Wstęp do programowania
semestr zimowy 2025/2026

Dr Anna Muranova
UWM w Olsztynie

Wykład 4

Funkcje w Python

Funkcja - podprogram zwracający wynik na podstawie przekazanych argumentów.
Funkcje definiuje się używając słowa `def`. Po nim następuje nazwa identyfikująca funkcji, następnie para nawiasów, które mogą zawierać kilka nazw zmiennych (parametrów, argumentów), a na końcu dwukropek. Poniżej zaczyna się blok wyrażeń, które są częścią tej funkcji.
Ogólna składnia funkcji:

```
def nazwa_funkcji(parametr_1, parametr_2, ):
    polecenie_1
    polecenie_2
    ... i td.

    return wartosc
```

Instrukcja return powoduje zakończenie wykonywania funkcji i zwrócenie wartości.
W języku Python nazwy funkcji powinny być zgodne z konwencją `snake_case`, czyli wszystkie słowa z małych liter, oddzielone znakiem podkreślenia `_`. Ta konwencja określona jest przez twórców tego języka i większość programistów się jej trzyma. Nie jest nam ona obca, bo używamy jej również przy nazywaniu zmiennych.

Funkcja zwraca None

W Python domyślnie funkcja zwraca **None**

- ```
  def my_sum(a, b):
      a = + b

  print(my_sum(3,5))
  ```
  None
- ```
  def przywitanie():
      print ("Witaj")

  przywitanie() # Wywołanie funkcji.
  print(przywitanie() )
  a = przywitanie()
  print(a)
  ```
  Witaj
  Witaj
  None
  Witaj
  None

Funkcja z argumentami domyślanymi

Ostatni parametr lub parametry w podpisie funkcji może być przypisany argument domyślny, co oznacza, że obiekt wywołujący może pominąć argument podczas wywoływania funkcji, chyba że chcą określić inną wartość.
Domyślny argument:

```
def przywitanie_imienne(imie, zyczenia = "zdrowia" ):
    print ("Witaj, " + imie + ". Zycze Tobie " + zyczenia)

przywitanie_imienne("Jacek") # Wywołanie funkcji z domyslnym argumentem
przywitanie_imienne("Jacek", "szczęścia") # Wywołanie funkcji.
przywitanie_imienne(imie ="Jacek")
```

Witaj, Jacek. Zycze Tobie zdrowia
Witaj, Jacek. Zycze Tobie szczęścia
Witaj, Jacek. Zycze Tobie szczęścia

```
def przywitanie_imienne(zyczenia = "zdrowia", imie ):
    print ("Witaj, " + imie + ". Zycze Tobie " + zyczenia)

przywitanie_imienne("Jacek") # Wywołanie funkcji z domyslnym argumen
przywitanie_imienne("Jacek", "szczęścia")
```

Funkcje: domyślny argument

```
def suma(a, b=0, c=0, d=0):
    return a + b + c + d

print(suma(12, 4))#16
print(suma(3, 4, 5, 7))#19
```

```
def prod(a, b=1, c=1, d=1):
    return a * b * c * d

print(prod(12, 4))#48
print(prod(3, 4, 5, 7))#420
```

Funkcje: dwie funkcje z rożną ilością argumentów nie są możliwe

```
def area(l, b):
  return l * b

def area(r):
  return 3.14 * r ** 2


area(3, 4)
area(7)
```

Funkcja: dla różnych typów argumentów

```
def aFunction(a):
    if type(a) == str:
         print('you gave string as argument')
    if type(a) == int:
         print('you gave a int as argument')

aFunction('test')#you gave string as argument
aFunction(2.0)
aFunction(5)#you gave a int as argument
```

Przykłady funkcje

- Suma:

```
def suma(n):
    s = 0
    for i in range(1,n):
        s+=i
    return s

print(suma(6))#15
```

- Silnia:

```
def silnia(n):
    s = 1
    for i in range(1,n):
        s*=i
    return s

print(silnia(6))#120
```

Przykłady funkcje: zwracanie dwóch wartości

```
def sumaprod(n):
    s = 0
    p = 1
    for i in range(1,n):
        s+=i
        p*=i
    return s, p

print(sumaprod(6))
a , b = sumaprod(6)
print(a)
print(b)
a = sumaprod(6)[0]
b = sumaprod(6)[1]
print(a,' ',b)
```

```
(15, 120)
15
120
15 120
```

Funkcja `main()`

W wielu językach programowania (jak Java czy C++), nie jest możliwe, aby wyrażenia podobne do tych na końcu programu widniały ot tak, samodzielnie. Wymagane jest by były one częścią specjalnej funkcji o nazwie `main()` (funkcja główna).
Pomimo tego, że nie jest to wymagane w języku Python, nie będzie to zły pomysł, jeżeli włączymy je w logiczną strukturę programu.

```
x = 5
y = 3
print(x+y)#8
```

```
def main():
    x = 5
    y = 3
    print(x + y)

main()
```

```
8
```

