Zaawansowane programowanie w Pythonie

Wykład 2

dr Agnieszka Zbrzezny

28 października 2024

Funkcje w języku Python – powtórzenie

Funkcje

- Funkcja to nazwana sekwencja instrukcji realizująca specyficzne zadanie.
- Celem definiowania funkcji jest podzielenie programu na fragmenty odpowiadające sposobowi myślenia o danym problemie.
- Funkcje stosuje się, aby uprościć program i zwiększyć jego czytelność.
- Funkcje ukrywają złożone obliczenia za pomocą jednej instrukcji – wywołania funkcji.
- Utworzenie funkcji pozwala zmniejszyć program poprzez eliminację powtarzającego się kodu.

Funkcje w języku Python – powtórzenie

Składnia

- Składnia definicji funkcji jest następująca:

```
def nazwa(argumenty):
    zestawInstrukcji
```

- Pierwsza linia funkcji zaczyna się od słowa kluczowego **def** a kończy dwukropkiem. Linia ta nazywa się nagłówkiem funkcji.
- Nazwa funkcji musi być identyfikatorem (nie może być słowem kluczowym).
- Lista argumentów może być pusta lub może składać się z dowolnej ilości argumentów oddzielonych przecinkami.

Funkcje w języku Python – powtórzenie

Funkcje jako obiekty

- W trakcie wykonywania polecenia **`def`** następuje utworzenie obiektu funkcji.
- Następnie do zmiennej, której nazwą jest nazwa funkcji, przypisywane jest odniesienie do obiektu funkcji.
- Z faktu, że funkcje są obiektami wynika iż mogą być one:
  - przechowywane w kolekcjach zbiorów danych,
  - przekazywane jako argumenty do innych funkcji,

Funkcje w języku Python – powtórzenie

Komentarze dokumentacyjne

- Bezpośrednio po nagłówku funkcji może, ale nie musi, wystąpić komentarz dokumentacyjny.
- Komentarze dokumentacyjne ujmuje się zazwyczaj w potrójne cudzysłowy, """ dzięki czemu mogą one zajmować dowolnie wiele linii.
- Składnia definicji funkcji uwzględniająca komentarz dokumentacyjny jest następująca:

```
def nazwaFunkcji(argumenty):
    """Opcjonalny komentarz dokumentacyjny
    mogący zajmować wiele linii"""
    zestawInstrukcji
```

Funkcje w języku Python – powtórzenie

Wartości zwracane przez funkcję

- Każda funkcja Pythona posiada wartość zwrotną. Domyślnie wartością zwrotną jest **None**, chyba że z wnętrza funkcji zostanie zwrócona inna wartość za pomocą instrukcji o następującej składni:

```
return wyrażenie
```

- Wykonanie powyższej instrukcji powoduje obliczenia wartości wyrażenia `wyrażenie`, po czym wartość ta staje się wartością zwrotną danej funkcji.
- Wartość zwrotna może być pojedynczą wartością bądź krotką wartości.
- Wartość zwrotna może zostać zignorowana przez wywołującego funkcję; będzie ona wówczas odrzucona.

Argumenty funkcji

Argumenty pozycyjne

```
def main():
    info("Bob", 34)
    info("Bob", age=34)
    info(name="Bob", age=34)
    info(age=34, name="Bob")
    # SyntaxError: positional argument
    # follows keyword argument
    # info(name = "Bob", 34)

    # TypeError: info() got multiple values
    # for argument 'name'
    # info(34, name = "Bob")

def info(name, age):
    print(name, age)
```

Argumenty funkcji

Argumenty pozycyjne

```
def main():
    a = float(input("Podaj liczbę rzeczywistą: "))
    b = float(input("Podaj liczbę rzeczywistą: "))
    print("Średnia wynosi ", srednia(a, b))
    print("Średnia wynosi ", srednia(x = a, y = b))
    print("Średnia wynosi ", srednia(y = b, x = a))
    print("Średnia wynosi ", srednia(a, y = b))
    # print("Średnia wynosi ", srednia(x = a, b))
    # print("Średnia wynosi ", srednia(a, x = b))

def srednia(x, y): return (x + y) / 2

main()
```

