`Programování` 1 pro matematiky

6. cvičení, 11-11-2021

tags: `Programovani 1 2021`, `čtvrtek`

Obsah:

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 3 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. Teda - pokud jsme posledně všechno prošli - prošli?

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:

```
def hafni(n:int): # Uvádíme očekávaný typ parametru
    for _ in range(n): # Používáme nepojmenovanou proměnnou
        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

```
def plus(x,y):
    return x+y

print(plus(1,2))
print(plus("Ne","hafnu!"))
```

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

Nepovinné parametry

```
def hafni(krat:int = 1, zvuk:str = "Haf"):
    for _ in range(krat):
        print(zvuk)

hafni()
hafni(5)
hafni(zvuk = "Miau!")
hafni(krat = 5, zvuk = "Kokrh!")
```

Viditelnost proměnných: lokální a globální jmenný prostor

```
zvuk = "Kuku!"
kolik_hodin = 0

def zakukej():
    print(zvuk)
    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?**

```
zvuk = "Kuku!"
kolik_hodin = 0

def zakukej():
    global kolik_hodin
    print(zvuk)
    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 vypočíst 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 ni.

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:

  ```
  # Fibonacci numbers recursive
  def fib(n):
      if n < 2:
          return n
      else:
          return fib(n-1) + fib(n-2)

  print(fib(5))

  ```

  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.

    ```
    # Fibonacci non-recursive

    def fib(n):
        if n < 2:
            return n
        else:
            fpp = 0
            fp = 1
            for i in range(1,n):
                fp, fpp = fp + fpp, fp

        return fp

    print(fib(35))

    ```

    - Můžeme rekurzivní funkci "vypomoct" zvenčí 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 $ax2 + bx + c = 0$ 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 potReba ho převést na seznam např. pomocí `list`

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

Můžeme taky použít vlastní funkci:

```
>>> seznam = [[1,2], [2,3,4], [4,5,6,7], [7,8,9], [9,10]]
>>> def number_from_digits(cislice):
    quotient = 1
    number = 0
    for d in reversed(cislice):
        number = number * 10 + d
    return number

>>> list(map(number_from_digits, seznam))
[21, 432, 7654, 987, 109]
>>>
```

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

```
>>> def len2(cisla) -> bool:
    return len(cisla)>2

>>> filter(len2, seznam)
<filter object at 0x0000021310E6C0A0>
>>> list(filter(len2, seznam))
[[2, 3, 4], [4, 5, 6, 7], [7, 8, 9]]

```

Funkce vytvářející funkce: dekorátory

Pojďme se vrátit k rekurzivní implementaci výpočtu Fibonacciho čísel:

```
# Fibonacci numbers recursive
def fib(n):
    if n < 2:
        return n
    else:
        return fib(n-1) + fib(n-2)

print(fib(5))

```

Chtěli bychom, aby se funkce volala jen v nevyhnutných případech, tedy když se počítá pro novou hodnotu `n`. Pro tento účel nebudeme upravovat funkci zevnitř, ale ji zabalíme:

- Vytvoříme funkci `memoize`, která jako parametr dostane původní "nahou" funkci `fib` a vrátí její upravenou verzi se zapamatováváním.
- Sice zatím nemáme úplně dobrou metodu jak si pamatovat sadu hodnot, pro které známe nějaký údaj, například sadu `n`, pro které známe `fib(n)`, ale můžeme si lehko pomoci dvojicí seznamů.

```
# Momoised Fibonacci

def memoize(f):
    values = [0,1]
    fibs = [0,1]
    def inner(n):
        if n in values:
            return fibs[values.index(n)]
        else:
            result = f(n)
            values.append(n) # musime aktualizovat najednou
            fibs.append(result)
            return result
    return inner

@memoize
def fib(n):
    if n < 2:
        return n
    else:
        return fib(n-1) + fib(n-2)

print(fib(100))
```

```
24
```

Abychom si ukázali další použití dekorátorů, zkusme zjistit, jak roste počet volání `fib(n)` u rekurzivní verze. Dekorátor, který na to použijeme, využívá pro ten účel zřízený atribut funkce:

```
# Dekorátor, počítající počet volání funkce
def counted(f):
    def inner(n):
        inner.calls += 1 # inkrementujeme atribut
        return f(n)
    inner.calls = 0 # zřizujeme atribut funkce inner
    return(inner)

@counted
def fib(n):
    if n < 2:
        return n
    else:
        return fib(n-1) + fib(n-2)

@counted # pro porovnání přidáme i nerekurzivní verzi funkce
def fib2(n):
    if n < 2:
        return n
    else:
        f, fp = 1, 0
        for i in range(1,n):
            f, fp = f+fp, f
        return f

for i in range(30):
    fib.calls = 0 # musíme resetovat počítadla
    fib2.calls = 0
    print(i, fib(i), fib.calls, fib2(i), fib2.calls)

```

Dekorátory umožňují změnit chování funkcí bez toho, aby bolo potřebné měnit kód, který je volá. Je to pokročilé téma, ale učí nás, že s funkcemi je možné dělat divoké věci. Není například problém zkombinovat dekorátory pro memoizaci a počítání volání:

```
@counted
@memoize
def fib(n):
    ...
```

bude bez problémů fungovat.

Taky funkce...

Lambda-funkce

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

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

```
>>> def my_range(n):
        k = 0
        while k < n:
            yield k
            k += 1
        return

>>> list(my_range(5)

[0,1,2,3,4]
```

Příklad 2 vám bude povědomý:

```
def read_list():
    while True:
        i = int(input())
        if i == -1:
            break
        yield i
    return

for i in read_list():
    print(f"Načetlo se číslo {i}.")

print("Konec cyklu 1")

for j in read_list():
    print(f"Teď se načetlo číslo {i}")

print("Konec cyklu 2")
```

Tento příklad je důležitý, protože umožňuje výrazně vylepšit strukturu kódu, který jste doteď vytvářeli pro domácí úkoly (ne že bych měl v plánu pronásledovat někoho, kdo bude načítat věci "postaru"). Generátor nám umožňuje oddělit načítání seznamu od jeho zpracování a to i v případě, že zpracováváme seznam sekvenčně, tedy ho nikam neukládáme.