Zanim interpreter języka Python uruchomi Twój program, definiuje on kilka specjalnych zmiennych. Jedną z nich jest zmienna `__name__`, do której automatycznie przypisywana jest wartość `"__main__"`w momencie, gdy dany kod uruchamiany jest niezależnie, jako samodzielny program. Z drugiej strony, gdy dany kod importowany jest z poziomu innego programu, wartość zmiennej `__name__` ustawiana jest jako nazwa tego właśnie modułu. Oznacz to, że wiemy, czy moduł uruchamiany jest jako niezależny program czy może używany jest przez inny program. Bazując na tej wiedzy, możemy zdecydować, czy uruchamiać wybraną część kodu.
Załóżmy przykładowo, że napisaliśmy szereg funkcji, wykonujących proste obliczenia matematyczne. Wywołania tych funkcji możemy zamieścić w funkcji `main`. Jest jednak bardziej prawdopodobne, że funkcje te będą zaimportowane przez inny program, do zupełnie innych celów. W tym przypadku, nie będziemy raczej chcieli wywoływać funkcji `main`.

```
def main():
    x = 5
    y = 3
    print(x + y)

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

```
8
```

W tym kodzie znajdziemy wyrażenie `if`, aby sprawdzić wartość zmiennej `__name__`. Jeżeli będzie to `__main__`, wtedy wywołana będzie funkcja main. W innym przypadku możemy założyć, że powyższy program został zaimportowany z poziomu innego programu, nie będziemy zatem chcieli wywołać funkcji `main`, gdyż ów nowy program będzie używać powyższych funkcji wedle potrzeb. Możliwość warunkowego uruchamiania funkcji `main` może się okazać bardzo potrzebna, gdy piszemy program, który potencjalnie będzie używany przez innych. Umożliwia nam też dodanie tej funkcjonalności, której użytkownik kodu nie będzie potrzebował. Najczęściej będą to procedury testowe sprawdzające poprawne funkcjonowanie funkcji.

Funkcje: nie znana ilość argumentów

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

print(suma(12, 4)) #16
```

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

def suma(a, b, c):
    return a + b + c

print(suma(12, 4))
print(suma(3, 4, 5))
```

Funkcje: *args

- ```
  def suma(a, b=0, c=0, d=0):
      return a + b + c + d

  print(suma(12, 4))
  print(suma(3, 4, 5, 7))
  ```
- ```
  def suma(*values):
      s = 0
      for i in values:
          s+=i
      return s

  print(suma(12, 4))# 16
  print(suma(3, 4, 5, 7)) #29
  ```

Funkcje wbudowane sum() i len()

- ```
  def suma(*values):
      #print(values)
      return sum(values)

  print(suma(12, 4))
  print(suma(3, 4, 5, 7))
  #print(sum(3, 4, 5, 7))błąd
  print(sum((3,4,5,7)))#omówimy później
  ```
- ```
  def srednia(*values):
      return (sum(values)) / (len(values))

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

Funkcja długości, napisana samodzielne

- ```
  def dlug(*values):
      i = 0
      for x in values:
      i += 1
      return i

  print(dlug(12, 4)) #2
  print(dlug(3, 4, 5, 7)) #4
  ```

- ```
  def dlug(*values):
      i = 0
      for _ in values:
      i += 1
      return i

  print(dlug(12, 4)) #2
  print(dlug(3, 4, 5, 7)) #4
  ```

Argumenty oprocz *args:

- ```
  def func(potega, *values):
      s = 0
      for i in values:
          s+=i**potega
      return s


  print(func(2, 3, 4, 6)) #61
  print(func(3, 4)) #64
  print(func(3)) #0
  ```
- ```
  def func(*values, potega):
      s = 0
      for i in values:
          s+=i**potega
      return s
  #print(func(2,3, 5))blad
  #print(func(potega = 5, 2,3))blad
  print(func(2,3,potega = 5)) #275
  ```

Argumenty domyślne z *args:

```
def func(potega = 2, *values):
    s = 0
    for i in values:
        s+=i**potega
    return s


print(func(2,3, 4, 6)) #61
print(func(3, 4)) #64
#print(func(3, 4, potega = 3)) blad
#print(func( potega = 3, 3, 4)) blad
print(func(3)) #0
```

Argumenty domyślne z *args:

```
def func(*values, potega = 2):
    s = 0
    for i in values:
        s+=i**potega
    return s


print(func(2,3, 4, 6)) #65
print(func(3, 4)) #25
print(func(3, 4, potega = 3)) #91
#print(func( potega = 3, 3, 4)) blad
print(func(3)) #9
```

