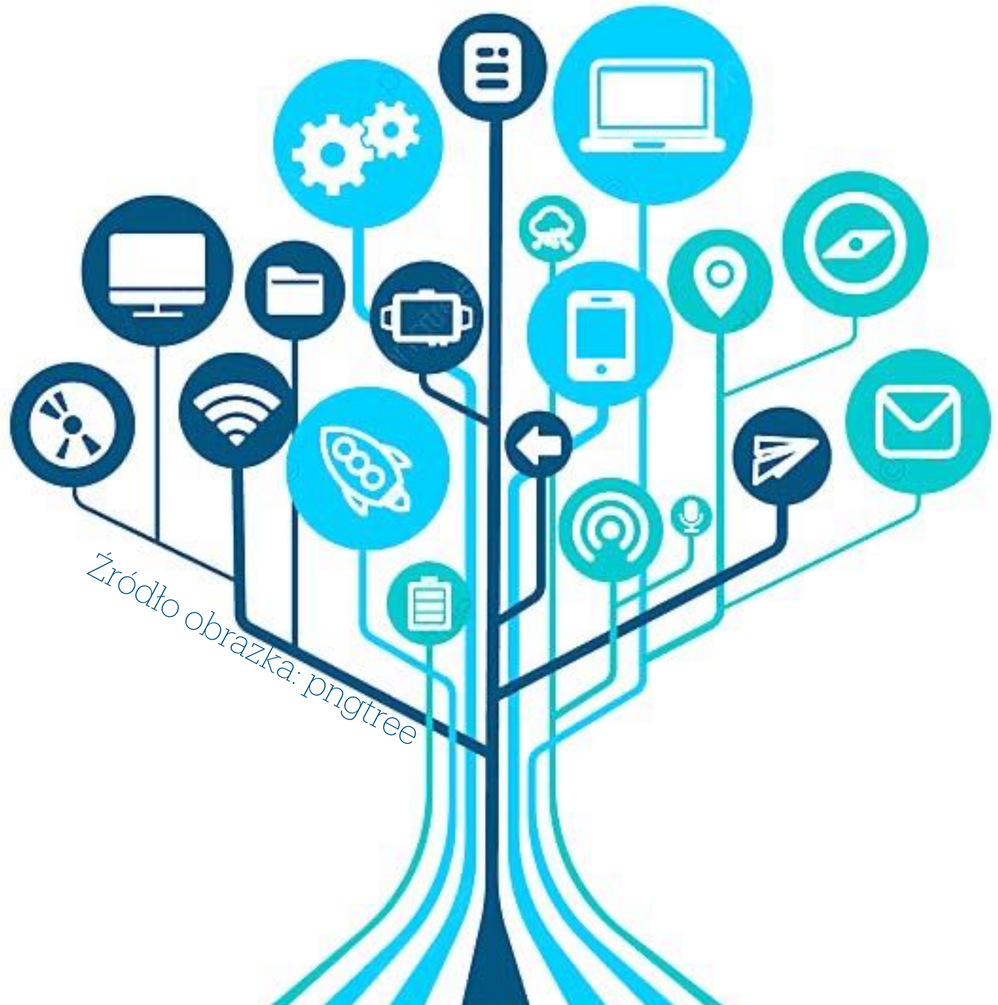
Technologie informacyjne i komunikacyjne

Wykład 8, dn. 27.04.2026



Źródło obrazka: pngtree

Wstęp

Programowania nie da się nauczyć - nawet z najlepszych materiałów - na dzień przed egzaminem czy kolokwium. Programowanie to nie tylko wiedza o składni i regułach języka, ale przede wszystkim umiejętność ich stosowania w praktyce, którą można posiąść tylko pisząc programy.

Źródło: Wikipedia

Czym jest Python?

Python to język programowania wysokiego poziomu, ceniony za prostą składnię i czytelność kodu. Został zaprojektowany tak, aby programy dało się pisać zwięźle i jasno, dlatego jest często polecany jako pierwszy język programowania.

```
print("Hello, world!")
```

Już jedna linijka kodu tworzy działający program.