Argumenty funkcji

Argumenty pozycyjne

```
#!/usr/bin/env python3

def main():
    a, b = 3, 5
    print(a, b)
    zamiana(a, b)
    print(a, b)

def zamiana(x, y):
    x, y = y, x

if __name__ == "__main__":
    main()
```

Argumenty funkcji

Argumenty domyślne

```
def main():
    a = float(input("Podaj liczbę rzeczywistą: "))
    b = float(input("Podaj liczbę rzeczywistą: "))
    print("Średnia wynosi ", srednia(a, b, a + b))
    print("Średnia wynosi ", srednia(x = a, y = b))
    print("Średnia wynosi ", srednia(y = b, x = a))
    print("Średnia wynosi ", srednia(a, b, z = 6))
    print("Średnia wynosi ", srednia(a, y = b, z = 6))

def srednia(x, y, z = 0):
    return (x + y + z) / 3

main()
```

Argumenty funkcji

Argumenty specjalne

- Na końcu listy argumentów pozycyjnych można opcjonalnie użyć jednej lub obu form specjalnych `*args` i `**kwargs`.
- Nie ma nic szczególnego w nazwach `args` i `kwargs` – można użyć dowolnego identyfikatora w każdej specjalnej formie.
- Jeżeli obie formy są obecne, to forma z dwiema gwiazdkami musi być druga.
- Forma `*args` określa, że wszelkie dodatkowe argumenty pozycyjne dla wywołania są gromadzone w (ewentualnie pustej) krotce, i są związane w wywołaniu z nazwą `args`.
- Podobnie `**kwargs` określa, że wszelkie dodatkowe nazwane argumenty zostają zebrane w (ewentualnie pustym) słowniku, którego elementami są nazwy i wartości tych argumentów i są związane z nazwą `kwargs`.

Argumenty funkcji

Dowolna liczba argumentów

```
def func(*args, **kwargs):
    print("args =", args)      # nienazwane argumenty
    print("kwargs =", kwargs) # nazwane argumenty
    print()

func()
func(3)
func(3, 5)
func(jan = 31)
func(jan = 31, feb = 28)
func(jan = 31, feb = 28, mar = 31)
func(3, jan = 31, feb = 28)
func(3, 5, jan = 31, feb = 28)
func(3, 5, 8, jan = 31, feb = 28)
```

Argumenty funkcji

Rozpakowywanie argumentów wywołania

- W listach argumentów wywołania funkcji i metod można zastosować specjalną składnię w celu „rozpakowania” dowolnej liczby argumentów.
- Jeżeli `argpos` jest obiektem iterowalnym, a `argklucz` jest słownikiem, to następująca instrukcja

```
func(*argpos, **argklucz)
```

  wywoła funkcję `func` z argumentami pozycyjnymi pobranymi z obiektu `argpos` oraz z argumentami kluczowymi pobranymi ze słownika `argklucz`.
- Składnię tę dodano po to, aby była symetryczna ze składnią argumentów nagłówka funkcji,

Argumenty funkcji

Rozpakowywanie argumentów wywołania (stars.py)

```
def func(*args, **kwargs):
    print("args =", args)      # nienazwane argumenty
    print("kwargs =", kwargs) # nazwane argumenty
    print()

func(3, 5, 8, jan = 31, feb = 28, mar = 31, apr = 30)

a = [3, 5, 8]
d = {"jan" : 31, "feb" : 28, "mar" : 31, "apr" : 30}
func(*a, **d)
func(*d, **d)

s = "Python"
func(*s, **d)
func(*"Python", **d)
func(*range(5), **dict())
```

Argumenty funkcji

Dowolna liczba nienazwanych argumentów pozycyjnych

```
def main():
    a = float(input("Podaj liczbę rzeczywistą: "))
    b = float(input("Podaj liczbę rzeczywistą: "))
    c = float(input("Podaj liczbę rzeczywistą: "))
    print("Średnia wynosi ", srednia(a, b, c))

def srednia(*numbers):
    if len(numbers) == 0: return None
    suma = 0
    for a in numbers: suma += a
    return suma / len(numbers)

main()
```

