

# Python

## Programowanie w języku Python

---

Radosław Roszczyk

12 września 2021

Politechnika Warszawska

# O czym będzie

1. O języku
2. Proste typy danych
3. Złożone typy danych
4. Instrukcje sterujące
5. Obsługa wyjątków w Python

# O języku

---

# Początki języka Python

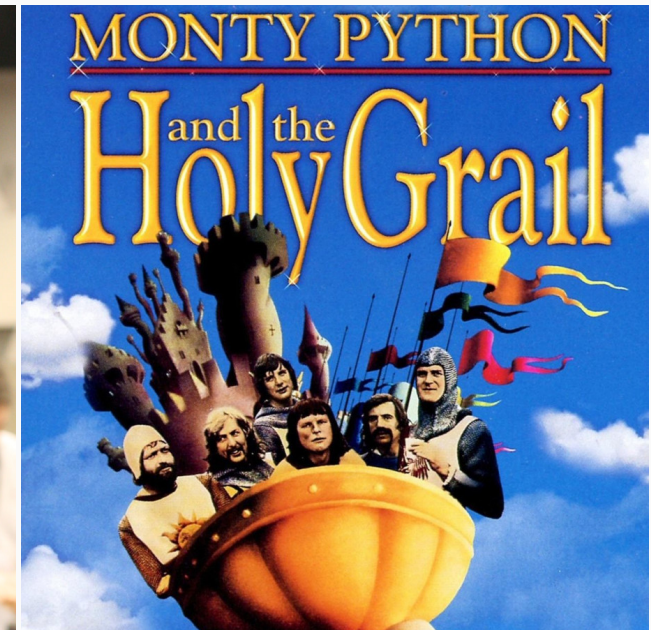
Python powstał we wczesnych latach 90 jako następca języka ABC. Pierwotna wersja języka stworzona została w Centrum Matematyki i Informatyki w Amsterdamie.

## Bieżący rozwój

- Python Software Foundation (PSF)

## Wersje języka

- 3.9 – październik 2025
- 3.8 – październik 2024
- 3.7 – czerwiec 2023
- 3.6 – grudzień 2021
- 2.7 – styczeń 2020



**Rysunek 1:** Guido van Rossum - **Rysunek 2:** Fragment plakatu  
szef zespołu tworzącego Pythona promującego film Monty Python

# Proste typy danych

---

# Typy liczbowe

## Typ całkowity (integer)

- Stałe: 256, 0x100, 0o400, 0b100000000
- Maksymalna wartość:  
9223372036854775807\*

## Typ logiczny (boolean)

- Prawda: True, > 0, niepusty element
- Fałsz: False, 0, None, [],

## Liczba zespolona (complex)

- Stałe: 30+30j, a+30j,
- Konstruktor: complex(30, 30),  
complex(a, b)

## Liczba zmiennopozycyjna (float)

- Stałe: 3.141592, 100., .001, 10e4
- Maksymalna wartość:  
1.7976931348623157e+308\*

# Typy znakowe

## Typ znakowy (string)

- Stałe:
  - `"tekst"`,
  - `'tekst'`,
  - `"""bardzo długi tekst  
w kilku liniach"""`
- Napisy indeksujemy od 0
- `s[0]`, `s[1]`, `s[-1]`, `s[1 : 4]`, `s[3 : len(s)]`

## Podstawowe funkcje

- `chr()` – konwertuje liczbę na znak
- `ord()` – konwertuje znak na liczbę
- `len()` – zwraca długość napisu w znakach
- `str()` – reprezentuje obiekt w postaci znakowej

# Konwersja między typami

## Użycie nazwy typu jako operatora rzutowania

- `int(3.141592)`
- `float(7)`
- `complex(5, 15)`
- `bool(5)`
- `str(4543)`

## Konwersja podczas wykonania

- $1/4 * 3 \rightarrow 0,75$
- `int(1/4 * 3)  $\rightarrow$  0`
- $1 * 4 // 3 \rightarrow 1$
- $1 * 4 / 3 \rightarrow 1.3333333333333333$
- `int("1234") * 10 + 5  $\rightarrow$  12345`
- `int("15") * 4  $\rightarrow$  60`
- `"15" * 4  $\rightarrow$  15151515`



# Złożone typy danych

---

# Złożone typy danych

## Kolekcje

- Listy: [1, 2, 3]
- Krotki: (7, "napis", "drugi", (1, 2, 0x7), 30+30j)
- Słowniki: "jeden": 1, "dwa": 2, "trzy": 3
- Zbiory
- Obiekty

# Listy

Lista jest podstawową strukturą danych służącą do reprezentacji danych dynamicznych.

