

Lekcja

Temat: Wstęp do programowania w Python

Historia języka programowania Python

Python został stworzony przez **Guido van Rossum** w latach 1989-1991 w Holandii. Nazwa nie pochodzi od węża, lecz od brytyjskiej grupy komediowej **Monty Python**, której Guido był fanem.

Prace nad nim rozpoczęły się pod koniec **1989** roku w **Centrum Matematyki i Informatyki (CWI)** w Amsterdamie, gdzie van Rossum szukał projektu, który mógłby zająć go w okresie świątecznym. **Python miał być następcą języka ABC**, który również rozwijano w CWI, ale van Rossum chciał stworzyć coś bardziej uniwersalnego i przyjaznego dla programistów.

Kluczowe etapy rozwoju:

- **1991: Premiera pierwszej wersji** – 20 lutego 1991 roku ukazała się publiczna wersja 0.9.0. Python był od początku projektem open source, co pozwoliło na szybki rozwój dzięki wkładowi społeczności.
- **Lata 90.: Rozwój w CWI i CNRI** – Do 1995 roku van Rossum rozwijał Pythona w CWI, a następnie przeniósł się do Corporation for National Research Initiatives (CNRI) w USA, gdzie wydał wersje do 1.6 włącznie. W tym okresie język zyskał kompatybilność z licencją GPL.
- **2000: Python 2.0 i BeOpen.com** – Van Rossum i zespół przenieśli się do BeOpen.com, gdzie wydano Pythona 2.0. Wkrótce potem powstała Python Software Foundation (PSF), która do dziś zarządza rozwojem języka.
- **2008: Python 3.0** – Wprowadzono znaczące zmiany, takie jak lepsza obsługa Unicode, co jednak spowodowało niekompatybilność z Pythonem 2.x. Przejście na "trójkę" trwało lata, a wsparcie dla Pythona 2 zakończyło się w 2020 roku.
- **2018: Rezygnacja van Rossuma** – Guido van Rossum, znany jako "Benevolent Dictator for Life" (BDFL), zrezygnował z roli lidera. Od tego czasu rozwój nadzoruje Steering Council wybrany przez społeczność.

Instalacja interpretera

Windows

1. Wejdź na <https://www.python.org/downloads/windows/>
2. Pobierz **Windows installer (64-bit)**
3. Zainstaluj (opcje standardowe)
4. Wykonaj polecenie: **py -3 -version**. Jeśli zainstalowałeś, poprawnie powinieneś otrzymać komunikat: **Python 3.14.2**
5. Kliknij **Win + I**
6. Kliknij **Aplikacje**, wybierz **Zaawansowane ustawienia aplikacji**. Po czym kliknij **Aliasy wykonywania aplikacji**
7. Zobaczysz listę. Przewiń w dół i znajdź:
 - python.exe
 - python3.exe

Przy nich są przełączniki (ON / OFF). Wyłącz dwie opcje python.exe, python3.exe

Uruchomienie skryptu

Wykonaj polecenie, w katalogu skryptu Python:

python nazwa_skrypu.py

Rozszerzony przykład Hello World

```
import sys
print("Hello, World!")

print("Witaj w świecie Pythona!")
print(f"Używasz Pythona w wersji: {sys.version}")

imie = input("Jak masz na imię? ")
print(f"Cześć {imie}! Miło Cię poznać!")
```

Podstawowe typy komentarzy

Komentarz jednowierszowy - zaczyna się od #

```
# To jest komentarz jednowierszowy
x = 10 # To też jest komentarz - po kodzie
```

Komentarz wielowierszowy - używamy trzech cudzysłówów """ lub ''''

To jest komentarz wielowierszowy.
Może zawierać wiele linii tekstu.
Często używany na początku pliku
lub do dokumentowania funkcji.

'''

To też jest poprawny
komentarz wielowierszowy.
Używaj zgodnie z konwencją projektu.

def oblicz_srednia(liczby):

'''

Funkcja oblicza średnią arytmetyczną z listy liczb.

Args:

liczby (list): Lista liczb całkowitych lub zmiennoprzecinkowych

Returns:

float: Średnia arytmetyczna

'''

return sum(liczby) / len(liczby) if liczby else 0

Oznaczanie TODO i FIXME

```
# TODO: Dodać obsługę błędów dla pustej listy
# FIXME: Naprawić wyciek pamięci przy dużych danych
# HACK: Tymczasowe rozwiązanie - wymaga refaktoryzacji
# NOTE: Ważna uwaga dotycząca wydajności
# OPTIMIZE: Można przyspieszyć przez caching

def przetworz_dane(dane):
    # TODO: Implementować walidację danych wejściowych
    # FIXME: Dla pustych danych zwracamy None - do poprawy
    if not dane:
        return None # FIXME: Powinien być wyjątek

    # ... reszta kodu ...
```

Wcięcia Pythonie:

```
if x > 0:
    print(x)
```

- **Dwukropiek** : mówi: zaraz będzie blok
- **Wcięcie** mówi, co należy do tego bloku

Przykład błędu

```
x = 5
if x > 0:
    print("x jest dodatnie") # Błąd: brak wcięcia
```

Wynik:

IndentationError: expected an indented block

Poprawnie

```
x = 5
if x > 0:
    print("x jest dodatnie") # wcięcie = 4 spacje (standard)
```

1. Typy numeryczne int - liczby całkowite

```
# Przykłady
liczba_calkowita = 42
ujemna = -10
duza_liczba = 1_000_000 # podkreślniki dla czytelności, równoważne 1000000
binarna = 0b1010 # 10 w systemie dziesiętnym
""" od prawej liczymy
    • 1 × 23 = 8
    • 0 × 22 = 0
    • 1 × 21 = 2
    • 0 × 20 = 0
```

```
"""
```

```
szesnastkowa = 0xFF # 255

# Zastosowania: liczniki, ID, indeksy, obliczenia finansowe
cena_produktu = 299
ilosc_sztuk = 5
id_uzytkownika = 12345
```

float - liczby zmiennoprzecinkowe

```
# Przykłady
pi = 3.14159
temperatura = -5.5
duza_liczba = 1.23e6 #  $1.23 \times 10^6 = 1230000.0$ 
```

```
# Zastosowania: obliczenia naukowe, finanse, pomiary
cena_netto = 19.99
waga = 2.5
srednia = 4.75
```

complex - liczby zespolone

```
# Przykłady
liczba_zespolona = 3 + 4j
inna = complex(2, -3) #  $2 - 3j$ 
```

```
# Zastosowania: inżynieria, fizyka, przetwarzanie sygnałów
impedancja = 50 + 100j # zamiast i używa się j (zgodnie z konwencją inż elektryków).
```

2. Typy tekstowe str - ciągi znaków

```
# Przykłady
imie = "Anna"
nazwisko = 'Kowalska'
wielolinijkowy = """To jest
wielolinijkowy
tekst"""
f_string = f"Witaj {imie}!" # f-string (Python 3.6+)
```

```
# Zastosowania: komunikaty, dane tekstowe, parsowanie
email = "user@example.com"
wiadomosc = "Dziękujemy za zakupy!"
sciezka = "C:/Users/Dokumenty"
```