Argumenty funkcji

Dowolna liczba argumentów nazwanych

```
>>> d = dict(jan = 31)
>>> d
{'jan': 31}
>>> d = dict(jan = 31, feb = 28)
>>> d
{'jan': 31, 'feb': 28}
>>> d = dict(jan = 31, feb = 28, mar = 31)
>>> d
{'jan': 31, 'feb': 28, 'mar': 31}
>>> d = dict(jan = 31, feb = 28, mar = 31, apr = 30)
>>> d
{'jan': 31, 'feb': 28, 'mar': 31, 'apr': 30}
>>>
```

Argumenty funkcji

Dowolna liczba argumentów nazwanych

```
def main():
    pokaz_sume(1, 2, a = 5, b = 8)

def pokaz_sume(x, y, **inne):
    print(
        "x: {0}, y: {1}, Klucze w 'inne': {2}".
        format(x, y, list(inne.keys())))
    suma = 0
    for k in inne:
        suma = suma + inne[k]
    print("Suma wartości w 'inne' wynosi {0}".
        format(suma))

main()
```

Argumenty funkcji

Dowolna liczba argumentów nazwanych

```
def main():
    s = "My surname is {0}.\n"
    s = s + "My first name is {first}.\n"
    s = s + "My middle name is {middle}.\n"
    s = s + "My third name is {third}.\n"
    print(s.format("Zbrzezny",
        first = "Andrzej",
        middle = "Robert",
        third = "Tomasz")
    )

main()
```

Argumenty funkcji

Ogólna postać definicji funkcji

```
def func(a, b, *args, c, d = 6, **kwargs):
    # a i b są obowiązkowymi argumentami pozycyjnymi
    print("a =", a, "b =", b )
    # args jest krotką nienazwanych
    # argumentów pozycyjnych
    print("args =", args)
    # c i d są argumentami, które są przekazywane
    # wyłącznie za pomocą słów kluczowych
    print("c =", c, "d =", d)
    # kwargs jest słownikiem
    print("kwargs =", kwargs, "\n")

func(1, 2, 3, 4, 5, c = 5, k1 = 11, k2 = 12)
func(1, 2, 3, 4, 5, c = 5, k1 = 11, k2 = 12, d = 7)
```

Składnia definicji funkcji

Argumenty tylko pozycyjne (począwszy od wersji 3.8)

- Istnieje nowa składnia argumentów funkcji `/`, która wskazuje, że niektóre argumenty funkcji muszą być określone pozycyjnie i nie mogą być używane jako argumenty będące słowami kluczowymi.
- W poniższym przykładzie argumenty `a` i `b` są tylko pozycyjne, podczas gdy `c` lub `d` mogą być argumentami pozycyjnymi lub argumentami będącymi słowami kluczowymi, natomiast `e` oraz `f` muszą być słowami kluczowymi:

```
def f(a, b, /, c, d, *, e, f):
    print(a, b, c, d, e, f)
```

- Poniżej znajduje się prawidłowe wywołanie:

```
f(10, 20, 30, d=40, e=50, f=60)
```

Składnia definicji funkcji

Argumenty tylko pozycyjne

- Zakładając tę samą definicję funkcji

```
def f(a, b, /, c, d, *, e, f):
    print(a, b, c, d, e, f)
```

zauważmy, że poniższe dwa wywołania są niepoprawne:

```
# b cannot be a keyword argument
f(10, b=20, c=30, d=40, e=50, f=60)
# e must be a keyword argument
f(10, 20, 30, 40, 50, f=60)
```

Składnia definicji funkcji

Argumenty tylko pozycyjne

- Jednym z przypadków użycia tej notacji jest to, że umożliwia ona czystym funkcjom Pythona pełną emulację zachowań istniejących funkcji zakodowanych w języku C.
- Przykładowo, funkcja wbudowana `divmod` nie akceptuje argumentów będących słowami kluczowymi:

```
def divmod(a, b, /):
    "Emulate the built in divmod() function"
    return (a // b, a % b)
```