Dlaczego Python?

Python jest:

- prosty do nauki,
- wszechstronny,
- posiada ogromną liczbę gotowych bibliotek,
- używany zarówno przez początkujących, jak i profesjonalistów.

Jednym językiem można rozwiązywać bardzo różne problemy.

Python świetnie nadaje się do wykonywania powtarzalnych zadań, np. zmiana nazw wielu plików, przetwarzanie danych z arkuszy, wysyłanie automatycznych maili, tworzenie prostych botów i skryptów. Zamiast robić coś ręcznie 100 razy można napisać program.

Analiza danych

Python jest jednym z najpopularniejszych narzędzi do pracy z danymi.

Używa się go do:

- obliczeń,
- statystyki,
- wizualizacji danych,
- analizy dużych zbiorów danych.

Popularne biblioteki: NumPy, pandas, Matplotlib

Pierwszy program

Tradycyjnie naukę nowego języka programowania zaczyna się od bardzo prostego programu:

```
print("Hello, world!")
```

Choć wygląda niepozornie, zawiera kilka ważnych pojęć.

Czym jest program? Program to po prostu lista instrukcji dla komputera — przepis mówiący mu, co ma zrobić. Podobnie jak przepis kulinarny składa się z kolejnych kroków, program składa się z poleceń wykonywanych przez komputer. Na przykład:

1. Wyświetl tekst.
2. Pobierz dane od użytkownika.
3. Wykonaj obliczenia.
4. Pokaż wynik.

To już jest program.

Pierwszy program

Instrukcje są wykonywane po kolei.
Domyślnie Python wykonuje kod od góry do dołu, instrukcja po instrukcji.

```
print("Pierwsza linia")
print("Druga linia")
print("Trzecia linia")
```

Wynik:

```
Pierwsza linia
Druga linia
Trzecia linia
```

Program wykonuje polecenia w podanej kolejności.

Co robi `print()`?

Funkcja `print()` wyświetla tekst (lub inne dane) na ekranie.

```
print("Cześć!")
print(2 + 3)
```

Wynik:

```
Cześć!
5
```

Może wypisywać zarówno tekst, jak i wyniki obliczeń.

Pierwszy program

Co oznacza zapis z nawiasami?

```
print("Hello, world!")
```

Składa się z:

- `print` — nazwa funkcji,
- `(...)` — nawiasy przekazujące dane do funkcji,
- `"Hello, world!"` — tekst do wyświetlenia.

Tekst zapisany w cudzysłowie nazywamy **napisem** (*string*).

Dlaczego „Hello, world!”?

To tradycyjny pierwszy program w nauce programowania. Jest prosty i pozwala sprawdzić, że wszystko działa poprawnie. Nie chodzi o sam napis - równie dobrze można napisać:

```
print("Witaj Pythonie!")
```

Pierwszy program

Pierwsza modyfikacja programu: Możemy już napisać własny program:

```
print("Mam na imię Anna")

print("Uczę się Pythona")
```

To już program złożony z dwóch instrukcji.

Najważniejsza idea: Programowanie zaczyna się od wydawania komputerowi prostych poleceń.

`print()` jest naszym pierwszym takim poleceniem. Od niego zaczynamy budować coraz bardziej złożone programy.

Zmienne

```
imie = "Anna"
wiek = 20
```

Przechowywanie danych: W programowaniu często potrzebujemy **zapamiętać jakieś informacje**, aby móc ich później używać. Do tego służą **zmienne**. Zmienne można traktować jak „etykiety” przypięte do danych — pozwalają przechować wartość i odwoływać się do niej w dowolnym miejscu programu.

Na przykład:

```
imie = "Anna"
print(imie)
```

Program zapamiętuje tekst `"Anna"` i może go później wyświetlić lub wykorzystać w obliczeniach.

Zmienne

Nazywanie wartości: Zmienne pozwalają nadać nazwę danym, dzięki czemu kod staje się czytelniejszy.

Zamiast pisać:

```
print("Anna")
print(20)
```

możemy napisać:

```
imie = "Anna"
wiek = 20

print(imie)
print(wiek)
```

Dzięki temu od razu wiadomo, co oznaczają dane wartości.