Funkcje: dwie funkcje z rożną ilością argumentów nie są możliwe, ale!

```
def area(l, b):
  return l * b

def area(r):
  return 3.14 * r ** 2


area(3, 4)
area(7)
```

```
def area(*values):
  if len(values)==2:
      return values[0]*values[1]
  if len(values)==1:
      return 3.14 * values[0] ** 2

print(area(3, 4)) #12
print(area(7)) #153.86
```

Funkcje: dwie funkcje z rożną ilością argumentów nie są możliwe, ale jeszcze raz

```
def area(l= None, b= None, r = None):
    if r == None:
        return l*b
    else:
        return 3.14 * r ** 2


print(area(l=3, b=4))#12
print(area(r=7))#153.86
```

Funkcje rekurencyjne

Funkcje rekurencyjne są to funkcje, których charakterystyką jest to, iż przed zakończeniem bieżącego wywołania funkcji następuje kolejne wywołanie tej funkcji (czyli w ciele danej funkcji następuje wywołanie tejże funkcji).

Ich zawsze można i lepiej(!) zastąpić pętlą.

Silnia

```
def silnia(n):
    if n:
        return silnia(n-1)*n
    else:
        return 1

print(silnia(5)) #120
print(silnia(0)) #1
```

```
def silnia(n):
    s = 1
    for i in range(1,n+1):
        s*=i
    return s

print(silnia(5)) #120
print(silnia(0)) #1
```

Potęgowanie

```
def pot(n, p):
    if p ==0:
        return 1
    else:
        return n*pot(n,p-1)

print(pot(5,2))#25
print(pot(8,3))#512
```

```
def pot(n, p):
    prod = 1
    for i in range (1,p+1):
        prod *= n
    return prod

print(pot(5,2))#25
print(pot(8,3))#512
```

Potęgowanie binarne

```
def pot(n, p):
    if not p:
        return 1
    if p%2 == 0:
        temp = pot(n,p//2)
        return temp*temp
    else:
        temp = pot(n,p//2)
        return temp*temp*n

print(pot(2,10))#1024
```

Potęgowanie binarne: ilość operacje

```
global count
count = 0
def pot(n, p):
    global count
    if not p:
        return 1
    if p%2 == 0:
        temp = pot(n,p//2)
        count += 1
        return temp*temp
    else:
        temp = pot(n,p//2)
        count += 2
        return temp*temp*n

print(pot(2,10))
print(count)#6

count = 0
print(pot(2,100))
print(count)#10
```

Potęgowanie zwykle: ilość operacje

```
count = 0
def pot(n, p):
    global count
    if p ==0:
        return 1
    else:
        count += 1
        return n*pot(n,p-1)

print(pot(2,10))
print(count) #10

count = 0
print(pot(2,100))
print(count) #100
```

Potęgowanie binarne

Uwaga! Potęgowanie binarne można zrobić nie rekurencyjne, ale dlatego będziemy potrzebowali listy. Dlatego zrobimy to później.

Ciąg Fibonacciego

```
def Fib(n):
    if n < 2:
        return n
    else:
        return Fib(n-1) + Fib(n-2)

print(Fib(12)) #144
```

```
def Fib1(n):
    if (n == 0):
        return 0
    i0 = 0
    i1 = 1
    for j in range(1,n):
        i0, i1 = i1, i0+i1
    return i1

print(Fib1(12)) #144
```

Dlaczego rekurencja w ciągu Fibonacciego jest złym rozwiązaniem

```
count = 0
def Fib(n):
    global count
    count += 1
    if n < 2:
        return n
    else:
        return Fib(n-1) + Fib(n-2)

print(Fib(12))
print(count)#465

#print(Fib(30))
#print(count) #2692537
```

Dlaczego rekurencja w ciągu Fibonacciego jest złym rozwiązaniem

```
count = 0
def Fib1(n):
    global count
    if (n == 0):
        return 0
    i0 = 0
    i1 = 1
    for j in range(1,n):
        i0, i1 = i1, i0+i1
        count += 1
    return i1

print(Fib1(12))
print(count)#11

print(Fib1(30))
print(count) #29
```

Algorytm Euklidesa

Algorytm Euklidesa —- algorytm wyznaczania największego wspólnego dzielnika dwóch liczb. Został opisany przez greckiego matematyka, Euklidesa w jego dziele „Elementy", w księgach siódmej oraz dziesiątej.
Największym wspólnym dzielnikiem dwóch liczb jest największa z liczb, która dzieli obie te liczby bez reszty
Dla liczb całkowitychk, m oraz n załóżmy, że k jest wspólnym czynnikiem liczb a oraz b. Niech

$$a = bq + r.$$

Wtedy k jest wspólnym czynnikiem liczb b oraz r. I odwrotnie. Tzn,

$$NWD(a, b) = NWD(q, r)$$

Dzialamy, dopoki $r \neq 0$.

