06_Funkce

C2184 Úvod do programování v Pythonu

6. Funkce

Funkce

- Objekt, který lze volat (pomocí závorek za jménem funkce)
- *Function*, *callable*
- Funkce při volání
  1. Něco vezme (**argumenty**)
  2. Něco udělá
  3. Něco vrátí (**návratovou hodnotu**)
- Příklad: funkce `abs`
  1. Vezme 1 argument: číslo x
  2. Spočítá absolutní hodnotu $|x|$
  3. Vrátí návratovou hodnotu: $|x|$

[1]:
```
y = abs(-5)
```

[2]:
```
y
```

[2]: 5

- Příklad: funkce `print`
  1. Vezme libovolný počet argumentů: libovolných objektů
  2. Převede všechny argumenty na řetězce a vypíše je na výstup
  3. Vrátí návratovou hodnotu: `None`

[3]:
```
y = print('ahoj', 5, True)
```

```
ahoj 5 True
```

[4]:
```
y
```

- Příklad: funkce `input`
    1. Vezme 0 argumentů
    2. Počká na vstup od uživatele
    3. Vrátí návratovou hodnotu: řetězec zadaný uživatelem

```
[5]: y = input()
```

```
[6]: y
```

```
[6]: 'Hello'
```

Argumenty

- **Poziční** (*positional arguments, args*)
- **Pojmenované** (*keyword arguments, kwargs*)

Příklad:

```
[7]: print(1, 2, 'A', sep='-', end=';\n')
```

```
1-2-A;
```

- 3 poziční argumenty: `1`, `2`, `'A'`
- 2 pojmenované argumenty: `'-'`, `';\n'`
- Pojmenované argumenty lze přehazovat

```
[8]: print(1, 2, 'A', sep='-', end=';\n')
```

```
1-2-A;
```

```
[9]: print(1, 2, 'A', end=';\n', sep='-')
```

```
1-2-A;
```

- Ale vždy se uvádějí nejdřív poziční, pak pojmenované

```
[10]: print(1, 2, sep='-', end=';\n', 'A')
```

```
  File "<ipython-input-10-75dd255cfd07>", line 1
    print(1, 2, sep='-', end=';\n', 'A')
                                   ^
SyntaxError: positional argument follows keyword argument
```

Metody

- Funkce, které jsou součástí objektu
- Voláme je pomocí tečky
- Objekt, ke kterému patří (`self`), je jakoby argumentem

[11]:
```
'ukazatel'.count('a')
```

[11]: 2

Můžeme si vytvořit vlastní funkce

- **Proč?**
  - Nejakou operaci provádíme často a nechceme psát vždy to stejné (*DRY – Don't Repeat Yourself*)
    –> vytvoříme na to funkci
  - Máme dlouhý program a chceme ho zpřehlednit (*SoC – Separation of Concerns*)
    –> rozdělíme ho na několik snadno pochopitelných funkcí
- **Jak?**
  - Pomocí klíčového slova `def`

[12]:
```
import math

r1 = 1.0
V1 = 4/3 * math.pi * r1**3
print(f'Koule o poloměru {r1:.2f} má objem {V1:.2f}.')

r2 = 5.0
V2 = 4/3 * math.pi * r2**3
print(f'Koule o poloměru {r2:.2f} má objem {V2:.2f}.')

r3 = 10.0
V3 = 4/3 * math.pi * r3**3
print(f'Koule o poloměru {r3:.2f} má objem {V3:.2f}.')
```

```
Koule o poloměru 1.00 má objem 4.19.
Koule o poloměru 5.00 má objem 523.60.
Koule o poloměru 10.00 má objem 4188.79.
```

[13]:
```
def print_sphere_volume(r):
    V = 4/3 * math.pi * r**3
    print(f'Koule o poloměru {r:.2f} má objem {V:.2f}.')

print_sphere_volume(1.0)
print_sphere_volume(5.0)
print_sphere_volume(10.0)
```

```
Koule o poloměru 1.00 má objem 4.19.
Koule o poloměru 5.00 má objem 523.60.
Koule o poloměru 10.00 má objem 4188.79.
```