Składnia definicji funkcji

Argumenty tylko pozycyjne

- Innym przykładem użycia jest wykluczenie argumentów będących słowami kluczowymi, gdy nazwa argumentu nie jest pomocna.
- Przykładowo, wbudowana funkcja `len` ma następującą sygnaturę: `len(obj, /)`. To wyklucza niewygodne wywołania, takie jak:

```
# Argument "obj" będący słowem kluczowym
# pogarsza czytelność
len(obj="hello")
```

Składnia definicji funkcji

Argumenty tylko pozycyjne

- Kolejną zaletą oznaczania argumentu jako tylko pozycyjnego jest możliwość zmiany nazwy argumentu w przyszłości bez ryzyka złamania kodu klienta.
- Przykładowo, w module `statistics` nazwy argumentów funkcji `correlation` mogą zostać zmienione w przyszłości. Jest to możliwe dzięki następującej specyfikacji funkcji:

```
def quantiles(dist, /, *, n=4, method='exclusive')
    ...
```

Składnia definicji funkcji

Argumenty tylko pozycyjne

- Ponieważ argumenty po lewej stronie symbolu / nie są widoczne jako możliwe słowa kluczowe, nazwy argumentów pozostają dostępne do użycia w **kwargs:

```
def f(a, b, /, **kwargs):
    print(a, b, kwargs)

# a and b are used in two ways
f(10, 20, a=1, b=2, c=3)

# 10 20 {'a': 1, 'b': 2, 'c': 3}
```

Składnia definicji funkcji

Argumenty tylko pozycyjne

- To znacznie upraszcza implementację funkcji i metod, które muszą akceptować dowolne argumenty nienazwane.
- Oto przykładowy fragment kodu w module `collections`:

```
class Counter(dict):
    def __init__(self, iterable=None, /, **kwds):
        # Note "iterable" is a possible
        # keyword argument
```

- A to przykład użycia:

```
>>> from collection import Counter
>>> c = Counter("abba", iterable = 1, b = 2, c = 3)
>>> c
Counter({'b': 4, 'c': 3, 'a': 2, 'iterable': 1})
```

Listy składane

Uwagi wstępne

- Listy składane (ang. list comprehension) to konstrukcja, która pozwala na zwięzłe tworzenie list.
- Typowym zastosowaniem jest utworzenie listy, której elementy są wynikiem zastosowania pewnej operacji do każdego elementu innej sekwencji lub obiektu iterowalnego.
- Innym typowym zastosowaniem jest utworzenie listy z tych elementów pewnej sekwencji, które spełniają pewne warunki.
- Listę składaną tworzy się z nawiasów kwadratowych zawierających wyrażenie, po którym występuje klauzula `for`, a następnie zero lub więcej klauzul `for` lub `if`.

Listy składane

Przykład

```
>>> squares = []
>>> for x in range(10):
...     squares.append(x**2)
...
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Przykład

```
>>> squares = [x**2 for x in range(10)]
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Listy składane

Przykład

```
>>> znaki = []
>>> for c in "Ala ma psa":
...     if c != " ":
...         znaki.append(c)
...
>>> znaki
['A', 'l', 'a', 'm', 'a', 'p', 's', 'a']
```

Przykład

```
>>> znaki = [c for c in "Ala ma psa" if c != " "]
>>> znaki
['A', 'l', 'a', 'm', 'a', 'p', 's', 'a']
```

Listy składane

Przykład

```
>>> pary = []
>>> for x in [1, 2, 3]:
...     for y in [3, 1, 4]:
...         if x != y:
...             pary.append((x, y))
...
>>> pary
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

Przykład

```
>>> [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

Listy składane

Przykład

```
>>> wektor = [-4, -2, 0, 2, 4]
>>> # tworzy listę zawierającą podwojone wartości
>>> [x * 2 for x in wektor]
[-8, -4, 0, 4, 8]
```

Przykład

```
>>> # tworzy listę nie zawierającą wartości ujemnych
>>> [x for x in wektor if x >= 0]
[0, 2, 4]
```