Algorytm Euklidesa

```
def NWD(a, b):
    if b:
        return NWD(b,a%b)
    return a

print(NWD(12,18)) #6
```

Na ćwiczeniach: napisać wersja nierekurencyjna

Funkcje wbudowane

Built-in Functions

A
abs()
aiter()
all()
anext()
any()
ascii()

B
bin()
bool()
breakpoint()
bytearray()
bytes()

C
callable()
chr()
classmethod()
compile()
complex()

D
delattr()
dict()
dir()
divmod()

E
enumerate()
eval()
exec()

F
filter()
float()
format()
frozenset()

G
getattr()
globals()

H
hasattr()
hash()
help()
hex()

I
id()
input()
int()
isinstance()
issubclass()
iter()

L
len()
list()
locals()

M
map()
max()
memoryview()
min()

N
next()

O
object()
oct()
open()
ord()

P
pow()
print()
property()

R
range()
repr()
reversed()
round()

S
set()
setattr()
slice()
sorted()
staticmethod()
str()
sum()
super()

T
tuple()
type()

V
vars()

Z
zip()

_
__import__()

Link do dokumentacji
`https://docs.python.org/3/library/functions.html`

Funkcje wbudowane

```
print (abs(-10))#10

def abs(x):
    return 5

print (abs(-10))#5
```

- `abs()` – wartosc bezwzgledna
- `min()` – minimum
- `max()` – maksymum
- `pow()` – potega

Slowo kluczowe pass

Instrukcja pass służy jako symbol zastępczy dla przyszłego kodu, zapobiegając błędom z pustych bloków kodu.
Jest zwykle używana, gdy kod jest zaplanowany, ale jeszcze nie został napisany.

```
def future_function():
    pass

# this will execute without any action or error
future_function()
print(future_function())#None
```

Funkcje: import math

Link do dokumentacji https://docs.python.org/3/library/math.html

- ```
  import math

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

  print(math.e)
  print(math.exp(1))
  print(math.ceil(math.e))
  print(math.floor(math.e))
  print(math.factorial(10))
  ```

Funkcje: import math

Link do dokumentacji `https://docs.python.org/3/library/math.html`

- ```
  import math

  print(math.gcd())
  print(math.gcd(20,16))
  print(math.gcd(20,15,35,25))
  ```
- ```
  import math

  #print(math.prod())
  #print(math.prod(0,2))
  print(math.prod((0,2)))
  print(math.prod((12,3,21)))
  print(math.prod((12,3,21), start=2))
  ```

Funkcje: potęga

```
import math

print(math.pow(2,100))#1.2676506002282294e+30
print(pow(2,100))#1267650600228229401496703205376
print(2**100)#1267650600228229401496703205376

#print(math.pow(-2,0.5))
print(pow((-2),0.5))#(8.659560562354934e-17+1.4142135623730951j)
print((-2)**0.5)#(8.659560562354934e-17+1.4142135623730951j)

print(math.exp(1e-5) - 1)  # gives result accurate to 11 places
print(math.expm1(1e-5) )
```

Oprócz tego `math.exp(x)` dokładniej niż `math.e ** x` oraz `pow(math.e, x)`.

Funkcje: reszta

```
import math

print(math.remainder(8,3))#-1.0
print(math.fmod(8,3))#2.0
print(8%3)#2

print(math.remainder(8.5,3))#-0.5
print(math.fmod(8.5,3))#2.5
print(8.5%3)#2.5

print(math.remainder(-1e-100,1e100))#-1e-100
print(math.fmod(-1e-100,1e100))#-1e-100
print(-1e-100 % 1e100)#1e+100
```

Slowo kluczowe pass

Instrukcja pass służy jako symbol zastępczy dla przyszłego kodu, zapobiegając błędom z pustych bloków kodu.
Jest zwykle używana, gdy kod jest zaplanowany, ale jeszcze nie został napisany.

```
def future_function():
    pass

# this will execute without any action or error
future_function()
print(future_function())#None
```

Slowo kluczowe assert

assert – pozwala użyć asercji, aby móc skontrolować działanie programu. Jeśli warunek asercji nie zostanie spełniony kompilator zgłosi błąd AssertionError.

```
x = "hello"

#if condition returns True, then nothing happens:
assert x == "hello"

#if condition returns False, AssertionError is raised:
assert x == "goodbye"
```

Możemy użyć assert, żeby sprawdzić działanie funkcje:

```
assert abs(-10) == 10
assert abs(10) == 10
assert abs(0) == 0
```

Slowo kluczowe assert

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

def main():
    assert suma(3,5)==8
    assert suma(-2,1)==-1
    assert suma('a', 'b') == 'ab'
    #assert suma(0.2,0.3) == 0.5
    #assert suma (249.99, 22.41) == 272.40
    #print(suma (249.99, 22.41))

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