Definice (vytvoření) funkce

- Pomocí klíčového slova `def`

  ```
  def identifier(parameters...):

      body...
  ```

- Funkce má svůj název (*indentifier*), parametry (*parameters*) a tělo (*body*)
- Funkce musí být nejdřív definována, až pak ji můžeme zavolat
- Tělo funkce se nevykoná, dokud funkci nezavoláme

[14]:
```
def print_sphere_volume(r):
    V = 4/3 * math.pi * r**3
    print(f'Koule o poloměru {r:.2f} má objem {V:.2f}.')
```

[15]:
```
print_sphere_volume
```

[15]:
```
<function __main__.print_sphere_volume(r)>
```

[16]:
```
print_sphere_volume(1.0)
```

```
Koule o poloměru 1.00 má objem 4.19.
```

Volání funkce

1. Hodnoty *argumentů* se dosadí do *parametrů* v definici funkce
2. Provede se tělo funkce
3. Vrátí se hodnota uvedena za klíčovým slovem `return`

[18]:
```
def square_area(side):
    print('Počítám obsah čtverce...')
    area = side**2
    return area
```

[19]:
```
S = square_area(5)
```

```
Počítám obsah čtverce…
```

[20]:
```
S
```

[20]:
```
25
```

Návratová hodnota funkce (*return value*)

- Hodnota, která je výsledkem volání funkce
- Pomocí klíčového slova `return` v těle funkce
- Jakmile se provede `return`, funkce skončí a zbývající část těla se ignoruje (podobné `break`)!

[21]:
```
def square_area(side):
    print('Počítám obsah čtverce...')
    area = side**2
    return area
    print('**********')
```

[22]:
```
S = square_area(5)
```

```
Počítám obsah čtverce…
```

[23]:
```
S
```

[23]:
```
25
```

Defaultní návratová hodnota

- Provede-li se celé tělo funkce bez nalezení `return`, funkce vrátí `None`
- Pouhé `return` taky vrátí `None`

[24]:
```
def greet(name):
    print(f'Hello {name}!')
```

[25]:
```
result = greet('Bob')
```

```
Hello Bob!
```

[26]:
```
print(result)
```

```
None
```

[27]:
```
def greet(name):
    print(f'Hello {name}!')
    return

result = greet('Alice')
print(result)
```

```
Hello Alice!
None
```

Parametry a argumenty funkce

- Při volání funkce se argumenty dosazují do parametrů funkce
  - Poziční argumenty po pořadí
  - Pojmenované argumenty podle názvu

[28]:
```
def cylinder_volume(radius, height):
    volume = math.pi * radius**2 * height
    return volume
```

- Poziční argumenty:

[29]:
```
cylinder_volume(1, 5)
```

[29]:
```
15.707963267948966
```

- Pojmenované argumenty:

[30]:
```
cylinder_volume(radius=1, height=5)
```

[30]:
```
15.707963267948966
```

[31]:
```
cylinder_volume(height=5, radius=1)
```

[31]:
```
15.707963267948966
```

- Počet argumentů musí sedět

[32]:
```
cylinder_volume(1)
```

```
     ␣
↪---------------------------------------------------------------------------

     TypeError                                 Traceback (most␣
↪recent call last)

     <ipython-input-32-8d9a8bfffc5b> in <module>
 ----> 1 cylinder_volume(1)


     TypeError: cylinder_volume() missing 1 required positional␣
↪argument: 'height'
```

[33]:
```
cylinder_volume(1, 5, 8)
```

```
        TypeError                                 Traceback (most
recent call last)

        <ipython-input-33-bbf7881e0941> in <module>
    ----> 1 cylinder_volume(1, 5, 8)


        TypeError: cylinder_volume() takes 2 positional arguments but
3 were given
```

Defaultní hodnoty parametrů

- Můžeme nastavit v definici funkce pomocí =
- Parametry s defaultní hodnotou musí být na konci výčtu parametrů

[34]:
```
def greet(name, repeat=1):
    for i in range(repeat):
        print(f'Hello {name}!')
```