Listy składane

Przykład

```
>>> wektor = [-4, -2, 0, 2, 4]
>>> # stosuje funkcję do wszystkich elementów
>>> [abs(x) for x in wektor]
[4, 2, 0, 2, 4]
```

Przykład

```
>>> # wywołuje metodę dla wszystkich obiektów listy
>>> koszyk = [' banan', ' śliwka   ', '   kiwi ']
>>> [owoc.strip() for owoc in koszyk]
['banan', 'śliwka', 'kiwi']
```

Listy składane

Przykład

```
>>> # tworzy listę par (liczba, kwadrat_liczby)
>>> # nawiasy obejmujące parę są konieczne
>>> [(x, x**2) for x in range(6)]
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
```

Przykład

```
>>> # spłaszcza listę używając dwóch klauzul 'for'
>>> wektor = [[1,2,3], [4,5,6], [7,8,9]]
>>> [num for elem in wektor for num in elem]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Listy składane

Przykład

```
>>> # wyrażenie generujące może zawierać złożone
>>> # podwyrażenia i zagnieżdżone wywołania funkcji
>>> from math import pi
>>> [str(round(pi, j)) for j in range(1, 6)]
['3.1', '3.14', '3.142', '3.1416', '3.14159']
```

Przykład (Tablicowanie)

```
from math import pi, sin
left = round((-2.5 * pi) * 50)
right = round((2.5 * pi) * 50)
points = [ (j, round(sin(j / 50) * 100))
         for j in range(left, right + 1) ]
print(points)
```

Listy składane

Zagnieżdżone listy składane

- Początkowe wyrażenie w liście składanej może być dowolnym wyrażeniem, w tym zawierającym inna listę składaną.

Przykład

```
>>> matrix = [
...     [1, 2, 3, 4],
...     [5, 6, 7, 8],
...     [9, 10, 11, 12],
... ]
>>> # Poniższe wyrażenie utworzy macierz transponowaną:
>>> [[row[col] for row in matrix] for col in range(4)]
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

Listy składane

Zagnieżdżone listy składane

- Konstrukcja listy składanej z poprzedniego slajdu jest równoważna ciągowi instrukcji pokazanemu w poniższym przykładzie.

Przykład

```
>>> transponed = []
>>> for col in range(4):
...     transponed.append([row[col] for row in matrix])
...
>>> transponed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

Listy składane

Zagnieżdżone listy składane

- Przykład z poprzedniego slajdu jest równoważny poniższemu przykładowi.

Przykład

```
>>> transponed = []
>>> for j in range(4):
...     # implementacja zagnieżdżonej listy
...     transponed_row = []
...     for row in matrix:
...         transponed_row.append(row[j])
...     transponed.append(transponed_row)
...
>>> transponed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

Listy składane

Zagnieżdżone listy składane

- O ile to możliwe, lepiej jest użyć funkcji wbudowanej zamiast skomplikowanej konstrukcji z poprzedniego przykładu.
- Wbudowana funkcja `zip` pobiera odpowiadające sobie elementy z jednego lub więcej obiektu iterowalnego i tworzy z nich krotki aż do wyczerpania najkrótszego obiektu iterowalnego.

Przykład

```
>>> list(zip([1, 2, 3], [4, 5], [6, 7, 8, 9]))
[(1, 4, 6), (2, 5, 7)]
>>> list(zip([[1, 2, 3], [4, 5], [6, 7, 8, 9]]))
[([1, 2, 3],), ([4, 5],), ([6, 7, 8, 9],)]
>>> list(zip(*[[1, 2, 3], [4, 5], [6, 7, 8, 9]]))
[(1, 4, 6), (2, 5, 7)]
```

Listy składane

Zagnieżdżone listy składane

- Funkcji `zip` możemy użyć w celu wykonania operacji transponowania macierzy pamiętając, że w macierzy transponowanej wiersze będę reprezentowane przez krotki, a nie przez listy.
- W razie potrzeby krotki można przekonwertować na listy.

Przykład

```
>>> transponed = list(zip(*matrix))
[(1, 5, 9), (2, 6, 10), (3, 7, 11), (4, 8, 12)]
>>> transponed = [list(row) for row in transponed]
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```