C2184 Úvod do programování v Pythonu

6. Funkce

Funkce

- Objekt, který lze volat (pomocí závorek za jménem funkce)
- 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*|

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

In [2]: y

Out[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`

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

```
ahoj 5 True
```

In [4]:
```
y
```

- Příklad: funkce `input`

    1. Vezme 0 argumentů
    2. Počká na vstup od uživatele
    3. Vrátí návratovou hodnotu: řetězec zadaný uživatelem

In [7]:
```
y = input()
```

```
Něco píšu...
```

In [8]:
```
y
```

Out[8]:
```
'Něco píšu...'
```

Argumenty

- **Poziční** (*positional*)
- **Pojmenované** (*keyword*)

Příklad:

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

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

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

- Pojmenované argumenty lze přehazovat

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

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

In [11]:
```
print(1, 2, 'A', end=';\n', sep='-')
```

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

- Ale vždy se uvádějí nejdřív poziční, pak pojmenované

In [12]:
```
print(1, 2, sep='-', end=';\n', 'A')
```

```
  File "<ipython-input-12-75dd255cfd07>", line 1
    print(1, 2, sep='-', end=';\n', 'A')
                                       ^
SyntaxError: positional argument follows keyword argument
```

Metody

- Normální funkce, pouze je jinak voláme (pomocí tečky)
- Vážou se ke konkrétnímu objektu, který je jakoby parametrem

In [13]:
```
'ukazatel'.count('a')
```

Out[13]:
```
2
```

Můžeme si vytvořit vlastní funkce

- **Proč?**
  - Nejakou operaci provádíme často a nechceme psát vždy to stejné

    → vytvoříme na to funkci
  - Máme dlouhý program a chceme ho zpřehlednit

    → rozdělíme ho na několik funkcí
- **Jak?**
  - Pomocí klíčového slova `def` ...

In [15]:
```
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.
```

In [16]:
```
def vypis_objem_koule(r):
    V = 4/3 * math.pi * r**3
    print(f'Koule o poloměru {r:.2f} má objem {V:.2f}.')

vypis_objem_koule(1.0)
vypis_objem_koule(5.0)
vypis_objem_koule(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 identifikator(parametry):
    telo_funkce
```

- Funkce má svůj *identifikátor* (název)
- Funkce musí být nejdřív definována, až pak ji můžeme zavolat

- Funkce se nevykoná, dokud ji nezavoláme

In [17]:
```
def vypis_objem_koule(r):
    V = 4/3 * math.pi * r**3
    print(f'Koule o poloměru {r:.2f} má objem {V:.2f}.')
```

In [18]:
```
vypis_objem_koule
```

Out[18]:
```
<function __main__.vypis_objem_koule>
```

In [19]:
```
vypis_objem_koule(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`

In [20]:
```
def obsah_ctverce(a):
    print('Počítám obsah čtverce...')
    S = a**2
    return S
```

In [21]:
```
S1 = obsah_ctverce(5)
```

```
Počítám obsah čtverce...
```

In [22]:
```
S1
```

Out[22]:
```
25
```

- Parametry a proměnné vytvořené uvnitř funkce existují pouze uvnitř funkce

In [23]:
```
a
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-23-3f786850e387> in <module>()
----> 1 a

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

In [24]:
```
S
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-24-df23348f6d3c> in <module>()
----> 1 S

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

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`)!

In [25]:
```
def obsah_ctverce(a):
    print('Počítám obsah čtverce...')
    S = a**2
    return S
    print('**********')
```

In [26]:
```
S1 = obsah_ctverce(5.0)
```

```
Počítám obsah čtverce...
```

In [27]:
```
S1
```

Out[27]:
```
25.0
```

Defaultní návratová funkce

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

In [28]:
```
def pozdrav(jmeno):
    print(f'Hello {jmeno}!')
```

In [29]:
```
a = pozdrav('Bob')
```

