`Programování` 1 pro matematiky

8. cvičení, 18-11-2025

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 průvodních textů k cvičením.
2. **Domácí úkoly**
   - pyramida - jednoduchý úkol, ojediněle nadměrně komplikovaná řešení, a dvě řešení, která řešila něco jiného.
   - sloučení setříděných posloupností - většina správných řešení.

Dnes

Začali jsme mluvit o funkcích a dnes budeme pokračovat, dnes se ale nejdřív vrátíme ke kontejnerům.

Domácí úkoly

Sloučení setříděných posloupností

Toto je docela důležitý kus kódu, který je základní součástí algoritmu *MergeSort*. Tento algoritmus má robustně složitost $O(n \ln n)$ a je to tedy velice výkonný třídící algoritmus. Stručně, algoritmus rekurzivně děli pole čísel na poloviny, dokud nezůstanou segmenty o velikosti 1, které se triviálně setřídí a postupně slučují pomocí algoritmu, který jste měli napsat.

Typické řešení kódu vypadá takto:

```
a = list(map(int, input().split()))
b = list(map(int, input().split()))
result = [0] * (len(a) + len(b))    # předem alokujeme výsledné pole

i = 0   # index do a
j = 0   # index do b
k = 0   # index do result

while i < len(a) and j < len(b):
    if a[i] <= b[j]:
```

```
        result[k] = a[i]
        i += 1
    else:
        result[k] = b[j]
        j += 1
    k += 1
while i < len(a):
    result[k] = a[i]
    i += 1
    k += 1
while j < len(b):
    result[k] = b[j]
    j += 1
    k += 1

print(*result)
```

Tento kód má složitost $O(n)$ - všechno děláme v jediném průchodu a každé porovnání vede k zařazení jednoho prvku do výsledné posloupnosti. V tomto kódu neměníme tvar datových struktur a výsledné pole alokujeme najednou.

Mnohým z vás jsem vyčítal řešení, které to dělá přesně naopak: prvky přidává k výslednému poli po jednom:

```
a = list(map(int, input().split()))
b = list(map(int, input().split()))
result = []

i = 0   # index do a
j = 0   # index do b

while i < len(a) and j < len(b):
    if a[i] <= b[j]:
        result.append(a[i])
        i += 1
    else:
        result.append(b[j])
        j += 1

result.extend(a[i:])
result.extend(b[j:])

print(*result)
```

Tento kód má stejnou složitost, $O(n)$. Podívejme se, proč bude přesto pomalejší.

