

Programování 1 pro matematiky

5. cvičení, 3-11-2022

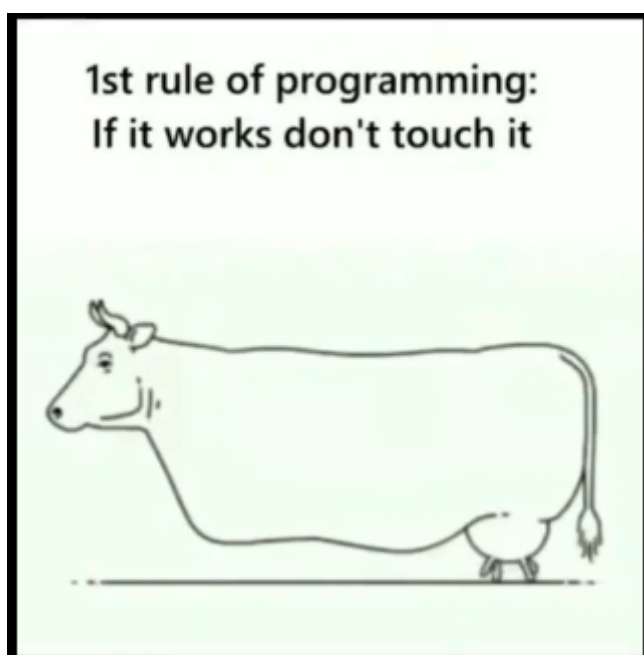
tags: Programování 1 2022, čtvrtek

Obsah:

0. Farní oznamy
1. Domácí úkoly
2. Opakování
3. Ještě medián: algoritmus s náročností $O(n)$
4. Třídění a binární vyhledávání
5. (možná) Pythonovské funkce

Farní oznamy

1. **Materiály k přednáškám** najdete v GitHub repozitáři <https://github.com/PKvasnick/Programovani-1>. Najdete tam také kód ke cvičením.
2. **Domácí úkoly** Minulý týden nebylo cvičení, ale dostali jste tři nové domácí úkoly. Už se k nim sešlo hodně řešení a zatím jediný problém vidím v tom, že možná byly příliš lehké.
Taky jsme uzavřeli dvě trojice úloh z předchozích týdnů. Řešení najdete na GitHubu. Několika poznámkami se k těmto úkolům ještě ve cvičení vrátím.



- Opravování či vylepšování kódu může vyjít draho - v kódu můžou být ukryté jemnosti, které nejsou na první pohled patrné.
 - Na druhé straně, dokázat přecíst kód a vylepšit ho - přeorganizovat anebo zrychlit - je součástí práce dobrého programátora.
-

Kvíz

Co se vypíše?

```
1 >>> print(len("".split(" ")))
2 >>> ???
```

- 1. 0
- 2. 1
- 3. 2
- 4. Chyba (výjimka)
- 5. Něco jiného

`string.split` - rozstříhá řetězec na kousky v definovaných místech, vrátí seznam

`string.join` - spojí řetězce ze seznamu tak, že mezi ně vkládá předepsaný řetězec

Domácí úkoly

Několik poznámek k expirovaným domácím úkolům.

Srozumitelnost kódu: Medián

Úloha o mediánu si vyžaduje setřídít seznam a vrátit buď prostřední hodnotu nebo průměr dvou prostředních hodnot, podle toho, je-li počet prvků v seznamu lichý anebo sudý:

$$Med(1, 2, \underline{3}, 4, 5) = 3$$
$$Med(1, \underline{2}, \underline{3}, 4) = \frac{2 + 3}{2} = 2.5$$

Z některých odevzdaných řešení by ale člověk mohl nabýt dojmu, že vypočítat něco takového je docela náročný úkol:

```
1 ...
2 nums = sorted(nums)
3 if len(nums)%2==0:
4     print((nums[int((len(nums))/2)]+nums[int((len(nums))/2)-1])/2)
5 else:
6     print(nums[int(len(nums)/2)])
```

nebo dokonce

```
1 ...
2 list.sort()
3 if len(list)%2!=0:
4     print(list[int((len(list)-1)/2)])
5 else:
6     print(float((list[int(len(list)/2)]+list[int((len(list)-2)/2])/2))
```

Takovýto kód je nepřehledný. Nešlo by to napsat nějak srozumitelněji, aby bylo jasné na první pohled, že to je správně?

```
1 (5) 0 1 2 3 4 -> 2
2 (7) 0 1 2 3 4 5 6 -> 3
```

tedy pro lichý počet prvků máme `len(list)//2`

```
1 (6) 0 1 2 3 4 5 -> 2,3
2 (8) 0 1 2 3 4 5 6 7 -> 3,4
```

a pro sudý počet máme `len(list) //2 - 1` a `len(list) //2`, takže kód může vypadat nějak takto:

```
1 midpoint = len(list) // 2
2 if len(list) % 2 == 1:
3     median = list[midpoint]
4 else:
5     median = 0.5*(list[midpoint-1] + list[midpoint])
```

Na medián se dnes ještě jednou podíváme poněkud jinak.

