21-03/index.html)

Uderzenia i kolizje

collideRect() sprawdza kolizje — trafienie niszczy wroga, a wróg może zniszczyć statek.

Niszczyć wrogów

Zderzenie kuli z wrogiem sprawdza gotową metodę `Rect.colliderect()` — po trafieniu oba są usuwane ze swoich list, a wynik się zwiększa:

unichtozhenie_vragov.py

```
novye_puli = []
novye_vragi = list(vragi)
for pulya in puli:
    popala = False
    for vrag in list(novye_vragi):
        if pulya.colliderect(vrag):
            novye_vragi.remove(vrag)
            schet += 10
            popala = True
            break
    if not popala:
        novye_puli.append(pulya)
puli = novye_puli
vragi = novye_vragi
```

Dlaczego nie usuwać elementów bezpośrednio podczas iterowania po liście?

Usuń element z listy `vragi` bezpośrednio w pętli `for vrag in vragi:` może pominąć sąsiednie elementy — lista „przesuwa się" pod nogami pętli. Bezpieczniej zebrać **nowe** listy ocalałych pocisków i wrogów, jak pokazano powyżej. Ostateczna wersja (Rozdział 21.9) rozwiązuje ten sam problem bez ręcznych list – za pośrednictwem `pygame.sprite.Group` oraz `groupcollide()` .

Real Window: Pomarańczowy błysk w momencie, gdy kula trafiła wroga, wynik wzrósł do 100

Prawdziwe okno — moment trafienia: wróg zniszczony, wynik zwiększył się dokładnie jeden raz.

Gdy wróg niszczy statek

Jeśli wróg dotrze na dół ekranu lub zderzy się ze statkiem, gra kończy się:

unichtozhenie_korablya.py

```
for vrag in vragi:
    if vrag.bottom >= VYSOTA or vrag.colliderect(korabl):
        igra_okonchena = True
        break
```

Natychmiastowy koniec gry — również rozwiązanie tymczasowe
Tutaj jedno starcie natychmiast kończy grę. Ostateczna wersja (sekcja 21.16) zastępuje to systemem żyć z tymczasową niewrażliwością po trafieniu, dzięki czemu trzech wrogów zderzających się jednocześnie ze statkiem nie odbiera trzech żyć naraz.

Praktyka: Pociski, wrogowie i zderzenia statków
Pygame otwiera natywne okno Python – uruchomione lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/21-06/index.html)

Koniec gry i ekran zakończenia gry

Każda klatka jest rysowana od nowa w całości — a po zakończeniu gry na ekranie pojawia się końcowy napis.

Przerysuj ramkę

Po wszystkich kontrolach i aktualizacjach ramka zostaje całkowicie przerysowana — tło, statek, pociski, wrogowie i liczba kolejnych, w takiej kolejności (tak by późniejsze elementy były na wierzchu więcej wczesny):

pererisovka.py

```
screen.fill((10, 10, 20))
pygame.draw.rect(screen, (80, 220, 120), korabl)
for pulya in puli:
    pygame.draw.rect(screen, (240, 220, 80), pulya)
for vrag in vragi:
    pygame.draw.rect(screen, (230, 60, 60), vrag)

tablo = shrift.render(f"Ocena: {schet}", True, (255, 255, 255))
screen.blit(tablo, (10, 10))

pygame.display.flip()
```

Ekran Koniec gry

Kiedy `igra_okonchena` staje się prawdziwym zamiast zwyczajnym Logika gry, wyświetlany jest ostatni ekran:

game_over.py

```
if igra_okonchena:
    nadpis = shrift_bolshoj.render(„GRA SIĘ SKOŃCZYŁA”, True,
(255, 255, 255))
    rect = nadpis.get_rect(center=(SHIRINA // 2, VYSOTA // 2))
    screen.blit(nadpis, rect)
```

Prawdziwe okno: przezroczysta, ciemna nałożona na cały ekran, duże GAME OVER, Enter z podkładem i podpowiedzą – znowu

Ostateczna wersja dodaje na ekran Game Over wynik, zapis sesji oraz wyraźną wskazówkę, jak zacząć od nowa (sekcja 21.19).

get_rect(center=...) — wygodne wyśrodkowanie tekstu
Zamiast ręcznie obliczać współrzędne tekstu na podstawie jego szerokości i wysokości, `get_rect(center=(x, y))` sam znajduje potrzebny lewy górny róg tak, aby tekst był dokładnie wyśrodkowany w punkcie `(x, y)` .

Nie ma wyjścia z tego ekranu bez ponownego uruchomienia Python

Zmienna `igra_okonchena` nie ma już drogi powrotu, jedynym sposobem na ponowne uruchomienie programu jest ponowne uruchomienie programu. Rozdział 21.18 zastąpi jedną zmienną boolowską jawną `enum.Enum` z czterema stanami (MENU/PLAYING/PAUSED/GAME_OVER) i wyraźnymi przejściami między nimi, w tym powrotem do gry.

Praktyka: Zawiera ostatni ekran

Pygame otwiera natywne okno Python – uruchomione lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/21-06/index.html)

Pierwsza działająca wersja gry

Sekcje 21.1–21.7 razem dają działającą grę, w którą można już grać — a rozdział trwa dalej.

Zbierając razem sekcje 21.1–21.7, otrzymujemy pierwszą roboczą wersję gry, w którą można już grać — to właśnie na niej kończy się rozdział w książce papierowej. Ruch statku, wrogów i pocisków nie zależy już od liczby klatek na sekundę: prędkości są podane w pikselach na sekundę, a pojawienie się wrogów — w timerze w sekundach, tak jak w sekcjach 21.2–21.4. Następnie, od sekcji 21.9, książka kontynuuje w wersji cyfrowej: ten sam projekt przekształca się w architekturę opartą na klasach, zyskuje własną grafikę i dźwięk, system żyć, zwiększanie trudności oraz pełnoprawne stany gry.

Cyfrowa kontynuacja projektu — poniżej w spisie treści rozdziału

Sekcje 21.9 do 21.26 w pasku bocznym nie są osobnym tematem, lecz kolejnym etapem rozwoju tej samej gry: te same mechaniki, ale kod jest czystszy, dokładniejszy i bardziej niezawodny na każdym etapie.

★★ Zadanie samodzielne

Życie zamiast natychmiastowego końca

Dodaj zmienną zhizni = 3 – gdy statek zderzy się z wrogiem, odbierz jedno życie i usuń przeciwnika, zamiast natychmiast kończyć grę. (Rozdział 21.16 pokazuje gotowe rozwiązanie z tymczasową niewrażliwością.)

Challenge

Poziomy trudności

Stopniowo zmniejszaj interwał pojawiania się wrogów wraz ze wzrostem wyniku — wrogowie powinni pojawiać się częściej. (Rozdział 21.17 pokazuje, jak to zrobić bez gwałtownego skoku.)

Praktyka: Pierwsza wersja robocza

Pygame otwiera natywne okno Python – uruchomione lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/21-08/index.html)

Dzielenie gry na klasy

Pięć klas z jasnymi odpowiedzialnościami zamiast tylko globalnych zmiennych i list.

Kod historyczny dla sekcji 21.1 do 21.7 przechowuje statek, wrogów i pociski jako oddzielne zmienne oraz listy `Rect` , ale cała logika jest w samej pętli gry. To jest Działa w małej grze, ale rozwija się słabo: im więcej zasad, tym trudniej zrozumieć, która z nich Za co odpowiada kod. Ostateczna wersja dzieli grę na pięć klas z jasnymi obowiązkami.

Game kontroluje cały stan gry i koordynuje pozostałe klasy – sam nie losuje przeciwnika i nie przesuwa pocisków bezpośrednio.

Game przechowuje jedną kopię Player.

Game utrzymuje jeden pygame.sprite.Group dla Bullet, Enemy i Explosion.

`Bullet`, `Enemy` oraz `Explosion` są spadkobiercami `pygame.sprite.Sprite` (Rozdział 20.19 już pokazała, dlaczego potrzebne są sprite'y: obiekt ma `.image` oraz `.rect`, które rozumieją gotowe funkcje kolizji i grup). Przechowywać je w `pygame.sprite.Group`, a nie w zwykłej liście, wygodne z tego samego powodu co w sekcji 21.6: grupa sama potrafi usuwać obiekt (`.kill()`) bez ryzyka „przerwania” `pygame.sprite.groupcollide()` (rozdział 21.15) sprawdza kolizje między całymi grupami jednym wywołaniem.

fragment_game_init.py

```
class Game:
    def __init__(self, *, rng=None, debug=False):
        pygame.init()
        self.screen = pygame.display.set_mode((SHIRINA,
VYSOTA))
        self.clock = pygame.time.Clock()
        self.assets = AssetStore(IMAGE_DIR, AUDIO_DIR)
        self.rng = rng if rng is not None else random.Random()

        self.state = GameStatus.MENU
        self.bullets = pygame.sprite.Group()
        self.enemies = pygame.sprite.Group()
        self.explosions = pygame.sprite.Group()
        self.player = self._make_player()
        self.score = 0
        self.lives = STARTING_LIVES
```

To szkic, a nie pełny plik
Kompletna, gotowa do uruchomienia sala lekcyjna `Game` — w aktach `projects/pygame/space-shooter/space_shooter.py`, który szczegółowo omawia sekcję 21.25.

Ta architektura nie jest prawem uniwersalnym
Podział na Game/Player/Bullet/Enemy/Explosion — wygodne rozwiązanie właśnie dla projektu tego rozmiaru, a nie jedyny poprawny sposób budowania każdej gry. Rozdział 20.8 już wyjaśniał: różne narzędzia i projekty mogą mieć bardzo różną architekturę.

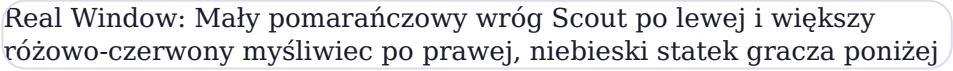
Praktyka: szkic klasy Game
Pygame otwiera natywne okno Python – uruchomione lokalnie w VS Code, PyCharm lub Jupyter
Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/21-09/index.html)

Zasoby gry: grafika i dźwięk

Prawdziwe sprite'y zamiast prostokątów — i klasę, która ładuje je dokładnie raz.

Do tej pory statek, wrogowie i pociski były rysowane prostokątami i kółkami — Przydatne na pierwsze kroki (Rozdział 20.3). Ostateczna wersja używa własnych sprite'ów, narysowane specjalnie na potrzeby tego projektu, zamiast geometrycznych włóczek.

Sprite'y Projektu



Trzy własne sprite'y jednocześnie: zwiadowca (scout), myśliwiec (fighter) i statek gracza) – różne sylwetki i kolory, dzięki czemu przeciwnicy są czytelni na pierwszy rzut oka.

Wszystkie obrazy są zapisywane jako pliki `.png` z przezroczystym tłem. Po załadowaniu przechodzą do Surface obsługującego kanał alfa (flaga `pygame.SRCALPHA` przy tworzeniu takiej Surface, rozdział 20.17, — to jest właściwość samej Surface w pygame, a nie pliku PNG), dlatego przezroczyste piksele pozostają przezroczyste również na ekranie. Sprite'y rysowane są bezpośrednio przez `pygame.draw` w

osobnym piśmie `scripts/generate_chapter_21_assets.py`, a nie pobrana skądś — proste kształty geometryczne, bez żadnego istniejącego komercyjnego sprite'a.

Plik	Rola	Funkcje
player_ship.png	Statek Gracza	Sylwetka w kształcie strzały, jasna kabina, światło silnika
enemy_scout.png	Szybki, tani wróg	Mały, pomarańczowo-czerwony trójkąt
enemy_fighter.png	Powoli, drogi Wrogu	Większy, purpurowo-czerwony, z bocznymi modułami
bullet.png	Pocisk Gracza	Mała, jasnożółta kapsułka
explosion_sheet.png	Animacja eksplozji	6 klatek z rzędu – sekcja 21.20 przecina je przez subsurface()

AssetStore - Załaduj raz

Rozdział 20.24 już ostrzegała: pobieranie pliku w pętli gry to częsta przyczyna Spadki klatek. Tutaj wszystko jest ładowane raz podczas tworzenia gry, w osobnej klasie:

fragment_asset_store.py

```
class AssetStore:
    def __init__(self, image_dir, audio_dir):
        self.images = {{}}
        for key in ("player_ship", "enemy_scout",
"enemy_fighter", "bullet"):
            self.images[key] = pygame.image.load(image_dir /
f"{{key}}.png").convert_alpha()

        self.sounds = {{}}
        for key in ("laser", "explosion", "player_hit"):
            path = audio_dir / f"{{key}}.wav"
            try:
                self.sounds[key] =
pygame.mixer.Sound(str(path))
            except (pygame.error, FileNotFoundError):
                self.sounds[key] = None
```

convert_alpha() — to samo znaczenie, co w rozdziale 20

`pygame.image.load(...)` już zachowuje przezroczystość załadowanego PNG samo w sobie — `.convert_alpha()` nie „tworzy” przezroczystości, lecz przygotowuje kopię w formacie pikseli ekranu do szybkiego ponownego rysowania (rozdział 20.19). Tutaj jest wywoływana zaraz po załadowaniu, dopóki obrazów jest mało i nie trzeba tego odkładać.

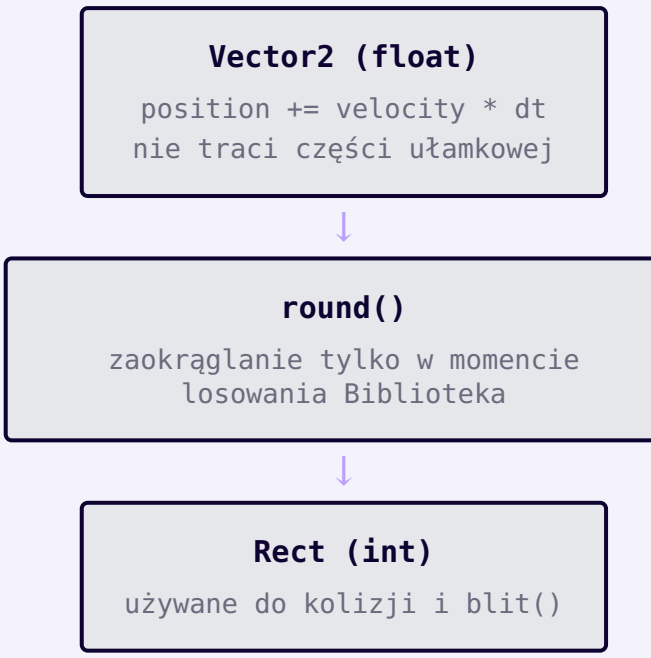
Dźwięk jest opcjonalny na początku gry

Jeśli w systemie nie ma urządzenia dźwiękowego (np. przy automatycznym testowaniu), `pygame.mixer.Sound(...)` może się nie zdać — wtedy `self.sounds[key]` pozostałości `None` , ale gra nadal się uruchamia i działa, tylko bez dźwięku. Rozdział 21.21 pokazuje, jak odtwarzanie uwzględnia to przy każdym połączeniu.

Dokładny ruch z Vector2

Rect przechowuje tylko liczby całkowite — pozycja żyje osobno, jak Vector2 w formacie zmiennoprzecinkowym.

`pygame.Rect` jest przyjazny dla kolizji i renderowania, ale nadal się utrzymuje tylko liczby całkowite. Jeśli dodasz małą ułamek do współrzędnej na każdej ramce wartość cicho traci się podczas okrążania – jazda z bardzo niską prędkością może Nie chce się ruszyć, mimo że kod wygląda poprawnie.



Pozycja zawsze żyje jako zmiennoprzecinkowa Vector2; Rect każda klatka jest z niej ponownie złożona, zamiast bezpośrednio aktualizowana.

fragment_player_move.py

```
class Player(pygame.sprite.Sprite):
    def __init__(self, image, center):
        super().__init__()
        self.image = image
        self.rect = self.image.get_rect()
        self.position = pygame.Vector2(center)
        self.rect.center = (round(self.position.x),
round(self.position.y))

    def move(self, direction, dt, playfield):
        if direction.length_squared() > 0:
            direction = direction.normalize()
        self.position += direction * self.speed * dt
        self.rect.center = (round(self.position.x),
round(self.position.y))
```

direction.normalize() jest tą samą zasadą diagonalną co w sekcji 20.16

Wektor kierunku `(1, 1)` (jednocześnie w prawo) bez normalizacji długości większej niż jeden wynosi √2 ≈ 1,41. Gdyby nie została skrócona do jednostki długości, statek poruszałby się po przekątnej o 41% szybciej niż ściśle wzdłuż jednej osi.

direction.length_squared() > 0 zamiast > 0 natychmiast dla długości

Te dwie linie mają różne zadania. Weryfikacja `> 0` jest to, co nie pozwala ci powodować `normalize()` dla wektora zerowego (gdy gracz wcale nie naciska klawiszy ruchu): normalizacja dzieli wektor przez jego długość, a długość wektora zerowego wynosi zero. A `length_squared()` (kwadrat długości) zamiast `length()` jest potrzebny tylko do unikania ekstrakcyjnego pierwiastka kwadratowego, gdy wystarczy wiedzieć, czy wektor jest niezerowy, co jest powszechną mikrooptymalizacją w rzeczywistym kodzie na Vector2.

Plansza gry nie wypuszcza statku

Każdy ruch jest natychmiast ograniczony do granic pola gry (Rozdział 21.12) – tak jak `max(0, min(...))` w sekcji 21.3, ale obecnie dotyczy zarówno X, jak i Y, ponieważ statek porusza się we wszystkich czterech kierunkach.

Praktyka: Vector2, normalizacja i clamp

Zeskanowanie bezpośrednio w przeglądarce – nie wymaga instalacji Pygame

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/21-11/index.html)

Boisko i interfejs zawodnika

Liczenie i życia na górze znajdują się na własnym pasie — obiekty gry nie wchodzą do niego fizycznie.

Rozdział 19 już pokazał, co się dzieje, gdy interfejs nakłada się na boisko: ważny tekst może znajdować się pod ruchomym obiektem. Ostateczna wersja strzelca od samego początku się odróżnia ekran do dwóch wyraźnie różnych obszarów: pola gry i paska z odczytami dla gracza – wynik, życia, liczba fali. Taka passa na szczycie świata gier w branży nazywana jest zwykle HUD (Heads-Up Display, „dashboard”

Real Window: Górny ciemny pasek ścieżki i HUD żyje oddzielony cienką linią od głównego pola gry, z gwiaździstym tłem i statkiem poniżej

Okno rzeczywiste — pasek interfejsu u góry (64 piksele) jest oddzielony cienką linią od pola gry; Żaden wróg ani pocisk nie może przekroczyć tego limitu.

fragment_playfield.py

```
HUD_HEIGHT = 64

self.playfield = pygame.Rect(0, HUD_HEIGHT, SHIRINA, VYSOTA -
HUD_HEIGHT)
```

Pole gry jest normalne `Rect`, nie tylko numer wcięcia: ma Natychmiast jedz `.top`, `.bottom`, `.left`, `.right` jest takie samo jak zwroty `korabl.top` w sekcji 21.2, tylko teraz już nie jeden obiekt, ale cały dozwolony obszar ruchu.

Co gdzie się dzieje	Obszar
Liczą, Życia, Machają	HUD-pas startowa (0 — HUD_HEIGHT)
Statek, wrogowie, kule, eksplozje	Pole gry (HUD_HEIGHT — dół ekranu)
Wrogowie	Tuż nad szczytem pola gry
Ucieczka wroga (Rozdział 21.16)	Przekroczenie dolnej granicy boiska

Statek jest ograniczony do pola gry, a nie do całego okna
`Player.move()` (Rozdział 21.11) zaciska pozycję w granicach `self.playfield`, nie całe okno `SHIRINA × VYSOTA` — w przeciwnym razie statek mógłby przejechać pod HUD i ukryć się za tekstem ustawy.

Szybkostrzelność i odstępy między strzałami

Przytrzymanie spacji strzela z ograniczoną częstotliwością — w sekundach, a nie w klatkach.

Rozdział 21.4 Zwolniona podczas zdarzenia `KEYDOWN` – to się dzieje dokładnie raz na jedno fizyczne naciśnięcie spacji, więc nawet trzymanie klawisza przez dłuższy czas dawało Tylko jeden strzał. Ostateczna wersja gry wymaga serii strzałów podczas zaciskania spacji, a dla wymaga innego źródła danych — `pygame.key.get_pressed()`, Który informuje, czy klawisz jest teraz przytrzymywany przy każdej aktualizacji. Nieograniczone oznaczałoby to strzał przy każdej aktualizacji gry, czyli szybkostrzelność opartą na FPS: 60 strzały na sekundę przy 60 FPS i 120 przy 120 FPS. Dlatego strzelanie przez `get_pressed()` zawsze towarzyszy minimalny odstęp między strzały (cooldown, w profesjonalnej mowie – czas odnowienia), oddawane w sekundy.

fragment_fire_cooldown.py

```
FIRE_INTERVAL = 0.20   # sekund między strzałami

self.fire_cooldown = max(0.0, self.fire_cooldown - dt)

if keys[pygame.K_SPACE] and self.fire_cooldown <= 0.0:
    self._spawn_bullet()
    self.fire_cooldown = FIRE_INTERVAL
```

Interwał mierzony jest w sekundach, a nie w klatkach
Gdyby zamiast `fire_cooldown -= dt` licznik zmniejszany o jeden w każdej klatce (`kadrov_do_vystrela -= 1`), prędkość strzału zależałaby od FPS dokładnie tak samo, jak nieskorygowany ruch z sekcji 20.16: przy 120 FPS statek strzelałby dwa razy częściej niż przy 60 FPS, przy tym samym kodzie.

Dlaczego tutaj nie używa się while, jak w timerze pojawiania się wrogów

Rozdział 21.14 pokaże ostatni licznik pojawiania się wrogów: przesuwa czas do przodu `+= interval`, zachowując resztę, oraz `while` jest tam potrzebne, jeśli jeden przetworzony etap pokryje kilka przedziałów jednocześnie. Interwały między strzałami są ułożone inaczej:

	Timer pojawiania się wrogów	Interwał między strzałami zawodników
Źródło zdarzenia	Symulacja offline (sama gra decyduje, kiedy nastąpi)	Wprowadzanie gracza (człowiek decyduje, kiedy)
Długi etap przetwarzania	Jeśli krok może obejmować wiele interwałów, timer może przetwarzać wiele zdarzeń z rzędu	nie wolno wystrzeliwać kilku pocisków jednocześnie w jednej klatce
Trik w kodzie	while: kontynuować, dopóki jest czas	if: nie więcej niż jedno ujęcie na klatkę

W bieżącej grze drugi wiersz nie następuje

`MAX_DT` nie pozwala, aby pojedynczy przetworzony krok był dłuższy od minimalnego interwału pojawiania się wrogów (szczegóły — w rozdziale 21.14) — dlatego w tej konkretnej grze ani przerwa debuggera, ani zawieszenie systemu nie powodują, że gra „nadgania" cały nagromadzony czas rzeczywisty naraz od kilku wrogów.

To świadomy wybór, a nie przeoczenie

Po naprawdę długim strzale (pauza debugera, zamrożenie systemu) nagła salwa dziesięciu pocisków wydaje się raczej błędem niż uczciwym zachowaniem — więc interwał strzału celowo daje maksymalnie jeden nowy strzał na odświeżenie, nawet jeśli `dt` okazała się duża.

Praktyka: Przerwa między ujęciami

Zeskanowanie bezpośrednio w przeglądarce – nie wymaga instalacji Pygame

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/21-13/index.html)

Jak pojawiają się wrogowie

Timer pojawienia się w sekundach — z tym samym akumulatorem while-, co timer animacji w rozdziale 20.

W sekcji 21.3 mierzyliśmy już czas do następnego wroga w sekundach: licznik zmniejszał się o `dt` każdą klatkę, a po wyzwalaniu dodawałem interwał ponownie, i nie przyjął ją, więc reszta nie została utracona.

Teraz przenieśmy tę samą zasadę do finalnej części architektura gry oraz uczynić timer odpowiednim dla interwału zmieniającego się wraz z złożoność.

fragment_spawn_timer.py

```
self.spawn_timer -= dt
while self.spawn_timer <= 0.0:
    self._spawn_enemy()
    self.spawn_timer += interval_poyavleniya_vraga(self.score)
```

Dwa niezależne mechanizmy, nie jeden

`self.spawn_timer += interval_poyavleniya_vraga(self.score)` — to, co potrzebne w każdym kadrze: pozostały czas ponad interwał nie jest odrzucany, lecz przenoszony na następnego przeciwnika, dokładnie tak samo jak w sekcji 21.3 i jak licznik animacji w sekcji 20.23. To zachowanie reszty działa również z zwykłym `if` — nie wymaga `while`. Sam `while` jest potrzebne do czegoś innego: jeśli jeden przetworzony krok trwałby dłużej niż kilka odstępów pojawiania się wrogów jednocześnie, szczerze mówiąc, stworzyłby kilku wrogów pod rząd podczas jednego wezwania, oraz `if` tylko przyspieszyłoby pojawienie się jednego wroga i po cichu traciło resztę zgromadzonego czasu.

W tej konkretnej grze drugi przypadek jeszcze nie ma miejsca

Ostateczna wersja gry ogranicza `dt` znaczenie `MAX_DT = 0.05` sekund przed każdą aktualizacją (ochrona przed skakaniem po pauzie debugera lub zamrożeniu systemu), a dolna granica interwału pojawiania się wrogów to `MIN_SPAWN_INTERVAL = 0.35` sekundy: nawet najdłuższa dopuszczalna klatka jest krótsza niż jakikolwiek możliwy odstęp między wrogami. Zatem ciało `while` w tej grze nie robi się już więcej niż raz na klatkę — i jest zapisane `if` kod zachowywałby się tutaj dokładnie tak samo. `while` pozostawiona jako bardziej ogólna i niezawodna technika: nie zepsuje się, jeśli później podniesiesz MAX_DT lub skracz interwał pojawiania się wrogów bardziej niż teraz, oraz `if` w tym przypadku musiałoby zostać zmienione.

Gdzie dokładnie pojawia się wróg?

Współrzędna X powinna zapewnić, że przeciwnik całkowicie mieści się na polu gry — bez częściowego „przycinania”

fragment_spawn_x.py

```
def x_poyavleniya_vraga(rng, shirina_vraga, pole):
    levaya, pravaya = pole.left, pole.right - shirina_vraga
    if pravaya <= levaya:
        return float(levaya)
    return rng.uniform(levaya, pravaya)
```

To odrębna, czysta funkcja — akceptuje `rng` wyraźnym argumentem (Rozdział 21.24 wyjaśnia dlaczego) zamiast zajmować się globalnym problemem `random` bezpośrednio, dzięki czemu można ją wygodnie testować bez uruchamiania całej gry.

Dwa typy wrogów

Real Window: Wielu przeciwników o różnych rozmiarach i kolorach zjeżdża na planszę jako grupa

Rzeczywiste okno — fala wrogów: małe, szybkie zwiadowce i większe myśliwce pojawiają się na przemian.

	Scout (Zwiadowca)	Fighter (wojownik)
Speed	Powyżej	Poniżej
Rozmiar	mniej	Więcej
Punkty do zniszczenia	100	200
Prawdopodobieństwo pojawienia się	Wyżej na początku gry	Rośnie wraz z liczeniem (Rozdział 21.17)

fragment_enemy_spec.py

```
@dataclass(frozen=True)
class EnemySpec:
    name: str
    base_speed: float
    points: int
    image_key: str

ENEMY_SPECS = {
    "scout": EnemySpec("scout", base_speed=150.0, points=100,
image_key="enemy_scout"),
    "fighter": EnemySpec("fighter", base_speed=85.0,
points=200, image_key="enemy_fighter"),
}
```

EnemySpec opisuje typ, a nie pojedynczego wroga

`@dataclass` z rozdziału 14 automatycznie tworzy konstruktor dla czterech pól. Parametr `frozen=True` zabrania zmiany tych pól po utworzeniu obiektu: ogólna specyfikacja scout lub fighter pozostaje niezmieniona, a współrzędne i bieżąca prędkość są przechowywane w każdej instancji `Enemy`. Aby dodać typ wroga w ćwiczeniu 21.25, stwórz nowy `EnemySpec` zamiast modyfikować istniejący.

Praktyka: timer pojawiania się wrogów

Zeskanowanie bezpośrednio w przeglądarce – nie wymaga instalacji Pygame

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/21-14/index.html)

Trafianie wrogów i zdobywanie punktów

groupcollide() sprawdza kolizje całych grup jednym wyzwaniem – ale punkty trzeba przyznawać ostrożnie.

Rozdział 21.6 sprawdzała kolizje pocisków i wrogów ręcznie, w zagnieżdżonych pętlach z list. Grupy sprite'ów (Rozdział 21.9) rozwiązują ten sam problem jednym wywołaniem:

fragment_bullet_enemy_collisions.py

```
def resolve_bullet_enemy_collisions(self):
    collisions = pygame.sprite.groupcollide(self.bullets,
self.enemies, True, True)
    destroyed = {{vrag for vragi in collisions.values() for
vrag in vragi}}
    for vrag in destroyed:
        self._spawn_explosion(vrag.rect.center)
    return ochki_za_unichtozhennyh(destroyed)
```

`pygame.sprite.groupcollide(a, b, True, True)` każdy porównuje Sprite z `a` z każdym sprite'm z `b` i zwraca słownik: pocisk → lista wrogów, których dotknął. Obie flagi `True` oznaczają „usuń ze swojej grupy przy kolizji” — sama grupa przejmuje to samo zadanie, które ręcznie rozwiązywały nowe listy w rozdziale 21.6.

Real Window: Błyskowa eksplozja, gdzie kula trafia wroga, wynik na górze wzrasta

Prawdziwym oknem jest moment uderzenia: zarówno pocisk, jak i wróg zostali już usunięci ze swoich grup, eksplozja została wystrzelona.

Dlaczego liczenie punktów odbywa się przez set, a nie przez sumę wszystkich par

Jeśli kilka pocisków trafi tego samego wroga w tej samej aktualizacji, `groupcollide()` przywróci tego wroga do list dla każdego z nich — ale fizycznie to ten sam zniszczony wróg. Zbierając wszystkich dotkniętych wrogów w zbiór (`set`) przed punktowaniem każdy przeciwnik jest liczony dokładnie raz, niezależnie od tego, ile kul trafi go jednocześnie.

Praktyka: kolizje i liczenie punktów

Zeskanowanie bezpośrednio w przeglądarce – nie wymaga instalacji Pygame

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/21-15/index.html)

Życia, obrażenia i tymczasowa niewrażliwość

Spotkania i ucieczki wrogów to dwa różne źródła śmierci z różnymi zasadami.

Rozdział 21.6 zakończyła grę pojedynczą kolizją statku z wrogiem. Wersja ostateczna Zamiast tego liczy się życie — ale jest tu mniej oczywista pułapka: co jeśli jest ich kilka naraz Wrogowie zderzają się ze statkiem w tym samym momencie?

fragment_player_damage.py

```
def resolve_enemy_player_collisions(self):
    hit = pygame.sprite.spritecollide(self.player,
self.enemies, True)
    for vrag in hit:
        self._spawn_explosion(vrag.rect.center)
    return len(hit) > 0

# w aktualizacji ramek:
collided = self.resolve_enemy_player_collisions()
if collided and not self.player.is_invulnerable:
    self.lives -= 1
    self.player.take_hit()
```

Trzech wrogów jednocześnie – minus jedno życie, nie minus trzy

`spritecollide()` bezwarunkowo usuwa wszystkie zderzające się wrogie sprite'y — ale obrażenia statku nie są odejmowane przez osobny test na więcej niż jedno życie na odświeżenie klatki, niezależnie od liczby wrogów uderzających w statek jednocześnie. Bez tej rozdzielczości trzech jednoczesnych wrogów zabrałoby trzy życia w jednej klatce, co wydawałoby się niesprawiedliwie surowe.

Tymczasowa odporność

Po trafieniu aktywowany jest krótki okres niewrażliwości – podczas niego kolizje trwają Usuń wrogów, ale nie odbieraj już życia:

fragment_invulnerability.py

```
PLAYER_INVULNERABLE_SECONDS = 1.2

@property
def is_invulnerable(self):
    return self.invulnerable_timer > 0.0

def update_timers(self, dt):
    self.invulnerable_timer = max(0.0, self.invulnerable_timer
- dt)

def take_hit(self):
    self.invulnerable_timer = PLAYER_INVULNERABLE_SECONDS
```

Realne Okno: Cienki niebieski pierścień wokół statku gracza jest wizualnym wskaźnikiem tymczasowej niewrażliwości

Prawdziwe okno — równe koło wokół statku, a nie miganie: spokojny, nieirytujący wskaźnik nietykalności.

Pierścień prosty zamiast migać

Migający sprite — popularny trik, ale przy szybkim miganiu męczy oczy. Tutaj zamiast tego — stała obręcz wokół statku, która po prostu znika, gdy `invulnerable_timer` spada do zera.

Ucieczka wroga za dolną granicę

Gra ma drugie, odrębne źródło utraty życia. Wróg uważa się za ucieczkę, tylko gdy jego sprite całkowicie przekroczy dolną krawędź pola gry (`vrag.rect.top > self.playfield.bottom`). Każdy taki wróg zostaje usunięty i kosztuje gracza jedno życie:

fragment_enemy_escapes.py

```
def resolve_enemy_escapes(self):
    escaped = [
        vrag for vrag in self.enemies
        if vrag.rect.top > self.playfield.bottom
    ]
    for vrag in escaped:
        vrag.kill()
    return len(escaped)

# w aktualizacji ramek:
self.lives -= self.resolve_enemy_escapes()
```

Limit „jednego życia na klatkę”

Escape nie jest trafieniem statku, więc tymczasowa niewrażliwość go nie blokuje. Jeśli dwóch wrogów opuści pole w tej samej aktualizacji, gra odejmuje dwa życia; jeśli nie ma już żyć, stan staje się `GAME_OVER`. To zasada celowa: wrogowie zderzający się jednocześnie tworzą jedno zdarzenie uderzenia, a każdy nietrafiony przeciwnik to osobny wynik, który liczy się dokładnie raz.

Praktyka: Obrażenia i niewrażliwość

Zeskanowanie bezpośrednio w przeglądarce – nie wymaga instalacji Pygame

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/21-16/index.html)

Jak wzrasta trudność gry

Trzy formuły niezależne od wyniku – z limitem od góry i dołu, bez ostrych skoków.

Stała trudność szybko staje się albo zbyt łatwa, albo zbyt trudna. Ostateczna wersja zwiększa trudność stopniowo, przez trzy niezależne czyste funkcje od aktualnego wyniku.

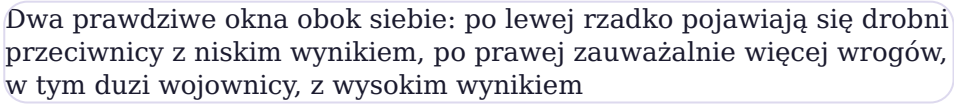
fragment_difficulty.py

```
def interval_poyavleniya_vraga(score):
    return max(MIN_SPAWN_INTERVAL, BASE_SPAWN_INTERVAL - score
* SPAWN_INTERVAL_SCORE_FACTOR)

def mnozhitel_skorosti_vraga(score):
    return 1.0 + min(MAX_ENEMY_SPEED_BONUS, score *
ENEMY_SPEED_SCORE_FACTOR)

def veroyatnost_istrebitelya(score):
    return min(MAX_FIGHTER_PROBABILITY, score *
FIGHTER_PROBABILITY_SCORE_FACTOR)
```

Funkcja	Rośnie wraz z wynikiem	Ograniczone do góry/dołu
interval_poyavleniya_vraga	Spadki – Wrogowie pojawiają się częściej	Nie niżej niż MIN_SPAWN_INTERVAL
mnozhitel_skorosti_vraga	Zwiększa się – przeciwnicy są szybsi	Nie wyżej niż 1,0 + MAX_ENEMY_SPEED_BONUS
veroyatnost_istrebitelya	Więcej wojowników	Nie wyżej niż MAX_FIGHTER_PROBABILITY

Dolne i górne granice nie są przypadkowe

Bez `max()` / `min()` odstęp pojawiania się wrogów przy bardzo wysokim wyniku mógłby dojść do zera lub liczby ujemnej. Wtedy `while self.spawn_timer <= 0.0:` w sekcji 21.14 nigdy by się nie skończyło: dodanie niedodatniego przedziału nie mogło podnieść timera powyżej zera, a pętla generowałaby wrogów w nieskończoność na tej samej klatce. Ograniczenia chronią formuły przed ich własnym wzrostem.



Dwa prawdziwe okna obok siebie: po lewej rzadko pojawiają się drobni przeciwnicy z niskim wynikiem, po prawej zauważalnie więcej wrogów, w tym duzi wojownicy, z wysokim wynikiem

Porównanie rzeczywistego świata to ten sam wzór trudności stosowany do niskiego i wysokiego liczenia.

Liczba falowa – tylko interfejs

fragment_wave.py

```
WAVE_SCORE_STEP = 500

def nomer_volny(score):
    return 1 + score // WAVE_SCORE_STEP
```

Wave nie jest osobnym systemem z własnymi zasadami, lecz wygodnym sposobem na pokazanie Postęp gracza: raz na 500 punktów liczba fali wzrasta o HUD, choć sama trudność wzrasta Nieustannie, bez nagłych skoków w tej chwili.

> **Praktyka: Formuły złożoności**
> Zeskanowanie bezpośrednio w przeglądarce – nie wymaga instalacji Pygame
> Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/21-17/index.html)

Menu, Gra, Pauza i Koniec

Jawne wyliczanie GameStatus oraz tabeli dozwolonych przejść zamiast jednej zmiennej boole'owskiej.

Rozdział 21.7 przechowywała końcową fazę gry w jednej zmiennej `igra_okonchena` nie ma odwrotu. Ostateczna wersja wykorzystuje jawne wyliczanie, ten sam trik co `SostoyanieIgry` w sekcji 20.25:

fragment_game_status.py

```
class GameStatus(Enum):
    MENU = auto()
    PLAYING = auto()
    PAUSED = auto()
    GAME_OVER = auto()
```

Spoza stanu	W stanie	Kiedy
MENU	PLAYING	naciskał Enter
PLAYING	PAUSED	kliknęli P
PAUSED	PLAYING	znów P kliknął

Spoza stanu	W stanie	Kiedy
PLAYING	GAME_OVER	życia dobiegły końca
GAME_OVER	PLAYING	kliknął Enter (restart)

Exit (`Esc` lub zamknięcie okna) działa z dowolnego stanu — Dlatego w powyższej tabeli nie jest powiązany z konkretnym wierszem: nie jest przejściem między grą stanów, ale także samego ukończenia programu.

Trzy prawdziwe okna obok siebie: puste pole gry, ekran pauzy z przezroczystą nakładką, ekran Game Over z wynikiem

Rzeczywiste kadry trzech różnych stanów tej samej gry.

Załoga statku musi wiedzieć o stanie

Jeśli zapomnisz o statusie, zamelduj `handle_input()`, statek będzie nadal poruszać się i strzelać nawet na ekranie pauzy lub Game Over — ten sam typ błędu, który sekcja 21.7 prawie już popełniła z `igra_okonchena`.

Restart gry bez ponownego uruchomienia programu

Restart z pewnością zresetuje każdy fragment stanu przejściowego — nie tylko wynik.

Rozdział 21.7 nie miała już drogi powrotnej Game Over. Ostateczna wersja zaczyna się od razu Proces — ale restart musi zresetować każdy bit stanu przejściowego, nie tylko wynik.

fragment_start_new_game.py

```
def start_new_game(self):
    self.bullets.empty()
    self.enemies.empty()
    self.explosions.empty()
    self.player = self._make_player()
    self.score = 0
    self.lives = STARTING_LIVES
    self.spawn_timer = interval_poyavleniya_vraga(0)
    self._reset_stars()
    self.state = GameStatus.PLAYING
```

Reset	Ocalony
Pozycja statku	Nagranie sesji (high_score)
Wszystkie kule, wrogowie, eksplozje	
Hrabia i Żywoty	
Timer pojawiania się wrogów i interwał strzału	
Timer niewrażliwości	

Zapomniany pocisk z poprzedniej gry to prawdziwy błąd, a nie hipotetyczny
Jeśli `self.bullets.empty()` chybi, stare pociski z poprzedniej próby pozostaną wiszące na ekranie nowej gry – wizualnie nieszkodliwe, ale wyraźnie błędne i łatwe do przeoczenia podczas testów ręcznych, jeśli nie sprawdzisz dokładnie restartu z niepustymi grupami.

Rzeczywiste okno: czyste pole gry zaraz po ponownym uruchomieniu, wynik 0, trzy życia

Prawdziwe okno zaraz po restarcie – liczenie, życia, liczniki czasu i wszystkie grupy sprite'ów wracają do stanu wyjściowego.

Animacja eksplozji

Ten sam while- rejestrator czasu, co w rozdziale 20 — teraz do wybuchu, a nie do chodzenia.

Explosion to ten sam sprite co pocisk lub przeciwnik, ale zamiast ruchu ma własną animację z kilku klatek. Rozdział 20.23 już pokazała poprawny trik z timerem animacji — tutaj Używa się go po raz drugi, tym razem do wybuchu:

fragment_explosion.py

```
class Explosion(pygame.sprite.Sprite):
    def __init__(self, frames, center):
        super().__init__()
        self.frames = frames
        self.frame_index = 0
        self.animation_time = 0.0
        self.image = self.frames[0]
        self.rect = self.image.get_rect(center=center)

    def update(self, dt):
        self.animation_time += dt
        while self.animation_time >= EXPLOSION_FRAME_INTERVAL:
            self.animation_time -= EXPLOSION_FRAME_INTERVAL
            self.frame_index += 1
            if self.frame_index >= len(self.frames):
                self.kill()
                return
            self.image = self.frames[self.frame_index]
```

Trzy prawdziwe okna obok siebie: pocisk lecący w górę, wróg przed trafieniem, pomarańczowa eksplozja błyskająca po trafieniu

Faktyczna sekwencja: strzał, zbliżenie do wroga, wybuch w momencie trafienia.

kill() zaraz po ostatnim strzale

Jak tylko `frame_index` wychodzi poza listę klatek, sprite usuwa się ze wszystkich grup (`.kill()`) — bez tego zakończone wybuchy gromadziłyby się w `self.explosions` nieskończenie, zajmując pamięć i czas renderowania bez żadnej korzyści.

Praktyka: timer animacji eksplozji

Zeskanowanie bezpośrednio w przeglądarce – nie wymaga instalacji Pygame

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/21-20/index.html)

Dźwięki wystrzału, trafienia i eksplozji

Trzy krótkie dźwięki – pobrane raz i bezszelestnie przemijane bez karty dźwiękowej.

Trzy krótkie dźwięki uzupełniają grę: strzał, eksplozja i trafienie w statek. Wszystkie one są oryginalne, syntetyzowane bezpośrednio w Python przez wbudowany moduł `wave` (krótkie sygnały sinusoidalne i tłumiony szum), bez pojedynczego nagrania pobranego z Internetu.

fragment_play_sound.py

```
def play(self, key):
    sound = self.sounds.get(key)
    if sound is not None:
        sound.play()

# podczas strzelania:
self.assets.play("laser")

# podczas niszczenia wroga:
self.assets.play("explosion")

# przy trafieniu w statek:
self.assets.play("player_hit")
```

sound może być None — i to jest w porządku

Rozdział 21.10 już wykazała, że jeśli urządzenie audio nie jest dostępne, `AssetStore` sklepy `None` zamiast przedmiotu `Sound`. Weryfikacja `if sound is not None` przed wszystkimi `.play()` zapewnia, że gra (i automatyczne testy, sekcja 21.24) nie zawiesza się bez karty dźwiękowej – po prostu gra cicho.

Dźwięk ładuje się raz, tak jak obrazy

Rozdział 20.24 już ostrzegała: `pygame.mixer.Sound(...)` czyta plik z dysku — wywołanie go w pętli gry przy każdym strzale oznaczałoby czytanie tego samego pliku dziesiątki razy na sekundę. Tutaj ładowanie odbywa się tylko raz w `AssetStore.__init__()` i tylko `.play()` w już załadowanym obiekcie.

Gwiazdy tła i kolejność rysunku

Proceduralne gwiazdy zamiast gotowego obrazu — i wyraźny porządek warstw, by HUD zawsze pozostawał na wierzchu.

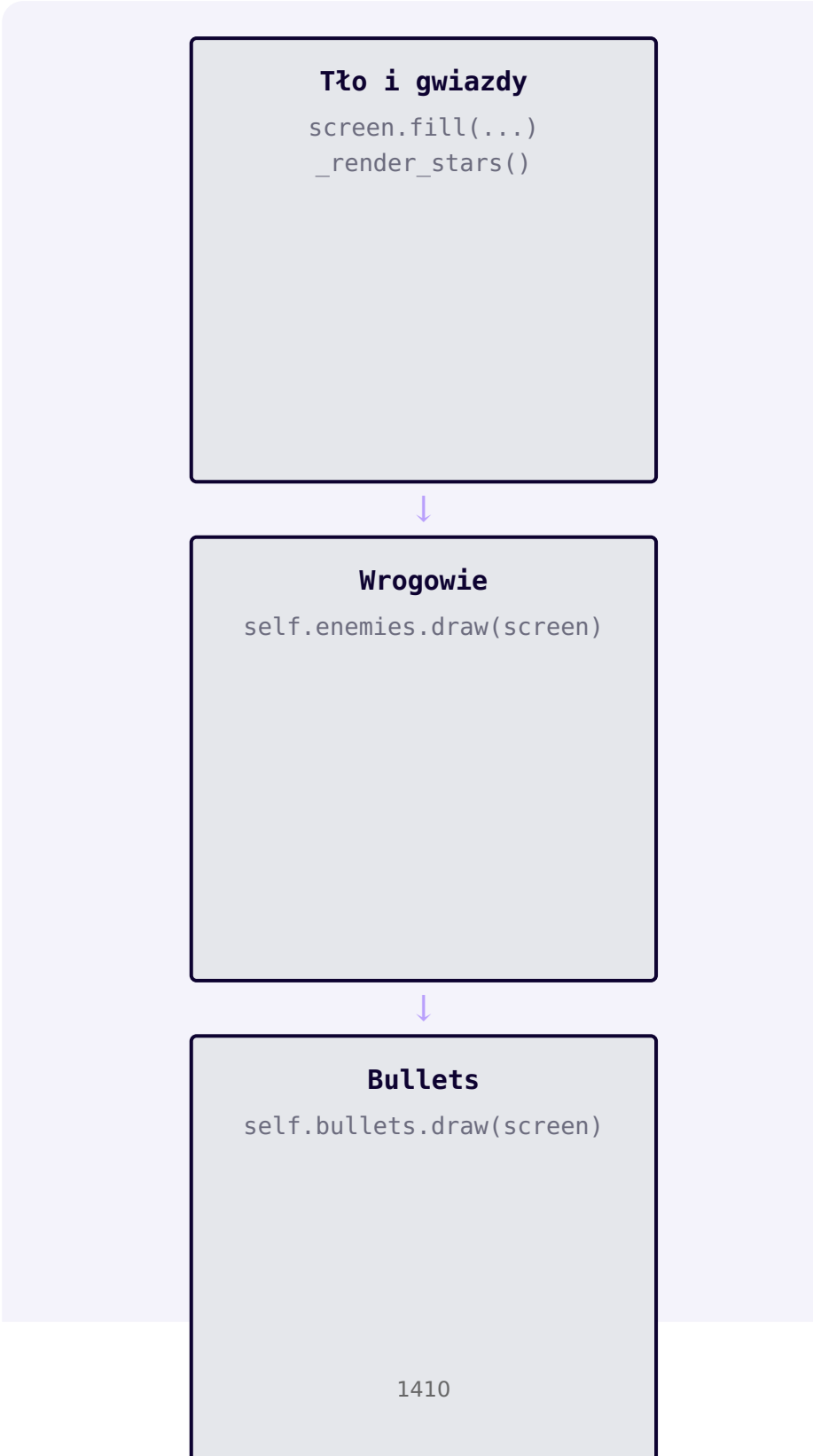
Ostatnim akcentem atmosfery jest gwiaździste tło. Zamiast gotowego obrazu rysowane są gwiazdy Proceduralnie: lista małe kropki o własnej szybkości i kolorze, z których każda pętle od góry do dołu.

fragment_stars.py

```
def _update_stars(self, dt):
    for star in self.stars:
        star.y += star.speed * dt
        if star.y > self.playfield.bottom:
            star.y = self.playfield.top
            star.x = self.rng.uniform(0, SHIRINA)
```

Różne gwiazdy mają różną prędkość (od 20 do 90 px/s), dlatego tworzą lekkie wrażenie głębi: szybkie punkty są bliżej, wolne — dalej, ten sam trik wzrokowy jak paralaksa w bardziej złożonych grach, tylko bez oddzielnych warstw.

Kolejność losowania decyduje, co jest „na górze”


Tło i gwiazdy
screen.fill(...)
_render_stars()
Wrogowie
self.enemies.draw(screen)
Bullets
self.bullets.draw(screen)

Każda kolejna warstwa jest rysowana na poprzedniej – HUD i nakładka stanu są zawsze widoczne, niezależnie od tego, co dzieje się na boisku.

Kolejność — liniowa lista, a nie losowe dopasowanie
Rozdział 20.13 (Doświadczenie mobilne) i 21.12 już podkreślają, że interfejs użytkownika nie powinien ginąć pod obiektami gry. Rozdział HUD i nakładka są tu rysowane na końcu, właśnie dlatego, że nie mogą zostać fizycznie trafione przez wroga ani eksplozję.

Jak znaleźć i naprawić błędy w strzelance

Kompendium według objawu — plus szczegółowa analiza trzech najmniej zauważalnych błędów projektu.

W trakcie tego rozdziału natknęliśmy się już na ponad dwadzieścia sposobów przypadkowego zepsucia strzelanki – od starych, technik opartych na klatkach, po subtelne błędy z podwójnym punktowaniem. Tutaj są zebrane w Jedna książka referencyjna.

Objaw	Przyczyna	Gdzie dowiedzieć się więcej
Prędkość statku zależy od FPS	Ruch jest ustawiony na piksele na klatkę, a nie px/s przez dt Funkcje	21.11
Zwolnione tempo w ogóle się nie porusza	Pozycja jest przechowywana tylko w Rect (liczbach całkowitych) – część ułamkowa jest tracona	21.11
Statek diagonalny jest szybszy niż linia prosta	Wektor kierunku nie jest znormalizowany	21.11
Statek wylatuje z pola gry	Zapomniano clamp po X i/lub Y w Player.move()	21.11
Statek przepływa pod HUD	Granice ruchu to całe okno, a nie self.playfield	21.12
Spacja nagrywa każdą renderowaną klatkę	Brak sprawdzenia fire_cooldown przed strzałem	21.13
Szybkostrzelność zależy od FPS	Interwał między ujęciami mierzony jest w klatkach (-= 1), a nie sekundach (-= dt)	21.13

Objaw	Przyczyna	Gdzie dowiedzieć się więcej
Po długiej pauzie debuggera — salwa z dziesiątek kul	Interwał Shot jest wprowadzany po while zamiast if	21.13
Pociski lecą wiecznie, gra zwalnia coraz bardziej z każdą minutą	Bullet nie wycofuje się z pola gry	21.4 (usuwanie pocisków zza ekranu)
Pojawienie się wrogów zależy od FPS	Timer pojawiania się wrogów liczy klatki, a nie sekundy	21.14
Po osłabieniu FPS planowani przeciwnicy „znikają”	Timer pojawiania się wrogów używa if zamiast while	21.14
Wróg pojawia się w połowie krawędzi ekranu	x_poyavleniya_vraga nie uwzględnia szerokości przeciwnika	21.14
Jeden wróg jest dwukrotnie oznaczany jako zniszczony	Lista zmienia się podczas brutalnej siły (jak w sekcji 21.6)	21.15
Jednemu wrogowi nalicza się punkty dwukrotnie	Wszystkie pary pocisków-przeciwników są liczone, a nie wiele unikalnych przeciwników	21.15
Trzech jednoczesnych wrogów odbiera trzy życia	Obrażenia są przypisywane na każdą kolizję zamiast raz na klatkę	21.16
Nietykalność tyka po zatrzymaniu	Aktualizacja timera nieśmiertelności poza update_playing() bez sprawdzania state	21.16, 21.18
Na ekranie Game Over statek wciąż się porusza	handle_input() nie sprawdza self.state przed obsługą kluczy	21.18

Objaw	Przyczyna	Gdzie dowiedzieć się więcej
Po restarcie pozostają stare pociski lub przeciwnicy	start_new_game() nie czyści wszystkich grup sprite’ów	21.19
Po restarcie wróg pojawia się natychmiast	spawn_timer nie zostaje zresetowany do początkowego interwału	21.19
Animacja eksplozji „połyka”	Explosion.update() używa if zamiast while	21.20
Gra zwalnia przed każdym strzałem lub eksplozją	Obraz lub dźwięk jest ładowany wewnątrz pętli gry, a nie w AssetStore	21.10, 21.21
Gra zawiesza się bez karty dźwiękowej	Sound.play() jest wywoływany bez sprawdzania None	21.21
HUD zablokowane przez wroga lub eksplozję	Kolejność renderowania nie obejmuje HUD i nakłada się na ostatnie warstwy	21.22
rect i position z czasem „rozchodzą się”	Rect zaktualizowany bezpośrednio, a nie odbudowany od position	21.11
Test kolizji przechodzi i zawiesza się bez zmian kodu	Random jest używany bezpośrednio, bez przekazywania seed	21.24

Wbudowany nakładka debugowania

Naciśnij F3włączyć lub wyłączyć nakładkę debugowania w finalnej grze. On wyznacza czerwone granice `Rect` statek, wrogowie i kule, i niżej pokazuje rzeczywistą FPS. W ten sposób można zobaczyć prawdziwe hitboxy podczas parsowania kolizji i Rozróżnić spadki liczby klatek spowodowane błędem w obliczaniu ruchu przez `dt`. Nakładka zmienia tylko renderowanie: stan świata gry, Timery i kolizje pozostają takie same.

Przyjrzyjmy się bliżej trzem najbardziej niepozornym błędom

Ruch subpikselowy jest zanikany na wolnych obiektach

Objaw: obiekt o odległości mniejszej niż jeden piksel na klatkę w ogóle się nie przesuwa. **Wersja zepsuta:** `self.rect.x += self.speed * dt` – Rect przechowuje tylko liczby całkowite, a ułamkowy przyrost jest zaokrąglany do zera na każdej klatce osobno. **Dlaczego:** zaokrąglenie odbywa się za każdym razem, a nie raz na końcu. **Poprawka:** kumulować ruch w `self.position` (Vector2, Rozdział 21.11), oraz `rect.center` z niego złożyć ponownie `round()` każda klatka. **Zasada:** pozycja zawsze jest float, Rect jest jej pochodną, nigdy odwrotnie.

Jeden wróg liczył dwa razy

Objaw: wynik czasem skacze o 200 zamiast 100 dla jednego harcerza. **Wersja zepsuta:** `for pulya, vragi in collisions.items(): schet += sum(v.points for v in vragi)` to suma wszystkich par pocisków-przeciwników. **Dlaczego:** jeśli dwie kule trafią tego samego wroga w jednej aktualizacji, zostanie ona uwzględniona w obu listach punktów. **Poprawka:** zbierać wszystkich trafionych wrogów `set` przed punktacją (Rozdział 21.15). **Zasada:** liczyć punkty za unikalnie zniszczone obiekty, a nie za liczbę kolizji.

Trzech wrogów zabiera jednocześnie trzy życia

Objaw: bezpośrednie zderzenie z grupą wrogów natychmiast przenosi grę na Game Over, choć były trzy życia. **Wersja zepsuta:** `for vrag in hit: self.lives -= 1` - Obrażenia w cyklu od zderzających się wrogów. **Dlaczego:** cykl odpisuje życie każdego wroga, a nie samego samego zderzenia. **Poprawka:** sprawdzić „czy była przynajmniej jedna kolizja” **Zasada:** obrażenia mierzone są przez zdarzenie „dotknięte”

Praktyka: Znajdź i napraw błąd

Pygame otwiera natywne okno Python – uruchomione lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/21-08/index.html)

Jak uczynić grę wygodną do testów automatycznych

Czyste funkcje, Game(rng=...) a uruchamianie bez okna to gra, którą można sprawdzić automatycznie.

Gry wielkości tego projektu nie da się sprawdzić tylko ręcznym uruchomieniem — zbyt dużo kombinacji stanów. Ostateczna wersja napisana jest tak, aby można ją było testować automatycznie, bez żadnego rzeczywistego okna.

Funkcje czyste: logika bez Pygame

Formuły trudności (rozdział 21.17), współrzędna pojawienia się wroga (rozdział 21.14) i liczenie punktów (rozdział 21.15) — zwykłe funkcje od liczb i list, bez odwoływania się do `self` lub ekran:

fragment_pure_functions.py

```
def ochki_za_unichtozhennyh(vragi):
    return sum(vrag.points for vrag in vragi)

# Czysta funkcja nie potrzebuje prawdziwego Enemy ze sprite'em
i obrazkiem —
# każdy obiekt posiadający atrybut points: jest wystarczający
class ZaglushkaVraga:
    def __init__(self, points):
        self.points = points

assert ochki_za_unichtozhennyh([]) == 0
assert ochki_za_unichtozhennyh(
    [ZaglushkaVraga(points=100), ZaglushkaVraga(points=200)]
) == 300
```

To nie obejście reguł, ale bezpośredni skutek tego, że `ochki_za_unichtozhennyh` w ogóle nie sprawdza, jakiej klasy są przekazane obiekty — potrzebuje tylko atrybutu `.points` dla wszystkich. Taka funkcja może być testowana przez każdego obiekt z tym atrybutem, aż do prostego stubu, bez rozpoczynania Pygame i bez tworzenia prawdziwi wrogowie.

Sterowany przypadek: Game(rng=...)

Gra przyjmuje generator liczb losowych jako jawny parametr, a nie odwołuje się do globalnego `random` bezpośrednio:

fragment_rng_injection.py

```
class Game:
    def __init__(self, *, rng=None):
        self.rng = rng if rng is not None else random.Random()

# w prawdziwej grze to typowa nieprzewidywalna losowość:
game = Game()

# w testach i przy tworzeniu zrzutów ekranu — powtarzalne:
game = Game(rng=random.Random(42))
```

Ten sam seed, ten sam rezultat

`random.Random(42)` z ustalonym ziarnem (seed) daje tę samą sekwencję losowych liczb przy każdym starcie – dzięki temu pojawienie się wrogów, ich typ i pozycja stają się powtarzalne: wygodne zarówno w testach, jak i scenariuszach deterministycznych – na przykład dla zrzutów ekranu z konkretnego, wcześniej znanego stanu gry.

