Funkcje

Funkcje pozwalają na wielokrotne używanie tego samego kodu. Znacznie poprawiają także czytelność kodu i go porządkują.

Definiowanie funkcji

```
def hello():
    print('hello world')
```

Konwencja nazewnicza funkcji

- CamelCase? Nie?! Używanie `_` w nazwach (snake_case)
- Funkcje o nazwie zaczynającej się od `_` przez konwencję są traktowane jako prywatne (w Pythonie nie ma private/protected/public).
- Funkcje o nazwie zaczynającej się od `__` i kończących się na `__` przez konwencję są traktowane jako systemowe.
- Nazwy opisowe funkcji

Argumenty do funkcji

Argumenty funckji to wartości na których ta funckja wykonuje operacje. W idealnym przypadku wartość wyjściowa funkcji powinna zależeć jedynie od jej argumentów.

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

>>> dodaj(1, 2)
3
```

Nazwy argumentów

Każdy argument ma swoją nazwę, przez którą uzyskujemy dostęp do wartości argumentu w ciele funkcji. Ta nazwa może też być używana do przypisania wartości przy wywołaniu funkcji.

```
>>> dodaj(a=1, b=2)
3

>>> podziel(a, b):
...     return a/b

>>> podziel(a=1, b=2)
0.5

>>> podziel(b=2, a=1)
0.5
```

Argumenty z wartością domyślną

Argument funkcji może mieć także wartość domyślną, z której funkcja skorzysta jeżeli użytkownik nie zdefiniuje tego argumentu.

```
>>> def hello(tekst='hello world'):
...     print(tekst)

>>> hello(tekst='alloha')
alloha

>>> hello()
hello world

>>> def convert(value, to='bin'):
...     if to=='bin':
...         return bin(value)
...     elif to=='hex':
...         return hex(value)
...     elif to=='oct':
...         return oct(value)
...     else:
...         raise ValueError('`to` should be either bin, hex or oct!!')
```

Zwracanie wartości

Zwracanie wartości prostych

```
def foo1():
    return True

def foo2():
    return None

def foo3():
    return 'bar'

def foo4():
```

```
    return [10, 20]

def foo5():
    return foo1

def foo6():
    pass

def foo7():
    return 10, 20, 30, 5, 'a'

def foo8():
    return {'imie': 'Ivan', 'nazwisko': 'Ivanovic'}
```

Zwracanie typów złożonych

```
def foo9():
    return [
        {'imie': 'Max', 'nazwisko': 'Peck'},
        {'imie': 'Ivan', 'nazwisko': 'Ivanovic'},
        {'imie': 'José', 'nazwisko': 'Jiménez'}]
```

Rozpakowywanie wartości zwracanych

```
>>> napiece, natezenie, *args = foo7()
>>> napiecie, *_ = foo7()
```

```
>>> value, _ = function()
>>> value, *args = function()
```

Operator `*` i `**`

Argumenty `*args`, `**kwargs`

Użycie operatora * przy definicji funkcji powoduje umożliwienie przekazywanie do funkcji dodatkowych parametrów anonimowych. Zazwczaj zmienna, która jest przy tym operatorze nazywa się `*args` (arguments) Użycie operatora `**` przy definicji funkcji powoduje umożliwienie przekazywania do niej dodatkowych argumentów nazwanych. Zazwczaj zmienna, która jest przy tym operatorze nazywa się `**kwargs` (keyword arguments)

```
def foo(a, *args, **kwargs):
    print(f"zmienna a: {a}")
    print(f"zmienna args: {args}")
    print(f"zmienna kwargs: {kwargs}")
```

Przy wywołaniu funkcji

Wywołując powyższą funkcję z argumentami:

```
>>> foo(1, 2, 3, 4, c=5, d=6)
zmienna a: 1
zmienna args: (2, 3, 4)
zmienna kwargs: {'c': 5, 'd': 6}
```

Sprawi, że wewnątrz funkcji będziemy mieli dostępną zmienną `a` o wartości 1, zmeinną `args`, zawierającą listę elementów (2, 3, 4) oraz zmienną słownikową `kwargs`, która ma klucze 'c' i 'd', które przechowują wartości, odpowiednio, 5 i 6.

```
def bar():
    return range(0, 5)

jeden, dwa, *reszta = bar()

print(jeden, dwa, reszta)


def foobar(a, b, *args):
    print(locals())

foobar(1, 2, 5, 7)


def foobar(a, b, **kwargs):
    print(locals())

foobar(1, 2, c=5, d=7)
```

Inne przykładowe zastosownaie operatorów `*` i `**` polega na wykorzystaniu ich przy wywołaniu funkcji. Wtedy, wykorzystując operator `*`, kolejne elementy listy albo krotki będą przekazane jako kolejne argumenty funkcji, a wykorzystując operator `**` kolejne elementy zmiennej słownikowej będą przekazane jako nazwane argumenty. Oznacza to, że na przykład argument `x` funkcji, przyjmie wartość `dict_vec['x']`.

```
def myfunc(x, y, z):
    print(x, y, z)

tuple_vec = (1, 0, 1)
dict_vec = {'y': 1, 'x': 0, 'z': 1}

>>>  myfunc(*tuple_vec)
1, 0, 1

>>> myfunc(**dict_vec)
0, 1, 1
```

Przykładowe zastosowanie

```
class Osoba:
    first_name = 'Max'
    last_name = 'Peck'

    def __str__(self):
        return '{first_name} {last_name}'.format(**self.__dict__)
```

```
def create_or_update():
    return True, [
        {'id': 1, 'imie': 'Ivan', 'nazwisko': 'Ivanovic'},
        {'id': 2, 'imie': 'José', 'nazwisko': 'Jiménez'},
    ], 10, str('asd')


czy_utworzone, *args  = create_or_update()

print(czy_utworzone)
```

Zadania kontrolne

Konwersja liczby na zapis słowny

Napisz program `numer.py`, który zamieni wprowadzony przez użytkownika ciąg cyfr na formę tekstową:

- znaki nie będące cyframi mają być ignorowane
- konwertujemy cyfry, nie liczby, a zatem:
  - 911 to "dziewięć jeden jeden"
  - 1100 to "jeden jeden zero zero"
- Napisz testy sprawdzające przypadki brzegowe.

```
>>> int_to_str(999)
'dziewiećset dziewięćdziesiąt dziewięć'
>>> int_to_str(127.32)
'sto dwadzieścia siedem i trzydzieści dwa setne'
```

Zakres

: - 6 cyfr przed przecinkiem - 5 cyfr po przecinku

Rzymskie

Zadanie 1

: Napisz program, który przeliczy wprowadzoną liczbę rzymską na jej postać dziesiętną.

Zadanie 2

: Zrób drugą funkcję, która dokona procesu odwrotnego.