`Programování` 1 pro matematiky

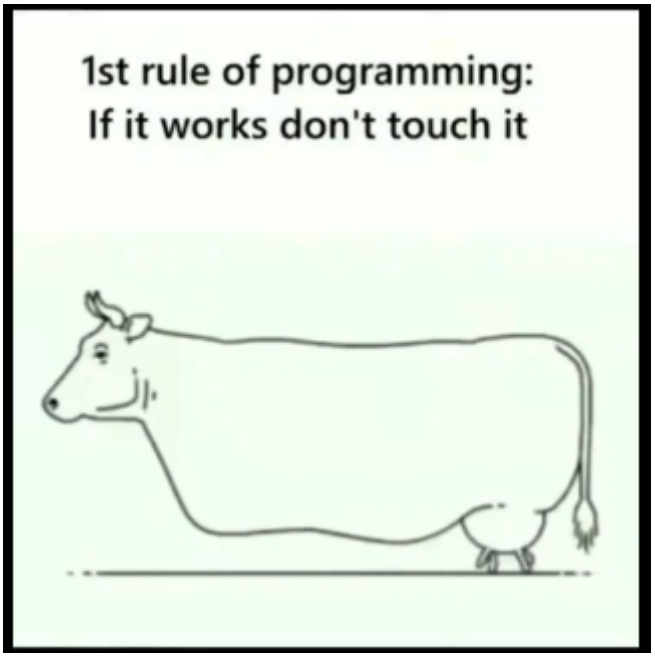
5. cvičení, 29-10-2024

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** Minulý týden jste dostali tři nové domácí úkoly. Už se k nom sešlo hodně řešení a zatím jediný problém vidím v tom, že možná byly ptíliš lehké.

   Také jsme uzavřeli další trojici úloh z před-předchozího týdne. Řešení najdete na GitHubu.



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

Kvíz

Ještě kopírování seznamů

```
x = [[1,2],[3,4]]
y = x.copy()
y[0][0] = 10
x
[[10, 2], [3, 4]]
```

Řešení: `deepcopy`:

```
from copy import deepcopy

x = [[1,2],[3,4]]
y[0][0] = 10
x
[[1, 2], [3, 4]]
y
[[10, 2], [3, 4]]
```

`else` v cyklu `for`

```
for i in range(3):
    user_pwd = input().strip()
    if user_pwd == "top_secret_pwd":
        print("Logging in...")
        break
else:
    print("Invalid password in 3 attempts. Quitting.")
```

`else` slouží typicky tam, kde v cyklu `for` opakujeme pokusy o provedení nějaké akce. Větev `else` pak slouží k obsloužení neúspěchu.

Domácí úkoly

Několik poznámek:

Srozumitelnost kódu: Medián

Úloha o mediánu si vyžaduje setřídit seznam a vrátit buď prostřední hodnotu nebo průměr dvou prostředních hodnot, podle toho, zda je 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`. Pro sudý počet prvků

```
(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])
```

Elegantní, ale poněkud kryptické:

```
midpoint = len(list) // 2
median = 0.5*(list[midpoint] + list[~midpoint])
```

~ je operátor bitové negace, např. `~3 = ~0b11 = -0b100 = -4`, přidal se znaménkový bit.

Na medián se dnes ještě jednou podíváme poněkud jinak.

Seznamy

```
>>> cisla = [1,2,3,4,5]
>>> type(cisla)
list
>>> cisla[0] # V Pythonu číslujeme od 0
```

```
1
>>> cisla[4] # takže poslední prvek je počet prvků - 1
5
>>> len(cisla) # počet prvků je len
5
>>> cisla[-1] # Indexování je velmi flexibilní
5
>>> cisla[1:3]
[2, 3]
>>> cisla[0:5]
[1, 2, 3, 4, 5]
>>> cisla[:3]
[1, 2, 3]
>>> cisla[3:]
[4, 5]
>>> cisla.append(6) # Přidání nového prvku do seznamu
>>> cisla
[1, 2, 3, 4, 5, 6]
```

Seznamy můžou obsahovat různé věci:

```
zaci = ["Honza", "Jakub", "Franta"]
matice = [[1,2,3],[2,3,1]] # Neužitečná implementace matice
matice[0]
>>> [1,2,3]
matice[1][1]
>>> 3
>>> [1,2] + [3,4] # aritmetika pro seznamy
[1, 2, 3, 4]
>>> [1,2]*3
[1, 2, 1, 2, 1, 2]
```