Uruchamianie bez okien: Pusty sterownik SDL- (headless)

Ten sam trik co we wszystkich testach rozdziału 20: zmienne środowiskowe przełączają Pygame na "pusty" (dummy) sterownik wideo i audio SDL – okno fizycznie nie pojawia się nigdzie indziej, lecz całe Logika (Surface, Rect, kolizje, dźwięk) działa naprawdę. Taki start bez Realne okno w terminologii angielskiej nazywa się headless – "bez głowy", czyli bez ekran i brak interfejsu użytkownika.

fragment_headless_env.py

```
import os

os.environ.setdefault("SDL_VIDEODRIVER", "dummy")
os.environ.setdefault("SDL_AUDIODRIVER", "dummy")

import space_shooter as ss

game = ss.Game()   # prawdziwa Game, brak prawdziwego okna
```

Nie wymaga ciężkiej pracy przy prostym imporcie modułu

`space_shooter.py` nie powoduje `pygame.init()` lub `Game().run()` na poziomie modułu — cała inicjalizacja odbywa się wewnętrznie `Game.__init__()`, ale nieskończona pętla występuje tylko wewnątrz `if __name__ == "__main__":`. W przeciwnym razie to po prostu `import space_shooter` w teście prowadziłby prawdziwą grę.

Składanie ostatecznej wersji gry

Wszystkie sekcje od 21.9 do 21.24 są w jednym czytelnym, gotowym do uruchomienia pliku.

Wszystkie sekcje 21.9–21.24 są zebrane w jednym pliku — `projects/pygame/space-shooter/space_shooter.py`. Nie dzieli się na Wiele modułów: projekt jest na tyle kompaktowy, że pojedynczy czytelny plik jest czytelniejszy niż rozproszonych kilkunastu małych.

projects/pygame/space-shooter/space_shooter.py

Rzeczywisty stan działającej gry w konkretnym momencie — to, co faktycznie jest przechowywane w pamięci między klatkami.

Co się zmieniło w porównaniu z pierwszą wersją roboczą (21.8)

	Rozdział 21.8 (checkpoint)	Finalna wersja (21.25)
Organizacja Kodeksu	Funkcje i słowniki/listy na Rect	Zajęcia Game/Player/Bullet/Enemy/Explosion oprócz pygame.sprite.Group
Wniosek	px/s + dt współrzędne ułamkowe są przechowywane oddzielnie od Rect	px/s + dt przez Vector2 wewnątrz klas Player/Bullet/Enemy
Grafika	Wypełnione prostokąty	Posiadanie PNG-sprite'ów
Sound	Tak	Trzy oryginalne syntetyczne brzmienia
Koniec gry	Jedna kolizja, natychmiastowy koniec	System życia nieśmiertelności
Złożoność	Stała	Płynnie rośnie wraz z muzyką
Stany	Jeden booleowski zmienna	GameStatus: MENU/PLAYING/PAUSED/GAME_OVER
Restart	Restartuj tylko Python	Enter na ekranie Game Over
Eksplozje	Tak	Animacja wieloklatkowa

	Rozdział 21.8 (checkpoint)	Finalna wersja (21.25)
Testy	7 testów regresyjnych timerów i ruchu (rozdział 21.3)	38 testów klasy Game i funkcji pomocniczych, w tym niezależność FPS-

Realne okno: Bogata rama rozgrywki – niebieski statek, kilku wrogów i pocisków, gwiaździste tło, czytelny HUD z wynikiem 640

Właściwa rama ostatecznej wersji to ten sam statek co w sekcji 21.2, ale rozwinął się w pełnoprawną grę.

Challenge
Kolejny typ wroga
Dodaj trzeci EnemySpec — na przykład „bomber”

Challenge
Bonusy na planszy gry
Dodaj klasę PowerUp: dobieranego obiektu, tymczasowo zmniejszającego FIRE_INTERVAL po kolizji z statkiem — użyj tej samej metody timera, co dla nietykalności.

Praktyka: Zbuduj i wydaj finalną grę
Pygame otwiera natywne okno Python – uruchomione lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/21-25/index.html)

Co zbudowaliśmy i jak dalej rozwijać grę

Od pierwszego okna w rozdziale 20 do testowanej, omawianej gry z własną grafiką.

Space shooter dobiegł końca — a wraz z nim zamykamy drugi ważny temat kursu: Pełnoprawny Pygame rozwoju gier, od pierwszego okna w Rozdziale 20 po testowalny, dubbingowy Gry z niestandardową grafiką są już dostępne.

Co zbudowaliśmy w tym rozdziale

- Pełnoprawna gra z pięcioma klasami: `Game`, `Player`, `Bullet`, `Enemy`, `Explosion` zamiast zmiennych globalnych i list.
- Precyzyjne poruszanie się `Vector2` i delta time — ta sama zasada z rozdziału 20, zastosowana do bieżącego projektu z wieloma typami obiektów naraz.
- Timery rozróżniające symulację offline (pojawianie się wrogów, animacje – zachowanie pozostałego czasu poza while) interwałem oraz reakcją na dane gracza (odstęp między strzałami to nie więcej niż jedno zdarzenie na klatkę).
- Kolizje przez `pygame.sprite.groupcollide()` oraz `spritecollide()`, z ochroną przed podwójnym punktowaniem i utratą wielu żyć w jednym jednoczesnym zderzeniu.
- Jawne stany gry (`GameStatus`) oraz kontrakt restartowy, który naprawdę resetuje cały stan przejściowy — nie tylko wynik.
- Własna grafika i dźwięk, załadowane raz, oraz grę, którą można automatycznie sprawdzić — poprzez wdrożenie generatora liczb losowych i uruchomienie bez okna pod „pustym” sterownikiem SDL- (headless).

Dokąd rozwijać projekt dalej

Pomysły na rozwój osobisty

WIĘCEJ TYPÓW PRZECIWNIKÓW

```
Własne wzorce ruchu (zygzak, celowanie w
statek)
Niestandardowe sprite'y i okulary
```

BONUSY I POWER-UPY

Tymczasowe efekty dzięki temu samemu ruchowi z timerem co niewrażliwość
Tarcza, Podwójny Strzał, Dodatkowe Życie

SZEFOWIE

Duży wróg z własnym zdrowiem (nie jedno trafienie)
Własny zestaw ataków

UTRZYMANIE REKORDU

Pisanie high_score do pliku między kolejnymi przejściami (Rozdział 15)
Nie tylko podczas jednej sesji

Kolejny rozdział zmienia kierunek z lokalnych gier na Pygame na rozwój stron internetowych na Python. Zasady nauczone tutaj to wyraźny podział obowiązków między klasami, wyraźny stany, czyli testowany kod, przydadzą się tam, tylko w innym środowisku.

ROZDZIAŁ 22 · STR. 1423

Tworzenie stron internetowych z Python

Wprowadzenie do struktury aplikacji internetowych: przeglądarka i serwer, HTTP i HTTPS, HTML/CSS/JavaScript, Flask, bazy danych, SQL, JSON i API. Bardziej szczegółowe studium kontynuowane jest w osobnym kursie z zakresu tworzenia stron internetowych na temat Python.

Podstawy tworzenia stron internetowych: klient, serwer i Python

Aplikacja webowa to dialog między klientem a serwerem. Ustalmy, kto jest kim.

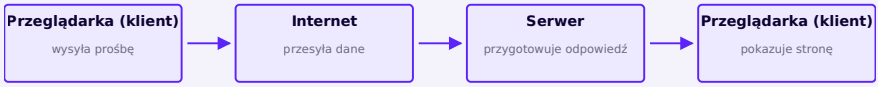
Aplikacja webowa zazwyczaj składa się z dwóch stron współdziałających: klienta i serwera. Klientem najczęściej jest przeglądarka. Wysyła HTTP- żądań i wyświetla otrzymane odpowiedzi. Backend przyjmuje żądania, wykonuje logikę aplikacji, jeśli jest to konieczne Uzyskuje dostęp do bazy danych i generuje odpowiedź.

Python jest często używany do programowania backendu aplikacji webowej. W tym Aplikacja TChapter Server jest napisana w Python przy użyciu Flask.

> **A co z Python w przeglądarce?**
> To nie znaczy, że Python nie może działać w przeglądarce w zasadzie. Istnieją środowiska oparte na WebAssembly, takie jak Pyodide — które stanowią podstawę interaktywnej praktyki tej książki — które pozwalają Python wykonywać po stronie klienta. Jednak jest to inny model wykonania; tutaj badane jest klasyczne programowanie stron serwerowych.

Klient i serwer

Klient — program, który wysyła zapytanie i otrzymuje na nie odpowiedź. **Serwer** to program, który przyjmuje i odpowiada na żądania. Word „serwer”



Jeden cykl: zapytanie leci do serwera, odpowiedź wraca i jest wyświetlana w przeglądarce

Ta wymiana żądań i odpowiedzi podlega ogólnej zasadzie, zwanej protokołem **HTTP** (HyperText Transfer Protocol, protokół transferu hipertekstu). Więcej informacji można znaleźć w sekcjach 22.7-22.10; tutaj wystarczy wiedzieć, że przeglądarka i serwer Wymieniać się wiadomościami według tych samych zasad, niezależnie od tego, kto je napisał.

co robi przeglądarka

Przeglądarka to program kliencki, który pod adresem użytkownika buduje i wysyła HTTP-żądanie, otrzymuje HTTP-odpowiedź i wyświetla ją: analizuje HTML, stosuje CSS, wykonuje JavaScript, pokazuje obrazy i inne zasoby strony.

Co robi aplikacja serwerowa

Aplikacja serwera akceptuje żądanie, decyduje, co zrobić (otwórz lista zadania, zapisz nowy rekord, zwróć dane w JSON) formacie, jeśli to konieczne, do bazy danych i generuje odpowiedź — najczęściej stronę lub dane HTML-.

Gdzie Python używany

W typowej aplikacji webowej Python działa po stronie serwera: kod działa na a gotowy wynik jest wysyłany do przeglądarki. Dalej w tym rozdziale tak to wygląda — Python jest wykonywany przez serwer zbudowany na Flask.

Internet i World Wide Web

Internet to siatka, która fizycznie łączy komputery ze sobą i wie, jak przesyłać dane między nimi. **World Wide Web** (World Wide Web, Web) to jeden ze sposobów wykorzystania tej sieci: strony, połączenia między

nimi, protokół HTTP. Inne usługi również działają w tym samym Internecie – poczta, komunikatory, gry online, - Internet to tylko jedna z nich, choć najbardziej widoczna.

Adres strony internetowej: domena i adres IP-

Aby poinformować przeglądarkę, do którego serwera ma się udać, strona ma **domena** — Nazwa gatunkowa `python.org` to łatwo zapamiętać. Ale Komputery w Internecie odnajdują się nawzajem za pomocą adresów numerycznych — **IP-adresy**. Specjalna usługa, DNS, zamienia domenę w adres IP- przed wysyłanie żądania. Cała ta podróż jest szczegółowo omówiona w sekcji 22.7 oraz urządzenie Sam adres znajduje się w sekcji 22.8.

Frontend i backend

Co użytkownik widzi i z czym wchodzi w interakcje w przeglądarce — HTML, CSS, JavaScript — są nazywane **frontend** (front — „przed”). Program, który działa na serwerze i przygotowuje odpowiedzi, nazywa się **backendem** (back – „tył”

Strona statyczna i dynamiczna aplikacja

Najprostsza strona to po prostu pliki `.html` leżąc na Serwer: Serwer zwraca je bez zmian, to jest **statyczny** strona. Jeśli odpowiedź jest za każdym razem tworzona na nowo — z uwzględnieniem tego, kto pytał, co jest w bazie danych, co człowiek właśnie wysłał w formularzu — taką aplikację nazywa się **dynamika**. Strona internetowa, którą stworzysz w tym rozdziale na Flasku, — dynamiczna: buduje stronę HTML z danych bezpośrednio podczas zapytania.

Poniższe trzy sekcje zawierają krótkie wprowadzenie do języków front-endowych: HTML (Rozdział 22.2), CSS (sekcja 22.3) oraz JavaScript (sekcja 22.4). Następnie pierwszy backend na Python z Biblioteka **Flask** (Rozdział 22.5).

HTML: struktura strony internetowej

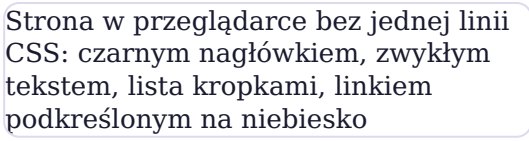
HTML jest językiem znaczników, a nie programowaniem: opisuje strukturę, a nie zachowanie.

HTML (HyperText Markup Language - Język znaczników hipertekstowych) opisuje nie zachowanie ani wygląd, lecz *struktura* stron: gdzie jest tytuł, gdzie jest akapitem tekstu, gdzie jest obrazem,

gdzie jest linkiem. HTML jest językiem znaczników, a nie językiem Programowanie: nie ma warunków, pętli ani zmiennych, jedynie opis tego, co i W jakiej kolejności pokazać.

stranica.html

```
<!doctype html>
<html lang="ru">
<head>
  <meta charset="utf-8">
  <title>Моя первая страница</title>
</head>
<body>
  <h1>Привет, мир!</h1>
  <p>Это мой первый сайт на HTML.</p>
  <ul>
    <li>Учу Python</li>
    <li>Учу HTML</li>
  </ul>
  <a href="https://python.org">Официальный сайт Python</a>
</body>
</html>
```



Ten sam plik otwarty w przeglądarce, bez jednej reguły CSS — przeglądarka domyślnie wybiera czcionkę, wcięcia i kolor linku.

Tagi zazwyczaj występują parami

`<h1>` to otwierający tag tytułowy, `</h1>` (ze ukośnikiem) to zamykająca się forma, a razem z zawartością między nimi tworzą jedno **element** strony. Niektóre elementy, takie jak `<img>`, brak zamykającego taga i zagnieżdżonej zawartości — nazywa się to **void- element**: Wszystkie niezbędne informacje są przekazywane przez atrybuty.

Atrybuty — dodatkowe właściwości tagu

W oznaczeniu otwierającym możesz określić **atrybuty** — pary `имя="значение"`. Na linku `<a>` atrybut `href` ustawia adres, blisko zdjęcia `<img>` — `src` określa plik, oraz `alt` określa alternatywę

obrazową tekstową, która odpowiada Jej znaczenie na stronie: jest używana przez czytniki ekranu i wyświetlana, Jeśli plik nie zostanie przesłany:

atributy.html

```
<img src="kotik.jpg" alt="Рыжий кот спит на подоконнике">
```

Częste tagi

Tag	Co oznacza
`<h1>` … `<h6>`	Nagłówki od najważniejszych (h1) po najmniejsze (h6)
`<p>`	Akapit tekstu
`<ul>` / `<li>`	Lista i jej pozycje
`<a href="…">`	Link do innej strony
`<img src="…" alt="…">`	Obraz z opisem tekstowym
`<label>` / `<input>`	Pole formularza i jego sygnatura

Nagłówki — to nie tylko duża czcionka

Poziomy `h1` - `h6` ustalić logiczną strukturę strony, podobnie jak spis treści książki: `h1` jest głównym nagłówkiem, zwykle ma go na stronie, oraz `h2` są nagłówki sekcji wewnątrz. Czytniki ekranu i wyszukiwarki korzystają z tej struktury, a nie z tego, czy coś jest narysowane dużym drukiem.

Formy

Formularz zbiera wpisane przez użytkownika informacje i przesyła je na serwer:

forma.html

```
<form>
  <label for="imya">Ваше имя</label>
  <input type="text" id="imya" name="imya">
  <button type="submit">Отправить</button>
</form>
```

`<label for="imya">` związane z `<input id="imya">` przez dopasowany identyfikator — kliknij by caption umieszcza kursor w polu, a czytnik ekranu odczytuje podpis, gdy Pole jest skupiane. Co dzieje się z tymi danymi na serwerze, znajduje się w sekcji 22.13.

Praktyka: HTML struktura i style CSS
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/22-02/index.html)

CSS: projektowanie i układ elementów strony internetowej

CSS określa wygląd i lokalizację elementów HTML- dokumentu.

CSS (Cascading Style Sheets - Kaskadowe arkusze stylów) to język tabelowy stylów, zaprojektowanych do opisywania wyglądu i układu elementów dokumentu, Napisane HTML lub innym językiem znaczników. Z pomocą CSS ustawionych kolorów, czcionek, rozmiary, wypełnienia, ramki, układ elementów oraz dostosowywanie strony do różnych rozmiarów ekran.

HTML ustala strukturę dokumentu i CSS determinuje, jak ta struktura pojawia się na ekran: bez własnych stylów przeglądarka korzysta ze standardowego projektu — zwykłej czcionki, tekst, minimalne oddzielenie. CSS wybiera elementy strony i ustawia ich właściwości:

stili.css

```
body {
  font-family: sans-serif;
  background: #f7f7fb;
  color: #1a1a2e;
}

h1 {
  color: #4a2fbd;
}

p {
  line-height: 1.6;
}
```

Każda zasada CSS — to **selektor** (które elementy reguła wybiera: tag, klasa lub identyfikator) oraz para nawiasów **Nieruchomość: Wartość**. Podłączenie pliku CSS- do strony HTML- można wykonać jednym wierszem wewnątrz `<head>`:

podklyuchenie.html

```
<link rel="stylesheet" href="stili.css">
```

Ta sama strona, CSS: jasnofioletowym tłem, ciemnofioletowym nagłówkiem i zwiększoną odstępami między wierszami w akapicie

Dokładnie ten sam HTML z sekcji 22.2, ale z podłączonym stili.css: tłem, kolor nagłówka i odstęp między wierszami teraz są ustawione jawnie, a nie zostawione do uznania przeglądarki.

Klasy — stylizujemy nie wszystkie tagi po kolei

Selektor według tagu (`p { }`) maluje wszystkie akapity naraz. Aby wyróżnić tylko niektóre elementy, nadaje im atrybut `class` , i w CSS odnieść się do tego przez kropkę:

klass.html

```
<p class="preduprezhdenie">Осторожно, ступенька!</p>
```

klass.css

```
.preduprezhdenie {
  color: #b3261e;
  font-weight: 700;
}
```

Kaskada i dziedziczenie to dwa różne mechanizmy

Kaskada decyduje, którą z kilku konkurujących reguł zastosować do tego samego elementu: w przypadku uproszczonym, gdy pozostałe warunki są równe, bardziej szczegółowy selektor wygrywa mniej szczegółowy, a jeśli szczegółowość jest taka sama, decyduje o kolejności reguł w pliku. Jest to część algorytmu kaskadowego, a nie kaskady jako całości. **Dziedzictwo** jest osobnym mechanizmem: niektóre właściwości (na przykład, `font-family`) przejmuje domyślny element od rodzica, jeśli nic nie jest dla niego wyraźnie określone.

Model blokowy

Każdy element ma swoją zawartość i jest wcięty w ramce wokół niego (`padding`), sama ramka (`border`) oraz wcięcie od zewnątrz do sąsiednich elementów (`margin`):

blochnaya_model.css

```
.kartochka {
  padding: 16px;
  border: 1px solid #ccc;
  margin: 12px;
}
```

Układ elementów: display i flexbox

Nieruchomości `display` określa, czy pierwiastek zachowuje się jak blok Pełnowymiarowa (`block` jako `<p>` oraz `<div>`) lub jako część stringa (`inline` jako `<a>` oraz `<span>`). Ustawiać wiele pozycji z w równych odstępach, najczęściej stosowanych `flexbox`:

flexbox.css

```
.panel {
  display: flex;
  gap: 12px;
  align-items: center;
}
```

Responsywny układ: jedna strona, różne ekrany

Zapytanie medialne (media query) stosuje część reguł tylko przy określonych warunkach — na przykład tylko na wąskich ekranach:

adaptivnost.css

```
@media (max-width: 480px) {
  .panel {
    flex-direction: column;
  }
}
```

Dzięki temu ta sama strona może ustawiać elementy w rzędzie na szerokim ekranie i w kolumnie — na wąskim, bez osobnej strony mobilnej. Końcowy projekt rozdziału (rozdział 22.35) używa właśnie tego zabiegu.

Oficjalna dokumentacja: MDN — CSS.

Praktyka: HTML struktura i style CSS
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/22-02/index.html)

JavaScript: programowanie w przeglądarce

Język programowania odpowiedzialny za zachowanie strony w przeglądarce: reakcję na działania użytkownika i zmiany w treści.

JavaScript (JS) — język programowania, szeroko stosowany w web-developmencie. Rdzeń języka jest standaryzowany przez specyfikację ECMAScript (ECMA-262). W przeglądarce JavaScript realizuje logikę programową strony: obsługuje zdarzenia, pracuje z DOM, zmienia zawartość strony i może wysyłać zapytania sieciowe.

> **JavaScript działa nie tylko w przeglądarce**
> JavaScript nie jest używany tylko w przeglądarkach. Działa także w środowiskach serwerowych i innych środowiskach uruchomieniowych, takich jak Node.js. Dlatego JavaScript nie jest tylko „językiem dla przeglądarki”

Główne właściwości JavaScript

Jaki język to JavaScript

PISANIE I PAMIĘĆ

Dynamiczne typowanie
Automatyczne zarządzanie pamięcią (garbage collector)

MODEL OBIEKTOWY

Model prototypowy obiektów
Funkcje – wartości pierwszorzędne

STYLE PROGRAMOWANIA

Styl Imperatywny
Styl obiektowy
Styl funkcjonalny

ECMAScript, DOM i Web API to nie to samo

ECMAScript to standard opisujący sam język: składnię, typy, operatory, funkcje. **DOM** (Document Object Model) i funkcje takie jak `fetch()` nie są zawarte w ECMAScript są oddzielnymi interfejsami programistycznymi (Web API), które przeglądarka udostępnia kodowi JavaScript oprócz samego języka.

JavaScript dodaje stronie *zachowanie*: reakcja na kliknięcia, zmiana zawartości bez przeładowania strony, sprawdzanie formularza przed wysłaniem. JavaScript nie jest potrzebny na każdej stronie: prosty artykuł lub dokument dobrze działają na jednym HTML (plus CSS do formatowania) — przeglądarka wyświetla taką stronę bez ani jednej linijki JavaScript. JavaScript potrzebny jest właśnie tam, gdzie strona musi reagować na działania użytkownika bez nowego żądania do serwera.

DOM jest widokiem strony, z którym pracuje JavaScript

Gdy przeglądarka analizuje HTML, tworzy drzewo obiektów w pamięci z tagów — DOM (Document Object Model). JavaScript nie edytuje tekstu źródłowego pliku HTML- — znajduje pożądany węzeł w tym drzewie i go zmienia:

povedenie.html

```
<button id="knopka">Нажми меня</button>
<p id="tekst">Пока ничего не произошло.</p>

<script>
  const knopka = document.getElementById("knopka");
  const tekst = document.getElementById("tekst");

  knopka.addEventListener("click", function () {
    tekst.textContent = "Кнопку нажали!";
  });
</script>
```

Strona z przyciskiem "Click me" i tekstem "Nothing happened yet." przed kliknięciem

Przed kliknięciem: Opiekun jest podpisany, ale zdarzenie jeszcze nie nastąpiło.

Ta sama strona zaraz po kliknięciu przycisku: tekst zmienił się na „Przycisk naciśnięty!”

Po kliknięciu: ta sama strona, bez przeładowania — JavaScript po prostu zastąpił zawartość węzła #tekst.

`addEventListener("click", ...)` podpisuje funkcję na **wydarzenie** — kliknij przycisk. Jest wiele wydarzeń: `"click"`, `"submit"` (złożenie formularza), `"input"` (wprowadzanie tekstu) i inne – kod jest wykonywany tylko wtedy, gdy zdarzenie jest ważne wydarzyło się.

Konstrukcje znane w innej składni

Zarówno Python, jak i JavaScript mają zmienne, funkcje, warunki i pętle. Jednak ich systemy typów, modele obiektów i czasy uruchomieniowe znacznie się różnią: na przykład JavaScript używa prototypowego modelu obiektowego i zawijanych nawiasów zamiast wcięcia. Jeśli rozumiesz Python, opanowanie podstaw JavaScript będzie znacznie łatwiejsze niż zaczynanie od zera.

Weryfikacja formularza bezpośrednio w przeglądarce

JavaScript może sprawdzić pole formularza przed wysłaniem — i od razu wyświetlić podpowiedź bez Czekam na odpowiedź serwera:

validaciya.js

```
forma.addEventListener("submit", function (event) {
  if (pole.value.trim() === "") {
    event.preventDefault();
    oshibka.textContent = "Поле не должно быть пустым";
  }
});
```

`event.preventDefault()` powstrzymuje regularne wysyłanie formularzy — strona nie zostanie ponownie załadowana, dopóki błąd nie zostanie naprawiony. Nadal ten sam sprawdzenie Musi być powtarzane na serwerze (sekcja 22.31): użytkownik może wysyłać Żądanie bez przeglądarki, omijając wszelkie JavaScript.

---

fetch() — żądanie do serwera bez ponownego ładowania strony

Funkcja `fetch()` wysyła zapytanie HTTP- z JavaScript i otrzymuje odpowiedź, nie przeładowując całej strony:

fetch.js

```
fetch("/api/tasks")
  .then(function (response) { return response.json(); })
  .then(function (data) { console.log(data); });
```

Co to jest `/api/tasks` i co dokładnie odpowiada na takie żądanie — Więcej informacji można znaleźć w sekcji 22.16.

Trzy języki razem — HTML, CSS JavaScript — tworzą właśnie „frontend”

Oficjalna dokumentacja: MDN — JavaScript, ECMA-262 (ECMAScript).

Pierwsza aplikacja webowa na Flask

Flask — lekki framework webowy do tworzenia aplikacji webowych w Python.

Flask jest lekkim frameworkiem webowym do tworzenia aplikacji webowych w tym języku Python. Zapewnia routing, obiekty żądań i odpowiedzi, pracując z szablonami Jinja oraz innych podstawowych narzędzi wymaganych przez aplikację serwerową. Sam Flask nie akceptuje Bezpośrednie połączenia sieciowe – obsługiwane przez osobny serwer (Rozdział 22.19 to wyjaśnia granicy w bardziej szczegółowym zakresie); podczas lokalnego działania Flask tymczasowo przejmuje tę rolę poprzez Wbudowany Werkzeug. instalacja serwera deweloperskiego — jak każda biblioteka:

terminal.txt

```
pip install flask
```

Najmniejszy Flask-app

app_minimal.py

```
from flask import Flask

app = Flask(__name__)

@app.route("/")
def glavnaya():
    return „Witaj, świecie!”

if __name__ == "__main__":
    app.run(debug=True)
```

@app.route(„/”
Linia powyżej funkcji to **dekorator**. `@app.route("/")` rejestruje trasę: łączy adres `/` z funkcją `glavnaya()`, który obsługuje zapytanie na ten adres i generuje odpowiedź. Ten mechanizm jest szczegółowo omówiony w sekcji 22.11.

Szablony – HTML z wyrażeniami Jinja

Zwracanie HTML bezpośrednio jako łańcuch znaków w Python jest niewygodne. Flask używa silnika szablonowego **Jinja** (sekcja 22.12 omawia go szczegółowo) — to osobny język szablonów ze swoją składnią: zwykły plik HTML-, do którego można wstawiać wyrażenia i konstrukcje sterujące Jinja w nawiasach klamrowych `{{ }}`:

templates/index.html

```
<h1>Мой список задач</h1>
<ul>
  {% for zadacha in zadachi %}
    <li>{{ zadacha }}</li>
  {% endfor %}
</ul>
```

app.py

```
from flask import Flask, render_template

app = Flask(__name__)
zadachi = [„Naucz się podstaw Python”, „Zbudować stronę na
Flask”]

@app.route("/")
def glavnaya():
    return render_template("index.html", zadachi=zadachi)
```

Adresy dynamiczne

Część adresu można zamienić w parametr trasowania — Flask przekazuje jego wartość jako argument funkcji-obslugi:

dinamicheskij_marshrut.py

```
@app.route("/privet/<imya>")
def privet(imya):
    return render_template("privet.html", imya=imya)
```

/privet/sergey → imya = „Sergey”
Otwierając adres w przeglądarce `/privet/Сергей` wejdziesz do `imya` linia `"Сергей"` to część adresu Flask przekazuje funkcji obsługiwającej jako argument.

Strona przeglądarki: nagłówek "Hey Ada!" oraz link "Powrót do listy zadań"

Projekt końcowy wykorzystuje tę ścieżkę: adres /privet/A w przeglądarce, a Flask zastępuje nazwę z URL do szablonu.

Akceptuj dane z formularza

Aby użytkownik mógł coś wysłać na serwer (np. dodać zadanie), używana jest forma HTML- i trasa przyjmująca metodę `POST` (dlaczego POST, a nie GET znajduje się w sekcji 22.13):

dobavlenie.py

```
from flask import request, redirect, url_for

@app.route("/dobavit", methods=["POST"])
def dobavit():
    novaya_zadacha = request.form.get("zadacha", "").strip()
    if novaya_zadacha:
        zadachi.append(novaya_zadacha)
    return redirect(url_for("glavnaya"))
```

Ministrona z gotową listą zadań

Trasy, szablony, formularz i statyczny plik CSS z tej sekcji zostały zebrane w gotową mini-stronę „Lista zadań”:

projects/flask/todo-app/app.py

Uruchom stronę internetową na swojej stronie

`python app.py` w terminalu wewnątrz `projects/flask/todo-app/` , następnie otwórz `http://127.0.0.1:5000/` w przeglądarce. W ten sposób uruchamia się serwer deweloperski; czym różni się od środowiska produkcyjnego — w sekcji 22.34.

Obecnie zadania są przechowywane w zwykłej liście Python i znikają po ponownym uruchomieniu programu — dlaczego to problem, wyjaśnia rozdział 22.20, a rozdziały 22.20-22.29 stopniowo zastąpią lista bazą danych SQLite.

★★ Zadanie samodzielne

Licznik zadań

Dodaj wiersz do index.html pokazujący łączną liczbę zadań: {{ zadachi| length }}.

Challenge

Usuwanie zadania

Dodaj trasę /udalit/<int:indeks>, która usuwa zadanie według jego numeru w liście i przekierowuje na stronę główną.

Praktyka: trasy, szablony i formy

Moduł flask nie jest zainstalowany w środowisku przeglądarki Pyodide – działa lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/22-05/index.html)

Wyniki pierwszego projektu internetowego

Od żądania HTTP do własnej strony na Flask — i dalsze kroki przed nami, w cyfrowym wydaniu rozdziału.

Czego nauczyliśmy się w tej części rozdziału

- Strona internetowa — to dialog według protokołu HTTP:, przeglądarka (klient) wysyła żądanie, serwer odpowiada.
- Frontend (to, co widzi użytkownik) składa się z trzech języków: HTML (struktura), CSS (projektowanie) oraz JavaScript (zachowanie).
- Backend to program na serwerze, który przygotowuje odpowiedzi. Na Python biblioteka Flask. jest często wykorzystywana do tego celu
- `@app.route("/адрес")` przypisuje adres funkcji, która na niego reaguje.
- Jinja szablony pozwalają wstawić Python dane bezpośrednio do HTML przez `{{ }}` oraz `{% %}`.
- Formy z POST i `request.form` umożliwiają otrzymywanie danych wprowadzonych przez użytkownika w przeglądarce.

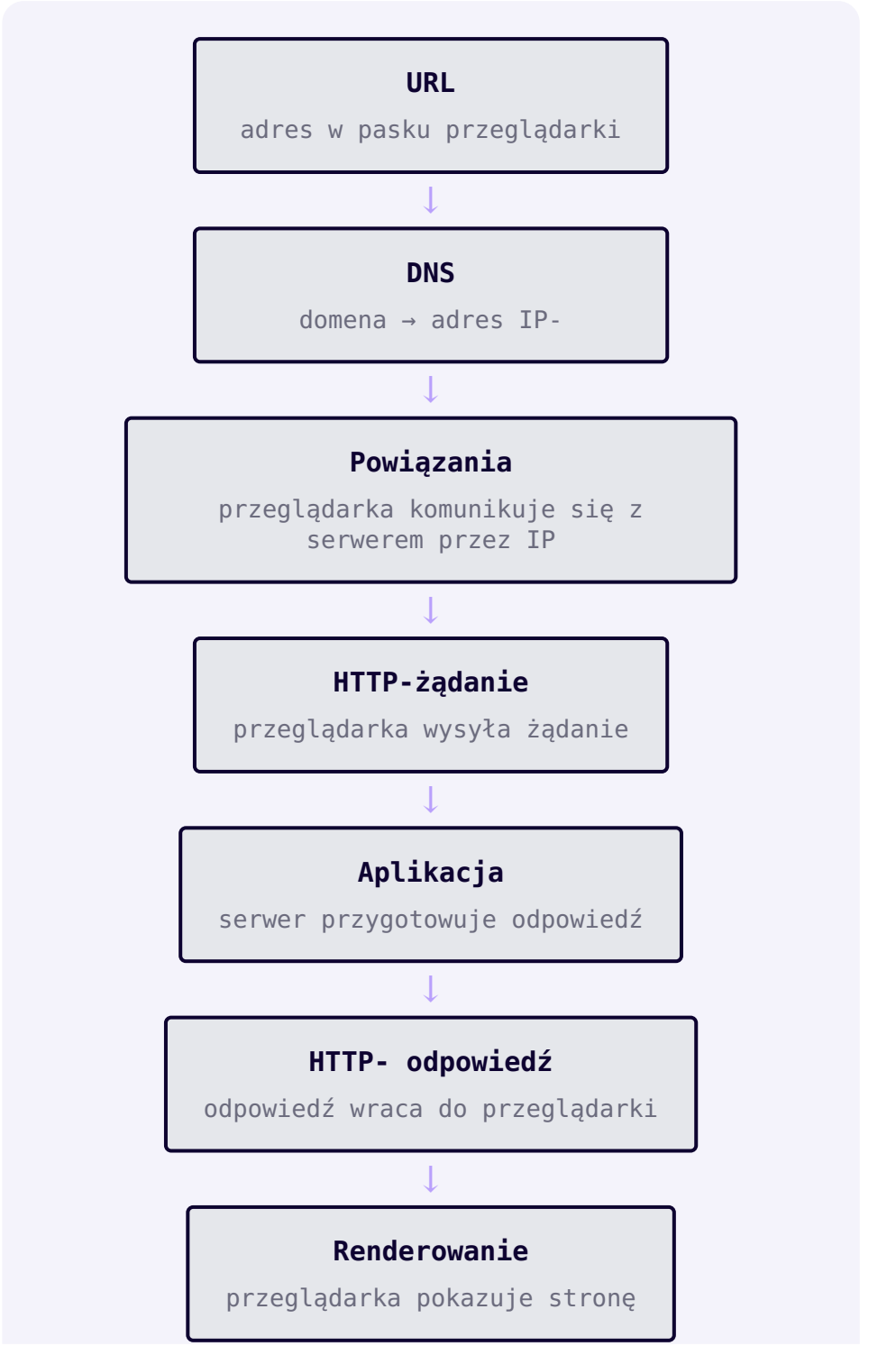
Tu kończy się historyczna sekcja drukowana rozdziału — ale nie sam rozdział. Lista zadań jest na razie przechowywana w zwykłej liście Python i znika przy każdym ponownym uruchomieniu: strona jest w pełni funkcjonalna, ale dane na niej nie przeżywają restartu programu. Wydanie cyfrowe książki kontynuuje tę samą tematykę dalej: jak działa ścieżka zapytania do serwera i z powrotem, jak są zorganizowane

HTTP i HTTPS, jak połączyć Flask z bazą danych i jak sprawdzić gotową aplikację testami — zaczynając od sekcji 22.7 w menu bocznym.

Jak przeglądarka pobiera stronę internetową

Od naciśnięcia Enter w pasku adresowym do ukończonej strony na ekranie, ścieżka pojedynczego żądania.

Rozdział 22.1 pokazała szerszy obraz: przeglądarka wysyła żądanie, serwer odpowiada. Teraz Rozłóżmy tę ścieżkę krok po kroku — co dokładnie dzieje się między naciśnięciem Enter w adresie i strona pojawia się na ekranie.


URL
adres w pasku przeglądarki
DNS
domena → adres IP-
Powiązania
przeglądarka komunikuje się z serwerem przez IP
HTTP-żądanie
przeglądarka wysyła żądanie
Aplikacja
serwer przygotowuje odpowiedź
HTTP- odpowiedź
odpowiedź wraca do przeglądarki
Renderowanie
przeglądarka pokazuje stronę

Ścieżka od paska adresowego do gotowej strony na ekranie

DNS: jako nazwa domeny wiąże się z adresami sieciowymi

Domain wydaje się być `python.org` wygodne dla ludzi, ale komputery są w Internet znajduje się nawzajem za pomocą numerów **IP-adresy**. Usługa **DNS** (Domain Name System — system nazw domen) przechowuje odpowiedniki między domenami a zapisami, które pomagają znaleźć odpowiedni serwer, i zamienia domenę w IP-adres, zanim przeglądarka będzie mogła wysłać żądanie.

Jeden domena — niekoniecznie jeden stały IP-adres
W przypadku dużych lokalizacji ta sama domena może faktycznie odpowiadać różnymi adresami IP- w różnych momentach lub regionach — na przykład, aby rozłożyć obciążenie na wiele serwerów lub przyspieszyć dostarczanie danych użytkownikom w różnych częściach świata. Model „jedna domena, dokładnie jeden serwer na zawsze”

Połączenie i zapytanie

Po otrzymaniu adresu IP- przeglądarka nawiązuje połączenie z serwerem i wysyła **HTTP-żądanie** jego struktura jest szczegółowo omówiona w Sekcji 22.9. Serwer (w naszym przypadku aplikacja na Flask) przetwarza żądanie i wysyła je **HTTP- odpowiedź**: Zazwyczaj HTML- strona, ale czasem dane w JSON formacie (Rozdział 22.15), obraz lub plik.

Renderowanie

Po otrzymaniu odpowiedzi przeglądarka analizuje HTML i buduje z niej drzewo DOM (Rozdział 22.4). Jako analizując analizę, znajduje odniesienia do CSS i JavaScript i zadaje im osobne HTTP- zapytania (Rozdział 22.14); strona może być aktualizowana stopniowo, jako Zdobądź potrzebne dane, nie tylko po załadowaniu wszystkiego.

Celowo nie analizujemy TCP/IP na poziomie pakietu

W połączeniu między przeglądarką a serwerem istnieją głębsze poziomy techniczne — dzielenie danych na pakiety, potwierdzenie dostarczenia i tak dalej. Na potrzeby tworzenia stron internetowych na poziomie tego rozdziału wystarczy model mentalny „przeglądarka wysyła żądanie — serwer odpowiada”; protokoły sieciowe tego poziomu to temat osobnego kursu.

URL: domena, ścieżka, parametry i fragment

Każdy adres w sieci ma jasną strukturę — przeanalizujmy go na części, używając jednego przykładu.

URL (Uniform Resource Locator — jednolity identyfikator zasobu) — to pełny adres czegoś w sieci: strony, obrazu, API-odpowiedzi. Ma wyraźną strukturę i każda część odpowiada za swoje:

Rozłożenie jednego URL na części

Część	Znaczenie na przykładzie	Za co odpowiada
Schemat	https	Którego protokołu użyć (Rozdział 22.10)
Host (domena)	example.com	Do którego serwera się zwracać
Port	443	Numer portu sieciowego, do którego następuje połączenie — często nie podaje się go jawnie, ponieważ schemat ma port domyślny
Ścieżka	/products/42	Jakie zasoby na serwerze są potrzebne
łańcuch znaków zapytania	?sort=price	Dodatkowe parametry w formacie klucz=wartość
Fragment	#details	Przestrzeń wewnątrz samej strony

Fragment nie trafia na serwer
Część z adresem po `#` obsługiwane przez przeglądarkę — zwykle, aby przewinąć stronę do elementu z takim identyfikatorem. Nie jest przesyłane na serwer jako część HTTP-żądania: serwer, który odpowiedział na `/products/42?sort=price#details`, fizycznie nie widzi `#details` - Otrzymuje tylko żądanie `/products/42?sort=price`.

łańcuch znaków zapytań — kilka parametrów

Po `?` parametry są wymieniane przez `&`: `?sort=price&page=2` — dwa parametry, `sort` oraz `page`. Flask czyta je `request.args`:

query_params.py

```
@app.route("/tovary")
def tovary():
    sortirovka = request.args.get("sort", "name")
    return f"Sortowanie: {{sortirovka}}"
```

Praktyka: analiza URL
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/22-08/index.html)

HTTP: żądania, odpowiedzi, metody i kody stanu

Wspólny zestaw reguł, według których przeglądarka i serwer wymieniają wiadomości.

HTTP (HyperText Transfer Protocol) to protokół wymiany aplikacji Wiadomości między klientem a serwerem w sieci: określa wspólny zestaw reguł, na podstawie których Przeglądarka i serwer wymieniają żądania i odpowiedzi. Każda wiadomość — prośba lub odpowiedź — składa się z podobnych części.

HTTP-żądanie	HTTP- odpowiedź
Metoda: Co robić (GET, POST...)	Kod statusu: jak poszło (200, 404...)
Ścieżka: jaki zasób jest potrzebny	Tytuły: Metadane odpowiedzi
Nagłówki: metadane żądania	Sedno: Treść – HTML, JSON, plik...
Ciało: dane opcjonalne (np. formularz)	

Podstawowe metody HTTP

Metoda	Znaczenie zwyczajne
GET	Uzyskaj dane — otwórz stronę, zobacz lista

Metoda	Znaczenie zwyczajne
POST	Wyślij nowe dane — dodaj zadanie, wyślij formularz
PUT	Całkowicie zastąpić istniejący zasób
PATCH	Częściowa modyfikacja istniejącego zasobu
DELETE	Usuń zasób

Są to powszechne konwencje, a nie prawo fizyczne: żadna z tych metod zmusza serwer do określonego zachowania – Flask wykona każdy kod, który ty zapis do handlera, niezależnie od metody. Ale przestrzeganie tych konwencji rzeczywiście Aplikacja jest zrozumiała dla innych deweloperów oraz dla narzędzi takich jak pamięć podręczna przeglądarki. Nie każda aplikacja korzysta ze wszystkich pięciu metod — mała strona sobie radzi tylko GET i POST jak w sekcji 22.5.

GET nie jest „bezpieczniejszy” niż POST — i odwrotnie

GET i POST nie różnią się w ochronie przesyłanych danych: jest to określane przez protokół (HTTP lub HTTPS, sekcja 22.10), a nie przez metodę. Różnica jest inna: z definicji HTTP GET przeznaczony do odbierania danych i nie powinien powodować stopniowych zmian na serwerze — to podstawa zachowania przeglądarek, proxy cache'owych i innych narzędzi, które uznają GET za bezpieczne do powtarzania: odświeżanie strony lub zakładki nie powinno nic zmieniać. Dlatego usunięcie zadania przez link GET- jest złym pomysłem (sekcja 22.35 pokazuje poprawne): przypadkowe odświeżenie strony przez przeglądarkę nie powinno niczego usuwać.

Kody statusowe: Jak przebiegło żądanie

Zakres	Znaczenie	Przykłady
2xx	Sukces	200 OK, 201 Created
3xx	Przekierowanie	302 Found, 303 See Other — „zasób musi być wyświetlany pod innym adresem”

Zakres	Znaczenie	Przykłady
4xx	Błąd klienta	400 Bad Request, 404 Not Found
5xx	Błąd serwera	500 Internal Server Error

Rozdział 22.13 pokazuje kod 303 w działaniu — jako część wzorca POST-Redirect-GET, a sekcja 22.31 — kody 400 i 404 podczas obsługi błędów wejściowych.

Treści wiadomości to nie tylko HTML

Ciało odpowiedzi HTTP- może zawierać stronę HTML-, ale równie łatwo — dane w formacie JSON (sekcja 22.15), obraz, plik CSS-, plik JavaScript- lub dowolny inny plik. To właśnie nagłówek `Content-Type` informuje przeglądarkę, jak zrozumieć ciało Odpowiedź — `text/html`, `application/json`, `image/png` i tak dalej.

> **Praktyka: Klasyfikacja kodów statusowych**
> interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji
> Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/22-09/index.html)

HTTPS: zabezpieczenie połączenia ze stroną

HTTPS chroni kanał między przeglądarką a serwerem, ale nie zastępuje bezpieczeństwa samej aplikacji.

HTTPS — użycie protokołu HTTP na TLS (Transport Layer Security): te same wiadomości zapytanie-odpowiedź, ale połączenie, którym są przesyłane, jest zabezpieczone. To ważne, ponieważ zwykłe zapytanie HTTP- przechodzi przez sieć bez szyfrowania transportowego: każdy, kto technicznie potrafi przechwycić ruch między przeglądarką a serwerem (np. w sieci publicznej Wi-Fi), może go odczytać lub niezauważalnie zmienić.

	HTTP	HTTPS
Dane w ścieżce	Bez szyfrowania	Zaszyfrowane

	HTTP	HTTPS
Czy można przechwycić i odczytać	Tak, technicznie jest to możliwe	Praktycznie żadnych – bez klucza szyfrującego treść jest nieczytelna
Czy możliwe jest podszywanie się pod dane podczas transportu, nie będąc zauważonym?	Tak	Nie – Podszywanie się narusza sprawdzenie integralności i połączenie nie jest uznawane za nieprawidłowe
Weryfikacja tożsamości serwera	Tak	Certyfikat potwierdza, że serwer jest tym, za kogo się podaje

TLS rozwiązuje trzy problemy: **poufność** (dane w trakcie przesyłania są nieczytelne dla osób trzecich), **integralność** (odbiorca może wykryć, że dane zostały zmienione w trakcie przesyłania) i **autentyczność serwera** (certyfikat potwierdza, że Porozmawiaj z tym serwerem, nie z tym zastępczym).

> **HTTPS chroni kanał, a nie sama aplikacja**
> HTTPS nie czyni strony „bezpieczną" *po drodze* między przeglądarką a serwerem, ale jeśli wystąpi błąd w samej aplikacji (na przykład SQL- injection – sekcja 22.32), HTTPS go w żaden sposób nie naprawia: dane trafiają do serwera w całości i będą tam niepoprawnie przetwarzane. Są to różne warstwy ochrony, a jedna nie zastępuje drugiej.

Celowo nie analizujemy matematyki szyfrowania — na tym poziomie w tworzeniu stron internetowych Z rozdziału wystarczy wiedzieć, dlaczego HTTPS jest potrzebny i co dokładnie chroni. W rozwoju lokalnym (Rozdział 22.34) najczęściej używa zwykłego HTTP na swoim komputerze — szyfruje kanał nie jest wymagane w jednej maszynie; HTTPS staje się potrzebne, gdy aplikacja jest rzeczywista szczegółowy i dostępny przez Internet.

Routing w Flask: URL i funkcje obsługi

Od HTTP- żądania do wywołania funkcji - Flask. urządzenie routingu

Rozdział 22.5 już pokazał `@app.route("/")` — ale co dokładnie Flask to robi z tym dekoratorem? Przyjrzyjmy się całej ścieżce pojedynczego żądania wewnątrz aplikacji.

Pojedyncza ścieżka żądań wewnątrz Flask- aplikacji

Każde wyzwanie `@app.route(...)` rejestry **rządy** (URL rule) to na przykład sam szablon ścieżki `/task/<int:task_id>`. Całość wszystkich reguł nazywana jest **trasowanie** (routing). Każda zasada również **punkt końcowy** (endpoint) to symboliczna nazwa, pod którą Flask zna powiązaną funkcja: domyślnie jest to nazwa samej funkcji. Przez końcowy, a nie przez URL bezpośrednio, `url_for(...)` tworzy adres — pozwala to zmienić samą ścieżkę, bez dotykając kodu, który się do niego odnosi. Gdy nadejdzie żądanie, Flask przeszukuje

Zarejestrowana reguła, która odpowiada ścieżce i metodzie oraz wywołuje powiązane **funkcję-obsługę** – już widzieliście ten mechanizm w sekcji 22.5, po prostu Bez tytułów.

Parametr ścieżki to typowa część URL

Część ścieżki w nawiasach kątowych Flask przekazuje funkcji jako argument. Możesz ją określić Oczekiwany typ jest zgodny z regułą:

path_param.py

```
@app.route("/task/<int:task_id>")
def poluchit_zadachu(task_id):
    return f"Zadanie nr {{task_id}}"
```

<int:task_id> — niebędący liczbą adres odchyla się przed wywołaniem funkcji

Jeśli otworzyć `/task/abc` , Flask nie spowoduje `poluchit_zadachu()` ogólnie to reguła `<int:task_id>` wymaga liczby całkowitej, a żądanie otrzyma standardową odpowiedź 404 (sekcja 22.9) zanim dotrze do kodu.

Funkcja obsługiwa zwraca treść odpowiedzi

To, co zwraca funkcja, staje się ciałem HTTP-odpowiedzi: łańcuch znaków, renderowany szablon (rozdział 22.12) lub na przykład słownik — taki słownik Flask przekształca w JSON (rozdziały 22.15-22.16). Dlatego kontrolery nazywają *funkcjami widoku* (view functions): ich zadaniem jest przygotowanie widoku danych dla konkretnego zapytania.

Praktyka: Trasy i parametry ścieżek

Moduł flask nie jest zainstalowany w środowisku przeglądarki Pyodide – działa lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/22-11/index.html)

Jinja: szablony danych HTML

szablon HTML-a z Jinja wyrażeniami i Python danymi oraz wbudowaną ochroną przed przypadkowym znaczeniem.

Rozdział 22.5 już używa szablonów, więc przyjrzyjmy się, jak działa silnik **Jinja**, którego Flask używa domyślnie. Szablon jest regularny HTML-file, gdzie wstawiane są dwa typy struktur: `{{ выражение }}` (wyświetl wartość) i `{% команда %}` (warunek, pętla, dziedziczenie).

templates/spisok.html

```
<ul>
  {% for zadacha in zadachi %}
    <li>{{ zadacha.title }}</li>
  {% endfor %}
</ul>

{% if not zadachi %}
  <p>Задач пока нет.</p>
{% endif %}
```

Automatyczne osłony – Jinja domyślnie chroni cię

Jeśli wartość wstawia `{{ }}`, zawiera HTML- znacznik (na przykład użytkownik wpisał `<b>`), Jinja domyślnie ucieka znaki specjalne — zamienia je `<` w `<` i tak dalej. Przeglądarka pokaże to jako zwykły tekst zamiast jako znacznik. To się nazywa **automatyczne osłony** (autoescaping), i jest włączony dla szablonów HTML-. Flask domyślnie.

|safe wyłącza dokładnie tę ochronę

Jinja ma filtr `|safe`, który wyłącza automatyczne escapowanie dla określonej wartości, Jinja wkleja go tak, jak jest, włącznie z marżą. To jest odpowiednie dla tekstu, nad którym masz pełną kontrolę (na przykład HTML zakodowany na stałe w kodzie aplikacji), ale **nie dla danych wprowadzonych przez użytkownika**: Do czego to prowadzi w praktyce, pokazano w sekcji 22.32. Jeśli masz wątpliwości, nie dodawaj `|safe`.

Dziedziczenie szablonów — wspólny szkielet strony

Nie powtarzać `<head>`, nawigacja i połączenia CSS w każdym szablonie stwórz bazowy szablon z nazwanymi blokami, które się zapełniają Szablony potomne:

templates/base.html

```
<!doctype html>
<html lang="ru">
<head>
  <meta charset="utf-8">
  <title>{% block title %}Список задач{% endblock %}</title>
  <link rel="stylesheet" href="{{ url_for('static',
filename='style.css') }}">
</head>
<body>
  {% block content %}{% endblock %}
</body>
</html>
```

templates/index.html

```
{% extends "base.html" %}

{% block content %}
  <h1>Мой список задач</h1>
  ...
{% endblock %}
```

`{% extends "base.html" %}` mówi: „Weź podstawy szablon i wstaw moją treść do oznaczonych pól.” `base.html` i kilka Szablony dziecięce.

Praktyka: Szablony i automatyczne ucieczki
Moduł flask nie jest zainstalowany w środowisku przeglądarki Pyodide – działa lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/22-12/index.html)

HTML-forms i przesyłanie danych do serwera

Od HTML- formy do zapisanych danych — i dlaczego po zapisaniu potrzebne jest jeszcze jedno zapytanie.

Rozdział 22.2 pokazał HTML- formularz, rozdział 22.5 — trasę, która go obsługuje. Teraz przeanalizujemy tę ścieżkę w całości, z jednym ważnym szczegółem: co robić *po* tego, jak formularz jest przetworzony.

templates/forma.html

```
<form action="{{ url_for('dobavit') }}" method="post">
  <label for="zadacha">Новая задача</label>
  <input type="text" id="zadacha" name="zadacha" required>
  <button type="submit">Добавить</button>
</form>
```

`method="post"` mówi przeglądarce o przesłaniu danych formularza jako treść żądania POST-, a nie jako część adresu. Atrybut `name="zadacha"` pole definiuje klucz, pod którym wartość dotrze na serwer.

app.py

```
@app.route("/dobavit", methods=["POST"])
def dobavit():
    novaya_zadacha = request.form.get("zadacha", "").strip()
    if novaya_zadacha:
        zadachi.append(novaya_zadacha)
    return redirect(url_for("glavnaya"), code=303)
```

`request.form.get("zadacha", "")` odzyskuje wartość pola przez imię; drugi argument dotyczy tego, co zwrócić, jeśli nie było żadnego pola. Jakie inne sprawdzenia należy wykonać Dodaj przed zapisaniem wprowadzonej wartości – w sekcji 22.31.

POST-Redirect-GET — po co jest redirect na końcu

Obsługa formularza nie kończy się na renderowaniu strony, lecz na `redirect(url_for("glavnaya"), code=303)` – HTTP-odpowiedź z kodem 303 See Other (Rozdział 22.9), który informuje przeglądarkę o adres, z którego należy żądać wynik używając metody GET. Przeglądarka Yandex wykonuje drugie, teraz bezpieczne żądanie i pokazuje Page.

POST-Redirect-GET: dwa osobne zapytania zamiast jednego

Bez ponownego wysyłania redirect duplikatów danych

Jeśli obsługa żądań POST- natychmiast zwraca gotową stronę (zamiast redirect), przeglądarka kojarzy już wyświetloną stronę z tym POST- żądaniem, a odświeżenie strony (F5) ponownie przesła formularz, cicho duplikując zadanie. Redirect zapewnia, że po odpowiedzi serwera bieżąca operacja przeglądarki będzie bezpieczna GET którą można powtarzać dowolną liczbę razy.

Praktyka: forma i POST-Redirect-GET
Moduł flask nie jest zainstalowany w środowisku przeglądarki Pyodide – działa lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/22-13/index.html)

Pliki statyczne: CSS, obrazy i JavaScript

Jedna strona — kilka niezależnych żądań: HTML jest ponownie składana, statyka jest serwowana w obecnej formie.

Otwierając stronę ze stylem, przeglądarka faktycznie tworzy kilka oddzielnych HTTP- żądań: po jednym dla HTML, po jednym dla każdego pliku CSS-, obrazu, oraz JavaScript-, do którego odwołuje się strona. Takie pliki nazywane są **statyczny**: serwer przekazuje je bez zmian, w przeciwieństwie do HTML, który Jinja renderuje na nowo dla każdego żądania.

Pojedyncza strona – wiele niezależnych HTTP- żądań

Flask domyślnie oczekuje plików statycznych w folderze `static/` obok aplikacji i poda je pod adresami takimi jak `/static/имя_файла`:

struktura_proekta.txt

```
todo-app/
  app.py
  templates/
    base.html
  static/
    style.css
```

Wygodniej jest nie wpisywać adresu statycznego pliku do szablonu ręcznie, lecz otrzymywać go przez `url_for('static', filename='style.css')` – Ta funkcja się buduje URL opiera się na konfiguracji trasowania Flask, a nie tylko sklejaniu sznuru. To również Oszczędza to dużo pracy, jeśli struktura folderów się zmieni:

templates/base.html

```
<link rel="stylesheet" href="{{ url_for('static',
filename='style.css') }}">
```

templates/ i static/ to różne foldery, różne role

`templates/` przechowuje pliki, które Jinja renderują na nowo dla każdego żądania, zastępując dane z Python. `static/` przechowuje gotowe pliki udostępniane w obecnej formie, bez przetwarzania, takie jak CSS, obrazy, JavaScript, czcionki.

JSON: format wymiany danych

Popularny format tekstu dla danych strukturalnych, który jest rozumiany przez programy w różnych językach.

JSON (JavaScript Object Notation) to format tekstowy do transmisji Dane strukturalne między programami. Nazwa przypomina JavaScript, ale JSON od dawna jest używany przez programy w każdym języku, w tym Python — to tylko tekstowe Format, nie kod, który gdzieś działa.

Znaczenie JSON	Zgodność w Python
cel `{{"ключ": значение}}`	`dict`
tablica `[1, 2, 3]`	`list`
łańcuch znaków `"текст"`	`str`
liczba `42` / `3.14`	`int` / `float`
`true` / `false`	`True` / `False`
`null`	`None`

JSON to nie to samo, co pisanie słownika Python
JSON ma ścisłą składnię: klucze i wartości łańcuch znaków muszą być w cudzysłowie. `{{'name': 'Ada'}}` — to wydruk słownika Python, a nie JSON:, w poprawnym JSON wygląda to tak `{{"name": "Ada"}}`. Pojedyncze cudzysłowy, `True` wielką literą, przecinek przed zamknięciem przełamuje JSON, nawet jeśli wygląda niemal poprawnie.

Moduł json to standardowa biblioteka Python

json_primer.py

```
import json

zadacha = {{"title": „Kup chleb”, "done": False}}

# Python -> tekst JSON
tekst = json.dumps(zadacha, ensure_ascii=False)
print(tekst)   # {{"title": "Kup chleb", "done": false}}
14

# tekst JSON -> Python
obratno = json.loads(tekst)
print(obratno["title"])   # Kup chleb
```

ensure_ascii=False — aby cyrylica pozostała cyrylicą
Domyślnie `json.dumps()` zamienia wszystkie nie-ASCII znaki na sekwencje escape- takie jak `\u041a` jest ważne, ale dla osoby JSON. `ensure_ascii=False` pozostawia cyrylicę bez zmian; wynik pozostaje taki sam JSON, po prostu wygodniejszy do czytania.