[35]:
```
greet('Bob')
```

```
Hello Bob!
```

[36]:
```
greet('Bob', repeat=3)
```

```
Hello Bob!
Hello Bob!
Hello Bob!
```

Globální a lokální proměnné

- Globální proměnné (*globals*) – zadefinované mimo funkce
- Lokální proměnné (*locals*) – zadefinované v těle funkce
- Lokální proměnné a parametry existují pouze v rámci konkrétního volání funkce, z vnějšku jsou nedostupné

[37]:
```
def square_area(side):
    area = side**2
    return area

square_area(5)
```

[37]: 25

[38]:
```
area
```

```
↪---------------------------------------------------------------------------

NameError                                 Traceback (most
↪recent call last)

<ipython-input-38-cb240d40b889> in <module>
----> 1 area


NameError: name 'area' is not defined
```

[39]:
```
side
```

```
↪---------------------------------------------------------------------------

NameError                                 Traceback (most
↪recent call last)

<ipython-input-39-c4e50750d870> in <module>
----> 1 side


NameError: name 'side' is not defined
```

- Globální proměnné jsou viditelné zevnitř funkce, ale nelze do nich zapisovat

[40]:
```
the_name = 'Bob'

def print_the_name():
    print('The name is:', the_name)

print_the_name()
```

```
The name is: Bob
```

[41]:
```
the_name = 'Bob'

def change_the_name(new_name):
```

```
    the_name = new_name  # toto je lokální proměnná, která zakrývá␣
↪globální proměnnou the_name

change_the_name('Alice')
print(the_name)
```

```
Bob
```

[42]:
```
the_name = 'Bob'

def print_and_change_the_name(new_name):
    print('The name is:', the_name)  # globální proměnná the_name?
    the_name = new_name  # ale kdepak, the_name je lokální proměnná
    # skončí chybou, protože lokální proměnnou nelze vypsat před její␣
↪nastavením!

print_and_change_the_name('Alice')
print(the_name)
```

```
        ␣
↪---------------------------------------------------------------------------

        UnboundLocalError                         Traceback (most␣
↪recent call last)

        <ipython-input-42-e63e97afe197> in <module>
          6     # skončí chybou, protože lokální proměnnou nelze␣
↪vypsat před její nastavením!
          7
    ----> 8 print_and_change_the_name('Alice')
          9 print(the_name)


        <ipython-input-42-e63e97afe197> in␣
↪print_and_change_the_name(new_name)
          2
          3 def print_and_change_the_name(new_name):
    ----> 4     print('The name is:', the_name)  # globální proměnná␣
↪the_name?
          5     the_name = new_name  # ale kdepak, the_name je lokální␣
↪proměnná
          6     # skončí chybou, protože lokální proměnnou nelze␣
↪vypsat před její nastavením!
```

```
UnboundLocalError: local variable 'the_name' referenced before␣
↪assignment
```

- Klíčové slovo `global` umožňuje zápis do globální proměnné

[43]:
```
the_name = 'Bob'

def change_the_name(new_name):
    global the_name
    the_name = new_name  # toto je globální proměnná the_name

change_the_name('Alice')
print(the_name)
```

```
Alice
```

- Použití `global` se nedoporučuje!
  - Více funkcí může měnit proměnné zadefinované na různých místech
  - Špatná čitelnost kódu
  - Náchylnost na chyby

[44]:
```
the_name = 'Bob'

def print_the_name():
    print('The name is:', the_name)

the_name = 'Alice'
print_the_name()
```

```
The name is: Alice
```

[45]:
```
def calculate_rectangle_area(a, b):
    global rectangle_area
    rectangle_area = a*b

def calculate_triangle_area(a, b):
    global triangle_area
    calculate_rectangle_area(a, b)
    triangle_area = rectangle_area / 2

calculate_rectangle_area(2, 3)
calculate_triangle_area(10, 20)
print('Rectangle area:', rectangle_area)
print('Triangle area:', triangle_area)

# FUJ!
```

```
Rectangle area: 200
Triangle area: 100.0
```

- Místo globálních proměnných používat:
  - parametry (když chci dostat data do funkce)
  - návratovou hodnotu (když chci dostat data z funkce)
- Použití globálních konstant ve funkci je OK

[46]:
```
GREETING = 'Hello'

def greet(name):
    print(f'{GREETING}, {name}!')

greet('Cyril')
```

```
Hello, Cyril!
```

Dokumentace

- Aby bylo jasné, co funkce dělá, je zvykem doplnit *docstring* na začátek funkce.
- Nepovinné, ale užitečné, zejména u větších projektů a při spolupráci více lidí.