Výplně

aneb proč se oplatí strávit chvíli s tužkou a papírem

Máme matici $n \times n$. Úloha je vyplnit ji spirálově směrem od levého horního rohu do středu čísly $1, 2, \dots, n^2$.

```
1 1 2 3 4
2 12 13 14 5
3 11 16 15 6
4 10 9 8 7
```

Plánujeme a kreslíme

Implementace: Matici umíme implementovat jako seznam seznamů. Například matici $n \times n$ vyplněnou čísly od 1 do n^2 můžeme vytvořit takto:

```
1 >>> n = 5
2 >>> matrix = [[i+n*j for i in range(n)] for j in range(n)]
3 >>> matrix
4 [[0, 1, 2, 3, 4], [5, 6, 7, 8, 9], [10, 11, 12, 13, 14], [15, 16, 17, 18, 19], [20, 21, 22, 23, 24]]
```

Maticí procházíme střídavě po řádcích, tedy po všech prvcích jediného podseznamu, a po sloupcích, tedy po stejném prvku všech podseznamů:

```
1 >>> for col in range(n):
2     print(matrix[0][col], end = ' ')
3
4 0 1 2 3 4
5 >>> for row in range(n):
6     print(matrix[row][n-1], end = ' ')
7
8 4 9 14 19 24
```

```

9  >>> for col in range(n-1,-1,-1):
10     print(matrix[n-1][col], end = ' ')
11
12  24 23 22 21 20
13  >>> for row in range(n-1,-1,-1):
14     print(matrix[row][0], end = ' ')
15
16  20 15 10 5 0

```

Poslední, co potřebujeme, je spirálovité procházení. Teď jsme procházeli řádky a sloupce 0 a n-1. Pro spirálovité vyplňování potřebujeme posouvat horní, pravý, spodní a levý okraj tak, že po přejetí ho vždy posuneme o 1. Zavedeme proměnné `top`, `right`, `bot`, `left`, a pak to už jen všechno sestavíme:

```

1  # vyplňujeme matici nxn spirálovitě z levého horního rohu doprostředka čísla
2  # 1,2,...n*n.
3
4  n = int(input())
5
6  matrix = [[1 for _ in range(n)] for _ in range(n)] # _ protože proměnné
7  nepotřebujeme
8
9  left, right, top, bot = 0, len(matrix[0])-1, 0, len(matrix)-1
10 i = 1 # číslo, které vyplňujeme
11 while left <= right and top <= bot: # pokračujeme, dokud je kam vyplňovat
12     # Horní řádek zleva doprava
13     for col in range(left, right+1):
14         matrix[top][col] = i # vyplníme a inkrementujeme
15         i += 1
16     top += 1 # číslování je shora zleva
17
18     for row in range(top, bot + 1):
19         matrix[row][right] = i
20         i += 1
21     right -= 1
22
23     for col in range(right, left-1, -1):
24         matrix[bot][col] = i
25         i += 1
26     bot -= 1
27
28     for row in range(bot, top-1, -1):
29         matrix[row][left] = i
30         i += 1
31     left += 1
32
33 for row in range(n):
34     for col in range(n):
35         print(f'{matrix[row][col]:3}', end = '')
36     print('\n')

```

Opakování

Seznamy, řezy, metody

Index vrací položku, řez (slice) vrací seznam

```
1 | P y t h o n
2 | 0 1 2 3 4 5 6
3 | 0 1 2 3 4 5
```

```
1 | >>> s = [i for i in range(10)]
2 | >>> s
3 | [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
4 | >>> s[3]
5 | 3
6 | >>> s[3:4]
7 | [3]
8 | >>> s[:3]
9 | [0, 3, 6, 9]
10 | >>> s[10::-1]
11 | [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
12 | >>> s[::-1]
13 | [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

Metody seznamů

- `list.append(x)` přidává položku na konec seznamu.
- `list.extend(iterable)` rozšíří seznam připojením všech prvků `iterable` na konec seznamu.
- `list.insert(i, x)` vloží položku `x` na pozici `i`.
- `list.remove(x)` odstraní ze seznamu první položku s hodnotou `x`.
- `list.pop([i])` odstraní položku na zadané pozici v seznamu anebo poslední položku, pokud index není zadán.
- `list.clear()` odstraní všechny položky ze seznamu.
- `list.index(x[, začátek[, konec]])` vrátí index (počítaný od 0) v seznamu, kde se nachází první položka s hodnotou rovnou `x`.
- `list.count(x)` určí, kolikrát se `x` nachází v seznamu.
- `list.sort(, klíč=None, reverse=False)` utřídí položky seznamu *na místě*.
- `list.reverse()` na místě obrátí pořadí prvků v seznamu.
- `list.copy()` vrací plytkou kopii seznamu.

Logický operátor `in`

Zjišťuje, zda se v iterovatelném objektu nachází daná hodnota.