3. Typy sekwencyjne list - listy (mutable)

```
# Przykłady
lista_liczb = [1, 2, 3, 4, 5]
lista_mieszana = [1, "dwa", 3.0, True]
lista_2d = [[1, 2], [3, 4]]
```

```
# Zastosowania: kolekcje elementów, wyniki zapytań, tymczasowe przechowywanie
```

```
zakupy = ["mleko", "chleb", "jajka"]
wyniki_testu = [85, 92, 78, 90]
```

tuple - krotki (immutable)

Tuple (krotka) to uporządkowana kolekcja danych, której **nie da się zmienić po utworzeniu**, idealna do bezpiecznych, stałych struktur.

```
# Przykłady
wspolrzedne = (10, 20)
kolory_rgb = (255, 128, 0)
pojedyńczy_element = (5,) # UWAGA: przecinek jest konieczny!
```

```
# Zastosowania: stałe zbiory danych, zwracanie wielu wartości z funkcji
wymiary = (1920, 1080) # rozdzielcość
data_urodzenia = (1990, 5, 15)
```

range - zakresy

range to **typ sekwencyjny**, który reprezentuje **zakres liczb. Nie tworzy od razu listy liczb**. Generuje je „w locie” (jest wydajny pamięciowo). Najczęściej używany w pętlach for

```
# Przykłady
zakres1 = range(5)      # 0, 1, 2, 3, 4
zakres2 = range(1, 6)    # 1, 2, 3, 4, 5
zakres3 = range(0, 10, 2) # 0, 2, 4, 6, 8

# Zastosowania: iteracje w pętlach, generowanie indeksów
for i in range(3):
    print(f"Powtóżenie {i}")

indeksy = list(range(len(["a", "b", "c"]))) # [0, 1, 2]
```

4. Typy mapujące

dict - słowniki

Mapowanie = przyporządkowanie wartości do **unikalnego klucza**. W Pythonie typ mapujący to **dict**. Każdy klucz **może wystąpić tylko raz**. Wartości mogą się powtarzać i mogą być **dowolnego typu**

```
# Przykłady
student = {
    "imie": "Jan",
    "nazwisko": "Kowalski",
    "wiek": 21,
    "oceny": [4.5, 5.0, 4.0]
}

pusty_słownik = {}
słownik_z_kluczami = dict(a=1, b=2)

# Zastosowania: konfiguracje, bazy danych w pamięci, mapowania
konfiguracja = {
    "host": "localhost",
    "port": 8080,
    "debug": True
```

```
}
```

```
ksiazka_telefoniczna = {
    "Anna": "123-456-789",
    "Jan": "987-654-321"
}
```

5. Typy zbiorów

set - zbiory (mutable, unikalne elementy)

```
# Przykłady
zbior_liczb = {1, 2, 3, 3, 2} # {1, 2, 3}
zbior_mieszany = {"a", 1, 3.14, True}

# Zastosowania: usuwanie duplikatów, operacje matematyczne na zbiorach
unikalne_slowa = set(["kot", "pies", "kot", "ptak"]) # {"kot", "pies", "ptak"}
```

frozenset - zamrożone zbiory (immutable)

frozenset to: **niemodyfikowalny (immutable) zbiór**. Działa podobnie do **set**. Przechowuje **unikalne elementy** oraz podobnie **działa jak tuple**. Frozenset **nie można go zmieniać**.

Różnica: set vs frozenset

set – można zmieniać

```
s = {1, 2, 3}
s.add(4)    # OK
s.remove(2) # OK
```

frozenset – NIE można zmieniać

```
fs = frozenset([1, 2, 3])

fs.add(4)    # AttributeError
fs.remove(2) # AttributeError
```

```
# Przykłady
zamrozony_zbior = frozenset([1, 2, 3, 3, 2]) # frozenset({1, 2, 3})

# Zastosowania: klucze w słownikach, stałe zbiory
klucze_słownika = {
    frozenset([1, 2]): "wartość",
    frozenset(["a", "b"]): "inna wartość"
}
```

6. Typy logiczne

bool - wartości logiczne

```
# Przykłady
prawda = True
falsz = False
wynik_porownania = 10 > 5 # True

# Zastosowania: warunki, flagi, stany
jest_zalogowany = True
ma_dostep = False
```

7. Typy binarne bytes - bajty (immutable)

bytes to: **niemodyfikowalna (immutable) sekwencja bajtów**. Każdy bajt ma wartość **0-255**. Czyli nie tekst, nie liczby, **surowe dane binarne**

```
# Przykłady
bajty = b'hello'
bajty_od_listy = bytes([65, 66, 67]) # b'ABC'

# Zastosowania: dane binarne, pliki, sieć
dane_plikowe = b'\x89PNG\r\n\x1a\n' # nagłówek PNG
```

bytearray - tablice bajtów (mutable)

bytearray to: **modyfikowalna (mutable) sekwencja bajtów**. Każdy element: **liczba 0-255** („bajty do edycji”)

```
# Przykłady
modyfikowalne_bajty = bytearray(b'hello')
modyfikowalne_bajty[0] = 72 # H zamiast h -> b'Hello'

# Zastosowania: modyfikacja danych binarnych, bufore
bufor = bytearray(1024) # bufor 1KB
```

memoryview - widok pamięci

memoryview to: **widok na istniejący blok pamięci, w którym nie można kopiowania danych**. Działa na: bytes, bytearray, array, numpy itd. To **nie są dane** – to **okno na dane**.

```
# Przykłady
bajty = b'abcdef'
widok = memoryview(bajty)
print(widok[1:4].tobytes()) # b'bcd'

# Zastosowania: efektywna praca z dużymi danymi binarnymi
```

8. Specjalne typy NoneType - wartość None

None to: **specjalna wartość oznaczająca „brak wartości”**

```
x = None  
print(type(x)) # <class 'NoneType'>
```

None to nie 0, nie "" i nie False. Zastosowanie, kiedy coś **nie istnieje**, coś **jeszcze nie zostało ustawione**, funkcja **nic nie zwraca**, operacja **nie ma sensu**

```
# Przykłady  
brak_wartosci = None  
domyslna = None  
  
# Zastosowania: reprezentacja braku wartości, domyślne argumenty  
def znajdz_uzytownika(id):  
    if id in baza_danych:  
        return baza_danych[id]  
    return None
```

9. Typy z modułów

Decimal - liczby dziesiętne (dokładne)

```
from decimal import Decimal  
  
# Przykłady  
cena = Decimal('19.99')  
podatek = Decimal('0.23')  
kwota = cena * (1 + podatek)  
  
# Zastosowania: obliczenia finansowe (unikamy błędów zaokrągleń)
```