JSON ograniczenia formatu

Format nie ma osobnego typu daty i godziny — zwykle jest przekazywany jako łańcuch znaków do uzgodniony format (np. `"2026-08-23"`) i już są demontowane na boku. Nie ma też typu dla danych binarnych (obraz, plik) — zazwyczaj są one kodowane do tekstu (na przykład Base64) lub przekazane jako osobny JSON zapytania. oraz liczby rzeczywiste tak ściśle jak Python: bardzo duże lub bardzo dokładne liczby przy Transfery między różnymi językami programowania czasem wymagają dokładności.

JSON jest formatem danych, a nie bazą danych i sam w sobie nie jest API

JSON opisuje, *jak nagrywać* dane w formie tekstu. Nie przechowuje on danych na stałe samodzielnie (do tego potrzebna jest baza danych — sekcja 22.20) i nie jest samym interfejsem programu (do tego potrzebny jest kod, który go przetwarza — przykład takiego kodu znajduje się w sekcji 22.16). JSON jest powszechnym formatem tekstowym do wymiany danych między różnymi częściami systemu.

Praktyka: Przekształcanie danych między Python a JSON
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/22-15/index.html)

HTTP API: wymiana danych między programami

Ten sam serwer, ten sam Flask — ale odbiorca odpowiedzi to teraz inny program, a nie przeglądarka użytkownika.

Do tej pory aplikacja Flask- odpowiadała HTML- stronami — są one czytane przez przeglądarkę. Ale odpowiedź serwera może mieć też innego odbiorcę: aplikację mobilną, inny serwer lub JavaScript-code na tej samej stronie (sekcja 22.4 już pokazywała `fetch()`). Taka odpowiedź zwykle nie pojawia się w postaci HTML, lecz w postaci Typ danych – najczęściej JSON.

Interfejs programowy, przez który jeden program odbiera dane od innego, nazywa się **API** (Application Programming Interface — interfejs programowy załączniki). Konkretny adres w obrębie API nazywany jest również **punkt końcowy** jest szersze, bardziej powszechne znaczenie słowa niż nazwa końcowa trasy Flask z Rozdział 22.11: Dotyczyła wewnętrznego identyfikatora dla `url_for()`, tutaj — po prostu o konkretnym URL, który zwraca dane, a nie HTML-stronę.

	Regularna trasa	API-endpoint
Kto czyta odpowiedź	Man przez przeglądarkę	Program: JavaScript, aplikacja, inny serwer
Format odpowiedzi	HTML	Najczęściej JSON

	Regularna trasa	API-endpoint
Content-Type odpowiedzi	text/html	application/json

Mały punkt końcowy JSON-point na Flask

app.py

```
from flask import jsonify

@app.route("/api/tasks")
def api_tasks():
    return jsonify([{"id": z["id"], "title": z["title"],
"done": z["done"]}} for z in zadachi])
```

`jsonify(...)` robi dwie rzeczy jednocześnie: zamienia dane Python w tekst JSON (jak `json.dumps()` w sekcji 22.15), oraz Ustala tytuł odpowiedzi `Content-Type: application/json`, aby odbiorca wiedział, jak odczytać treść odpowiedzi.

Odpowiedź /api/tasks w przeglądarce: tablica obiektów JSON-object z polami done, id i title, cyrylica w title jest pokazana jako sekwencje ucieczki w postaci \u0418

Prawdziwa odpowiedź GET /api/tasks końcowego projektu. To jest poprawna JSON — ale domyślnie jsonify() wymyka się z cyrylicy w \uXXXX-sequence, tak jak opisano w sekcji 22.15 o json.dumps() bez ensure_ascii=False.

REST jest powszechnym stylem, a nie obowiązkowym standardem

Wiele API jest budowanych w stylu **REST** (Representational State Transfer): ścieżka nazywa zasób (`/api/tasks`), a metoda określa akcję na niej — GET otrzymuje lista POST tworzy nowy rekord. To popularna i użyteczna konwencja, ale nie jedyny możliwy sposób na budowanie API — ważne jest, aby zrozumieć zasadę „danych zamiast HTML”

Rozdział 22.35 doda taki endpoint do projektu końcowego — GET /api/tasks, zwracający bieżące lista zadań w formacie JSON.

Praktyka: formatowanie danych dla odpowiedzi API
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/22-16/index.html)

Frameworki internetowe dla Python: Flask, Django, FastAPI i innych

Framework webowy to zestaw gotowych narzędzi do tworzenia aplikacji webowych. Flask, Django, FastAPI i Starlette inny zestaw wbudowanych funkcji.

Web Framework to zestaw bibliotek, reguł i gotowych komponentów, które uproszczenie tworzenia aplikacji webowych: routing, przetwarzanie HTTP- żądań i odpowiedzi, z szablonami, walidacją danych, dostępem do baz danych i innymi popularnymi zadaniami. Flask — nie jedyny framework webowy dla Python; pozostałe mają inny zestaw wbudowanych funkcji.

Flask
Flask — lekki WSGI- framework do tworzenia aplikacji webowych na Python (rozdział 22.19). Jego podstawowe jądro jest stosunkowo niewielkie, a dodatkowe funkcje zazwyczaj są dołączane poprzez biblioteki i rozszerzenia.

Django
Django jest pełnowymiarowym frameworkiem webowym na Python. Zawiera ORM, migrację, routing, szablony, narzędzia uwierzytelniania oraz interfejs administracyjny.

FastAPI
FastAPI to internetowy framework do tworzenia HTTP API na Python. Wykorzystuje Python adnotacje typu do opisu i walidacji danych oraz może automatycznie generować API dokumentację na podstawie OpenAPI.

Starlette
Starlette to lekki framework ASGI- i zestaw narzędzi do tworzenia asynchronicznych usług sieciowych w Python. FastAPI wykorzystuje Starlette jako jeden z kluczowych komponentów.

Każdy framework zajmuje się częścią typowej pracy: routingiem (kto jest odpowiedzialny za Jaki jest adres – sekcja 22.11), przetwarzanie żądań i odpowiedzi, czasem – szablony, weryfikacja Dane wejściowe, praca z bazą danych, autoryzacja, narzędzia testowe. Różnice między ramami — w tym, że *dokładnie ile* podejmuje i jak łatwo się podejmuje Wymień którąkolwiek z tych części na swoją.

Inne frameworki internetowe dla Python

Oprócz Flask, Django, FastAPI Starlette Python również korzysta z tworzenia stron internetowych innych projektów. Żaden z nich nie jest szczegółowo omawiany w tym rozdziale – są wspomniane, Abyś rozpoznał nazwiska, gdy spotkasz je w prawdziwych projektach.

Ramy	Główny obszar	Interfejs/model	Typowe zastosowanie
Litestar	Wydajność i silne pisanie	ASGI	API-usługi, gdzie typy i walidacja są ważne
Quart	API, zgodny z Flask	ASGI	Portowanie aplikacji Flask-aplikacji na model asynchroniczny
Pyramid	Elastyczna architektura bez narzuconej struktury	WSGI	Projekty, które rozwijają się z małego skryptu do dużej aplikacji
Tornado	Asynchroniczne I/O, długotrwałe połączenia	Własnościowy silnik asynchroniczny + ASGI/WSGI- kompatybilny	WebSockety, długotrwałe połączenia
Bottle	Minimalizm, jeden plik	WSGI	Bardzo małe usługi i prototypy

Nie wybieraj systemu opartego na wynikach popularności

Benchmarki wydajności i listy «najszybszych frameworków» w dużej mierze zależą od tego, co dokładnie jest mierzone, i szybko się dezaktualizują. Rozsądniej jest pytać: co dokładnie powinno robić moje aplikacja, jakie doświadczenie ma już zespół, co jest tutaj już używane. Bardziej merytoryczne porównanie pod kątem konkretnych zadań, a nie po abstrakcyjnym «co jest lepsze», — w rozdziale 22.18.

Oficjalna dokumentacja: Flask, Django, FastAPI, Starlette.

Porównanie Flask, Django i FastAPI

Flask, Django i FastAPI rozwiązują różne klasy problemów i oferują różne wbudowane funkcje.

Rozdział 22.17 przedstawiała ogólną mapę. Teraz porównajmy trzy najbardziej znaczące ramy – Flask, Django i FastAPI opierają się na konkretnych zadaniach, a nie na ogólnych wrażeniach.

Kryteria	Flask	Django	FastAPI
Cel	Ogólnego przeznaczenia, często strony z HTML- stronami	Do ogólnego użytku, aplikacje wokół bazy danych	Przede wszystkim, HTTP API
Trasowanie	Wbudowane	Wbudowane	Wbudowane
Szablony HTML	Jinja	Wbudowany silnik szablonów	To nie jest główne zadanie — ramy skupiają się na API
ORM	Brak własności – Poprzez ORM osoby trzeciej (Rozdział 22.26)	Tak, wbudowany	Nie posiada własnego — przez zewnętrzny ORM
Migracje	Jeśli nie masz własnego — przez narzędzie zewnętrzne	Tak, wbudowane (makemigrations/migrate)	Jeśli nie masz własnego — przez narzędzie zewnętrzne

Kryteria	Flask	Django	FastAPI
Interfejs administracyjny	Nie mam własnego	Tak, wbudowana automatyka	Nie mam własnego
Uwierzytelnianie	Nie jest własny — przez rozszerzenia	Tak, wbudowany system użytkownika	Jeśli nie masz własnej — przez biblioteki firm trzecich
Weryfikacja danych	Ręcznie lub przez rozszerzenia	Poprzez formularze i serializatory Django/DRF	Wbudowane, oparte na wskazówkach typów Python
OpenAPI	Nie mam własnego	W rdzeniu nie ma własnego	Tak, wbudowane, automatyczne
WSGI / ASGI (rozdział 22.19)	WSGI, z ograniczonym wsparciem dla asyncopiekunów	Obsługuje zarówno WSGI, jak i ASGI tryby wdrażania	ASGI początkowo
Typowe projekty	Małe i średnie strony, proste API	Duże aplikacje z bazą danych i panelem administracyjnym	HTTP API, usługi z danymi typowymi

Django REST Framework — oddzielny projekt, a nie część jądra Django

Django od razu po wyjęciu z pudełka może dać HTML- strony i ma panel administracyjny, ale pełnoprawny zestaw narzędzi dla REST API to osobny pakiet, Django REST Framework (DRF) instalujesz i podłączasz Django.

Punkt wyjścia do wyboru

Potrzebuję strony internetowej z formularzami, szablonami, bez gotowego panelu administracyjnego	→ **Flask**
Potrzebuję systemu biznesowego z bazą danych i prostym panelem administracyjnym	→ **Django**
Potrzebujesz API- usługi z automatyczną dokumentacją	→ **FastAPI**
Wymaga niskiego poziomu kontroli nad aplikacją ASGI-	→ **Starlette**

Projekty rzeczywiste często łączą narzędzia — to są punkty wyjścia, a nie sztywne zasady

Dlaczego Flask wybrano do projektu edukacyjnego

W tym rozdziale pierwszy projekt serwera jest zbudowany na Flask. Flask zapewnia podstawy Narzędzia aplikacji webowych – routowanie, przetwarzanie żądań/odpowiedzi oraz praca z szablony — bez dużego zestawu obowiązkowych podsystemów. Dlatego w małej szkole edukacyjnej Dla projektu wygodne jest rozpatrywanie każdego z tych mechanizmów kolejno.

Django oferuje bardziej gotowe komponenty, takie jak ORM, migracje, uwierzytelnianie oraz interfejs administracyjny. FastAPI koncentruje się głównie na tworzeniu HTTP API oraz Uważnie wykorzystuje adnotacje typu Python.

Flask wybrane tutaj jako odpowiednie narzędzie dla pierwszej aplikacji do nauki internetowej. To To nie znaczy, że Flask jest najlepszym rozwiązaniem dla każdego projektu.

Praktyka: Wybór ram według zadania
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/22-18/index.html)

WSGI i ASGI: komunikacji między aplikacją webową a serwerem

Ramy opisują, co zrobić z żądaniem. WSGI i ASGI to umowa dotycząca sposobu, w jaki żądanie do niego dotrze.

Rozdział 22.18 wspomniała WSGI i ASGI jako różnicę techniczną między ramami. Przyjrzyjmy się temu Co oznaczają na poziomie idei, bez szczegółów wdrożenia.

WSGI (Web Server Gateway Interface) oraz **ASGI** (Asynchronous Server Gateway Interface) — standardowe interfejsy opisujące, w jaki sposób serwer aplikacji przekazuje zapytanie HTTP do aplikacji Python i otrzymuje od niej odpowiedź. W normalnym wdrożeniu połączenia sieciowe obsługuje serwer aplikacji lub inna infrastruktura serwerowa, a nie sam framework (Flask, Django, FastAPI) bezpośrednio — WSGI/ASGI to właśnie ogólne porozumienie między nimi.

	WSGI	ASGI
Pełne imię i nazwisko	Web Server Gateway Interface	Asynchronous Server Gateway Interface
Model przetwarzania żądań	Synchronous – Jedno żądanie jest przetwarzane w całości, zanim rozpocznie się kolejne w tym wątku	Obsługuje przetwarzanie asynchroniczne – aplikacja może czekać na wolną operację bez blokowania całego procesu
Długotrwałe połączenia (WebSocket)	Nie jest przeznaczony na taki scenariusz	Wspierane przez kompatybilne frameworki i serwery
Kto go używa	Flask	Starlette, FastAPI

Django wspiera WSGI i ASGI
Modern Django obsługuje oba tryby wdrożenia — tradycyjne WSGI i ASGI — w zależności od konfiguracji projektu. Nie oznacza to, że każda część Django działa równie dobrze asynchronicznie, ale sam framework nie jest sztywno powiązany tylko z jednym protokołem.

Flask i async to nie to samo co framework ASGI-framework

Flask wie, jak przywołać `async def`-obsługujące, ale samo aplikacja w tym czasie pozostaje WSGI-aplikacją według architektury: do wykonania takiego handlera Flask uruchamia pętlę zdarzeń i wykonuje w niej korutynę, zajmując przy tym jeden handler żądania na cały czas jej działania. Jest to wygodne, gdy konkretny handler musi czekać na powolną operację wejścia-wyjścia, ale nie przekształca Flask w pełnoprawny ASGI-framework jak FastAPI — jeśli aplikacji zasadniczo zależy na asynchronicznym modelu na wszystkich poziomach, ASGI-framework będzie lepszy.

Framework i serwer aplikacji to różne rzeczy

Flask lub FastAPI opisuje *co* robić z zapytaniem. Oddzielny program — **serwer aplikacji** jest odpowiedzialny za faktyczne zaakceptowanie połączenia przez nasiewanie sieci i przekazywanie żądania do frameworka przez protokół WSGI lub ASGI:

Protokół	Przykłady serwerów aplikacyjnych
WSGI	Gunicorn, Waitress
ASGI	Uvicorn, Hypercorn

Do tego rozróżnienia wrócimy w Sekcji 22.34 – Serwer deweloperski wbudowany w Flask, oraz Serwer aplikacji desktopowych wykonuje to samo zadanie na różne sposoby.

Praktyka: WSGI czy ASGI?
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/22-19/index.html)

Dlaczego aplikacja webowa potrzebuje bazy danych

Lista Python w pamięci przechowuje dane tylko do momentu ponownego uruchomienia procesu — a prędzej czy później proces zostanie ponownie uruchomiony.

Lista zadań w sekcji 22.5 jest przechowywana w następujący sposób:

app.py

```
zadachi = [„Naucz się podstaw Python”, „Zbudować stronę na Flask”]
```

To zwykły zmienna Python — lista, który żyje w pamięci RAM aż do momentu Proces aplikacji jest w toku. Taka pamięć ma ograniczenia, które są niewidoczne na Mały przykład, ale staje się poważnym problemem wraz ze wzrostem aplikacji i obciążenia na nim.

Problem	Co się dzieje
Restart	Zatrzymałem proces Python (zaktualizowałem kod, zrestartowałem serwer) — lista zniknął razem z nim
Procesy wielokrotne	Wdrożenie produkcyjne (Rozdział 22.34) często uruchamia wiele procesów aplikacyjnych jednocześnie— każdy ma własną pamięć, własny lista i nie widzą zmian nawzajem
Zapytanie o dane	Listę można przeszukiwać i filtrować za pomocą Python, ale nie może szybko znaleźć potrzebnych wpisów wśród milionów bez pełnego wyszukiwania
Spójność przy jednoczesnych zmianach	Dwa żądania, które pojawiają się niemal jednocześnie, mogą nieprzewidywalnie zrujnować lista, jeśli oba zmienią je jednocześnie

DBMS (system zarządzania bazami danych) to system programowy, tworzenie, przechowywanie, zapytania i zarządzanie bazami danych; **baza Dane** są zorganizowane przechowywane dane, z którymi pracuje DBMS. DBMS zapewnia mechanizmy trwałego przechowywania, żądań, transakcji, limitów i Konsekwentna praca z danymi — rozwiązuje dokładnie te problemy. Konkretne zabezpieczenia dla zależą od wybranego DBMS, zaprojektowanego schematu, wykorzystania transakcji oraz kodu sama aplikacja: DBMS nie rozwiązuje automatycznie problemu, jeśli tego nie robi Używaj poprawnie.

Trwałość — dane, które przetrwają ponowne uruchomienie programu
Właściwość przechowywania danych tak, aby nie znikały po zakończeniu procesu, nazywa się **z zachowaniem danych między uruchomieniami**, czyli persistence. List Python nie posiada tej właściwości w pamięci; Plik na dysku lub bazie danych to robi.

Sekcje 22.21-22.28 analizują, jak bazy danych są zorganizowane i w jakim języku są pracę, a Rozdział 22.29 zastępuje lista Python w rozdziale roboczym bazą danych SQLite.

Relacyjne bazy danych: tabele, klucze i relacje

Dane w łańcuch znakówx i kolumnach — oraz relacje między tabelami poprzez klucze pierwotne i obce.

Najczęściej spotykanym typem bazy danych jest **relacyjna**: Przechowywane dane w **tabele**wyglądają jak arkusz tabeli obliczeniowej. Każdy **łańcuch znaków** — jeden wpis, każdy **kolumna** jest jedną z jego właściwości z predefiniowanym typem.

Jednemu użytkownikowi users może odpowiadać kilka wierszy tasks

id	title	done
1	Poznaj podstawy Python	1
2	Zbudować stronę na Flask	0

Taki stół — `tasks`: Każdy wiersz ma `id` jest unikalną liczbą, dzięki której można ją jednoznacznie znaleźć. Kolumna, która to robi (zazwyczaj `id`) są nazywane **klucz główny** (primary key).

Relacja między tabelami – klucz obcy

Gdyby każde zadanie miało właściciela, drugi stół `users` przechowywał użytkowników, a `tasks` jest połączeniem do struny W niej:

id	title	done	user_id
1	Poznaj podstawy Python	1	7
2	Zbudować stronę na Flask	0	7

Kolumna `user_id` to się odnosi `id` w innej tabeli nazywa się **kluczem obcym** (foreign key).

Skąd wzięło się słowo „relacyjny”
Nazwa pochodzi od matematycznego pojęcia relation (relacja), leżącego u podstaw relacyjnego modelu danych: relacja to zbiór krotek tej samej struktury, a w praktyce przedstawia się ją w formie tabeli. Relacje między tabelami przez klucze obce są ważną częścią projektowania baz danych, ale nie źródłem nazwy modelu: jedna tabela bez żadnego klucza obcego nadal pozostaje relacyjną.

Ostateczny projekt pozostaje bez użytkowników — i jest to celowe

Jak szczegółowo wyjaśnia sekcja 22.31, aby nie rozszerzać tego rozdziału do pełnej autentyczności, ostateczny szkic (Rozdział 22.35) wykorzystuje tylko jedną tabelę, `tasks`, bez `users` i bez kluczy obcych. Relacje między tabelami — ważna idea, ale nie obowiązkowy element najmniejszej działającej aplikacji.

SQL: tworzymy, czytamy i edytujemy dane

Jeden język – stwórz tabelę, przeczytaj ją, zmodyfikuj i usuń dane.

SQL (Structured Query Language — język zapytań strukturalnych) — język definiowania struktury danych i wykonywania zapytań do relacyjnej bazy danych: za pomocą niego tworzą tabele, czytają, dodają, modyfikują i usuwają wiersze. SQL ma wspólny standard, ale każda konkretna system (SQLite, PostgreSQL, MySQL) — ma swoje rozszerzenia i cechy na jego bazie, dlatego mówi się nie o „różnych językach SQL", lecz o różnych **dialektach** jednym języku. Przykłady na tej stronie przedstawiono w SQLite (Rozdział 22.23).

CREATE TABLE — opisujący strukturę

sozdanie_tablicy.sql

```
CREATE TABLE tasks (
    id INTEGER PRIMARY KEY,
    title TEXT NOT NULL,
    done INTEGER NOT NULL DEFAULT 0
);
```

NOT NULL zabrania wartości NULL, a nie pustego łańcuch znaków

`NOT NULL` uniemożliwia zapisanie specjalnej wartości SQL-owej do kolumny `NULL` („wartość nieznana/brakująca" `''` nadal jest ważną wartością typu TEXT, oraz `NOT NULL` nie odrzuci jej. Jeśli aplikacja musi zabronić właśnie pustego (lub składającego się ze spacji) nagłówka zadania, potrzebne jest oddzielne sprawdzenie — po stronie aplikacji (rozdział 22.31) lub wyraźne ograniczenie `CHECK`, jak pokazano w sekcji 22.29.

`DEFAULT 0` ustawia wartość domyślną, jeśli nie podano jej jawnie.

Cztery operacje CRUD

Akcja	SQL	Co robi
Create	INSERT	Dodaj nową linię
Read	SELECT	Przeczytaj jedną lub więcej linijek
Update	UPDATE	Edytuj istniejący wiersz
Delete	DELETE	Usuń wiersz

insert.sql

```
INSERT INTO tasks (title) VALUES ('Купить хлеб');
```

select.sql

```
SELECT id, title, done FROM tasks WHERE done = 0 ORDER BY id
LIMIT 10;
```

`WHERE` zostawia tylko odpowiednie linie, `ORDER BY` ustala kolejność, `LIMIT` ogranicza liczbę wyników.

update.sql

```
UPDATE tasks SET done = 1 WHERE id = 1;
```

delete.sql

```
DELETE FROM tasks WHERE id = 1;
```

SQL Python: modułu sqlite3

sql_iz_python.py

```
import sqlite3

baza = sqlite3.connect("zadachi.db")
baza.execute(
    "INSERT INTO tasks (title) VALUES (?)",
    („Kup chleb”,),
)
baza.commit()

stroki = baza.execute("SELECT id, title, done FROM
tasks").fetchall()
for stroka in stroki:
    print(stroka)
```

znak zapytania zamiast f-strings nie jest wyborem stylistycznym

`?` w SQL-string to **parametryzowane zapytanie**: wartość jest podstawiona przez sam moduł `sqlite3` bezpiecznie oddzielone od tekstu prośby. Jeśli zamiast tego zapytanie zostanie złożone za pomocą łańcuch znaków f- z wejściem użytkownika w środku, otwiera to drogę do wstrzyknięcia SQL-injection – co dokładnie może pójść nie tak i dlaczego parametryzacja jest obowiązkowa, a nie pożądana, pokazuje sekcja 22.32.

Praktyka: CRUD na SQLite

interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/22-22/index.html)

SQLite: Wbudowany relacyjny DBMS

SQLite jest relacyjnym DBMS, który działa jako biblioteka w procesie aplikacji, bez osobnego serwera.

DBMS — system zarządzania bazą danych: oprogramowanie, który tworzy, przechowuje, przetwarza zapytania i zarządza bazami danych. **SQLite** to wbudowany relacyjny DBMS implementowany jako biblioteka programowa. W W przeciwieństwie do PostgreSQL czy

MySQL, SQLite nie wymaga osobnego procesu serwera bazy danych dane: aplikacja uzyskuje dostęp do biblioteki SQLite bezpośrednio, w ramach własnej biblioteki Proces.

W normalnym użyciu plików zawartość bazy danych jest SQLite przechowywana w jednym głównym pliku pliku. Podczas pracy SQLite może tworzyć dodatkowe pliki usługowe, takie jak rollback journal lub WAL. SQLite również utrzymuje bazę danych w pamięci (`:memory:`), która w ogóle nie zapisuje się na dysku — jest to wygodne, na przykład do testów (rozdział 22.33).

Pracować z SQLite w Python, `sqlite3`, który zapewnia interfejs DB-API wersji 2.0. W zwykłych wersjach CPython jest dostępny bez instalacji osobny pakiet przez pip, ale technicznie rzecz biorąc `sqlite3` odnosi się do opcjonalnych modułów konkretnego zestawu interpretera i wykorzystuje SQLite instalowane w systemie lub składane z Python.

	Konwencjonalny DBMS klient-serwer	SQLite
Oddzielny dodawca dokumentów	Tak, działa cały czas	None – Działa w procesie aplikacji
Główne przechowywanie	Zarządzane przez serwer bazy danych	Główny plik bazy danych (pliki serwisowe są możliwe journal/ WAL)
Przygotowanie do rozpoczęcia	Zainstaluj i uruchom serwer, skonfiguruj dostęp	Nie wymaga osobnej konfiguracji serwera
Kilka serwerów aplikacji pisze jednocześnie	Standardowy scenariusz	Ograniczone wsparcie dla jednoczesnego nagrywania

Gdzie SQLite szczególnie dobrze pasuje
SQLite jest pełnoprawnym silnikiem SQL-, który jest również używany przez bardzo duże aplikacje, zarówno jako główne przechowywanie danych lokalnych, jak i jako część większego systemu. SQLite doskonale nadaje się do aplikacji lokalnych, szkoleń, prototypów, testów (Rozdział 22.33) oraz wielu małych i średnich wdrożeń, gdzie model współbieżnego dostępu jest SQLite wystarczający. Baza danych klient-serwer jest wybierana nie dlatego, że SQLite jest gorsza jako silnik, lecz wtedy, gdy kilka serwerów aplikacji musi niezawodnie zapisywać się równolegle do tej samej bazy, co jest szczegółowo omówione w Sekcji 22.24.

Połączenie i kursor

podklyuchenie.py

```
import sqlite3

baza = sqlite3.connect("zadachi.db")
baza.row_factory = sqlite3.Row   # Dostęp do kolumn według
nazwy

kursor = baza.execute("SELECT id, title FROM tasks")
for stroka in kursor:
    print(stroka["id"], stroka["title"])

baza.close()
```

`sqlite3.connect(...)` otwiera się (lub tworzy, jeśli plik nadal jest nie) pliku bazy danych. `row_factory = sqlite3.Row` pozwala Określenie kolumn wyników po nazwie, a nie tylko przez indeks liczbowy, jest wygodniejsze i jest bardziej niezawodny, zwłaszcza jeśli kolejność kolumn w zapytaniu zmienia się.

Oddzielne połączenie dla każdego strumienia
Domyślnie jedno połączenie `sqlite3.connect(...)` nie może być używany z wielu wątków bez ostrożności. Praktycznym sposobem organizacji połączeń w aplikacji Flask- jest otwarcie połączenia dla każdego żądania i natychmiastowe jego zamknięcie, jak pokazano w Sekcji 22.29.

Oficjalna dokumentacja: SQLite, Python — moduł sqlite3.

Praktyka: połączenie z SQLite przez sqlite3
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/22-23/index.html)

PostgreSQL, MySQL/MariaDB i SQLite: główne różnice

Te DBMS różnią się architekturą, możliwościami i wymaganiami eksploatacyjnymi — wybór zależy od wymagań projektu.

SQLite nie jest jedynym relacyjnym DBMS. Zobaczmy, co on i siebie nawzajem robią Istnieją dwie szeroko stosowane opcje klient-serwer.

PostgreSQL
PostgreSQL jest obiektowo-relacyjnym DBMS o architekturze klient-serwer: aplikacja łączy się z osobno działającym serwerem PostgreSQL przez sieć internetową. Znana jest z szerokiego zakresu wbudowanych typów danych i rozszerzalności.

MySQL
MySQL to relacyjny system DBMS klient-serwer, jeden z najczęściej stosowanych w tworzeniu stron internetowych.

MariaDB
MariaDB jest osobnym relacyjnym DBMS, który historycznie powstał jako fork MySQL i pozostaje z nim w dużej mierze kompatybilny.

SQLite
SQLite — wbudowana relacyjna BAZA DANYCH: działa jako biblioteka wewnątrz procesu aplikacji, bez osobnego serwera (sekcja 22.23).

MySQL i MariaDB nie są wymiennymi synonimami
MariaDB powstał jako fork MySQL i pozostaje z nim w dużej mierze kompatybilny, ale są to dwa odrębne projekty o własnej historii rozwoju — nie są identyczne i nie zawsze rozwijają się synchronicznie. W dokumentacji i wakatach czasem są wymieniane razem („MySQL/MariaDB”

Wybór między PostgreSQL, MySQL/MariaDB a SQLite prawie nigdy nie sprowadza się do tego, „który z nich są ogólnie lepiej przygotowane”

Oficjalna dokumentacja: PostgreSQL, MySQL, MariaDB, SQLite.

NoSQL: głównych typów baz danych nierelacyjnych

NoSQL łączy kilka modeli pamięci nierelacyjnej: key-value, dokumenty, szerokokolumnowe przechowywanie oraz wykresy.

NoSQL to powszechna nazwa dla systemów zarządzania bazami danych nierelacyjnymi danych wykorzystujących modele danych inne niż klasyczny model tabeli relacyjnej.

> **NoSQL to "nie tylko SQL", a nie "nigdy SQL"**
> Termin NoSQL często interpretowany jest jako Not Only SQL – „nie tylko SQL”

Relacyjne bazy danych (sekcje 22.21-22.24) przechowują dane w tabelach z predefiniowaną struktura. Czasami taka struktura jest niewygodna, na przykład, jeśli forma wpisów jest bardzo Różni się to w zależności od dokumentu, musisz bardzo szybko pracować nad kluczem lub danymi są naturalnie reprezentowane jako węzły i połączenia. Poniżej przedstawiono cztery główne rodziny NoSQL-dbms, Każdy ma swój własny model danych.

Klucz – Wartość

Każdy wpis jest dostępny pod unikalnym kluczem. Wartością może być łańcuch znaków, liczba, struktura danych lub inny obiekt, w zależności od DBMS.

Przykład: Redis jest serwerem struktur danych, w którym każdy obiekt ma klucz; Redis obsługuje łańcuch znaków tekstów, list, zbiory, skróty, strumienie i inne struktury danych.

primer_redis.txt

```
SET session:42 '{{"user_id": 42, "expires": "2026-08-24"}}'
GET session:42
```

Typowe zastosowania: cache, liczniki, przechowywanie sesji/stanów, kolejki i strumienie danych — często jako uzupełnienie głównej bazy, a nie jej zamiennik.

Bazy danych zorientowane na dokumenty

Dane są przechowywane w dokumentach składających się z pól i wartości. Dokumenty mogą zawierać zagnieżdżone struktury.

Przykład: MongoDB przechowuje dokumenty w BSON formacie, który jest strukturyzowany przez Blisko JSON.

primer_dokument.json

```
{{
  "name": "Механическая клавиатура",
  "price": 8900,
  "characteristics": {{"switches": "red", "layout": "ru"}}
}}
```

Wygodne, gdy forma zapisów może różnić się w zależności od dokumentu.

Bazy danych kolumn szerokich

Dane są zorganizowane wokół kluczy partycji (partition keys) oraz wierszy z zestawem kolumn. Model ten został zaprojektowany do rozproszonego przechowywania dużych ilości danych i z wyprzedzeniem Zaawansowane szablony zapytań.

Przykład: Apache Cassandra jest rozproszonym systemem DBMS o szerokich kolumnach, dostępny w swoim własnym języku zapytań CQL.

primer_wide_column.txt

```
Партиция: device_id = 42
Строки внутри партиции упорядочены по date
```

Typowe zastosowanie: zdarzenia i metryki z prędkością zapisu, rozproszone według urządzenia, użytkownika lub innego klucza partycji.

Bazy danych grafów

Dane są reprezentowane przez węzły, połączenia między nimi oraz właściwości. Model ten jest użyteczny, gdy Same połączenia są ważną częścią zadania.

Przykład: Neo4j jest grafowym DBMS, gdzie zapytania eksplicitnie działają na węzłach i Kontakty.

primer_graf.txt

```
(Ада)-[:ДРУГ]->(Борис)-[:ДРУГ]->(Вера)
```

Typowe zastosowanie: powiązania społeczne, rekomendacje, sieci zależności, analiza powiązań przy wykrywaniu oszustw.

Specjalistyczne systemy przechowywania i odzyskiwania

Istnieją systemy wyszukiwania pełnego tekstu, bazy danych wektorowe i czasowe osobno — nie są Należą do czterech wymienionych powyżej rodzin kanonicznych. Te kategorie mogą się nakładać oraz z już wymienionymi modelami, a niektóre produkty łączą kilka modeli w jeden DBMS (multi-model).

Ogólne cechy systemów NoSQL-

Wiele NoSQL-D baz danych ma bardziej elastyczny schemat danych niż model relacyjny, który nie jest oznacza, że w ogóle nie ma obwodu. Wiele NoSQL-D jest zaprojektowanych z Praca rozproszona i skalowanie poziome, ale konkretne gwarancje (spójność, wsparcie transakcyjne, odporność na błędy) zależą od produktu i jego ustawienia. Wydajność zależy od modelu danych, konkretnego DBMS oraz rodzaju zapytania — Nie ma zasady „NoSQL zawsze jest szybsze.”

Kryteria	Relacyjny DBMS	NoSQL- Systemy
Model danych	Tabele, wiersze, kolumny	Zależy od typu: dokumenty, klucz-wartość, wide-column, wykres
Schemat	Zwykle określane jawnie	Często bardziej elastyczne; Specyficzne dla DBMS
Powiązania	Klucze zewnętrzne, JOIN i inne operacje relacyjne	Implementowane różnie: linki, dokumenty zagnieżdżone, połączenia grafowe i inne

Kryteria	Relacyjny DBMS	NoSQL- Systemy
Język zapytań	SQL i jego dialekty	Natywne języki zapytań, polecenia lub API; niektóre obsługują interfejsy podobne do SQL-
Transakcje	Zależą od DBMS, zwykle rozwinięta obsługa transakcji	Zależy od konkretnego DBMS; wiele nowoczesnych systemów obsługuje również transakcje
Skalowanie	Możliwe architektury pionowe i rozproszone	Wiele systemów początkowo projektuje się do pracy rozproszonej
Kiedy wybrać	Gdy model relacyjny dobrze dopasowuje dane i zapytania	Gdy dany model nierelacyjny lepiej odpowiada danym i obciążeniu

Co wybrać

Relacyjną bazę danych warto rozważyć, gdy dane mają naturalną strukturę relacyjną, istotne są związki i ograniczenia, zapytania dobrze pasują do SQL, a integralność transakcyjna jest ważna dla zadania. Konkretną NoSQL-BD relacyjną warto brać pod uwagę, gdy jeden z modeli — dokumentowy, klucz-wartość, szerokolumnowy lub grafowy — naturalnie pasuje do zadania, wzorzec dostępu do danych jest dobrze znany z góry, a wymagania dotyczące dystrybucji i skalowania skłaniają wybór ku konkretnej nierelacyjnej architekturze.

Dla prostego zastosowania wystarczy jeden odpowiedni DBMS

Duże systemy rzeczywiste często wykorzystują więcej niż jedną technologię pamięci masowej, ale powinieneś rozpocząć projekt szkoleniowy od najprostszej i najbardziej odpowiedniej. W typowym CRUD- zastosowaniu, takim jak lista zadań, nie potrzebujesz jednocześnie PostgreSQL, MongoDB, Redis ani bazy wektorowej, co zwiększa złożoność działania bez realnych korzyści. Ostateczny szkic tego rozdziału (Rozdział 22.35) to jedna tabela na SQLite.

Oficjalna dokumentacja: MongoDB, Redis, Apache Cassandra, Neo4j.

ORM: pracy z bazą danych za pomocą Python obiektów

Wiersze tabeli jako obiekty Python — wygodne, ale ORM i tak tworzy i wykonuje zwykły SQL.

Rozdział 22.22 napisała tekstowe zapytania w SQL-. **ORM** (Object-Relational Mapper — mapowanie obiektowo-relacyjne) — to biblioteka, która pozwala pracować z łańcuch znakówtabelami jak z zwykłymi obiektami Python, a nie tworzyć SQL-tekst ręcznie. Różne ORM mają różny API — oto to samo zapytanie i ta sama wstawka w dwóch najbardziej widocznych ekosystemach ORM i Python:

Akcja	SQL bezpośred-nio	Django ORM	SQLAlchemy 2.x
Przeczytać nie-wykonane za-dania	`SELECT id, title FROM tasks WHERE done = 0`	`Task.ob-jects.filter( done=False)`	`se-lect(Task).wh ere(Task.done .is_(False))`
Dodaj nowe za-danie	`INSERT INTO tasks (title) VALUES (?)`	`Task.ob-jects.create( title="Купить хлеб")`	`ses-sion.add(Task (ti-tle="Купить хлеб"))`

SQLAlchemy (opisuje siebie jako „zestaw narzędzi SQL i konwerter obiektowo-relacyjny dla Python” **ORM Django** jest częścią samego frameworka i jest mu bliski zintegrowane to różne biblioteki o różnych API, a nie dwie nazwy tego samego narzędzi.

ORM nie wyklucza potrzeby zrozumienia SQL

ORM wygodnie eliminuje rutynę ręcznego zbierania tekstu zapytania dla każdej operacji. Ale nie usuwa samego modelu relacyjnego: ORM nadal generuje i wykonuje SQL- zapytań na podstawie operacji obiektowych, a zrozumienie, jak działają tabele, relacje i zapytania (sekcje 22.21-22.22), ma bezpośredni wpływ na efektywność i poprawność ORM — zwłaszcza gdy zapytania przez ORM nie są tak szybkie jak się spodziewasz i trzeba zrozumieć, jakiego rodzaju SQL wykonał.

Dlaczego projekt końcowy wykorzystuje sqlite3

Do wersji roboczej tego rozdziału (Rozdział 22.29) wybrano moduł bezpośredni `sqlite3`, nie ORM: celem tego rozdziału jest zrozumienie, co dokładnie Dzieje się z danymi na każdym etapie, nie ukrywając ich wcześniej za warstwą abstrakcji jak ta abstrakcja stanie się jasna. ORM jest użytecznym i powszechnym narzędziem dla prawdziwych projektów, ale tutaj odciągałby uwagę od samej idei rozdziału.

> **Praktyka: SQL i jego prezentacja w ORM**
> interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji
>
> Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/22-26/index.html)

Migracje schematów baz danych

Struktura tabeli zmienia się razem z aplikacją — migracja zapisuje tę zmianę przewidywalnie.

Słowo „migracja” **migracja schematu** - Zmiana struktury bazy danych z czasem. Drugi, czyli przenoszenie danych między bazami danych, jest omówiony w 22.28.

Aplikacja rośnie, a struktura tabeli zaprojektowana na początku przestaje wystarczać. Na przykład zadanie otrzymuje oznaczenie „wykonano”:

Wersja 1	Wersja 2
tasks(id, title)	tasks(id, title, done)

Migracja schematu — to zapisane zmiany w strukturze, które można zastosować do bazy danych tak samo przewidywalnie na każdym komputerze: u dewelopera lokalnie, u kolegi, na serwerze. W SQL do tego jest komenda `ALTER TABLE`:

migraciya.sql

```
ALTER TABLE tasks ADD COLUMN done INTEGER NOT NULL DEFAULT 0;
```

Stare wiersze nie znikają — każdy wiersz ma nową kolumnę `done` otrzymuje wartość domyślną, `0`.

migraciya.py

```
import sqlite3

baza = sqlite3.connect("zadachi.db")
baza.execute("ALTER TABLE tasks ADD COLUMN done INTEGER NOT
NULL DEFAULT 0")
baza.commit()
```

Narzędzia, które zapisują migracje za Ciebie

W małym projekcie wystarczy wykonać `ALTER TABLE` ręcznie raz. W większym projekcie z wieloma programistami i wieloma środowiska migracji są zazwyczaj formatowane z osobnymi plikami z numerami i kolejnością ich użycia – tak aby każda zmiana w strukturze mogła być zastosowana, śledzona i, jeśli zajdzie taka potrzeba, cofnij się. W ekosystemie SQLAlchemy w tym celu, **Alembic** — osobne narzędzie do migracji schematu, które potrafi porównać aktualną strukturę bazy z opisem w kodzie i wygenerować szkic migracji; taki automatycznie wygenerowany plik — to nie gotowe rozwiązanie, a „szkic migracji” (candidate migration)), który trzeba sprawdzić i w razie potrzeby ręcznie poprawić przed zastosowaniem. W Django migracje są wbudowane w sam framework: polecenie `makemigrations` tworzy pliki migracji dla wykryte zmiany modelu, oraz `migrate` stosuje je do baza danych.

migracja schematu dotyczy struktury, a nie treści

migracja schematu odpowiada na pytanie "jak tabela jest teraz ułożona", a nie "co przeniosło się z jednej bazy do drugiej". To zasadniczo inne zadanie niż migracja danych, która zostanie omówiona w Sekcji 22.28 — nie myl tych dwóch znaczeń słowa "migracja", gdy spotykasz je w prawdziwych projektach.

Praktyka: migracja schematu SQLite

interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/22-27/index.html)

Transfer danych między bazami danych

Nie chodzi o strukturę tabeli, lecz o same rekordy — jak są one przekazywane między bazami danych i dlaczego transfer jest koniecznie sprawdzany.

Drugie znaczenie słowa «migracja» — **migracja danych** między bazami lub systemami: na przykład, ze starej bazy do nowej, z jednego formatu przechowywania do drugiego. Nie mieszaj z migracją schematu z rozdziału 22.27 — tam zmieniano strukturę tabeli, tutaj przenoszone są same rekordy.

Przenoszenie danych: odczyt, przekształcenie, zapis, sprawdzenie

JSON (rozdział 22.15) jest wygodny jako *średniozaawansowany* format w tym procesie jest taki, że on tekstowy, czytelny dla człowieka i łatwy do odczytania w każdym języku programowania. Ale JSON — nie uniwersalny format bezstratnej łącznicy: nie ma osobnego typu dla dat, dla dane binarne, a także nie przechowuje ograniczeń i relacji między tabelami (klucze obce, Rozdział 22.21) – muszą być przywracane osobno po stronie docelowej bazy.

Mały powtarzalny przykład

eksport_v_json.py

```
import json
import sqlite3

istochnik = sqlite3.connect("zadachi.db")
istochnik.row_factory = sqlite3.Row

stroki = istochnik.execute("SELECT title, done FROM
tasks").fetchall()
dannye = [dict(stroka) for stroka in stroki]

with open("zadachi.json", "w", encoding="utf-8") as fajl:
    json.dump(dannye, fajl, ensure_ascii=False, indent=2)
```

import_iz_json.py

```
import json
import sqlite3

with open("zadachi.json", encoding="utf-8") as fajl:
    dannye = json.load(fajl)

cel = sqlite3.connect("novaya_zadachi.db")
cel.execute("CREATE TABLE tasks (id INTEGER PRIMARY KEY, title
TEXT NOT NULL, done INTEGER NOT NULL DEFAULT 0)")
for zapis in dannye:
    cel.execute(
        "INSERT INTO tasks (title, done) VALUES (?, ?)",
        (zapis["title"], zapis["done"]),
    )
cel.commit()
```

Identyfikatory to rozwiązanie, które wymaga wyraźnego przemyślenia i zweryfikowania

W powyższym przykładzie nowa baza sama przypisuje nowe wartości `id` po wstawieniu stare ID nie są migrowane. To świadomy wybór, a nie przypadek: jeśli ważne jest, aby aplikacja zachowała oryginał `id` (na przykład są odwoływane zewnętrznie), muszą być migrowane jawnie, a następnie weryfikowane, że pozostają unikalne. Każda migracja danych musi zakończyć się sprawdzeniem, czy liczba wierszy się zgadza oraz czy reprezentatywne rekordy w źródle i docelowym się zgadzają.

Transfer między plikami lokalnymi SQLite w tym przykładzie odtwarza tę samą zasadę, i przenieść się do przemysłowej bazy takiej jak PostgreSQL — czytać, transformować, pisać, Sprawdzone. Różnią się tylko konkretne narzędzia i sterowniki połączenia; po pierwsze — Obowiązkowa kopia kopii zapasowej danych źródłowych.

Praktyka: Migracja danych przez JSON

interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/22-28/index.html)

Flask i SQLite: zapisują dane w bazie danych

Lista Python w sekcji 22.5 staje się tabelą SQLite, która przetrwa restart.

Teraz lista Python z sekcji 22.5 został zastąpiony bazą danych SQLite. Schemat jest prosty — Jedna tabela:

schema.sql

```
CREATE TABLE IF NOT EXISTS tasks (
    id INTEGER PRIMARY KEY,
    title TEXT NOT NULL,
    done INTEGER NOT NULL DEFAULT 0 CHECK (done IN (0, 1))
);
```

`CHECK (done IN (0, 1))` jest dodatkowym ograniczeniem: SQLite nie pozwoli ci pisać do `done` nic poza 0 lub 1, Nawet jeśli gdzieś w kodzie aplikacji pojawi się jakiś błąd.

Jedno połączenie na żądanie

Otwieranie połączenia z bazą przy każdym małym połączeniu to marnotrawstwo, ale utrzymanie jednego Wspólne połączenie w całej aplikacji bez ostrożności jest ryzykowne przy równoległych rozwiązaniach zapytania. Flask oferuje wygodne miejsce na dane, które muszą być dokładnie w jednym miejscu zapytanie, jest obiektem `g`:

app.py

```
import sqlite3
from flask import Flask, g

app = Flask(__name__)
app.config["DATABASE"] = "zadachi.db"


def get_db():
    if "db" not in g:
        g.db = sqlite3.connect(app.config["DATABASE"])
        g.db.row_factory = sqlite3.Row
    return g.db
```

```
@app.teardown_appcontext
def close_db(exception=None):
    db = g.pop("db", None)
    if db is not None:
        db.close()
```

g jest powiązany z kontekstem aplikacji, a nie dosłownie z żądaniem

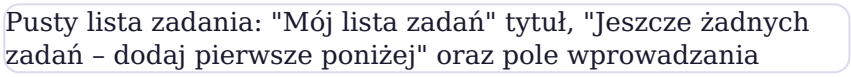
`g` przechowuje dane w *kontekst aplikacji* Flask. W normalnym przetwarzaniu HTTP- żądania ten kontekst występuje dokładnie od początku jego zakończenia i jest tworzony na nowo dla każdego kolejnego żądania — zatem w praktyce, `g` zachowuje się jak magazyn jednego zapytania, a nie jak wspólna zmienna, z której dane mogą przypadkowo „wyciec” do innego zapytania. `teardown_appcontext` jest wywoływane, gdy kontekst aplikacji jest zamknięty, nawet jeśli handler zawiódł, połączenie jest zawsze zamykane.

Trasy na górze bazy danych

app.py

```
@app.route("/")
def glavnaya():
    db = get_db()
    zadachi = db.execute("SELECT id, title, done FROM tasks
ORDER BY id").fetchall()
    return render_template("index.html", zadachi=zadachi)


@app.route("/dobavit", methods=["POST"])
def dobavit():
    title = clean_title(request.form.get("zadacha", ""))
    if title is None:
        flash(„Nazwa zadania nie może być pusta i nie dłuższa
niż 200 znaków.”)
        return redirect(url_for("glavnaya"), code=303)
    db = get_db()
    db.execute("INSERT INTO tasks (title) VALUES (?)",
(title,))
    db.commit()
    return redirect(url_for("glavnaya"), code=303)
```

Proszę zauważyć: `?` w SQL- zapytaniu i wartość jako osobny argument — to samo parametryzowane zapytanie z rozdziału 22.22, teraz wewnątrz aplikacji roboczej. `flash(...)` zapisuje pojawioną wiadomość Na następnej stronie po przekierowaniu następna pokazuje dokładnie, jak to działa Rozdział, 22:30. `clean_title(...)` jest prostą funkcją kontrolną z Rozdział 22.31: Pusta lub zbyt długa nazwa jest odrzucana przed dostępem do bazy danych.



Lista zadań bezpośrednio po init_db() – Tabela została utworzona, ale jest pusta.

Po wysłaniu formularza: INSERT przeszedł, przekierowanie wróciło na / — zadanie jest widoczne na liście.

Po kliknięciu na kółko: UPDATE tasks SET done = 1 – łańcuch znaków oznaczone jako ukończone.

Pełny plik roboczy — ze wszystkimi trasami, obsługą błędów i API — można znaleźć tutaj:

projects/flask/todo-app/app.py

Sprawdź sam: restartuj – zadania nadal są

Uciekaj `python app.py` dodaj kilka zadań, zatrzymaj proces (Ctrl+C) i zacznij od nowa. Lista pozostanie taka sama – dokładnie to, czego brakowało wersji w sekcji 22.5.

Prawdziwa kontrola: proces Flask został całkowicie zatrzymany i wznowiony na tym samym pliku bazy danych. „Kup chleb”

Praktyka: Flask na SQLite
Moduł flask nie jest zainstalowany w środowisku przeglądarki Pyodide – działa lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/22-29/index.html)

Cookie i sesje: Status między HTTP- żądaniami

LCC1ZCCquery są od siebie niezależne — mechanizmy cookie i sesji pozwalają na powiązanie stanu między nimi.

HTTP- żądania są niezależne od siebie: sam protokół nie wiąże pojedynczego żądania z następującymi. Ale aplikacja w sekcji 22.29 już musi przechowywać coś pomiędzy żądaniami — Na przykład komunikat o błędzie, który powinien pojawić się na następnej stronie `flash(...)`. W tym celu organizowane są cookie i sesje mechanizmy.

Cookie jest niewielką wartością, którą przeglądarka przechowuje

Cookie to mały fragment danych, o który serwer prosi przeglądarkę zapisuje, a przeglądarka następnie stosuje je do kolejnych żądań do tej samej strony. Dokładnie Serwer może powiązać wiele żądań ze sobą jako pochodzące z tego samego przeglądarka.

Cookie przesyłany jest razem z każdym żądaniem do tej samej strony

Sesja — mechanizm przechowywania stanu między zapytaniami

Sesja (session) to mechanizm na poziomie aplikacji, który wykorzystuje cookie łączyć wiele żądań ze sobą i przechowywać część danych. Implementacje są różne: dane sesji mogą być przechowywane na serwerze (w bazie danych lub osobnej pamięci masowej, i tylko identyfikator może być przekazany do cookie), albo możesz zrobić to bezpośrednio do cookie po stronie przeglądarki. Flask domyślnie wybiera drugą opcję i zapewnia gotowy obiekt `session`, który zachowuje się jak słownik:

session_primer.py

```
from flask import session

session["poslednij_vizit"] = "22-30"
# przy następnym żądaniu tej samej przeglądarki:
session.get("poslednij_vizit")   # "22-30"
```

Funkcja `flash(...)` użyte w sekcji 22.29 Na tym samym session: mechanizmie wiadomość jest przechowywana w sesji dla kolejnego żądania i jest automatycznie usuwany po przeczytaniu `get_flashed_messages()` w szablonie.

Podpisany nie oznacza szyfrowania
Domyślnie Flask przechowuje zawartość sesji bezpośrednio w cookie przeglądarki, chronioną sygnaturą `SECRET_KEY`. Podpis gwarantuje **autentyczność**: Jeśli ktoś zmieni treść cookie, podpis przestanie się zgadzać i Flask odrzuci sesję. Ale podpis — **nie szyfrowanie**: Domyślnie zawartość sesji można odczytać bez znajomości tajnego klucza — po prostu nie można go niewykrywalnie zmienić. Nigdy nie umieszczaj haseł ani innych danych w sesji, które powinny pozostać nieczytelne dla użytkownika, którego przeglądarka je przechowuje.

Szkic tego rozdziału celowo się kończy tutaj: pełne logowanie użytkownika (uwierzytelnienie) dodałoby do rozdziału tom osobnego kursu. Gdzie temat się toczy dalej — Więcej informacji można znaleźć w sekcji 22.36.

Praktyka: Komunikacja przez flash() i session
Moduł flask nie jest zainstalowany w środowisku przeglądarki Pyodide – działa lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/22-30/index.html)

Walidacja danych wejściowych i obsługa błędów

Formularz w przeglądarce to nie jedyny sposób wysłania żądania, dlatego weryfikacja na serwerze jest obowiązkowa.

Rozdział 22.4 pokazywała weryfikację formularza w przeglądarce za pomocą JavaScript. To jest przydatne dla użytkownik, ale nigdy jedyna linia obrony: żądanie może być wysłane bez żadnych przeglądarka —

na przykład program, który bezpośrednio uzyskuje dostęp do `/dobavit` lub `/api/tasks`, bez żadnej linii JavaScript. Input Dane muszą być ponownie weryfikowane po stronie serwera w każdej trasie, którą akceptuje.

> **Nie ufaj opinii tylko dlatego, że pochodzi z twojej formy**
> Formularz HTML z sekcji 22.13 ustawia `required` oraz `maxlength` — ale to są tylko powiadomienia przeglądarkowe, a nie gwarancja. Nic nie stoi na przeszkodzie, by wysłać POST- żądanie do `/dobavit` bezpośrednio, omijając formę i wszelkie jej ograniczenia.

Sprawdzenie w projekcie rozdziału

app.py

```
MAX_TITLE_LENGTH = 200


def clean_title(raw_title):
    title = raw_title.strip()
    if not title or len(title) > MAX_TITLE_LENGTH:
        return None
    return title
```

`strip()` usuwa przypadkowe spacje na końcach (na przykład, jeśli użytkownik nacisnął spację, a następnie Enter). Pusta łańcuch znaków po tym i za długa łańcuch znaków — oba przypadki są uznawane za nieprawidłowe i zwracają `None`.

HTML-trasa i API-trasa reagują na błąd inaczej

	HTML-form (/dobavit)	API (/api/tasks)
Kto czyta odpowiedź	Człowiek w przeglądarce	Program
Reakcja na nieprawidłowe dane	flash(...) z tekstem czytelnym i przekierowaniem z powrotem (sekcja 22.30)	kod 400 i JSON ciało z opisem błędu

	HTML-form (/dobavit)	API (/api/tasks)
Przykładowa odpowiedź	Strona z listą zadań i wiadomością z góry	`{{"error": "title не может быть пустым"}}`

Lista zadań z różowym komunikatem na górze: „Nazwa zadania nie może być pusta i nie dłuższa niż 200 znaków.”

Prawdziwy efekt złożenia formularza tylko z odstępami: clean_title() zwrócił None, flash() pojawiła się wiadomość, przekierowanie wróciło na tę samą stronę, a zadanie nie zostało dodane.

app.py

```
data = request.get_json(silent=True)
if not isinstance(data, dict) or not
isinstance(data.get("title"), str):
    return jsonify({{"error": „title pole musi i musi być
łańcuch znaków za linią”}}), 400
title = clean_title(data["title"])
if title is None:
    return jsonify({{"error": „title nie może być pusty i nie
dłuższy niż 200 znaków”}}), 400
```

404 — odwołanie do czegoś, czego nie ma

Jeśli spróbujesz oznaczyć jako wykonane lub usunąć zadanie nieistniejące `id` trasa wzywa `abort(404)` — Flask natychmiast zwraca standardową stronę błędu (lub JSON dla ścieżek API-, sekcja 22.9), bez dalszego wykonywania kodu.

Praktyka: Zweryfikować dane wejściowe
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/22-31/index.html)

Podstawy bezpieczeństwa aplikacji internetowych

Kilka typowych błędów i zasad, które musisz znać nawet przy małym projekcie.

Bezpieczeństwo w sieci to duży temat; oto tylko najczęstsze błędy i zasady, Że musisz wiedzieć nawet przy małym projekcie. Deep learning jest na przyszłość kurs specjalizacyjny (sekcja 22.36).

SQL- Zastrzyk

Jeśli złożysz zapytanie SQL-, łącząc łańcuch znaków tekstów z wejściem użytkownika, część wpisanej Tekst może być postrzegany jako część samego zapytania, a nie jako dane:

opasno.py

```
# NIE MOŻESZ TEGO ZROBIĆ
db.execute(f"SELECT * FROM tasks WHERE title =
'{{user_input}}'")
```

bezopasno.py

```
db.execute("SELECT * FROM tasks WHERE title = ?",
(user_input,))
```

Ograniczanie execute() nie czyni f-stringów bezpiecznymi
Metoda `sqlite3.execute()` w Python nie pozwala na kilka SQL-instrukcji w jednym wywołaniu: atak złożony taki jak `x'; DROP TABLE tasks; --` zawiodę `ProgrammingError`. To ograniczenie konkretnego sterownika Python-, a nie uniwersalna ochrona SQL. To nie chroni przed wstrzyknięciami jednoskładnikowymi: wprowadzenie `' OR '1'='1` zmienia stan `WHERE` i zwraca wszystkie linie. Nigdy nie wstawiaj niepewnego wejścia do tekstu SQL — przekazuj je tylko jako parametr.

Rozdział 22.22 już pokazała tę technikę — **parametryzowane zapytanie**, gdzie znaczenie jest przekazywane oddzielnie od SQL tekstu i nigdy nie jest z nim mieszane. Moduł `sqlite3` odpowiada za to, by wartość pozostała danymi, Cokolwiek w niej jest, nawet fragment wyglądający jak SQL.

XSS — Skrypty międzyokręgowe

Jeśli obcy tekst trafi na HTML- stronę bez przetworzenia, może zawierać kod wykonywalny, który zostanie uruchomiony w przeglądarce innego użytkownika — na przykład, `<script>...</script>` w nazwie zadania. Rozdział 22.12 już wykazało ochronę: w

kontekście HTML- z auto-shielding Jinja zastępuje specjalny znaki HTML odpowiadające mu sekwencje escape-, więc wpisany tekst nie jest interpretowane jako marż. To nie czyni bezpiecznego wstawiania wartości gdziekolwiek — na przykład wewnątrz `<script>` lub URL potrzebuje już innej ochrony, — właśnie dlatego filtr `|safe`, co zazwyczaj jest wyłącza escaping, nie może być zastosowana do wejścia, nad którym nie masz pełnej kontroli siebie.