append / extend vs. prealokace

Nejjednodušší způsob, jak změřit časovou náročnost nějaké kusu kódu je použít "magické slovíčko" `%%timeit` v IPythonu:

```
In [1]: %%timeit
   ...: n = 1_000_000
   ...: pole = []
```

```
   ...: for i in range(n):
   ...:     pole.append(i)
   ...:
   ...:
37.1 ms ± 961 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)

In [2]: %%timeit
   ...: n = 1_000_000
   ...: pole = [0] * n
   ...: for i in range(n):
   ...:     pole[i] = i
   ...:
   ...:
24.7 ms ± 503 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Jak vlastně funguje `list.append` a jaká je jeho složitost? Model:

- Prázdné pole se nealokuje prázdné, ale s nějakou zvolenou velikosti pro k položek.
- Dokud se pole nezaplní, připojení nového prvku se děje s konstantní složitostí.
- Když se pokusíte připojit položku, pro kterou už v poli není místo, alokuje se někde jinde nové pole o dvounásobné velikosti, prvky ze stávajícího pole se do něj překopírují a připojí se nový prvek. Tedy n + 1 operací.

Tedy typicky je pro přidání nového prvku potřebný konstantní čas, ale pro některé prvky je potřebný čas úměrný počtu stávajících prvků. Abychom si udělali lepší představu, uvažujme vybudování pole o velikosti n pomocí `list.append()`:

- k přidání s konstantní složitostí
- k prvků zkopírovat do nového pole
- k přidání s konstantní složitostí
- 2k prvků zkopírovat do nového pole
- 2k přidání s konstantní složitostí
- 4k prvků zkopírovat do nového pole.
- 4k přidání s konstantní složitostí
- 8k prvků zkopírovat do nového pole

Tedy pro přidání $2^m k$ prvků potřebujeme $2^{m+1} - 1)k$ operací a průměrná (*amortizovaná*) složitost na jedno přidání je konstantní. Na druhé straně, když pole alokujete předem, řada činností jako kopírování do pole dvounásobné velikosti se vůbec neděje, takže takový kód bude určitě rychlejší. Kromě toho pořád potřebujete mít k dispozici 1.5 - násobek paměti, potřebné na uložení pole.

Než se vrátíme k funkcím, probereme z rychlíku základní datové kontejnery. O všech jsme už mluvili, ale teď si povíme víc o jejich vlastnostech a metodách.

Znakové řetězce v Pythonu

Znakový řetězec je objet třídy `str`.

- Přístup k jednotlivým znakům a podřetězcům řetězce je stejný jako u seznamu
- Na rozdíl od řetězce je seznam *neměnný* (immutable), takže nefungují žádné funkce pro modifikaci řetězce "na místě" (např. `sort`). Fungují ale operátory pro vyhledávání v řetězci jako `index` a `count`.
- Funguje operátor `+` a logické operátory `==/!=` a `</>`, přičemž se používá lexikografické srovnání (podle UTF-8, ne podle češtiny, takže nebude respekovat např. české pořadí hlásek).

Metody třídy `str`

- hledání / nahrazování v řetězci: `count`, `find`/`rfind`, `index`/`rindex`, `replace`,
- velikost písmen: `capitalize`, `lower`, `upper`, `title`, `casefold`, `swapcase`
- zarovnání a vyplnění v poli: `center`, `ljust`/`rjust`, `expandtabs`, `strip`/`lstrip`/`rstrip`, `zfill`
- formátování řetězce: `format`
- dotazy na obsah řetězce: , `startswith`, `endswith`, `isalnum`, `isalpha`, `isascii`, `isdecimal`, `isdigit`, `isnumeric`, `islower`/`isupper`, `isspace`
- spojování a rozdělování `join`, `split`/`rsplit`, `splitlines`, `partition`/`rpartition`,

Úplnější seznam

Upozornění Tam, kde metoda vrací upravený řetězec, vrací upravenou kopii a původní řetězec se *nemění*.

Metoda	Popis
capitalize()	První písmeno na velké
casefold()	Všechna písmena na malá
center()	Vycentruje řetězec do pole s danou šířkou a výplní
count()	Počet výskytů znaku nebo řetězce v řetězci
endswith()	True pokud řetězec končí daným znakem/řetězcem
expandtabs()	Nahradí `\t` zadaným počtem mezer
find()	Vyhledá podřetězec a vrátí jeho pozici v řetězci
format()	Formátuje zadané hodnoty v řetězc
index()	Zděděné od seznamu, hledá pouze 1 znak
isalnum()	True pokud jsou všechny znaky alfanumerické
isalpha()	True pokud jsou všechny znaky písmena
isascii()	True pokud jsou všechny znaky ascii
isdigit()	True pokud jsou všechny znaky číslicemi
islower()	True pokud jsou všechna písmena malá

Metoda	Popis
isnumeric()	True, pokud jsou všechny znaky numerické
isspace()	True, pokud jsou všechny znaky ekvivalentní mezeře
isupper()	True pokud jsou všechna písmena velká
join()	Spojí prvky seznamu do řetězce proloženého daným řetězcem.
ljust()	Vrací doleva zarovnanou verzi řetězce s danou délkou a výplní
lower()	Skonvertuje písmena v řetězci na malá
lstrip()	Vrátí verzi řetězce ořezanou zleva na danou délku
partition()	Vyhledá řetězec a vrátí trojici všechno-před, hledaný-řetězec, všechno-za.
replace()	Vrátí řetězce, kde je podřetězec nahrazený jiným podřetězcem.
rfind()	Vrací pozici posledního výskytu
rindex()	Vrací pozici posledního výskytu
rjust()	Vrací doprava zarovnanou verzi s danou šířkou a výplní
rpartition()	Jako partition, ale hledá zprava
rsplit()	Jako split, ale hledá oddělovač zprava
rstrip()	Vrátí sprava ořezanou verzi řetězce
split()	Rozdělí řetězec u požadovaného separátoru a vrátí seznam
splitlines()	Rozdělí seznam u znaků nového řádku a vrátí seznam
startswith()	True, pokud řetězec začíná daným podřetězcem
strip()	Vrátí ořezanou verzi řetězce
swapcase()	Promění velikost, malá na velká a naopak
upper()	Vrátí řetězec, ve kterém jsou malá písmena nahrazena velkými
zfill()	Vyplní řetězce zleva předepsaným počtem nul

Množiny a slovníky

n-tice

n-tice je neměnná (immutable) struktura, která obsahuje několik objektů, které logicky patří k sobě, například souřadnice x, y bodu v rovině, den, měsíc a rok v datumu a pod.

