

Programování 1 pro matematiky

7. cvičení, 10-11-2021

tags: Programování 1 2021, středa

Obsah:

- 0. Farní oznamy
- 1. Pythonovské funkce

Farní oznamy

1. **Materiály k přednáškám** najdete v GitHub repozitáři <https://github.com/PKvasnick/Programovani-1>. Najdete tam také kód ke cvičením a pdf soubory textů cvičením.
2. **Domácí úkoly** Dostali jste zatím 15 úkolů k prvním pěti cvičením.
 - Nominální počet bodů (bez "bonusových" úloh) - 100%: 115
 - Minimální počet bodů: 80
 - V pásmu ohrožení se nachází 5 z vás
 - Na poslední úkoly zatím přišlo docela málo řešení.

Kde se nacházíme Dnes nebude opakování, opakovali jsme celé minulé cvičení, a rovnou jdeme na Pythonovské funkce.

Funkce

Pokud chceme izolovat určitou část kódu, například proto, že dělá dobře definovaný generalizovatelný úkol anebo úkol často používaný, používáme funkce. Funkce je jeden ze základních nástrojů pro organizaci a vytváření opakovaně použitelného kódu (Dalším jsou třídy).

```
1 def hafni():
2     print("haf!")
3
4 hafni()
5 hafni()
```

Funkce má jméno, pro které platí běžná pravidla pro vytváření identifikátorů. Kde to je vhodné, doporučuji používat rozkazovací způsob.

```
1 def hafni(n):
2     for i in range(n):
3         print("haf!")
```

`n` je tady parametr funkce. Do hodnoty `n` se při spuštění funkce překopíruje hodnota z volání funkce a platí tady všechny varování ohledně kopírování - o tom budeme vícekrát mluvit později.

Máme Python 3.9, takže modernější verze funkce bude vypadat takto:

```
1 def hafni(n:int): # Uvádíme očekávaný typ parametru
2     for _ in range(n): # Používáme nepojmenovanou proměnnou
3         print("haf!")
```

Uvedením typu parametru zabezpečíme, že interpret nás bude varovat, pokud použijeme parametr nesprávného typu. To někdy pomáhá, a jindy to nechceme, protože Python nám umožňuje psát generický kód, jak vidíme hned v následujícím příkladu.

Návratová hodnota a příkaz return

```
1 def plus(x,y):
2     return x+y
3
4 print(plus(1,2))
5 print(plus("Ne", "hafnu!"))
```

Příkaz `return výraz` ukončí vykonávání funkce a vrátí jako hodnotu funkce `výraz`.

Nepovinné parametry

```
1 def hafni(krat:int = 1, zvuk:str = "Haf"):
2     for _ in range(krat):
3         print(zvuk)
4
5 hafni()
6 hafni(5)
7 hafni(zvuk = "Miau!")
8 hafni(krat = 5, zvuk = "Kokrh!")
```

Viditelnost proměnných: lokální a globální jmenný prostor

```
1 zvuk = "kuku!"
2 kolik_hodin = 0
3
4 def zakukej():
5     print(zvuk)
6     kolik_hodin += 1
```

Proměnné `kolik_hodin` přiřazujeme, a Python ji musí uvnitř funkce zřídit. Implicitní předpoklad je, že chcete zřídit novou proměnnou. Pokud chcete použít proměnnou z globálního prostoru jmen, musíte to Pythonu říci. **Proč je to tak dobře?**

```
1 zvuk = "kuku!"
2 kolik_hodin = 0
3
4 def zakukej():
5     global kolik_hodin
6     print(zvuk)
7     kolik_hodin += 1
```

Mimochodem, tato funkce dělá něco, čemu se typicky chceme vyhnout: ovlivňuje proměnnou, která není jejím parametrem. Toto nazývá vedlejší efekt a je to nejčastěji symptom špatného programování.

Správná funkce by měla být čistá, tedy by měla vypočítat a odevzdat svou návratovou hodnotu bez toho, aby měnila hodnoty nějakých proměnných, včetně svých parametrů.

Příklady funkcí, které určitě nejsou čisté, jsme viděli: jsou to metody seznamu, které nějak přetvářejí seznam na místě: `sort`, `reverse`. Tyto funkce mění seznam, který je volá a nevracejí hodnotu. Je to proto, že jde spíše o metody třídy `List`, tedy funkce, které patří do nějaké vyšší datové struktury a operují nad ní.