```
1 | list = [10, 20, 30, 40, 50 ];
2 |
3 | if 10 in list:
4 |     print("y se nachází v seznamu")
```

Pomůcka - opakování

- `*[a, b, c, d]` rozbalí seznam na `a, b, c, d`.
- `_` zastupuje proměnnou, kterou jinak nepoužíváme a nepotřebujeme definovat její jméno.
- 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 neuvedete `sep`, použije se mezera.

Ještě medián

- Vaše řešení obsahovala v nějakém kroku setřídění načtené posloupnosti, něco jako

```
seznam.sort
```


anebo

```
seznam = sorted(seznam)
```
- Je ale potřebné seznam setřídít? O hodnotě mediánu přece nerozhoduje pořadí největších a nejmenších hodnot, takže bychom potenciálně mohli ušetřit nějakou práci.

Příbuzná úloha: Najít v seznamu k -tou největší hodnotu. Pro posloupnost s lichým počtem n členů je medián $n // 2$ -á největší (nebo nejmenší) hodnota, pro sudé n potřebujeme členy $n//2 - 1$ a $n//2$, takže pokud vyřešíme tuto obecnější úlohu, bude její řešení použitelné i pro medián.

Rozděl a panuj Budeme postupovat tak, že vytvoříme sérii částečných uspořádání tak, abychom po každém kroku mohli eliminovat část posloupnosti, ve které se medián určitě nenachází.

1. Zvolíme si hodnotu - *pivot* - a rozdělíme posloupnost na dva podseznamy, *menší* a "větší", s hodnotami menšími, resp. většími nežli pivot.
2. Podle toho, do kterého seznamu připadne hledaný index k , pokračujeme jenom s jedním z těchto podseznamů a vracíme se do kroku 1.
3. Pokračujeme, dokud nedospějeme k podseznamu délky 1. Hodnota, kterou obsahuje, je to, co hledáme.

Otázka je, jak pro daný seznam zvolit pivot. My použijeme náhodný pivot - prostě náhodně zvolíme za pivot některou hodnotu ze seznamu. To není optimální, ale funguje to docela dobře.

```
1 seznam = [randint(1,100) for _ in range(10)]
2 # while (hodnota := int(input())) != -1:
3 #     seznam.append(hodnota)
4
5 k = int(input())
6 print(f"k={k}")
7
8 assert(0 <= k < len(seznam))
9
10 low = 0
11 high = len(seznam)
12 while high - low > 1:
13     pivot = seznam[randint(low, high - 1)]
14     print(f"low={low} high={high} pivot={pivot} seznam={seznam}")
15     low_numbers = [x for x in seznam[low:high] if x <= pivot]
```

```

16     high_numbers = [x for x in seznam[low:high] if x > pivot]
17     seznam = seznam[:low] + low_numbers + high_numbers + seznam[high:]
18     mid = low + len(low_numbers)
19     if k - 1 < mid:
20         high = mid
21     else:
22         low = mid
23
24
25     print(seznam[low])
26     ----- výstup -----
27     k=7
28     low=0 high=10 pivot=27 seznam=[98, 84, 47, 27, 46, 48, 21, 65, 32, 59]
29     low=2 high=10 pivot=59 seznam=[27, 21, 98, 84, 47, 46, 48, 65, 32, 59]
30     low=2 high=7 pivot=59 seznam=[27, 21, 47, 46, 48, 32, 59, 98, 84, 65]
31     low=2 high=7 pivot=48 seznam=[27, 21, 47, 46, 48, 32, 59, 98, 84, 65]
32     59
33

```

Proč má toto náročnost $O(n)$?

$$T(n) \approx n + \frac{n}{2} + \frac{n}{4} + \dots = 2n$$

Při náhodném výběru pivotu má algoritmus tuto náročnost pouze v průměru. Pro deterministický algoritmus potřebujeme nějak inteligentněji zvolit pivot: medián mediánů.

Binární vyhledávání a třídění

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.

Algoritmus: Půlení intervalu (proto *binární*).

Náročnost: $\log(n)$

