`Programování` 1 pro matematiky

7. cvičení, 11-11-2025

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**
    - Dostal jsem mnoho dobrých řešení, přibývají řešení s nápadem nebo poctivým kódem.

Dnešní program

1. Domácí úkoly a opakování
2. Pythonovské funkce a funkcím podobně věci

> Clear beats clever. Everytime.
>
> *Tom, the single dad janitor*

V praktickém balení: první člověk, který bude číst váš kód a snažit se ho pochopit, budete vy. Ulehčete si to.

Chytrý kód je užitečný, když dělá něco, co jinak nelze udělat.

Kvíz

```
a = [x for x in range(5)]
b = [x for x in range(7) if x in a and x%2==0]
print(b)
```

Co se vytiskne?

Domácí úkoly

Stížnosti

- Čím méně úrovní vnoření kódu, tím lepší (= čitelnější, srozumitelnější kód). Používejte `return` a `sys.exit()` pro snížení úrovně vnoření.
- Dávejte svým proměnným smysluplná jména. Váš kód nepoběží rychleji, když se vaše proměnné budou všechny jmenovat x, y, i, j, k, l, m.
- Klidně přidejte komentář tam, kde děláte něco, co není z vašeho kódu zřejmé.

Vyhledávání

Nemůžu v rámci kontroly domácích úkolů opakovaně vysvětlovat, co jsme probírali ve cvičení.

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

Paměť a rychlost kódu

Každá alokace/dealokace paměti si vyžaduje potenciálně dlouhý čas. Co můžete udělat najednou, nedělejte po kouskách.

Potenciálně dlouhý čas znamená, že ten čas může být relativně dlouhý v náhodných případech. Např. paměť pro seznam se přiděluje po blocích, tedy když seznam potřebuje dodatečnou paměť pro přidání položky, přidělí se mu paměť pro několik položek.

Tedy

- Pokud znáte velikost seznamu, alokujte jej najednou se správným typem, např. `seznam = [""] * len(data)`.
- Používejte index do předem alokovaného seznamu namísto `` `list.append `` nebo `list.pop`.
- Když jste použili položku ze seznamu, nemusíte ji hned mazat. Python to urobí za vás na konci programu.
- Každé pravidlo má výjimky a někdy musíte vzít hospodaření s pamětí do svých rukou a odstraňovat objekty, abyste si uvolnili paměť nebo vybudovali požadovanou strukturu.

```
seznam = list(range(1000))
seznam.clear()  # smaže všechny položky seznamu, proměnná "seznam" ukazuje na prázdný seznam.
del seznam      # smaže celý objekt seznamu. Proměnnou "seznam" Python nadále nezná.
```

Jemnosti algoritmů binárního vyhledávání

Algoritmus pro binární vyhledávání je poněkud jemný v tom, jak přesně nastavujeme okno pro hledání, tedy levý a pravý okraj.

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 = [randint(1,10) for _ in range(20)].sort

print(data)

target = int(input("Zadej číslo: "))

left = 0
right = len(data) - 1
```

```
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}")
```

Důležitý detail je ostrá nerovnost pro levou hranici a neostrá pro pravou. Když hledání skončí, pravá hranice je na místě první ze série stejných hodnot. Pokud nerovnosti obrátíme, bude levá hranice na místě poslední ze série stejných hodnot.

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

Jiné řešení

Abych jenom nekritizoval, několik z vás použilo pro řešení úkolu o vyhledávání v seznamu dobrou myšlenku: index: zapamatujeme si všechny unikátní hodnoty v seznamu spolu s prvním indexem, kde se vyskytnou.

```
data = [1, 1, 2, 2, 2, 3, 4, 4, 5, 5, 5]

index:
1: 0
2: 2
3: 5
4: 6
5: 8
```

Index lze nejlépe implementovat jako slovník - `dict`.

- Klíče obsahují všechny unikátní hodnoty v seznamu
- Hodnotou pro daný klíč je první index, kde se daná hodnota vyskytuje.

```
data = [1, 1, 2, 2, 2, 3, 4, 4, 5, 5, 5]

index = dict()
for i, val in enumerate(data):
    if val in index.keys():
        pass
    else:
        index[val] = i

target = int(input("Zadej hodnotu: "))
if target in index.keys():
    print(f"Hodnota nalezena na pozici {index[target]}.")
```

```
else:
    print("Hodnota nenalezena.")
```

Toto je efektivní, pokud vytvoření indexu vede ke kompresi dat, tedy když máte velký počet dat a dotazů ve srovnání s počtem unikátních hodnot. Vyhledávání ve slovníku je velice rychlé (konstantní čas).

Ještě jedno úplně nové řešení/

Je to varianta předchozího řešení s indexem, ale vytvářeným postupně. U této úlohy v tom není podstatný rozdíl, ale v případech, kdy mnohokrát voláme složitou funkci s různými parametry a nechceme zbytečně opakovat výpočet pro stejnou sadu parametrů, nemusí být první přístup vůbec použitelný, pokud předem neznáme kombinace parametrů, pro které budeme funkci počítat. Níže uvedený kód uvidíte v budoucnu vícekrát.