Fraction - ułamki zwykłe

```
from fractions import Fraction  
  
# Przykłady  
polowa = Fraction(1, 2)  
trzecia = Fraction(1, 3)  
suma = polowa + trzecia # Fraction(5, 6)  
  
# Zastosowania: obliczenia symboliczne, matematyka
```

Lekcja

Temat: Operatory, wyrażenia, if – elif – else, Pętle: For (z range i iterable), while; break, continue, else w pętlach w Python

1. Operatory Arytmetyczne

Używane do wykonywania podstawowych operacji matematycznych.

```
# Przykłady operatorów arytmetycznych
a = 10
b = 3

print(f"a = {a}, b = {b}")
print(f"Dodawanie: a + b = {a + b}")           # 13
print(f"Odejmowanie: a - b = {a - b}")         # 7
print(f"Mnożenie: a * b = {a * b}")            # 30
print(f"Dzielenie: a / b = {a / b}")           # 3.3333333333333335
print(f"Dzielenie całkowite: a // b = {a // b}") # 3
print(f"Reszta z dzielenia: a % b = {a % b}")  # 1
print(f"Potęgowanie: a ** b = {a ** b}")        # 1000

# Operatory przypisania z operacją
c = 5
c += 2 # równoważne: c = c + 2
print(f"c += 2: {c}") # 7

c *= 3 # c = c * 3
print(f"c *= 3: {c}") # 21
```

2. Operatory Porównania

Zwracają wartości logiczne (True/False) porównując wartości.

```
x = 10
y = 5
z = 10

print(f"x = {x}, y = {y}, z = {z}")
print(f"Równe: x == y -> {x == y}")          # False
print(f"Równe: x == z -> {x == z}")          # True
print(f"Różne: x != y -> {x != y}")          # True
print(f"Większe: x > y -> {x > y}")          # True
print(f"Menniejsze: x < y -> {x < y}")       # False
print(f"Większe lub równe: x >= z -> {x >= z}") # True
print(f"Menniejsze lub równe: y <= x -> {y <= x}") # True

# Porównania łańcuchów
print(f"'abc' == 'abc' -> {'abc' == 'abc'}") # True
print(f"'abc' != 'def' -> {'abc' != 'def'}") # True
"""

Każdy znak ma swój numer Unicode:
ord('a') # 97
```

```
    ord('b') # 98
"""
print(f"'a' < 'b' -> {'a' < 'b'}")           # True (porównanie leksykograficzne)
```

3. Operatory Logiczne

Używane do łączenia wyrażeń logicznych.

```
# Podstawowe operatory Logiczne
a = True
b = False

print(f"a = {a}, b = {b}")

Tabela prawdy - and
Operator and zwraca True tylko wtedy, gdy oba warunki są True.
a           b           a and b
True        True        True
True        False       False
False       True        False
False       False       False
"""

print(f"AND: a and b -> {a and b}")      # False
"""

Tabela prawdy - or
Operator or zwraca True, jeśli chociaż jeden warunek jest True.
a           b           a or b
True        True        True
True        False       True
False       True        True
False       False       False
"""

print(f"OR: a or b -> {a or b}")          # True
print(f"NOT: not a -> {not a}")            # False

# Praktyczne zastosowanie
wiek = 25
zarobki = 5000

# Sprawdzanie wielu warunków
if wiek >= 18 and zarobki > 3000:
    print("Możesz wziąć kredyt") # Wykona się

# Sprawdzanie czy wartość jest w zakresie
if 18 <= wiek <= 65:
    print("W wieku produkcyjnym") # Wykona się

# łączenie operatorów Logicznych
x = 10
```

```

if x > 5 and x < 15 and x != 12:
    print("x spełnia wszystkie warunki") # Wykona się

```

4. Operatory Bitowe

```

# Operacje bitowe
"""

System binarny to dzielenie przez 2:
10 / 2 = 5 reszta 0
5 / 2 = 2 reszta 1
2 / 2 = 1 reszta 0
1 / 2 = 0 reszta 1
"""

a = 10 # 1010 w systemie binarnym bin(10), bin(10)[2:], format(10, 'b')
b = 6 # 0110 w systemie binarnym

print(f"a = {a} (bin: {bin(a)}), b = {b} (bin: {bin(b)})")
print(f"AND bitowe: a & b = {a & b}") # 2 (0010)
print(f"OR bitowe: a | b = {a | b}") # 14 (1110)
print(f"XOR bitowe: a ^ b = {a ^ b}") # 12 (1100)
print(f"NOT bitowe: ~a = {~a}") # -11 (dopełnienie)
"""

10 (dziesiątkie) = 1010 (binarnie)
Bo 1*8 + 0*4 + 1*2 + 0*1 = 10
<< 1 oznacza:
    ↪ przesuwamy wszystkie bity o jedno miejsce w lewo
    ↪ na końcu dopisujemy 0
10100
1*16 + 0*8 + 1*4 + 0*2 + 0*1 = 20

"""

print(f"Przesunięcie w lewo: a << 1 = {a << 1}") # 20 (10100)

"""

10 (dziesiątkie) = 1010 (binarnie)
Bo 1*8 + 0*4 + 1*2 + 0*1 = 10
>> 1 oznacza:
    ↪ przesuwamy wszystkie bity o jedno miejsce w prawo
        1010 (10)
    >> 1
        -----
        0101 (5)
    0*8 + 1*4 + 0*2 + 1*1 = 5

"""

print(f"Przesunięcie w prawo: a >> 1 = {a >> 1}") # 5 (0101)

# Praktyczne zastosowanie - sprawdzanie parzystości
"""

print(f"7 = {bin(7)[2:]} ") # 111
print(f"1 = {bin(1)[2:]} ") # 1 czyli 001
Operacja AND (&)
    Bit A   Bit B   Wynik
    1       1       1

```

```

1      0      0
0      1      0
0      0      0
  111  (7)
& 001  (1)
-----
  001
...
liczba = 7
if liczba & 1: # Ostatni bit = 1 dla liczb nieparzystych
    print(f"{liczba} jest nieparzysta")
else:
    print(f"{liczba} jest parzysta")

```

5. Priorytety Operatorów

Kolejność wykonywania działań.

```

# Przykłady priorytetów
wynik = 2 + 3 * 4 # Mnożenie ma wyższy priorytet
print(f"2 + 3 * 4 = {wynik}") # 14, nie 20

# Używanie nawiasów do zmiany kolejności
wynik = (2 + 3) * 4
print(f"(2 + 3) * 4 = {wynik}") # 20

# Przykład złożony
wynik = 5 + 2 ** 3 * 4 / 2 - 1
print(f"5 + 2 ** 3 * 4 / 2 - 1 = {wynik}")
# Kolejność: potęgowanie -> mnożenie/dzielenie -> dodawanie/odejmowanie

# Tabela priorytetów (od najwyższych do najniższych):
# 1. ** (potęgowanie)
# 2. +x, -x, ~x (jednoargumentowe)
# 3. *, /, //, %
# 4. +, - (dwuargumentowe)
# 5. <<, >> (przesunięcia bitowe)
# 6. & (AND bitowe)
# 7. ^ (XOR bitowe)
# 8. | (OR bitowe)
# 9. ==, !=, >, >=, <, <= (porównania)
# 10. not (Logiczne NOT)
# 11. and (Logiczne AND)
# 12. or (Logiczne OR)