... ale také položky různého druhu:

```
>>> lst = [1,"Peter",True]
>>> lst
[1, 'Peter', True]
>>> del lst[0]
>>> lst
['Peter', True]
>>>
```

Pozor na kopírování seznamů:

```
>>> a = ['jeden', 'dva', 'tri']
>>> b = a
>>> a[0] = 'jedna'
>>> b
['jedna', 'dva', 'tri']
>>> c = [[1,2]]*3
>>> c
[[1, 2], [1, 2], [1, 2]]
>>> c[0][0] = 0
>>> c
[[0, 2], [0, 2], [0, 2]]
```

Seznam umíme lehce setřídit nebo obrátit:

```
>>> c = [2,4,1,3]
>>> sorted(c)
[1,2,3,4]
>>> reversed[c]
[3,1,4,2]
```

O třídění budeme mluvit na následujícím cvičení.

Cyklus `for`

```
In [9]: cisla = [1,1,2,3,5,8]

In [10]: for cislo in cisla:
    ...:     print(cislo, end = "-")
    ...:
1-1-2-3-5-8-
```

Cyklus for je *stručný* - na rozdíl od *while* nepotřebujeme inicializovat logickou podmínku ani inkrementovat či jinak měnit proměnné v cyklu.

Často chceme, aby cyklus probíhal přes jednoduchou číselnou řadu, např. $1, 2, \ldots, n$. Na generování takovýchto řad slouží funkce `range`:

```
>>> for i in range(5):
    print(i, end = ' ')

0 1 2 3 4
```

```
seznam = [1, 2, 3, 4]
slovo = ["P", "y", "t", "h", "o", "n"]
list = [i for i in range(10)]
```

Poslední příkaz: *list comprehension* - umožňuje vytvořit seznam z jiného seznamu.

Přístup k položkám a řezy (slices) seznamů

- Index vrací položku
- Řez (slice) vrací seznam

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

slovo[1] = "y" # prvek
slovo[1:2] = ["y"] # seznam
```

```
>>> 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]
```

Výrazy s indexem i řezy můžou také fungovat jako l-values - tedy jim můžeme něco přiřadit:

```
>>> s
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> s[9] = 8
>>> s
[0, 1, 2, 3, 4, 5, 6, 7, 8, 8]
>>> s[9:10] = [9]
>>> s
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> s[9:10] = []
>>> s
[0, 1, 2, 3, 4, 5, 6, 7, 8]
>>> s[2:4] = []
>>> s
[0, 1, 4, 5, 6, 7, 8]
>>> s[2:2] = [2,3]
>>> s
[0, 1, 2, , 4, 5, 6, 7, 8]
```

Metody seznamů

- `list.append`(*x*)

  Přidává položku na konec seznamu. Ekvivalent `a[len(a):] = [x]`.

- `list.extend`(*iterable*)

  Rozšíří seznam připojením všech prvků `iterable` na konec seznamu. Ekvivalent `a[len(a):] = iterable`.

- `list.insert`(*i*, *x*)

Vloží položku na danou pozici. První argument je index prvku, před který se má vkládat, takže `a.insert(0, x)` vkládá na začátek seznamu a `a.insert(len(a), x)` je ekvivalentní `a.append(x)`.

- `list.remove`(*x*)

  Odstraní ze seznamu první položku s hodnotou x. Vyvolá `ValueError` pokud se taková položka v seznamu nenajde.

- `list.pop`([*i*])

  Odstraní položku na zadané pozici v seznamu a vrátí tuto položku. Pokud index není zadán, `a.pop()` odstraní a vrátí poslední hodnotu v seznamu.

- `list.clear`()

  .Odstraní všechny položky ze seznamu. Ekvivalent: `del a[:]`.

- `list.index`(*x*[, *zacatek*[, *konec*]])

  Vrátí index (počíítaný od 0) v seznamu, kde se nachází první položka s hodnotou rovnou *x*. Pokud taková hodnota v seznamu neexistuje, vyvolá `ValueError` . Volitelné argumenty *zacatek* a *konec* se interpretují jako v notaci řezů a používají se k omezení hledání na určitou oblast seznamu. Výsledný index vrácený funkcí se ale vždy počítá vzhledem k začátku seznamu a ne k poloze *zacatek* .

- `list.count`(*x*)

  Určí, kolikrát se *x* nachází v seznamu.

- `list.sort`(, *klíč=None*, *reverse=False*)

  Utřídí položky seznamu *na místě*. Argumenty můžou být použity na upřesnění požadovaného třídení.

- `list.reverse()`

  Na místě obrátí pořadí prvků v seznamu.

- `list.copy`()`

  Vrací plytkou kopii seznamu. Ekvivalent `a[:]`.

Logický operátor `in`

Zjišťuje, zda se v iterovatelném objektu nachází daná hodnota.