Zmienne

Jak działa przypisanie? Operator **=** w Pythonie oznacza przypisanie wartości do zmiennej.

```
wiek = 20
```

Oznacza: „utwórz nazwę **wiek** i przypisz do niej wartość **20**”.

To nie jest równanie matematyczne - nie pytamy, czy coś jest równe, tylko zapisujemy informację.

Zmienne mogą się zmieniać: Wartość zmiennej można później nadpisać:

```
liczba = 5
liczba = 10

print(liczba)
```

Wynik:

```
10
```

Oznacza to, że zmienna przechowuje aktualną wartość, a nie „stałą”.

Zmienne

Dlaczego zmienne są ważne? Zmienne pozwalają: zapisywać dane w programie, wielokrotnie ich używać, budować bardziej złożone obliczenia, pisać czytelny kod.

Prosty przykład użycia

```
imie = "Anna"
wiek = 20

print("Imię:", imie)
print("Wiek:", wiek)
```

Dzięki zmiennym program staje się elastyczny - łatwo zmienić dane bez zmieniania całego kodu.

Najważniejsza idea: Zmienne to sposób na nadawanie nazw danym i przechowywanie ich w programie, aby móc je później wygodnie wykorzystać.

Podstawowe typy danych

W Pythonie każda wartość ma swój **typ danych**, który określa, czym ta wartość jest i jakie operacje można na niej wykonywać. Najważniejsze typy na początku nauki to: liczby, napisy oraz wartości logiczne.

- `int` → liczby całkowite
- `float` → liczby z ułamkiem
- `str` → tekst
- `bool` → prawda / fałsz

Każdy typ danych zachowuje się inaczej i jest używany do innych zadań, dlatego ich rozróżnianie jest podstawą programowania w Pythonie.

Liczby (`int`, `float`)

Python rozróżnia dwa podstawowe typy liczb:

`int` – liczby całkowite

Są to liczby bez części ułamkowej.

```
wiek = 20
liczba_dni = 365
```

Możemy na nich wykonywać standardowe działania:

```
print(2 + 3)
print(10 * 5)
```

`float` – liczby zmiennoprzecinkowe

Są to liczby z częścią ułamkową.

```
pi = 3.14
temperatura = 21.5
print(5.0/ 2)
```

Wynik:

```
2.5
```

Napisy (`str`)

Napisy (*stringi*) to dane tekstowe zapisane w cudzysłowie.

```
imie = "Anna"
miasto = "Warszawa"
```

Można je wyświetlać i łączyć:

```
print("Cześć " + imie)
```

Wynik:

```
Cześć Anna
```

Napisy mogą zawierać litery, cyfry i znaki specjalne - są po prostu tekstem.

Wartości logiczne (`bool`)

Wartości logiczne

Logika rządzi się własnymi prawami, które pozwalają precyzyjnie wyznaczać wartość logiczną zdań, czyli ustalać, czy zdanie złożone z prostszych zdań jest prawdziwe, czy fałszywe. Ponieważ operujemy tu tylko dwiema wartościami i można opisywać ich zależności za pomocą równań, zasady logiki określa się mianem algebry dwuwartościowej, częściej nazywanej **algebrą Boole'a**, od nazwiska osoby, która jako pierwsza sformalizowała ten rachunek.

Dwie wartości logiczne - prawdę i fałsz - często oznacza się symbolami 1 i 0.
W Pythonie wartość prawdy można zapisać zarówno jako liczbę `1`, jak i za pomocą specjalnego obiektu `True`, natomiast fałsz jako `0` lub `False`. Te specjalne obiekty, nazywane po angielsku *booleans* (czyli wartościami Boole'a), nie są konieczne do działania rachunku logicznego; wprowadzono je po to, by wyraźnie odróżnić sytuacje, w których posługujemy się algebrą dwuwartościową, od tych, w których używamy zwykłych wartości liczbowych.