```

6. Wyrażenia Warunkowe

Instrukcje warunkowe i wyrażenia warunkowe (ternarne).

```

# Podstawowa instrukcja if-elif-else
liczba = 15

if liczba > 0:
    print("Liczba dodatnia")
elif liczba < 0:
    print("Liczba ujemna")
else:
    print("Zero")

# Zagnieżdżone warunki
wiek = 20
ma_prawo_jazdy = True

if wiek >= 18:
    if ma_prawo_jazdy:
        print("Możesz prowadzić samochód")
    else:
        print("Jesteś pełnoletni, ale nie masz prawa jazdy")
else:
    print("Jesteś niepełnoletni")

# Wyrażenie warunkowe (ternarne)
x = 10
y = 20

# Składnia: wartość_if_true if warunek else wartość_if_false
mniejsza = x if x < y else y
print(f"mniejsza liczba: {mniejsza}") # 10

# Praktyczny przykład - kategoria wiekowa
wiek = 25
kategoria = "dziecko" if wiek < 18 else ("dorosły" if wiek < 65 else "senior")
print(f"Wiek {wiek}: {kategoria}") # dorosły

# Operator walrus (:=) - dostępny od Python 3.8
# Pozwala na przypisanie wartości w ramach wyrażenia
if (n := len([1, 2, 3, 4])) > 3:
    print(f"Lista ma {n} elementów, czyli więcej niż 3")

```

if - elif - else

Używasz, gdy masz kilka możliwych warunków.

Przykład 1: Oceny szkolne

```

# System oceniania
punkty = 85

if punkty >= 90:
    ocena = "A"
    print("Celujący!")
elif punkty >= 80:
    ocena = "B"
    print("Bardzo dobry!")
elif punkty >= 70:

```

```

ocena = "C"
print("Dobry!")
elif punkty >= 60:
    ocena = "D"
    print("Dostateczny!")
else:
    ocena = "F"
    print("Niedostateczny!")

print(f"Za {punkty} punktów otrzymujesz ocenę: {ocena}")

```

Przykład 2: Weryfikacja wieku

```

wiek = int(input("Podaj swój wiek: "))

if wiek < 0:
    print("Wiek nie może być ujemny!")
elif wiek < 13:
    print("Jesteś dzieckiem")
elif wiek < 18:
    print("Jesteś nastolatkiem")
elif wiek < 65:
    print("Jesteś osobą dorosłą")
else:
    print("Jesteś seniorem")

# Sprawdzenie dodatkowych uprawnień
if wiek >= 18:
    print("Jesteś pełnoletni(a)")
    if wiek >= 21:
        print("Możesz legalnie pić alkohol w USA")
    if wiek >= 65:
        print("Przysługuje Ci emerytura")

```

2. Zagnieżdżone Warunki

Zagnieżdżone `if` pozwalają na tworzenie bardziej złożonych struktur decyzyjnych.

Przykład 1: System logowania

```

# Symulacja systemu Logowania
poprawny_login = "admin"
poprawne_haslo = "secret123"
login_wprowadzony = "admin"
haslo_wprowadzone = "secret123"
kod_2fa = "7890"
if login_wprowadzony == poprawny_login:
    print("Login poprawny")

```

```

if haslo_wprowadzone == poprawne_haslo:
    print("Hasło poprawne")

    if input("Podaj kod 2FA: ") == kod_2fa:
        print("Logowanie udane! Witamy w systemie.")
    else:
        print("Błędny kod 2FA!")
else:
    print("Błędne hasło!")
else:
    print("Błędny login!")

```

3. Ternary Operator (Operator Trójargumentowy)

Skrócona składnia dla prostych warunków.

Składnia: wartość_if_true if warunek else wartość_if_false

Przykład 1: Podstawowe użycie

```

# Tradycyjne podejście
liczba = 10
if liczba % 2 == 0:
    parzystosc = "parzysta"
else:
    parzystosc = "nieparzysta"

# To samo z ternary operator
liczba = 10
parzystosc = "parzysta" if liczba % 2 == 0 else "nieparzysta"
print(f'Liczba {liczba} jest {parzystosc}')

```

1. Pętla for z range()

```

# range(stop) - generuje liczby od 0 do stop-1
print("range(5):")
for i in range(5):
    print(i) # 0, 1, 2, 3, 4

# range(start, stop) - od start do stop-1
print("\nrange(2, 7):")

```

```

for i in range(2, 7):
    print(i) # 2, 3, 4, 5, 6

# range(start, stop, step) - z krokiem
print("\nrange(0, 10, 2):")
for i in range(0, 10, 2):
    print(i) # 0, 2, 4, 6, 8

# Ujemny krok - Liczenie wstecz
print("\nrange(10, 0, -2):")
for i in range(10, 0, -2):
    print(i) # 10, 8, 6, 4, 2

```

2. Pętla for z iterable obiektami

Iteracja przez różne typy danych

```

# Listy
print("Iteracja przez listę:")
liczby = [10, 20, 30, 40, 50]
for liczba in liczby:
    print(liczba * 2)

# Napisy (stringi)
print("\nIteracja przez string:")
for litera in "Python":
    print(litera.upper(), end=" ")

# Krotki (tuple)
print("\n\nIteracja przez krotkę:")
wspolrzedne = (3, 5, 7, 9)
for wartosc in wspolrzedne:
    print(f"Wartość: {wartosc}")

# Słowniki
print("\nIteracja przez słownik:")
student = {'imie': 'Jan', 'wiek': 21, 'kierunek': 'Informatyka'}
for klucz in student:
    print(f"{klucz}: {student[klucz]")

# Zbiory
print("\nIteracja przez zbiór:")
unikalne_liczby = {1, 3, 5, 3, 7, 1}
for liczba in unikalne_liczby:
    print(f"Unikalna: {liczba}")