```
>>> a = 1
```

```
>>> b = 2
>>> t = (a,b) # sbalení
>>> t
(1, 2)
>>> a = 2
>>> t
(1, 2)
>>> t[0]
1
>>> t[1]
2
>>> t[0] = 3
Traceback (most recent call last):
  File "<pyshell#292>", line 1, in <module>
    t[0] = 3
TypeError: 'tuple' object does not support item assignment
>>> x, y = t # rozbalení
>>> x
1
>>> y
2
```

Metody n-tic

Method	Description
count()	Kolikrát se daná hodnota nachází v n-tici
index()	Vrací index, na kterém se nachází určená hodnota

Funkce `enumerate` a `zip`

Chceme položky i s indexy. Standardní kód je iterovat přes index:

```
>>> mesta = ["Praha", "Brno", "Ostrava"]
>>> for i in range(len(mesta)):
    print(i, mesta[i])


0 Praha
1 Brno
2 Ostrava
>>> for i, mesto in enumerate(mesta):
    print(i, mesto)


0 Praha
1 Brno
2 Ostrava
>>> for u in enumerate(mesta):
    print(u)
```

```
(0, 'Praha')
(1, 'Brno')
(2, 'Ostrava')
```

Načítáme města a jejich souřadnice, a pak chceme iterovat přes trojice. Standardní kód je zase iterovat přes index:

```
>>> text = """
Praha -2 0
Brno 0 -1
Ostrava 1 1
"""
>>> mesta = []
>>> x = []
>>> y = []
>>> for radek in text.split("\n"):
    if len(radek) == 0:
        continue
    veci = radek.split()
    mesta.append(veci[0])
    x.append(float(veci[1]))
    y.append(float(veci[2]))


>>> mesta, x, y
(['Praha', 'Brno', 'Ostrava'], [-2.0, 0.0, 1.0], [0.0, -1.0, 1.0])

# Standardní způsob:
>>> for i in range(len(mesta)):
    print(mesta[i], x[i], y[i])


Praha -2.0 0.0
Brno 0.0 -1.0
Ostrava 1.0 1.0

# S využitím funkce zip:
>>> for mesto, x, y in zip(mesta, x, y):
    print(mesto, x, y)


Praha -2.0 0.0
Brno 0.0 -1.0
Ostrava 1.0 1.0
>>>
```

Množiny

Množiny jsou vysoce optimalizované kontejnery s rychlým vyhledáváním (vyhledávání / přidávání / odebírání O(1)).

Množny jsou

- **neměnné** ve smyslu, že nemůžete modifikovat položky, ale můžete je přidávat nebo odebírat.
- **neuspořádané** ve smyslu, že pořadí položek nezávisí od pořadí ve kterém byly přidávány.
- **neindexované** - k položkám nemáte přístup přes index.

```
>>> zvířata = {"kočka", "pes", "lev", "pes", "lev", "tygr"}
>>> zvířata
{'pes', 'tygr', 'lev', 'kočka'}
>>> "tygr" in zvířata # O(log n)
True
>>> set(["a", "b", "c"])
{'b', 'c', 'a'}
set("abrakadabra")
{'d', 'b', 'a', 'r', 'k'}
>>> set() # prázdná množina
set()
>>> {} # není prázdná množina!
{}
>>> type({})
<class 'dict'>
```

Množiny využívají stromové struktury a algoritmy pro rychlé vyhledávání a modifikaci. Vytváření množin a operace:

```
set("abrakadabra")
{'d', 'b', 'a', 'r', 'k'}
>>> a=set("abrakadabra")
>>> b=set("popokatepetl")
>>> "".join(sorted(a))
'abdkr'
>>> a & b # průnik
{'k', 'a'}
>>> a | b # sjednoceni
{'d', 'b', 'o', 'l', 'p', 'e', 'a', 'r', 't', 'k'}
>>> a - b # rozdíl
{'d', 'b', 'r'}
>>> a.remove("r")
>>> a
{'d', 'b', 'a', 'k'}
>>> b.add("b")
>>> b
{'o', 'b', 'l', 'p', 'e', 'a', 't', 'k'}
>>> a == b
False
```

Metody třídy množina

Metoda	Popis
add()	Přidání položky k množině
clear()	Vyprázdní množinu
copy()	Vrátí kopii množiny
difference()	Vrátí množinový rozdíl dvou nebo více množin
difference_update()	Odstraní z množiny prvky, které jsou již v jiné množině
discard()	Odstraní určenou položku
intersection()	Vrátí průnik dvou nebo více množin
intersection_update()	Odstraní z množiny položky, které se nenachází v jiné množině
isdisjoint()	True, pokud mají množiny prázdný průnik
issubset()	True, pokud je množina podmnožinou jiné
issuperset()	True, pokud množina obsahuje jinou množinu
pop()	Odstraní prvek z množiny
remove()	Odstraní určený prvek z množiny
symmetric_difference()	Vrací symetrický rozdíl dvou množin
symmetric_difference_update()	Vloží symetrický rozdíl této a jiné množny
union()	Vrací sjednocení dvou nebo více množín
update()	Přidá do množiny jinou množinu nebo seznam

Slovníky

```
>>> teploty = { "Praha": 17, "Dill´ı": 42,
"Longyearbyen": -46 }
>>> teploty
{'Praha': 17, 'Dill´ı': 42, 'Longyearbyen': -46}
>>> teploty["Praha"]
17
>>> teploty["Debrecen"]
Traceback (most recent call last):
  File "<pyshell#387>", line 1, in <module>
    teploty["Debrecen"]
KeyError: 'Debrecen'
>>> teploty["Debrecen"] = 28
>>>
>>> del teploty["Debrecen"]
```

```
>>> "Debrecen" in teploty
False
>>> teploty["Miskolc"]
Traceback (most recent call last):
  File "<pyshell#394>", line 1, in <module>
    teploty["Miskolc"]
KeyError: 'Miskolc'
>>> teploty.get("Miskolc")
None
>>> teploty.get("Miskolc", 20)
20

# Iterujeme ve slovníku:
>>> for k in teploty.keys():
    print(k)

Praha
Dill´ı
Longyearbyen
>>> for v in teploty.values():
    print(v)

17
42
-46
>>> for k, v in teploty.items():
    print(k, v)

Praha 17
Dill´ı 42
Longyearbyen -46
>>>
```

Metody slovníků

Method	Description
clear()	Odstraní všechny položky ze slovníku
copy()	Vrátí kopii slovníku
fromkeys()	Vrací slovník s určenými klíči a případně hodnotami
get()	Vrací hodnotu u určeného klíče
items()	Vrací seznam, obsahující dvojici pro každou dvojici klíč-hodnota
keys()	Vrací seznam klíčů slovníku
pop()	Odstraní a vrátí prvek s určeným klíčem
popitem()	Odstraní posledně vložený pár klíč-hodnota

Method	Description
setdefault()	Vrací hodnotu u požadovaného klíče. Pokud se klíč v seznamu nenachází, vrátí defaultní hodnotu a vytvoří v slovníku nový pár klíč-hodnota.
update()	Přidá do slovníku nové páry klíč-hodnota.
values()	Vrátí seznam všech hodnot ve slovníku.

Comprehensions pro množiny a slovníky:

```
>>> [i % 7 for i in range(50)]
[0, 1, 2, 3, 4, 5, 6, 0, 1, 2, 3, 4, 5, 6, 0, 1, 2, 3, 4, 5, 6, 0, 1, 2, 3, 4, 5, 6, 0, 1, 2, 3, 4, 5, 6, 0, 1, 2, 3, 4, 5, 6, 0]
>>> {i % 7 for i in range(50)}
{0, 1, 2, 3, 4, 5, 6}
>>> {i : i % 7 for i in range(50)}
{0: 0, 1: 1, 2: 2, 3: 3, 4: 4, 5: 5, 6: 6, 7: 0, 8: 1, 9: 2, 10: 3, 11: 4, 12: 5, 13: 6, 14: 0, 15: 1, 16: 2, 17: 3, 18: 4, 19: 5, 20: 6, 21: 0, 22: 1, 23: 2, 24: 3, 25: 4, 26: 5, 27: 6, 28: 0, 29: 1, 30: 2, 31: 3, 32: 4, 33: 5, 34: 6, 35: 0, 36: 1, 37: 2, 38: 3, 39: 4, 40: 5, 41: 6, 42: 0, 43: 1, 44: 2, 45: 3, 46: 4, 47: 5, 48: 6, 49: 0}
>>>
```

Počítání věcí

Toto je jedno z nejdůležitějších využití slovníku.

`defaultdict` - slovník s defaultní hodnotou *pro počítání*