[47]:
```
def cylinder_volume(radius, height):
    '''Return the volume of a cylinder with specified radius and␣
↪height.'''
    volume = math.pi * radius**2 * height
    return volume
```

Typové anotace

- Můžeme označit typy parametrů a návratové hodnoty.
- Nepovinné, ale užitečné, zejména u větších projektů a při spolupráci více lidí.
- VSCode používá docstrings i typové anotace při napovídání

[48]:
```
def cylinder_volume(radius: float, height: float) -> float:
    '''Return the volume of a cylinder with specified radius and␣
↪height.'''
    volume = math.pi * radius**2 * height
    return volume
```

[49]:
```
def greet(name: str, repeat: int = 1) -> None:
    '''Print repeat greetings to a person called name.'''
    for i in range(repeat):
        print(f'Hello {name}!')
```

- Interpret nekontroluje typy – funkce poběží i když argumenty budou jiných typů.
- Kontrolu typů lze provést pomocí modulu `mypy`.

[50]:
```
greet('Alice')
```

```
Hello Alice!
```

[51]:
```
greet([1, 2, 3])
```

```
Hello [1, 2, 3]!
```

- Pomocí modulu `typing` můžeme přesněji specifikovat typy:
  - `List[A]` – seznam prvků typu `A`
  - `Set[A]` – množina prvků typu `A`
  - `Tuple[A, B, C]` – n-tice s prvním prvkem typu `A`, druhým typu `B`, třetím typu `C`
  - `Dict[A, B]` – slovník s klíči typu `A` a hodnotami typu `B`
  - `Iterable[A]` – iterovatelný objekt s prvky typu `A`
  - `Union[A, B]` – objekt typu `A` nebo typu `B`
  - `Optional[A]` – objekt typu `A` nebo `None`
  - `Any` – sedí na libovolný typ
  - …
- https://docs.python.org/3/library/typing.html

[52]:
```
def min_avg_max(numbers: list) -> tuple:
    '''Return the minimum, average, and maximum of the numbers.'''
    minimum = min(numbers)
    average = sum(numbers) / len(numbers)
    maximum = max(numbers)
    return (minimum, average, maximum)
```

[53]:
```
from typing import Tuple, List, Dict

def min_avg_max(numbers: List[int]) -> Tuple[int, float, int]:
    '''Return the minimum, average, and maximum of the numbers.'''
    minimum = min(numbers)
    average = sum(numbers) / len(numbers)
    maximum = max(numbers)
    return (minimum, average, maximum)

min_avg_max([1, 8, 5, 3])
```

[53]:
```
(1, 4.25, 8)
```

[54]:
```
def word_indices(words: List[str]) -> Dict[str, int]:
    '''Return dictionary with index of each word from words.'''
    return {word: i for i, word in enumerate(words)}

word_indices('they have been contaminated by pollution'.split())
```

[54]:
```
{'they': 0, 'have': 1, 'been': 2, 'contaminated': 3, 'by': 4,␣
 ↪'pollution': 5}
```

- Všechny typy jsou podtypem typu `object`
- Tj. hodnota libovolného typu je zároveň typu `object`

[55]:
```
type(5)
```

[55]:
```
int
```

[56]:
```
isinstance(5, int)
```

[56]:
```
True
```

[57]:
```
isinstance(5, object)
```

[57]:
```
True
```

[58]:
```
from typing import List

def last(elements: List[object]) -> object:
    '''Return the last element of a list.'''
    return elements[-1]
```

Rekurze

- Když funkce volá sama sebe.

[59]:
```
def factorial(n: int) -> int:
    '''Calculate factorial of number n.'''
    if n == 1:
        return 1
    else:
        return n * factorial(n - 1)
```

[60]:
```
factorial(5)
```

[60]:
```
120
```

- Pozor, hrozí zacyklení (např. volání `factorial(0)`).