```
# Python program to illustrate
# 'in' operator
x = 24
y = 20
list = [10, 20, 30, 40, 50 ];

if ( y in list ):
   print("y is present in given list")
else:
   print("y is NOT present in given list")

if ( x not in list ):
   print("x is NOT present in given list")
else:
   print("x is  present in given list")
```

Pozor To, že `in` je krátké slovíčko neznamená, že hledání, zda se nějaký prvek nachází v seznamu, je nějak zvlášť efektivní. Není, seznam se prohledá položku po položce. Pokud chcete kolekci s opravdu rychlým vyhledáváním, použijte množinu nebo slovník.

Binární vyhledáváni a třídění

V této části nezavedeme žádnou novou část jazyka, ale budeme cvičit práci se seznamy na dvou důležitých příkladech. Obě funkce jsou součástí API seznamů, my se je pokusíme naivně implementovat, abychom si procvičili programovací svaly.

- **Binární vyhledávání**
- **Třídění seznamů**
- **Ještě medián**

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)

```
#!/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

```
# 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

```
# 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)
```

Ještě medián

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

  `seznam.sort`

  anebo

  `seznam = sorted(seznam)`

- Je ale potřebné seznam setřrídit? O hodnotě mediánu přece nerozhoduje pořadí největších a nejmenších hodnot, takže bychom potenicálně 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 tři podseznamy, *menší* , “pivoty” a "větší", s hodnotami menšími, rovnými, resp. většími nežli pivot.
2. Podle toho, do kterého seznamu připadne hledaný index k, skončíme anebo 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 optimálné, ale funguje to docela dobře.

```
from random import randint

seznam = [randint(1,100) for _ in range(10)]
# while (hodnota := int(input())) != -1:
#     seznam.append(hodnota)

k = int(input())
print(f"{k=}")

assert(0 <= k - 1 < len(seznam))

low = 0
high = len(seznam)
while high - low > 1:
    pivot = seznam[randint(low, high - 1)]
    print(f"{low=} {high=} {pivot=} {seznam=}")
    low_numbers = [x for x in seznam[low:high] if x < pivot]
    pivots = [x for x in seznam[low:high] if x == pivot]
    high_numbers = [x for x in seznam[low:high] if x > pivot]
    seznam = seznam[:low] + low_numbers + pivots + high_numbers + seznam[high:]
    mid1 = low + len(low_numbers)
    mid2 = mid1 + len(pivots)
    if k-1 < mid1:
        high = mid1
    elif k-1 >= mid2:
        low = mid2
    else:
        low = mid1
        high = mid2
        break

print(seznam)
print(seznam[low:high][0])

--------- Výstup ------------
3
```

```
k=3
low=0 high=10 pivot=6 seznam=[7, 6, 5, 6, 7, 7, 4, 3, 3, 5]
low=0 high=5 pivot=5 seznam=[5, 4, 3, 3, 5, 6, 6, 7, 7, 7]
low=0 high=3 pivot=3 seznam=[4, 3, 3, 5, 5, 6, 6, 7, 7, 7]
[3, 3, 4, 5, 5, 6, 6, 7, 7, 7]
4

Process finished with exit code 0
```

Proč má toto náročnost O(n)? Když zanedbáme délku seznamu "pivoty", máme

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

Při náhodném výběru pivotu má algoritmus tuto náročnost pouze v průměru. Pro deterministický algoritmus potřebujeme nějak inteligentněji zvolit pivot. Typicky se pro tento účel používá medián mediánů: seskupíme data do skupin po pěti, a z každé vypočteme medián. Z těchto mediánů vypočteme medián, který slouží jako pivot, i když můžeme hierarchicky postupovat se seskupováním do pětic.

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 nám to zabraňuje psát generický kód.

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.

```
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.

Domácí úkoly

1. Najít v posloupnosti dat nejdelší *souvislou* rostoucí podposloupnost
2. Najít dělící bod posloupnosti
3. Pyramidová výplň