## Przykładowe listy

```
1 lista1 = [1, 2, 3]
2 lista2 = ["a", "b", "c"]
3 lista3 = ["4", lista2, "napis", 5]
```

## Alternatywne tworzenie list

```
1 lista_znakow = list(("1", "2", "3"))
2 lista_liczb = list((1, 2, 3))
```

# Operacje na listach

## Inicjalizacja listy

```
1 kolekcja = [i for i in range(0, 10, 2)]  
2 kolekcja = [i * i + 2 for i in range(0, 10, 2)]  
3 parzyste = [i for i in range(0, 20) if i % 2 == 0]
```

## Przekształcenie list

```
1 kolekcja_orig = ['kot', 'pies', 'chomik']  
2 nowa_kolekcja = ['mój ' + i for i in kolekcja_orig]
```

# Krotki

Krotka jest strukturą przechowującą stałe wartości o różnych typach danych. Krotki są strukturami niemodyfikowalnymi.

## Przykładowe krotki

```
1 krotka1 = ('jeden', 'dwa', 'trzy', 'cztery', 'siedem', 'osiem')
2 krotka2 = (1, 2, 3, 4, 5);
3 krotka3 = "a", "b", "c", "d"
4 krotka4 = (1, 'dwa', [3, 4.5])
5 krotka5 = ()
6 krotka6 = (50,)
```

## Odwoływanie się do poszczególnych elementów

```
1 print("krotka1[0]: ", krotka1[0])
2 print("krotka1[1:5]: ", krotka1[1:5])
```

# Operacje na krotkach

## Funkcje wbudowane

```
1 tup = ( 'jeden', 'dwa', 'trzy', 'cztery', 'piec', 'szesc' )
2
3 # len() zwraca liczbe elementów krotki
4 print(len(tup))
5
6 # index() zwraca indeks pierwszego wystąpienia elementu w krotce
7 print(tup.index( 'dwa' ))
8
9 # count() zwraca liczbe wystąpień elementu w krotce
10 print(tup.count( 'dwa' ))
11
12 # min() i max() zwracają najmniejszy i największy element krotki
13 # sum() zwraca ęsum elementów krotki
14 tupN = (1,2,3) # dla tych metod krotka musi zawierać tylko liczby
15 print(min(tupN))
16 print(max(tupN))
17 print(sum(tupN))
```

# Po co stosować krotki

## Zalety

- Krotki stosujemy dla danych tylko do odczytu, np: lista dni tygodnia
- Krotki mogą przechowywać różne typy danych
- Krotki są znacznie szybsze w obliczeniach niż listy

## Porównanie wydajności krotek i list

```
1 roszczyr@laptop ~ → python3 -m timeit "x=(1,2,3,4,5,6,7,8)"
2 5000000 loops, best of 5: 8 nsec per loop
3 roszczyr@laptop ~ → python3 -m timeit "x=[1,2,3,4,5,6,7,8]"
4 500000 loops, best of 5: 42.1 nsec per loop
5 roszczyr@laptop ~ →
6 roszczyr@laptop ~ → python3 -m timeit -s "x=(1,2,3,4,5,6,7,8)" "y=x[3]"
7 2000000 loops, best of 5: 19.8 nsec per loop
8 roszczyr@laptop ~ → python3 -m timeit -s "x=[1,2,3,4,5,6,7,8]" "y=x[3]"
9 1000000 loops, best of 5: 20.8 nsec per loop
```

# Słowniki

Słownik jest jednym z wbudowanych typów danych i przechowuje dane w postaci pary: klucz – wartość.

## Przykładowe słowniki

```
1 slownik_plen = { "kon": "horse", "pies": "dog", "matematyka": "mathematics" }  
2 moj_slownik = { 'klucz1': 'wartosc1', 'klucz2': 'wartosc2' }  
3 slow1 = { 'klucz1': 'napis', 'klucz2': 123, 'klucz3': ['i1', 'i2', 'i3'] }
```

## Alternatywne tworzenie słowników

```
1 moj_slownik1 = {}  
2 moj_slownik1['klucz1'] = 'wartosc1'  
3 moj_slownik1['klucz2'] = 'wartosc2'
```



# Operacje na słownikach

## Usuwanie elementów

```
1 del moj_sownik['klucz1']
2 moj_sownik.pop('klucz2')
3 moj_sownik.clear()
```

## Funkcje wbudowane

```
1 dict1 = {'k1': 'w1', 'k2': 'w2'} # zdefiniowanie słownika
2
3 # Sprawdzanie liczby kluczy w słowniku
4 len(dict1)
5
6 # Zwracanie listy elementów słownika
7 dict1.items() # wyświetlenie wszystkich elementów słownika: par klucz - wartość
8 dict1.keys()  # wyświetlenie wszystkich kluczy słownika
9 dict1.values() # wyświetlenie wszystkich wartości słownika
```

# Instrukcje sterujące

---

# Instrukcja warunkowa

## Instrukcja warunkowa I

```
1 x = 3
2 if x > 0:
3     print('dodatnia')
4 elif x < 0:
5     print('ujemna')
6 else : print('zero')
```

