`Programování` 1 pro matematiky

6. cvičení, 3-11-2021

tags: `Programovani 1 2021`, `středa`

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%: 95

     Minimální počet bodů: 66
   - Ve vašich úkolech se objevuje čím dál víc dobrého kódu.

   Kde se nacházíme V posledník cvičeních jsem zpomalil a více času věnujeme psaní kódu. Toto tímto cvičením končí, zase nabereme nějaké nové věci.

Domácí úkoly

Psaní dobrého kódu: minimální a maximální součet k členů

V úloze o minimálním a maximálním součtu jste měli za úlohu najít k členů posloupnosti, kterých součet je minimální, resp. maximální, a tyto dva součty vypsat na konzoli.

Většina z vás řešení lehko našla: hledané součty budou součty pěti nejmenších a pěti největších hodnot posloupnosti. Nejlehčí řešení je tedy načíst posloupnost do seznamu, ten seřadit, a vypsat součet pěti prvních a pěti posledních prvků:

```
# Vypiš minimální a maximální součet k prvků posloupnosti

seznam = []
while True:
    a = int(input())
    if a == -1:
        break
     else:
        seznam.append(a)
k = int(input())

seznam.sort()
print(seznam[:k], seznam[-k:])
```

Takovýto postup může být drahý, pokud je délka posloupnosti podstatně větší než `k`. V takovém případě může být levnější najít `k` nejmenších a největších hodnot manuálně, například takto:

```
...
n = len(seznam)
for i in range(k): # k nejmenších hodnot
    pmin = i # poloha minima
    for j in range(i+1, n): # hledáme minimum ve zbývající části
        if seznam[j] < seznam[pmin]:
            pmin = j
    x[i], x[pmin] = x[pmin], x[i] # prohodíme s i-tým prvkem
minsum = sum(seznam[:k])
for i in range(n-1,n-k-1,-1):
    pmax = i
    for j in range(i-1,-1,-1):
        if seznam[j] > seznam[pmax]:
            pmax = j
    seznam[j], seznam[pmax] = seznam[pmax], seznam[j]
maxsum = sum(seznam[-k:])

```

Ještě bychom se měli zamyslet, jak bychom implementovali řešení pro velice dlouhé číselné řady. Tady jdeme nad rámec domácího úkolu, protože bychom potřebovali znát `k` před začátkem načítání posloupnosti.

```
# Průběžné hledání pěti největších a pěti nejmenších čísel
# v posloupnosti

from random import randint
# Místo načítání si seznam vygenerujeme
seznam = [randint(1,100) for _ in range(100)]

k = 5

low_list = [float("Inf")] * (k+1)
hi_list = [float("-Inf")] * (k+1)

for i in seznam:
    low_list[k] = i
    low_list.sort() # Podle potřeby později nahradíme
    hi_list[0] = i
```

```
    hi_list.sort()

print(low_list[:k], hi_list[-k:])

```

Tady můžeme pochopitelně ušetřit na opakovaném třídění seznamů `low_list` a `hi_list`. Nepotřebujeme třídit, pokud nová hodnota není zajímavá:

```
# Průběžné hledání pěti největších a pěti nejmenších čísel
# v posloupnosti

from random import randint
# Místo načítání si seznam vygenerujeme
seznam = [randint(1,100) for _ in range(100)]

k = 5
n_sorts = 0 # Budeme sledovat, kolik setřídění potřebujemeů

low_list = [float("Inf")] * (k+1)
hi_list = [float("-Inf")] * (k+1)

for i in seznam:
    if i < low_list[k-1]:
        low_list[k] = i
        low_list.sort()
        n_sorts += 1
    if i > hi_list[1]:
        hi_list[0] = i
        hi_list.sort()
        n_sorts += 1

print(low_list[:k], hi_list[-k:], n_sorts)

