Wprowadzenie do języka Python - Wykład 4

Funkcje

Funkcje

```
def functionname( parameters ):
   "function_docstring"
   function_suite
   return [expression]
```

```
1  def printme(str):
2      """Funkcja wyświetlająca string"""
3      print(str)
4      return
5
6
7  printme("abc")
8  print(printme.__doc__)
```

```
abc
Funkcja wyświetlająca string
```

Przekazywanie przez referencję

Ale trzeba pamiętać podział typów:

```
1  def foo(a):
2      a += 1
3      print("w", a)
4
5
6  a = 7
7  print(a)
8  foo(a)
9  print(a)
```

```
7
w 8
7
```

```python
def changeme(lista):
    print("Przed zmianą: ", lista)
    lista[2] = 50
    print("Po zmianie: ", lista)
    return


mylist = [10, 20, 30]
changeme(mylist)
print("Poza funkcją: ", mylist)
```

```
Przed zmianą:  [10, 20, 30]
Po zmianie:  [10, 20, 50]
Poza funkcją:  [10, 20, 50]
```

```python
def changeme(lista):
    lista = [2, 3, 4]
    print("Wewnątrz funkcji: ", lista)
    return


lista = [10, 20, 30]
changeme(lista)
print("Poza funkcją: ", lista)
```

```
Wewnątrz funkcji:  [2, 3, 4]
Poza funkcją:  [10, 20, 30]
```

```
1  def changeme():
2      global lista
3      lista = [2, 3, 4]
4      print("Wewnątrz funkcji: ", lista)
5      return
6
7
8  changeme()
9  print("Poza funkcją: ", lista)
```

```
Wewnątrz funkcji:  [2, 3, 4]
Poza funkcją:  [2, 3, 4]
```

Obowiązkowy argument (ang. positional argument)

```
1 def printme(str):
2     print(str)
3     return
4
5
6 printme()
```

```
## TypeError: printme() missing 1 required
positional argument: 'str'
```

Keyword argument

```
1 def kwadrat(a):
2     return a*a
3
4
5 print(kwadrat(a=4))
```

```
16
```

Domyślny argument

```
1 def sumsub(a, b, c=0, d=0):
2     return a - b + c - d
3
4
5 print(sumsub(12, 4))
6 print(sumsub(3, 4, 5, 7))
```

```
8
-3
```

```python
def srednia(first, *values):
    return (first + sum(values)) / (1 + len(values))


print(srednia(2, 3, 4, 6))
print(srednia(45))
```

```
3.75
45.0
```

```python
def f(**kwargs):
    print(kwargs)


f()
f(pl="Polish", en="English")
```

```
{}
{'pl': 'Polish', 'en': 'English'}
```

Inne symbole

- symbol / oznacza, że wcześniejsze argumenty są pozycyjne
- symbol * oznacza, że późniejszej argumenty są typu keyword

```
1 def f(a, b, /, c, d, *, e, f):
2     print(a, b, c, d, e, f)
3
4
5 f(10, 20, 30, d=40, e=50, f=60)
```

```
10 20 30 40 50 60
```

Funkcje matematyczne

Link do dokumentacji

https://docs.python.org/3/library/math.html

```
1 import math
2
3 a=0
4 b=math.sin(2*math.pi)
5 print(b)
6 print(math.isclose(a,b, rel_tol=1e-09, abs_tol=1e-09))
```

```
-2.4492935982947064e-16
True
```

Wyrażenia lambda

Skąd nazwa? https://pl.wikipedia.org/wiki/Rachunek_lambda

Wyrażenie

```
1 def identity(x):
2     return x
```

zapisujemy jako

```
lambda x: x
```

Jeśli uruchomimy to w konsoli, to możemy wywołać

```
1 _(5)
```

To tzw. przykład funkcji anonimowej.