```
Hello Bob!
```

In [30]:
```
print(a)
```

```
None
```

In [31]:
```
def pozdrav(jmeno):
    print(f'Hello {jmeno}!')
    return

a = pozdrav('Alice')
print(a)
```

```
Hello Alice!
None
```

Parametry a argumenty funkce

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

In [32]:
```
def objem_valce(polomer, vyska):
    objem = math.pi * polomer**2 * vyska
    return objem
```

- Poziční argumenty:

In [33]:
```
objem_valce(1.0, 5.0)
```

Out[33]:
```
15.707963267948966
```

- Pojmenované argumenty:

In [34]:
```
objem_valce(polomer=1.0, vyska=5.0)
```

Out[34]:
```
15.707963267948966
```

In [35]:
```
objem_valce(vyska=5.0, polomer=1.0)
```

Out[35]:
```
15.707963267948966
```

- Počet argumentů musí sedět

In [36]:
```
objem_valce(1)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-36-d9ccf585cdeb> in <module>()
----> 1 objem_valce(1)

TypeError: objem_valce() missing 1 required positional argument: 'vyska'
```

In [37]:
```
objem_valce(1, 5)
```

Out[37]:
```
15.707963267948966
```

In [38]:
```
objem_valce(1, 5, 1)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-38-1232cba1b0e1> in <module>()
----> 1 objem_valce(1, 5, 1)

TypeError: objem_valce() 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

In [39]:
```
def pozdrav(jmeno, opakovani=1):
    for i in range(opakovani):
        print(f'Hello {jmeno}!')
```

In [40]:
```
pozdrav('Bob')
```

```
Hello Bob!
```

In [41]:
```
pozdrav('Bob', opakovani=3)
```

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

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.

In [42]:
```
def objem_valce(r, h):
    """Spočítej a vrať objem válce o polomeru r a výšce h."""
    V = math.pi * r**2 * h
    return V
```

Typové anotace

- Můžeme označit typy parametrů a návratové hodnoty.
- Nepovinné, ale užitečné, zejména u větších projektů a při spolupráci více lidí.
- Funkce poběží i když argumenty budou špatných typů – kontrolu lze provést pomocí modulu `mypy`.

In [43]:
```
def objem_valce(r: float, h: float) -> float:
    """Spočítej a vrať objem válce o polomeru r a výšce h."""
    V = math.pi * r**2 * h
    return V
```

In [44]:
```
def pozdrav(jmeno: str, opakovani: int = 1) -> None:
    """Vypiš pozdrav osobě jmeno opakovani-krát."""
    for i in range(opakovani):
        print(f'Hello {jmeno}!')
```

- Pomocí modulu `typing` můžeme přesněji specifikovat typy:

In [45]:
```
def min_max_pocet(cisla: list) -> tuple:
    """Vrať nejmenší, největší číslo a počet čísel."""
    return (min(cisla), max(cisla), len(cisla))
```

In [46]:
```
from typing import Tuple, List
def min_max_pocet(cisla: List[float]) -> Tuple[float, float, int]:
    """Vrať nejmenší, největší číslo a počet čísel."""
    return (min(cisla), max(cisla), len(cisla))
```

- VSCode používá docstrings i typové anotace při napovídání

Rekurze

- Když funkce volá sama sebe.

In [47]:
```
def faktorial(n: int) -> int:
    """Spočítej faktoriál čísla n."""
    if n == 1:
        return 1
    else:
        return n * faktorial(n - 1)
```

In [48]:
```
faktorial(5)
```

Out[48]:
```
120
```

- Pozor, hrozí zacyklení (např. volání `faktorial(0)`).
- Nepřímá rekurze: např. funkce a volá funkci b, b volá a.

Rekurze – příklad

Platy zaměstnanců máme uloženy ve slovníkové struktuře rozdělené podle hierarchie univerzity (fakulty, ústavy apod.).

Chceme spočítat součet platů všech zaměstnanců.