```

Jak udělat efektivnější třídění než je generický `list.sort`? Potřebovali bychom nějak využít, že vkládáme prvek do setříděného pole. Už jsme si ale říkali, jaká je výhoda setříděného pole: Umíme v něm vyhledávat v logaritmickém čase. Takže i místo, kam přijde nová hodnota, by mělo jít vyhledat rychle pomocí binárního vyhledávání:

```
# Insert a new value into a sorted list

from random import randint
# Generujeme setříděný seznam
seznam = [randint(1,100) for _ in range(100)]
seznam.sort()

hodnota = 55
p_vlozeni = None

if hodnota < seznam[0]:
    p_vlozeni = 0

elif hodnota >= seznam[-1]:
    p_vlozeni = len(seznam)

else:
```

```
    l = 0
    p = len(seznam) - 1
    # Z předchozího seznam]]<=hodnota<seznam[p]
    while p - l > 0:
        m = (p+l)//2
        print(l, m, p)
        if l == m:
            break
        if seznam[m] > hodnota:
            p = m
        elif seznam[m] <= hodnota:
            l = m
    p_vlozeni = p

print(p_vlozeni)

seznam.insert(p_vlozeni, hodnota)
print(seznam)

```

To vypadá na složitý kód, ale počet iterací `l,m, p` je úměrný logaritmu velikosti seznamu, takže takovéto vyhledávání se dívá jen na zlomek hodnot v seznamu.

Opakování

Drobnosti

- superfunkce nad seznamy: `min`, `max`, `sum`, `len`
- generování náhodných posloupností: `random.randint`
- Hezčí tisk seznamů:
  - `*[a, b, c, d]` rozbalí seznam na `a, b, c, d`.
  - separátor v příkazu `print`: `print(1, 2, 3, 4, sep="x")` vytiskne `1x2x3x4\n`. Pro připomenutí, už jsme měli parametr `end`: `print(1, 2, 3, 4, end = "x")` vytiskne `1 2 3 4x`. Jestli neuvedete `end`, použije se znak konce řádku `\n`, pokud neuvedente `sep`, použije se mezera .

Vyhledávání v setříděném seznamu

To je to, co potřebují dělat funkce `index` a `count` - najít hodnotu v setříděném seznamu, nebo zjistit, jestli se tam nachází, nebo v kolikrát.

S tímto jsme se už dnes setkali.

```
# Binární vyhledávání v setříděném seznamu

kde = [11, 22, 33, 44, 55, 66, 77, 88]
co = int(input())

# Hledané číslo se nachazí v intervalu [l, p]
l = 0
p = len(kde) - 1

while l <= p:
    stred = (l+p) // 2

```

```
    if kde[stred] == co:   # Našli jsme
        print("Hodnota ", co, " nalezena na pozici", stred)
        break
    elif kde[stred] < co:
        l = stred + 1      # Jdeme doprava
    else:
        p = stred - 1      # Jdeme doleva
else:
    print("Hledaná hodnota nenalezena.")
```

Toto je celkem silná zbraň, například pro řešení algebraických rovnic či minimalizaci.

Druhá odmocnina

```
# Najděte přibližnou hodnotu druhé odmocniny x

x = int(input())

l = 0
p = x # Velkorysé počáteční meze

while l < p:
    m = 0.5 * (l+p)
    # print(l, m, p)
    if m*m == x: # konec
        print(f"{x} is a perfect square of {m}") # format string
        break
    elif m*m < x:
        l = m
    else:
        p = m
    if p-l <= 1.0e-6:
        print(f"Square root of {x} is approximately {m}")
        break
```

"Bisection" je bezpečná, ale nikoli rychlá metoda hledání kořenů rovnice a minimalizace.

Například efektivnější aproximaci odmocniny získáme Newtonovou aproximací:

Třídění

Toto jenom prosvištíme, protože jsme podobný kód dnes už viděli:

Opakovaný výběr minima

Opakovaně vybíráme minimum a příslušnou hodnotu umísťujeme na začátek seznamu.

```
# Třídění opakovaným výběrem minima

x = [31, 41, 59, 26, 53, 58, 97]

n = len(x)
for i in range(n):
    pmin = i
    for j in range(i+1, n):
        if x[j] < x[pmin]:
```

```
            pmin = j
    x[i], x[pmin] = x[pmin], x[i]

print(x)

```

Bublinové třídění

Postupně "probubláváme" hodnoty směrem nahoru opakovaným srovnávaním se sousedy

```
# Třídění probubláváním

x = [31, 41, 59, 26, 53, 58, 97]

n = len(x)
for i in range(n-1):
    nswaps = 0
    for j in range(n-i-1):
        if x[j] > x[j+1]:
            x[j], x[j+1] = x[j+1], x[j]
        nswaps += 1
    if nswaps == 0:
        break

print(x)

```

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).

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

hafni()
hafni()
```

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

```
def hafni(n):
    for i in range(n):
        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.