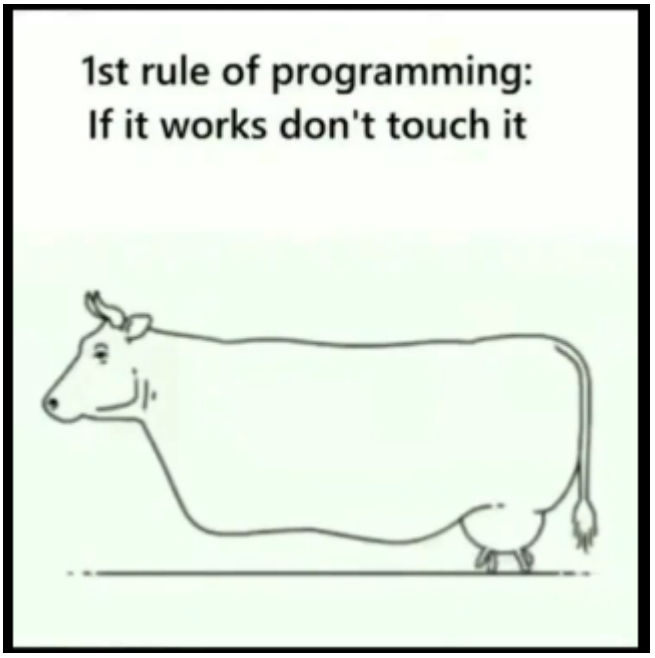
`Programování` 1 pro matematiky

5. cvičení, 28-10-2025

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 úkol** byl tento týden pouze jeden a byl lehký. Tento týden zadám dva domácí úkoly. (Naopak studijní text je podstatně kratší.)
3. **Příští týden** se už uvidíme na cvičení.



- 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. *Toto se musíte naučit i v době AI.*

Drobnosti

Ještě kopírování seznamů

Ukazovali jsme si, že kopírování seznamů je záludné:

```
>>> a = ['jeden', 'dva', 'tri']
>>> b = a   # nový ukazatel na stejný seznam
>>> a[0] = 'jedna'
>>> b
['jedna', 'dva', 'tri']
>>> c = [[1,2]]*3   # ukazatel na seznam [1, 2] se třikrát nakopíruje do c
>>> c
[[1, 2], [1, 2], [1, 2]]
>>> c[0][0] = 0
>>> c
[[0, 2], [0, 2], [0, 2]]
```

Pomáhá představit si, že seznam je reprezentovaný ukazatelem (pointrem) na začátek seznamu, a to, co se předává při přiřazení, je právě jenom tento ukazatel. Tedy při přiřazení `b = a` nedostáváme nový seznam, ale jenom druhý ukazatel na stejný seznam.

Pokud chceme vytvořit kopii seznamu, musíme to Pythonu říci:

```
>>> a = ['jeden', 'dva', 'tri']
>>> b = a.copy()    # kopie seznamu
>>> a[0] = 'jedna'
>>> b
['jeden', 'dva', 'tri']
>>> c = [[1,2] for _ in range(3)]   # v každé iteraci se vytvoří nový seznam
>>> c
[[1, 2], [1, 2], [1, 2]]
>>> c[0][0] = 0
>>> c
[[0, 2], [1, 2], [1, 2]]
```

Pokud máme vnořené seznamy, nepomůže ani `list.copy()`:

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

Řešení: `deepcopy` - rekurzivní kopírování seznamu a všech vnořených podseznamů:

```
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.

Programujeme

V ReCodExu najdete hromadu úloh typu

> Ze standardního vstupu načtěte posloupnost celých čísel. Každé číslo se nachází na novém řádku a posloupnost je ukončena číslem -1, které do posloupnosti nepatří. Vypočtěte a vypište na standardní výstup XYZ.

Tedy vstupná data pro úlohu budou vypadat nějak takto:

```
2
3
5
8
-1
```

Jeden (úplně legální) způsob, jak takováto data načíst, je jednoduché použití cyklu `while`:

```
seznam = []
n = int(input())    # první číslo musíme načíst mimo cyklu
while n != -1:
    seznam.append(n)
    n = int(input())

... (uděláme něco se seznamem) ...
```

Všimněte si toku logiky:

- první číslo načítáme mimo cyklu
- nové číslo načítáme na konci iterace

Je to způsobené tím, že nemůžeme číslo načíst v hlavičce cyklu `while`, protože bychom přišli o načtenou hodnotu, protože v Pythonu (na rozdíl třeba od C / C++) nemá přiřazení hodnotu.

Jiná varianta načítání proto přesouvá testování dovnitř cyklu. Tak dostaneme jednodušší logický tok:

```
seznam = []
while TRUE:
    n = int(input())
    if n = -1:
        break
    seznam.append(n)

... (uděláme něco se seznamem) ...
```

Tady alespoň všechno probíhá v logickém pořadí.

Mroží operátor

V Pythonu existuje speciální operátor přiřazení, který má hodnotu - mroží (walrus) operátor `:=`. Tento operátor má úzkou oblast použití - umožňuje "ukrást" hodnotu například z logických výrazů, např. z testu v hlavičce `while`:

```
seznam = []
while (n := int(input()) != -1:
    seznam.append(n)

... (uděláme něco se seznamem) ...
```

V běžném kódu nemá mroží operátor široké použití, protože například namisto `a = (b := 5)` můžete v Pythonu klidně napsat `a = b = 5`. Ale tady nám posloužil hezky, protože takto vypadá kód přímočaře a kompaktně.

sys.stdin

Můžete samozřejmě načíst data pomocí `sys.stdin`, ale tady musíme ohlídat situaci, že by za řádkem s -1 následoval prázdný řádek. To by vyvolalo výjimku při konverzi na `int`. Taková data přitom nejsou v doslovném rozporu se zadáním.

```
from sys import stdin

seznam = [int(line) for line in stdin.readlines() if line]
seznam.pop()    # odstraníme -1

... (uděláme něco se seznamem) ...
```

Můžete použít libovolný z těchto způsobů načítání. Co nesmíte udělat je toto:

```
seznam = []
while TRUE:
    n = input()
    if n = "-1":
        break
    seznam.append(int(n))

... (uděláme něco se seznamem) ...
```

I když takovýto kód vypadá správně, obsahuje závažnou chybu:

> ⚠ **Warning**
>
> Nikdy se nesmíte spoléhat na to, že na načteném řádku bude právě očekávaný řetězec. Musíte počítat s dalšími "bílými" znaky - mezerou, znakem nového řádku a pod. Kromě toho to, co `input()` skutečně načte se liší podle toho, jestli kód spouštíte na svém laptopu nebo v ReCodExu.

Konverze na `int` odstraní případné přebytečné znaky a předchodzí kódy proto nemají problém. Znaky můžete odstranit také a pak také tento poslední kód bude fungovat bez problémů:

```
seznam = []
while TRUE:
    n = input().strip()
    if n = "-1":
        break
    seznam.append(int(n))

... (uděláme něco se seznamem) ...
```

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. Pro tyto úlohy máme metody, které využívají hladkosti funkce kolem extrému nebo kořene, např. Newtonovu metodu. Takovéto metody bývají ale často méně robustní než bisekce.

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. **Medián** - nalézt medián posloupnosti celých čísel
2. **Najít dělící bod posloupnosti** - tedy takový index v posloupnosti, že všechny hodnoty nalevo jsou menší a napravo větší než hodnota na tomto indexu.