In [49]:
```
platy = {
    'PřF': {
        'Biologie': {'Alice': 30, 'Bob': 30},
        'Chemie': {
            'Organika': {'Cyril': 35},
            'Anorganika': {'Dana': 28}
        },
        'Fyzika': {'Emil': 27}
    },
    'LF': {'Filip': 34, 'Gertruda': 33},
    'FSpS': {'Hana': 30}
}
```

In [50]:
```
def rekurzivni_soucet(celek):
    if isinstance(celek, dict):  # Testuje jestli cast je typu dict.
        return sum(rekurzivni_soucet(cast) for cast in celek.values())
    else:
        return celek
```

In [51]:
```
rekurzivni_soucet(platy)
```

Out[51]:
```
247
```

Rozbalování argumentů (*unpacking*)

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

In [52]:
```
cisla = [3, 2, 1]
formatovani = {'sep': ', ', 'end': '.'}
print(*cisla, **formatovani)
```

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

In [53]:
```
print(cisla)
```

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

In [54]:
```
print(*cisla)  # Ekvivalentní print(3, 2, 1)
```

```
3 2 1
```

In [55]:
```
print(*cisla, **formatovani)  # 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

In [56]:
```
def foo(a, b, *args, **kwargs):
    print(a)
    print(b)
    print(args)
    print(kwargs)
```

In [57]:
```
foo(1, 2, 3, 4, 5, 6, x=100, y=200)
```

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

Anonymní funkce lambda

- Vytvoření funkce beze jména

In [58]:
```
studenti = [('Alice', 'Nováková'), ('Cyril', 'Veselý'), ('Bob', 'Marley')]
sorted(studenti)  # Řadí podle křestního jména
```

Out[58]:
```
[('Alice', 'Nováková'), ('Bob', 'Marley'), ('Cyril', 'Veselý')]
```

In [59]:
```
def prijmeni(osoba):
    return osoba[1]
sorted(studenti, key=prijmeni)  # Řadí podle příjmení
```

Out[59]:
```
[('Bob', 'Marley'), ('Alice', 'Nováková'), ('Cyril', 'Veselý')]
```

- Ekvivalent bez pojmenování funkce:

In [60]:
```
sorted(studenti, key = lambda osoba: osoba[1])  # Řadí podle příjmení
```

Out[60]:
```
[('Bob', 'Marley'), ('Alice', 'Nováková'), ('Cyril', 'Veselý')]
```

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`)

In [61]:
```
# Generátorová funkce
def ozvena(slovo):
    while len(slovo) > 0:
        yield slovo
        slovo = slovo[1:]
```

In [62]:
```
generator = ozvena('ahoj')  # Vytváříme generátor
generator
```

Out[62]:
```
<generator object ozvena at 0x7fd18c493830>
```

In [63]:
```
next(generator)  # Vygenerujeme 1. hodnotu
```

Out[63]:
```
'ahoj'
```

In [64]:
```
for s in generator:  # Vygenerujeme zbylé hodnoty
    print(s)
```

```
hoj
oj
j
```

In [65]:
```
for s in generator:  # Generátor se už vyčerpal, nevypíše se nic
    print(s)
```

In [66]:
```
next(generator)  # Generátor se už vyčerpal, vyhodí se chyba typu StopIteration
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-66-2665f78e1bf9> in <module>()
----> 1 next(generator)  # Generátor se už vyčerpal, vyhodí se chyba typu Stop
Iteration

StopIteration:
```

In [67]:
```
generator = ozvena('ahoj')  # Nový generátor, opět můžeme generovat
next(generator)
```

Out[67]:
```
'ahoj'
```

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

In [68]:
```
def suda_cisla():
    i = 2
    while True:
        yield i
        i += 2
```

In [69]:
```
generator = suda_cisla()
generator
```

Out[69]:
```
<generator object suda_cisla at 0x7fd18c3c56d0>
```

In [70]:
```
for x in generator:
    print(x)
    if x >= 10:
        break
```

```
2
4
6
8
10
```

In [71]:
```
# 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

In [72]:
```
iterator = iter('ahoj')
iterator
```

Out[72]:
```
<str_iterator at 0x7fd18c3bd828>
```

In [73]:
```
next(iterator)
```

Out[73]:
```
'a'
```

In [74]:
```
next(iterator)
```

Out[74]:
```
'h'
```

In [75]:
```
for x in iterator:
    print(x)
```

```
o
j
```

In [76]:
```
next(iterator)
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-76-4ce711c44abc> in <module>()
----> 1 next(iterator)

StopIteration:
```