```
from random import randint
from bisect import bisect_left

data = [1, 1, 2, 2, 2, 3, 4, 4, 5, 5, 5]

queries = [randint(1, 5) for _ in range(20)]

def cached_bisect(data, queries):
    results = []
    cache = {}
    for query in queries:
        if query not in cache:  # query jsme zatím neviděli
            result = bisect_left(data, query)
            if result < length(data) and data[result] == query:
                cache[query] = result
            else:
                cache[query] = -1
        results.append(cache[Vquery])
```

Všimněte si prosím skrytý problém této implementace: Do hledací funkce jsme museli zabalit všechna data a všechny dotazy: musíme všechno udělat najednou, protože `cache` by se nezachovával mezi voláními funkce a určitě ho nechceme mít jako globální proměnnou - nechceme znečišťovat globální prostor vnitřnostmi funkcí. Ale už v příštím cvičení se naučíme, že pro kešování hodnot stačí v Pythonu udělat něco podivuhodně jednoduchého:

```
from random import randint
from bisect import bisect_left
from functools import cache

data = [1, 1, 2, 2, 2, 3, 4, 4, 5, 5, 5]

queries = [randint(1, 6) for _ in range(20)]

@cache
def bisect(query):
    result = bisect_left(data, query)
```

```
    if result < length(data) and data[result] == query:
        return result
    return -1

results = [bisect(query) for query in queries]
print(*results)
```

Na příštím cvičení si vysvětlíme, co jsme to vlastně udělali.

Minimální a maximální součet

Základní algoritmus v této úloze je setřídit data a vrátit součet prvních a posledních `k` hodnot. Ale nešlo by to ještě nějak jinak?

1. Na to, abychom našli `k` nejmenších a `k` největších čísel, musíte každé číslo vidět a porovnat ho s něčím jiným. Tedy nelze dosáhnout nic lepšího než lineární složitosti O(n).
2. Současně pomocí třídění *count sort* dokážete seznam celých čísel setřídit v lineárním čase O(n).

To znamená, že nijaký převratný přístup neexistuje, i když to neznamená, že všechny algoritmy se složitostí O(n) jsou stejně rychlé, robustní a stejně dobře se implementují. Algoritmus se složitostí 100n je pro většinu praktických aplikací pomalejší než algoritmus se složitostí 3n.log n.

Základní úvahy, které vedou k alternativním algoritmům, jsou tyto:

1. **Jaký je vztah velikosti seznamu** `n` a `k`? Pokud jsou `n` a `k` porovnatelná, potom setřídění seznamu bude nejrozumnější možností. Pokud je ale `n ≫ k`, pak je možná úspornější prostě vyhledat příslušný počet minim a maxim. Nebo je vyhledat v jednom průchodu pomocí struktury, zachovávající `k` nejmenších (největších) dosud viděných prvků.
2. **Co vlastně potřebujeme setřídit?** k prvních ani posledních čísel nemusí být setříděných, jenom musí být menší, resp. větší než zbývající čísla. Ani čísla uprostřed nemusí být setříděná.

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.

Podobný příklad: Quicksort

Toto je jeden z algoritmů se složitostí O(n log n).

Princip: Zvolíme si "prostřední" hodnotu, *pivot*, a rozdělíme data na tři části: hodnoty menší než pivot, hodnoty rovné pivotu a hodnoty větší než pivot. Postup rekurzivně opakujeme pro nesetříděné části.

```
from random import randint

def quicksort(data):
    if len(data) < 1:
        return data
    if len(data) == 2:
        return  [min(data), max(data)]

    pivot = data[randint(0, len(data)-1)]
    left = [i for i in data if i < pivot]
    pivots = [i for i in data if i == pivot]
    right = [i for i in data if i > pivot]
    return quicksort(left) + pivots + quicksort(right)


data = [randint(1,100) for _ in range(20)]
print(data)
sdata = quicksort(data)
print(sdata)

---
[14, 38, 34, 75, 69, 84, 57, 69, 67, 64, 60, 66, 93, 3, 60, 67, 27, 45, 64, 36]
[3, 14, 27, 34, 36, 38, 45, 57, 60, 60, 64, 64, 66, 67, 67, 69, 69, 75, 84, 93]
```

Toto není moc dobrá implementace, protože vyžaduje přibližně log n kopií tříděného seznamu v paměti.

Počet srovnání: V každém kroku přibližně n, kroků je přibližně $\log_2 n$.-

Funkce jako plnoprávný Pythonovský objekt

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

(Super-)funkce map, filter a sum

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]]
```

Další super-funkce: `functools.reduce` **a** `itertools.accumulate`

`reduce` rekurzivně aplikuje funkci dvou argumentů na předchozí výsledek (nebo počáteční hodnotu) a následující prvek posloupnosti, takže "redukuje" posloupnost na jediný výsledek.

```
from functools import reduce

data = list(range(10))
print(data)

print(reduce(lambda x, y: x + y, data) # stejné jako sum(data)
```

Funkce `accumulate` z modulu `itertools` je generátor, který funguje podobně jako `reduce`, ale namísto výsledné hodnoty vrací seznam mezivýsledků:

```
from itertools import accumulate
from random import shuffle

data = list(range(10))
shuffle(data)
print(*accumulate(data, max))

---
2 4 6 7 5 1 9 3 8 0
2 4 6 7 7 7 9 9 9 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.

--

Domácí úkoly

1. **Pyramidová výplň**: Vytvořte čtvercovou matici s předepsaným číselným vzorem.
2. **Sloučení dvou setříděných posloupností**: Máte dvě setříděné posloupnosti a máte je sloučit. Tato úloha je základem třídícího algoritmu *mergesort*, jednoho z rychlých algoritmů se složitostí $O(n \ln n)$.