```

3. Pętla while

Podstawowe zastosowania

```
# 1. Proste liczenie
print("Liczenie do 5:")
licznik = 1

while licznik <= 5:
    print(licznik)
    licznik += 1

# 2. Symulacja gry - zgadywanie liczby
import random
print("\nGra w zgadywanie liczby:")

# losuje liczbę całkowitą (int) z zakresu od 1 do 50. Włącznie z 1 i 50
liczba_do_zgadniecia = random.randint(1, 50)
prob = 0
zgadniete = False

while not zgadniete and prob < 7:
    prob += 1
    strzal = int(input(f"Próba {prob}/7. Zgadnij liczbę (1-50): "))

    if strzal == liczba_do_zgadniecia:
        print(f"Brawo! Zgadłeś za {prob}. razem!")
        zgadniete = True
    elif strzal < liczba_do_zgadniecia:
        print("Za mało!")
    else:
        print("Za dużo!")

if not zgadniete:
    print(f"Przegrałeś! Liczba to: {liczba_do_zgadniecia}")

# 3. Kalkulator - menu z opcjami
print("\nProsty kalkulator (wpisz 'koniec' aby wyjść):")
while True:
    print("\n1. Dodawanie")
    print("2. Odejmowanie")
    print("3. Mnożenie")
    print("4. Dzielenie")
    print("5. Koniec")

    wybor = input("Wybierz opcję (1-5): ")
```

```

if wybór == '5' or wybór.lower() == 'koniec':
    print("Do widzenia!")
    break

if wybór in ['1', '2', '3', '4']:
    a = float(input("Podaj pierwszą liczbę: "))
    b = float(input("Podaj drugą liczbę: "))

    if wybór == '1':
        print(f"{a} + {b} = {a+b}")
    elif wybór == '2':
        print(f"{a} - {b} = {a-b}")
    elif wybór == '3':
        print(f"{a} * {b} = {a*b}")
    elif wybór == '4':
        if b != 0:
            print(f"{a} / {b} = {a/b}")
        else:
            print("Błąd: Nie można dzielić przez zero!")
    else:
        print("Nieprawidłowy wybór!")

```

4. Instrukcje break, continue i else

break - natychmiastowe przerwanie pętli

```

# Szukanie pierwszej liczby podzielnej przez 7
print("Szukanie pierwszej liczby podzielnej przez 7:")
for i in range(1, 51):
    if i % 7 == 0:
        print(f"Znaleziono: {i}")
        break # przerywa pętlę natychmiast
    print(f"Sprawdzam {i}...")
print("Koniec wyszukiwania")

# Wyszukiwanie w liście
print("\nWyszukiwanie w liście książek:")
książki = [
    "Harry Potter",
    "Władca Pierścieni",
    "Gra o Tron",
    "Mały Książę",
    "Hobbit"
]
szukana = "Mały Książę"

```

```

for ksiazka in ksiazki:
    if ksiazka == szukana:
        print(f"✓ Znaleziono: {ksiazka}")
        break
    print(f"Sprawdzam: {ksiazka}")
else:
    print("Książka nie została znaleziona")

# Break w pętli while
print("\nLimitowane próby logowania:")
haslo = "tajne123"
proby = 3

while proby > 0:
    wprowadzone = input(f"Pozostało prób: {proby}. Podaj hasło: ")
    if wprowadzone == haslo:
        print("Logowanie udane!")
        break
    else:
        print("Błędne hasło!")
        proby -= 1

if proby == 0:
    print("Konto zablokowane!")

```

continue - pominięcie bieżącej iteracji

```

# Wyświetlanie tylko liczb nieparzystych
print("Liczby nieparzyste 1-10:")
for i in range(1, 11):
    if i % 2 == 0: # jeśli parzysta
        continue # pomin resztę tej iteracji
    print(i)

# Przetwarzanie danych z pominięciem błędnych
print("\nPrzetwarzanie listy z błędnymi danymi:")
dane = [10, 0, 5, "błąd", 15, 0, 20]

for wartosc in dane:
    if not isinstance(wartosc, (int, float)):
        print(f"Pomijam nieprawidłowy typ: {wartosc}")
        continue

    if wartosc == 0:
        print("Pomijam zero (nie można dzielić)")
        continue

    print(f"100 / {wartosc} = {100/wartosc:.2f}")

```

```

# Filtrowanie listy
print("\nFiltrowanie listy słów:")
slowa = ["kot", "pies", "", "papuga", None, "chomik", " "]

for słowo in slowa:
    if not słowo or not isinstance(słowo, str):
        continue

    słowo = słowo.strip()
    if not słowo: # puste po usunięciu spacji
        continue

    print(f"Długość słowa '{słowo}': {len(słowo)}")

```

else w pętlach - wykonuje się gdy pętla nie została przerwana przez break

```

# Sprawdzanie czy liczba jest pierwsza
def czy_pierwsza(n):
    if n < 2:
        return False

    for i in range(2, int(n**0.5) + 1):
        if n % i == 0:
            print(f"{n} nie jest pierwsza (dzieli się przez {i})")
            break
    else:
        # Wykona się tylko jeśli pętla nie została przerwana przez break
        return True
    return False

print("Sprawdzanie liczb pierwszych:")
for liczba in [2, 3, 4, 5, 17, 21, 29]:
    if czy_pierwsza(liczba):
        print(f"{liczba} JEST liczbą pierwszą")
    else:
        print(f"{liczba} NIE JEST liczbą pierwszą")

# Szukanie elementu w liście
print("\nSzukanie wartości w liście:")
lista = [10, 20, 30, 40, 50]
szukana = 35

for element in lista:
    if element == szukana:
        print(f"Znaleziono {szukana}")
        break
else:

```

```

# Wykona się tylko jeśli nie znaleziono elementu
print(f"Nie znaleziono {szukana} w liście")

# Weryfikacja danych wejściowych
print("\nWeryfikacja wprowadzonych liczb:")
while True:
    try:
        liczba = int(input("Podaj liczbę dodatnią: "))
        if liczba <= 0:
            print("Liczba musi być dodatnia!")
            continue
        break # jeśli doszliśmy tutaj, liczba jest poprawna
    except ValueError:
        print("To nie jest poprawna liczba!")
else:
    # W pętlach while, else wykonuje się jeśli nie było break
    # (ale tutaj break jest, więc else się nie wykona)
    print("Ta wiadomość się nie pojawi")

print(f"Wprowadzono poprawną liczbę: {liczba}")

```

Lekcja

Temat: Funkcje: Definiowanie i argumenty w Python

Podstawowa składnia:

```

def nazwa_funkcji(parametry):
    """Dokumentacja (opcjonalna)"""
    # ciało funkcji
    return wartość # opcjonalnie

```

Przykłady:

```

# Prosta funkcja bez parametrów
def przywitaj():
    print("Cześć!")

# Funkcja z parametrem
def powitaj(imie):
    print(f"Cześć, {imie}!")

