

Funkcje w pythonie

Każda funkcja składa się ze słowa *def*, nazwy, przyjmowanych argumentów(jeśli jakieś przyjmuje) oraz ciała funkcji.

```
def nazwa_funkcji(argument_1, argument_2):  
    ciało funkcji(po wcięciu)
```



Funkcje, które już znacie

- **print('tekst')** - jako argument przyjmuje string-a i wypisuje go na konsolę
- **input()** - wczytuje tekst z klawiatury
- **max(a, b)** – zwraca większą z wartości **a** i **b**
- **randint(a, b)** – zwraca losową liczbę całkowitą z przedziału **a** i **b**
- **sqrt(a)** – zwraca pierwiastek z **a**

Przykładowa funkcja 1

Prosta funkcja przyjmująca jakieś liczby **a** i **b**.
Zwraca ich sumę.

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

Wywołanie:

```
suma = dodaj (34, 67)  
print (suma)
```

Rodzaje argumentów

Argumentami mogą być dowolne typy, nawet te stworzone przez nas(czyli klasy). Typy które poznaliśmy:

- **bool** – wartość **True** albo **False**
- **str** – string, czyli napis, np. 'mama'
- **int** - liczba całkowita
- **float** – liczba zmiennoprzecinkowa, czyli np. 2.5
- **list** – lista, czyli np. [2.5, 3.5, 11.2]

W drugiej części zajęć stworzymy klasy, czyli nasze własne typy.

Przykładowa funkcja 2

```
def wypisz_liste(lista):  
    for element in lista:  
        print(element)  
  
def policz_srednia(lista):  
    suma = 0  
    for element in lista:  
        suma += element  
    return suma/len(lista)
```

Przykładowa funkcja 2 - wywołanie

```
lista = [2, 3, 4, 5, 6]  
wypisz_liste(lista)  
srednia = policz_srednia(a)  
print(srednia)
```

Zły typ argumentu

Co się stanie, jeśli jako argument prześlemy wartość złego typu? Czyli:

```
a = 5  
policz_srednia(a)
```

Dostaniemy błąd, bo funkcja będzie próbowała przejść pętlą **for** przez zmienną, która nie jest listą.

```
Traceback (most recent call last):  
  File "C:/Users/jerzy/Desktop/funkcje2.py", line 9, in <module>  
    srednia = policz_srednia(a)  
  File "C:/Users/jerzy/Desktop/funkcje2.py", line 3, in policz_srednia  
    for element in lista:  
TypeError: 'int' object is not iterable
```


Jak sobie z tym poradzić?

Możemy np. użyć tzw. **type hint**-ów, czyli wskazówek, jakiego typu oczekujemy argumentów.

```
def wypisz_liste(lista: list):  
    for element in lista:  
        print(element)
```

↓
Dzięki temu w definicji funkcji widzimy, jakiego typu funkcja chce argument.

Domyślne wartości argumentów

A co jeśli chcemy, żeby jakieś argumenty, jeśli nie zostaną podane, miały domyślną wartość?

Np. mamy taką funkcję:

```
def wypisz_tekst(tekst, ile_razy = 1):  
    print(tekst*ile_razy)
```

Jeśli nie podamy, ile razy chcemy wypisać tekst, wypisze się tylko raz (bo domyślna wartość to 1).

Domyślna wartość argumentu - przykład

```
wypisz_tekst('mama')  
wypisz_tekst('tata', 10)
```

W pierwszym wypadku argument **ile_razy** będzie równy 1, w drugim wypadku 10.

Zadania

Zadania i ta prezentacja na
github.com/jerzyklos/sobota

$$d(A, B) = \sqrt{(x_A - x_B)^2 + (y_A - y_B)^2}.$$

Wzór na odległość w 2D

$$d(A, B) = \sqrt{(x_A - x_B)^2 + (y_A - y_B)^2 + (z_A - z_B)^2}$$

Wzór na odległość w 3D