Operacje logiczne

Koniunkcja: Jeśli mamy dwa zdania, a oraz b, ich koniunkcją nazywamy wyrażenie, które jest prawdziwe tylko wtedy, gdy oba zdania są prawdziwe. W matematyce zapisuje się ją jako $a \wedge b$, natomiast w Pythonie odpowiada jej operator `and`.

Działanie operatora koniunkcji można przedstawić, rozważając wszystkie możliwe pary wartości logicznych:

a	b	a and b
True	True	True
True	False	False
False	True	False
False	False	False

Operacje logiczne

Alternatywa: Jeśli mamy dwa zdania, a oraz b, ich alternatywą nazywamy wyrażenie, które jest prawdziwe wtedy, gdy przynajmniej jedno z tych zdań jest prawdziwe. Fałszywe jest tylko wtedy, gdy oba zdania są fałszywe. W matematyce zapisuje się ją jako $a \vee b$, natomiast w Pythonie odpowiada jej operator `or`.

Działanie operatora alternatywy można przedstawić, rozważając wszystkie możliwe pary wartości logicznych:

a	b	a or b
True	True	True
True	False	True
False	True	True
False	False	False

Operacje logiczne

Zaprzeczenie: Jeśli mamy zdanie a, jego zaprzeczeniem nazywamy wyrażenie, które ma wartość przeciwną do zdania wyjściowego: jest prawdziwe, gdy a jest fałszywe, i fałszywe, gdy a jest prawdziwe. W matematyce zapisuje się je jako $\neg a$, natomiast w Pythonie odpowiada mu operator `not`.

Działanie operatora alternatywy można przedstawić, rozważając wszystkie możliwe pary wartości logicznych:

a	not a
True	False
False	True

Wartość logiczna wyrażeń w Pythonie

Gdy używamy operatorów porównania (`==`, `!=`, `<=`, `>=`, `<`, `>`), wynikiem jest jedna z dwóch wartości logicznych: `True` albo `False`. Większość innych wyrażeń nie ma jednak wartości logicznej w sensie matematycznym — na przykład napis `"Hello, World"` nie jest ani prawdziwy, ani fałszywy w sensie logiki.

Mimo to w Pythonie - podobnie jak w wielu innych językach programowania - każdy obiekt ma przypisaną wartość logiczną określaną według ustalonych reguł. Dzięki temu dowolne wyrażenie może zostać użyte jako warunek w instrukcjach sterujących, takich jak `if` czy `while`, ponieważ każde wyrażenie zwraca jakiś obiekt, a Python potrafi ocenić go jako prawdziwy lub fałszywy.

Wartość logiczna wyrażeń w Pythonie

W przypadku obiektów, które nie są po prostu `True` ani `False`, interpretacja ich wartości logicznej opiera się na kilku prostych zasadach:

- **Liczby**: liczba `0` jest traktowana jako fałsz, a każda inna liczba jako prawda.
- **Sekwencje i kolekcje** (np. napisy, listy, zbiory): obiekty puste, czyli o długości `0`, są fałszywe; niepuste są prawdziwe.
- **Pozostałe obiekty**: domyślnie są uznawane za prawdziwe, chyba że ich twórca specjalnie zdefiniował inne zachowanie.

Obliczanie wartości wyrażeń logicznych w Pythonie

W wyrażeniach zawierających operatory `and` lub `or` Python często może ustalić wynik całego wyrażenia bez obliczania drugiego operandu. Mechanizm ten nazywa się **wykonaniem warunkowym** (*short-circuiting*).

Działa to następująco:

- W przypadku operatora `and`, jeśli pierwszy operand jest fałszywy, drugi nie jest już obliczany, ponieważ całe wyrażenie i tak będzie fałszywe.
- W przypadku operatora `or`, jeśli pierwszy operand jest prawdziwy, drugi nie jest obliczany, ponieważ wynik całego wyrażenia jest już przesądzony.

Obliczanie wartości wyrażeń logicznych w Pythonie

W tabelach prawdy dla `and` i `or` rozpatruje się zwykle operacje na wartościach logicznych `True` i `False`. W Pythonie jednak argumentami tych operatorów mogą być dowolne obiekty, którym — zgodnie z omówionymi wcześniej regułami — można przypisać wartość logiczną. W takim przypadku wynik działania operatorów logicznych nie musi być wartością `True` ani `False`.