```

```
# Funkcja zwracająca wartość
def dodaj(a, b):
    wynik = a + b
    return wynik
```

Argumenty funkcji

1. Argumenty pozycyjne - najprostsze, kolejność ma znaczenie:

```
def opisz_osobe(imie, wiek, miasto):
    print(f"{imie}, lat {wiek}, z {miasto}")

opisz_osobe("Anna", 25, "Warszawa") # OK
```

2. Argumenty nazwane (keyword arguments):

```
# Można podawać w dowolnej kolejności
opisz_osobe(miasto="Kraków", imie="Piotr", wiek=30)
```

3. Argumenty domyślne:

```
def powitaj(imie, język="polski"):
    if język == "polski":
        return f"Cześć, {imie}!"
    elif język == "angielski":
        return f"Hello, {imie}!"
    else:
        return f"Witaj, {imie}!"

print(powitaj("Anna")) # używa domyślnego "polski"
print(powitaj("John", "angielski"))
```

⚠ Ważne: Argumenty domyślne muszą być na końcu listy parametrów:

```
# Poprawnie
def funkcja(a, b=10): ...
```

```
# Błędnie - parametr domyślny przed wymaganym
def funkcja(a=10, b): ... # SyntaxError
```

4. *args - dowolna liczba argumentów pozycyjnych:

przekazywane **bez nazw**. Nie piszemy sumuj_wszystko(a=10, b=20)

```
def sumuj_wszystko(*liczby):
    """Przyjmuje dowolną liczbę argumentów"""
    return sum(liczby)

print(sumuj_wszystko(1, 2, 3))      # 6
print(sumuj_wszystko(10, 20))      # 30
print(sumuj_wszystko(1, 2, 3, 4, 5)) # 15
```

5. **kwargs - dowolna liczba argumentów nazwanych:

```
def pokaz_dane(**dane):
    """Przyjmuje dowolną liczbę argumentów nazwanych"""
    for klucz, wartosc in dane.items():
        print(f"{klucz}: {wartosc}")

pokaz_dane(imie="Anna", wiek=25, miasto="Warszawa")
# Wyświetli:
# imie: Anna
# wiek: 25
# miasto: Warszawa
```

6. Łączenie różnych typów argumentów:

```
def skomplikowana_funkcja(a, b, *args, opcja=None, **kwargs):
    print(f"a={a}, b={b}") # pamiętać o wcięciach
    print(f"args={args}")
    print(f"opcja={opcja}")
    print(f"kwargs={kwargs}")

# Przykład wywołania:
skomplikowana_funkcja(1, 2, 3, 4, 5, opcja="test", x=100, y=200)
```

Kolejność parametrów musi być:

1. Argumenty pozycyjne
2. *args
3. Argumenty domyślne/nazwane
4. **kwargs

Lekcja

Temat: lambda. Zasięg zmiennych (scope) Łańcuchy znaków: Stringi, formatowanie (f-strings, format()), metody (split, join, replace) w Python

lambda pozwala tworzyć **małe, anonimowe funkcje** (czyli funkcje bez nazwy)
Czyli zamiast:

```
def dodaj(a, b):  
    wynik = a + b  
    return wynik
```

Skrótowa wersja w lambda:

```
lambda a, b: a + b
```

Składnia:

lambda argumenty: wyrażenie

Istotne:

- Zawierać może **tylko jedno wyrażenie**
- Nie możesz używać w niej instrukcji typu **if, for, while, return**

Zasięg to **obszar kodu**, w którym dana nazwa (zmienna, funkcja, klasa) jest widoczna i można ją zastosować.

Python stosuje **regułę LEGB** – szuka nazwy dokładnie w tej kolejności:

Litera	Nazwa	Gdzie szuka
L	Local	wewnątrz aktualnej funkcji / metody
E	Enclosing	w funkcjach otaczających (nested functions)
G	Global	na poziomie modułu (poza wszystkimi funkcjami)

B	Built-in	wbudowane nazwy (print, len, int, range...)
----------	-----------------	---

2. Rodzaje zasięgu – przykłady

a) Local (lokalny)

```
def metoda():
    x = 10      # ← zmienna lokalna
    print(x)    # 10

metoda()
# print(x)      # NameError – x nie istnieje poza funkcją
```

b) Global (globalny)

```
x = 5
def metoda():
    print(x)

metoda()
```

c) Enclosing (otaczający) + nonlocal

```
def zewnetrzna():
    x = 10          # enclosing scope

    def wewnetrzna():
        nonlocal x      # Nie tworzy nowej zmiennej lokalnej. Użyje zmiennej z najbliższego
        zewnętrznego scope (enclosing scope).
        x = 20

    wewnetrzna()
    print(x)        # 20

zewnetrzna()
```

Bez nonlocal:

```
def zewnetrzna():
    x = 10
```

```
def wewnetrzna():
    x = 20      # tworzy nową zmienną LOKALNA!
    print(x)    # 20
```

```
wewnetrzna()
print(x)      # 10 (oryginalna nie zmieniła się)
```

```
zewnetrzna()
```

d) Built-in

```
print(len([1, 2, 3])) # len i print są z built-in scope
```

built-in scope to **najbardziej zewnętrzny zakres**, który zawsze istnieje.
Zawiera funkcje i typy wbudowane w Pythona, np.:

- print
- len
- int
- str
- list
- dict
- sum
- min
- Max

3. Modyfikowanie zmiennych globalnych – global

```
licznik = 0
```

```
def zwiększa():
    global licznik    # bez tego dostalibyśmy UnboundLocalError: cannot access local
                      # variable 'licznik' where it is not associated with a value
    licznik += 1
```

```
zwiększa()
```

```
zwięks()
print(licznik)      # 2
```

Przykład z lambdą (klasyczna pułapka):

```
funcs = []
for i in range(3):
    funcs.append(lambda: i)

print([f() for f in funcs])

# Poprawnie:
funcs = []
for i in range(3):
    funcs.append(lambda x=i: x)  # domyślny argument "zamraża" wartość

print([f() for f in funcs])
```

globals() i locals() – funkcje do inspekcji aktualnego scope

1. globals() – zwraca słownik wszystkich zmiennych globalnych (poziom modułu)

```
# plik: test_scope.py

a = 10
b = "hello"
c = [1, 2, 3]

print("==== globals() na poziomie modułu ===")
print(globals())

# === globals() na poziomie modułu ===
# {
#     'a': 10,
#     'b': 'hello',
#     'c': [1, 2, 3],
#     '__name__': '__main__',
#     '__builtins__': <module 'builtins' ...>,
#     ...
# }
```

2. locals() – zwraca słownik zmiennych lokalnych aktualnego scope

Przykład wewnątrz funkcji:

```
a = 90
def metoda_testowa():
    x = 42
    y = "Python"
    z = [10, 20, 30]

    print("==== locals() wewnątrz funkcji ===")
    print(locals())

# === locals() wewnątrz funkcji ===
# {'x': 42, 'y': 'Python', 'z': [10, 20, 30]}

metoda_testowa()
```

```
global_var = "globalna"

def test():
    local_var = "lokalna"

    print("globals() zawiera global_var:", "global_var" in globals()) # True
    print("locals() zawiera local_var:", "local_var" in locals()) # True
    print("locals() zawiera global_var:", "global_var" in locals()) # False

test()
```

Modyfikacja przez słownik:

```
nowa = 123

def demo():
    x = 100
    locals()["x"] = 999
    print(x)

    globals()["nowa"] = 42
    print(nowa)
```

```
demo()
```

ŁAŃCUCHY ZNAKÓW (STRINGS)