By uruchomić to też w skrypcie, to potrzebujemy argumentu:

```
1  print((lambda x: x + 1)(2))
```

lub

```
1  add_one = lambda x: x + 1
2  print(add_one(2))
```

Możemy też tworzyć złożenia funkcji (funkcje wyższych rzędów):

```
1 high_ord_func = lambda x, func: x + func(x)
2 print(high_ord_func(2, lambda x: x * x))
3 print(high_ord_func(2, lambda x: x + 3))
```

```
6
7
```

Możemy również skorzystać z różnych opcji argumentów jak dla funkcji:

```
1 print((lambda x, y, z: x + y + z)(1, 2, 3))
2 print((lambda x, y, z=3: x + y + z)(1, 2))
3 print((lambda x, y, z=3: x + y + z)(1, y=2))
4 print((lambda *args: sum(args))(1,2,3))
5 print((lambda **kwargs: sum(kwargs.values()))(one=1, two=2, three=3))
6 print((lambda x, *, y=0, z=0: x + y + z)(1, y=2, z=3))
```

```
6
6
6
6
6
6
```

Typ

```
1 string = 'abc'
2 print(lambda string: string)
```

```
<function <lambda> at 0x000001C64E7DC3A0>
```

Różnice względem zwykłej funkcji:

```
1  def cube(y):
2      print(f"Finding cube of number:{y}")
3      return y * y * y
4
5  lambda_cube = lambda num: num ** 3
6  print("invoking function defined with def keyword:")
7  print(cube(30))
8  print("invoking lambda function:", lambda_cube(30))
```

```
invoking function defined with def keyword:
Finding cube of number:30
27000
invoking lambda function: 27000
```

Przykłady praktyczne

```
1  r = lambda a: a + 15
2  print(r(10))
3  r = lambda x, y: x * y
4  print(r(12, 4))
```

```
25
48
```

```python
subject_marks = [('English', 88), ('Science', 90), ('Maths', 97), ('Social
print("Original list of tuples:")
print(subject_marks)
subject_marks.sort(key=lambda x: x[1])
print("\nSorting the List of Tuples:")
print(subject_marks)
```

```
Original list of tuples:
[('English', 88), ('Science', 90), ('Maths', 97), ('Social sciences', 82)]

Sorting the List of Tuples:
[('Social sciences', 82), ('English', 88), ('Science', 90), ('Maths', 97)]
```

```python
nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print("Original list of integers:")
print(nums)
print("\nEven numbers from the said list:")
even_nums = list(filter(lambda x: x % 2 == 0, nums))
print(even_nums)
print("\nOdd numbers from the said list:")
odd_nums = list(filter(lambda x: x % 2 != 0, nums))
print(odd_nums)
```

```
Original list of integers:
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

Even numbers from the said list:
[2, 4, 6, 8, 10]

Odd numbers from the said list:
[1, 3, 5, 7, 9]
```

```
array_nums1 = [1, 2, 3, 5, 7, 8, 9, 10]
array_nums2 = [1, 2, 4, 8, 9]
print("Original arrays:")
print(array_nums1)
print(array_nums2)
result = list(filter(lambda x: x in array_nums1, array_nums2))
print("\nIntersection of the said arrays: ", result)
```

```
Original arrays:
[1, 2, 3, 5, 7, 8, 9, 10]
[1, 2, 4, 8, 9]

Intersection of the said arrays:  [1, 2, 8, 9]
```

Wywoływalne obiekty tzw. calle

W Pythonie callable to termin odnoszący się do obiektów, które można “wywołać” (ang. call) jako funkcje. W praktyce oznacza to, że obiekt może być użyty z nawiasami okrągłymi i potencjalnie przyjmować argumenty.