Operatory `and` i `or` działają wtedy według następujących zasad:

- `x and y` — jeśli `x` jest fałszywe, zwracane jest `x`; w przeciwnym razie zwracane jest `y`.
- `x or y` — jeśli `x` jest prawdziwe, zwracane jest `x`; w przeciwnym razie zwracane jest `y`.

Obliczanie wartości wyrażeń logicznych w Pythonie

Dla wartości `True` i `False` reguły te dają dokładnie wyniki znane z tabel prawdy. W przypadku innych obiektów pozwalają jednak budować wygodne „pseudo-zdania logiczne”. Przykładowo, jeśli chcemy użyć wartości domyślnej, gdy użytkownik poda pusty napis, można napisać:

```
print(napis or "(podałeś pusty napis)")
```

Jeśli `napis` jest pusty (a więc fałszywy), użyty zostanie drugi operand.

Obliczanie wartości wyrażeń logicznych w Pythonie

Powyższe zasady można ująć jeszcze ogólniej:

1. Najpierw obliczany jest lewy operand.
2. Jeśli jego wartość logiczna nie pozwala jeszcze rozstrzygnąć wyniku całego wyrażenia, obliczany jest prawy operand.
3. Wynikiem jest **ostatni obliczony operand**, czyli ten, który decyduje o wartości logicznej całego wyrażenia.

To właśnie połączenie wykonania warunkowego i zwracania operandów, a nie tylko wartości `True` lub `False`, sprawia, że operatory logiczne w Pythonie są szczególnie użyteczne.

Wartości logiczne (`bool`)

Typ logiczny ma tylko dwie możliwe wartości:

- `True` (prawda)
- `False` (fałsz)

```
czy_dorosly = True
czy_znajomy = False
```

Są one często wynikiem porównań:

```
print(5 > 3)
print(2 == 10)
```

Wynik:

```
True
False
```

Operatory i obliczenia

W Pythonie operatory to symbole, które pozwalają wykonywać działania na danych - przede wszystkim obliczenia matematyczne oraz porównania.

Operatory arytmetyczne (obliczenia): Najprostsze operacje to działania matematyczne.

```
2 + 3
7 * 4
10 / 2
```

Podstawowe operatory:

- `+` dodawanie
- `-` odejmowanie
- `*` mnożenie
- `/` dzielenie

Przykład:

```
a = 10
b = 3


print(a + b)
print(a * b)
print(a / b)
```

Operatory i obliczenia

Inne ważne operatory matematyczne

Python ma też dodatkowe operacje:

`//` dzielenie całkowite

`%` reszta z dzielenia

`**` potęgowanie

```
print(10 // 3)
print(10 % 3)
print(2 ** 3)
```

Operatory i obliczenia

Operatory porównania służą do sprawdzania relacji między wartościami. Wynikiem zawsze jest wartość logiczna: `True` lub `False`.

Najważniejsze operatory:

- `==` równe
- `!=` różne
- `>` większe
- `<` mniejsze
- `>=` większe lub równe
- `<=` mniejsze lub równe

Przykłady:

```
print(5 == 5)
print(5 != 3)
print(10 > 2)
print(4 < 1)
```

Wynik:

```
True
True
True
False
```

Operatory i obliczenia

Do czego to się przydaje? Operatory porównania są podstawą podejmowania decyzji w programie:

```
wiek = 18

print(wiek >= 18)
```

Wynik:

```
True
```

Najważniejsza idea:

- operatory arytmetyczne → liczą
- operatory porównania → sprawdzają
- wynik porównań → `True` lub `False`

Dzięki nim program może nie tylko liczyć, ale też „decydować”.

Instrukcja warunkowa `if`

Instrukcja `if` pozwala programowi **podejmować decyzje**. Dzięki niej kod może wykonywać się tylko wtedy, gdy spełniony jest określony warunek.

```
if wiek >= 18:
    print("pełnoletni")
```

W tym przykładzie program sprawdza, czy zmienna `wiek` jest większa lub równa 18. Jeśli tak - wykonuje instrukcję wypisania tekstu.