CSRF - Fałszerstwo żądań międzysymetrycznych

CSRF (Cross-Site Request Forgery) jest istotne, gdy przeglądarka działa automatycznie poświadczenia (np. sesja cookie, sekcja 22.30) dla żądań, a samo żądanie się zmienia Status serwera: Użytkownik zalogowany na stronie otwiera inną, złośliwą stronę Page, a teoretycznie ten Page mógłby wysłać prośbę w jego imieniu do Ciebie Strona — sama przeglądarka będzie stosować ten sam cookie. Dopóki w szkicu rozdziału nie ma logowania użytkownika, Ryzyko CSRF ograniczone, ale w zastosowaniach, gdzie autoryzacja lub status zależy od Automatycznie wysyłane przez przeglądarkę cookie zmieniające status żądań muszą być są chronione przed CSRF przez odpowiedni mechanizm, zwykle zapewniony przez sam ramowy system, lub Ekspansja.

Hasła

Hasła nigdy nie są przechowywane jako tekst zwykły, a jedynie jako skrót uzyskany przez Sprawdzone narzędzie stworzone specjalnie do haseł. Wymyśl własne Schemat haszowania haseł nie jest potrzebny i nie jest tego wart — są gotowe, starannie przygotowane zaufanych bibliotek. Projekt tego rozdziału w ogóle nie przechowuje haseł: sekcja 22.30 już wyjaśniono, dlaczego pełne logowanie użytkownika wykraczało poza zakres tego rozdziału.

Sekrety

`SECRET_KEY` aplikacje, hasła do baz danych, klucze zewnętrzne API — wszystko to nie powinno trafiać do publicznego kodu źródłowego. Zazwyczaj takie wartości Przechodzić przez zmienne środowiskowe zamiast zapisywać je bezpośrednio do pliku z kodeks, w sekcji 22.34.

HTTPS — znowu, krótko

Rozdział 22.10 już wyjaśniła: HTTPS chroni dane *po drodze* między przeglądarką a ale nie naprawia błędów w samej aplikacji – SQL-wstrzykiwania, XSS czy wycieku HTTPS nie powstrzyma tajemnic. To różne, uzupełniające się poziomy ochrony.

Praktyka: Parametryzacja vs. wstrzyknięcie
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/22-32/index.html)

Automatyczne testowanie Flask-aplikacji

Klient testowy sprawdza logikę aplikacji szybko i bez odwoływania się do sieci.

Każdą zmianę ręcznie sprawdzam w przeglądarce, powoli i łatwo zapominam powtórzyć po następnej edycji. Flask ma wbudowany **klient testowy** — on Pozwala wysyłać żądania do aplikacji z kodu testowego i weryfikować odpowiedź.

Klient testowy nie otwiera portu sieciowego
`app.test_client()` nie tworzy połączenia sieciowego HTTP-. Tworzy zapytanie wewnątrz procesu i przekazuje je bezpośrednio do obsługującego Flask przez testowy interfejs biblioteki Werkzeug, na której zbudowany jest Flask — szybko i bez sieci, ale z tą samą logiką przetwarzania zapytania. Pełna ścieżka zapytania przez przeglądarkę i sieć była pokazana w rozdziale 22.7 — testowy klient celowo pomija właśnie tę część.

test_app.py

```
import app as todoapp

def test_glavnaya_pustoj_spisok(baza_dannyh):
    client = todoapp.app.test_client()
    otvet = client.get("/")
    assert otvet.status_code == 200
    assert „Jeszcze żadnych zadań” in
otvet.get_data(as_text=True)
```

Sprawdzanie odpowiedzi JSON-

test_api.py

```
def test_api_tasks_vozvrashchaet_json(baza_dannyh):
    client = todoapp.app.test_client()
    otvet = client.get("/api/tasks")
    assert otvet.status_code == 200
    assert otvet.content_type == "application/json"
    dannye = otvet.get_json()
    assert isinstance(dannye, list)
```

`otvet.get_json()` sam analizuje ciało odpowiedzi jako JSON — to to samo, co `json.loads(otvet.get_data())` z sekcji 22,15, ale krócej.

Oddzielna baza danych do testów

Testy nie powinny ingerować w tę samą bazę danych, na której pracujesz ręcznie, — w przeciwnym razie przebieg testów może usunąć lub uszkodzić zapisane w niej zadania. Przed każdym testem tworzona jest tymczasowa baza danych, a po nim — usuwana:

conftest.py

```
import tempfile
import os
import pytest

import app as todoapp


@pytest.fixture
def baza_dannyh():
    fd, put = tempfile.mkstemp()
    todoapp.app.config["DATABASE"] = put
    with todoapp.app.app_context():
        todoapp.init_db()
    yield put
    os.close(fd)
    os.unlink(put)
```

Pełny zestaw testów projektu — w `tests/test_chapter22_todo_app.py` w repozytorium książek; sprawdza trasy, zapisuje je do bazy danych, waliduje JSON API oraz ochrona przed zastrzykami SQL- i XSS z sekcji 22.31-22.32.

Praktyka: Testy dla aplikacji Flask-a.
Moduł flask nie jest zainstalowany w środowisku przeglądarki Pyodide – działa lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/22-33/index.html)

Rozwijanie Flask-aplikacji

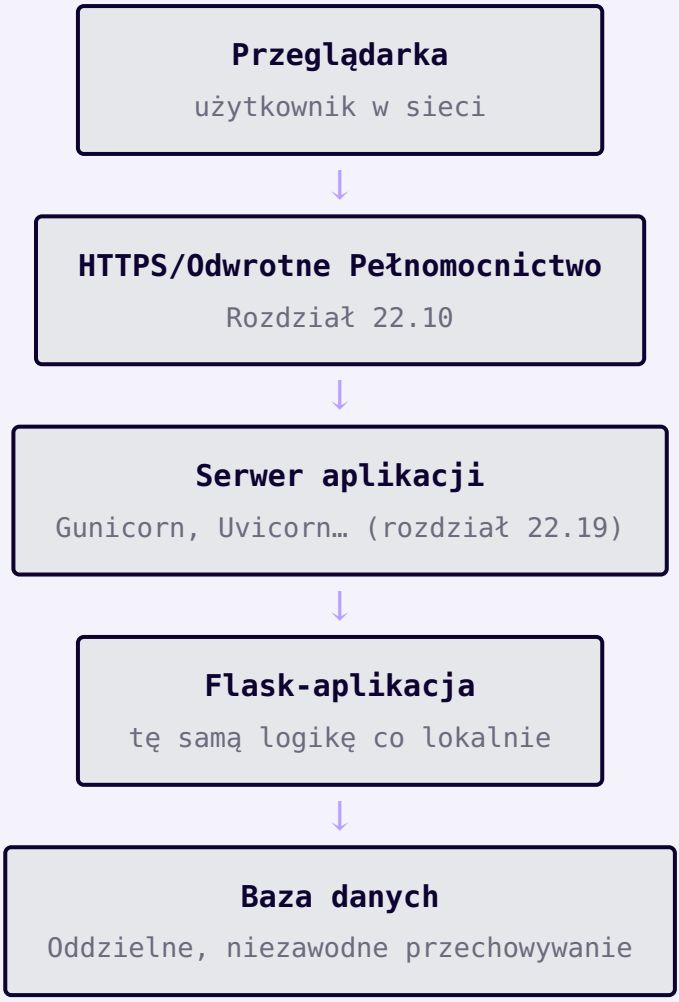
Serwer deweloperski jest Flask wygodny lokalnie, ale nie dla czegoś, co jest dostępne dla całego Internetu.

Rozdział 22.5 uruchamiał aplikację poleceniem `python app.py`, która wewnątrz wywołuje `app.run(debug=True)`. To **serwer deweloperski** (development server) jest wbudowany w Flask, wygodny do eksploatacji lokalnej, ale nie przeznaczonej do rzeczywistej eksploatacji. Rozdział „Deploying to Production”

	Środowisko rozwojowe	Wdrożenie robocze
Kto go używa	Wy sami, lokalnie	Prawdziwi użytkownicy przez Internet
Serwer	Wbudowany serwer Flask deweloperski	Oddzielny serwer aplikacji (Rozdział 22.19)
debug=True	Wygodne – szczegółowe błędy, automatyczny restart	Niebezpieczny – Może pokazać szczegóły i dane wewnętrznego kodu osobom z zewnątrz
Sekrety (SECRET_KEY itd.)	Można użyć tymczasowej wartości do lokalnej pracy	Wymagana jest osobna, niepublikowana wartość

debug=True w otwartej aplikacji — prawdziwa luka
W trybie debugowania Flask może pokazać interaktywny debugger z pełnym dostępem do wykonania kodu bezpośrednio w przeglądarce, gdy błąd nie zostanie rozwiązany. To ogromna lokalna wygoda i poważne ryzyko, jeśli aplikacja jest dostępna dla kogoś innego — tryb debugowania powinien być wyłączony w każdym wdrożeniu, które nie jest dostępne tylko dla Ciebie.

Ścieżka żądań do aplikacji produkcyjnej



Ten sam kod Flask — ale między przeglądarką a nim jest teraz kilka dodatkowych warstw

Konfiguracja za pomocą zmiennych środowiskowych

Szkic tego rozdziału już odczytuje ścieżkę do bazy danych oraz tajny klucz ze zmiennych środowisko, z wyraźną wartością wpadkową dla rozwoju lokalnego:

app.py

```
app.config["DATABASE"] = os.environ.get("TODO_APP_DB",
str(DEFAULT_DB_PATH))
app.config["SECRET_KEY"] = os.environ.get(
    "TODO_APP_SECRET_KEY", "dev-only-secret-do-not-use-in-
production"
)
```

Ten sam kod działa zarówno lokalnie (bez jednej konfiguracji), jak i w innym środowisku — W razie potrzeby wartości zmiennych środowiskowych są po prostu ustawiane inaczej, bez wpływu na sam kod.

Logi, wiele procesów, trwałość

Istnieją inne praktyczne kwestie związane z pracą w misji, które wykraczają poza tego rozdziału: **logi** (zapisana historia tego, co się dzieje, aby rozłożyć problemy po fakcie), start **wiele procesów lub wątków** jednocześnie do obsługi równoległych żądań, i fakt, że baza danych musi fizycznie znajdować się w niezawodnym, nie znikającym magazynie. Świadomie nie zamieniamy tego rozdziału w podręcznik użytkowania — gdzie ten temat jest kontynuowany, pokazuje rozdział 22.36.

Oficjalna dokumentacja: Flask — Deploying to Production.

Projekt końcowy: lista zadania dla Flask i SQLite

To samo zadanie, które lista rozpoczęło ten rozdział, teraz z trwałym przechowywaniem, JSON API i walidacją danych wejściowych.

Wszystko, co analizowano w sekcjach 22.7-22.34, łączy się w jednej działającej aplikacji — tej samej liście zadań, od której rozpoczął się rozdział, teraz ze stałym przechowywaniem w SQLite.

projects/flask/todo-app/app.py

Plik	Rola
app.py	Trasy, współpraca z bazą danych, API
schema.sql	Struktura tabeli tasks
templates/base.html	Ogólnie strona wireframe, w tym flash() posty
templates/index.html	Lista zadań, dodaj formularz
templates/privet.html	Historyczna strona powitalna z sekcji 22.5
templates/404.html	Strona Nieznaleziona
static/style.css	Style niestandardowe, bez zewnętrznych CDN

Co może zrobić gotowy wniosek

Pięć możliwości jednej małej aplikacji

LISTA ZADAŃ

GET / – pokazuje wszystkie zadania
Pusty stan, jeśli brak zadań

DODAWANIE

POST /dobavit
Walidacja danych wejściowych (Rozdział 22.31)
POST-Redirect-GET (Rozdział 22.13)

ZNACZNIK WYKONANIA

POST /vypolnit/
Przełączniki done między 0 a 1

USUWANIE

POST /udalit/
Całkowicie usuwa łańcuch znaków z bazy danych

JSON API

GET /api/tasks – lista
POST /api/tasks – dodawanie z ciałem JSON-

Podsumowanie lista zadań w przeglądarce: dwa ukończone (przekreślone) zadania na górze i dwa jeszcze nieukończone poniżej, pole dodawania oraz przycisk „Powiedz cześć”

Lokalnie uruchomiona aplikacja Flask w przeglądarce — końcowy wygląd projektu: wykonane i niewykonane zadania, formularz dodawania oraz link do strony historycznej /privet/<imię> z rozdziału 22.5.

Usunięcie i Zakończenie również są POST, a nie GET
Rozdział 22.9 już wyjaśniła: GET nie powinno zmieniać stanu na serwerze, ponieważ przeglądarka może go powtórzyć bez ostrzeżenia, na przykład podczas odświeżania strony. Dlatego przyciski "Wykonane" i "Usuń" w szablonie mają własne małe formularze `method="post"` zamiast prostych odniesień `<a href="/udalit/…">`.

Uruchomienie

terminal.txt

```
cd projects/flask/todo-app
python app.py
```

Pierwsze uruchomienie samo tworzy plik bazy danych i tabelę — nie jest potrzebny oddzielny polecenie do tego. Otwórz `http://127.0.0.1:5000/` : dodaj zadanie, Oznacz jako zakończony, usuń ją, odśwież stronę w trakcie procesu, samodzielnie ją zrestartuj Przetwarzaj i upewnij się, że lista pozostaje bez zmian.

Aplikacja działa także na wąskich ekranach: układ CSS (sekcja 22.3) został przebudowany dla szerokości telefonu komórkowego bez poziomego przewijania strony.

★★ Zadanie samodzielne
Licznik pozostałych zadań
Dodaj w index.html wiersz z liczbą niewykonanych zadań — {{ zadachi|selectattr('done', 'equalto', 0)|list|length }}.

Challenge
Sortowanie według statusu
Zmień zapytanie w glavnaya() tak, aby nieukończone zadania zawsze były wyświetlane przed ukończonymi — ORDER BY done, id.

Praktyka: praca z gotowym projektem
Moduł flask nie jest zainstalowany w środowisku przeglądarki Pyodide – działa lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/22-35/index.html)

Dalsze badania nad tworzeniem stron internetowych na Python

Od pierwszego HTTP- żądania do działającej aplikacji z bazą danych — i dokąd iść dalej.

Co jest teraz jasne po tym rozdziale

- Jak działa dialog przeglądarki i serwera: prośba, odpowiedź, HTTP i HTTPS.
- Z czego składa się frontend — HTML, CSS, JavaScript — i co każdy z nich robi.
- Jak Flask zamienia Python funkcje w HTTP- obsługi żądań za pomocą tras.
- Jak szablony Jinja tworzą HTML z danych i dlaczego automatyczne maskowanie jest domyślnie włączone.
- Czym są JSON i API oraz czym różni się odpowiedź z danymi od HTML-strony.
- Czym różnią się między sobą Flask, Django i FastAPI — i czym są WSGI i ASGI.
- Dlaczego potrzebujesz bazy danych, czym jest SQL i czym bazy relacyjne różnią się od tych nierelacyjnych.
- Czym jest migracja schematu i migracja danych — i dlaczego są to dwa różne znaczenia tego samego słowa.
- Podstawy bezpieczeństwa: parametryzowane zapytania, automatyczne osłony, sekrety
- Jak testować aplikację Flask- za pomocą testów i czym serwer deweloperski różni się od wdrożenia produkcyjnego.

Przeszedłeś od HTTP wiadomości między przeglądarką a serwerem do pracy Aplikacje z bazą danych, JSON API i testami to solidna podstawa, ale nie wszystkie Tworzenie stron internetowych. Osobny kurs Cartesian School „Web Development on the Python: from HTTP to deployment”

Czym zajmie się przyszły specjalistyczny kurs

FRAMEWORKS DEEPER

Architektura Flask poza podstawami
Django w całości
FastAPI i napisałem API

BAZY DANYCH

PostgreSQL w praktyce
SQLAlchemy i Alembic
ORM i Django migracyjne

API I WYMIANA DANYCH

REST szczegółowo
OpenAPI
Uwierzytelnianie i autoryzacja

RÓWNOLEGŁOŚĆ I CZAS RZECZYWISTY

Python asynchroniczny
ASGI w praktyce
WebSocket

INFRASTRUKTURA SERWISU

```
Zadania w tle
Redis i buforowanie
Docker, odwrotne proxy, CI/CD
```

EKSPLOATACJA

```
Konfiguracja i tajemnice
Logi, metryki, obserwowalność
Wdrożenie robocze
```

A kolejny rozdział książki całkowicie wraca do mini-projektów stołowych na Python Druga część języka to działanie.

ROZDZIAŁ 23 · STR. 1509

Pierwszy projekt na GitHub: SafeSort

Napisz program, który uporządkuje pliki i doprowadzi je do stanu gotowego do GitHub: testów, historii zatwierdzeń Pull Request.

Co stworzymy: SafeSort

Program, który porządkuje pliki — najpierw pokazuje, co zrobi, a dopiero potem wykonuje to na polecenie.

SAFESORT · CZĘŚĆ 1 Z 6

Git i GitHub → Planowanie → Projekt →

Wdrożenie → Testy i CI → Premiera

Jak dotąd pisaliśmy programy częściowo: funkcje, gry, interfejsy. Teraz Złóż jeden projekt w całości i doprowadź go do stanu, który można umieścić na GitHub. Program nazywa się **SafeSort**, i oto co robi.

TAK BYŁO

Downloads/

report.pdf

photo.jpg

archive.zip

notes.txt

→

copy_of_notes.txt

STAŁO SIĘ

Downloads/

Sorted/

```
documents/
    report.pdf
    notes.txt
    copy_of_notes.txt
```

```
images/
    photo.jpg
archives/
    archive.zip
```

notes.txt i copy_of_notes.txt — ta sama zawartość; SafeSort poinformuje o tym osobno, a nie usunie pliku samodzielnie.

Na początku program pokazuje tylko to, co ma zrobić. Pliki przeniesienia może być tylko osobnym, wyraźnie nazwanym poleceniem:

TYLKO CZYTAĆ

`scan` – pokazuje znalezione pliki

`plan` - Pokazuje plan podróży

`duplicates` – pokazuje dopasowania

ZMIENIAJĄ PLIKI

`apply` - Przenosi pliki

`undo` – przywraca pliki na miejsce

```
~/safesort $ safesort plan ~/Downloads
5 move operations planned.
No files have been changed.
```

System plików po tym poleceniu pozostał taki sam — `plan` opisuje jedynie przyszłe ruchy. To rozróżnienie między "pokazem" a "zrobione" jest Główna idea całego projektu i wrócimy do niej więcej niż raz.

Co powinna umieć pierwsza wersja

Program powinien potrafić:

- wyświetlają znalezione pliki posegregowane według kategorii;
- pokazać plan przyszłych ruchów bez żadnych zmian;
- wykonywać ruchy tylko na wyraźne polecenie;
- znaleźć pliki o tej samej treści;
- cofnąć ostatnią wykonaną operację.

Jak powinien się zachowywać program:

- Nie modyfikuj plików bez wyraźnego polecenia.
- nie nadpisują istniejącego pliku bezszelestnie;
- ten sam wejście daje ten sam plan, nie ma tu losowości;
- całe przetwarzanie odbywa się lokalnie, bez sieci;
- logikę można sprawdzić automatycznymi testami, nie dotykając prawdziwych plików.

Pierwszy lista w rozwoju nazywa się **funkcjonalny Wymagania** — co program robi. Drugi — **niefunkcjonalne**: nie «co», lecz «jak», jakie właściwości posiada zachowanie programu niezależnie od polecenia.

Czego nie zrobimy w pierwszej wersji

Aby ukończyć projekt i nie rozciągać go w nieskończoność, natychmiast ograniczmy zadanie:

Nie wchodzi w pierwszą wersję	Dlaczego
Automatyczne usuwanie duplikatów	Usuwanie danych bez wyraźnego potwierdzenia — ryzyko, a nie wygoda
GUI	Linia poleceń jest łatwiejsza do implementacji, testowania i wyjaśniania
Przesyłanie plików do chmury	Program działa tylko z lokalnym systemem plików
Klasyfikacja według zawartości pliku	Klasyfikacja przez rozszerzenie już rozwiązuje główny problem

Krótko

- scan, plan i duplicates odczytują tylko system plików; apply sortuje pliki, a undo wykonuje ruch odwrotny.
- "Pokaż plan" i "wykonaj plan" to różne etapy, a ta różnica definiuje całą architekturę programu.
- Wymagania funkcjonalne — co robi program; niefunkcjonalne — jakie właściwości ma jego zachowanie.

Git≠GitHubCzym jest Git i czym jest GitHub

Git i GitHub nie są synonimy: Git działa lokalnie, a GitHub zapewnia usługę sieciową na górze repozytorium.

SafeSort · Część 1 z 6**Git i GitHub**

≠ GitHub

Poprzednia sekcja pokazała, co SafeSort. zrobić Zanim napiszesz kod, przyjrzyjmy się narzędzi, wokół których zbudowana jest reszta rozdziału, i to z czterema podobne słowa, które łatwo pomylić. Git i GitHub: różne produkty, I nie dwa imiona tego samego.

Koncepcja	Co to jest
Projekt Python-	kod źródłowy i SafeSort pliki na dysku, z którym pracujemy
Git-repozytorium	historia zmian tych plików, przechowywana w ukrytym katalogu .git
GitHub-repozytorium	Git-repozytorium na serwerze GitHub razem z interfejsem webowym
GitHub Project	osobne narzędzie planistyczne: Issues, rada i widoki

Git działa całkowicie lokalnie na twoim komputerze i Przechowuje historię zmian plików. Nie potrzebuje konta GitHub, Ani Internetu, ani zdalnego repozytorium. Możesz zainstalować Git, tworzyć repozytorium, wykonaj sto commitów i nigdy nie łącz się z siecią. **GitHub** działa jako oddzielna usługa internetowa nad Git: przechowuje

zdalną kopię repozytorium i dodaje to, czego sam Git nie ma: Issues, Pull Request, Projects, Actions i Releases. GitHub jest zbudowany wokół Git, ale odwrotnej zależności nie ma: Git działa świetnie bez GitHub, podczas gdy GitHub bez Git, przeciwnie, nie może działać. Ta usługa przechowuje i obsługuje Git- repozytoria.

Git samodzielnie pracuje z historią lokalną. GitHub zapewnia usługę sieciową na Git. Zależność jest jednokierunkowa: Git nie potrzebuje GitHub, a GitHub korzysta z repozytoriów Git-.

Repozytorium na GitHub — to nie to samo, co GitHub Project

To jedno z najczęstszych nieporozumień wśród początkujących. Repozytorium przechowuje kod i jego historię. GitHub Project służy jako osobne, opcjonalne narzędzie do planowania: możesz mieć repozytorium bez Project i stworzyć Project, który łączy Issues z kilku repozytoriów jednocześnie. Część II tego rozdziału szczegółowo przeanalizuje tę różnicę.

Oficjalne zasoby Git

Projekt Git posiada własną stronę internetową, książkę referencyjną i książkę do nauki. Oni odpowiadają Na różne pytania, więc warto od razu wiedzieć, gdzie szukać instalatora, co jest dokładne składnia polecenia lub pełne wyjaśnienie idei.

Zasoby	Do czego jest potrzebny
Oficjalna strona Git	Git pobrania, szczegóły projektów oraz linki do materiałów szkoleniowych.

Zasoby	Do czego jest potrzebny
Dokumentacja pomocnicza Git	Dokładne opisy poszczególnych komend: git init, git clone, git add, git commit, git branch, git switch, git fetch, git pull i git push.
Pro Git książki	Kolejne wyjaśnienie modelu Git. To podręcznik, a nie tylko zestaw parametrów komend.
Git Source Code Mirror on GitHub	Lustrzane odbicie kodu źródłowego Git na GitHub. Sam Git nie zależy od GitHub:, umieszczenie lustrzanego odbicia pokazuje, że Git i GitHub pozostają różnymi systemami.

Git logo

Znak koloru na tej stronie pochodzi z oficjalnego zestawu Git bez ponownego rysowania oraz zmiany kolorów. Git Logo by Jason Long, licencja CC BY 3.0.

OFICJALNA DOKUMENTACJA

Oficjalna strona Git
Dokumentacja pomocnicza Git
Pro Git książki
Git Source Code Mirror on GitHub
Git logos

OFICJALNA DOKUMENTACJA

Start your journey: What is GitHub?

Tłumaczenie i edukacyjna adaptacja oficjalnej dokumentacji GitHub. materiałów źródłowych – CC BY 4.0.

Krótko

- Git działa lokalnie i śledzi historię plików; nie potrzebuje Internetu.
- GitHub udostępnia zdalną kopię repozytorium oraz narzędzi Issues, Pull Request, Projects, Actions i Releases.
- Repozytorium przechowuje kod i historię, a GitHub Project służy do planowania. Są to niezależne koncepcje.

Instaluj Git

Git jest umieszczane oddzielnie od Python, raz dla całego systemu – metoda zależy od systemu operacyjnego.

SafeSort · Część 1 z 6**Git i GitHub**

Git – nie jest wbudowaną częścią Python:, trzeba go zainstalować osobno, raz dla całego systemu. Sposób instalacji zależy od systemu operacyjnego.

```
~/safesort $ git --version
git version 2.47.3
```

Tak wygląda udany test – program podaje swoją wersję i nic więcej.

Linux (Debian/Ubuntu)

Udokumentowane Dowództwo — Zobacz git-scm.com/download/linux

```
sudo apt update
sudo apt install git
```

git, a nie git-all

Pakiet `git` już zawiera wszystko, czego potrzebujesz do tego kursu. Pakiet `git-all` to meta-pakiet, który dodaje dodatkowe integracje (na przykład Emacs), które tutaj nie są potrzebne).

Dla dystrybucji opartych na Fedora/RHEL:

Udokumentowany zespół

```
sudo dnf install git
```

Windows

Oficjalny instalator — **Git for Windows**: Sam program organizuje `git` i terminal **Git Bash**, w których pracują Wszystkie komendy w tym rozdziale pozostają niezmienione.

OFICJALNA DOKUMENTACJA

Git for Windows

macOS

Na macOS git-scm.com nie ma już jednego oficjalnego instalatora: osobnego Binary installer był wspierany tylko do wersji 2.33.0 (2021) i nie był wspierany zaktualizowane. git-scm.com wyraźnie stwierdza, że już się do niego nie odnosi i nie odnosi się planuje wznowić wsparcie. Zamiast tego macOS korzysta z jednego z trzech Opcje:

Metoda	Komenda
Xcode Command Line Tools; Odpowiednie, jeśli nie jest potrzebna pełna Xcode	xcode-select --install
Homebrew; Wspólny menedżer pakietów dla macOS	brew install git
MacPorts; Alternatywny Kierownik Pakietów	sudo port install git

Te buildy nie są wspierane przez git-scm.com

git-scm.com ostrzega: wszelkie dystrybucje Git poza kodem źródłowym są tworzone i utrzymywane przez podmioty trzecie (Apple dla Xcode Command Line Tools, zespół Homebrew, zespół MacPorts) i nie muszą być aktualizowane natychmiast po wydaniu Git. Każda z trzech opcji jest odpowiednia dla tego kursu. Polecenie git -version poniżej pokaże Ci, co jest faktycznie zainstalowane.

OFICJALNA DOKUMENTACJA

Git for macOS

Sprawdzanie instalacji

Niezależnie od systemu, wynik jest taki sam: `git --version` w terminalu drukuje numer wersji. Jeśli zamiast tego terminal odpowiada „command not found” (lub podobną wiadomością w Windows) — Git nie jest zainstalowany lub go nie ma w PATH; ponowna instalacja zwykle rozwiązuje problem.

OFICJALNA DOKUMENTACJA

Git — Downloads

Krótko

- Git jest ustalane tylko raz dla systemu, oddzielnie od Python.
- Linux: sudo apt install git (lub dnf install git). Windows: Git for Windows + Git Bash. macOS: Xcode Command Line Tools, Homebrew czy MacPorts; od 2021 roku nie ma jednego oficjalnego instalatora.
- Polecenie git-version sprawdza wynik instalacji.

Pierwsze ustawienie Git

Zanim wykonasz pierwszy commit, Git trzeba raz powiedzieć, kim jesteś — jest to zapisywane w każdym commicie.

SafeSort · Część 1 z 6**Git i GitHub**
Przed pierwszym commitem Git musieli podać nazwę i email. raz Informacje te są zapisywane w każdym commite i pozostają w historii na zawsze.

```
~/git_demo $ git init
Initialized empty Git repository in .../git_demo/.git/
~/git_demo $ git config user.name „Twoje imię”
~/git_demo $ git config user.email "you@example.com"
~/git_demo $ git config --local --list
core.repositoryformatversion=0
core.filemode=true
core.bare=false
core.logallrefupdates=true
user.name=Twoje imię
user.email=you@example.com
```

Ta nazwa jest zapisana w commitach, nie tylko w profilu GitHub
`user.name` / `user.email` nie oznaczają nazwy użytkownika GitHub i nie są tym samym co imię w profilu. To są dane, które dosłownie trafiają do każdego commitu jako autora zmiany i są widoczne dla wszystkich, którzy oglądają historię, także po publikacji na GitHub.

Flaga `--local` dotyczy tylko tego jednego repozytorium. Aby nie powtarzać konfiguracji przy każdym nowym projekcie, powszechnie stosuje się ją `--global` — wtedy imię i e-mail stosowane są do wszystkich repozytoriów na tym komputerze:

Terminal

```
git config --global user.name "Ваше Имя"
git config --global user.email "you@example.com"
```

Domyślna nazwa gałęzi

Łatwo tu popełnić błąd: Git nadal nazywa pierwszą gałąź nowego repozytorium `master` jeśli jego nazwa nie jest w żaden sposób skonfigurowana, jest to inline Zachowanie się nie zmieniło. W wersji 2.28 Git zdobyła tylko siebie *Przygotowanie* `init.defaultBranch`, pozwalająca ustawić inną nazwę domyślną, — a nie nowe zachowanie „out-of-the-box”. Nazwa `main` stało się nawykiem od GitHub, GitLab wielu redaktorów konfiguruje tę opcję samodzielnie podczas instalacja, ale Git sam jej nie ustawia. Dlatego Cartesian School ją ustawia wprost, a nie Zależy od tego, co zostanie skonfigurowane na konkretnym komputerze:

```
Terminal
git config --global init.defaultBranch main
```

Git 2.28 dodała ustawienie zamiast nowego zachowania
Łatwo pomylić te dwie rzeczy. Git 2.28 (wydana w 2020 roku) dodała opcję `init.defaultBranch` — Wcześniej takie ustawienie w ogóle nie istniało, a pierwszą gałąź można było przemianować ręcznie tylko po utworzeniu repozytorium. Ale sama opcja nic nie zmienia, dopóki jej nie ustawisz: bez niej Git nadal tworzy gałąź `master` w każdej wersji aż do najnowszych wersji. Możesz to sprawdzić w czystym środowisku bez `~/.gitconfig` — tam `git init` tworzy dokładnie `master`.

OFICJALNA DOKUMENTACJA

Set up Git

Krótko

- user.name/user.email są zapisywane w każdym commitcie — to są dane autora, a nie GitHub. logowania
- --global stosuje to ustawienie do wszystkich repozytoriów na maszynie jednocześnie.
- Git 2.28 dodała ustawienie init.defaultBranch, ale wbudowana nazwa pierwszej gałęzi nadal jest master — main musi być ustawiona wyraźnie.

Tworzymy i chronimy konto GitHub

Nazwa użytkownika, email i uwierzytelnianie dwuskładnikowe — kilka decyzji warto podjąć świadomie od samego początku.

SafeSort · Część 1 z 6**Git i GitHub**
Współpraca z GitHub wymaga konta. Łatwo go zdobyć, ale na tym etapie jest kilka rozwiązań Warto je brać świadomie.

Nazwa użytkownika

GitHub nazwa użytkownika staje się częścią adresu każdego z twoich repozytoriów (`github.com/имя/репозиторий`) i widoczna dla wszystkich – warto wybrać Coś, co nie będzie szkoda użyć w CV czy portfolio, i nie jest to chwilowe przezwisko.

Email

GitHub wymaga potwierdzonego email. Jeśli nie chcesz, aby Twój prywatny adres był uwzględniony w Commit history podczas publikowania repozytorium GitHub zapewnia prywatne `@users.noreply.github.com` adres – zwłaszcza do tego sprawa.

Hasło i uwierzytelnianie dwuskładnikowe

GitHub obsługuje logowanie hasłem z uwierzytelnianiem dwuskładnikowym (2FA) oraz passkey. Warto od razu włączyć 2FA: konto na GitHub to nie tylko zestaw plików, ale dostęp do publikacji kodu w twoim imieniu.

Sytuacja	Rozwiązanie
Osobisty email w zobowiązaniach nie jest pożądany	użyj noreply-adresu GitHub
Potrzebna jest metoda logowania zapasowego	zapisz kody odzyskiwania po włączeniu 2FA
Zapomniałem hasła	korzystać z oficjalnego odzyskiwania dostępu, a nie usług firm trzecich

OFICJALNA DOKUMENTACJA

Signing up for a new GitHub account

About two-factor authentication

Krótko

- Nazwa użytkownika staje się częścią adresu każdego repozytorium – wybierz mądrze.
- Prywatny adres noreply- GitHub ukrywa prywatny email w historii zatwierdzenia.
- Uwierzytelnianie dwuskładnikowe warto włączyć od razu, zamiast opóźniać.

HTTPS, SSH i uwierzytelnianie

Browser, Git HTTPS i Git przez SSH weryfikować tożsamość na trzy różne sposoby – kurs wykorzystuje SSH.

SafeSort · Część 1 z 6**Git i GitHub**
Otwarcie github.com w przeglądarce i praca z Git z terminala to dwa różne sposoby Potwierdź, że to naprawdę Ty i że mają różne mechanizmy.

Trzy różne sposoby, żeby potwierdzić, że to Ty — każdy ma swój mechanizm

Hasło bezpośrednio dla git push już nie działa

Wcześniej możliwe było użycie hasła do konta bezpośrednio podczas git push używania HTTPS — GitHub wyłączyłem tę metodę. Teraz HTTPS wymaga personal access token przez credential helper lub logowania przez GitHub CLI (`gh auth login`) zamiast bezpośrednio hasła.

HTTPS	SSH
URL rodzaje https://github.com/OWNER/REPO.git	URL rodzaje git@github.com:OWNER/REPO.git
Działa praktycznie wszędzie, także w sieciach o ścisłych firewall	Może wymagać otwartego portu 22
Uwierzytelnianie przez credential helper / GitHub CLI	Uwierzytelnianie za pomocą pary SSH- kluczy

Kurs używa **SSH** jako główna ścieżka: po skonfigurowanej parze Keys działa dla dowolnej liczby repozytoriów bez ponownego wprowadzania tokena — i kolejnego Page przeprowadzi Cię przez całe przygotowanie. To nie znaczy, że SSH obiektywnie lepszy We wszystkich przypadkach: w sieci z twardym firewall blokującym port 22, HTTPS może być jedyna działająca opcja.

OFICJALNA DOKUMENTACJA

About authentication to GitHub

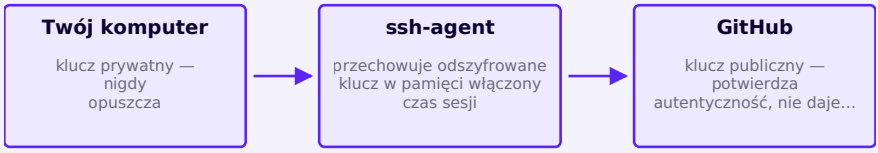
Krótko

- logowanie do przeglądarki, HTTPS Git i Git SSH to trzy różne mechanizmy uwierzytelniania.
- Hasło konta bezpośrednio dla git push już nie działa — tylko klucz token lub SSH-.
- Trasa wykorzystuje SSH jako główną ścieżkę; HTTPS pozostaje działającą alternatywą.

Konfigurujemy SSH

Klucz prywatny pozostaje na komputerze i potwierdza tożsamość bez opuszczania go; Klucz publiczny trafia do GitHub.

SafeSort · Część 1 z 6**Git i GitHub**



Klucz prywatny potwierdza tożsamość bez opuszczania komputera; tylko klucz publiczny wchodzi do GitHub

uwierzytelnianie SSH- opiera się na parze kluczy: klucz prywatny pozostaje na twoim komputerze i nigdy nie jest nikomu udostępniane, publicznie jest przesyłane do ustawień GitHub. GitHub Potwierdza, że masz prywatną połowę pary, nie widząc jej osobiście.

Stwórz klucz

Udokumentowane polecenie — patrz oficjalna instrukcja GitHub

```
ssh-keygen -t ed25519 -C "you@example.com"
```

Na pytanie o plik do zapisania zwykle wystarczy nacisnąć Enter (domyślna ścieżka), a frazę hasła (passphrase) warto ustawić — to dodatkowa ochrona klucza na dysku.

Dodaj klucz do ssh-agent

Udokumentowany zespół

```
eval "$(ssh-agent -s)"
ssh-add ~/.ssh/id_ed25519
```

Dodaj klucz publiczny do GitHub

Zawartość pliku `~/.ssh/id_ed25519.pub` jest kopiowany do Settings → SSH and GPG keys → New SSH key.

Sprawdzamy połączenie

Udokumentowany zespół

```
ssh -T git@github.com
# Hi USERNAME! You've successfully authenticated, but GitHub
does not provide shell access.
```

Ta wiadomość nie jest pomyłką

GitHub nie zapewnia interaktywnej powłoki SSH — fraza „does not provide shell access”

OFICJALNA DOKUMENTACJA

About SSH

Generating a new SSH key and adding it to the ssh-agent

Adding a new SSH key to your GitHub account

Krótko

- Klucz prywatny nigdy nie opuszcza komputera; tylko klucz publiczny jest pobierany na GitHub.
- ssh-keygen tworzy parę ssh-add dodaje ją do ssh-agent na czas trwania sesji.
- ssh -T git@github.com sprawdza połączenie — komunikat o braku shell-dostępu oznacza sukces.

Tworzymy repozytorium SafeSort na GitHub

Repozytorium prawdziwego Cartesian-School/safesort to to, które służy do reszty rozdziału.

SafeSort · Część 1 z 6**Git i GitHub**

GITHUB WŁAŚCIWE REPOZYTORIUM TEGO ROZDZIAŁU

Konto jest gotowe, uwierzytelnianie jest skonfigurowane – czas stworzyć prawdziwe repozytorium dla SafeSort. To konkretne repozytorium `Cartesian-School/safesort`, Użyte na wszystkich pozostałych stronach rozdziału: każda Issue, każda gałąź, każda Pull Request i ostateczne wydanie, które są pokazane tutaj, są prawdziwe.

Główna strona prawdziwego repozytorium Cartesian-School/safesort GitHub: pliki, README, historię zatwierdzeń, karty Issues/Pull requests/Actions/Projects

Rzeczywiste repozytorium SafeSort — to, które jest używane w całej pozostałej części rozdziału.

Pola formularza tworzenia repozytorium

Pole	Co to znaczy
Owner	konto lub organizacja, której należy repozytorium
Repository name	część adresu github.com/OWNER/NAZWA — krótka, bez spacji

Pole	Co to znaczy
Description	jeden łańcuch znaków, widoczny na liście repozytoriów oraz w wyszukiwarce
Public / Private	, czy repozytorium jest widoczne dla wszystkich, czy tylko dla tych, których zaprosisz
Add a README	, czy stworzyć startera README.md od razu po utworzeniu
.gitignore template	gotowy szablon wyjątku specyficzny dla języka
License	licencji, na podstawie której kod jest dystrybuowany, na przykład MIT

Puste repozytorium czy bezpośredni README?

Są dwa sposoby: od razu utworzyć repozytorium z licencją README/.gitignore/ dla GitHub lub Stwórz go pustym i wciskaj do niego gotowy lokalny projekt. Ten rozdział jest drugi Way: puste repozytorium na GitHub oraz pierwszy commit (README, LICENSE, .gitignore, pyproject.toml) pochodzi z lokalnej maszyny, więc od samego początku jest jasne, co dokładnie był pierwszym commitem i nie pojawił się „samodzielnie”

OFICJALNA DOKUMENTACJA

Creating a new repository

Krótko

- Cartesian-School/safesort jest faktycznym repozytorium użytym przez resztę rozdziału.
- Repository name staje się częścią adresu; Public/Private decyduje o widoczności.
- Kurs tworzy puste repozytorium na GitHub i wysyła do niego gotowy lokalny first commit.

Sklonuj repozytorium

git clone tworzy pełną lokalną kopię repozytorium jednocześnie—pliki, historię i GitHub.

SafeSort · Część 1 z 6**Git i GitHub**
Repozytorium istnieje na GitHub — ale żeby pisać kod, musisz go samodzielnie zdobyć komputer. `git clone` pobiera repozytorium w całości, razem z całą jego historią commitów, i od razu ustawia połączenie z GitHub jako `origin` (następna strona będzie miała więcej szczegółów).

git clone jest jedynym zespołem, który tworzy pełną lokalną kopię naraz

```
~/safesort $ git clone https://github.com/Cartesian-
School/safesort.git
Cloning into 'safesort'...
~/safesort $ cd safesort
~/safesort $ git status
```

```
On branch main
Your branch is up to date with 'origin/main'.

nothing to commit, working tree clean
~/safesort $ git log --oneline -5
fe610cf docs: fill in CHANGELOG for 0.1.0 (#23)
c376fba feat: add command-line interface (#21)
01989e0 feat: add duplicate detection with byte-level
confirmation (#20)
29c6328 feat: record operation manifest and add undo
(#19)
c5e34d5 feat: add explicit apply operation (#18)
```

Jak obecnie wygląda repozyto Cartesian-School/safesort rium literatury, z pełną historią

Repozytorium referencyjne przygotowane wcześniej dla kursu
git log — oneline zaraz po klonowaniu pokazuje nie pustą historię, lecz wszystkie commity, które już znajdują się w repozytorium na GitHub — klonowanie pobiera pełną historię, a nie tylko najnowszy stan plików. I to nie jest puste repozytorium: Cartesian-School/safesort przygotowane do kursu z wyprzedzeniem, z wszystkimi Issue, gałęziami, Pull Request i v0.1.0 wydania, już gotowymi. Kroki samouczka w tym rozdziale pokazują, jak taki projekt powstaje od zera, ale prawdziwa historia repozytorium referencyjnego to nie jest zapis tych kroków klatka po klatce, lecz jego własna, oddzielna i całkowicie uczciwa sekwencja prawdziwych commitów. Niżej w rozdziale wyraźne notatki rozróżniają "PRAWDZIWE DOWODY REPOZYTORIUM" (co naprawdę jest w tej historii) a "STUDIUM PRZYPADKU" (ilustracja techniki na osobnym materiale szkoleniowym).

OFICJALNA DOKUMENTACJA

Cloning a repository

Krótko

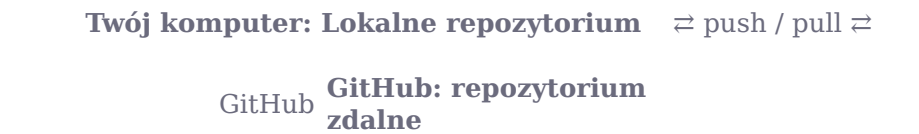
- git clone pobiera cały repozytorium — pliki i całą historię commitów, nie tylko najnowszy stan.
- Po klonowaniu git status od razu pokazuje powiązanie z origin — GitHub jest już ustawiony jako zdalne repozytorium.
- git log --oneline po sklonowaniu od razu pokazuje prawdziwą historię projektu.

Lokalne i zdalne repozytorium

origin — po prostu nazwa zdalnego repozytorium, którą git clone przypisuje domyślnie.

SafeSort · Część 1 z 6**Git i GitHub**
`origin` nie jest serwerem specjalnym ani zarezerwowanym słowo Git, ale po prostu nazwa, która `git clone` dał go jednemu Domyślnie konkretne zdalne repozytorium. Nic nie stoi na przeszkodzie, by nazwać usunięte repozytoria są inne — ale `origin` do „tam, gdzie wszystko się zaczęło”

```
~/safesort $ git remote -v
origin https://github.com/Cartesian-School/
safesort.git (fetch)
origin https://github.com/Cartesian-School/
safesort.git (push)
```



origin to nazwa, a nie adres; sam adres jest przechowywany osobno i widoczny przez git remote -v

Pojedyncze lokalne repozytorium może mieć wiele zdalnych — na przykład "origin" dla główne repozytorium oraz "upstream" dla oryginału. **Fork** jest twoja Twoją własną kopię cudzego repozytorium na GitHub: możesz ją swobodnie zmieniać bez wpływu na to oryginalny. Taki pakiet origin/upstream się przyda, jeśli zdecydujesz się Przejdź przez własne fork `python-mini-projects` jest jedna z dwóch równoważnych opcji praktyki, porównywalna z utworzeniem osobnego repozytorium od zera (patrz Aneks).

OFICJALNA DOKUMENTACJA

About remote repositories

Managing remote repositories

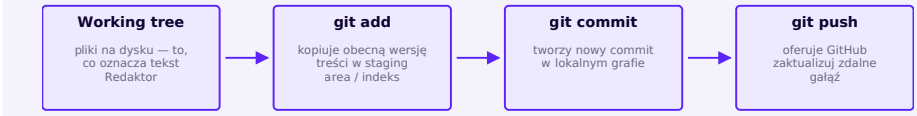
Krótko

- origin — nazwa zdalnego repozytorium, którą git clone ustawia domyślnie, a nie specjalne słowo Git.
- git remote v pokazuje prawdziwe URL stojące za nazwą origin.
- Jeden lokalny repozytorium może mieć kilka zdalnych — na przykład origin i upstream podczas pracy z fork.

Branch, HEAD origin/main: Git rozumie, gdzie jesteśmy

Najpierw oddzielmy bieżącą lokalną gałąź od lokalnej wiedzy o GitHub. Wtedy switch, fetch, pull i push przestaną być zbiorem magicznych poleceń.

SafeSort · Część 1 z 6**Git i GitHub**
Przed pierwszą działającą gałęzią zbudujemy jeden spójny model: gdzie leży zmiana, do którego commita wskazuje bieżąca gałąź i co dokładnie Git aktualizuje przy `fetch`, `pull` oraz `push`.



plik pozostaje w working tree: git add zamiast go przesuwać, zatwierdza wybraną treść do indeksu

`git add` nie przenosi pliku w inne miejsce. Plik roboczy pozostaje na dysku i może się ponownie zmieniać. Polecenie indeksuje migawkę swojej *aktualna zawartość*to będzie częścią następnego commitu.

Commity tworzą graf

Każdy commit zna commit nadarcy. Dlatego story nie jest jak folder z wersje oraz na wykresie. **Gałęzią** nazwę ruchomą, która wskazuje na pojedynczy commit. **HEAD** zwykle wskazuje na wybraną gałąź. Nowy commit otrzymuje aktualne Commituj nadrzędne i przesuwaj tę gałąź dalej.

```
~/safesort $ git log --oneline --graph --decorate --
all
* c31a4d2 (HEAD -> feature/scanner) Add directory
scanner
```

```
* 82b19aa Add scanner test
* 4f10c77 (main, origin/main) Initial project
structure
```

HEAD wskazuje wybrany feature/scanner; main i origin/main nadal znajdują się na poprzednim commitcie

```
      C3  feature/scanner  ← HEAD
     /
C1──C2  main, origin/main
```

Nazwa	Gdzie mieszka	Co oznacza
main	repozytorium lokalne	twoja lokalna gałąź
origin/main	repozytorium lokalne	ostatni znany Git odległym stanie main
main on GitHub	repozytorium zdalne	prawdziwa gałąź na serwerze GitHub

`origin/main` nie pokazuje stanu GitHub w czasie rzeczywistym. To lokalne remote-tracking odniesienie, które zmienia się, gdy Git otrzymuje informacje z serwera, na przykład po `git fetch`.

fetch aktualizuje wiedzę o remote; pull dodaje integrację; push przechodzi w przeciwnym kierunku

```
~/safesort $ git status
```

```
On branch main
Your branch is up to date with 'origin/main'.
~/safesort $ git branch -vv
* main 4f10c77 [origin/main] Initial project structure
~/safesort $ git fetch origin
~/safesort $ git log --oneline --graph --decorate --
all
```

Komenda	Co pokazuje
git status	które pliki zostały zmienione, które już są w staging, które Git w ogóle nie śledzi
git diff	wierszowymi zmianami w drzewie roboczym, jeszcze nie dodanymi do staging
git diff --staged	zmiany linijka po linijce już dodane do staging będą zawarte w commit-cie
git branch -vv	lokalnych oddziałów, obecny oddział i ich upstream- powiązania
git log --oneline --graph --decorate --all	Commit graph i link locations

Krótko

- Gałąź jest ruchomym odniesieniem do zatwierdzenia; HEAD zwykle wskazuje na obecnie wybraną gałąź.
- main, origin/main i main na GitHub oznaczają trzy rozróżnialne stany, a nie trzy pisowni tego samego obiektu.
- fetch aktualizuje remote-tracking linki; pull dodaje integrację; push aktualizuje zdalną gałąź.

OFICJALNA DOKUMENTACJA

Git Branching — Branches in a Nutshell
git-fetch
git-pull
git-push

Repository i GitHub Project – jaka jest różnica

Repository przechowuje kod; GitHub Project śledzi status zadań — dwa różne, niezależne obiekty.

SAFESORT · CZĘŚĆ 2 Z 6

Git i GitHub → **Planowanie** → Projekt →
Wdrożenie → Testy i CI → Premiera

Część I już oddzieliła Git i GitHub. Oto drugie częste zamieszanie: repozytorium oraz **GitHub Project** — też różne, niezależne pojęcia, i łatwo uznać, że skoro oba słowa zaczynają się od „projekt”, to to samo.

Repository	GitHub Project
kod, pliki, historia commitów	zadania, ich status, widoki (tablica/ tabela)
gałęzie, tagi, Pull Request	może łączyć Issues z wielu repozyto-riów jednocześnie
odpowiada na pytanie „co już zostało zrobione”	odpowiada na pytanie «co trzeba zrobić i w jakiej kolejności»
jest wymagane – bez niego nie ma kodu	jest opcjonalny — można pracować całkowicie bez Project

Project nie przechowuje kodu — śledzi status zadań odwołujących się do repozytorium

Jeden Project może obejmować kilka repozytoriów

GitHub Project nie jest częścią repozytorium i nie znajduje się w nim: jest osobnym obiektem, który może jednocześnie wyświetlać Issues i Pull Request z kilku repozytoriów w tej samej organizacji. Dla SafeSort, ten rozdział używa jednego Project na repozytorium — ale nie zawsze tak jest.

OFICJALNA DOKUMENTACJA

About Projects

Krótko

- Repository przechowuje kod i historię; GitHub Project śledzi status zadań — są to różne obiekty.
- Project nie jest obowiązkowy: można prowadzić repozytorium w ogóle bez niego.
- Pojedynczy Project może łączyć Issues z wielu repozytoriów w Twojej organizacji.

Jak zaplanować Project: najlepsze praktyki

Project łatwe do stworzenia w sekundę i równie łatwe do zagracenia tuzinem niewykorzystanych pól — kilka decyzji należy podjąć wcześniej.

SafeSort · Część 2 z 6**Planowanie**
Stworzenie Project to kwestia sekund; błąd, który popełnia niemal każdy początkujący — natychmiast dodaj tuzin pól i trzy widoki „na przyszłość”

Ustalić zakres z wyprzedzeniem

Project można powiązać z jednym repozytorium lub połączyć w nim kilka. Dla SafeSort odpowiedź jest prosta — jeden Project na jedno repozytorium, ponieważ cała praca tego rozdziału odbywa się w `Cartesian-School/safesort` nie Nie ma sensu pobierać zadań z innych repozytoriów kursów do jednego lista.

Mniej dziedzin jest lepszych

Zamiast	Lepiej
pole „na wszelki wypadek”	pole, które odpowiada na konkretne pytanie (kto jest ważniejszy, która część kodu)
wolny tekst tam, gdzie opcji naprawdę jest niewiele	Single select z wcześniej znaną listą wartości
nowy widok dla każdego pomysłu „a co jeśli się przyda”	jeden lub dwa przedstawienia, które naprawdę otwierają się codziennie

Status – Jedno pole prawie zawsze wystarcza do śledzenia postępów
U GitHub Project jest już wbudowane pole Status. Zanim dodasz coś jeszcze, warto zapytać: czy nowe pole odpowiada na pytanie, którego Status nie obejmuje? Dla SafeSort jest to Priority (co robić w pierwszej kolejności) oraz Area (do której części kodu odnosi się zadanie) — oba pola pojawią się później w tej części rozdziału.

Pola i reprezentacje można zmieniać później

Nic z rozwiązanych w tym kroku nie jest wyryte w kamieniu: GitHub pozwala dodać, zmienić nazwę lub usunąć pole i widok w dowolnym momencie, nie tracąc już zebranych danych. Błąd nowicjusza — nie „wybrać niewłaściwe pole”, lecz rozwiązać wszystko naraz i nigdy nie przeglądać ponownie.

OFICJALNA DOKUMENTACJA

Best practices for Projects

Krótko

- Zanim utworzysz Project, powinieneś zdecydować, czy jest to jedno repozytorium, czy wiele repozytoriów.
- Mniej pól i widoków, z których każde jest faktycznie używane, lepiej tuzin „na przyszłość”
- Status jest polem wbudowanym; powinieneś dodawać własne pola tylko wtedy, gdy Status nie odpowiada na żądane pytanie.

Tworzenie GitHub Project

Przed napisaniem kodu GitHub Project daje miejsce na listę zadań związanych z rzeczywistym Issues repozytorium.

SafeSort · Część 2 z 6**Planowanie**
Zanim napiszesz kod SafeSort, warto stworzyć lista tego, co trzeba zrobić, — GitHub Project daje do tego gotowe miejsce, powiązane z rzeczywistymi Issues repozytoriami.

Co jest wypełniane podczas tworzenia

Pole	Znaczenie
Owner	organizacja lub konto, tutaj Cartesian-School
Title	na przykład „SafeSort jest pierwszym wydaniem”
Template	pusty Project lub jeden z gotowych szablonów (Board, Roadmap…)
Visibility	czy Project jest widoczny dla wszystkich, czy tylko dla członków organizacji

Właśnie w ten sposób powstał prawdziwy Project tego rozdziału — „SafeSort — pierwsze wydanie" (Cartesian-School, Project nr 1): Owner — Cartesian-School, Title — bez numeru wersji, Template — pusty Project. Jak wiele roboczych tablic planowania, ten Project jest widoczny tylko dla uczestników organizacji Cartesian-School (Visibility z tabeli powyżej) — bezpośredni link do niego w tekście jest celowo pominięty, aby nie obiecywać dostępu, którego czytelnik nie ma; cała jego zawartość pokazana jest poniżej na rzeczywistych zrzutach ekranu.

Table- przedstawienie prawdziwego Project „SafeSort — pierwsze wydanie": wszystkie 14 Issues, pola Status, Priority, Area

Ten Project SafeSort jest pierwszym wydaniem stworzonym w ramach organizacji Cartesian-School.

Tytuł bez numeru wersji

Tytuł Project tego rozdziału brzmi **„SafeSort — pierwsze wydanie”**, bez Pokoje `0.1.0` : część VI wprowadzi wersje dopiero wtedy, gdy pierwsza wersja rzeczywiście będzie gotowa, a nazywanie Project według numeru przed czasem — wyprzedzanie bez potrzeby.

OFICJALNA DOKUMENTACJA

Creating a project

Krótko

- GitHub Project jest tworzony dla konkretnego właściciela (organizacji lub konta) z nazwą i poziomem widoczności.
- Nazwa Project dla SafeSort nie zawiera numeru wersji — wersja pojawia się później.
- Zaraz po utworzeniu Project jest pusty — kolejne sekcje pokazują jego reprezentacje i pola, a następnie, jak trafiają do niego zadania.

Board, Table i Roadmap

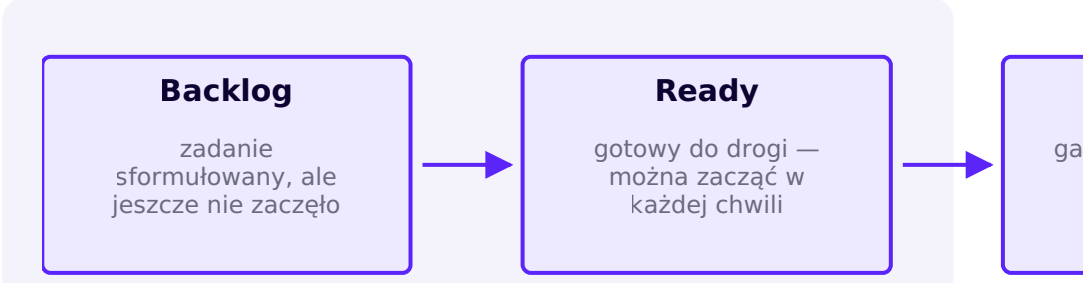
Board, Table i Roadmap to różne sposoby patrzenia na ten sam zestaw zadań, a nie oddzielne zbiory danych.

SafeSort · Część 2 z 6**Planowanie**
Ten sam zestaw problemów można postrzegać na różne sposoby – GitHub Project to nazywa **przedstawienia** (views): jeden zestaw elementów, kilka sposobów na Żeby go zobaczyć.

Wprowadzenie	Kiedy przydatne
Board	zobaczyć postęp według kolumn statusu: Backlog / Ready / In Progress / In Review / Done
Table	zobaczyć wszystkie zadania i ich pola od razu — sortować, filtrować, grupować
Roadmap	zobaczyć zadania na osi czasu — przydatne dla dat i terminów

Nie każdy projekt potrzebuje wszystkich widoków
Dla SafeSort w tym rozdziale jest wystarczająco dużo Board (możesz zobaczyć, co się dzieje teraz) i Table (możesz zobaczyć wszystkie zadania w całości). Roadmap ma sens, gdy zadania mają daty rozpoczęcia i zakończenia — tutaj tak nie jest i nie musisz tego dodawać tylko dlatego, że GitHub to oferuje.

Statusy dla SafeSort



Pięć statusów, przez które przechodzi każde zadanie SafeSort

Prawdziwa tablica Project SafeSort — wszystkie 14 zadań już w Done, ponieważ wydanie 0.1.0 już wyszło.

OFICJALNA DOKUMENTACJA

Changing the layout of a view

Krótko

- Board, Table i Roadmap to widoki tego samego zestawu zadań, a nie oddzielne zbiory danych.
- SafeSort wykorzystuje pięć statusów: Backlog, Ready, In Progress, In Review, Done.
- Reprezentację warto dodawać tylko wtedy, gdy naprawdę będzie z niej korzystać.