```

1  #!/usr/bin/env python3
2  # Binární vyhledávání v setříděném seznamu
3
4  kde = [11, 22, 33, 44, 55, 66, 77, 88]
5  co = int(input())
6
7  # Hledané číslo se nachází v intervalu [l, p]
8  l = 0
9  p = len(kde) - 1
10
11 while l <= p:
12     stred = (l+p) // 2
13     if kde[stred] == co:    # Našli jsme
14         print("Hodnota ", co, " nalezena na pozici", stred)
15         break
16     elif kde[stred] < co:
17         l = stred + 1      # Jdeme doprava

```

```

18     else:
19         p = stred - 1      # jdeme doleva
20     else:
21         print("Hledaná hodnota nenalezena.")

```

Aplikace: Řešení algebraických rovnic, minimalizace

Celočíselná druhá odmocnina

```

1  # Emulate math.isqrt
2
3  n = int(input())
4
5  l = 0
6  p = n # velkorysé počáteční meze
7
8  while l < p:
9      m = int(0.5 * (l+p))
10     # print(l, m, p)
11     if m*m == n: # konec
12         print(f"{n} is a perfect square of {m}") # format string
13         break
14     elif m*m < n:
15         l = m
16     else:
17         p = m
18     if p-l <= 1:
19         print(f"{n} is not perfect square, isqrt is {l}")
20         break

```

Úloha: Odmocnina reálného čísla

Řešení rovnice $\cos(x) = x$

```

1  # solve x = cos(x) by bisection
2  from math import pi, cos
3
4  l = 0.0
5  p = pi/2.0
6
7  while p - l > 1.0e-6: # Tolerance
8      m = 0.5*(l + p)
9      print(l, m, p)
10     if m - cos(m) == 0:
11         print(f"Found exact solution x = {m}")
12         break
13     elif m - cos(m) < 0:
14         l = m
15     else:
16         p = m
17 else:
18     e = 0.5 * (p-l)
19     print(f"Converged to solution x = {m}+/-{e}")

```

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

Třídění

Opakovaný výběr minima

Opakovaně vybíráme minimum a příslušnou hodnotu umísťujeme na začátek seznamu.

```
1  # Třídění opakovaným výběrem minima
2
3  x = [31, 41, 59, 26, 53, 58, 97]
4
5  n = len(x)
6  for i in range(n):
7      pmin = i
8      for j in range(i+1, n):
9          if x[j] < x[pmin]:
10             pmin = j
11         x[i], x[pmin] = x[pmin], x[i]
12
13 print(x)
14
```

Bublinové třídění

Postupně "probubláváme" hodnoty směrem nahoru opakovaným srovnáváním se sousedy

```
1  # Třídění probubláváním
2
3  x = [31, 41, 59, 26, 53, 58, 97]
4
5  n = len(x)
6  for i in range(n-1):
7      nswaps = 0
8      for j in range(n-i-1):
9          if x[j] > x[j+1]:
10             x[j], x[j+1] = x[j+1], x[j]
11             nswaps += 1
12         if nswaps == 0:
13             break
14
15 print(x)
16
```

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:

```
1 def hafni(n:int): # Uvádíme očekávaný typ parametru
2     for _ in range(n): # Používáme nepojmenovanou proměnnou
3         print("haf!")
```

Uvedením typu parametru zabezpečíme, že interpret nás bude varovat, pokud použijeme parametr nesprávného typu. To někdy pomáhá, a jindy nám to zabraňuje psát generický kód.

Návratová hodnota a příkaz return

```
1 def plus(x,y):
2     return x+y
3
4 print(plus(1,2))
5 print(plus("Ne", "hafnu!"))
```

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

Nepovinné parametry

```
1 def hafni(krat:int = 1, zvuk:str = "Haf"):
2     for _ in range(krat):
3         print(zvuk)
4
5 hafni()
6 hafni(5)
7 hafni(zvuk = "Miau!")
8 hafni(krat = 5, zvuk = "Kokrh!")
```

Viditelnost proměnných: lokální a globální jmenný prostor

```
1 zvuk = "kuku!"
2 kolik hodin = 0
3
4 def zakukej():
5     print(zvuk)
6     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.

```
1  zvuk = "kuku!"
2  kolik hodin = 0
3
4  def zakukej():
5      global kolik.hodin
6      print(zvuk)
7      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 vypočítat 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 ní.