Stringi to sekwencje znaków, które mogą zawierać litery, cyfry, symbole i spacje.

```
# Różne sposoby tworzenia stringów
imie = "Anna"
nazwisko = 'Kowalska'
zdanie = """To jest
wielolinijkowy
tekst"""

print(imie)      # Anna
print(nazwisko)  # Kowalska
print(zdanie)    # To jest\nwielolinijkowy\ntekst

# Podstawowe operacje na stringach
powitanie = "Cześć, " + imie + "!"  # konkatenacja (łączenie)
print(powitanie)  # Cześć, Anna!

powtorzenie = "git" * 3  # powielanie
print(powtorzenie)  # gitgitgit

# Indeksowanie i wycinanie (slicing)
tekst = "Python"
print(tekst[0])      # P (pierwszy znak)
print(tekst[-1])     # n (ostatni znak)
print(tekst[1:4])    # yth (od indeksu 1 do 3)
print(tekst[:3])     # Pyt (od początku do indeksu 2)
print(tekst[3:])     # hon (od indeksu 3 do końca)
```

Metoda format()

```
imie = "Piotr"
wiek = 35

# Pozyczynie
print("Nazywam się {} i mam {} lat.".format(imie, wiek))

# Z indeksami
print("Nazywam się {0} i mam {1} lat. {0} lubi
programować.".format(imie, wiek))

# Z nazwami
print("Nazywam się {name} i mam {age} lat.".format(name=imie,
age=wiek))

# Formatowanie Liczb
liczba = 3.14159
print("Liczba Pi to {:.2f}".format(liczba))
```

METODY STRINGÓW

split() - dzielenie stringa na listę

```
# Podstawowe split()
zdanie = "Python jest świetny"
slowa = zdanie.split()
print(slowa) # ['Python', 'jest', 'świetny']

# Split z własnym separatorem
data = "2024-01-15"
czesc_daty = data.split("-")
print(czesc_daty) # ['2024', '01', '15']
```

```
# Split z limitem podziałów
tekst = "jabłko,gruszka,banan,wiśnia"
owoce = tekst.split(", ", 2)
print(owoce) # ['jabłko', 'gruszka', 'banan,wiśnia']

# Praktyczny przykład - parsowanie danych
dane_logowania = "user:haslo123"
login, haslo = dane_logowania.split(":")
print(f"Login: {login}, Hasło: {haslo}")
```

join() - łączenie listy w string

```
# Podstawowe join()
owoce = ['jabłko', 'gruszka', 'banan']
tekst = ", ".join(owoce)
print(tekst) # jabłko, gruszka, banan

# Różne separatory
slowa = ['Python', 'jest', 'fajny']
print(" ".join(slowa)) # Python jest fajny
print("-".join(slowa)) # Python-jest-fajny
print("") .join(slowa)) # Pythonjestfajny

# łączenie z transformacją
liczby = [1, 2, 3, 4, 5]
# Najpierw musimy zamienić liczby na stringi
tekst = ", ".join(str(x) for x in liczby)
print(tekst) # 1, 2, 3, 4, 5

# Praktyczny przykład - tworzenie ścieżki
katalogi = ['home', 'user', 'dokumenty']
sciezka = '/'.join(katalogi)
print(sciezka) # home/user/dokumenty
```

replace() - zamiana fragmentów stringa

```
tekst = "Lubię koty. Koty są fajne."
nowy_tekst = tekst.replace("koty", "psy")
print(nowy_tekst)

tekst = "Lubię koty. Koty są fajne."
nowy_tekst = tekst.replace("koty", "psy").replace("Koty", "Psy")
print(nowy_tekst)

tekst = "jabłko jabłko jabłko"
print(tekst.replace("jabłko", "gruszka", 2))
tekst = "Hello, World!!!"
czysty = tekst.replace(",", "").replace("!", "")
print(czysty)

numer_telefonu = "+48 123-456-789"
czysty_numer = numer_telefonu.replace("+48 ", "").replace("-", "")
print(czysty_numer)
```

Przykład 1: Czyszczenie danych wejściowych

```
def czysc_dane(tekst):
    tekst = tekst.strip()
    while " " in tekst:
        tekst = tekst.replace(" ", " ")
    tekst = tekst.lower()
    return tekst

dane_wejsciowe = " To Jest Tekst "
czyste = czysc_dane(dane_wejsciowe)
print(czyste)
```

Przykład 2: Generowanie raportów

```
def generuj_raport(studenci):
    linie = []
    for imie, ocena in studenci:
        linie.append(f"{imie:10} - {ocena}")
    return "\n".join(linie)

studenci = [("Anna", 5), ("Jan", 4), ("Katarzyna", 4.5)]
print(generuj_raport(studenci))
# Wynik:
# Anna      - 5
# Jan       - 4
# Katarzyna - 4.5
```

Lekcja

Temat: Zastosowania programowania obiektowego (OOP) w Pythonie. Wyjaśnienie kluczowych pojęć OOP: Klasa (Class), Obiekt (Object), Metody (Methods), Atrybuty (Attributes), init (konstruktor)

Programowanie obiektowe (OOP) w Pythonie to **paradygmat** programowania, który pozwala na **organizowanie kodu wokół "obiektów"** zamiast funkcji i procedur. Jest to jedna z kluczowych cech Pythona, czyniąca go elastycznym i łatwym w utrzymaniu dla dużych projektów.

Kluczowe pojęcia OOP

1. Klasa (Class)

Klasa to szablon lub blueprint **definiujący strukturę i zachowanie obiektów**.

Definiuje, jakie atrybuty (dane) i metody (funkcje) będą miały obiekty stworzone na jej

podstawie. Klasa sama w sobie nie przechowuje danych – to robią jej instancje (obiekty).

Przykład:

```
class Pies:  
    pass      # klasa nic nie robi, czyli blok istnieje ale jest pusty  
  
moj_pies = Pies()  
print(type(moj_pies))
```

2. Obiekt (Object)

Obiekt to instancja (konkretny egzemplarz) **klasy**. **Obiekt ma własne atrybuty i może wywoływać metody zdefiniowane w klasie**. Tworzyś obiekt wywołując klasę jak funkcję (np. `Klasa()`). Obiekty są unikalne, nawet jeśli pochodzą z tej samej klasy.

Przykład:

```
class Pies:  
    def szczekaj(self):  
        print("Hau hau!")  
  
moj_pies = Pies()  
inny_pies = Pies()  
  
moj_pies.szczerkaj()  
print(moj_pies == inny_pies)
```

2. a) self

self to odniesienie (referencja) do bieżącej instancji klasy (konkretnego obiektu). To sposób, w jaki obiekt "widzi samego siebie" i może operować na swoich własnych danych.

Kiedy definiujesz metodę wewnątrz klasy, **Python automatycznie przekazuje instancję obiektu jako pierwszy argument tej metody**. **self pozwala metodzie odwoływać się do atrybutów i innych metod tej konkretnej instancji**.