Příklady

Napište funkci, která

- vrátí nejmenší ze tří čísel
- vrátí n-té Fibonacciho číslo

U tohoto úkolu se zastavíme. Jednoduchá implementace požadované funkce vychází z faktu, že *Pythonovská funkce zná sebe samu*, takže ji v jejím těle můžeme volat:

```
1 # Fibonacci numbers recursive
2 def fib(n):
3     if n < 2:
4         return n
5     else:
6         return fib(n-1) + fib(n-2)
7
8 print(fib(5))
9
```

Takováto implementace je velice srozumitelná, ale má vadu: zkuste si spočítat `fib(35)`. Důvodem je, že každé volání funkce vede ke dvěma dalším voláním, takže počet volání potřebný pro výpočet `fib(n)` exponenciálně roste. Existují dva způsoby, jak vyřešit takovýto problém s rekurzí:

- Jedná se o primitivní, tedy odstranitelnou rekurzi, takže není složité vytvořit nerekurzivní implementaci.

```
1 # Fibonacci non-recursive
2
3 def fib(n):
4     if n < 2:
5         return n
6     else:
7         fpp = 0
8         fp = 1
9         for i in range(1,n):
10             fp, fpp = fp + fpp, fp
11
12     return fp
13
14 print(fib(35))
15
```

- Můžeme rekurzivní funkci "vypomocť" zvenčit tak, že si někde zapamatujeme hodnoty, které se již vypočetly, a tyto hodnoty budeme dodávat z paměti a nebudeme na jejich výpočet volat funkci. O této možnosti si víc řekneme někdy později.
- spočítá, kolik je v seznamu sudých čísel
- vybere ze seznamu sudá čísla (a vrátí jejich seznam)
- dostane dva seřazené seznamy čísel a vrátí jejich průnik
- dostane koeficienty kvadratické rovnice $ax^2 + bx + c = 0$ a vrátí seznam jejích kořenů.

Funkce jako plnoprávný Pythonovský objekt

Funkce může být přiřazována proměnným, předávána jiným funkcím jako parametr, a může být i návratovou hodnotou.

Funkce map a filter

Funkce map aplikuje hodnotu funkce na každý prvek seznamu nebo jiného iterovatelného objektu. Výsledkem je iterátor, takže pro přímé zobrazení je potřeba ho převést na seznam např. pomocí `list`

```
1 >>> seznam = [[1,2], [2,3,4], [4,5,6,7], [7,8,9], [9,10]]
2 >>> delky = map(len, seznam)
3 >>> delky
4 <map object at 0x00000213118A2DC0>
5 >>> list(delky)
6 [2, 3, 4, 3, 2]
7 >>> list(map(sum, seznam))
8 [3, 9, 22, 24, 19]
```

Můžeme taky použít vlastní funkci:

```
1 >>> seznam = [[1,2], [2,3,4], [4,5,6,7], [7,8,9], [9,10]]
2 >>> def number_from_digits(cisllice):
3     quotient = 1
4     number = 0
5     for d in reversed(cisllice):
6         number = number * 10 + d
7     return number
8
9 >>> list(map(number_from_digits, seznam))
10 [21, 432, 7654, 987, 109]
11 >>>
```

Podobně jako map funguje funkce `filter`: aplikuje na každý prvek seznamu logickou funkci a podle výsledku rozhodne, zda se má hodnota v seznamu ponechat.

```

1  >>> def len2(cisla) -> bool:
2      return len(cisla)>2
3
4  >>> filter(len2, seznam)
5  <filter object at 0x0000021310E6C0A0>
6  >>> list(filter(len2, seznam))
7  [[2, 3, 4], [4, 5, 6, 7], [7, 8, 9]]
8

```

Taky funkce...

Lambda-funkce

Kapesní funkce jsou bezejmenné funkce, které můžeme definovat na místě potřeby. Šetří práci například u funkcí jako `sort`, `min/max`, `map` a `filter`.

```

1  >>> seznam = [[0,10], [1,9], [2,8], [3,7], [4,6]]
2  >>> seznam.sort(key = lambda s: s[-1])
3  >>> seznam
4  [[4, 6], [3, 7], [2, 8], [1, 9], [0, 10]]

```

Generátory a příkaz yield

Už jsme se setkali s iterátory, např. víme, že dáme-li list comprehension do kulatých závorek místo hranatých, dostaneme namísto seznamu iterátor. Generátorem nazýváme funkci, která může fungovat jako iterátor - lze ji opakovaně volat, a ona pokaždé vrátí následující hodnotu z nějaké posloupnosti.

Příklad 1.

```

1  >>> def my_range(n):
2      k = 0
3      while k < n:
4          yield k
5          k += 1
6      return
7
8  >>> list(my_range(5))
9
10 [0,1,2,3,4]

```

Příklad 2 vám bude povědomý:

```

1  def read_list():
2      while True:
3          i = int(input())
4          if i == -1:
5              break
6          yield i

```

```
7     return
8
9     for i in read_list():
10         print(f"Načetlo se číslo {i}.")
11
12     print("konec cyklu 1")
13
14     for j in read_list():
15         print(f"Teď se načetlo číslo {j}")
16
17     print("konec cyklu 2")
```