Przykład prosty:

```
1 def greet(name):
2     return f"Hello, {name}!"
3
4
5 print(greet("Tom"))
```

```
Hello, Tom!
```

```python
greet_lambda = lambda name: f"Hello, {name}!"
print(greet_lambda("Anna"))
```

```
Hello, Anna!
```

```python
greet_lambda = lambda name: f"Hello, {name}!"
print(callable(greet_lambda))  # True
print(callable(42))  # False
```

```
True
False
```

Słowo kluczowe `yield` - leniwe wyrzucanie

Słowo kluczowe `yield` jest używane w funkcjach generatorów. Generator to specjalny rodzaj iteratora, który pozwala na iterowanie (przechodzenie) przez sekwencję wartości w sposób leniwy, co oznacza, że wartości są generowane na żądanie, a nie na początku. (leniwe wyrzucanie)

Gdy generator napotyka słowo kluczowe yield, zwraca wartość po yield i wstrzymuje wykonywanie, pamiętając swój stan. Wywołanie funkcji next() na iteratorze sprawia, że generator wznowi działanie od miejsca, w którym się zatrzymał, aż napotka kolejne słowo kluczowe yield lub zakończy działanie.

```python
def simple_number_generator(max_number):
    num = 1
    while num <= max_number:
        yield num
        num += 1

# Użycie generatora
for number in simple_number_generator(5):
    print(number)
```

```
1
2
3
4
5
```

```
def even_numbers_generator(max_number):
    num = 2
    while num <= max_number:
        yield num
        num += 2


for even_number in even_numbers_generator(10):
    print(even_number)
```

```
2
4
6
8
10
```

```
def infinite_sequence():
    num = 0
    while True:
        yield num
        num += 1


gen = infinite_sequence()
print(next(gen))
print(next(gen))
print(next(gen))
print(next(gen))
```

```
0
1
2
3
```

```python
import math

def is_prime(num):
    if num <= 1:
        return False
    for i in range(2, int(math.sqrt(num)) + 1):
        if num % i == 0:
            return False
    return True

def prime_number_generator():
    num = 2
    while True:
        if is_prime(num):
            yield num
        num += 1

# Użycie generatora
primes = prime_number_generator()
for _ in range(10):  # Wypiszemy pierwsze 10 liczb pierwszych
    print(next(primes))
```

```
2
3
5
7
11
13
17
19
23
29
```

Funkcja a type hinting

```
1  def stringify(num: int) -> str:
2      return str(num)
3
4
5  def plus(num1: int, num2: int) -> int:
6      return num1 + num2
```

```
1 def show(value: str, excitement: int = 10) -> None:
2     print(value + "!" * excitement)
```

Typowanie tzw. calli

```
1  from typing import Callable
2
3
4  def f(a: int, b: float) -> float:
5      return a + b
6
7
8  x: Callable[[int, float], float] = f
```

Typowanie generatorów:

```python
from typing import Iterator


def gen(n: int) -> Iterator[int]:
    i = 0
    while i < n:
        yield i
        i += 1
```

Kilka różnych typów:

```
1  from typing import Union, Optional
2
3
4  def send_email(address: Union[str, list[str]],
5                 sender: str,
6                 cc: Optional[list[str]],
7                 bcc: Optional[list[str]],
8                 subject: str = '',
9                 body: Optional[list[str]] = None
10                ) -> bool:
11     pass
```

```python
def quux(x: int, /,  *, y: int) -> None:
    pass


quux(3, y=5)  # Ok
# quux(3, 5)  # error: Too many positional arguments for "quux"
# quux(x=3, y=5)  # error: Unexpected keyword argument "x" for "quux"
```

Bibliografia

- https://realpython.com/python-print/, dostęp online 5.01.2022
- https://www.programiz.com/python-programming/operators, dostęp online 5.01.2022
- https://realpython.com/python-conditional-statements/, dostęp online 5.01.2022
- https://realpython.com/python-for-loop/, dostęp online 5.01.2022
- https://realpython.com/python-while-loop/, dostęp online 5.01.2022
- https://towardsdatascience.com/a-guide-to-python-comprehensions-4d16af68c97e, dostęp online 20.02.2023.
- https://mypy.readthedocs.io/, dostęp online 20.03.2023.
- https://realpython.com/introduction-to-python-generators/, dostęp online 22.03.2023.