Pola Project: wbudowane i niestandardowe

Część pól Project jest wbudowana w GitHub, część można dodać samemu — pod konkretny problem, który mają rozwiązywać.

SafeSort · Część 2 z 6**Planowanie**
Każdy element Project jest zbiorem pól: niektóre z nich są wbudowane w GitHub i są Dla każdego Project możesz dodać niektóre samodzielnie do konkretnego zadania.

Wbudowane pola

Pole	Skąd się bierze?
Title	nagłówek Issue lub Pull Request

Pole	Skąd się bierze?
Assignees	kto dostaje zadanie w tym samym Issue
Status	Board kolumna – zarządzana przez Project, nie przez repozytorium
Labels	etykiety Issue/PR z repozytorium
Repository	które repozytorium Project łączy wiele
Linked pull requests	PR związane z Issue przez Closes #N

Project sam nie wymyśla tych pól — albo odczytuje je z repozytorium (Title, Assignees, Labels) lub kontroluje je tylko w sobie (Status).

Rodzaje pól użytkownika

Typ pola	Kiedy używać
Text	Krótka, arbitralna notatka, na przykład link do dyskusji
Number	wartość liczbową — na przykład ocenę trudności w godzinach
Date	konkretna data — na przykład deadline, jeśli istnieje
Single select	stały lista wartości wybierany jest z listy rozwijanej
Iteration	powtarzające się okresy – sprinty, tygodnie

Dwa niestandardowe pola SafeSort

Pole	Typ	Wartości
Priority	Single select	High / Medium / Low

Pole	Typ	Wartości
Area	Single select	Packaging / CLI / File-system / Safety / Dupli-cates / Testing / Docu-mentation / CI

Oba pola — Single select, a nie Text: lista wartości są wcześniej znane i skończone, a Single select dodatkowo pozwala grupować i filtrować Table według wartości, czego wolny tekst nie daje. `Iteration` tutaj nie jest potrzebny – SafeSort Nie jest wykonywany w formie sprintów, zadania nie mają powtarzalnych okresów czasowych.

Table prawdziwego Project SafeSort z wypełnionymi kolumnami Priority i Area dla wszystkich 14 zadań

Priority i Area są wykonywane dla wszystkich 14 zadań tego Project SafeSort.

OFICJALNA DOKUMENTACJA

Understanding fields

Krótko

- Wbudowane pola (Title, Assignees, Status, Labels) są w każdym Project bez konfiguracji.
- Pola niestandardowe mogą być tekstowe, numeryczne, datowe, Single select lub Iteration.
- SafeSort używa dwóch Single select pól — Priority i Area — ponieważ lista ich wartości jest znane i skończone z góry.

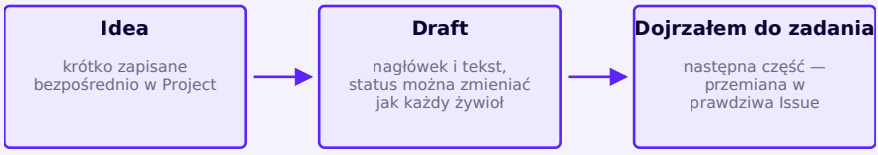
Szkice: Zadanie bez Issue

Szkic (draft issue) uchwyca myśl w Project zanim stanie się formalnym repozytorium Issue.

SafeSort · Część 2 z 6**Planowanie**

Nie każda myśl o przyszłym zadaniu od razu dojrzewa do pełnoprawnego Issue z Problem, Expected outcome i lista kontrolna Acceptance criteria. Do takich notatek średniozaawansowanych Project **draft** (draft issue) to element z tytułem i tekstem, który istnieje tylko w Project i nie jest jeszcze powiązany z żadnym repozytorium.

Issue	Draft (draft issue)
żyje w konkretnym repozytorium	istnieje tylko wewnątrz Project
ma numer (#14 itd.)	brak numeru — to jeszcze nie jest zapis w repozytorium
jest widoczny na liście Issues repozytorium	jest widoczny tylko dla tych, którzy mają dostęp do Project
można powiązać z Pull Request za pomocą „Closes #N”	powiązać Pull Request nie można, dopóki nie stanie się Issue



Szkic — sposób na utrwalenie myśli w Project zanim stanie się ona formalnym Issue

SafeSort nie miał roboczych wersji — ale warto znać metodę

Wszystkie 14 zadań SafeSort zostało sformułowanych na tyle jasno od samego początku, by od razu stać się pełnoprawnym Issues (poniższe sekcje) — szkice nie były używane w ich historii. Technika ta jest jednak przydatna w projektach, gdzie praca nie jest wykonywana na wstępnie zaplanowanej liście, lecz w trakcie procesu: jeśli podczas tworzenia zauważysz problem, natychmiast zapisujesz szkic w Project, nie rozpraszając się formułowaniem pełnoprawnego Issue.

OFICJALNA DOKUMENTACJA

Adding items to your project

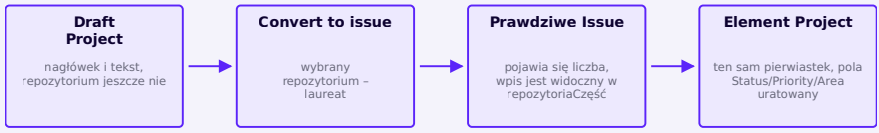
Krótko

- Draft (draft issue) to element Project bez repozytorium i bez numeru, dla myśli, które jeszcze nie są gotowe do Issue.
- Szkic ma tytuł, tekst i status, ale nie ma linku do Pull Request.
- SafeSort nie miał wersji roboczej, ponieważ wszystkie 14 zadań zostało sformułowanych natychmiast jako Issues.

Zamień draft w Issue

Convert to issue zamienia szkic w prawdziwy Issue repozytorium jednym działaniem — bez ręcznego kopiowania tekstu.

SafeSort · Część 2 z 6**Planowanie**
Kiedy szkic dojrzewa do zrozumiałego zadania, można go przekształcić w prawdziwy Issue — jednym działaniem bezpośrednio z Project, bez ręcznego kopiowania tekstu.



Konwersja zmienia typ elementu, ale nie tworzy nowego — to ten sam element Project

Pola nie są resetowane podczas konwersji
Jeśli szkic już miał wypełniony status lub pole Priority po konwersji na Issue, wartości te pozostają bez zmian — konwersja zmienia jedynie typ elementu (szkic → Issue) i dodaje link do repozytorium, zamiast odtwarzać element od nowa.

Repozytorium wybierane jest w momencie konwertowania

Szkic w ogóle nie ma repozytorium — GitHub pyta, które repozytorium Dodawaj przyszłe Issue tylko wtedy, gdy konwersja jest w trakcie. Dla Project z wieloma Repozytoria są świadomym wyborem; dla SafeSort, gdzie Project jest powiązany z jednym repozytorium, odpowiedź zawsze jest taka sama — `Cartesian-School/safesort`.

OFICJALNA DOKUMENTACJA

Converting draft issues to issues

Krótko

- Convert to issue zamienia szkic w prawdziwy Issue w jednym kroku, bez ręcznego kopiowania tekstu.
- Już wypełnione pola przedmiotów (Status, Priority i inne) są zapisywane po konwersji.
- Repozytorium-odbiorca wybierane jest w momencie konwersji — wersja robocza nie miała go wcześniej.

Stwórz Issues i dodaj do Project

Każda część SafeSort jest sformułowana Issue przed napisaniem pojedynczego łańcuch znaków kodu.

SafeSort · Część 2 z 6**Planowanie**
Przed napisaniem kodu każda część SafeSort jest formułowana jako **Issue** — zapis opisujący zadanie, zanim pojawi się jakikolwiek łańcuch znaków kod.

Część Issue	Przykład SafeSort
Title	«Add directory scanner»
Problem	SafeSort nie mogę znaleźć plików w katalogu

Część Issue	Przykład SafeSort
Expected outcome	scan() przegląda katalog i zwraca lista FileInfo
Acceptance criteria	lista kontrolna: przyjmuje Path, wyklucza .git/.venv, nie podąża za odnośnikami symbolicznymi, pokryty testami

Wszystkie 14 SafeSort numerów rzeczywiście są tak sformatowane w prawdziwym repozytorium — nie są retroaktywnie edytowane i tworzone jako zadanie przed rozpoczęciem pracy nad NEY:

Repository Issues listę Cartesian-School/safesort: 14 zamkniętych numerów, każdy od scanner do wydania

Prawdziwe 14 Issues repozytorium SafeSort.

Jedno Issue nie zawsze jest jednym Pull Request
W repozytorium SafeSort 9 z 14 Issues zamknięte Pull Request — ale nie „każdy swoim”: kolizje nazw (Issue nr 6) okazały się na tyle ściśle powiązane z planem przesunięć (Issue nr 3), aby trafić do jednego PR nr 17; to samo z manifestem i undo (Issues nr 7 i nr 8 — jedno PR nr 19). Pozostałe pięć Issues (obsługa błędów systemu plików, ustawienia i logowanie, zestaw testowy, CI, przygotowanie wydania) nie mają własnego wyodrębnionego PR: ta praca weszła do kodu w miarę realizacji odpowiednich modułów i została zamknięta wpisem wyjaśniającym, gdzie dokładnie się znalazła — prawdziwa historia projektu, a nie wymyślona dla ładnej tabeli „14 Issues → 14 PR”.

Issue dodaje się do Project

Issue istnieje w repozytorium niezależnie Project; Dodanie Project to osobna, opcjonalna akcja

Wszystkie 14 Issues repozytorium SafeSort rzeczywiście zostało dodanych do tego Project „SafeSort jest pierwszym wydaniem”

OFICJALNA DOKUMENTACJA

About issues

Creating an issue

Adding items to your project

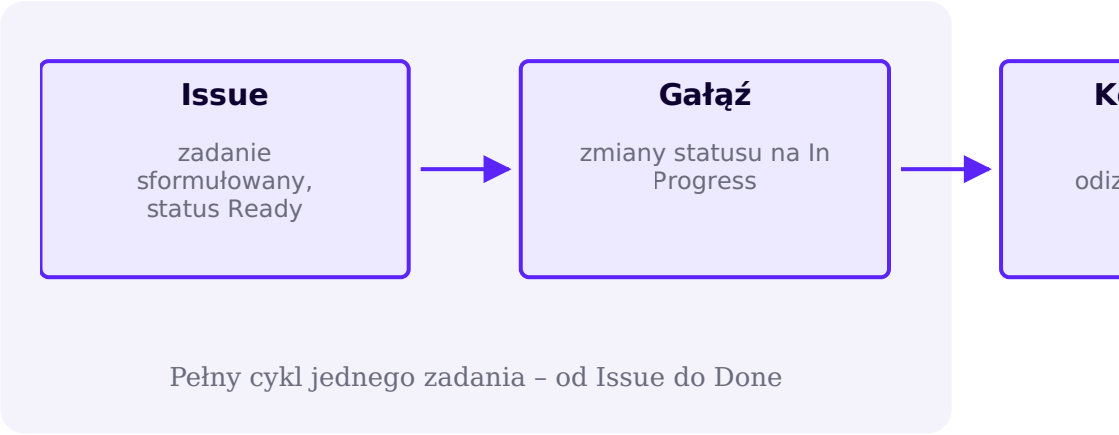
Krótko

- Issue formułuje zadanie — co należy zrobić i jak sprawdzić wynik — zanim kod zostanie napisany.
- Każdy Issue SafeSort ma Title, Problem, Expected outcome oraz listę kontrolną Acceptance criteria.
- Issue istnieje w repozytorium niezależnie Project; Dodawanie do Project to osobny krok.

Pierwsza pętla: Issue → Branch → Pull Request

Issue → rozgałęzia → kod i testuje → Pull Request → CI → scalanie — pętla, która powtarza się dla większości zadań SafeSort, ale nie według sztywnej zasady "jeden Issue — jeden PR.

SafeSort · Część 2 z 6**Planowanie**
Issue sformułowany, teraz przeanalizujmy pełny cykl jego życia, od zadania zadania do zamknij. Ten cykl powtarza się dla większości zadań SafeSort, zaczynając od następnego części rozdziału — ale nie według sztywnej zasady „jeden Issue — jeden Pull Request”



Pełny cykl jednego zadania – od Issue do Done

Terminal

```
git switch -c feat/directory-scanner
# ...пишем код и тесты, коммитим изменения...
git push -u origin feat/directory-scanner
```

Po push otwórz repozytorium w przeglądarce i kliknij **Compare & pull request**, wybierz `base: main`, dodaj tytuł i struna `Closes #1` do opisu. `git` pracuje z historią i Pull Request jest tworzona w interfejsie GitHub.

Closes #N zamyka Issue automatycznie — a jeden PR może zamknąć od razu kilka

Jeśli ciało Pull Request zawiera frazę w rodzaju `Closes #1` GitHub automatycznie zamyka Issue numer 1 w momencie łączenia tego PR — nie ma potrzeby zamykania Issue ręcznie. Nic nie stoi na przeszkodzie w zapisywaniu `Closes #3, Closes #6` jeśli PR rozwiązuje dwa blisko powiązane problemy jednocześnie, to właśnie tak stało się w repozytorium SafeSort z planem ruchu i obsługą konfliktów nazw.

Nie każdy Issue zamyka się po PR

Formułowanie zadania przez Issue nie zobowiązuje cię do zamknięcia go Pull Request. Jeśli praca została wykonana po drodze, jako część innego zadania lub nie wymagała żadnego kodu, można Issue zamknąć ręcznie — z komentarzem wyjaśniającym, co się wydarzyło. Prawdziwa historia projektu jest ważniejsza niż sztucznie płaska tabela „N zadań — N PR”

Następna część rozdziału przechodzi przez ten cykl naprawdę, krok po kroku, po raz pierwszy Prawdziwe zadanie — skaner katalogowy.

OFICJALNA DOKUMENTACJA

About pull requests

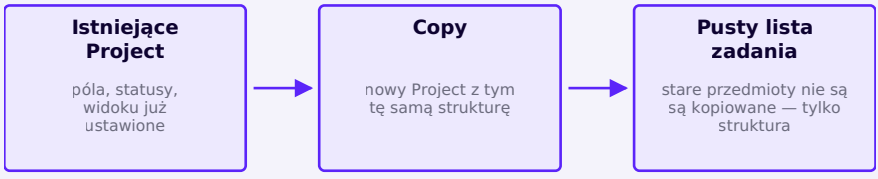
Krótko

- Issue → branch → kod i testuje → Pull Request → CI, a → merge check to pełny cykl jednego zadania.
- „Closes #N”
- Ta seria opisuje większość zadań, które SafeSort, ale nie jest to sztywna zasada — czasem PR zamyka się kilka blisko powiązanych Issues, a niektóre zadania są zamykane bez osobnego PR.

Opcjonalnie: Skopiuj istniejący Project

Copy Project przenosi strukturę — pola, statusy, widoki — ale nie przenosi samych zadań.

SafeSort · Część 2 z 6**Planowanie**
Tworzenie Project od podstaw to nie jedyny sposób, by zacząć. Jeśli Twoja organizacja już to zrobiła Project z odpowiednim zestawem statusów, pól i widoków, GitHub pozwala **kopia** to: nowy Project otrzymuje tę samą strukturę, ale pusty lista Zadania są po Issues, Pull Request i historia statusu nie są przenoszone.



Kopiowanie przekazuje strukturę Project, a nie jej zawartość

SafeSort tworzony od zera — na razie nie ma czego kopiować
W momencie przygotowywania tego rozdziału nie Cartesian-School jeszcze gotowego Project z odpowiednią strukturą, więc Project „SafeSort jest pierwszym wydaniem”

Gdy kopiowanie się opłaca

Kopiowanie jest użyteczne, gdy zespół już opracował wygodny zestaw statusów i pól i nie chce zbierać go za każdym razem od nowa, — typowy przypadek: kilka podobnych wydań z rzędu lub kilka podobnych projektów edukacyjnych jeden po drugim. Dla pierwszego Project w organizacji nie ma po prostu czego kopiować, a tworzenie od zera — to nie kompromis, a jedyna dostępna droga.

OFICJALNA DOKUMENTACJA

Copying an existing project

Krótko

- Kopiowanie Project przenosi jego strukturę — pola, statusy, widoki — ale nie same zadania.
- SafeSort stworzony od podstaw, ponieważ nie było jeszcze odpowiedniego Project do kopiowania.
- Kopiowanie się opłaca, gdy ta sama struktura jest potrzebna dla kilku podobnych projektów z rzędu.

Nieobowiązkowe: edytujemy elementy Project

Status, priorytet i inne pola przedmiotu mogą być zmieniane pojedynczo, przez panel lub jednocześnie hurtowo w Table.

SafeSort · Część 2 z 6**Planowanie**

Zadanie rzadko pozostaje takie samo od Backlog do Done – status, priorytet, a czasem Tytuł zmienia się w trakcie pracy. Project mają trzy sposoby, by to zrobić, w różnym stopniu masywności.

Metoda	Kiedy jest to wygodne
Przeciągnij kartę, żeby Board	zmiana statusu tylko jednego przedmiotu to najszybszy sposób
Otwórz panel przedmiotów	zmienić od razu kilka pól jednego elementu — Priority, Area, Assignees
Masowe zmiany w Table	zaznacz wiele wierszy i zastosuj tę samą wartość pola do wszystkich jednocześnie

Zmiana statusu — najczęstsza edycja elementu Project

Wszystkie 14 prawdziwych problemów SafeSort już w statusie Done — wydanie 0.1.0 jest dostępne, zmień je Status powrotny dla pięknego zrzutu ekranu oznaczałby kłamstwo na temat historii projektu. Dlatego, Poniżej znajduje się prawdziwe demo osobnego elementu demo, specjalnie dodane i prowadzone przez Ready → In Progress → In Review → Done w rzeczywistej Project SafeSort, a następnie usunięty, aby nie zniekształcać rzeczywistego stanu planszy:

Element demonstracyjny w kolumnie Ready tym Project SafeSort

1. Ready — karta demo jest dodana i gotowa do użycia.

Element demo przeciągnięty do kolumny In Progress

2. In Progress — ta sama karta jest przeciągana do następnej kolumny.

Element demo przeciągnięty do kolumny In Review

3. In Review — karta jest przeciągana ponownie, tym razem do przedostatniej kolumny.

element demonstracyjny przeciągnięty do kolumny Done, Done licznik stał się 15

4. Done — karta dotarła do ostatniej kolumny; licznik Done tymczasowo pokazuje 15 (14 rzeczywistych zadań + element demo).

Edycja masowa oszczędza czas przy tych samych zadaniach
Jeśli kilka Issues przechodzi w tę samą fazę jednocześnie – na przykład wszystkie zadania uczące się **Praktyka 23-20–Praktyka 23-23** są gotowe do pracy jednocześnie, wtedy wybieranie wielu wierszy w Table i zmiana pola Status raz dla wszystkich wybranych jest szybsze niż otwieranie każdego elementu osobno.

OFICJALNA DOKUMENTACJA

Editing items in your project

Krótko

- Przeciągnięcie i upuszczenie karty na Board to najszybszy sposób na zmianę statusu pojedynczego przedmiotu.
- Panel Element pozwala zmieniać wiele pól tego samego przedmiotu jednocześnie.
- Bulk Edit w Table stosuje tę samą wartość pola do wszystkich wybranych elementów jednocześnie.

Opcjonalnie: Filtruj, sortuj i grupuj

filtr zostawia elementy pod zbiór, sortowanie zmienia kolejność wierszy, grupowanie umieszcza cały lista w sekcjach.

SafeSort · Część 2 z 6**Planowanie**
Table i Board pokazują wszystkie elementy Project jednocześnie, ale wraz ze wzrostem listy zadań Potrzebujesz sposobu, by zobaczyć tylko tę część, którą chcesz — do tego wyświetlenia mają filtrowanie, sortowanie i grupowanie.

Chirurgia	Co robi	Przykład SafeSort
Filtr	pozostawia tylko elementy spełniające warunek	pokaż tylko Priority: High
Sortowanie	zmienia kolejność wierszy według wartości pola	posortować Table według Priority
Grupowanie	dzieli elementy na sekcje w zależności od wartości pola	grupować według Area, wszystkie zadania File-system razem

Filtrowanie i grupowanie rozwiązują różne problemy
Filtr usuwa zbędne i pozostawia podzbiór — wygodne, gdy interesuje tylko część listy (na przykład tylko zadania o statusie In Progress właśnie teraz). Grupowanie nic nie usuwa — organizuje cały lista według sekcji, więc od razu widać wszystkie zadania, ale rozłożone według Area lub według Priority.

Filtr — to łańcuch znaków zapytania

Filtr w Project jest zapisany jako zapytanie tekstowe, takie jak `status:"In Review" area:CLI` – można ją wprowadzić ręcznie w filtruj pasek lub zbieraj przez rozwijane menu. Używa się tej samej składni a w repozytorium Issues wyszukiwanie, więc umiejętność jest przenoszona.

OFICJALNA DOKUMENTACJA

Filtering projects

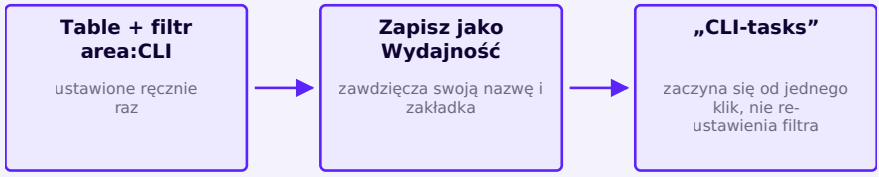
Krótko

- Filtr pozostawia tylko te elementy spełniające warunek — pozostałe są tymczasowo ukryte przed widokiem.
- Sortowanie zmienia kolejność wierszy; grupowanie rozkłada cały lista według wartości pola, nie usuwając elementów.
- Filtr zapisuje się jako tekstowe zapytanie w formie status: „In Review”

Opcjonalnie: Zarządzanie widokami

Skonfigurowane filtrowanie, sortowanie i grupowanie można zapisać jako nazwany widok z osobną kartą.

SafeSort · Część 2 z 6**Planowanie**
Skonfigurowany filtr, sortowanie i grupowanie nie muszą być za każdym razem ponownie składane — Możesz zapisać je jako osobny widok (view) z osobną zakładką i nazwą.



Zapisany widok to ustawienie, a nie kopia danych

Akcja	Rezultat
Stwórz widok	nowa karta ze swoim układem, filtrem i grupowaniem
Widok duplikowany	kopia istniejącej karty jest przydatna jako punkt wyjścia do podobnej konfiguracji
Widok na zmianę nazwy	zmienia tylko etykietę zakładki

Akcja	Rezultat
Zmienić kolejność kart	Zakładki typu Przeciągnij i upuść

Board i Table for SafeSort to dwie zachowane reprezentacje tego samego Project

Board (statusy w kolumnach) i Table (wszystkie pola naraz) nie dublują danych — to dwa różne sposoby spojrzenia na ten sam lista z 14 zadań. Zmiana elementu w jednym widoku jest od razu widoczna w drugim.

OFICJALNA DOKUMENTACJA

Managing your views

Krótko

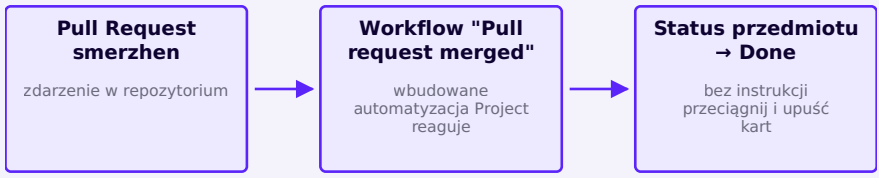
- Widok można zapisać z nazwą i własną kartą — nie trzeba rekonfigurować filtra i grupowania.
- Board i Table to dwa zapisane widoki tej samej listy zadań, a nie oddzielne kopie danych.
- Widoki można duplikować, zmieniać ich nazwy i przestawiać kolejność.

Nieobowiązkowe: wbudowana automatyzacja i auto-add

Wbudowane workflow same zmieniają status elementu w zależności od zdarzeń repozytorium i same dodają nowe Issues do Project.

SafeSort · Część 2 z 6**Planowanie**
Niektóre przejścia między statusami nie muszą być wykonywane ręcznie — Project mają wbudowane automatyzacji (built-in workflows) reagujące na zdarzenia w repozytorium.

Wbudowane workflow	Co robi
Item added to project	ustawia domyślny status nowego elementu
Item reopened	zwraca element do określonego statusu, jeśli Issue zostanie ponownie otwarty
Item closed	umieszcza przedmiot w Done statusie, gdy Issue lub PR jest zamknięty
Pull request merged	zmienia element na status Done przy scalaniu Pull Request
Auto-add to project	automatycznie dodaje nowe Issues i PR do Project pasujących do filtra
Auto-archive items	archiwizuje elementy pasujące do filtra — na przykład wszystko, co jest w Done statusie



Wbudowane workflow reagują na zdarzenia repozytorium i same zmieniają status

W obecnym Project SafeSort zadziałał podobny, ale nie ten sam workflow — wszystkie 14 Issues były już dodane zamknięte, więc włączyło się **Item closed**, nie «Pull request merged»:

Skonfigurowany workflow Item closed → Status: Done w niniejszym Project SafeSort

Prawdziwy workflow, który automatycznie konwertował wszystkie 14 zadań na Done.

Auto-add unikać ręcznego dodawania Issues

Workflow **Auto-add to project** jest konfigurowany przez filtr – na przykład, "Jakieś nowe Issue w repozytorium `Cartesian-School/safesort` ». Dzięki tej zasadzie każdy nowy Issue wchodzi w domyślny status Project I nie musisz pamiętać instrukcji „dodaj do Project”

Włączono workflow Auto-add to project z filtrem is:issue repozytorium safesort

Real, włączony filtr Auto-add już is:issue znajduje wszystkie 14 istniejących Issues.

Automatyzacja nie zastępuje rozwiązań — uwalnia od rutyny
Auto-add eliminuje jedynie mechaniczną akcję „ciągnięcia Issue do Project”

OFICJALNA DOKUMENTACJA

Using the built-in automations

Adding items automatically

Krótko

- Wbudowane workflow zmieniają status elementu w odpowiedzi na zdarzenie repozytorium — na przykład po scaleniu Pull Request.
- Auto-add to project automatycznie dodaje w Project nowe Issues i PR, pasujące do skonfigurowanego filtra.
- Automatyzacja usuwa rutynowe czynności, ale nie podejmowanie decyzji o tym, w jakim stanie powinno być zadanie.

Nieobowiązkowe: archiwizujemy i przywracamy elementy

Archiwizacja usuwa aktywny widok ukończonych zadań bez usuwania ich danych i historii.

SafeSort · Część 2 z 6**Planowanie**

Kiedy zadanie jest zamknięte i trafia do statusu Done, nie trzeba zostawiać jego karty na Board na zawsze — ale też nie warto usuwać wpisu z Project. Do tego jest pośredni krok: **archiwizacja**.

Akcja	Co się dzieje
Archiwum	element znika z aktywnych widoków, ale dane są zachowywane i można je przywrócić
Przywrócić	element jest zaznaczany we wszystkich widokach z tymi samymi polami co przed archiwizacją
Usuń	element jest usuwany z Project nieodwracalnie — sam Issue w repozytorium nie jest przy tym naruszany

Archiwizacja usuwa widok aktywny bez wymazywania historii

Archiwizację można zautomatyzować
Workflow **Auto-archive items** z poprzedniej sekcji robi to samo, filtrując automatycznie – na przykład archiwizuje każdy element, gdy tylko jego status staje się Done, dzięki czemu Board pozostaje czytelny bez ręcznego czyszczenia.

OFICJALNA DOKUMENTACJA

Archiving items from your project

Krótko

- Archiwum usuwa element z aktywnych widoków, ale zachowuje jego dane i pozwala na jego przywrócenie.
- Deletion, w przeciwieństwie do archiwizacji, usuwa element z Project trwale — sam Issue w repozytorium nie jest dotknięty.
- Auto-archive items automatyzuje archiwizowanie ukończonych zadań według filtra.

Opcjonalne: Szablony Project

Project oznaczone szablonem pojawiają się w galerii preset organizacji podczas tworzenia nowego Project.

SafeSort · Część 2 z 6**Planowanie**
Organizacja może oznaczyć gotowy Project jako **szablon** — potem jego struktura (pola, statusy, widoki, ale nie same zadania, jak przy kopiowaniu z Rozdział) pojawia się na liście dostępnych presety przy tworzeniu nowego Project do każdego członka organizacji.

Copy istniejące Project	Project-template
kopiowany raz, ręcznie wybrany Project	jest widoczny w galerii szablonów podczas tworzenia nowych Project
trzeba wiedzieć, który dokładnie Project kopiować	nie musisz szukać — szablon jest dostępny od razu w interfejsie
dostępne dla każdej Project, do której masz prawa	wymaga od właściciela wyraźnego włączenia flagi „utwórz szablon”

Potencjalny kolejny krok dla Cartesian-School, nie część obecnego rozdziału

Project «SafeSort — pierwsze wydanie» rozwiązuję konkretne zadanie tego rozdziału i sam w sobie nie jest oznaczony jako szablon. Ale jego struktura — statusy Backlog/Ready/In Progress/In Review/Done i pola Priority/Area — byłaby odpowiednia dla innych rozdziałów projektowych kursu; oznaczenie go jako szablon do organizacji Cartesian-School — rozsądny przyszły krok, a nie coś, co jest potrzebne SafeSort w tej chwili.

OFICJALNA DOKUMENTACJA

Managing project templates in your organization

Krótko

- Wzorzec Project- pojawia się w galerii szablonów przy tworzeniu nowego Project — nie trzeba szukać, co kopiować.
- Szablon to Project, dla którego właściciel organizacji wyraźnie włączył to ustawienie.
- SafeSort nie jest jeszcze oznaczony szablonem — nie jest to konieczne dla obecnego rozdziału, ale nadaje się jako przyszły etap kursu.

Opcjonalnie: Insights i wykresy Project

Insights renderuje wykresy bezpośrednio z Project pól — opcjonalna, bardziej zaawansowana funkcja.

SafeSort · Część 2 z 6**Planowanie**
Ostatni rozdział o mechanice Project — opcjonalny i bardziej zaawansowany: **Insights** renderuje wykresy bezpośrednio z Project danych, bez eksportowania do Narzędzie firm trzecich.

Wykres jest tworzony z tych samych pól, które już istnieją w elementach Project

Ustalanie harmonogramu	Przykład SafeSort
Grupowanie	przez Status — ile zadań znajduje się obecnie w każdej kolumnie
Filtr	na przykład tylko Area: CI ile zadań jest związanych z konfiguracją CI
Typ harmonogramu	wykres słupkowy, wykres kołowy i inne – zależy od pytania

Przydatne dla dużych list zadań, niewymagane dla czternastu
Insights ujawnia swoje zalety, gdy w Project jest wiele elementów i ręczne obliczanie rozkładu według statusów lub obszarów jest niewygodne. Przy 14 zadaniach SafeSort ten rozkład jest widoczny już po spojrzeniu na Board — Insights tutaj jest to bardziej demonstracja możliwości niż konieczność.

OFICJALNA DOKUMENTACJA

About insights for Projects

Creating charts

Krótko

- Insights renderuje wykresy bezpośrednio z Project pól — bez eksportowania danych do narzędzia firm trzecich.
- Graf jest konfigurowany według grupowania, filtra i typu wizualizacji.
- Korzyść z Insights rośnie wraz z liczbą zadań — na 14 elementach rozkład jest widoczny gołym okiem.

Pierwszy commit w sklonowanym repozytorium

Po klonowaniu w drzewie roboczym istnieje już zestaw plików startowych — pierwszy commit repozytorium SafeSort.

Repozytorium sklonowane (część I), Issues i GitHub Project przeanalizowane (część II) — najwyższy czas zajrzeć do katalogu, który pojawił się po `git clone` i zrozumieć, co już w niej jest.

```
~/safesort $ ls -la
CHANGELOG.md
.git/
.github/
.gitignore
LICENSE
pyproject.toml
README.md
src/
tests/
~/safesort $ cat .gitignore
__pycache__/
*.pyc
.venv/
dist/
build/
*.egg-info/
.pytest_cache/
.safesort/
```

```
safesort/
    .git/
    .github/
    .gitignore
```

README.md

LICENSE

CHANGELOG.md

pyproject.toml

```
                                                    src/


                                                    tests/
```

Prawdziwy zestaw startowy repozytorium SafeSort jest już w twoim katalogu po sklonowaniu.

To jest pierwsze zatwierdzenie repozytorium, `chore: initial project scaffold` : Drzewo robocze (pliki na dysku) plus `.git` (historia zmian) – razem nazywają się **repozytorium**. Co następny Część tej części dodaje treść do tego zestawu — README jest uzupełniona, Python-paczka w środku `src/` , a `.gitignore` już wyklucza pliki serwisowe (bajtkod, pamięć podręczna, wirtualna środowisko, buduj katalogi), aby nie zaśmiecały historii.

Krótko

- git clone natychmiast przynosi drzewo robocze — pliki na dysku — oraz . git jest historią zmian; razem jest to repozytorium.
- SafeSort ma już zestaw startowy: README, LICENSE, CHANGELOG, pyproject.toml, .gitignore — pierwszy commit repozytorium.
- . gitignore wymienia, czego Git nie powinien śledzić: pliki tymczasowe i wygenerowane.

Pierwszy README projektu

README odpowiada na pytania "co to jest?" i "jak zacząć?" jeszcze zanim ktoś otworzy kod.

SafeSort · Część 3 z 6**Projekt**

safesort/

.git/

README.md NOWE

README.md — jeden z plików startowego commita, widzianego w poprzedniej sekcji.

README — pierwszy plik, który widzi osoba, otwierając repozytorium: GitHub automatycznie pokazuje jego zawartość na stronie głównej projektu. W sklonowanym repozytorium README.md już istnieje i jest już zatwierdzony — to część początkowego commita `chore: initial project scaffold` to pokazano w poprzednim sekcję. Teraz jest bardzo krótki:

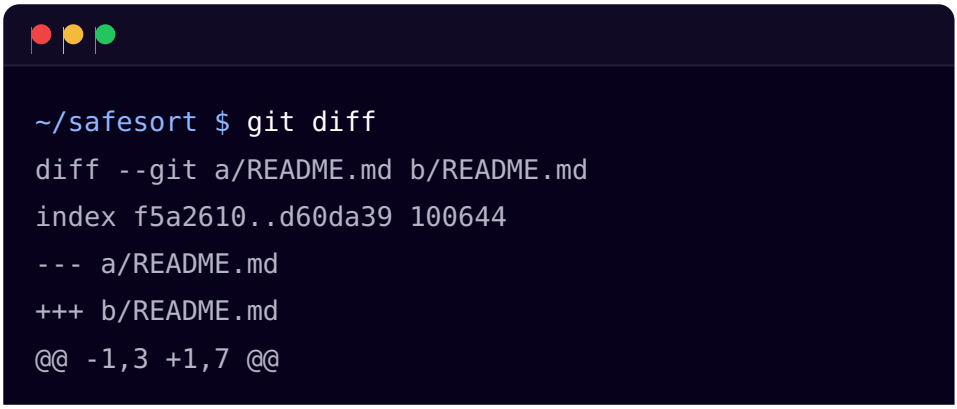
README.md

```
# SafeSort

SafeSort — программа для безопасной сортировки файлов по папкам.
```

Ponieważ plik jest już śledzony przez Git, `git status` w czystej pracy nie pokaże nic drzewu — nie ma nic do dodania ani nic do zobowiązania. Ciekawe zacznie się wtedy Zmiany zostaną wprowadzone w pliku: tak README rośnie w trakcie rozdziału – jest uzupełniany sekcjami, i nie przepisywana od zera. Dodajmy pierwszą taką sekcję:

README.md

```
# SafeSort

SafeSort — программа для безопасной сортировки файлов по папкам.

## Instalacja

pip install -e .
```



```
# SafeSort
SafeSort program do bezpiecznego sortowania plików w
folderach.
+
+## Instalacja
+
+pip install -e .
~/safesort $ git status
M README.md
```

git diff pokazuje różnice linia po linii przed zatwierdzeniem, git status — co dokładnie zmieniło się od ostatniego zatwierdzenia. W miarę pojawiania się w projekcie instalacji, komend i testów, w README dodadzą się takie same krótkie sekcje Usage i Tests. Pełny README projektu, już uzupełniony:

projects/python/safesort/README.md.

Bez wymyślonych ikon

Na stronie projektu na GitHub czasami pojawiają się kolorowe ikony (badge) rodzaju „tests: passing”). Taka ikona jest prawdziwa tylko wtedy, gdy naprawdę obsługuje ją skonfigurowany sprawdzian — wrócimy do tego, gdy podłączymy GitHub Actions. Wstawianie jej wcześniej oznacza pokazywanie czegoś, czego faktycznie jeszcze nie ma.

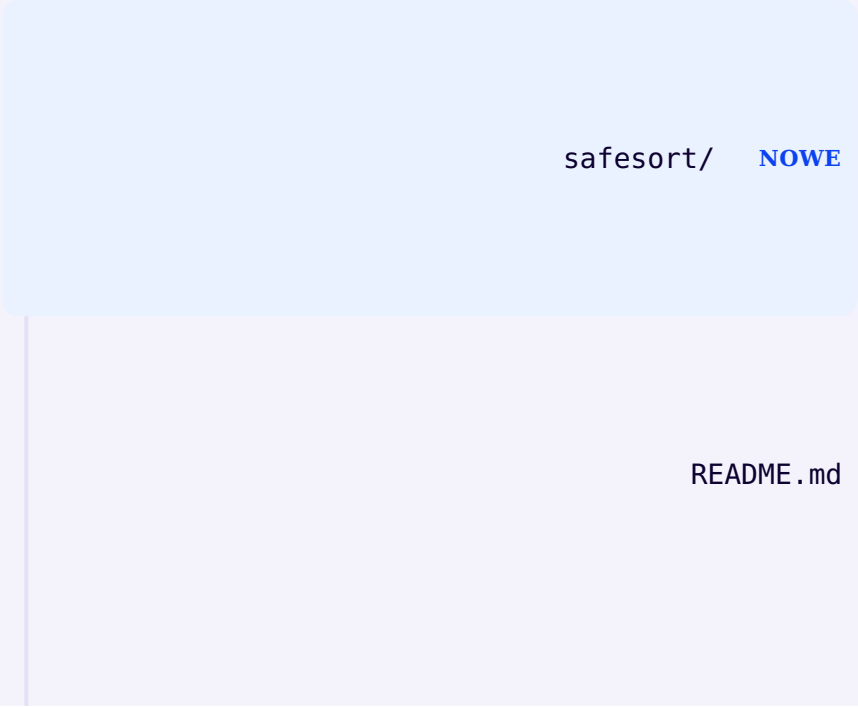
Krótko

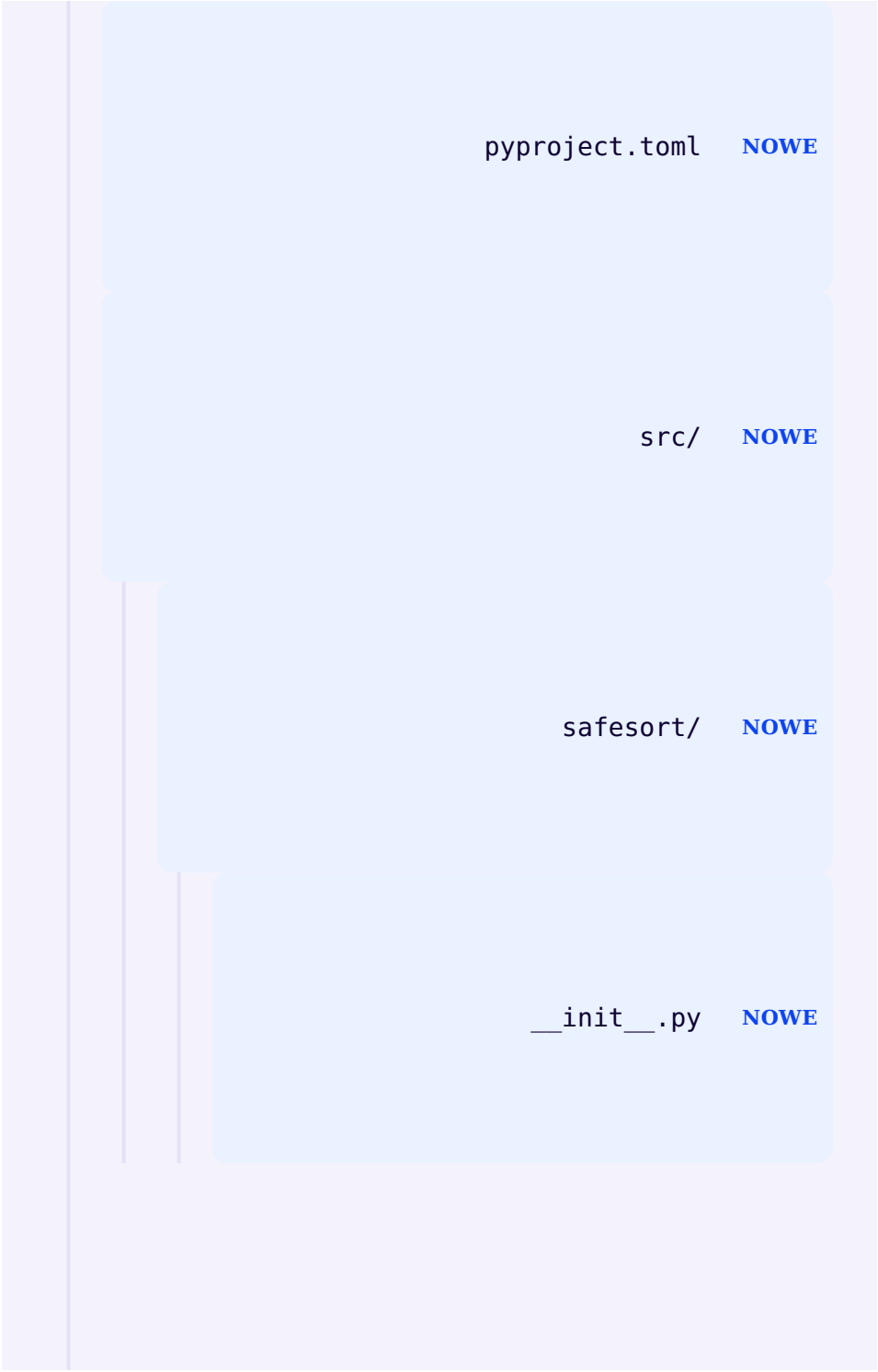
- README.md — pierwsze, co widzi człowiek, który otworzył repozytorium na GitHub.
- README można dodawać w miarę rozwoju projektu, zamiast pisać wszystko za pierwszym razem.
- git diff pokazuje zmiany linijka po linijce git status pokazuje, które pliki są dotknięte.

Planowanie struktury pakietu Python-

src-układ: pakiet znajduje się w src/safesort/, a nie w korzeniu repozytorium — i dlaczego jest to świadomy wybór, a nie dogmat.

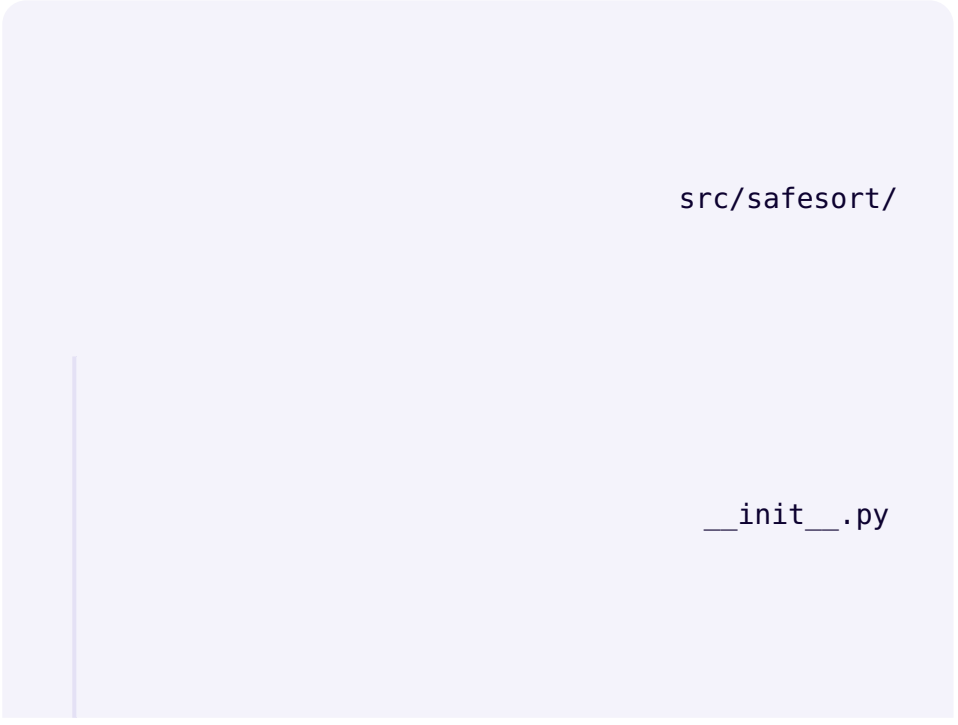
SafeSort · Część 3 z 6**Projekt**
Chaotyczna struktura katalogów nie przeszkadza programowi działać, ale utrudnia jego zrozumienie, testowanie i instalowanie jako pakiet. Dopóki kod nie jest napisany, zarejestrujemy tylko szkielet — i dodamy do niego po jednym pliku na raz, gdy pojawi się zadanie dla niego.




pyproject.toml
NOWE
src/
NOWE
safesort/
NOWE
__init__.py
NOWE

Szkielet drzewa projektu źródłowego: src/safesort/ zawiera importowalny pakiet, pyproject.toml — metadane projektu i ustawienia systemu budowania, a tests/ — testy.

Kod źródłowy znajduje się w środku `src/`, a nie bezpośrednio w katalogu głównym repozytorium — to się nazywa **src- Układ** (src layout). Bez niego na poziomie zainstalowany pakiet i katalog źródłowy byłyby takie same, co łatwo byłoby pomyłkować Sprawdź niewłaściwy kod, który faktycznie został zainstalowany — na przykład, jeśli zapomnisz o ponownej instalacji po edycji pliku.



```
        scanner.py

    classifier.py

        planner.py

      executor.py
```

```
cli.py

duplicates.py

manifest.py

config.py
```

Te pliki pojawią się później, gdy pojawi się zadanie — nie musisz ich teraz pamiętać.

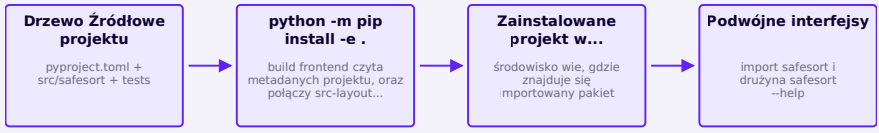
Krótko

- Pakiet na razie składa się tylko z README.md, pyproject.toml, src/safesort/__init__.py i tests/.
- src- układ umieszcza kod źródłowy w src/safesort/ zamiast w korzu repozytorium, aby testy nie mogły przypadkowo uzyskać dostępu do odinstalowanego kodu.
- Pozostałe moduły pojawią się pojedynczo, każdy z nich, gdy będzie miał konkretne zadanie.

pyproject.toml i ustawienie projektu

src/safesort jest importowany, pyproject.toml opisuje Projekt i editable install łączy go z aktualnym środowiskiem.

SafeSort · Część 3 z 6**Projekt**
Łatwo tu połączyć cztery różne obiekty. **Importowana paczka** `src/safesort/` zawiera `__init__.py` i moduły, które Python ładują z `import safesort`. **Projekt** (project) to biblioteka lub aplikacja zaprojektowana dla opakowanie; **drzewo projektu źródłowego** (project source tree) — jego pliki na dysku: `pyproject.toml`, `src/`, `tests/` i dokumentacja. **Pakiet dystrybucyjny** (distribution package) to instalowalna wersja projektu; jeśli chodzi o konkretny To jest archiwum dystrybucji: wheel lub sdist. **Zainstalowany projekt** — Co pojawia się w środowisku Python po instalacji: pliki importowanego pakietu plus metadane, na których narzędzia takie jak `pip` oraz `importlib.metadata` się dowiedzą o nim.



pyproject.toml nie zamienia katalogu w import package: przechowuje metadane i ustawienia projektu, które służą do budowy archiwów dystrybucji (wheel/sdist)

W src-layout momencie uruchomienie Python z rootu repozytorium nie dodaje `src/` w `sys.path`. Dlatego, aż do czasu Instalacje `import safesort` zwykle się kończy `ModuleNotFoundError`. Jest to właściwość oczekiwana układu, a nie znak, że `__init__.py` „nie zadziałało”

pyproject.toml

```
[project]
name = "safesort"
version = "0.1.0"
description = "A safe, non-destructive command-line file
organizer."
dependencies = []

[project.scripts]
safesort = "safesort.cli:main"
```

Pole `version` koniecznie identyfikuje aktualną wersję Pakiet. Na razie po prostu zapiszmy `0.1.0` jako liczba pierwszych edukacyjnych wersje. Jak ułożone są numery wersji i dlaczego są trzy cyfry – przeanalizujmy to bliżej końca Projekt.

String `safesort = "safesort.cli:main"` mówi do instrumentu Ustawienia: Po instalacji stwórz zespół w środowisku `safesort`, która nazywa `main()` z modułu `safesort.cli`. Ten moduł jeszcze nie istnieje; napiszemy to na następnej stronie.

Dlaczego dependencies pozostaje puste
SafeSort nie posiada ani jednej obowiązkowej zależności od podmiotu trzeciego: `pathlib`, `argparse`, `hashlib`, `shutil` oraz `json` są zawarte w standardowej bibliotece Python.

Zainstaluj projekt w trybie edycji

Edytowalna instalacja (editable install) łączy środowisko Python bezpośrednio do kodu źródłowego projektu: zmiany w plikach `src/safesort/` są widoczne natychmiast, bez konieczności ponownego montażu.

Terminal (środowisko aktywowane)

```
python -m pip install -e ".[dev]"
python -c "import safesort; print(safesort.__file__)"
safesort --help
```

OFICJALNA DOKUMENTACJA

Python Tutorial — Packages
Packaging Python Projects
Distribution package vs. import package
src layout vs flat layout
Writing pyproject.toml

Drużyna łańcuch znaków SafeSort

argparse analizuje pięć podinstrukcji SafeSort i przypisuje każdej osobnej funkcji obsługi.

SafeSort · Część 3 z 6**Projekt**

src/safesort/

__init__.py

cli.py NOWE

Pierwszy plik z prawdziwą logiką: cli.py.

SafeSort będzie sterowany pięcioma podkomendami: `scan`, `plan`, `apply`, `duplicates` oraz `undo`. Do analizy Moduł odpowiedzi na argumenty wiersza poleceń `argparse` z standardowej biblioteki — to on generuje tekst `--help`.

src/safesort/cli.py

```
def build_parser() -> argparse.ArgumentParser:
    parser = argparse.ArgumentParser(
        prog="safesort",
        description=(
            "SafeSort: a safe, non-destructive file organizer.
"
            "scan/plan/duplicates are read-only; 'apply' sorts
and 'undo' restores files."
        ),
    )
    subparsers = parser.add_subparsers(dest="command",
required=True)

    scan_parser = subparsers.add_parser("scan", help="...")
    scan_parser.add_argument("root", type=Path,
help="Directory to scan.")
    # ... Podobnie dla plan, apply, duplicates, undo
    return parser
```

add_subparsers(dest="command", required=True)
`dest="command"` umieszcza nazwę wywołanego podpokarmu w `args.command`. `required=True` wymusza argparse samodzielne wygenerowanie wyraźnego błędu, jeśli użytkownik uruchomi `safesort` bez żadnego podpolecenia nie musisz pisać tego czeku ręcznie.

Od rozdzielonych argumentów do wyniku

Każda podinstrukcja odpowiada jednej funkcji obsługi, a słownik wiąże nazwę Polecenia o pożądanej funkcji:

src/safesort/cli.py

```
_HANDLERS = {
    "scan": cmd_scan,
    "plan": cmd_plan,
    "apply": cmd_apply,
    "duplicates": cmd_duplicates,
    "undo": cmd_undo,
}

def main(argv: list[str] | None = None) -> int:
    parser = build_parser()
    args = parser.parse_args(argv)
    handler = _HANDLERS[args.command]
    return handler(args)
```

Taki słownik — ten sam trik co w grze „Kamień, papier, nożyczki” z załącznika do tej części: zamiast łańcucha `if args.command == "scan": ... elif ...` potrzebną funkcję po prostu wyszukuje się po kluczu.

Każda funkcja obsługi zwraca liczbę całkowitą: `0` at niezerowa wartość błędu to kod zwrotny programu, który widzi system operacyjny i każdy skrypt, który wywołuje `safesort` z innego miejsca.

Zainstaluj pakiet i uruchom polecenie — argparse generuje tekst pomocy sam, bez pojedynczego wiersza napisanego ręcznie:

```
~/safesort $ safesort --help
```

```
usage: safesort [-h]
{scan,plan,apply,duplicates,undo} ...

SafeSort: a safe, non-destructive file organizer.
scan/plan/duplicates are
read-only; 'apply' sorts and 'undo' restores files.

positional arguments:
{scan,plan,apply,duplicates,undo}
scan List files found under ROOT, grouped by category
(read-only).
plan Show the moves that would be made under ROOT,
without
changing anything (read-only).
apply Move files under ROOT into Sorted/<category>/
and
record an undo manifest.
duplicates Report groups of files with identical
content under
ROOT (read-only, never deletes).
undo Undo the most recent 'apply' run recorded under
ROOT.

options:
-h, --help show this help message and exit
```

Praktyka: Analizowanie argumentów wiersza poleceń
Interaktywny Laptop w przeglądarce: Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/23-07/index.html)

OFICJALNA DOKUMENTACJA

argparse — Parser for command-line options

Krótko

- argparse.ArgumentParser add_subparsers() analizuje pięć podkomend SafeSort i sam tworzy tekst — help.
- dest=„command”
- Każdy handler zwraca 0 przy sukcesie i wartość różną od zera przy błędzie — to jest kod zwrotny całego programu.

pathlib: pracujemy ze ścieżkami i katalogami

Path opisuje ścieżkę obiektu wraz z jego operacjami. Służy do budowy modelu danych SafeSort.

SAFESORT · CZĘŚĆ 4 Z 6

Git i GitHub → Planowanie → Projekt →
Wdrożenie → Testy i CI → Premiera

Wszystko, co SafeSort robi z plikami, zaczyna się od ścieżek. Moduł `pathlib` ze standardowej biblioteki opisuje ścieżkę nie jako łańcuch znaków, lecz jako Cel `Path` własnymi operacjami.

String	Path
"/home/anna/Downloads/report.pdf"	Path("/home/anna/Downloads/report.pdf")
ręczne przeszukiwanie . split('/')	gotowe atrybuty fajl.parent, fajl.name, fajl.suffix, fajl.stem
łączenie łańcuch znaków w celu połączenia ścieżek	operator / tworzy ścieżkę: koren / 'report.pdf'

pathlib_osnovy.py

```
from pathlib import Path

koren = Path("~/Downloads").expanduser()
fajl = koren / "otchet.pdf"
```

```
>>> fajl.name
'otchet.pdf'
>>> fajl.suffix
'.pdf'
>>> fajl.stem
'otchet'
>>> fajl.parent
PosixPath('/home/anna/Downloads')
```

Operator / dla torów

Klasa `Path` operator zostaje nadpisany `/` : nie dzieli liczb, lecz skleja część ścieżki nową nazwą, automatycznie podstawiając właściwy separator dla obecnego systemu operacyjnego. Zbieraj ścieżki przez konkatenację łańcuchów (`koren + "/" + "otchet.pdf"`) nie jest potrzebne.

SafeSort: FileInfo Model danych

Skaner SafeSort nie ogranicza się do listy ścieżek. Opisuje każdy znaleziony plik Mały obiekt niezmienny:

src/safesort/models.py

```
@dataclass(frozen=True)
class FileInfo:
    path: Path
    size: int
    extension: str
```

`frozen=True` zabrania przypisywania pól już utworzonych `FileInfo`. Ta nieruchomość *obiektu-wartości*, a nie ochronę systemu plików. Funkcja może otrzymywać obiekt frozen i nadal zapisywać pliki, uzyskiwać dostęp do sieci lub wykonywać inne skutki uboczne.

Dane niezmienne nie oznaczają czystej funkcji
`build_plan()` nie zmienia dysku według osobnego kontraktu: API zwraca plan, implementacja nie wywołuje move/write, a test sprawdza brak zmian. `frozen=True` tylko nie pozwala przekazać pól modelu. To różne granice i obie są potrzebne.

Praktyka: operacje z Path
Interaktywny Laptop w przeglądarce: Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/23-08/index.html)

OFICJALNA DOKUMENTACJA

pathlib — Object-oriented filesystem paths
dataclasses

Zeskanuj katalog

scan() czyta system plików ściśle — przechodzi przez katalog rekurencyjnie i nie zmienia ani jednego pliku.

