`Programování` 1 pro matematiky

7. cvičení, 22-11-2023

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**
   - *Těší mne velké množství řešení u každé série*. Bohužel občas nestíhám, klidně napište, když potřebujete, abych se na něco podíval, jinak se snažím cyklicky procházet odevzdaná řešení.
   - Pokud byste chtěli zpomalit, můžeme ke konci semestru snížit počet zadávaných domácích úkolů - např. ze současnýc 2 + 1 na 1 + 1.

Kde se nacházíme Měli jsme několik týdnů, kde jsme víc psali kód než se učili nové věci. To se teď změní a dnes začneme funkcemi..

Domácí úkoly

Stížnosti

- Zaklínání v kódu

Kód je racionální a není v něm místo pro magické úkony. Například není potřebné, aby každý váš skript měl na začátku `import random`. Také není potřebné, abyste každý seznam konvertovali na `tuple`, když se chcete podívat na třetí položku.

- Chlapácké kódování

Když potřebujete něco udělat s číslicemi celého čísla, nemusí si to nutně vyžadovat intenzivní využívání operátorů `**`, `pow` a funkcí jako `log`, `floor` a podobně. Méně je více - pohodlně vystačíte s celočíselnou aritmetikou a operátory `//` a `%`.

- Evergreen

Když čtete ze vstupu seznam, zakončený na posledním řádku řetězcem "-1", nemůžete předpokládat, že to se přesně načte. Funkce `input()` čte ze vstupu celý řádek, takže klidně může načíst "-1 " nebo "-1\n".

```
while radek != "-1": # Špatně !!!

while "-1" in radek: # Správně
nebo
while int(radek) != -1: # Správně
nebo
while radek.strip() != -1: # Správně
```

- Pozorně čtěte zadání

a nedělejte jiné předpoklady, než garantuje zadání. Snažte se obsloužit okrajové případy, které reálně můžou nastat, ale ne ty, u nichž zadání garantuje, že nenastanou. Například že nedostanete žádná data pro vyhledání položky. Nekontrolujte platnost vstupních dat, kde to zadání nevyžaduje a nedefinuje, co má váš kód v takovém případě dělat. Při kódování buďte pozorní, ale ne přehnaně defenzivní.

- Čím méně úrovní vnoření, tím lepší (= čitelnější, srozumitelnější kód)
- Dávejte svým proměnným smysluplná jména. Opravdu se všechny nemusí jmenovat i, j, k, l, m. Klidně kde-tu přidejte komentář, když děláte něco, co není z vašeho kódu zřejmé.

Vyhledávání

Nemůžu v rámci kontroly domácích úkolů nemalé části z vás vysvětlovat, co jsme probírali ve cvičení. Občas se alespoň podívejte, co se probírá ve cvičení (adresováno nepřítomným).

Některé věci, které tady probíráme, jsou důležité, a intuice, že *když máme setříděná data, můžeme v nich vyhledávat velice rychle*, je jedna z nich.

Jemnosti algoritmů

Pokud máme v setříděném poli opakované hodnoty a ze série stejných hodnot chceme vrátit hodnotu s nejmenším indexem, je špatné, pokud musíte při nalezení hodnoty proskákat přes data k první hodnoté ze série. Dlouho to trvá a je to neefektivní.

Namísto toho chcete, aby vaše vyhledávání rovnou skončilo na tom, co hledáte.

Řešení:

```
from random import randint
import sys

data = sorted([randint(1,10) for _ in range(20)])

print(data)

target = int(input("Zadej číslo: "))

left = 0
right = len(data) - 1

if data[left] == target:
    print(f"Hodnota nalezena v poloze {left}")
    sys.exit()

while right - left > 1: # Končíme, když se hranice sejdou
    mid = (left + right) // 2
```

```
    val = data[mid]
    if val < target: # Nemáme větev s ==: nekončíme, když nalezneme hodnotu
        left = mid
    else:
        right = mid

if data[right] != target:
    print(f"Hodnota {target} nenalezena")
else:
    print(f"Hodnota {target} nalezena v poloze {right}")
```

Je docela podstatné, že hledáte ne hodnotu, ale první výskyt této hodnoty - jsou to dvě různé věci a algoritmus to musí zohlednit.

Funkce

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

```
def hafni/():
    print("Haf!")

hafni()
hafni()
```

Toto je velice prostá funkce, která se vyznačuje tím, že nemá žádný vstup - udělá vždy to samé - a nemá žádný výstup, pouze něco vypíše. Obecně funkce vrací výstup, který je nějak závislý od parametrů - argumentů funkce.

Funkce má jméno, pro které platí běžná pravidla pro vytváření identifikátorů. Kde to je možné, doporučuji používat sloveso a rozkazovací způsob: `replace, insert, copy, read_file, get_password, print_results`.

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

`n` je tady parametr neboli *argument* 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) -> None: # Očekávaný typ parametru a návratové hodnoty
    for _ in range(n): # Používáme nepojmenovanou proměnnou
        print("haf!")
```

Uvedením typu parametru dokumentujeme, co funkce očekává. Tyto "type hints" však nejsou v Pythonu vynucovány, tedy jejich nedodržení nevede k chybě. Na druhé straně, existuje několik nástrojů - např. *mypy*, které dokážou při statické kontrole kódu ve vhodných vývojových prostředích odhalit nesrovnalosti v typech proměnných.

Uvedení typu objektu někdy pomáhá, a jindy typ nechceme specifikovat: Python není staticky typovaný jazyk a umožňuje nám 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`.

Pojmenované parametry a 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

`dir()` zobrazí obsah aktuálního jmenného prostoru:

```
 >>> def hafni() -> None:
...     print("Haf!")
...
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
'__package__', '__spec__', 'hafni']
>>> x = 50
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
'__package__', '__spec__', 'hafni', 'x']
```

Co se tady stane?

```
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 $ax^2 + bx + c = 0$ 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.

(Super-)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`

```
>>> 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], [9,0]]
>>> def number_from_digits(cislice):
    quotient = 10
    number = 0
    for d in cislice:
        number = number * quotient + d
    return number

>>> list(map(number_from_digits, seznam))
[12, 234, 4567, 789, 90]
>>>
```

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]]
```

Obě tyto funkce mají jednoduché a čitelnější náhrady: *list comprehensions*":

```
>>> [number_from_digits(cislice) for cislice in seznam]
[12, 234, 4567, 789, 90]
>>> [cislice for cislice in seznam if len2(cislice)]
[[2, 3, 4], [4, 5, 6, 7], [7, 8, 9]]
```

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`.

```
>>> 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

Dáme-li list comprehension do kulatých závorek místo hranatých, dostaneme namísto seznamu iterátor.

```
>>> r = (x for x in range(20) if x % 3 == 2)
>>> r
<generator object <genexpr> at 0x000001BC701E9BD0>
>>> for j in r:
...     print(j)
...
2
5
8
11
14
17
```

Následující ukázka demonstruje, jak Python interaguje s iterátorem:

```
>>> s = (x for x in range(3))
>>> next(s)
0
>>> next(s)
1
>>> next(s)
2
>>> next(s)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

`next(it)` vrací další hodnotu iterátoru, a pokud už další hodnota není, vyvolá iterátor výjimku `StopIteration`. To je standardní chování iterátoru. Co se skrývá pod kapotou? Toto:

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]
```

Pokud vracíme hodnoty z posloupnosti, lze použít příkaz `yield from`:

```
>>> def my_range2(n):
        yield from range(n)

>>> list(my_range2(5)

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

Příklad 3 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")
```

Takto můžeme lehce oddělit cyklus zpracování dat od cyklu jejich načítání.