Jak działa `if`? Instrukcja `if` składa się z dwóch części:

```
if warunek:
    instrukcje
```

- **warunek** → wyrażenie, które daje wynik `True` lub `False`
- **instrukcje** → kod wykonywany tylko wtedy, gdy warunek jest prawdziwy

Przykład w praktyce

wiek = 20

Instrukcja warunkowa `if`

Przykład w praktyce

```
wiek = 20
if wiek >= 18:
    print("Możesz wejść")
```

Jeśli wiek wynosi 20, warunek jest spełniony i program wyświetli komunikat.

Co jeśli warunek nie jest spełniony?

Jeśli warunek jest fałszywy, blok kodu nie zostaje wykonany:

```
wiek = 15
if wiek >= 18:
    print("Możesz wejść")
```

W tym przypadku nic się nie wyświetli.

Instrukcja warunkowa `if`

Wcięcia – bardzo ważne!

W Pythonie **wcięcia (indentation)** są obowiązkowe i mają znaczenie składniowe.

```
if wiek >= 18:
    print("pełnoletni")
```

Linia z `print()` musi być przesunięta w prawo - to oznacza, że należy do bloku `if`.

Co się stanie bez wcięcia?

```
if wiek >= 18:
print("pełnoletni")  # błąd
```

Python zgłosi błąd, ponieważ nie wie, które instrukcje należą do warunku.

Rozszerzenie: `if` / `else`

Często chcemy obsłużyć także sytuację przeciwną:

```
wiek = 16
if wiek >= 18:
    print("pełnoletni")
else:
    print("niepełnoletni")
```

Najważniejsza idea

- `if` pozwala podejmować decyzje
- warunek musi być `True` lub `False`
- wcięcia określają, co należy do warunku
- blok kodu wykonuje się tylko, gdy warunek jest spełniony

Dzięki `if` programy stają się „inteligentne” i reagują na różne sytuacje.

Pętle

Pętle pozwalają na wielokrotne wykonywanie tego samego fragmentu kodu. Dzięki nim nie trzeba powtarzać instrukcji ręcznie - program robi to automatycznie.

Na początku najważniejsza jest pętla `for`, a dopiero później `while`.

Pętla `for`

Pętla `for` służy do wykonywania kodu określoną liczbę razy lub dla kolejnych elementów. Najprostszy przykład:

```
for i in range(5):
    print(i)
```

Jak to działa?

```
for i in range(5):
```

- `range(5)` oznacza liczby: 0, 1, 2, 3, 4
- `i` to zmienna, która przyjmuje kolejne wartości
- pętla wykonuje się 5 razy

Wynik programu

```
0
1
2
3
4
```

Pętla `for`

Co to znaczy „powtórzenie kodu”?

Bez pętli musielibyśmy pisać:

```
print(0)
print(1)
print(2)
print(3)
print(4)
```

Pętla robi to automatycznie i krócej.

Zakres liczb

`range()` – podstawowe zastosowanie

`range(n)` generuje liczby od 0 do n-1:

```
range(3)  # 0, 1, 2
```

Możemy też sterować zakresem:

```
for i in range(2, 6):
    print(i)
```

Wynik:

```
2
3
4
5
```

Pętla `while` (dopiero później)

Pętla `while` działa trochę inaczej — wykonuje się dopóki warunek jest prawdziwy.

```
x = 0

while x < 5:
    print(x)
    x = x + 1
```

Różnica między `for` i `while`

`for` → wiemy ile razy coś powtórzyć

`while` → powtarzamy, dopóki warunek jest spełniony

Najważniejsza idea:

- pętle służą do powtarzania kodu
- `for` jest najprostsza i najczęściej używana na start
- `range()` pomaga generować kolejne liczby
- bez pętli kod szybko staje się długi i nieczytelny

Pętle to jeden z fundamentów programowania — pozwalają komputerowi robić „pracę za nas”.