- Nepřímá rekurze: např. funkce `a` volá funkci `b`, `b` volá `a`.
- Příklad rekurze: prohledávání vnořených seznamů:

```
[61]: 5 in [1, 2, [3, [4], [5, 6], 7], 8, 9]
```

```
[61]: False
```

```
[62]: def deep_search(deep_list: list, item: object) -> bool:
          '''Decide if item is present in deep_list or in any of its␣
      ↪elements, recursively.'''
          for x in deep_list:
              if x == item:
                  return True
              elif isinstance(x, list) and deep_search(x, item):
                  return True
          return False
```

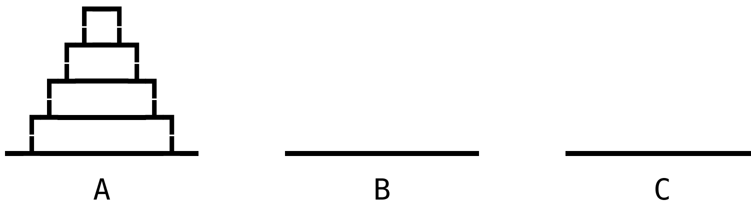
```
[63]: deep_search([1, 2, [3, [4], [5, 6], 7], 8, 9], 5)
```

```
[63]: True
```

```
[64]: deep_search([1, 2, [3, [4], [5, 6], 7], 8, 9], 0)
```

```
[64]: False
```

- Příklad rekurze: Hanojské věže



- Lze překládat vždy pouze jeden disk.
- Větší disk nelze položit na menší.
- Úkol: přesunout celou věž na políčko C.

Kolik tahů potřebujeme na přesunutí n disků?

- Příklad rekurze:
  - Platy zaměstnanců máme uloženy ve slovníkové struktuře rozdělené podle hierarchie univerzity (fakulty, ústavy apod.).
  - Chceme spočítat součet platů všech zaměstnanců.

```
[65]: salaries = {
          'PřF': {
```

```
        'Biologie': {'Alice': 30, 'Bob': 30},
        'Chemie': {
            'Organika': {'Cyril': 35},
            'Anorganika': {'Dana': 28}
        },
        'Fyzika': {'Emil': 27}
    },
    'LF': {'Filip': 34, 'Gertruda': 33},
    'FSpS': {'Hana': 30}
}
```

[66]:
```
def recursive_sum(organization):
    if isinstance(organization, dict):
        return sum(recursive_sum(part) for part in organization.
 ↪values())
    else:
        return organization
```

[67]:
```
recursive_sum(salaries)
```

[67]: 247

Anonymní funkce lambda

- Vytvoření funkce beze jména
- Příklad: chceme seřadit studenty podle příjmení – použijeme `sorted` s parametrem `key`

[68]:
```
students = [('Alice', 'Nováková'), ('Cyril', 'Veselý'), ('Bob',␣
 ↪'Marley')]
sorted(students)  # Řadí podle křestního jména
```

[68]: [('Alice', 'Nováková'), ('Bob', 'Marley'), ('Cyril', 'Veselý')]

- S pojmenovanou funkcí:

[69]:
```
def surname(person):
    return person[1]

sorted(students, key=surname)  # Řadí podle příjmení
```

[69]: [('Bob', 'Marley'), ('Alice', 'Nováková'), ('Cyril', 'Veselý')]

- S lambda funkcí:

[70]:
```
sorted(students, key = lambda person: person[1])  # Řadí podle␣
↪příjmení
```

[70]:
```
[('Bob', 'Marley'), ('Alice', 'Nováková'), ('Cyril', 'Veselý')]
```

Rozbalování argumentů (*unpacking*)

- Poziční argumenty můžeme rozbalit pomocí * (z iterovatelného objektu)
- Pojmenované argumenty můžeme rozbalit pomocí ** (ze slovníku)

[71]:
```
numbers = [3, 2, 1]
formatting = {'sep': ', ', 'end': '.'}
print(*numbers, **formatting)
```

```
3, 2, 1.
```