Bez self metoda nie wiedziałaby, do którego obiektu się odnosi – czy do atrybutów klasy, czy instancji.

self w metodach klasy

```
class Osoba:  
    def przedstaw_sie(self):  
        print(f"Jestem obiektem: {self}")  
        print(f"Mój typ to: {type(self)}")  
  
osoba1 = Osoba()  
osoba2 = Osoba()  
  
print("Wywołanie dla osoby1:")  
osoba1.przedstaw_sie() # self = osoba1  
  
print("\nWywołanie dla osoby2:")  
osoba2.przedstaw_sie() # self = osoba2
```

Wynik:

*Wywołanie dla osoby1: Jestem obiektem: <_main_.Osoba object at 0x7c84a9049be0>
Mój typ to: <class '_main_.Osoba'>
Wywołanie dla osoby2: Jestem obiektem: <_main_.Osoba object at 0x7c84a9049c10>
Mój typ to: <class '_main_.Osoba'>*

self w __init__

Metoda `__init__` jest wywoływana automatycznie podczas tworzenia obiektu. `self` w `__init__` odnosi się do **nowo tworzonego obiektu**:

```
class Pracownik:  
    def __init__(self, imie, pensja):  
        print(f"1. Tworzę obiekt: {self}")  
        print(f"2. Przypisuję imię '{imie}' do self.imie")  
  
        self.imie = imie  
        self.pensja = pensja  
  
        print(f"3. Obiekt po inicjalizacji: imię={self.imie}, pensja={self.pensja}")  
        print(f"4. ID obiektu: {id(self)}")  
  
p = Pracownik("Anna", 5000)  
print(f"\nUtworzony obiekt: {p}")  
print(f"ID obiektu z zewnątrz: {id(p)})
```

Wynik:

1. Tworzę obiekt: <__main__.Pracownik object at 0x7fa10867db50>
 2. Przypisuję imię 'Anna' do self.imie
 3. Obiekt po inicjalizacji: imię=Anna, pensja=5000
 4. ID obiektu: 140329607486288
- Utworzony obiekt: <__main__.Pracownik object at 0x7fa10867db50>
ID obiektu z zewnątrz: 140329607486288

self odwołuje się do atrybutów obiektu

```
class Samochod:  
    def __init__(self, marka):  
  
        print(f"{self}") # <__main__.Samochod object at 0x7b916db45be0>  
  
        self.marka = marka  
        self.predkosc = 0  
  
    def przyspiesz(self, wartosc):  
        # self.predkosc - odwołanie do atrybutu  
        self.predkosc += wartosc  
        print(f"{self.marka}: jadę z prędkością {self.predkosc} km/h")  
  
    def hamuj(self):  
        # self.predkosc - modyfikacja atrybutu  
        self.predkosc = 0  
        print(f"{self.marka}: zatrzymałem się")  
  
auto = Samochod("Toyota")  
auto.przyspiesz(50) # self to auto  
auto.hamuj() # self to auto
```

self pozwala odróżnić atrybuty obiektu od zmiennych lokalnych

```
class Przyklad:  
    def __init__(self, wartosc):  
        # wartosc - to parametr (zmienna Lokalna)  
        # self.wartosc - to atrybut obiektu  
        self.wartosc = wartosc  
  
    def pokaz(self, wartosc):  
        # wartosc - to parametr metody  
        # self.wartosc - to atrybut obiektu  
        print(f"Parametr metody: {wartosc}")  
        print(f"Atrybut obiektu: {self.wartosc}")  
  
ob = Przyklad(100)  
ob.pokaz(50) # Parametr: 50, Atrybut: 100
```

Python automatycznie przekazuje obiekt jako pierwszy argument przy wywołaniu metody:

```
python

class Test:
    def metoda(self, arg1, arg2):
        print(f"self: {self}")
        print(f"arg1: {arg1}, arg2: {arg2}")

t = Test()

# Te dwa wywołania są RÓWNOWAŻNE:
t.metoda(10, 20)          # Python automatycznie wstawia self
Test.metoda(t, 10, 20)     # Ręczne przekazanie obiektu jako self
```

3. Metody (Methods)

Metody to funkcje zdefiniowane wewnątrz klasy, które operują na obiektach.

Pierwszy parametr metody to zawsze self (odwołuje się do bieżącego obiektu).

Metody pozwalają na interakcję z obiektem. Mogą modyfikować atrybuty lub zwracać wartości. Są wywoływanie na obiekcie: obiekt.metoda().

```
class Samochod:
    def jedz(self, predkosc):
        print(f"Samochód jedzie z prędkością {predkosc} km/h.")

    def zatrzymaj(self):
        print("Samochód się zatrzymał.")

moj_samochod = Samochod()      # Obiekt
moj_samochod.jedz(100)         # Wyjście: Samochód jedzie z prędkością 100 km/h.
moj_samochod.zatrzymaj()       # Wyjście: Samochód się zatrzymał.
```

4. Atrybuty (Attributes)

Atrybuty to zmienne związane z klasą lub obiektem. Mogą być atrybutami klasy (wspólne dla wszystkich obiektów) lub **instancji** (unikalne dla każdego obiektu). Atrybuty przechowują dane. Dostęp do nich: obiekt.atrybut. Mogą być ustawiane w `__init__` lub bezpośrednio.

```
class Osoba:
    gatunek = "Człowiek"  # Atrybut klasy (wspólny)
```

```

def __init__(self, imie): # Więcej o __init__ poniżej
    self.imie = imie      # Atrybut instancji (unikalny)

jan = Osoba("Jan")          # Obiekt
print(jan.imie)             # Wyjście: Jan
print(jan.gatunek)           # Wyjście: Człowiek

anna = Osoba("Anna")
print(anna.imie)             # Wyjście: Anna
print(anna.gatunek)           # Wyjście: Człowiek (wspólny)

```

5. init (Konstruktor)

`__init__` to specjalna metoda (**konstruktor**), która jest wywoływana automatycznie przy tworzeniu obiektu. Służy do inicjalizacji atrybutów obiektu. Nie zwraca wartości, ale ustawia początkowe stany. Pierwszy parametr to `self`. Możesz podać argumenty przy tworzeniu obiektu.

```

class Ksiazka:
    def __init__(self, tytul, autor, rok=2023): # Konstruktor z domyślnym
        parametrem
        self.tytul = tytul                      # Inicjalizacja atrybutów
        self.autor = autor
        self.rok = rok

    def opis(self): # Funkcja używająca atrybutów
        return f"{self.tytul} autorstwa {self.autor} ({self.rok})"

moja_ksiazka = Ksiazka("Python dla początkujących", "Jan Kowalski") # Tworzenie z argumentami
print(moja_ksiazka.opis()) # Wyjście: Python dla początkujących autorstwa Jan Kowalski (2023)

inna_ksiazka = Ksiazka("Zaawansowany Python", "Anna Nowak", 2024)
print(inna_ksiazka.opis()) # Wyjście: Zaawansowany Python autorstwa Anna Nowak (2024)

```