# `Programování` 1 pro matematiky

# 5. cvičení, 27-10-2021

**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.
2. **Domácí úkoly** Dostali jste zatím 9 úkolů k prvním třem cvičením.
   - Nominální počet bodů (bez "bonusových" úloh) - 100%: 75

     Minimální počet bodů: 52
   - Těší mne, že několik z vás přešlo z kritického do bezpečného pásma a mnozí píšete hezký kód.
   - Přecházíme do pravidelnějšího rytmu - na poslední chvíli, protože průběžná kontrola odevzdaných příkladů mi začíná přerůstat přes hlavu

   **Kde se nacházíme** V posledník cvičeních se mírně odchylujeme od oficiálního sylabu
   - Některé věci jsme probrali dřív
   - Chci co nejvíc času věnovat psaní kódu.

---

## Domácí úkoly

### Srozumitelnost kódu: Medián

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

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

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

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

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

nebo dokonce

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

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

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

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

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

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

## Srozumitelnost kódu: čtverce v trojúhelníku

**Nebojte se při programování vzít do ruky tužku a papír.** Dobrý kus kódu si často vyžaduje kus plánování a někdy ikus matematiky.

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

Toto samozřejmě jde vyplňovat, například nějak takto (snažíme se, aby jsme pokud možno měli hezké indexy)

```
# Trojúhelník

```

```
n = int(input())

triangle = []
triangle.append([]) # řádek 0, kde nic není

row = 1 # délka řádku "row" je row.
number = 1 # první číslo jde do řádku 1

for row in range(1,n+1): # 1..n
    triangle.append([]) # zakladám nový řádek trojúhelníku
    for _ in range(1,row+1): # 1..row
        triangle[row].append(number)
        number += 1

triangle.remove([]) # pro pořádek odstraním nultý řádek
print(triangle)

5:
[[1], [2, 3], [4, 5, 6], [7, 8, 9, 10], [11, 12, 13, 14, 15]]
```

Teď už není problém spočíst kvadráty čísel v některém řádku.  Ale někteří z vás si také všimli čísla na pravém okraji trojúhelníku:

```
           1 = 1x2/2
         2   3 = 2x3/2
       4   5   6 = 3x4/2
     7   8   9  10 = 4x5/2
  11  12  13  14  15 = 5x6/2
         ....
```

To jsou trojúhelníková čísla

$$T_n = \frac{n(n+1)}{2}$$

a pak součet čísel v posledním řádku můžeme napsat jako:

$$\sum_{T_{n-1}+1}^{T_n} k^2 = \sum_{k=1}^{T_n} k^2 - \sum_{k=1}^{T_{n-1}} k^2$$

a protože víme, že

$$\sum_{k=1}^{N} k^2 = \frac{N(N+1)(2N+1)}{6}$$

umíme napsat řešení bez toho, abychom generovali trojúhelník. *Dokonce bez toho, abychom použili Python.*

**Nechci po vás geniální řešení**, ale důkladný kód - i když autorům důvtipných řešení občas příhodím bonusový bod.

## Podobný příklad

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

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

### Plánujeme a kreslíme

Implementace: Matici umíme implementovat jako seznam seznamů. Například matici vyplněnou jedničkami můžeme implementovat takto:

```
>>> n = 5
>>> matrix = [[i+n*j for i in range(n)] for j in range(n)]
>>> matrix
[[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:

```
>>> for col in range(n):
    print(matrix[0][col], end = ' ')

0 1 2 3 4
>>> for row in range(n):
    print(matrix[row][n-1], end = ' ')

4 9 14 19 24
>>> for col in range(n-1,-1,-1):
    print(matrix[n-1][col], end = ' ')

24 23 22 21 20
>>> for row in range(n-1,-1,-1):
    print(matrix[row][0], end = ' ')

20 15 10 5 0
```

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

```
# Vyplňujeme matici nxn spirálovitě z levého horního rohu doprostředka čísly
# 1,2,...n*n.

n = int(input())

matrix = [[1 for _ in range(n)] for _ in range(n)]  # _ protože proměnné
nepotřebujeme

left, right, top, bot = 0, len(matrix[0])-1, 0, len(matrix)-1
```

```
i = 1 # číslo, které vyplňujeme

while left <= right and top <= bot: # pokračujeme, dokud je kam vyplňovat
    # Horní řádek zleva doprava
    for col in range(left, right+1):
        matrix[top][col] = i # vyplníme a inkrementujeme
        i += 1
    top += 1 # číslování je shora zleva

    for row in range(top, bot +1):
        matrix[row][right] = i
        i += 1
    right -= 1

    for col in range(right, left-1, -1):
        matrix[bot][col] = i
        i += 1
    bot -= 1

    for row in range(bot, top-1, -1):
        matrix[row][left] = i
        i += 1
    left += 1

for row in range(n):
    for col in range(n):
        print(f'{matrix[row][col]:3}', end = '')
    print('\n')
```

# Opakování

## Seznamy, řezy, metody

Index vrací položku, řez (slice) vrací seznam

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

```
>>> s = [i for i in range(10)]
>>> s
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> s[3]
3
>>> s[3:4]
[3]
>>> s[::3]
[0, 3, 6, 9]
>>> s[10::-1]
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
>>> s[::-1]
[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*[, *zacatek*[, *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.

```
list = [10, 20, 30, 40, 50 ];

if 10 in list:
   print("y se nachází v seznamu")
```

**Pomůcka - opakování**

Odkud vezmeme posloupnost?

```
from random import randint

print(*[randint(1, 10) for _ in range(10)], sep=" ")
```

- `*[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 neuvedente `sep`, použije se mezera .

---

## Binární vyhledáváni a třídění

S tímto sme začli na konci předchozího dvičení.

### 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)

```
```

```python
#!/usr/bin/env python3
# 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.")
```

## Aplikace: Řešení algebraických rovnic, minimalizace

### Celočíselná druhá odmocnina

```python
# Emulate math.isqrt

n = int(input())

l = 0
p = n # Velkorysé počáteční meze

while l < p:
    m = int(0.5 * (l+p))
    # print(l, m, p)
    if m*m == n: # konec
        print(f"{n} is a perfect square of {m}") # format string
        break
    elif m*m < n:
        l = m
    else:
        p = m
    if p-l <= 1:
        print(f"{n} is not pefect square, isqrt is {l}")
        break
```

*Úloha:* Odmocnina reálného čísla

### Řešení rovnice cos(x) = x

```python
# solve x = cos(x) by bisection
from math import pi, cos

l = 0.0
p = pi/2.0

```

```
while p - l > 1.0e-6: # Tolerance
    m = 0.5*(l + p)
    print(l, m, p)
    if m - cos(m) == 0:
        print(f"Found exact solution x = {m}")
        break
    elif m - cos(m) < 0:
        l = m
    else:
        p = m
else:
    e = 0.5 * (p-l)
    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.

```
# 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 vyhledávání

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)
```

```
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

```
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.

```
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.