```
>>> from collections import defaultdict
>>> pocet = defaultdict(int)
>>> pocet["abc"]
0
# počítáme slova
>>> for w in "quick brown fox jumps over lazy dog".split():
    pocet[w] += 1
>>> pocet
defaultdict(<class 'int'>, {'abc': 0, 'quick': 1, 'brown': 1, 'fox': 1, 'jumps': 1, 'over': 1, 'lazy': 1, 'dog': 1})
>>> list(pocet.items())
[('abc', 0), ('quick', 1), ('brown', 1), ('fox', 1), ('jumps', 1), ('over', 1), ('lazy', 1), ('dog', 1)]

# počítáme délky slov
>>> podle_delek = defaultdict(list)
>>> for w in "quick brown fox jumps over lazy dog".split():
    podle_delek[len(w)].append(w)

>>> podle_delek
defaultdict(<class 'list'>, {5: ['quick', 'brown', 'jumps'], 3: ['fox', 'dog'], 4: ['over', 'lazy']})
```

```
>>>
```

`collections.Counter`

```
>>> from collections import Counter
>>>
>>> myList = [1,1,2,3,4,5,3,2,3,4,2,1,2,3]
>>> print Counter(myList)
Counter({2: 4, 3: 4, 1: 3, 4: 2, 5: 1})
>>>
>>> print Counter(myList).items()
[(1, 3), (2, 4), (3, 4), (4, 2), (5, 1)]
>>>
>>> print Counter(myList).keys()
[1, 2, 3, 4, 5]
>>>
>>> print Counter(myList).values()
[3, 4, 4, 2, 1]
```

Opakování: Funkce

Příklady

Napište funkci, která

- vrátí řešení rovnice $2^x + x = 11$.

  Začneme tím, že zjevně $0 < x < 4$. a v tomto intervalu bude právě jedno řešení, protože funkce vlevo je rostoucí a vpravo konstantní. Programujeme, to, že řešení vidíte na první pohled, je dobré - máme kontrolu.

  Snažíme se udělat obecnější řešení:

  ```
  def fun(x):
      return 2**x + x - 11

  def eqn_solve(f, l, p, eps = 1.0e-6):
      """We expect f(l) < 0 < f(p)"""
      if f(l) > 0 or f(p) < 0:
          print("l a p musí ohraničovat oblast, kde se nachází kořen. ")
          return None
      while abs(l-p) > eps:
          m = (l+p)/2
          if f(m)<0:
              l = m
          else:
              p = m
      return m

  def main():
      print(eqn_solve(fun, 0, 4))
  ```

```
main()
```

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]]
```

Zkuste použít lambda funkci i jako parametr funkce `eqn_solve`.

```
print(eqn_solve(lambda x: 2**x + x - 20, 0, 4))
```

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.

Rekurze: Permutace

Chceme vygenerovat všechny permutace množiny (rozlišitelných) prvků. Nejjednodušší je použít rekurzivní metodu.

```
def getPermutations(array):
    if len(array) == 1:
        return [array]
    permutations = []
    for i in range(len(array)):
        # get all perm's of subarray w/o current item
        perms = getPermutations(array[:i] + array[i+1:])
        for p in perms:
            permutations.append([array[i], *p])
    return permutations

print(getPermutations([1,2,3]))
```

Výhoda je, že dostáváme permutace setříděné podle původního pořadí.

Nevýhoda je, že dostáváme potenciálně obrovský seznam, který se nám musí vejít do paměti. Nešlo by to vyřešit tak, že bychom dopočítávali permutace po jedné podle potřeby?

Jiný příklad: Kombinace

A co generátor kombinací? Kombinace jsou něco jiné než permutace - permutace jsou pořadí, kombinace podmnožiny dané velikosti.

Začneme se standardní verzí, vracející seznam všech kombinací velikosti n. Všimněte si prosím odlišnosti oproti permutacím:

```
def combinations(a, n):
    result = []
    if n == 1:
        for x in a:
            result.append([x])
    else:
        for i in range(len(a)):
            for x in combinations(a[i+1:], n-1):
                result.append([a[i], *x])
    return result

print(combinations([1,2,3,4,5],2))

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

Nové domácí úkoly

- **Benfordův test** - máte v načteném seznamu čísel určit počty čísel, začínajících stejnou číslicí.
- **Inverze slovníku** - máte vytvořit slovník, kterého klíči budou hodnoty a hodnotami klíče původního slovníku.