SafeSort · Część 4 z 6**Wdrożenie**

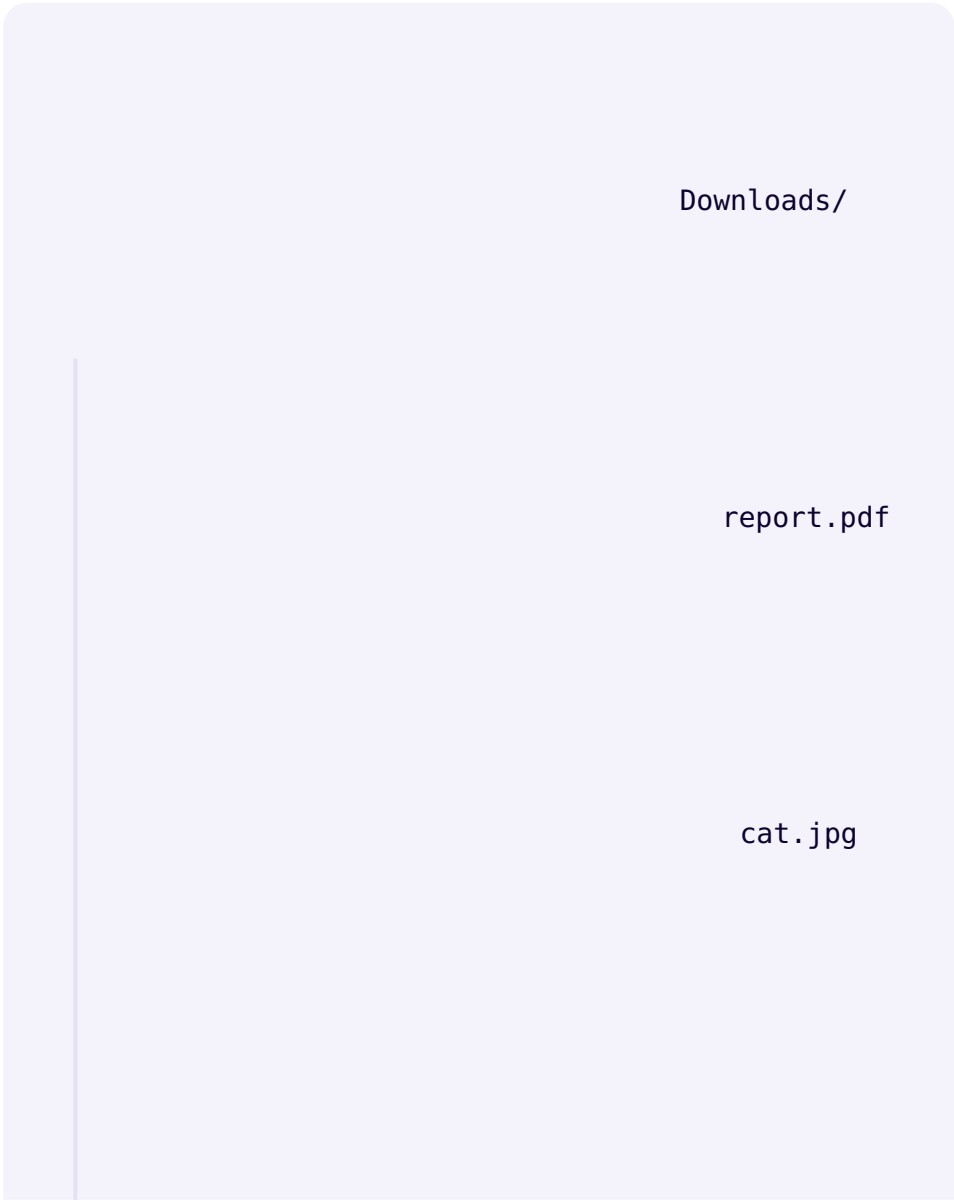
GITHUB ISSUE #1 · PROJECT SAFESORT PIERWSZE WYDANIE

Add directory scanner

Area: **Filesystem** Priority: **High**

```
git switch -c feat/directory-scanner
```

Pierwszy krok SafeSort — przejrzeć katalog i sporządzić lista plików. Funkcja `scan()` — jedna z trzech komend wyłącznie odczytujących SafeSort: ona tylko wywołuje `iterdir()` oraz `stat()`, Nigdy nie tworząc, nie przenosząc ani nie usuwając niczego z dysku.



```
                                                archive.zip
```

Ten łańcuch będzie rósł: następne strony dodadzą do niego nowe ogniwa.

src/safesort/scanner.py

```python
def scan(root: Path, config: Config) -> list[FileInfo]:
    root = Path(root)
    excluded = config.excluded_names()
    results: list[FileInfo] = []
    _scan_dir(root, excluded, results)
    return results
```

Samo przejście jest rekurencyjne: `_scan_dir()` wchodzi w każdy podkatalog i nazywa się ponownie:

src/safesort/scanner.py

```
for entry in entries:
    if entry.is_symlink():
        continue  # symboliczne linki pominięte — zobacz
następną stronę
    if entry.is_dir():
        if entry.name in excluded:
            continue  # Katalogi wykluczone są pomijane —
zobacz następną stronę
        _scan_dir(entry, excluded, results)
    elif entry.is_file():
        size = entry.stat().st_size
        results.append(FileInfo(path=entry, size=size,
extension=entry.suffix.lower()))
```

Katalog, którego nie można odczytać, nie powinien zatrzymywać całego skanowania

Jeśli nie ma wystarczających praw dostępu do podkatalogu, `iterdir()` przyczyny `PermissionError`. Implementacja `_scan_dir()` przechwytuje ten błąd dla każdego podkatalogu osobno, zapisuje ostrzeżenie w dzienniku i kontynuuje skanowanie pozostałych katalogów — dzięki temu jeden niedostępny podkatalog nie przerywa całego przeglądu. Później omówimy to szczegółowo wraz z innymi błędami systemu plików.

Skaner można sprawdzić teraz — rzeczywisty wydruk:

```
~/safesort $ safesort scan ~/Downloads
Files scanned: 9
Documents: 3
Images: 1
Archives: 1
Other: 4
```

Punkt kontrolny testu: Wymagania stają się regresją

tests/test_scanner.py

```
def test_scan_finds_file_without_changing_it(tmp_path):
    source = tmp_path / "report.pdf"
    source.write_bytes(b"report")

    files = scan(tmp_path, Config())

    assert [item.path for item in files] == [source]
    assert source.read_bytes() == b"report"
```

Najpierw wymaganie jest obserwowane na jednym przykładzie, następnie test zapisuje je jako kontrakt. Późniejsza sekcja testowania rozszerzy tę podstawę fixtures i edge cases.

Praktyka: Skanowanie prawdziwego katalogu
Potrzebny jest dostęp do prawdziwego systemu plików – uruchamianego lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/23-09/index.html)

Które katalogi nie wymagają przeglądania

Katalog wyników i katalog usług SafeSort zawsze wykluczone — a symboliczne linki nie są w ogóle śledzone.

SafeSort · Część 4 z 6**Wdrożenie**

Downloads/

```
report.pdf — ✓ skan

images — ✓ skan/

Sorted — × pomij/

.git — × skip/
```

```
                                                    . venv — ×
                                                    skip/
```

Skaner SafeSort pomija nie tylko nieistotne pliki, ale także całe katalogi — z dwóch różnych powodów. Pierwszy: niektóre katalogi z góry nie należą do plików użytkownika — na przykład, `.git` lub `.venv`. Druga, bardziej subtelna: własny katalog efektów, `Sorted/`, oraz katalog usług samego SafeSort, `.safesort/` powinno być wykluczony *zawsze* — inaczej powtórka już by się zaczęła sortować Posortowane pliki i niestandardowe dzienniki aktywności.

src/safesort/config.py

```
def excluded_names(self) -> frozenset[str]:
    return frozenset({*self.exclude, self.destination,
STATE_DIRNAME})
```

Wyjątki konfigurowalne i obowiązkowe — różne zestawy

`self.exclude` jest lista, którą użytkownik może zmienić w pliku ustawień, co omówimy później. Katalog wyników (`self.destination`, domyślnie `Sorted`) oraz katalog usług `.safesort` są dodawane do wyjątków dynamicznie, przy każdym wywołaniu — nie można ich przypadkowo usunąć z listy wyjątków poprzez ustawienia.

Odsyłacze symboliczne: świadomie pomijane

Symboliczne łącze — plik lub katalog, który w rzeczywistości wskazuje na inne miejsce w systemie plików. SafeSort nie podąża w ogóle za symbolicznymi łączami, ani do plików, ani katalogów:

Dlaczego odnośniki są pomijane	Co by się stało inaczej
Link może wskazywać poza zeskanowany katalog	SafeSort mógł przenieść plik, który nie należy do wybranego katalogu
Referencja katalogowa może tworzyć pętlę indeksowania	Rekurencyjne skanowanie mogłoby nigdy się nie zakończyć
Zachowanie związane z linkami symfonicznymi nie zawsze jest oczywiste nawet dla doświadczonych użytkowników	Pierwsza wersja programu wybiera przewidywalne zachowanie zamiast bardziej złożonych

To rozwiązanie łatwo sprawdzić: skaner używa `entry.is_symlink()` przed jakimkolwiek innym sprawdzeniem i, jeśli to symboliczne połączenie, prowadzi od razu do Do kolejnego elementu katalogowego — ten łańcuch znaków już został napotkany w `_scan_dir()` na poprzedniej stronie.

Praktyka: sprawdzanie wykluczonych katalogów
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/23-10/index.html)

CHECKPOINT · ISSUE #1

```
git commit -m "feat: add directory scanner and configuration"
```

Zlał się jak Pull Request #15. Ta sama gałąź przy okazji przyniosła podstawowy moduł Config, na którym później opierają się Issues #10 i #11 (strony 23-20–23-22).

Status prawdziwego Project: **Done**

Zdefiniuj kategorię pliku

Klasyfikacja z definicji jest praktyczną heurystyką, a nie dowodem na to, co naprawdę znajduje się w pliku.

SafeSort · Część 4 z 6**Wdrożenie**

GITHUB ISSUE #2 · PROJECT SAFESORT PIERWSZE WYDANIE

Add file classification

Area: **Filesystem** Priority: **High**

```
git switch -c feat/classifier
```

Każdemu znalezionemu plikowi trzeba przypisać kategorię — dokumenty, obrazy, wideo i tak dalej. SafeSort określa kategorię według rozszerzenia pliku: prosta i przewidywalna zasada, która obejmuje większość praktycznych przypadków.

Do łańcucha dodaje się nowe ogniwo — klasyfikator.

Plik	Kategoria
report.pdf	documents
photo.jpg	images
backup.zip	archives
unknown.xyz	other

src/safesort/config.py

```
DEFAULT_EXTENSIONS: dict[str, list[str]] = {
    "documents": [".pdf", ".docx", ".txt", ".odt"],
    "images": [".jpg", ".jpeg", ".png", ".webp"],
    "video": [".mp4", ".mkv", ".mov"],
    "audio": [".mp3", ".wav", ".flac"],
    "archives": [".zip", ".tar", ".gz", ".7z"],
    "code": [".py", ".js", ".ts", ".rs", ".java"],
    "data": [".json", ".csv", ".xml"],
}
```

src/safesort/classifier.py

```
OTHER_CATEGORY = "other"

def classify(extension: str, mapping: dict[str, list[str]]) ->
str:
    normalized = extension.lower()
    for category, extensions in mapping.items():
        lowered = {ext.lower() for ext in extensions}
        if normalized in lowered:
            return category
    return OTHER_CATEGORY
```

Klasyfikacja w rozszerzeniu jest heurystyką, a nie dowodem treści

Rozbudowa `.pdf` oznacza tylko, że nazwa pliku kończy się na `.pdf` nic nie stoi na przeszkodzie, by zmienić nazwę dowolnego pliku tak, by jego rozszerzenie nie pasowało do rzeczywistego formatu. SafeSort celowo nie zagląda do środka pliku: byłoby wolniejsze, mniej przewidywalne i wykraczałoby poza zadanie „sortowania plików według tego, czym się nazywają”

plik o rozszerzeniu, który nie należy do żadnej kategorii, należy do `"other"` — w ten sposób klasyfikacja pozostaje kompletna: dowolny plik Zawsze istnieje jakaś kategoria, nawet jeśli jest najszersza.

Praktyka: classify() i kategorie niestandardowe

Interaktywny Laptop w przeglądarce: Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/23-11/index.html)

CHECKPOINT · ISSUE #2

```
git commit -m "feat: add file classification"
```

Zlał się jak Pull Request #16 z osobnej, niezależnej gałęzi, która nie jest powiązana z innymi Issues.

Status prawdziwego Project: **Done**

Od analizy do planu działania

Plan ruchu to po prostu dane: można go wyświetlać, sprawdzać lub odrzucać bez dotykania pojedynczego pliku.

SafeSort · Część 4 z 6**Wdrożenie**

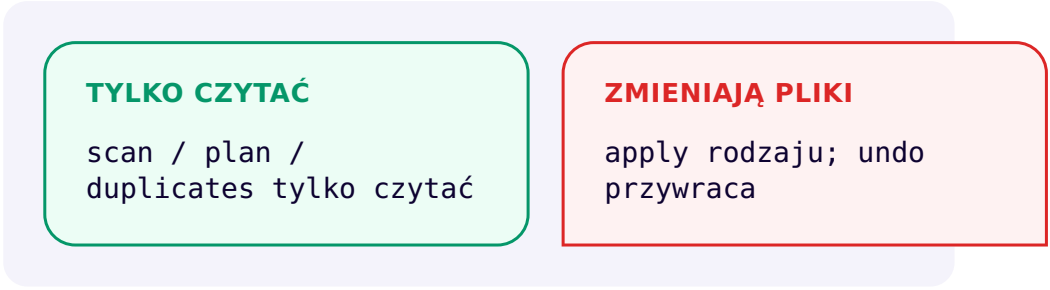
GITHUB ISSUE #3 · PROJECT SAFESORT PIERWSZE WYDANIE

Add safe organization plan

Area: **Safety** Priority: **High**

```
git switch -c feat/plan-and-collisions
```

SafeSort ma lista znalezionych plików oraz regułę klasyfikacji — ale to jest Jeszcze nie mam planu działania. **Plan** jest lista ruchów konkretnych: skąd i gdzie każdy plik przesuwa się, jeśli użytkownik potwierdzi wykonanie.



src/safesort/models.py

```
@dataclass(frozen=True)
class MoveOperation:
    source: Path
    destination: Path

@dataclass(frozen=True)
class SortPlan:
    root: Path
    operations: tuple[MoveOperation, ...]
```

Plan to dane, a nie działanie

`SortPlan` nie zawiera wywołań przenoszących pliki, tylko opis tego, co *może być* do zrobienia. Ta idea jest kluczowa dla całej architektury SafeSort: dopóki obiekt pozostaje danymi, może być wyświetlany, testowany, zapisywany lub po prostu odrzucany bez ryzyka użycia pojedynczego pliku użytkownika.

Funkcja `build_plan()` tworzy plan na podstawie listy plików z skaner bez dotykania dysku ani razu, chyba read-only sprawdzeniu, czy plik już istnieje z tą nazwą na miejscu docelowym. Co robić, jeśli jesteś, to temat kolejnego etapu.

src/safesort/planner.py

```
def build_plan(files: list[FileInfo], root: Path, config:
Config) -> SortPlan:
    dest_root = Path(root) / config.destination
    operations = []
    for file in files:
        category = classify(file.extension, config.extensions)
        dest_dir = dest_root / category
        candidate = dest_dir / file.path.name
        destination = _resolve_collision(candidate, reserved)
        operations.append(MoveOperation(source=file.path,
destination=destination))
    return SortPlan(root=root, operations=tuple(operations))
```

Punkt kontrolny testu: Czyste planowanie

tests/test_planner.py

```
def
test_build_plan_describes_move_without_executing_it(tmp_path):
    source = tmp_path / "report.pdf"
    source.write_bytes(b"report")
    file = FileInfo(source, source.stat().st_size, ".pdf")

    plan = build_plan([file], tmp_path, Config())

    assert plan.operations[0].destination == tmp_path /
"Sorted/documents/report.pdf"
    assert source.exists()
    assert not plan.operations[0].destination.exists()
```

Praktyka: Zbuduj plan na podstawie listy plików
interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/23-12/index.html)

CHECKPOINT · ISSUE #3

```
git commit -m "feat: add safe organization plan with
collision handling"
```

> Zlał się jak Pull Request #17. Nagłówek PR nazywa od razu obie rzeczy: ten sam PR zamknął też Issue #6 (obsługa konfliktu nazw), pokazany na osobnej stronie 23-14, ponieważ jako lekcja to dwa różne, samodzielne pojęcia, nawet jeśli w repozytorium trafiły do tej samej gałęzi.
>
> Status prawdziwego Project: **Done**

Tryb Podglądu

Komenda plan — podgląd bez efektów ubocznych: ten sam plan co apply, ale bez żadnej zmiany na dysku.

SafeSort · Część 4 z 6**Wdrożenie**

> GITHUB ISSUE #4 · PROJECT „SAFESORT — PIERWSZE WYDANIE”
>
> **Add dry-run CLI (scan/plan)**
>
> Area: **CLI** Priority: **Medium**
>
> `git switch -c feat/cli`

Komenda `plan` — jedyne, co trzeba zrobić z gotowym obiektem `SortPlan` uzyskać pełny **podgląd** (dry run): zbiera plan i go dedukuje rozmiar na ekranie, nie powodując żadnej zmiany dysku.

src/safesort/cli.py

```
def cmd_plan(args: argparse.Namespace) -> int:
    root = args.root
    config, code = _load_config(root)
    files = scan(root, config)
    plan = build_plan(files, root, config)
    print(f"{len(plan.operations)} move operations planned.")
    print("No files have been changed.")
    return 0
```

```
~/safesort $ safesort plan ~/Downloads
9 move operations planned.
No files have been changed.
```

Drugi łańcuch znaków wynik — `"No files have been changed."` — to nie forma. Sprawdzimy ją uczciwie: policzymy sumy kontrolne plików przed i po `plan` porównać.

```
~ $ sha256sum Downloads/* > before.txt
~ $ safesort plan ~/Downloads
9 move operations planned.
No files have been changed.
~ $ sha256sum Downloads/* > after.txt
~ $ diff before.txt after.txt
```

`diff` nie wydrukowało ani jednej linii, pliki naprawdę nie zmieniły się. Później ta sama kontrola zostanie wykonana przez test automatyczny, a nie ręczne porównanie.

scan, plan i duplicates używają tego samego pliku lista

Trzy read-only polecenia SafeSort — `scan`, `plan` oraz `duplicates` — zaczynają się jednakowo: wywołują `scan()` zdobyć lista plików. Wtedy ich ścieżki się rozdzielają: `plan` tworzy plan, `duplicates` szuka dopasowania w treści, oraz `scan` liczy pliki tylko według kategorii.

W historii repozytorium cli.py zbierano w jednym PR, na samym końcu

Ta książka wprowadza `cli.py` stopniowo: `cmd_plan()` tutaj, pozostałe podkomendy dalej w miarę gotowości funkcji, które wywołują. W prawdziwym repozytorium SafeSort było odwrotnie: argparse- opakowanie dla wszystkich pięciu podkomend (scan, plan, apply, duplicates, undo)) zostało napisane i scalone jednym PR na samym końcu, gdy wszystkie pięć funkcji już istniało. To dwa różne porządki: porządek, w którym wygodnie jest wyjaśniać, i porządek, w którym faktycznie pisano kod.

Krótko

- plan wyświetla ten sam SortPlan co apply, ale wyświetla tylko swój rozmiar na ekranie.
- scan, plan i duplicates zaczynają się tak samo — od wezwania do scan() — i idą dalej.
- Stwierdzenie „pliki nie zostały zmienione”

PUNKT KONTROLNY · ISSUE #4

```
git commit -m "feat: add command-line interface"
```

W rzeczywistej historii repozytorium jest to Pull Request #21, która była ostatnią z dziewięciu połączoną po Issue #9 (duplikaty), a nie bezpośrednio po planie, jak jest w kolejności tej książki: w tym momencie wszystkie funkcje potrzebujące powiązania CLI już istniały w repozytorium.

Status prawdziwego Project: **Done**

Bezpiecznie przenoszcie pliki

Bezpośrednie sortowanie jest wykonywane przez apply_plan(): przed każdym ruchem, funkcja ponownie sprawdza konflikt. Undo pozostaje osobną operacją odwrotną.

SAFESORT · CZĘŚĆ 4 Z 6

Git i GitHub → Planowanie → Projekt →
Wdrożenie → Testy i CI → Premiera

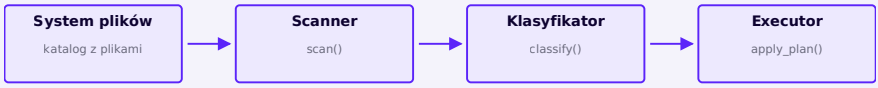
GITHUB ISSUE #5 · PROJECT SAFESORT PIERWSZE WYDANIE

Add explicit apply operation

Area: **Safety** Priority: **High**

```
git switch -c feat/apply-operation
```

Do tej pory żadna funkcja w łańcuchu bezpośredniego sortowania nie zmieniła SafeSort pliku system. Moduł `executor.py` wykonuje operację bezpośrednią: Funkcja `apply_plan()` ma gotowy plan i tylko się porusza pliki są w nim wymienione. Odwracaj działanie `undo` też przenosi pliki, ale przywraca je zgodnie z zarejestrowanym manifestem.



W łańcuchu sortowania do przodu rekord zaczyna się tylko od executor; undo tworzy osobny łańcuch odwrotny.

TAK BYŁO

Downloads/

report.pdf

photo.jpg

archive.zip

STAŁO SIĘ

```
images/
    photo.jpg
archives/
    archive.zip
```

apply wykonuje bezpośrednie sortowanie; undo później może wykonywać ruchy odwrotne na manifestie.

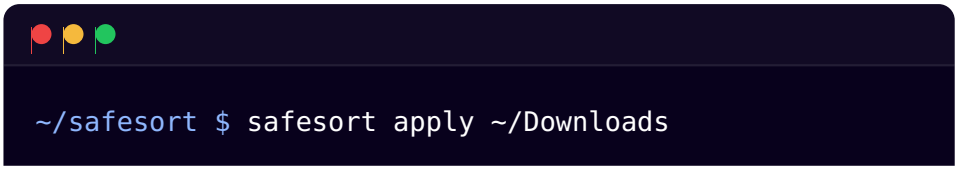
src/safesort/executor.py

```
def apply_plan(plan: SortPlan) -> list[CompletedMove]:
    results = []
    for operation in plan.operations:
        source, destination = operation.source,
operation.destination
        destination.parent.mkdir(parents=True, exist_ok=True)

        if destination.exists() or destination.is_symlink():
            results.append(CompletedMove(source, destination,
completed=False,
                                         error="destination
already exists"))
            continue

        shutil.move(str(source), str(destination))
        results.append(CompletedMove(source, destination,
completed=True))
    return results
```

Ruch batch nie jest transakcją bazy danych

Jeśli jeden plik w środku partii nie uda się przenieść (np. prawa dostępu zmieniły się między skanowaniem a wykonaniem), nie powinno to zatrzymywać przetwarzania pozostałych plików i nie powinno cofać już wykonanych przeniesień — system plików nie umie tak atomowo cofać grupy operacji, jak robi to baza danych. Dlatego `apply_plan()` przetwarza każdy ruch niezależnie i na końcu zwraca uczciwy raport o tym, co naprawdę stało się z każdym plikiem.

Zwracaj uwagę na weryfikację `destination.exists()` prosty Przed przeprowadzką, choć planista już uniknął tego konfliktu podczas fazy realizacji planu. Między tworzeniem planu a jego realizacją na dysku mógł pojawić się nowy plik, na przykład, jeśli sam użytkownik w danym momencie coś tam umieścił. Brak ponownej weryfikacji `shutil.move()` na systemach opartych na POSIX bezszelestnie nadpisywały Taki plik jest SafeSort nie nadpisuje bezgłośnie plików pod żadnym pozorem.



```
Applied 9 moves.
Manifest written to:
.safesort/history/20260824T085400685280.json
```

punkt kontrolny testu: apply zmienia tylko podaną ścieżkę

tests/test_executor.py

```
def test_apply_plan_moves_one_file(tmp_path):
    source = tmp_path / "report.pdf"
    destination = tmp_path / "Sorted/documents/report.pdf"
    source.write_bytes(b"report")
    plan = SortPlan(tmp_path, (MoveOperation(source,
destination),))

    [result] = apply_plan(plan)

    assert result.completed is True
    assert destination.read_bytes() == b"report"
    assert not source.exists()
```

Praktyka: Przenoszenie plików w katalogu tymczasowym
Potrzebny jest dostęp do prawdziwego systemu plików – uruchamianego lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/23-13/index.html)

PUNKT KONTROLNY · ISSUE #5

`git commit -m "feat: add explicit apply operation"`

Zlał się jak Pull Request #18. W opisie PR wyraźnie wskazano `Closes #5` w opisie, więc Issue #5 automatycznie zamyka się w momencie połączenia.

Status prawdziwego Project: **Done**

Co zrobić, jeśli nazwisko jest już zajęte

Ani na dysku, ani w ramach jednego planu dwa pliki nigdy nie otrzymają tej samej nazwy docelowej.

SafeSort · Część 4 z 6**Wdrożenie**

GITHUB ISSUE #6 · PROJECT SAFESORT PIERWSZE WYDANIE

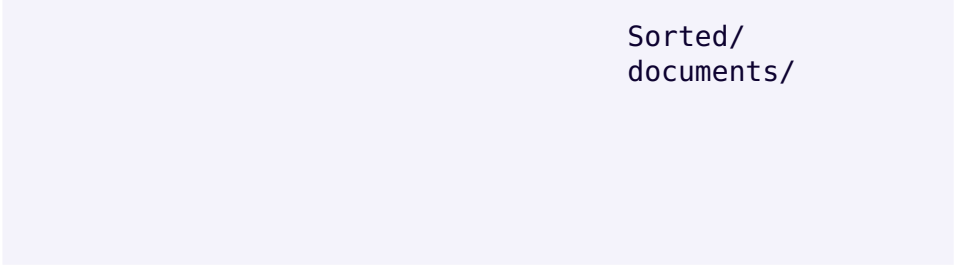
Handle destination name collisions

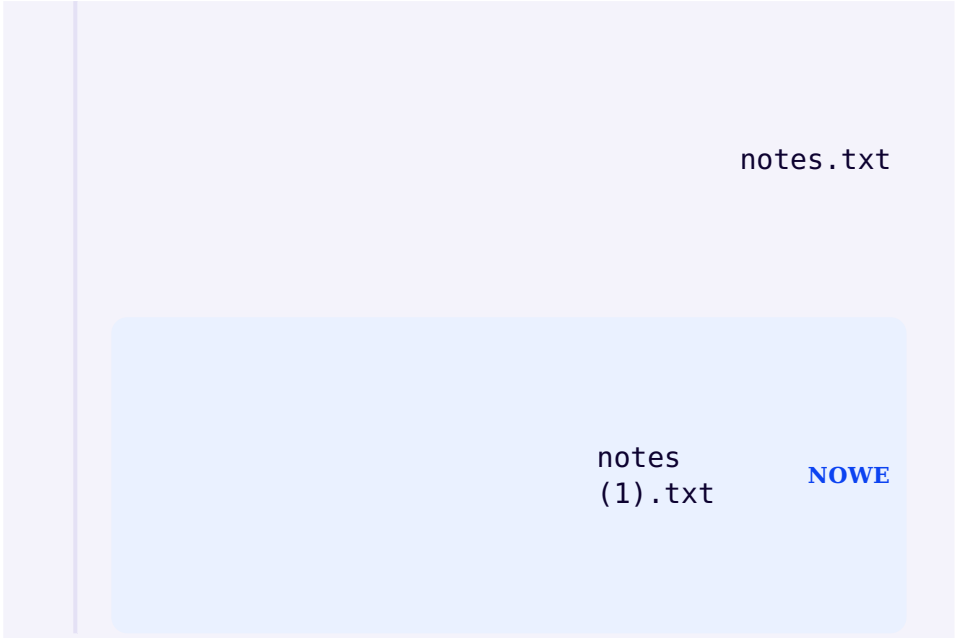
Area: **Safety** Priority: **Medium**

```
git switch -c feat/plan-and-collisions
```

Dwa pliki o tej samej nazwie mogą trafić do tej samej kategorii — na przykład dwa różne `notes.txt` z różnych podkatalogów katalogu źródłowego. Gdyby SafeSort po prostu przesunął plik na istniejący, drugi plik zniknąłby bezgłośnie — Katalog docelowy miałby tę samą nazwę `notes.txt`, ale z zawartością tylko jednego z dwóch plików, bez żadnego ostrzeżenia o tym.

```
~/safesort $ safesort apply ~/Downloads
Applied 1 moves.
```





Plik, który już był w Sorted/documents/, pozostał nietknięty; nowemu nadano wolne imię.

SafeSort nigdy nie rozwiązuje konfliktu nazewnictwa przez nadpisanie: zamiast tego znajduje wolne miejsce nazwa według jasnego schematu.

src/safesort/planner.py

```
def _resolve_collision(candidate: Path, reserved:
set[Path]) -> Path:
    if candidate not in reserved and not candidate.exists():
        return candidate

    stem, suffix, parent = candidate.stem, candidate.suffix,
candidate.parent
    counter = 1
    while True:
        alternative = parent / f"{stem} ({counter}){suffix}"
        if alternative not in reserved and not
alternative.exists():
            return alternative
        counter += 1
```

Przykład

```
otchet.pdf уже существует в Sorted/documents/
→ следующий otchet.pdf получит имя otchet (1).pdf
→ если и оно занято — otchet (2).pdf, и так далее
```

reserved — konflikty w obrębie jednego planu, nie tylko na dysku

Checki `not candidate.exists()` na tyle, by kolidować z już istniejącym plikiem na dysku — ale nie kiedy *dwa pliki z tego samego planu* ubiegają się o tę samą nazwę w tym samym czasie, dopóki żaden z nich nie został jeszcze przeniesiony. Wiele `reserved` pamięta imiona już użyte w tym planie, więc drugi plik zostanie przyjęty `otchet (1).pdf`, nawet jeśli nie ma `otchet.pdf`.

Praktyka: bezpieczna nazwa przy konflikcie

interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/23-14/index.html)

PUNKT KONTROLNY · ISSUE #6

```
git commit -m "feat: add safe organization plan with collision handling"
```

Ta sama gałąź i ten sam Pull Request #17, który zamykał Issue #3 na stronach 23-11. Planowanie i obsługa konfliktów nazw było jednym, ściśle powiązanym zadaniem w terminologii kodu, ale pozostają to dwie różne Issues i dwie różne lekcje.

Status prawdziwego Project: **Done**

Dziennik wykonanych operacji

Każdy apply tworzy manifest JSON-, pokazujący, co naprawdę się wydarzyło – to jedyne źródło danych do cofnięcia.

SafeSort · Część 4 z 6**Wdrożenie**

GITHUB ISSUE #7 · PROJECT SAFESORT PIERWSZE WYDANIE

Record operation manifest

Area: **Safety** Priority: **Medium**

`git switch -c feat/manifest-and-undo`

Każde udane połączenie `apply` pozostawia ślad — plik-manifest w formacie JSON, który opisuje, co dokładnie zostało wykonane. Bez tego dziennika polecenie `undo`, o którym będę mówić na następnej stronie, nie wiem Co dokładnie należy anulować.

apply nie tylko przesuwa pliki — rejestruje dokładnie, co zrobiła.

`.safesort/history/20260824T011640800152.json`

```
{
  "operation_id": "20260824T011640800152",
  "root": "/home/anna/Downloads",
  "timestamp": "2026-08-24T01:16:40",
  "moves": [
    {
      "source": "/home/anna/Downloads/otchet.pdf",
      "destination": "/home/anna/Downloads/Sorted/documents/
otchet.pdf",
      "completed": true,
      "error": null
    }
  ]
}
```

Ścieżka do manifestu — nieprzypadkowa: jest przewidywalnie tworzona od korzenia i identyfikatora operacji, a sam identyfikator — znacznik czasowy, sformatowany tak, aby porządek leksykograficzny zgadzał się z chronologicznym:

src/safesort/manifest.py

```
def new_operation_id() -> str:
    return datetime.now().strftime("%Y%m%dT%H%M%S%f")

def history_dir(root: Path) -> Path:
    return Path(root) / STATE_DIRNAME / "history"
```

Dlaczego właśnie ten format identyfikatora

Struna jak `20260824T011640800152` jest sortowany jako tekst dokładnie w tej samej kolejności, w jakiej operacje miały miejsce w czasie. To pozwala na `find_latest_manifest()` znaleźć tę ostatnią operację, po prostu sortując nazwy plików bez czytania zawartości każdego manifestu.

Modele `Path` w programie musi zostać przekształcone w zwykłe łańcuch znaków do zapisu JSON, format, który nie ma typu „ścieżki”

Praktyka: manifest jak zwykły JSON

interaktywny laptop bezpośrednio w przeglądarce – Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/23-15/index.html)

CHECKPOINT · ISSUE #7

```
git commit -m "feat: record operation manifest and add undo"
```

Zlał się jak Pull Request #19. Tytuł PR pokazuje, że jest to jedna gałąź dla dwóch ściśle powiązanych zadań: bez manifestu Issue #8 (undo, następna strona) po prostu nie byłoby nic do przywrócenia.

Status prawdziwego Project: **Done**

Cofnij ostatnią operację

undo przywraca pliki z dziennika — i odmawia usunięcia tego, co pojawiło się w pierwotnym miejscu.

SafeSort · Część 4 z 6**Wdrożenie**

GITHUB ISSUE #8 · PROJECT „SAFESORT — PIERWSZE WYDANIE”

Add undo

Area: **Safety** Priority: **High**

```
git switch -c feat/manifest-and-undo
```

Komenda `undo` znajduje ostatni manifest, odczytuje lista wykonanych przeniesień i zwraca pliki tam, skąd zostały wzięte. Tutaj obowiązuje ta sama zasada co w całym programie: **nic nie nadpisywać milcząco**.

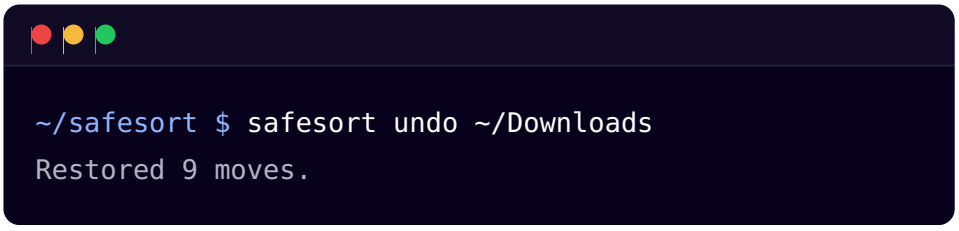
tą samą ścieżką co apply, ale w przeciwnym kierunku.

src/safesort/manifest.py

```
def undo(manifest: OperationManifest) -> UndoResult:
    restored, conflicts = [], []
    for move in manifest.moves:
        if not move.completed:
            continue
        source, destination = move.source, move.destination

        if source.exists() or source.is_symlink():
            conflicts.append(UndoConflict(source, destination,
                "a file already exists at the original
location"))
            continue

        shutil.move(str(destination), str(source))
        restored.append(CompletedMove(destination, source,
completed=True))
    return UndoResult(restored=tuple(restored),
conflicts=tuple(conflicts))
```

Cofnięcie nie przywraca tego, czego nie zostało zrobione
Pętla sprawdza `if not move.completed: continue` – Ruchy, które zawiodły podczas `apply`, nigdy nie dotykały dysku i nie ma czego cofnąć.

Najpierw standardowe skuteczne cofnięcie się:



Konflikt anulowania – rzeczywisty scenariusz

Co się stanie, jeśli uruchomimy `undo` ponownie, gdy już na swoich pierwotnych miejscach? Samo cofanie się wymaże to, co już tam jest. Zamiast ten SafeSort sprawdza oryginalną lokalizację *wcześniej* odzyskiwaniu każdego pliku i, Jeśli coś już tam jest, odmawia przywrócenia tego konkretnego pliku:

```
~/safesort $ safesort undo ~/Downloads
ERROR:safesort.manifest:Refusing to undo Sorted/other/
bigfile_b.dat -> bigfile_b.dat: a file already exists
at the original location: bigfile_b.dat
ERROR:safesort.manifest:Refusing to undo Sorted/
documents/notes.txt -> notes.txt: a file already
exists at the original location: notes.txt
... (Jeszcze 7 takich linijek)
Restored 0 moves.
9 moves could not be restored:
Sorted/other/bigfile_b.dat -> bigfile_b.dat: a file
already exists at the original location: bigfile_b.dat
... (Jeszcze 8 linijek)
```

Odmowa — a nie ciche nadpisanie. Każdy konflikt jest przetwarzany niezależnie: pliki bez konfliktu zostałyby przywrócone, nawet jeśli część listy odmówiła.

Punkt kontrolny testowy: Operacja odwrotna

tests/test_manifest.py

```
def test_undo_restores_original_location(tmp_path):
    source = tmp_path / "report.pdf"
    source.write_bytes(b"report")
    plan = build_plan(scan(tmp_path, Config()), tmp_path,
Config())
    moves = apply_plan(plan)
    manifest, _ = write_manifest(tmp_path, moves)

    result = undo(manifest)

    assert len(result.restored) == 1
    assert source.read_bytes() == b"report"
```

Praktyka: Cofnij i przywróć konflikt
Potrzebny jest dostęp do prawdziwego systemu plików – uruchamianego lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/23-16/index.html)

PUNKT KONTROLNY · ISSUE #8

```
git commit -m "feat: record operation manifest and add undo"
```

Ta sama gałąź i ten sam Pull Request #19, co zamknął Issue #7 na poprzedniej stronie.

Status prawdziwego Project: **Done**

Znajdź identyczne pliki

Kosztowne haszowanie treści odbywa się tylko wtedy, gdy faktycznie może zmienić odpowiedź – po zaznaczeniu według rozmiaru.

SafeSort · Część 4 z 6**Wdrożenie**

GITHUB ISSUE #9 · PROJECT SAFESORT PIERWSZE WYDANIE

Add duplicate detection

Area: **Duplicates** Priority: **Medium**

```
git switch -c feat/duplicate-detection
```

Drugim głównym elementem SafeSort jest wyszukiwanie plików o tej samej treści. Zadanie Wygląda to prosto: jeśli dwa pliki mają te same bajty, są duplikatami. Naiwne rozwiązanie jest Porównaj zawartość każdego pliku z zawartością każdego innego pliku – działa, ale dla tysięcy pliki oznaczają czytanie każdego pliku w kółko.

Rozmiar	Pliki
100 KB	a.jpg, b.jpg

Rozmiar	Pliki
240 KB	report.pdf (jeden plik nie prowadzi dalej)
3 MB	video.mp4, video-copy.mp4

Sprawdzane są tylko grupy dwóch lub więcej plików — pojedynczy rozmiar jednocześnie jest odrzucana, nie ma czego powielać.

Poszukiwanie duplikatów krok po kroku: każdy kolejny, droższy krok dotyczy tylko tego, co przeszło poprzedni

Plik o unikalnym rozmiarze nie może być duplikatem
Jeśli rozmiar pliku nie zgadza się z żadnym innym plikiem w przeskanowanym katalogu, na pewno nie ma duplikatu — i nie trzeba liczyć jego hasha. To sprawdzenie jest prawie darmowe (rozmiar jest już znany z `FileInfo`), i zapisuje dokładnie to, co jest najdroższe — czytanie i haszowanie zawartości plików.

Następnie jest sama funkcja skrótu, a następnie to, jak haszowanie wychodzi są grupowane w gotowe grupy duplikatów i przekazują ostatnie, bajtowe potwierdzenie.

Krótko

- Poszukiwanie duplikatów przebiega w czterech etapach: według rozmiaru, według skrótu, według pary (rozmiar, skrót), a następnie potwierdzenia bajtów.
- Skrót jest obliczany tylko dla plików, które już mają co najmniej jeden plik o tym samym rozmiarze.
- duplicates() to read-only komenda: raportuje tylko znalezione grupy, bez usuwania czegokolwiek.

SHA-256 i hash zawartości pliku

Ta sama treść zawsze daje ten sam skrót SHA-256. Odczyt bloków nie wymaga przechowywania całego pliku w pamięci jednocześnie.

SafeSort · Część 4 z 6**Wdrożenie**

Funkcja skrótu zamienia zawartość dowolnego pliku w struna o stałej długości — **skrót**. SafeSort używa **SHA-256** z modułu `hashlib`: bajty identyczne Zawsze podawaj ten sam skrót, a różne wersje gwarantują różne znaczenie Content. Hashowanie **many-to-one**: możliwych wejść jest więcej niż 256-bitowych wyników, więc różne wejścia matematycznie mogą mieć ten sam skrót. Pasujący skrót służy jako silny filtr, ale nie dowodzi równości plików. Ponieważ SafeSort przenosi prawdziwe pliki użytkownika, nie zatrzymuje się na pasowaniu skrótu — następna strona pokazuje ostatni krok, który przekształca dopasowanie hasha w potwierdzony duplikat.

src/safesort/duplicates.py

```
def sha256_stream(stream: BinaryIO, chunk_size: int = 1024 *
1024) -> str:
    digest = hashlib.sha256()
    while chunk := stream.read(chunk_size):
        digest.update(chunk)
    return digest.hexdigest()
```

Haszowanie i szyfrowanie rozwiązują różne problemy
SHA-256 zaprojektowane tak, że wyszukiwanie preobrazów w digest jest w praktyce niemożliwe obliczeniowo. Jest to właściwość inżynierska trwałości, a nie matematyczne stwierdzenie "niemożliwe do odwrócenia". Funkcja skrótu pomaga odpowiedzieć na pytanie "czy to ta sama treść" zamiast ją ukrywać. SafeSort używa SHA-256 wyłącznie do porównywania plików, a nie do ochrony danych.

Przy **efekt lawiny** mała zmiana wejścia zazwyczaj zmienia wiele bitów wyjścia; dla zachowania idealizowanego oczekuje się około połowy bitów wyjściowych. To nie oznacza, że w każdym doświadczeniu każda cyfra szesnastkowa musi się zmienić.

Dlaczego czytać plik częściowo

Cykl `while chunk := file.read(chunk_size)` czyta plik nie w całości, a blokami po megabajcie, i każdy blok od razu dodaje do streszczenia przez `digest.update()`. Jeśli funkcja odczyta plik w jednym wywołaniu `file.read()`, dla pliku o powierzchni kilku gigabajtów na program Musiałbyś przechowywać całą zawartość w pamięci RAM jednocześnie. W lekturze krok po kroku W pamięci w danym momencie znajduje się tylko jeden blok, niezależnie od rozmiaru całego pliku.

Sprawdzenie ręczne

```
>>> from pathlib import Path
>>> from safesort.duplicates import sha256_file
>>> sha256_file(Path("otchet.pdf"))
'784cc58b2286b83f67f58ffb1968ca4b80d1d0615863ad9b1ce9c3d05666f4e'
```

Praktyka: Hashuj zawartość na części
Interaktywny Laptop w przeglądarce: Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/23-17/index.html)

OFICJALNA DOKUMENTACJA

hashlib — Secure hashes and message digests
NIST FIPS 180-4 — Secure Hash Standard

Znajdź grupy duplikatów

Jedna funkcja zamienia lista pliki w grupy potwierdzonych duplikatów i nic nie usuwa.

SAFESORT · CZĘŚĆ 4 Z 6

Git i GitHub → Planowanie → Projekt →
Wdrożenie → Testy i CI → Premiera

Za pomocą funkcji skrótu z poprzedniej strony, duplikowane wyszukiwanie grupuje pliki według rozmiar, a według digestu SHA-256. Wiele implementacji na tym kończy. SafeSort idzie o krok dalej: zanim rozważy potwierdzoną grupę, porównuje Pliki w nim są wykonane bajt po bajcie.

src/safesort/duplicates.py

```
def files_equal(path_a: Path, path_b: Path, chunk_size: int =
DEFAULT_CHUNK_SIZE) -> bool:
    „Ostateczne potwierdzenie — zgodny skrót nie gwarantuje.”
    with path_a.open("rb") as file_a, path_b.open("rb") as
file_b:
        while True:
            chunk_a = file_a.read(chunk_size)
            chunk_b = file_b.read(chunk_size)
            if chunk_a != chunk_b:
                return False
            if not chunk_a:
```

```
                return True

def find_duplicates(files: list[FileInfo]) ->
list[DuplicateGroup]:
    by_size = defaultdict(list)
    for file in files:
        by_size[file.size].append(file)

    groups = []
    for size, candidates in by_size.items():
        if len(candidates) < 2:
            continue
        by_digest = defaultdict(list)
        for candidate in candidates:
            digest = sha256_file(candidate.path)
            by_digest[digest].append(candidate)
        for digest, matched in by_digest.items():
            if len(matched) < 2:
                continue
            exact_groups = []
            for candidate in matched:
                for exact_group in exact_groups:
                    if files_equal(exact_group[0].path,
candidate.path):
                        exact_group.append(candidate)
                        break
                else:
                    exact_groups.append([candidate])
            for exact_group in exact_groups:
                if len(exact_group) >= 2:
                    groups.append(DuplicateGroup(size=size,
digest=digest, files=tuple(exact_group)))
    return groups
```

Jeden bucket o tej samej wielkości i digestie pozostaje jedynie zestawem kandydatów. Algorytm porównuje kandydata z przedstawicielem każdej już znalezionej grupy: jeśli dodaje się tam, w przeciwnym razie tworzy się nowa klasa. Dlatego nawet sztucznie przywołane Kolizja SHA-256 może wygenerować wiele niezależnych grup o dokładnej zawartości.

Test-punkt kontrolny: dwa identyczne pliki

tests/test_duplicates.py

```
def test_equal_content_forms_one_group(tmp_path):
    a = tmp_path / "a.bin"
    b = tmp_path / "b.bin"
    a.write_bytes(b"same")
    b.write_bytes(b"same")

    groups = find_duplicates(scan(tmp_path, Config()))

    assert len(groups) == 1
    assert {item.path for item in groups[0].files} == {a, b}
```

files_equal() jest taka sama zasada, jak sha256_file(): czytać w blokach

Uzgadnianie bajtów po bajcie odczytuje oba pliki w megabajtach i porównuje chunk do chunku, zamiast ładować całe pliki do pamięci — to samo rozważanie co na poprzedniej stronie o `sha256_file()`. Gdy tylko jeden z elementów nie pasuje, porównanie natychmiast ustaje: nie ma potrzeby dokończać czytania reszty wyraźnie różnych plików.

Puste pliki też mogą być duplikatami

Dwa pliki zerobajtowe są identyczne: po prostu nie mają bajtów. `find_duplicates()` nie robi wyjątków w tym przypadku — naturalnie należą do tej samej grupy pod względem wielkości (`0`) i do jednej grupy na podstawie pustego digestu treści. Sprawdzimy to później osobnym testem.

Rzeczywiste polecenia `duplicates` katalogu z trzema egzemplarzami Pary meczów:

```
~/safesort $ safesort duplicates ~/Downloads
Found 3 duplicate group(s):
Group 1: 2 files, 2097289 bytes each,
sha256=44f70488694a224316549d0752fe6fdec636df12d58e85681e
```

```
Downloads/bigfile_b.dat
Downloads/bigfile_a.dat
Group 2: 2 files, 0 bytes each,
sha256=e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca4
Downloads/empty_b.bin
Downloads/empty_a.bin
Group 3: 2 files, 28 bytes each,
sha256=f3b439399a805a21e826bbd4111ca4c4615e492934f53bbb2b
Downloads/copy_of_notes.txt
Downloads/notes.txt
```

duplicates informuje o grupach i nie usuwa plików
Pierwsza wersja programu nie usuwa żadnego pliku z znalezionej grupy, a w kodzie `find_duplicates()` nie ma ani jednego wywołania, które usuwa pliki, nawet wyłączone lub wykluczone. Automatyczne usuwanie duplikatów jest celowo wykraczające poza zakres pierwszej wersji: decyzja, który z identycznych plików zachować, wymaga kontekstu, którego program nie posiada.

Praktyka: Grupuj pliki w duplikaty
Interaktywny Laptop w przeglądarce: Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/23-18/index.html)

CHECKPOINT · ISSUE #9

```
git commit -m "feat: add duplicate detection with byte-level confirmation"
```

Zlał się jak Pull Request #20. Trzy strony (23-17-23-19) książki odpowiadają jednemu PR w repozytorium: grupowanie według rozmiaru, hasza i potwierdzenia bajtów odnosi się do tego samego Issue, który jest wyjaśniony krok po kroku.

Status prawdziwego Project: **Done**

Obsługa błędów systemu plików

Specyficzne wyjątki zamiast except Exception: program nie ukrywa rzeczywistych błędów, lecz obsługuje jedynie te oczekiwane.

SafeSort · Część 4 z 6**Wdrożenie**

GITHUB ISSUE #10 · PROJECT SAFESORT PIERWSZE WYDANIE

Handle filesystem errors

Area: **Filesystem** Priority: **Medium**

Rzeczywisty system plików jest nieprzewidywalny: plik może zniknąć między skanowaniem i Read, dostęp do katalogu może zostać odrzucony, a dysk może być wadliwy. Python zgłasza takie sytuacje poprzez określone klasy wyjątków i SafeSort wykrywa dokładnie Nie wszystkie są ustawione w rzędzie.

Co się stało → jakie wyjątek

Plik zniknął między krokami	→ `FileNotFoundError`
Niewystarczające prawa dostępu	→ `PermissionError`
Inne błędy systemu plików	→ `OSError`

Wyjątek	Kiedy powstaje	Jak SafeSort reaguje
FileNotFoundError	Plik lub katalog znikał między krokami	pomija ten wpis, konty-nuuje resztę

Wyjątek	Kiedy powstaje	Jak SafeSort reaguje
PermissionError	Brak wystarczających uprawnień do odczytu lub zapisu	Pomija ten wpis, zapisuje ostrzeżenie w dzienniku
OSError	Ogólny błąd systemu plików (np. dysk przepełniony)	pomija ten wpis, kontynuuje resztę

except Exception — nie jest rozwiązaniem
Przechwycenie `except Exception: pass` pochłania nie tylko oczekiwane błędy systemu plików, ale także rzeczywiste błędy w samym programie — na przykład literówkę w nazwie zmiennej — i program działałby dalej, jakby nic się nie stało, ukrywając prawdziwy problem. SafeSort wykrywa tylko wyjątki, których faktycznie oczekuje w danym miejscu: `FileNotFoundError`, `PermissionError`, `OSError` — i nigdy nie wykrywa wyjątków bez określenia typu.

Przykład z `scanner.py` — odczyt katalogu, do którego może nie być uprawnień dostępu:

src/safesort/scanner.py

```
try:
    entries = list(directory.iterdir())
except PermissionError:
    logger.warning("Permission denied, skipping directory:
%s", directory)
    return
except FileNotFoundError:
    logger.warning("Directory vanished during scan, skipping:
%s", directory)
    return
except OSError as exc:
    logger.warning("Could not read directory %s: %s",
directory, exc)
    return
```

Każda gałąź przechwytuje swój konkretny przypadek, zapisuje wyraźną wiadomość do logu (jaki rodzaj logu jest na następnej stronie) i zwraca kontrolę - skan Reszta katalogów trwa tak, jakby nic się nie stało.

Krótko

- SafeSort przechwytuje konkretne oczekiwane wyjątki — FileNotFoundError, PermissionError, OSError — zamiast wszystkich.
- Każde przechwycenie jest oprzewiane wiadomością w logu, a nie cichą ignorancją.
- Błąd w jednym pliku lub katalogu nie powinien zatrzymywać przetwarzania pozostałych.

CHECKPOINT · ISSUE #10

```
git commit -m "feat: add directory scanner and configuration"
```

Ten Issue nie ma własnego Pull Request z frazą „Closes #10”PR #15, strona 23-08), a sam Issue #10 został zamknięty ręcznie już po sprawdzeniu, bez automatycznego powiązania. Nie każde zadanie zamyka się osobnym PR. Tak wygląda faktyczna historia repozytorium.

Status prawdziwego Project: **Done**

Dodaj log programu

Suma dla użytkownika jest drukowana bezpośrednio; Diagnostyka odbywa się osobnym kanałem – przez moduł logging.

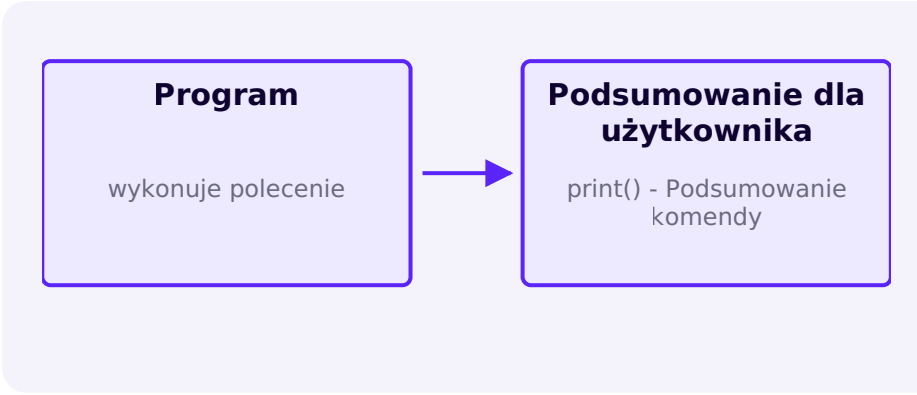
SafeSort · Część 4 z 6**Wdrożenie**

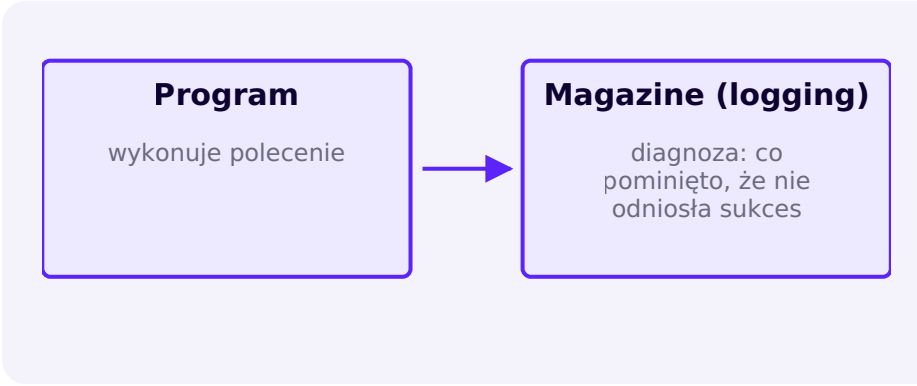
GITHUB ISSUE #11 · PROJECT SAFESORT PIERWSZE WYDANIE

Add configuration and logging

Area: **CLI** Priority: **Low**

Kod SafeSort już napotkał kilka wywołań `logger.warning(...)`. To nie to samo, co wyświetlanie na ekranie przez `print()` : Program ma dwa różne kanały wiadomości o różnych celach.





	Wyjście użytkownika (print)	Magazine (logging)
Do których jest skierowana	Dla osoby, która wywołała polecenie	Kogoś, kto rozumie, co się działo w środku
Co pokazuje	Podsumowanie poleceń – ile plików, ile ruchów	Diagnostyka: brakujące pliki, błędy dostępu
Format	Krótkie, stałe	Poziom ważności + Szczegóły

src/safesort/cli.py

```
def _configure_logging() -> None:
    logging.basicConfig(
        level=logging.WARNING,
        format="%(levelname)s:%(name)s:%(message)s",
    )
```

SafeSort używa trzech poziomów ważności, każdy dla innej sytuacji:

Poziom	Stosowany w SafeSort
INFO	Zwykłe zdarzenia: plik został przeniesiony, pominięto dowiązanie symboliczne
WARNING	Czegoś brakuje, ale program nadal działa: katalog nie jest dostępny
ERROR	Operacja zakończyła się niepowodzeniem: przeniesienie lub przywrócenie nieudane

Moduł otrzymuje logger po swojej nazwie
String `logger = logging.getLogger(__name__)` przechowuje nazwę modułu w `LogRecord.name`. Sama w sobie nie wyświetla jeszcze nazwy w terminalu: jest ona pokazana przez `%(name)s` w formatter. Bez niego nazwa pozostałaby w logu, ale użytkownik by jej nie zobaczył.

Krótko

- Wyjście użytkownika (print)) i dziennik diagnostyczny (logging) — dwa różne kanały o różnym przeznaczeniu, nie warto ich mieszać.
- Trzy poziomy ważności — INFO, WARNING, ERROR — obejmują wszystko, co musisz SafeSort bez nadmiernej konfiguracji.
- getLogger(__name__) zapisuje imię logger; formatter od %(name)s czyni je widocznym.

OFICJALNA DOKUMENTACJA

Logging HOWTO
LogRecord attributes

Ustawienia projektu

Plik ustawień jest opcjonalny: bez niego wbudowane domyślne ustawienia wchodzą w życie, a dzięki niemu można nadpisać część zachowania.

SafeSort · Część 4 z 6**Wdrożenie**
Domyślne kategorie i katalog wyników są w większości przypadków w porządku, ale Czasem warto je ustawić — na przykład rozłożone pliki nie powinny trafiać do `Sorted/`, a do katalogu o innej nazwie. Bez pliku konfiguracyjnego SafeSort używa wartości wbudowanych:

```
Wbudowane wartości domyślne (nigdzie w pliku nie zapisane — to zachowanie programu)
```

```
destination = "Sorted"
exclude = [".git", ".venv"]
```

Aby je zmienić, SafeSort szuka opcjonalnego pliku `safesort.toml` bezpośrednio w głównym katalogu skanowanym:

Sytuacja	Co się stanie
safesort.toml jest u podstaw katalogu	czytać i stosować ustawienia
safesort.toml nieobecny	używanie wbudowanych ustawień domyślnych

```
safesort.toml - Nadpisuje nazwę katalogu wyników i dodaje kategorię
```

```
destination = "Organized"
exclude = [".git", ".venv"]

[extensions]
books = [".epub"]
```

Priorytet ustawień: defaults → ustawienia użytkownika additions/ overrides → efektywny Config

Tabela `[extensions]` działa jako overlay. String `books = [".epub"]` dodaje kategorię books i nie usuwa documents lub images. Jeśli użytkownik ustawia to wyraźnie `documents = [".md"]` tylko lista documents zostanie wymienionych.

safesort.toml pozostaje wejściem konfiguracyjnym
Skaner zawsze pomija ten plik. Po `plan`, `apply` i ponownym uruchomieniu konfiguracja pozostaje na bazie głównej, więc następne uruchomienie ma te same ustawienia.

Żadna komenda SafeSort nie wymaga pliku ustawień
Jeśli `safesort.toml` nie znaleziono, używa się wbudowanych ustawień domyślnych. Program działa przewidywalnie i bez żadnej linii ustawień. Plik konfiguracyjny nadpisuje defaults, ale nie jest warunkiem wstępnym do uruchomienia.

Odczyt pliku używa `tomllib`, standard Biblioteki parsowania tylko do odczytu Python TOML:

src/safesort/config.py

```
def load_config(root: Path) -> Config:
    config_path = root / "safesort.toml"
    if not config_path.is_file():
        return Config()

    try:
        with config_path.open("rb") as handle:
            raw = tomllib.load(handle)
    except tomllib.TOMLDecodeError as exc:
        raise ConfigError(f"Could not parse config file
{config_path}: {exc}") from exc
    # ...
```

tomllib.load() akceptuje otwarty plik w trybie binarnym

Zwróć uwagę na `open("rb")`, nie `open("r")`: `tomllib` decyduje, jak zdekodować bajty pliku na tekst zgodnie ze specyfikacją TOML, więc oczekuje binarnego strumienia jako wejścia, a nie już odczytanego łańcuch znaków.