[72]:
```
print(numbers)
```

```
[3, 2, 1]
```

[73]:
```
print(*numbers)  # Ekvivalentní print(3, 2, 1)
```

```
3 2 1
```

[74]:
```
print(*numbers, **formatting)  # Ekvivalentní print(3, 2, 1, sep=',␣
↪', end='.')
```

```
3, 2, 1.
```

Rozšiřující učivo

Nenasytné parametry

- Pokud použijete * před názvem posledního (předposledního) parametru, tento parametr bude obsahovat všechny nadbytečné poziční argumenty
- Pokud použijete ** před názvem posledního parametru, tento parametr bude obsahovat všechny nadbytečné klíčové argumenty

[75]:
```
def foo(a, b, *args, **kwargs):
    print(a)
    print(b)
    print(args)
    print(kwargs)
```

[76]:
```
foo(1, 2, 3, 4, 5, 6, x=100, y=200)
```

```
1
2
(3, 4, 5, 6)
{'x': 100, 'y': 200}
```

Generátorové funkce (*generator functions*)

- Funkce, kterých návratovou hodnotou je *generátor*
  (*generátor* je typ *iterátoru*)
- Generátor generuje hodnoty až když jsou potřeba (ne už při zavolání funkce)
  - Jednu hodnotu vyžádáme pomocí funkce `next`
  - Více hodnot pomocí `for` cyklu
- Generované hodnoty se v těle funkce uvádějí slovem `yield` (místo `return`)

[78]:
```
from typing import Iterator

# Generátorová funkce
def echo(word: str) -> Iterator[str]:
    while len(word) > 0:
        yield word
        word = word[1:]
```

[79]:
```
generator = echo('Hello')  # Vytváříme generátor
generator
```

[79]:
```
<generator object echo at 0x7fbb03f0ad60>
```

[80]:
```
next(generator)  # Vygenerujeme 1. hodnotu
```

[80]:
```
'Hello'
```

[81]:
```
for s in generator:  # Vygenerujeme zbylé hodnoty
    print(s)
```

```
ello
llo
lo
o
```

[82]:
```
for s in generator:  # Generátor se už vyčerpal, nevypíše se nic
    print(s)
```

[83]:
```
next(generator)  # Generátor se už vyčerpal, vyhodí se chyba typu␣
 ↪StopIteration
```

```
    StopIteration                             Traceback (most␣
↪recent call last)

    <ipython-input-83-2665f78e1bf9> in <module>
  ----> 1 next(generator)  # Generátor se už vyčerpal, vyhodí se␣
↪chyba typu StopIteration


    StopIteration:
```

[84]:
```
generator = echo('ahoj')  # Nový generátor, opět můžeme generovat
next(generator)
```

[84]:
```
'ahoj'
```

- Generátor může generovat i nekonečnou posloupnost (není to problém, protože hodnoty se generují až když je potřeba)

[85]:
```
def odd_numbers() -> Iterator[int]:
    i = 2
    while True:
        yield i
        i += 2
```

[86]:
```
generator = odd_numbers()
generator
```

[86]:
```
<generator object odd_numbers at 0x7fbb03f6c200>
```

[87]:
```
for x in generator:
    print(x)
    if x >= 10:
        break
```

```
2
4
6
8
10
```

[88]:
```
# Generujeme dál
for x in generator:
```

```
    print(x)
    if x >= 20:
        break
```

```
12
14
16
18
20
```

- Funkce `iter` udělá z jakéhokoli iterovatelného objektu iterátor

[89]:
```
iterator = iter('ahoj')
iterator
```

[89]:
```
<str_iterator at 0x7fbb03f739d0>
```

[90]:
```
next(iterator)
```

[90]:
```
'a'
```

[91]:
```
for x in iterator:
    print(x)
```

```
h
o
j
```

[92]:
```
next(iterator)
```

```
     ␣
↪---------------------------------------------------------------------------

    StopIteration                             Traceback (most␣
↪recent call last)

    <ipython-input-92-4ce711c44abc> in <module>
  ----> 1 next(iterator)


    StopIteration:
```