## Instrukcja warunkowa II

```
1 if punkty >= 90:
2     print("ocena 5")
3 elif punkty >=80:
4     print("ocena 4")
5 elif punkty >=70:
6     print("ocena 3")
7 elif punkty >= 65:
8     print("ocena 2")
9 else:
10    print("ocena 1")
```

## Zastosowanie instrukcji warunkowej do inicjalizacji listy

```
1 tp = ['L: ' + str(i) + ' parzysta' if i % 2 == 0 else
2       'L: ' + str(i) + ' nieparzysta' for i in range(0, 20)]
3 print(tp)
```

# Instrukcje pętli

## Instrukcja for I

```
1 a = [1, 2, 3, 4]
2 for e in a:
3     print(e)
4 print('koniec')
```

## Instrukcja for II

```
1 suma = 0
2 for i in range(100):
3     suma = suma + i
4 print('suma = ', suma)
```

## Zastosowanie instrukcji for do inicjalizacji listy

```
1 parzyste = [i for i in range(0, 20) if i % 2 == 0]
2 print(parzyste)
```

# Instrukcje pętli

## Instrukcja while-else I

```
1 liczba = 0
2 while liczba < 5:
3     print("W petli")
4     liczba += 1
5 else:
6     print("Koniec przez 'else'")
```

## Instrukcja while-else II

```
1 liczba = 3
2 while liczba < 20 :
3     print("W pętli")
4     if (liczba <= 0):
5         break
6     liczba += 1
7     liczba *= liczba
8 else:
9     print("Koniec przez 'else'")
10 print("Koniec petli")
```

## Pętla pusta

```
1 while (True): pass # brak mozliwosci tworzenia pustych bloków
```

# Instrukcje pętli

## Instrukcja while-else II

```
1 liczba = 3
2 while liczba < 20 :
3     print("W ęptli")
4     if (liczba <= 0):
5         break
6     liczba += 1
7     liczba *= liczba
8 else:
9     print("Koniec przez 'else'")
10 print("Koniec petli")
```

## Instrukcja while-else III

```
1 liczba = -3
2 while liczba < 20 :
3     print("W ęptli")
4     if (liczba <= 0):
5         break
6     liczba += 1
7     liczba *= liczba
8 else:
9     print("Koniec przez 'else'")
10 print("Koniec petli")
```

Pętle while-else wykorzystujemy gdy chcemy wykonać określoną liczbę powtórzeń pętli oraz zakończyć je pewnym kodem. Dopuszczamy jednak możliwość, że w trakcie wykonywania pętli dojdzie do sytuacji, kiedy ta dodatkowa instrukcja nie powinna się wykonać.

# Klauzula with

Klauzula with służy do pracy z tzw. kontekstami. Najprościej można je rozumieć jak tworzenie bloku do pracy z pewnym zewnętrznym względem Pythona narzędziem. Przykładem może tu być praca z otwartym plikiem na dysku, zasobem internetowym, czy innym programem systemu operacyjnego.

## Przykład

```
1 with open('readme.txt', 'r') as plik:  
2     dane = plik.read()  
3     print('Czy plik jest zamknięty ', plik.closed)  
4 print(dane)  
5 print('Czy plik jest zamknięty ', plik.closed)
```

# Obsługa wyjątków w Python

---



# Obsługa wyjątków

Python podobnie jak większość języków programowania pozwala na tworzenie, zgłaszanie i obsługiwanie wyjątków.

Czym jest wyjątek? Jest sytuacją w działaniu programu, która uniemożliwia poprawne wykonanie części skryptu. Podobnie jak w Javie, kod który może nie zakończyć się poprawnym wykonaniem instrukcji w bloku try

## Przykład

```
1 tekst = "nie liczba"
2 try :
3     x = int(tekst)
4 except ValueError:
5     print("Zła wartosc, ale")
6 print("Ale mimo to udalo sie")
```

## Definiowanie wyjątków

```
1 class MyException(Exception):
2     pass
3 def call_exception():
4     raise MyException()
5 try:
6     call_exception()
7 except MyException:
8     print("Przechwycono wyjatek")
```

# Obsługa wyjątków

Python posiada podobnie do Javy klauzulę finally, czyli blok który musi się wykonać niezależnie do tego czy wyjątek został czy nie został zgłoszony.

## Przykład

```
1 def dzielenie(x, y):
2     try:
3         print("Dzielenie " + str(x) + "/" + str(y))
4         result= x/y
5     except ZeroDivisionError:
6         print("Bład dzielenia")
7     else:
8         print("Wynik to " + str(result))
9     finally:
10        print("A ten blok wykona sie zawsze")
11
12 dzielenie(2, 3)
13 dzielenie(1, 0)
```

Pytania?