Jeśli plik ustawień istnieje, ale zawiera niepoprawny TOML, SafeSort nie próbuje zgadywać zamiaru użytkownika. Wyświetla zrozumiały błąd `ConfigError` z plikiem i powodem, zamiast odpadać trudne do odczytania prześladowanie lub ciche kontynuowanie z domyślnymi ustawieniami.

Praktyka: przeczytaj i sprawdź ustawienia TOML-

Interaktywny Laptop w przeglądarce: Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/23-19/index.html)

OFICJALNA DOKUMENTACJA

tomllib — Parse TOML files

CHECKPOINT · ISSUE #11

```
git commit -m "feat: add directory scanner and
configuration"
```

> Podobnie jak Issue #10, to zadanie nie ma własnego rejestrowania PR:, a ustawienia są w tym samym commite co skaner (PR #15), a Issue #11 zamknęliśmy ręcznie. Książka dzieli logging (23-21) i safesort.toml (ta strona) na dwa lekcje, ponieważ są to dwie różne idee. W repozytorium odpowiadają im jeden Issue i jeden commit.
>
> Status prawdziwego Project: **Done**

Pisanie pierwszych testów automatycznych

Testy działają SafeSort tylko w tymczasowym katalogu pytest – żaden z nich nie dotyka faktycznych plików użytkownika.

SAFESORT · CZĘŚĆ 5 Z 6

Git i GitHub → Planowanie → Projekt → Wdrożenie → **Testy i CI** → Premiera

Wyobraźmy sobie: ktoś przypadkowo zepsuł klasyfikator — rozszerzenie `.PDF` wielkimi literami nie jest już uznawany za dokument. Jak się o tym dowiedzieć bez Sprawdzanie ręcznie za każdym razem?

```
~/safesort $ pytest tests/test_classifier.py -q
........F.
================================== FAILURES
==================================

_________________________
test_classify_is_case_insensitive
_________________________
```

```
assert classify(".PDF", DEFAULT_EXTENSIONS) ==
"documents"
E AssertionError: assert 'other' == 'documents'

1 failed, 9 passed in 0.04s
```

RED – test wykrywał awarię przed osobą.

Przywrócić normalizację rejestrów — i ten sam test potwierdza rozwiązanie:

```
~/safesort $ pytest tests/test_classifier.py -q
..........
10 passed in 0.02s
```

GREEN – zachowanie ponownie odpowiada oczekiwaniom.

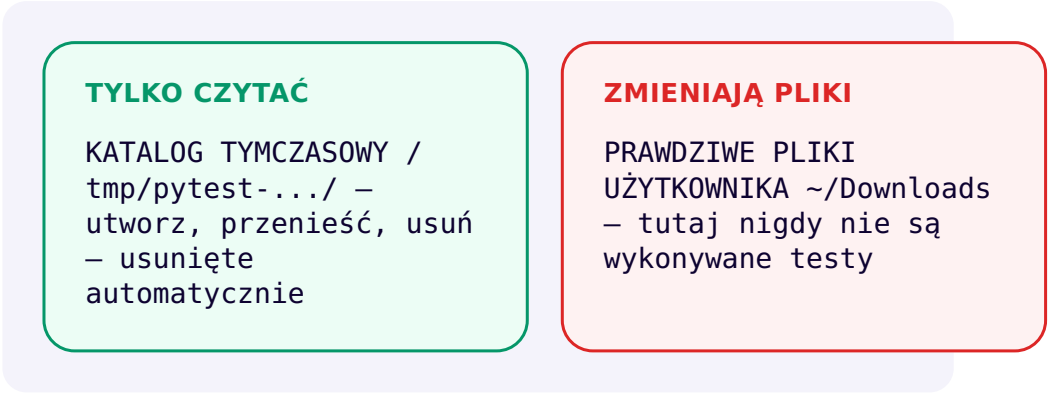
Kod, który sam sprawdza inny kod i informuje, jeśli zachowanie zmieniło się niezauważalnie dla człowieka, nazywa się **automatycznym testem**. W Python częściej całkowite użytkowanie `pytest`.

Terminal (środowisko aktywowane)

```
pip install -e ".[dev]"
pytest tests/ -v
```

Testy systemu plików nigdy nie używają rzeczywistych katalogów użytkownika

Test, który wywołuje `apply` bezpośrednio `~/Downloads`, przy pierwszym nieudanym uruchomieniu może uszczekać prawdziwe pliki. Wszystkie testy SafeSort uruchamiać w tymczasowym katalogu, który pytest sam tworzy i usuwa za pomocą wbudowanej funkcji `tmp_path` — jest przekazywany do funkcji testowej jako parametr regularny:



tests/test_classifier.py

```
from safesort.classifier import classify
from safesort.config import DEFAULT_EXTENSIONS

def test_classify_known_extension():

#Arrange: rozszerzenie i reguła klasyfikacji są już gotowe, nie
trzeba nic tworzyć
    # Act
    rezultat = classify(".pdf", DEFAULT_EXTENSIONS)
    # Assert
    assert rezultat == "documents"
```

Trzy części — **Arrange** (przygotowanie), **Act** (wywołać), **Assert** (sprawdzić) — pojawiają się prawie w każdym teście SafeSort. Funkcja `classify()` jest dobrym pierwszym testem nie bez powodu: jest czysty, nie dotyka systemu plików i nie zależy od niczego zewnętrznego. Następna strona jest przejmowana przez Bardziej złożone testy to te, które faktycznie tworzą pliki w tymczasowym katalogu.

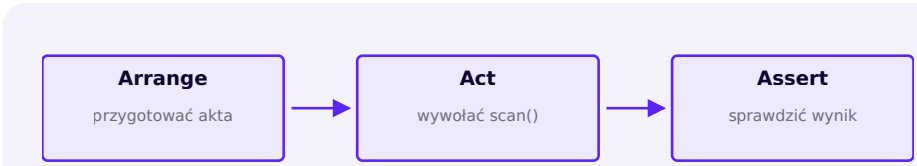
Krótko

- Test to kod, który sprawdza kod: wykonuje funkcję i porównuje wynik z oczekiwanym.
- tmp_path jest tymczasowym katalogiem, który pytest tworzy i usuwa dla każdego testu.
- Żaden SafeSort test nie ma dostępu do prawdziwego katalogu użytkowników, jak ~/Downloads.

Sprawdź skan i klasyfikację

Test na pustym katalogu i test ponownego skanowania Sorted/ — proste przypadki, które najczęściej psują się niezauważalnie.

SafeSort · Część 5 z 6**Testy i CI**



Testy dla `scan()` oraz `classify()` razem tworzą małą, w pełni kontrolowaną strukturę plików w tymczasowej a potem sprawdzić, czy skaner znalazł dokładnie te pliki, które tam zostały — Nic więcej, nic mniej.

tests/test_scanner.py

```
def test_scan_finds_nested_files(tmp_path):
    (tmp_path / "a").mkdir()
    (tmp_path / "a" / "otchet.pdf").write_text("...")
    (tmp_path / "photo.jpg").write_text("...")

    files = scan(tmp_path, Config())
    names = {f.path.name for f in files}
    assert names == {"otchet.pdf", "photo.jpg"}

def test_scan_skips_destination_directory(tmp_path):
    (tmp_path / "Sorted" / "documents").mkdir(parents=True)
    (tmp_path / "Sorted" / "documents" /
"staryj.pdf").write_text("...")

    files = scan(tmp_path, Config())
    assert files == []
```

Pusty katalog — też przypadek do sprawdzenia

Testowanie na pustym katalogu tymczasowym (bez pojedynczego pliku) wydaje się trywialne, ale to właśnie te ekstremalne przypadki najczęściej zawodzą podczas refaktoryzacji: upewnienie się, że `scan()` zwraca pusty lista zamiast zawieszać się z błędem, to prosty, ale realny test.

Drugi powyższy test to bezpośrednie sprawdzenie wymagań, które omawialiśmy wcześniej: katalogu Wynik nigdy nie jest ponownie skanowany. Bez takiego testu regresja (przypadkowy powrót starego, błędna logika) pozostawały niezauważone, dopóki ktoś nie zaczął SafeSort dwa razy z rzędu na tym samym katalogu i nie widziałem dziwnego wyniku ręcznie.

Praktyka: testujemy skaner i klasyfikator

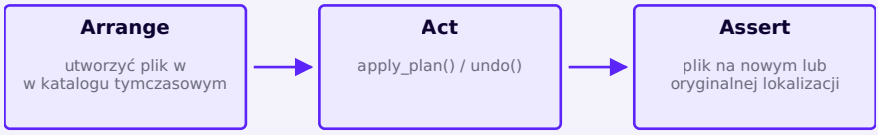
Potrzebny jest dostęp do prawdziwego systemu plików – uruchamianego lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/23-21/index.html)

Sprawdź przeprowadzkę i anulowanie

Każdy krok — przesunięcie, cofnięcie, konflikt przy cofnięciu — sprawdza osobny test, a nie jeden ogólny scenariusz.

SafeSort · Część 5 z 6**Testy i CI**



Sprawdzanie ruchu pliku jest trudniejsze niż sprawdzanie czystej funkcji: musisz stworzyć Pliki, zastosuj plan, upewnij się, że są w nowym miejscu, i dopiero wtedy Sprawdź anulowanie. Każdy krok jest sprawdzany osobnym testem, a nie jednym wielkim scenariuszem — Dzięki temu znacznie wyraźniej widać, co dokładnie jest uszkodzone, jeśli test nie przejdzie.

tests/test_executor.py

```
def test_apply_plan_moves_file_to_destination(tmp_path):
    source = tmp_path / "otchet.pdf"
    source.write_text("...")
    destination = tmp_path / "Sorted" / "documents" /
"otchet.pdf"
    plan = SortPlan(root=tmp_path,
operations=(MoveOperation(source, destination),))

    results = apply_plan(plan)

    assert results[0].completed is True
    assert not source.exists()
    assert destination.exists()
```

Test anulowania buduje plan, stosuje go, a następnie wywołuje `undo()` i weryfikuje, że plik dokładnie pochodzi z miejsca:

tests/test_manifest.py

```
def test_undo_restores_original_location(tmp_path):
    source = tmp_path / "otchet.pdf"
    source.write_text("...")
    plan = build_plan(scan(tmp_path, Config()), tmp_path,
Config())
    moves = apply_plan(plan)
    manifest_obj, _ = write_manifest(tmp_path, moves)

    result = undo(manifest_obj)

    assert source.exists()
    assert result.conflicts == ()
```

Test konfliktu anulowania — nie mniej ważny niż test udanego anulowania

Sam test „cofnij działa” `undo()` — i sprawdza, że SafeSort odmówił jego nadpisania, a nie sprawdza tylko „szczęśliwą ścieżkę”.

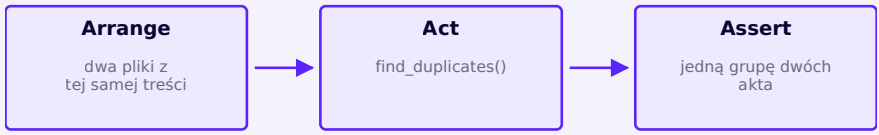
Praktyka: Testowanie ruchu i anulowanie
Potrzebny jest dostęp do prawdziwego systemu plików – uruchamianego lokalnie w VS Code, PyCharm lub Jupyter

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/23-20/index.html)

Sprawdzamy wyszukiwanie duplikatów

Puste pliki i wielomegabajtowy plik sprawdzają dwa przypadki, które łatwo przeoczyć w zwykłym teście.

SafeSort · Część 5 z 6**Testy i CI**



Testy dla `find_duplicates()` nie tylko „typowe”

tests/test_duplicates.py

```
def test_identical_content_files_are_grouped(tmp_path):
    a = tmp_path / "notes.txt"
    b = tmp_path / "copy_of_notes.txt"
    a.write_text(„ten sam tekst”)
    b.write_text(„ten sam tekst”)

    groups = find_duplicates(scan(tmp_path, Config()))

    assert len(groups) == 1
    assert len(groups[0].files) == 2

def test_empty_files_are_duplicates_of_each_other(tmp_path):
    (tmp_path / "a.txt").write_text("")
    (tmp_path / "b.txt").write_text("")

    groups = find_duplicates(scan(tmp_path, Config()))
```

```
    assert len(groups) == 1
    assert groups[0].size == 0
```

Wynik i sposób czytania są sprawdzane osobno
plik większy niż jeden blok i poprawny digest jedynie dowodzą wyniku funkcji. Taki test nie obserwuje wywołań `read()`: Implementacja mogłaby odczytać cały plik naraz i nadal zwracać ten sam SHA-256. bounded reads kontraktu jest sprawdzany przez osobny instrumented reader.

tests/test_duplicates.py

```
def test_sha256_large_file_has_correct_result(tmp_path):
    data = b"x" * (5 * 1024 * 1024)  # 5 MB — więcej niż jeden
blok odczytu
    path = tmp_path / "bolshoj_fajl.bin"
    path.write_bytes(data)

    assert sha256_file(path) ==
hashlib.sha256(data).hexdigest()
```

tests/test_duplicates.py: interaction test

```
class RecordingReader(io.BytesIO):
    def __init__(self, data: bytes) -> None:
        super().__init__(data)
        self.requested_sizes = []

    def read(self, size: int = -1) -> bytes:
        self.requested_sizes.append(size)
        return super().read(size)

def test_sha256_stream_uses_multiple_bounded_reads():
    reader = RecordingReader(b"abcdefghij")

    digest = sha256_stream(reader, chunk_size=4)

    assert digest == hashlib.sha256(b"abcdefghij").hexdigest()
    assert reader.requested_sizes == [4, 4, 4, 4]
    assert max(reader.requested_sizes) == 4
```

Pierwszym testem jest **result test (black-box)**: sprawdza, *co* wróciło. Drugi sprawdza interaction/implementation contract: zależność otrzymała kilka ograniczonych `read(4)`. Mały strumień nagrania Jest to obserwowane bez silnej infrastruktury mocking-.

Praktyka: Testy plików duplikatów i pustych
Interaktywny Laptop w przeglądarce: Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/23-22/index.html)

OFICJALNA DOKUMENTACJA

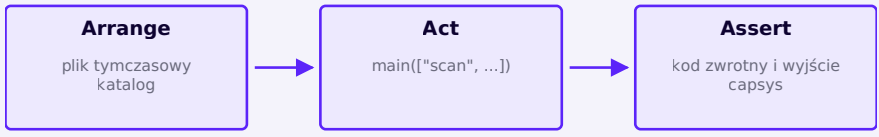
pytest — How to write and report assertions

unittest.mock — mock object library

Sprawdzamy interfejs wiersza poleceń

capsys przechwytuje tekst wydrukowany przez program. Testy sprawdzają go jak zwykły łańcuch znaków bez prawdziwego terminala.

SafeSort · Część 5 z 6**Testy i CI**



Ostatnią warstwą SafeSort, którą warto sprawdzić testami, jest samo łańcuch znaków polecenia: Czy to jest w porządku? `argparse` analizuje argumenty i sprawdza, czy kod jest poprawny Odbiór telefonu zostaje odbiór przez dzwoniącego.

tests/test_cli.py

```
def test_scan_subcommand_returns_zero_on_success(tmp_path,
capsys):
    (tmp_path / "otchet.pdf").write_text("...")

    code = main(["scan", str(tmp_path)])

    assert code == 0
    assert "Files scanned: 1" in capsys.readouterr().out

def test_missing_path_returns_nonzero(tmp_path):
    code = main(["scan", str(tmp_path / "net-takogo-
kataloga")])

    assert code != 0
```

capsys: wbudowane przechwytywanie pytest wyjścia
Funkcja `main()` drukuje wynik przez `print()` zamiast zwracać tekst bezpośrednio. Fixture `capsys` przechwytuje wszystko, co znajduje się w standardowym strumieniu wyjścia i błędów podczas testu, i pozwala sprawdzić to jako zwykły łańcuch znaków.

Weryfikacja `--help` również zasługuje na osobny test. `argparse` drukuje certyfikat i celowo go podnosi `SystemExit(0)`: Kod 0 oznacza pomyślne ukończenie, a nie błąd. Nieprawidłowe argumenty prowadzą także do `SystemExit`, ale przy kodzie niezerowym, zwykle 2.

tests/test_cli.py

```
def test_help_exits_zero():
    with pytest.raises(SystemExit) as excinfo:
        main(["--help"])
    assert excinfo.value.code == 0

def test_invalid_subcommand_exits_nonzero():
    with pytest.raises(SystemExit) as excinfo:
        main(["not-a-command"])
    assert excinfo.value.code != 0
```

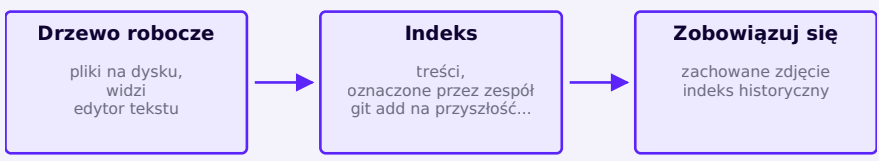
Praktyka: testy argumentów wiersza poleceń
Interaktywny Laptop w przeglądarce: Python 3.14 przez Pyodide, bez instalacji

Otwórz praktykę → (https://www.cartesianschool.org/pl/practice/23-23/index.html)

Zmiana samooceny i zaangażowanie dyscypliny

Git towarzyszył SafeSort od pierwszej linii kodu. Teraz przyzwyczaimy się do sprawdzania zmian przed każdym commitem.

SafeSort · Część 5 z 6**Testy i CI**
Git towarzyszył SafeSort od samego początku: każdy checkpoint w Części IV (strony od 23-08 do 23-22) oznaczał commit i Pull Request, zapisujący kolejną funkcję w historii. Tutaj nie otwieramy Git po raz pierwszy. Formalizujemy dyscyplinę samodzielnej kontroli przed commitem, zanim w następnym rozdziale kod przejdzie pełny cykl Issue → gałąź → Pull Request na GitHub. Między „plik zmieniony na dysku” a „zmiana zapisana w historii” są dwa kroki pośrednie, a zrozumienie różnicy między nimi oszczędza sporo zdumienia w przyszłości.



git add umieszcza migawkę bieżącej zawartości w indeksie; plik roboczy pozostaje na miejscu i może ponownie ulec zmianie

Terminal

```
git status
git diff
git add src/safesort/duplicates.py tests/test_duplicates.py
git commit -m "feat: add duplicate-file detection"
```

git diff pokazuje zawartość zmian
`git status` wymienia pliki, które zostały zmienione, ale nie pokazuje treści zmian. `git diff` pokazuje linia po linii, co dokładnie zostało dodane i co usunięte. Czytaj go przed każdym commitem, aby przypadkowo nie zatwierdzić czegoś niepotrzebnego: debugowanie `print()`, zapomniany plik z danymi osobowymi, tymczasowym kodem.

commitów booleańskich

Pojedynczy commit powinien opisywać jedną pełną, znaczącą zmianę, a nie „koniec Dzień pracy”

Dobrze	Zły
feat: add directory scanner	update
feat: add dry-run sort planning	fix
test: cover undo conflicts	stuff
docs: document safesort.toml options	final-final

Kiedy nie używać git add.
`git add .` dodaje wszystkie zmiany w bieżącym katalogu do indeksu jednocześnie. To przydatne, gdy commit naprawdę musi zawierać wszystko, ale łatwo przypadkowo wciągnąć coś nieistotnego do commitu. `git add путь/к/файлу` dodaje tylko te pliki, których potrzebujesz, i nadaje commitom więcej sensu.

Praktyka: czytamy git diff i grupujemy zmiany
Potrzebny Git- repozytorium. Wykonaj praktykę lokalnie w VS Code, PyCharm lub w terminalu

`Otwórz praktykę →` (https://www.cartesianschool.org/pl/practice/23-24/index.html)

GitHub: używać Issue, branch i Pull Request

Spójrzmy na Pull Request #20, który przeszedł test, został połączony i zamknięty Issue #9.

Część II już rozłożyła pętlę Issue → rozgałęzienie → kod → Pull Request → CI → scalanie Więcej informacji można znaleźć na stronach 23-proj-17. Teraz zastosuj ten schemat do zakończonej zmiany SafeSort: Issue #9 «Add duplicate detection». Tym razem przeanalizujemy sam Pull Request.

LOCAL GIT	GITHUB
git switch -c feat/duplicate-detection	Issue #9 opisuje to zadanie
Edycja + pytest lokalna	usunięta gałąź pojawi się po push
git add + git commit	Pull Request porównuje wątek do main
git push -u origin feat/duplicate-detection	Conversation, Commits, Checks, Files changed
po merge: git fetch / pull	review merge decyzja zamknięcie Issue

Granica przecina się na `git push`: lokalne commity stać się odległą gałęzią. Issue i Pull Request są GitHub obiektami, i `switch`, edycja, testy, staging i commit się dzieją lokalnie. Po merge lokalny Git poznaje nowy stan poprzez fetch/pull.

Issue #9 w Project

Issue #9 wcześniej dodano w Project „SafeSort: pierwsza wersja”. Na stronie 23-proj-09 już widzieliśmy, jak Issues trafiają do Project. Poniżej pokazano formularz GitHub z polami nagłówka, opisu i metadanych.

Formularz do tworzenia nowego Issue na GitHub: polu nagłówka, opisie, pasku bocznym Assignees/Labels/Type

Formularz Issue w GitHub, w którym sformułowano Issue #9 repozytorium SafeSort.

Oddział feat/duplicate-detection

Prace nad Issue #9 prowadzono w odosobnionej gałęzi. Ta sama technika była stosowana w każdy punkt kontrolny Część IV:

Terminal

```
git switch -c feat/duplicate-detection
# ...пишем код и тесты, коммитим изменения...
git push -u origin feat/duplicate-detection
```

Pull Request #20: Co uratowało GitHub

Przyjrzyjmy się danym repozytorium `Cartesian-School/safesort`:

Field PR #20	Znaczenie w rzeczywistości
Nagłówek	feat: add duplicate detection with byte-level confirmation
Gałąź → main	feat/duplicate-detection → main
Commits	1 zobowiązanie; cały Issue #9 mieści się w jednej logicznej zmianie
Files changed	2 pliki: src/safesort/duplicates.py (+113), tests/test_duplicates.py (+144)
Conversation	„Closes #9”
Checks	test: pass, 11s (workflow safesort-tests.yml)
Review	w edukacyjnym PR nie było osobnego reviewer; to fakt historyczny, a nie model do pracy zespołowej

Field PR #20	Znaczenie w rzeczywistości
Decyzja o fuzji	Merged zwykłym merge commit 01989e0c, ani squash, ani rebase

Cztery zakładki i dwa różne rodzaje sprawdzania

Tab	Na jakie pytanie odpowiada
Conversation	tego, co było omawiane, co Issue jest zamykane, jaki jest rezultat review
Commits	z jakich zapisanych kroków składa się gałąź?
Checks	przeszły automatyczne workflow i testy
Files changed	co dokładnie proponuje się dodać, zmienić lub usunąć

Reviewer czyta zmiany i może zostawić **comment**, wybierz **Approve** lub **Request changes**. CI odpowiedzi "czy zdałeś automatyczne kontrole?", a osoba odpowiada: "czy ta zmiana powinna być zaakceptowana?". Ani jedno z tych Zielony Checks nie zastępuje review inżynieryjnego, ani review nie zastępuje powtarzalnych testów.

zakładka Files changed tym repozytorium Pull Request Cartesian-School/safesort: dodanym plikom duplicates.py, Merged status

Pull Request SafeSort „Add duplicate detection with byte-level confirmation”, zamykający Issue nr 9.

Jeden PR nie zawsze odpowiada jednemu Issue
PR #20 pokazuje prosty przypadek: jedna gałąź, jeden commit, jeden Issue, automatycznie zamknięty frazą `Closes #9` podczas scalania. W historii SafeSort pojawiają się również inne warianty: jeden PR może zamknąć dwa powiązane Issue (strony 23-11 i 23-14 pokazują taki przykład), i Issues zamknięte ręcznie bez oddzielnego PR wcale (strony 23-20 i 23-22).

PR można otworzyć zanim kod będzie w pełni gotowy
Pull Request nie musi przesyłać już ukończonych prac. Można je otworzyć wcześniej, aby omówić podejście lub uzyskać tymczasowe opinie, i wyraźnie oznaczyć jako szkic. Czekanie „aż wszystko będzie idealne”

Krótko

- PR #20 pokazuje ten sam cykl Issue, gałęzi i Pull Request, który został podzielony w części II.
- Files changed, Commits, Checks i Conversation pokazują 2 pliki, 1 commit, zielony check, oraz „Closes #9”
- Jeden PR może zamknąć kilka Issues; Issue można też zamknąć ręcznie bez PR.

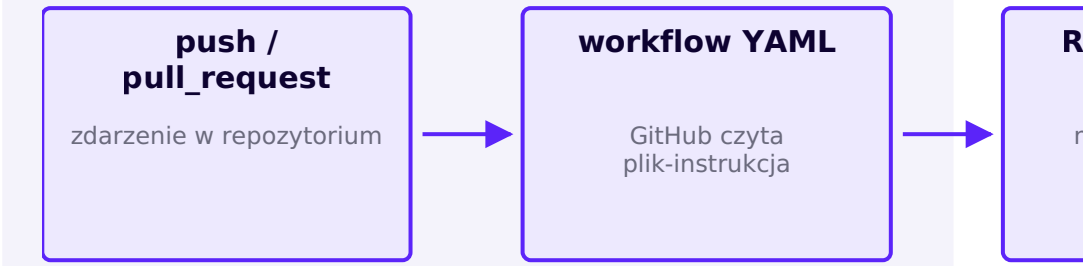
GitHub Actions: automatycznie wykonuje testy

Jeden plik workflow uruchamia testy SafeSort automatycznie przy każdej zmianie — bez ręcznej weryfikacji przed Pull Request.

SafeSort · Część 5 z 6**Testy i CI**

GITHUB **WALIDACJA (CI) TO CZWARTY KROK Z DIAGRAMU NA POPRZEDNIEJ STRONIE**

Łatwo zapomnieć o ręcznym sprawdzeniu testów przed każdym Pull Request. **GitHub Actions** jest usługą, która automatycznie wykonuje określone działania, gdy Konkretne zdarzenia w repozytorium, na przykład za każdym razem, gdy Pull Request. jest wypychany lub otwierany Dla SafeSort jest to przeprowadzanie testów.



Od zdarzenia repozytorium do zielonego lub czerwonego znaczka Pull Request

Lista przebiegów GitHub Actions repozytorium Cartesian-School/safesort: 19 rzeczywistych przebiegów; spośród dwóch czerwonych jeden odnosi się do wczesnego etapu przed CI roboczym, drugi do testu celowo zepsutego, a zaraz po nim zielony

Prawdziwa historia CI repozytorium SafeSort to 19 przebiegów: jeden wczesny czerwony przed CI roboczym oraz treningowa para celowo uszkodzonych startów i zielonych po naprawie (sekcja poniżej).

Oto plik instrukcji, który GitHub odczytuje przed każdym uruchomieniem:

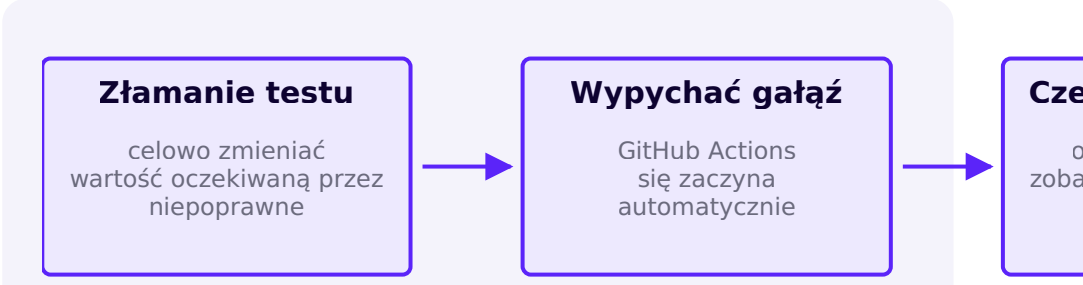
.github/workflows/safesort-tests.yml

```
name: SafeSort tests

on:
  push:
    branches: ["main"]
  pull_request:

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - name: Check out repository
        uses: actions/checkout@v7

      - name: Set up Python
        uses: actions/setup-python@v7
        with:
          python-version: "3.14"

      - name: Install SafeSort with dev dependencies
        run: pip install -e ".[dev]"

      - name: Run tests
        run: pytest tests/
```

on: nieograniczone, ponieważ cały repozytorium jest SafeSort
Tu nie ma klucza `paths` ograniczający działanie zmodyfikowanych plików: w osobnym repozytorium `safesort` jakiekolwiek Pull Request czy push w taki czy inny sposób dotyczy samego pakietu. Takie ograniczenie jest potrzebne tylko w repozytorium kursów ogólnych, gdzie SafeSort jest tylko jednym z wielu podkatalogów i nie ma potrzeby restartowania testów z powodu zmian gdzie indziej.

Kontrolowane sprawdzenie: celowo zepsyty test

Warto choć raz zobaczyć, jak wygląda czerwony (nieudany) test — i jak wygląda napraw to, zanim po raz pierwszy spotkasz się z tym w rzeczywistej sytuacji. W historii premiery To jest czerwony kółko powyżej: tymczasowe zatwierdzenie celowo zwracało wielkość liter porównanie rozszerzeń w klasyfikatorze (ten sam błąd co w sekcji 23-23 jako RED/GREEN przykład), został następnie cofnięty przez następne zatwierdzenie main nie przez sekundę pozostał w stanie złamanym.



Złamanie testu raz celowo to najszybszy sposób, by nauczyć się czytać magazyn GitHub Actions

Główna gałąź nie powinna pozostać czerwona
Celem tego ćwiczenia jest natychmiastowe sprawdzenie nieudanego testu w kontrolowanym środowisku i jego natychmiastowe naprawienie, a nie zostawienie go `main` w uszkodzonym stanie. Rzeczywista, czerwona weryfikacja na głównej gałęzi oznacza, że ktoś inny, kto pobierze kod teraz, otrzyma nie działającą wersję.

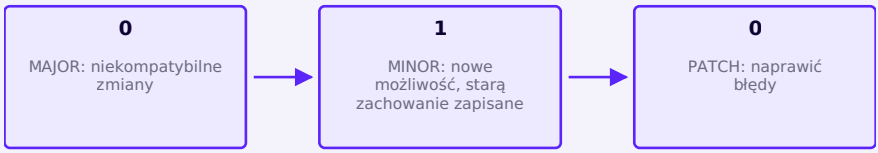
Krótko

- GitHub Actions automatycznie uruchamia określone działania — podczas naciskania lub otwierania Pull Request.
- W osobnym repozytorium workflow uruchamia się bez ograniczeń dla paths — jakakolwiek zmiana tutaj w ten czy inny sposób dotyczy SafeSort.
- Specjalnie popsuty, a następnie naprawiony test — szybki sposób, aby nauczyć się czytać dziennik kontroli.

Dokumentacja, wersja i pierwsze wydanie

MAJOR.MINOR.PATCH, CHANGELOG.md, Git tag i GitHub Release dają projektowi punkt, do którego można wrócić i do którego można się odwołać.

Projekt gotowy, sprawdzony testami i podłączony do automatycznej weryfikacji. Skończyliśmy pierwszą użyteczną wersję SafeSort.. Teraz potrzebuje numeru, aby odróżnić ją od przyszłych zmian. Ten numer już zapisaliśmy w `pyproject.toml` też w części III, gdzie było to tylko techniczne Pole. Teraz zobaczmy, co to oznacza.



W 0.1.0 każda z trzech liczb odpowiada za inny rodzaj zmiany

Wersowanie semantyczne

Ten schemat nazywa się **semantyczne wersjonowanie** (Semantic Versioning, SemVer), czyli umowa dotycząca zapisu numeru wersji jako `MAJOR.MINOR.PATCH`:

Część pokoju	Zmiany podczas
MAJOR	niekompatybilne zmiany; stary sposób używania przestaje działać
MINOR	dodano nową funkcję, zachowano stare zachowanie
PATCH	błąd został naprawiony, zachowanie zasadniczo się nie zmieniło

> **Umowa, a nie prawo fizyki**
> wersjonowanie semantyczne ustanawia powszechnie przyjętą konwencję, ale nie jest wbudowana w narzędzia: nic technicznie nie stoi na przeszkodzie, by złamać jej znaczenie. Zaletą z tego jest tylko tyle, na ile sam projekt konsekwentnie go przestrzega. To jest wskazówka dla tych, którzy instalują pakiet i chcą zrozumieć, czego mogą się spodziewać po nowej wersji.

Pierwsza wersja SafeSort ma numer `0.1.0`. Wersja mniejsza niż 1 tradycyjnie oznacza „interfejs może jeszcze ulec zmianie bez dodatkowej zgody”.

Co się zmieniło w tej wersji: CHANGELOG

CHANGELOG.md

```
## [0.1.0]

### Added

- scan, plan, apply, duplicates, undo commands

### Safety

- dry-run by default (scan/plan/duplicates never modify files)
- no automatic duplicate deletion
- no silent overwrite of existing files
```

CHANGELOG opisuje tylko rzeczywiste wersje
CHANGELOG nie powinno pojawiać się wcześniejszych wersji, które nie istniały. Opisuje to wprowadzone zmiany zaczynające się od pierwszej wersji projektu.

Zbieraj wheel i sdist

Wydanie zaczyna się od drzewa źródłowego, ale użytkownik otrzymuje distribution artifacts. Drużyna `python -m build` tworzy oba standardowe formaty:

Terminal (główna repozytorium SafeSort)

```
python -m pip install --upgrade build
python -m build
```

wheel zawiera gotowe archiwum Python distribution; sdist zawiera źródła do montażu

Artefakt	Cel
wheel (.whl)	gotowe archiwum distribution: instalacja zwykle nie uruchamia kompilacji projektu
sdist (.tar.gz)	archiwum źródłowe i metadane, z którego narzędzie może zbierać wheel

Smoke test w czystym środowisku

Editable install weryfikuje projekt, ale nie dowodzi, że zbudowany wheel zawiera niezbędne pliki i console script. Dlatego zainstaluj artifact w nowym środowisku:

POSIX shell

```
python -m venv .release-smoke
source .release-smoke/bin/activate
python -m pip install dist/safesort-0.1.0-py3-none-any.whl
python -c "import safesort; print(safesort.__file__)"
safesort --help
deactivate
```

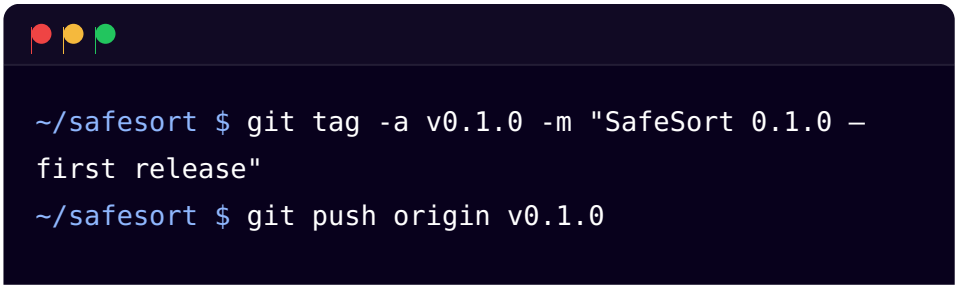
W Windows aktywacja odbywa się poleceniem `.release-smoke\Scripts\activate`. Tak samo zainstalowano distribution, a nie źródła `src/`.

GITHUB GITHUB RELEASE JEST ZBUDOWANY NA

Git tag i GitHub Release: różne obiekty

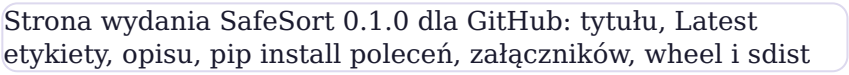
Tag (tag) przechowuje odniesienie Git-a do konkretnego commitu i zazwyczaj odpowiada numerowi wersji. Technicznie rzecz biorąc, tag można przenieść lub usunąć, ale Opublikowane tagi release- są uznawane za niezmienne w projekcie polityki. **GitHub Release** służy jako osobny obiekt GitHub utworzony wokół tagu. Ma Strona z opisem i dołączonymi tagami artifacts. Push sama nie tworzy Release.

Tag wersji dotyczy całego repozytorium w całości, nie tylko w jednej jego części. Dlatego SafeSort żyje w swoim własnym repozytorium `Cartesian-School/safesort`, nie tylko w podkatalogu kursów: `git tag v0.1.0` tutaj jednoznacznie oznacza „wersja 0.1.0 SafeSort”, bez dwuznaczności.



```
To https://github.com/Cartesian-School/safesort.git
* [new tag] v0.1.0 -> v0.1.0
```

Następnie w interfejsie webowym GitHub otwórz **Releases → Draft a new release**, wybierz istniejący `v0.1.0`, dodaj fragment CHANGELOG i załącz oba pliki z `dist/`. Następnie tag ma czytelną stronę wydania oraz dostępne do pobrania artifacts.



Wydanie SafeSort 0.1.0: Tagi, notatki do patcha oraz pakiety wheel i sdist.

Wydanie jest tworzone dopiero po tym, jak wszystko inne jest gotowe

Tag i release pojawiły się w ostatnim kroku, gdy CI był już zielony, CHANGELOG.md opisywały prawdziwe zmiany, editable install wheel/sdist budowy i komendę `safesort --help` zostały ponownie sprawdzone w czystym środowisku. Bez tych sprawdzeń numer wersji niczego nie gwarantuje.

Publikacja do PyPI pozostaje poza wersją 0.1.0

Instalacja przez `pip install git+https://github.com/Cartesian-School/safesort.git@v0.1.0` wystarczające do tworzenia i użytku osobistego. Publikuj pakiet na PyPI, aby mógł zostać zainstalowany przez zespół `pip install safesort` bez odwołania do repozytorium, pozostaje osobnym tematem z własnymi wymaganiami dotyczącymi konta i publikacji. Wersja 0.1.0 świadomie tego nie dotyczy.

Krótko

- MAJOR.MINOR.PATCH określa konwencję numeru wersji, a nie regułę wbudowaną w narzędzia.
- python-m build tworzy wheel i sdist; czyste środowisko sprawdza instalację wheel i console script.
- Git tag i GitHub Release reprezentują różne obiekty; Release jest tworzony wokół tagu i przechowuje artifacts.

OFICJALNA DOKUMENTACJA

Packaging flow
Packaging Python Projects — Generating distribution archives
Semantic Versioning 2.0.0
Managing releases in a repository

Rezultaty: pełna ścieżka projektu od pomysłu do premiery

Od pomysłu po pierwsze wydanie — cała ścieżka jest raz w całości, na prawdziwym projekcie.

SafeSort · Część 6 z 6**Premiera**

Pełna ścieżka projektu

Na początku tego rozdziału SafeSort był tylko pomysłem. Teraz to prawdziwe repozytorium na GitHub — Cartesian-School/safesort – z 14 zamkniętymi Issues, realnymi Pull Request, zielonymi CI weryfikacji i opublikowanymi Wydanie 0.1.0. Oto cała podróż, z tym, co pojawiło się na każdym etapie:

Idea
wymagania: co program robi, a czego nie robi

Git i GitHub
instalacja, uwierzytelnianie, prawdziwe repozytorium SafeSort

GitHub Project
14 Issues — problem jest formułowany dla każdej części

README.md, pyproject.toml
pierwszym zatwierdzeniu repozytorium dostępna staje się safesort poleceń

Scanner
scanner.py — Issue №1, PR №15

Klasyfikator
classifier.py — Issue №2, PR №16

Plan i kolizje
planner.py — Issues nr 3 i nr 6 razem, PR nr 17

Apply
executor.py — Issue №5, PR №18

Manifest i undo
manifest.py — Issues nr 7 i nr 8 razem, PR nr 19

Duplikaty
duplicates.py to potwierdzenie rozmiaru → SHA-256 → bajtów, Issue nr 9 PR nr 20

Drużyna łańcuch znaków
argparse, pięć podzespołów – Issue nr 4, PR nr 21 (ostatni, gdy wszystko inne już było na

Robaki, config, testy, CI
Issues nr 10–13 jest częścią implementacji odpowiadających modułów, bez osobnego PR

Wersja 0.1.0
CHANGELOG.md, editable install, wheel i sdist sprawdzone ponownie — Issue nr 14

Od pomysłu do publikacji – dokładna historia: 9 z 14 Issues zamknięto po 7 Pull Request (dwa PR zamknęły się dwa Issue), pozostałe pięć bez osobnego PR

TO, CO JUŻ TAM JEST

Git i GitHub są skonfigurowane

Repozytorium Real SafeSort

GitHub Project i 14 Issues

Zainstalowany pakiet

Działająca komenda łańcuch znaków

Bezpieczne podróże z anulowaniem

Znajdź duplikaty

Co już SafeSort zrobić

Komenda	Co robi	Zmienia pliki?
scan	znajduje i liczy pliki według kategorii	nie
plan	pokazuje plan ruchów	nie
duplicates	znajduje pliki z tą samą zawartością	nie
apply	wykonuje transfery z planu	tak, tylko na wyraźne polecenie
undo	cofa ostatnią wykonaną operację	tak, tylko odzyskiwanie

co dalej

Wersja 0.1.0 celowo nie obejmuje wszystkiego, co możliwe: pierwsza część tego rozdziału to Określił granice. Dalszy rozwój SafeSort wykracza poza zakres tego rozdziału, ale niektóre Instrukcje naturalnie kontynuują to, co już zostało zrobione: publikacja pakietu w PyPI, klasyfikacja nie tylko przez rozszerzenie, ale także przy dokładnym sprawdzeniu zawartości pliku, interfejs dla Ustawienia inne niż ręczna edycja pliku TOML-.

Aneks do tego rozdziału powtarza tę samą ścieżkę, od problemu do Pull Request, Sześć razy, już samodzielnie, przy sześciu małych projektach: Dodatkowa praktyka: sześć mini-projektów dla GitHub.

Czego nauczyliśmy się w tym rozdziale

- Przed napisaniem kodu wymagania projektu opisują nie tylko to, co program robi, ale także to, czego świadomie nie robi.
- Podział na read-only planowanie i dwie jawne operacje, bezpośrednią apply i odwrotną undo, chroni przed przypadkowymi zmianami plików użytkownika.
- Wyszukiwanie duplikatów wykonuje kosztowną operację — haszowanie — tylko wtedy, gdy faktycznie może zmienić odpowiedź: po wybraniu plików według rozmiaru.
- Testy automatyczne działają tylko na katalogach tymczasowych – żaden test nie powinien dotykać rzeczywistych plików użytkownika.
- Git i GitHub — część developmentu od samego początku, a nie końcowy krok: Issue formułuje zadanie, gałąź izoluje pracę, Pull Request otwiera ją do weryfikacji, GitHub Actions sprawdza testy automatycznie.

ROZDZIAŁ 24 · STR. 1665

Co dalej? Roadmap Python-dewelopera

Mapa rozwoju zawodowego po kursie: Python podstawowe, praktyka inżynierska, specjalizacje, portfolio oraz samodzielne studia.

24.15 Twój następny krok

Czego osiągnąłeś!

Przeszedłeś od Python pierwszego zespołu do projektu z testami, Pull Request, CI, pakietem i wydaniem.

Od pierwszej linii do cyklu inżynieryjnego

Na początku kursu program składał się z jednej komendy `print("Привет!")`. Teraz wiesz, jak przełożyć zadanie na dane, funkcje, moduły, interfejs i testy. To jest ważniejsze niż liczba zapamiętanych elementów Metody: Nauczyłeś się budować program częściowo i sprawdzać każdą część.

- **Wyjście i zmienne**
  Program przechowuje stan i pokazuje wyniki.
- **Warunki i cykle**
  Code podejmuje decyzje i powtarza działania.
- **Kolekcje i funkcje**
  dane są zorganizowane, logika dzieli się na operacje.
- **Klasy, Pliki, Wyjątki**
  Pojawiły się modele, przechowywanie i wyraźne odrzucenia.
- **Turtle, Tkinter, Pygame, Flask**
  Ta sama podstawa Python działa w różnych interfejsach.
- **Git, GitHub, Issues, Projects, PR, testy, CI**
  Zmiana przechodzi przez powtarzalny cykl inżynieryjny.
- **Pakiet i wydanie**
  Projekt można złożyć, zainstalować i udostępnić innej osobie.

Ścieżka kursu: od pierwszego zespołu do projektu, który można przetestować i udostępnić.

Co dokładnie możesz teraz zrobić

Kompetencje po Rozdziały 23

MODEL

Wybierz typy danych
Sformułować inwarianty
Oddzielny stan i zachowanie

IMPLEMENTOWAĆ

Funkcje i klasy zapisu
Praca z plikami
Obsługa spodziewanych błędów

BUDOWAĆ INTERFEJS

Turtle modeli wizualnych
Tkinter for GUI
Pygame pętli rozgrywki
Flask dla HTTP-aplikacji

SPRAWDŹ

Defect Reprodukcji

```
Napisz test
Sprawdź przypadki graniczne
```

WSPÓŁPRACOWAĆ

```
Issue i gałąź
znaczące commit
Pull Request i review
```

DO ZAOPATRZENIA

```
pyproject.toml
CI
Zbuduj pakiet
wersjonowana wersja
```

kompetencje potwierdza się przez działanie, znajomość tego terminu nie wystarcza.

Jak SafeSort połączyć wszystkie etapy pracy

SafeSort w Rozdział 23 było narzędziem szkoleniowym CLI-automatyzacji. To Wartość nie polega na rywalizacji z menedżerem plików. Na nim przeszedłeś przez kompletne Proces: wyjaśnienie zachowań, oddzielenie planu od realizacji, ograniczenie niebezpiecznych operacji, Dodałem testy, sformalizowałem zmianę w gałęzi, przeprowadziłem ją przez Pull Request i CI, a potem zbudował wydanie. Efekt można sprawdzić w następującym projekcie: ten sam proces powinien Pracuj bez instrukcji krok po kroku.

Co oznacza ukończenie kursu

Kurs wprowadzający nie obejmuje całego Python i nie przygotowuje do żadnej roli zawodowej. Teraz masz fundament i działający sposób na jej rozwijanie poprzez projekty, kontrole i analizę błędów.

Samobadanie bez samooszukiwania

Weź mały, nieznany projekt. Czy możesz opisać wejścia i wyjścia, podkreślić Czysta logika, przewiduj odmowę, napisz wiele testów, zapisz zmianę do Git i wyjaśnić rozwiązanie? Jeśli część pętli nadal wymaga podpowiednienia, nie unieważnia to Rezultat. To po prostu stał się konkretnym celem na następny miesiąc.

Twój wynik

- Czytasz zadanie jak system danych, reguł, interfejsów i awarii.
- Możesz doprowadzić mały projekt od pomysłu do weryfikowalnej zmiany.
- Znasz granice obecnej wiedzy i wiesz, jak zamienić je w plan pracy.

OFICJALNA DOKUMENTACJA

Python 3.14 documentation
Git reference
GitHub pull requests documentation
GitHub Actions documentation
Python Packaging User Guide

Pierwsze 30 dni po książce

Cztery tygodnie konsolidacji: po jednym ukończonym wyniku, z testami i analizą rozwiązań.

Plan czterotygodniowy

Ten plan jest zaprojektowany do regularnej praktyki po przeczytaniu książki. Nie twierdzi, że taki jest uniwersalne tempo. Jeśli istnieje Dwie krótkie sesje tygodniowo, rozciągnij tydzień planu do dwóch. Nie skracaj czeku i Analiza błędów dla celów kalendarza.

Week	Focus	Wyniki tygodnia
1	Powtarzaj podstawy bez kopiowania: funkcje, kolekcje, pliki, wyjątki.	One CLI- mini-projekt z README i testami.
2	Przebudować znany pomysł innym narzędziem.	Jeden wizualny dziennik projektowy i rozwiązań.
3	Dodać sprawdzenia projektu: gałąź, issue, lint, testy, CI.	Pull Request w swoim własnym repozytorium.
4	Wybierz jedną specjalizację i przeprowadź krótkie badanie.	Prototyp, retrospektywa i plan na kolejne 60 dni.

Bank Projektu

Wybierz jeden projekt na tydzień, nie wszystkie naraz. Poniżej nie ma gotowych rozwiązań. Minimalne Wersja wyznacza granicę zakończenia, a rozszerzenia dostarczają materiału na kolejną pętlę.

Konwertor walut

Pomysł: Przelicz kwotę między wybranymi walutami według wyraźnie określonego zestawu kursów.

Wersja minimalna: CLI lub Tkinter, sprawdzenie liczby i kodu waluty, kursy w JSON, zrozumiały błąd przy braku pary.

Rozszerzenia: Pobieranie kursu z HTTP API, pamięć podręczna z czasem aktualizacji, Decimal i historią operacji.

Co naprawia: słowniki, funkcje, JSON, walidacja, oddzielenie obliczeń od interfejsu

Wyścig w Pygame

Pomysł: Gracz zmienia pasy i omija przeszkód, przemieszczając się od góry do dołu.

Wersja minimalna: pętla gry, sterowanie, kolizje, liczenie i restart bez globalnego chaosu stanów.

Rozszerzenia: delta time, rosnąca trudność, dźwięki, tabela wyników w pliku.

Co naprawia: zdarzenia, współrzędne, stan gry, klasy, debugowanie ramek

Wariacje wzorów Turtle

Pomysł: Zbadać, jak kąt, krok, kolor i powtórzenie zmieniają wzór geometryczny.

Wersja minimalna: Trzy parametryzowane wzory, wprowadzanie parametrów i zapis wybranej konfiguracji.

Rozszerzenia: generator palet, symetria, eksport obrazu, porównanie czasu tworzenia.

Co naprawia: cykle, funkcje, parametry, współrzędne, myślenie eksperymentalne

Wąż na Pygame

Pomysł: Powtórz mechanikę projektu Turtle w innym modelu zdarzenia i grafii.

Wersja minimalna: Siatka, kierunek, wzrost, pożywienie, kolizja z granicą i własnym ciałem.

Rozszerzenia: pauzy, poziomów, przeszkód, sprawdzonej logiki ruchu poza Pygame.

Co naprawia: dekompozycja, kolekcje, stan, transfer modelu między ramami

Gra Memory

Pomysł: Otwieraj pary kart i zapisuj dopasowane pary.

Wersja minimalna: Przetasowanej siatki, nie więcej niż dwie otwarte karty, opóźnienie przed zamknięciem, koniec gry.

Rozszerzenia: różne rozmiary pól, timer, liczbę ruchów, powtarzalne tasowanie ciasta.

Co naprawia: dane dwuwymiarowe, stany końcowe, zdarzenia myszy, testowana logika

Gra na unik

Pomysł: Postać unika spadających przedmiotów, a trudność stopniowo rośnie.

Wersja minimalna: Ruch, Generowanie Przeszkód, Kolizja, Liczenie czasu i Koniec ekranu.

Rozszerzenia: różne typy przeszkód, bonusy, pauza, profil wydajności.

Co naprawia: pętli gry, losowości, kolizji, kontroli tempa

Rytm pracy

Jeden cykl samodzielnej praktyki.

Jeśli utkniesz, zmniejsz zadanie do następnego obserwowalnego zachowania. Na przykład, najpierw pokaż nieruchomy samochodzik, potem obsłuż jeden klawisz, a następnie dodaj jedną przeszkodę. Taki podział zachowuje związek przyczynowo-skutkowy między kodem a wynikiem.

OFICJALNA DOKUMENTACJA

Python 3.14 documentation
pytest documentation
Pygame documentation

Professional Python Core

Idiomy językowe, model iteracji, typy, zasoby, pakiety i współbieżność jako wspólna podstawa Python- programisty.

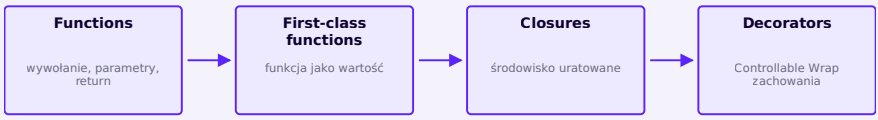
Warstwa między składnią a specjalizacją

Professional Python Core jest potrzebny we wszystkich kierunkach. Nie jest przekazywany w jednym kierunku i nie są badane przez poszczególne sztuczki. Każda jednostka ma założenie, działające zadanie i obserwowalne kryterium zrozumienia.

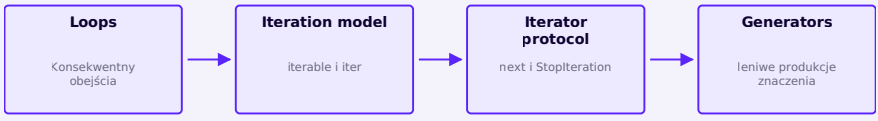
Blok	Po co i kiedy	Wymagania wstępne i kryteria
Comprehensions, rozpakowywanie się, krojenie	Kompaktowo transformuj kolekcje i jawnie rozkładaj wartości.	Po listach i pętlach. Możesz wybrać czytelną formę i nie ukryć złożonej logiki w jednej linii.
Iterable, iterator, generator	Przetwarzaj strumień wartości bez konieczności przechowywania całego wyniku.	Po funkcjach i pętlach. Wyjaśniasz różnicę między obiektem, iteratorem a leniwym generatorem.
Zamykania i dekoratory	Dostosowanie zachowania funkcji i oddzielna logika end-to-end.	After funkcje jako obiekty. Zapisujesz sygnaturę i rozumiesz kolejność wywołań.
Exceptions i context managers	Zagwarantuj uwolnienie zasobu i włącz odmowę do umowy.	Po try/except i plikach. Rozróżniasz oczekiwane wyjątek, cleanup i tłumienie błędów.
dataclasses i enums	Opisz dane i skończony zestaw stanów bez zbędnego kodu szablonowego.	Po klasach. Inwarianty nie są rozproszone w zestawie literałów tekstowych.

Blok	Po co i kiedy	Wymagania wstępne i kryteria
Typing, aliases, generics, Protocol	Udostępnić interfejsy czytelnikowi i analizatorowi statycznemu.	Po funkcjach i klasach. Pamiętaj, że same adnotacje nie są sprawdzane Python w czasie działania.

Uzależnienia w Core



Decorators opierają się na funkcjach jako obiektach i domknięciach; badanie ich jako osobnego triku składniowego nie wystarcza.



Generators stają się zrozumiałe po protokole iteratora, a nie przed nim.

Standardowa Biblioteka jako narzędzie robocze

Biblioteka według celu, nie alfabetycznie

KOLEKCJE

collections
collections.abc
odpowiednia struktura zamiast losowego list

CZAS

datetime
timezone-aware znaczenia
jawny format wymiany

ŚCIEŻKI I DANE

pathlib
json
kodowania UTF-8
schemat i walidacja

TEKST

re
szablon tylko tam, gdzie jest prostszy niż
jawne parsowanie
testy granicznych wierszy

ŹRÓDŁA

contextlib
with
deterministyczne zwolnienie

DIAGNOZA

```
logging
poziomy i kontekst
nie zastępować testów logami
```

Ucz się modułu, gdy rozwiązuje on prawdziwy problem projektu.

Moduły, środowiska i sposób realizacji

Kolejny poziom modułowości zaczyna się od wyraźnych importów i pakietu, który interfejs publiczny. Środowisko wirtualne izoluje zależności projektów. Opis w `pyproject.toml` zapisuje metadane i narzędzia budowania. Wersje zależności muszą być odtwarzalne w takim stopniu, w jakim wymaga tego projekt; ślepe przypięcie każdej wersji tranzytywnej i całkowity brak ograniczeń stwarzają różne ryzyka.

Asynchronia i konkurencyjność

Model	Odpowiednie	Nie rozwiązuje automatycznie
asyncio	Dużo operacji oczekiwania z bibliotekami obsługującymi async.	CPU-bound obliczenia i blokowanie wywołań w event loop.
threads	Blokowanie I/O i bibliotek za pomocą normalnych synchronicznych API.	Uproszczenie ogólnego stanu zmiennego.
processes	Izolacja i zadania równoległe CPU-bound.	Darmowe udostępnianie big data i natychmiastowy start.

Najpierw zmierz wzór obciążenia. Następnie wybierz model i wyraźnie zdefiniuj cancel, timeouts, zakończenie, próba ponownego wykonania operacji oraz wspólne granice stanów. Samo słowo `async` nie przyspiesza programu.

Procedura uczenia się

Zacznij od idiomów kolekcji i modelu iteracyjnego. Następnie pogłębiaj wyjątki, menedżery kontekstu, struktury danych i typing.. Pakiety i środowiska ucz się na realnym projekcie. Współbieżność dodawaj po pojawieniu się mierzalnego zadania.

OFICJALNA DOKUMENTACJA

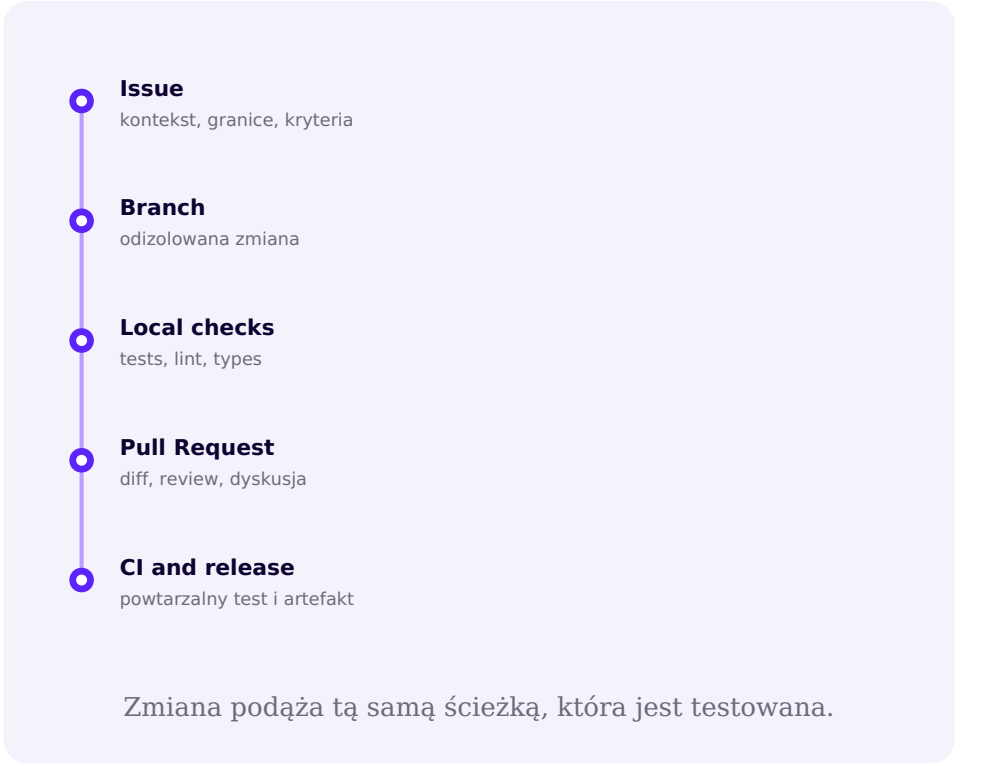
Python data model
typing: Support for type hints
contextlib: Utilities for with-statement contexts
The Python Standard Library
venv: Creation of virtual environments
Python Packaging User Guide
asyncio: Asynchronous I/O
Concurrent Execution

Software Engineering

Git, review, testowanie, analiza statyczna, diagnostyka, pakowanie i dostarczanie wchodzą w jeden proces roboczy.

Jak bezpiecznie zmienić kod

W skrypcie edukacyjnym czasami wystarczy uzyskać poprawny wynik tylko raz. W projekcie trzeba także bezpiecznie zmieniać kod dalej. Przepływ pracy łączy wymagania, realizację, testowanie i dostawę. Narzędzia mogą się zmieniać. Wymagania wobec procesu pozostają: można śledzić historię zmian, można zobaczyć awarię i powtórzyć kompilację.



Git i review głębsze niż podstawowe komendy

Naucz się czytać historię i diff, rozgałęziające, merge, bezpieczne rebase lokalnej gałęzi, revert, tagi i rozwiązywanie konfliktów. Celem nie jest zbiór komend. Musisz zrozumieć Jaki rodzaj grafu zatwierdzeń zostanie uzyskany i jak odzyskać błąd. Review sprawdza kontrakt, przypadki brzegowe, testy, czytelność i ryzyko operacyjne; styl kodu jest lepszy Deleguj kontrole automatyczną.

Strategia testowa

Oznacza pytest	Co modeluje	Ryzyko nadużycia
fixtures	Jawne przygotowanie stanu i zasobów.	Ukryty złożony fixture sprawia, że test jest niezrozumiały.

Oznacza pytest	Co modeluje	Ryzyko nadużycia
parametrize	Jeden kontrakt dla zestawu wejść i granic.	Duża tabela może ukrywać różne przyczyny awarii.
mock / monkeypatch	Kontrolowana granica: czas, sieć, proces, API.	Mock szczegóły wewnętrzne łączą test z implementacją.
coverage	Które rzędy lub gałęzie zostały wykonane.	Wysoki odsetek nie dowodzi jakości zdań.

Buduj piramidę przez ryzyko: dużo szybkich testów czystej logiki, mniej testów całkowania testy rzeczywistych granic i kilka scenariuszy end-to-end. Test musi mieć powód, Obserwowalna aprobata i kontrolowane środowisko.

Automatyczne kontrole i diagnostyka

Ruff może wykonywać lint i formatowanie, a mypy sprawdza spójność adnotacje. Wybierz zasady, zapisz je do `pyproject.toml` i uruchom równo lokalnie i w CI. W przypadku defektu zbieraj minimum odtworz, przeczytaj traceback, ustaw punkt przerwy lub dodaj Ustrukturyzowany log. Profiler jest potrzebny po zmierzeniu problemu, a nie dla Przedwczesna optymalizacja.

Pakiet, wersja i dostawa

Projekt potrzebuje jasnej struktury, metadanych, zależności, build backend oraz weryfikacji instalacji zbudowanego wheel w czystym środowisku. Przez PyPI publikują pakiety Python-, ale nie każdej aplikacji potrzeba takiej publikacji. Semantic Versioning przydaje się, gdy projekt deklaruje publiczny API i może konsekwentnie określić kompatybilne i niekompatybilne zmiany.

warstwa systemowa

Wiedza wokół kodu Python-

PROTOKOŁY

HTTP semantics
status codes
timeouts and retries
idempotentność

DANE

SQL
schemat
indeksy
transakcji
migracje

ŚRODOWISKO

Linux
shell
processes
permissions
environment variables

BEZPIECZEŃSTWO

walidacja wejścia
sekrety poza repozytorium
minimalne przywileje
Aktualizacja zależności

CI/CD

powtarzalne polecenia
chronione sekrety
rozdzielenie build i deploy
rollback

CONTAINERS

image
runtime config
health check
logs
nie mylić pojemnika z VM

W rzeczywistym projekcie kod Python-code współdziała z protokołami, danymi, systemem operacyjnym i środowiskiem.

OFICJALNA DOKUMENTACJA

pytest documentation
Ruff documentation
mypy documentation
Git reference
GitHub pull requests documentation
logging: Logging facility for Python
The Python Profilers
Python Packaging User Guide
Semantic Versioning 2.0.0
RFC 9110: HTTP Semantics

PostgreSQL documentation
GitHub Actions documentation
Docker Get Started

Jak wybrać miejsce docelowe

Krótkie, praktyczne eksperymenty pomagają wybrać specjalizację w zależności od rodzaju codziennej pracy.

Wybierz rodzaj zadań, a nie nazwę stanowiska

Rozsądne jest wybrać kierunek po kilku krótkich próbach. Proszę zauważyć, że Jaką pracę jesteś gotów powtarzać: projektować HTTP-kontrakty, eksplorować dane, automatyzuj procesy, buduj interfejsy lub dostosowuj pętlę gry. Zainteresowanie Gotowy produkt nie zawsze pokrywa się z zainteresowaniem codzienną pracą inżynierską nad nim.

Jeśli się podoba	Projekt pilotażowy	Prawdopodobna gałąź
Kontrakty, dane, status serwera	Mała API z bazą danych i testami	Backend
Pomiar, tabele, hipotezy, modele	Analiza powtarzalna z baseline i metrykami	Data / ML / AI
Eliminacja ręcznych operacji i narzędzi linkowania	CLI z dry-run, logami i CI	Automation / DevOps
Zdarzenia, formularze, stan lokalny, UX	Aplikacja desktopowa z oddzieloną logiką	Desktop
Interaktywność, fizyka, stan klatkowy	ukończył grę 2D-	Game Development

Dwutygodniowy eksperyment

1. Wybierz dwie naprawdę interesujące gałęzie.
2. Dla każdego wybierz trzy lub cztery zajęcia i stwórz minimalny projekt.
3. Zapisz, co było interesujące w procesie, co było męczące i gdzie brakowało wiedzy.

4. Nie porównywaj piękna rezultatu, lecz chęć dalszego rozwiązywania tego typu problemów.
5. Wybierz jedną główną gałąź na kolejne 8 lub 12 tygodni. Decyzję można zmienić.

Nie buduj swojego stacku wcześniej

Długie lista frameworków nie tworzy specjalizacji. Po pierwsze, potrzebujesz jednego projektu end-to-end i zrozumienia jego fundamentalnych granic. Dodaj drugie narzędzie, gdy będziesz mógł wskazać ograniczenia pierwszego lub wymagania nowego zadania.

Ogólny kontrakt dowolnej gałęzi

Niezależnie od wyboru, Python Core i Software Engineering. są zapisywane w każdym z nich Projekt powinien mieć jasne uruchomienie, kontrolowane zależności, krytyczne testy. logika, błędy możliwe do diagnozowania, dokumentacja rozwiązań oraz historia zmian. Te właściwości Pozwala na przenoszenie doświadczenia między specjalizacjami.

OFICJALNA DOKUMENTACJA

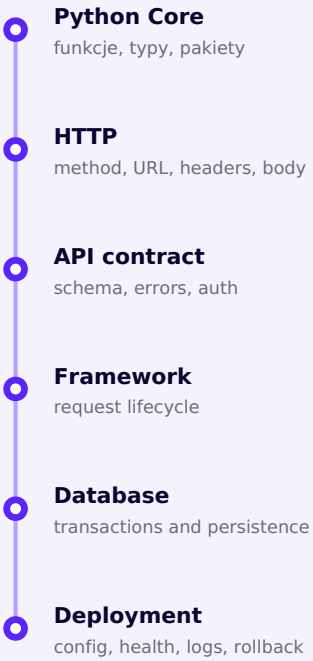
Python 3.14 documentation

Backend: roadmap

HTTP, API-kontrakty, SQL, bezpieczeństwo i eksploatacja przed świadomym wyborem web-frameworka.

Backend zaczyna się od protokołu i danych

Backend przyjmuje żądanie, sprawdza dane i uprawnienia, wykonuje reguły domeny, zmienia stan i zwraca odpowiedź. Framework pomaga wyrazić ten cykl, ale nie zastępuje zrozumienia HTTP, SQL, transakcji, awarii i bezpieczeństwa.



framework Backend-framework ma sens po Python Core, HTTP i API- umowy. Następnie dodaje się bazę danych i wymagania operacyjne.

Sekwencja treningowa

Scena	Explore	Dowód
1. Web fundamentals	HTTP semantics, URL, JSON, cookies, browser/server boundary.	Wyjaśnij prośbę i odpowiedź bez terminów specyficznych dla ram.
2. API and validation	Trasy, schematy, błędy, authn/authz, pagination.	Testy kontraktowe na skuteczne i błędne odpowiedzi.
3. SQL and persistence	Schemat, joins, indeksy, transakcje, migracje.	przechowywane są dane, ograniczenia bazy danych chronią inwarianty.
4. Operations	Konfiguracja, sekrety, logi, health checks, deploy, rollback.	Usługa powstaje z czystego środowiska i diagnozuje awarię.

Flask, FastAPI i Django

Tool	Siła	Kiedy rozważyć
Flask	Mały rdzeń LC0LCC i wyraźna kompozycja rozszerzeń.	Poznawanie granic web-applications, małej usługi, projektu z świadomym wyborem komponentów.
FastAPI	API zbudowany wokół Python type hints, schematów i ASGI.	HTTP API z typowanymi modelami i integracjami asynchronicznymi, jeśli naprawdę ich potrzebujesz.

Tool	Siła	Kiedy rozważyć
Django	Zintegrowany zestaw: ORM, migrations, auth, forms, admin oraz umowy projektowe.	Aplikacja, której przydatna jest spójna pełna platforma.

Flask rozdziały 22 pozostaje ważnym i użytecznym narzędziem. Przejdź do FastAPI lub Django tylko dlatego, że występują w innym roadmap, nie jest konieczne. Porównaj wymagania: WSGI lub ASGI, skład wbudowanych komponentów, model danych, Wielkość zespołu, środowisko operacyjne oraz koszt wsparcia.

Projekt do portfolio

Złóż usługę zadań lub rezerwacji: REST API, PostgreSQL, migracje, role, walidacja, testy logiki domenowej oraz API, structured logging, Docker image i CI. Add Opisz model zagrożeń i podejmuj decyzje o ponownym próbie. Nie dodawaj mikroserwisów, kolejkuj i przechowywać, aż monolit pokaże mierzalną potrzebę ich potrzeby.

Bezpieczeństwo jest sprawdzane na każdej warstwie

Sprawdzaj logowanie na krawędzi, parametryzuj SQL, przechowuj sekrety poza repozytorium, ograniczaj uprawnienia, ustaw limity czasu i nie ujawniaj klientowi traceback wewnętrznego. Framework zmniejsza część ryzyka, ale nie podejmuje decyzji architektonicznych za Ciebie.

OFICJALNA DOKUMENTACJA

RFC 9110: HTTP Semantics
Flask documentation
FastAPI documentation
Django 6.1 documentation
PostgreSQL documentation
Docker Get Started

Data Science, ML i AI: roadmap

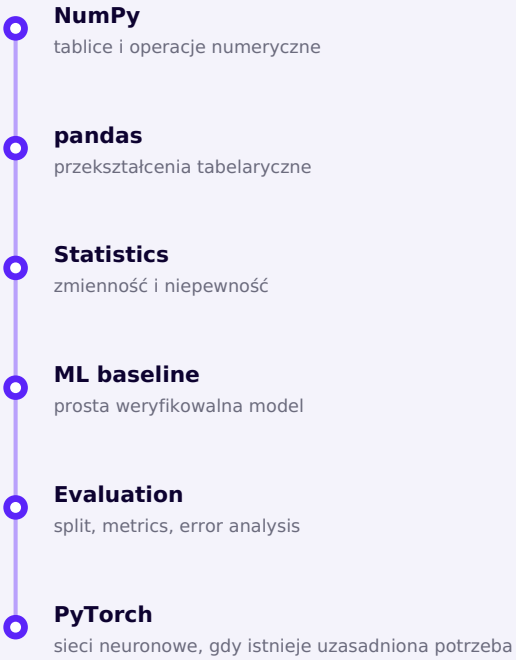
Dane, matematyka, eksperyment, ocena i dostarczenie inżynieryjne bez mieszania użycia modelu z jego treningiem.

Data, ML i AI to nie ta sama umiejętność

Analityk bada i wyjaśnia dane. ML-inżynier buduje i dostarcza system, który korzysta z wytrenowanego modelu. Badacz opracowuje i testuje metody. Role się nakładają, ale wymagają różnej głębokości matematyki, eksperymentu i eksploatacji.

Poziom pracy z modelem	to, co robisz	Co trzeba sprawdzić
Używanie gotowego modelu	Wywołaj lokalny lub zewnętrzny model wewnątrz produktu.	Licencja, dane, koszt, latency, zrzeczenie się, jakość ich skryptów.
Adaptacja i ocena	Konfigurujesz pipeline, cechy, parametry lub fine-tuning.	Baseline, split bez leakage, metryki, powtarzalność error analysis.
Szkolenie i badania	Projektować model treningowy lub procedurę.	Matematyczne wymagania wstępne, sterowanie eksperymentalne, budżet obliczeniowy, uczciwość statystyczna.

Zależności roadmap



Sieć neuronowa nie poprzedza danych, statystyki, baseline ani oceny modelu.

Minimalna dyscyplina naukowa

- Sformułuj zadanie i jednostkę obserwacji przed wyborem algorytmu.
- Zbadaj pochodzenie danych, luki, przesunięcia i dopuszczalność użycia.
- Uchwyć prosty baseline. Model złożony powinien wygrać na podstawie wcześniej wybranego wskaźnika.
- Dziel train, validation i test, aby informacje nie przeciekały między częściami.

- Spójrz nie tylko na średnią, ale także na błędy ważnych grup i scenariuszy.
- Zapisuj kod, konfigurację, seed, wersje danych i środowiska tak bardzo, jak to konieczne do odtworzenia.

Narzędzia według roli

Biblioteka nie zastępuje

NUMPY

n-dimensional arrays
vectorized operations
baza liczbowa

PANDAS

dane tabelaryczne
Czyszczenie i przebudowa
grouping and joins

MATPLOTLIB

graf jako test hipotez
sygnatury, skale uncertainty

SCIKIT-LEARN

preprocessing
classical ML

model selection and evaluation

PYTORCH

tensors
automatic differentiation
neural network training

ENGINEERING

tests for transformations
data contracts
batch or online serving
monitoring

Każde narzędzie odpowiada za swoją część pipeline.

Projekt mający na celu sprawdzenie kierunku

Weź otwarty zbiór danych, sformułuj jedno pytanie testowe, stwórz data audit, baseline i grywalny pipeline. Aby ML, dodaj właściwy podział, metryczne, error analysis i model card z ograniczeniami. Dla aplikacji z gotowym modelem AI- Dodaj zestaw testowy ze swoimi skryptami, timeoutem, obsługą niedostępności i granicą danych. Nie nazywaj demonstracji szkolenia dowodem jakości operacyjnej.

OFICJALNA DOKUMENTACJA

NumPy documentation
pandas documentation
Matplotlib documentation
scikit-learn documentation

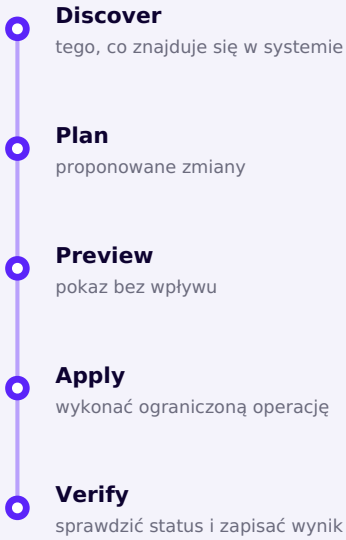
PyTorch documentation
Python Packaging User Guide

Automation i DevOps: roadmap

CLI, granice systemowe, integracje, CI/CD oraz kontenery z kontrolą oddziaływania i awarii.

Automatyzacja zaczyna się na granicy uderzenia

CLI- narzędzie zamienia powtarzalną akcję ręczną w weryfikowalną operację. Najpierw określ dane wejściowe, obserwowany wynik oraz obszar potencjalnych uszkodzeń. Następnie Dodaj preview, idempotencję, logi i kod zwrotny. SafeSort z Rozdział 23 służy jako takie studium przypadku. To projekt szkoleniowy, a nie gotowy menedżer plików.



Sekwencja automatyzacji bezpieczna.

Sekwencja treningowa

Warstwa	Tematy	Kryteria
CLI	argparse lub inny parser, stdin/stdout/stderr, exit codes, config.	Zespół jest gotowy do shell pipeline i dokumentuje błędy.
OS boundaries	pathlib, permissions, processes, signals, environment.	Testy są wykonywane w katalogu tymczasowym i nie zależą od katalogu domowego.
Integration	HTTP API, auth tokens, rate limits, retries, serialization.	Przerwy na żądanie i powtórka nie tworzą niekontrolowanych duplikatów.
CI/CD	workflow, artifact, secrets, environments, deployment gates.	Jedno polecenie walidacyjne działa lokalnie i w CI.
Containers	image layers, runtime config, volumes, health and logs.	Obraz jest powtarzalny, proces został poprawnie wykonany, a stan wyraźnie określony.

Linux i shell

Przeanalizuj model pliku, właścicieli i prawa, procesy, sygnały, wątki pipes, przekierowanie, zmienne środowiskowe i harmonogramowanie zadań. Shell jest przyjazny dla kompozycji istniejące programy. Python lepiej sprawdza się, gdy pojawiają się złożone dane, przenośność, testowalna logika oraz obsługa wielu klas awarii. Często Właściwe rozwiązanie łączy oba poziomy.

Od automatyzacji do DevOps

DevOps nie sprowadza się do Dockerfile ani YAML. To praca nad przepływem dostawy i informacją zwrotną z eksploatacji. Ucz się tworzyć artefakt raz, przenosić go między środowiskami, oddzielać konfigurację i tajemnice, weryfikować health, zbierać logi i mieć plan rollback. Zmiany infrastrukturalne wymagają review i weryfikacji podobnie jak kod.

Tryby awaryjne

Automatyzacja może powtórzyć błąd szybciej niż człowiek. Ogranicz zakres, sprawdź ścieżki i identyfikatory przed modyfikacją, unikaj ukrytych globalnych celów, ustaw limity czasu i spraw, by powtórki były bezpieczne, jeśli to możliwe.

OFICJALNA DOKUMENTACJA

subprocess: Subprocess management
The Python Standard Library
GNU Bash manual
GitHub Actions documentation
Docker Get Started

Desktop: roadmap

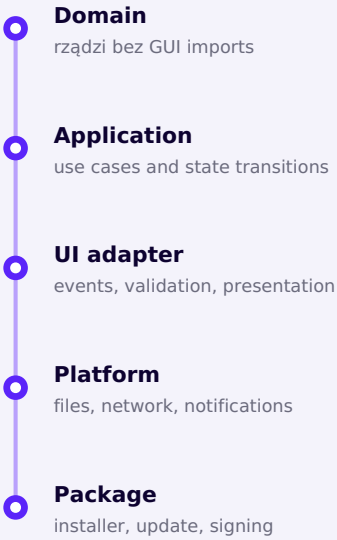
Tkinter i PySide6 w kontekście architektury opartej na zdarzeniach, operacji w tle, dostępności i dostarczania aplikacji.

Kiedy Tkinter wystarczy

Tkinter rozdziały 16 i kolejnych projektów pozostaje standardowym interfejsem Python do Tcl/Tk. Jest przydatny do nauki modelu zdarzeniowego, wewnętrznych narzędzi i aplikacji o umiarkowanej złożoności interfejsu. Dla bogatych interfejsów, złożonego modelu przedstawienia i szerokiego ekosystemu komponentów można rozważyć PySide6, oficjalne wiązania Python dla Qt.

Tool	Odpowiednie	Rozważ
Tkinter	Małe GUI wieloplatformowe, szkolenia i narzędzia wewnętrzne.	Architektura, zadania w tle, packaging i platform testing są nadal projektowane przez programistę.
PySide6 / Qt	Bogatsze widgets, model/view, QML oraz większe interfejsy desktop-.	Nowy model obiektowy, event loop, licencjonowanie komponentów i bardziej złożona dostawa.

Architektura przed projektem okien



GUI przekazuje zdarzenia logice aplikacji; cały program nie może znajdować się w obsługach okien.

Plan Rozwoju desktop- Developer

- Model zdarzenia: event loop, callbacks, stan formy oraz wyłączanie blokowania pracy w UI thread.
- Kompozycja interfejsu: layout, widgets, ostrość, nawigacja klawiaturą, accessibility oraz lokalizacja.
- Architektura: domain logic separacja, polecenia, modele widoku i adaptery zasobów.
- Operacje w tle: cancellation, progress, errors i bezpieczne przekazanie wyniku do UI.
- Dane: Ustawienia, migracje lokalnych schematów, backup i przywracanie.
- Dostawa: buduj pod docelowy system operacyjny, zależności, ikony, installer, aktualizuj i testuj na czystej maszynie.

Projekt mający na celu sprawdzenie kierunku

Zbieraj lokalny katalog notatek lub plików za pomocą wyszukiwania. Domain layer musi Testuj bez uruchamiania okna. Dodaj import i eksport, cofnij długą operację, wyraźny komunikat o błędzie i build dla co najmniej jednego docelowego systemu operacyjnego. Osobno sprawdzenie Sterowanie klawiaturą i szerokość okna na małym ekranie.

OFICJALNA DOKUMENTACJA

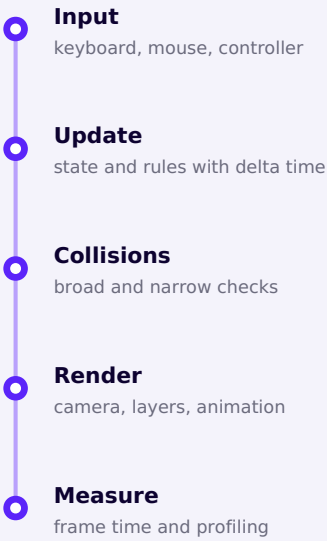
Python 3.14 documentation
Qt for Python documentation
Concurrent Execution

Game Development: roadmap

Pygame zrozumieć cykl, stan i czas 2D-; przejść do innych silników zgodnie z wymaganiami celu.

Pygame jako dobre laboratorium 2D

Pygame pozwala studiować pętlę gry, zdarzenia, współrzędne, sprite'y, kolizje, zarządzanie dźwiękiem i stanem. Dla szkoleń i niezależnych projektów 2D- są to wygodna platforma. Jednocześnie Python nie dominuje we wszystkich zawodach silniki gier i klasy komercyjne. Jeśli cel tego wymaga Dla konkretnego silnika kolejną wiedzą może być jego język, edytor i pipeline zasobów.



Jeden kadr pętli gry; przejścia stanów powinny pozostać jawne.

Pogłębianie porządku

Warstwa	Tematy	Rezultat
Game state	scene/state machine, pause, restart, save.	Przejścia nie zależą od losowego zestawu globalnych flag.
Time and motion	delta time, fixed time-step where needed, interpolation.	Szybkość logiki nie zmienia się wraz z FPS.
World	sprites, animation, collisions, camera, tile maps.	System pozostaje mierzalny i testowalny na granicach.
Content	assets, audio, levels, data-driven configuration.	Nowy poziom nie wymaga przepisywania engine logic.

Warstwa	Tematy	Rezultat
Delivery	settings, packaging, controller support, performance budget.	Gra uruchamia się na docelowej maszynie i ma zakończony cykl gry.

Matematyka na żądanie

Wektory, trygonometria, transformacje współrzędnych i prosta fizyka stają się przydatne, gdy pojawiają się ruch, kamera, kolizje i wskazywanie. Studiuj je wraz z obserwowaną mechaniką. W przypadku fizyki złożonej najpierw określ, co jest akceptowalne Zwiększaj i budżetuj kadrem, a następnie wybierz bibliotekę lub własny model.

Projekt do portfolio

Doprowadź małą grę 2D- do stanu, w którym jest początek, zasady, zakończenie, dźwięk, ustawienia i kompilacja. Dodaj diagram stanów, opis architektury, krótki film i tabelę znanych ograniczeń. Ukończony cykl gry daje sprawdzalny dowód działania, którego nie ma prototyp bez ekranu końcowego i instrukcji uruchomienia.

Kiedy zmienić narzędzie

Jeśli cel został powiązany z konkretnym silnikiem, 3D pipeline, platformą lub zespołem, zbadaj ich wymagania. Doświadczenie ze stanem, czasem, współrzędnymi, profilowaniem i Git będzie się przenosić nawet jeśli język i API się zmienią.

OFICJALNA DOKUMENTACJA

Pygame documentation
The Python Profilers
Unity Manual: Introduction to programming
Unreal Engine: Programming with C++

Portfolio dewelopera

Znaczące projekty z dowodami na rozwiązania, walidację i architekturę, zamiast powtarzających się materiałów szkoleniowych.

Portfolio pokazuje rozwiązania

Repozytorium jest przydatne, gdy inny programista rozumie zadanie, uruchamia projekt, Sprawdź podstawowe właściwości i zobacz, dlaczego architektura wygląda tak, a nie inaczej. Liczba projektów nie rekompensuje braku ukończenia i wyjaśnień.

Powtórzenia edukacyjne i portfolio inżynierskie

Powtórki z treningów	Portfolio inżynierskie
Powtarza wcześniej ustalone kroki i znany wynik.	Zaczyna się od samodzielnie zdefiniowanego problemu, użytkownika i granic.
Pokazuje tylko końcowy kod lub zrzut ekranu.	Zapewnia powtarzalną instalację, demo, testy oraz weryfikowalną wersję.
Ukrywa bieg zmian i powody podejmowania decyzji.	Zachowuje historię Git, Issues, małe PR i opis kompromisów architektonicznych.
Trudno odróżnić rozwiązania autora od instrukcji.	README i documentation wyjaśniają wkład, ograniczenia i przyszłe prace autora.

Drabina dojrzałości projektu

Poziom	Co widać w projekcie	Weryfikowalny wynik
Level 1	Skrypt rozwiązuje jedno zadanie lokalnie.	Autor może odtworzyć wynik.
Level 2	Istnieją funkcje, struktura katalogów README oraz obsługa spodziewanych błędów.	Nowy użytkownik rozpoczyna projekt zgodnie z instrukcjami.
Level 3	Krytyczna logika jest omawiana przez testy; lint i CI powtarzają lokalną weryfikację.	Zmiana przechodzi automatyczną kontrolę jakości.

Poziom	Co widać w projekcie	Weryfikowalny wynik
Level 4	Są jasne interfejsy, konfiguracja, obserwowalność, budowa pakietów i notatki do wydania.	Artefakt umieszcza się w czystym środowisku i diagnozuje awarię.
Level 5	Opisane są kompromisy, ograniczenia, model zagrożenia lub awarii oraz scenariusz operacyjny.	Rozwiązanie można omówić jako system inżynierski.

Ta drabinka opisuje dojrzałość dema, a nie rangę dewelopera. Nie wszyscy Projekt potrzebuje Level 5. Jeden głęboki projekt może to osiągnąć, a dwa mniejsze to pokażą różnorodnych zadań.

Jak wybierać projekty

Uniwersalna liczba projektów, a tym bardziej gwarancja zatrudnienia, nie istnieje. Wybierz minimalny zestaw, który pokazuje ogólny fundament, wybraną specjalizację i jakość procesu inżynieryjnego. Skład może być taki:

- jeden mały CLI lub biblioteka z czystym API i pakietem;
- jeden główny projekt specjalizacyjny z prawdziwą granicą: DB, GUI, data pipeline lub game loop;
- jednym z projektów, gdzie szczególnie widoczne są testy, CI i diagnostyka;
- w razie potrzeby jedna komenda lub open-source zakładka;
- opcjonalny eksperyment, pokazujący zainteresowanie badawcze.

Co powinno być w każdym repozytorium

Co czytelnik powinien znaleźć

- README z określeniem problemu, użytkownika, granic rozwiązania i statusu projektu.
- Zrzuty ekranu lub krótka demonstracja potwierdzająca podstawowy scenariusz bez podmieniania opisu technicznego.
- Powtarzalna instalacja: wersje, zależności, instalacja, uruchamianie, testy i komendy budowania.
- Schemat architektury, interfejsy, kluczowe kompromisy i znane ograniczenia.
- Krytyczne testy logiczne i CI powtarzające lokalne kontrole jakości.
- Historia Git, Issues i drobne zmiany pokazujące kierunek rozwiązań inżynieryjnych.
- Wersja wydania lub instalowalny artefakt z notatkami do wydania.
- LICENSE i dokumentację, wystarczające do zadeklarowanego sposobu użycia.

Nie komplikuj projektu przez listę technologii

Nie dodawaj Kubernetes, message brokera, mikroserwisów ani sieci neuronowych dla samej listy technologie. Dodaj komponent, gdy możesz sformułować wymagania, alternatywy, Koszt i sposób weryfikacji. Historia niewielkich, znaczących PR wpływa na rozwój projektu Wyraźniej niż jeden gigantyczny commit.

OFICJALNA DOKUMENTACJA

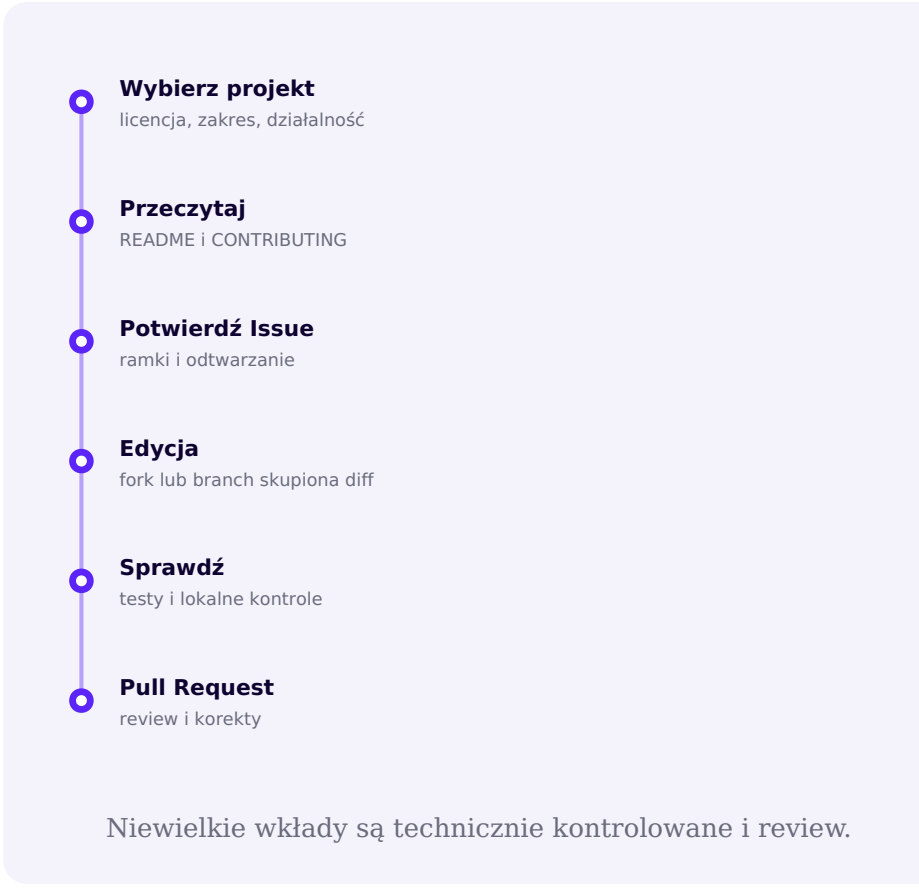
Git reference
GitHub pull requests documentation
pytest documentation
GitHub Actions documentation
Python Packaging User Guide

Pierwszy wkład w Open Source

Mały wkład do sprawdzenia, szacunek dla zasad projektu, review i licencji.

Pierwszy depozyt może być niewielki

Open Source nie wymaga zaczynać od nowego algorytmu w CPython. Poprawka literówki, powtarzalny bug report, test dla potwierdzonego błędu lub wyjaśnienie przykładu może być wartościowym wkładem. Wartość określana jest regułami konkretnego projektu i jakością interakcji.



Niewielkie wkłady są technicznie kontrolowane i review.

Sekwencja robocza

1. **Wybierz projekt:** sprawdzić zakres, działalność, licencję i przydatność swoich umiejętności.
2. **Przeczytaj README:** rozumieć przydział, instalację i wspierane wersje.
3. **Przeczytaj CONTRIBUTING:** stosować code of conduct, szablony i polecenia walidacyjne.
4. **Wybierz Issue:** znajdź małe zweryfikowane zadanie lub uzgodnij nowe.
5. **Powtórz:** zapisać minimalny scenariusz w obsługiwanej wersji.
6. **Utwórz fork lub branch:** stosować się do zasad projektu i wyizolować zmianę.
7. **Wprowadź zmianę:** diff skupiać się i nie mieszać niezwiązanych z tym zmian.
8. **Przeprowadz testy:** dodać weryfikowalny dowód i przeprowadzić lokalne weryfikacje.
9. **Otwórz Pull Request:** opisz powód, zakres i przeprowadzoną kontrolę.
10. **Przejdź review:** spokojnie odpowiadać na pytania oraz omawiać żądania i kompromisy.
11. **Wprowadź poprawki:** zaktualizować kod i testy; po rozwiązaniu podziękować uczestnikom.

Umiejętności społeczne są częścią inżynierii

Towarzyszący projektowi mają mało czasu. Krótki kontekst, powtarzalny przykład i ukierunkowany diff skracają czas na review.. Niezgody omawiaj poprzez wymagania i kompromisy. Nie traktuj uwagi do kodu jako oceny osoby i nie wymagaj natychmiastowej odpowiedzi.

Licencje i pochodzenie kodu

Publicznie dostępny kod nie oznacza pozwolenia na jego kopiowanie bez warunków. Przeczytaj licencję projektu źródłowego i docelowego, zachowaj obowiązkowe powiadomienia i nie wklejaj kodu nieznanego pochodzenia. Jeśli warunki są niejasne, zadaj pytanie opiekunom projektu lub wybierz własną implementację. To zasada edukacyjna, a nie porada prawna.

Dobry pierwszy gol

Szukaj zadania, które można zrozumieć i przetestować na kilku lekcjach. Mały, akceptowany diff uczy czytania cudzego kodu, zasad projektu i review lepiej niż duży, nieomawiany rewrite.

OFICJALNA DOKUMENTACJA

Git reference

GitHub pull requests documentation

pytest documentation

Jak dalej się uczyć

Jak wyjść z tutorial hell poprzez samorozkład, debugowanie eksperymentów i czytanie dokumentacji.

Jak nazywamy tutorial hell

Czasem nazywa się to stanem, w którym osoba pewnie powtarza krok po kroku lekcje, ale nie może rozpocząć własnego zadania bez gotowej sekwencji działań. To nie jest diagnoza ani powód do wstydu. Zazwyczaj nie ma wystarczającego przejścia od rozpoznania do niezależne rozwiązywanie problemów, dekompozycja i debugowanie.

Cykl samodzielnej nauki

Predict
co musi się wydarzyć

Do
minimalny krok bez kopiowania

Obserwować
wynik, traceback, status

Wyjaśnij
dlaczego wynik taki

Edycja
jeden warunek i ponownie przetestuj

W ten sposób błąd staje się częścią obserwowanego eksperymentu.

Protokół diagnostyczny

1. Zapisz oczekiwany i rzeczywisty rezultat.
2. Spróś problem do minimalnie powtarzalnego przykładu.
3. Przeczytaj typ wyjątku i ostatnią odpowiednią linię traceback.
4. Sprawdź wejścia, typy, stan i założenia na granicy defektu.
5. Sformułuj jedną hipotezę i zmień jedną zmienną.
6. Po poprawce dodaj test, który przedtem by upadł.
7. Zapisz powód, a nie tylko ostatnią linijkę kodu.

Cztery tryby dokumentacji

Mode	Pytanie czytelnika	Jak używać
Tutorial	Jak przejść tę ścieżkę z autorem po raz pierwszy?	Dla zapoznania; potem powtórz tę część bez instrukcji.
How-to	Jak rozwiązać konkretne praktyczne zadanie?	Kiedy cel jest już jasny i potrzebny jest sprawdzony przepis.
Reference	Jak dokładnie działa ten API?	Weryfikacja sygnatur, parametrów, wyjątków i części wersjonowanych.
Explanation	Dlaczego system został zaprojektowany w ten sposób?	Budować model, porównywać koncepcje i rozumieć kompromisy.

Oficjalna dokumentacja często oddziela te tryby. Znanym przykładem jest dokumentacja Django: ma osobne tutorial, topics guides (explanation), how-to guides oraz reference sam schemat Diátaxis wyrosł z doświadczenia przetwarzania tej dokumentacji. Nie próbuj czytać reference po rządzie jak podręcznika. Zacznij od tutorial lub explanation, i reference bądź blisko podczas wdrażania. Sprawdź wersję dokumentacji i minimum Przykład lokalnie.

Od dokumentacji do implementacji

Zacznij od oficjalnej dokumentacji odpowiedniej wersji. Tutorial pomaga po raz pierwszy przejść spójną ścieżkę, explanation buduje model, how-to rozwiązuje konkretne zadanie, a reference ustala dokładny kontrakt API. Jeśli zachowanie jest nadal niejasne, przeczytaj kod źródłowy odpowiedniego fragmentu i powiązane testy. Następnie sprawdź issue tracker: tam mogą być potwierdzone ograniczenia, poprawki i rozwiązania towarzyszących projektów. Kod źródłowy i Issues uzupełniają dokumentację, ale nie unieważniają publicznego kontraktu i notatek do wydania.

Korzystanie z wyszukiwarek i asystentów

Formułuj zapytanie przez obserwowany symptom, wersję, minimalny kod i już wykonane sprawdzenia. Traktuj uzyskaną odpowiedź jako hipotezę: porównaj ją z oficjalnym API, wykonaj test i wyjaśnij każdy wiersz. Nie wklejaj sekretów, zamkniętego kodu ani danych użytkownika do systemu zewnętrznego bez zgody.

Cotygodniowa Retrospektywa

- Co mogę napisać i wyjaśnić bez próbki?
- Jakiego błędu nauczyłem się diagnozować?
- Jakie rozwiązanie potwierdził test lub pomiar?
- Jaka kolejna przestrzeń ogranicza projekt?

OFICJALNA DOKUMENTACJA

Diátaxis documentation framework
Python 3.14 documentation
Django 6.1 documentation

Kolejne kursy Cartesian School

Pięć kierunków, które są w przygotowaniu. Daty wydania i formy zapisu jeszcze nie ma.

Kontynuacja w Cartesian School

Poniżej znajdują się wskazówki, które stanowią fundament tej książki. Są one w Przygotowanie. Karty nie są formą rejestracji, nie obiecują daty wydania i nie są widoczne fikcyjny postęp. Przed publikacją skorzystaj z roadmap tego rozdziału oraz oficjalnej dokumentacji.

Przygotowania

Advanced Python

zawodowy Python

Model iteracyjny, dekoratory, menedżery kontekstu, typowanie, architektura pakietów, współbieżność i profilowanie.

Wkrótce

Python Backend Developer

od HTTP- kontraktu do eksploatacji

API, SQL, PostgreSQL, uwierzytelnianie i autoryzacja, testowanie, obserwowalność, kontenery oraz wdrażanie.

Przygotowania

Python Testing & Software Engineering

zmiany, którym możesz zaufać

pytest, projektowanie testów, Git procesy, review, Ruff, statyczne typowanie, budowanie pakietów, CI oraz przygotowanie do wydawnictw.

Wkrótce

Python for Data & AI

danych, eksperymentów i ML- systemu

NumPy, pandas, wizualizacja, baseline, ocena modeli, reprodukowalność i granice inżynierskie AI-aplikacji.

Przygotowania

Python Automation & DevOps

niezawodnych narzędzi i przepływu dostaw

CLI, Linux, procesy, API- integracje, bezpieczne operacje plikowe, CI/CD, kontenery oraz obserwatorność.

Jak wybrać kolejny kurs

Nie wybieraj długości listy technologii. Porównaj wcześniejszą wiedzę z obecną projekty i znajdź kurs, który obejmuje najbliższe ograniczenia. Jeśli jest trudno projektuj funkcje i dane, najpierw wzmocnij Professional Python Core. Jeśli kod Działa, ale zmiany to psują, priorytetem będzie testowanie i Software Engineering. Wybierz specjalizację po małym, niezależnym prototypie.

Status materiałów

Wszystkie wymienione kursy nie zostały jeszcze wydane i są wyraźnie oznaczone statusem „Wkrótce” lub „W przygotowaniu”. Na tej stronie nie ma ceny, daty, linku do płatności ani fikcyjnej rejestracji.

Twój następny krok

Jedna wybrana ścieżka, jeden sprawdzany projekt i jedno pytanie, jakim Python- programistą chcesz zostać.

Wybierz jeden następny wynik

Nie trzeba przechodzić pięciu kierunków jednocześnie. Wybierz projekt, który można doprowadzić do obserwowalnego wyniku w najbliższych tygodniach. Opisz użytkownika, zadanie, granice pierwszej wersji i sposób weryfikacji. Następnie załóż Issue i utwórz gałąź.

Rozwiązanie	Record Now
Kierunek	Backend, Data / ML / AI, Automation / DevOps, Desktop lub Game Development.

Rozwiązanie	Record Now
Projekt	Jedno zdanie: kto dostaje jaki wynik.
Granica	Co będzie zawarte w pierwszej wersji, a co nie będzie świadomie uwzględnione.
Dowód	Test, skrypt demonstracyjny, metryka lub konfiguracja reprodukcyjna.
Rhythm	Realistyczne dni pracy i data krótkiej retrospektywy.

Pytanie, które pozostaje

Odpowiedź nie musi być ostateczna. Powinna być wystarczająco konkretna, aby dzisiaj otworzyć repozytorium, opisać Issue i wykonać pierwszy sprawdzalny krok.

Przeglądaj roadmap po każdej iteracji projektu.

Zanim zaczniesz

- mogę wyjaśnić, dlaczego ten projekt istnieje.
- ustaliłem minimalny wynik i celowo odłożyłem resztę.
- Wiem, jak testować krytyczne zachowania.
- jestem gotów zapisywać błędy i rozwiązania, a nie je ukrywać.

OFICJALNA DOKUMENTACJA

Python 3.14 documentation
Git reference
pytest documentation

Teraz pytanie przestaje być prawdziwe:

„Czy będę umiał programować?”

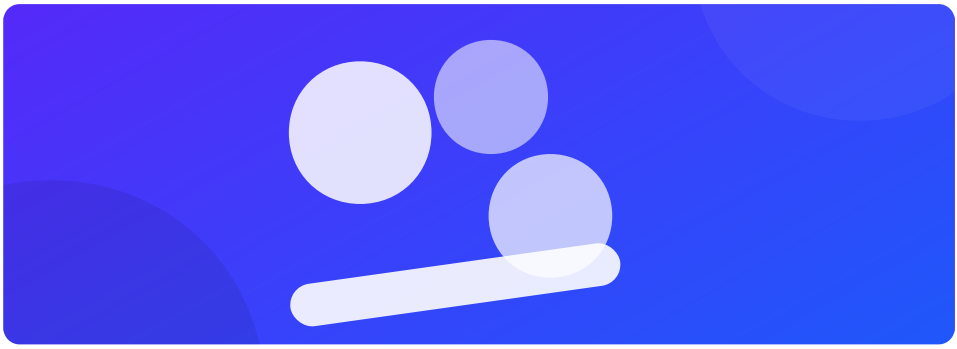
Następne pytanie:

„Jakim Python- programistą chcę zostać?”

Książka się skończyła. Twój roadmap nie.

Projekty

Dwanaście gotowych mini-projektów z otwartym kodem źródłowym — od „Kółko i krzyżyk” po pełnoprawną kosmiczną strzelankę. Kod źródłowy i działająca wersja każdego z nich — na stronie kursu.



Rysowalnia

Prawdziwa aplikacja do rysowania do Tkinter: wybierania kształtu, koloru i grubości linii, rysowania myszką na płótnie.

Tkinter GUI Canvas

Rozdział 18
Teoria, z której wyrosł ten projekt

Praktyka 18-07
18-07 · Pełny załącznik

Kod źródłowy

Kompletny, już zeskanowany plik – osobno:

projects/tkinter/paint-app/paint_app.py

Uruchom lokalnie: `python paint_app.py`

Wąż

Klasyczna gra „Wąż”

Turtle Gry Klawiatura

Rozdział 19
Teoria, z której wyrosł ten projekt

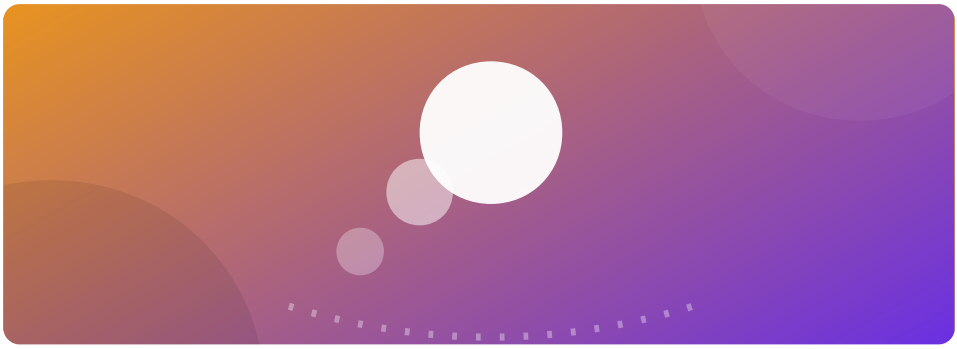
Praktyka 19-08
19-08 · Pełna gra

Kod źródłowy

Kompletny, już zeskanowany plik – osobno:

projects/turtle/snake/snake.py

Uruchom lokalnie: `python snake.py`



Piłka odbijająca się

Pierwsza minigra Pygame: ball odbija się od wszystkich czterech ścian ekranu w swojej własnej pętli rozgrywki.

Pygame Pętla rozgrywki

Rozdział 20
Teoria, z której wyrosł ten projekt

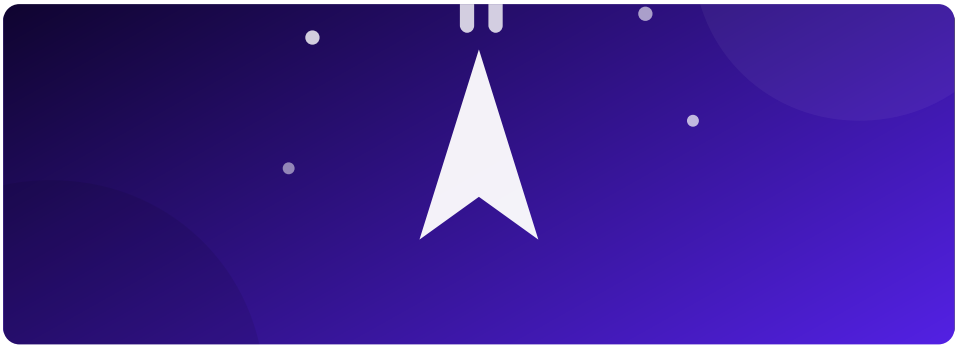
Praktyka 20-05
20-05 · Mini-projekt: odbijająca się piłka

Kod źródłowy

Kompletny, już zeskanowany plik – osobno:

projects/pygame/bouncing-ball/bouncing_ball.py

Uruchom lokalnie: `python bouncing_ball.py`



Space shooter

Pełna gra na Pygame: statkiem gracza, wrogami, strzelaniną, kolizjami, punktacją i ekranem Game Over.

Pygame Gry Klawiatura

Rozdział 21
Teoria, z której wyrosł ten projekt

Praktyka 21-08
21-08 · Pełny mecz

Kod źródłowy

Kompletny, już zeskanowany plik – osobno:

projects/pygame/space-shooter/space_shooter.py

Uruchom lokalnie: `python space_shooter.py`

Lista zadań

Ministrona o Flask z listą zadań: trasy, szablony HTML- Jinja2 oraz odbiór danych z formularza.

Flask Tworzenie stron internetowych HTML

Rozdział 22
Teoria, z której wyrosł ten projekt

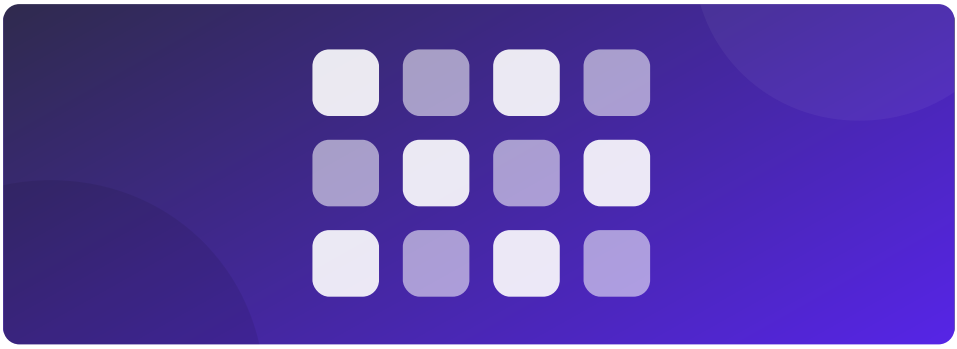
Praktyka 22-05
22-05 · Flask w Python

Kod źródłowy

Kompletny, już zeskanowany plik – osobno:

projects/flask/todo-app/app.py

Uruchom lokalnie: `python app.py`



Kalkulator

Kalkulator Tkinter: siatki przycisków i bezpiecznych obliczeń wyrażeń arytmetycznych.

Tkinter GUI

Rozdział 23
Teoria, z której wyrosł ten projekt

Praktyka 23-01
23-01 · Kalkulator z Tkinter

Kod źródłowy

Kompletny, już zeskanowany plik – osobno:

projects/tkinter/calculator/calculator.py

Uruchom lokalnie: `python calculator.py`

Generator losowych historii

wypełnia szablon zdania losowo wybranymi słowami z kilku list – otrzymujesz małe, absurdalne historie.

random **Strings**

Rozdział 23
Teoria, z której wyrosł ten projekt

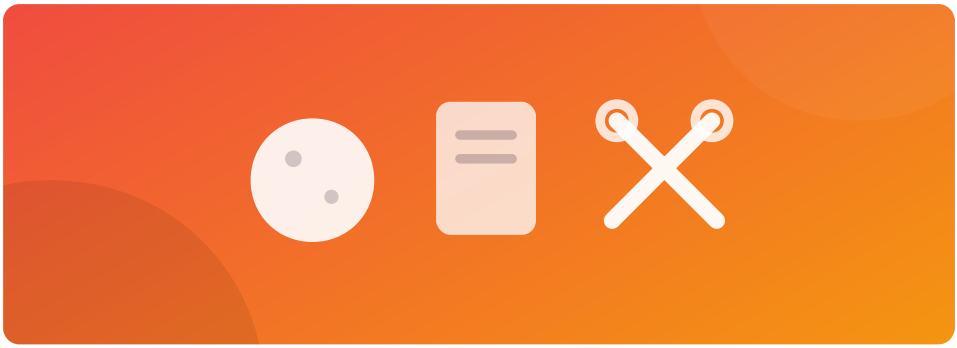
Praktyka 23-02
23-02 · Generator losowych historii

Kod źródłowy

Kompletny, już zeskanowany plik – osobno:

projects/console/story-generator/story_generator.py

Uruchom lokalnie: `python story_generator.py`



Papier, kamień, nożyce

Klasyczna gra przeciwko komputerowi: logika zwycięzcy w jednym słowniku i symulacja rund.

random Słowniki

Rozdział 23
Teoria, z której wyrosł ten projekt

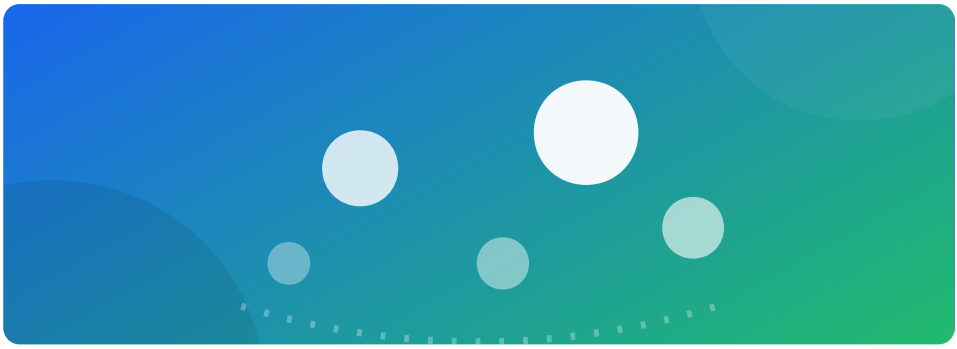
Praktyka 23-03
23-03 · Kamień, papier, nożyce

Kod źródłowy

Kompletny, już zeskanowany plik – osobno:

projects/console/rock-paper-scissors/rps.py

Uruchom lokalnie: `python rps.py`



Odbijająca się piłka — wersja z klasą

Ta sama odbijająca się kula, przepisana przez klasę Myach – co ułatwia nakręcanie kilku niezależnych piłek jednocześnie.

Pygame OOP

Rozdział 23
Teoria, z której wyrosł ten projekt

Praktyka 23-04
23-04 · Odbijająca się piłka od Pygame

Kod źródłowy

Kompletny, już zeskanowany plik – osobno:

projects/pygame/bouncing-balls-oop/bouncing_balls.py

Uruchom lokalnie: `python bouncing_balls.py`

Konwersja temperatury

Aplikacja na Tkinter do konwersji temperatur między Celsjuszem, Fahrenheitem i Kelvinem.

Tkinter GUI

Rozdział 23
Teoria, z której wyrosł ten projekt

Praktyka 23-05
23-05 · Przeliczanie temperatury

Kod źródłowy

Kompletny, już zeskanowany plik – osobno:

projects/tkinter/temperature-converter/temperature_converter.py

Uruchom lokalnie: `python temperature_converter.py`

Przypisy

Najpierw zapoznanie się z plikami i Tkinter: pola tekstowego, zapisując i ładując notatkę do zwykłego pliku na dysku.

Tkinter Pliki

Rozdział 23
Teoria, z której wyrosł ten projekt

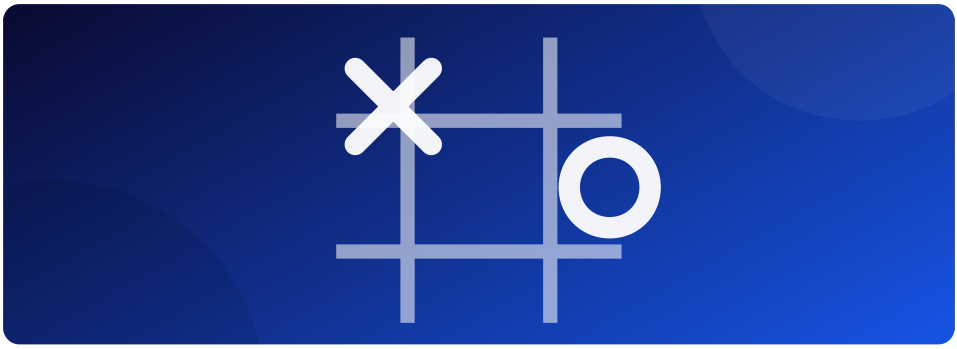
Praktyka 23-06
23-06 · Pliki i Tkinter

Kod źródłowy

Kompletny, już zeskanowany plik – osobno:

projects/tkinter/notes-app/notes_app.py

Uruchom lokalnie: `python notes_app.py`



Kółko i krzyżyk

Pełnoprawna gra „Kółko i krzyżyk” na Tkinter: siatka przycisków, ruchy na przemian i sprawdzanie wszystkich ośmiu linii zwycięskich.

Tkinter Gry

Rozdział 17
Teoria, z której wyrosł ten projekt

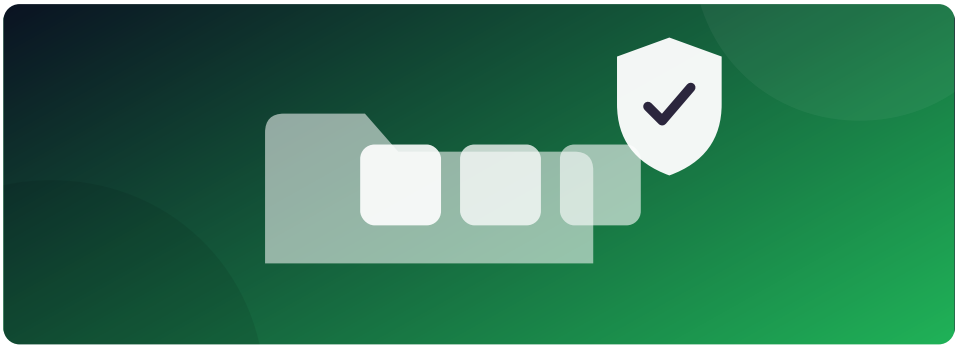
Praktyka 17-06
17-06 · Nowa gra i pełny program

Kod źródłowy

Kompletny, już zeskanowany plik – osobno:

projects/tkinter/tic-tac-toe/tic_tac_toe.py

Uruchom lokalnie: `python tic_tac_toe.py`



SafeSort

Lokalne narzędzie wiersza poleceń do bezpiecznego sortowania plików i znajdowania duplikatów: planuj ruchy bez zmiany plików, cofnij ostatnią operację.

CLI pathlib hashlib

Rozdział 23
Teoria, z której wyrosł ten projekt

Praktyka 23-08
23-08 · Operacje z Path

Kod źródłowy

Kompletny, już zeskanowany plik – osobno:

projects/python/safesort/README.md

Uruchom lokalnie: `python README.md`

Indeks rzeczowy

Szybki sposób, aby znaleźć rozdział obejmujący właściwy termin.

Terminy uporządkowane alfabetycznie. Link prowadzi do rozdziału, gdzie termin jest rozważany.

Symbole i angielskie terminy

A do Z