`Programování` 1 pro matematiky

4. cvičení, 22-10-2024

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** Dostali jste zatím 6 úkolů k prvním cvičením.

   Zatím dostávám velký počet převážně slušných řešení.
3. **Distanční cvičení** Tento a příští týden nemáme výuku. Prosím, přečtěte si tento materiál a vyřešte domácí úkol.

Kvíz

Co se vypíše?

```
>>> x = y = [3]
>>> x += y
>>> print(x,y)
>>> ???
```

1. Chyba
2. [3, 3], [3]
3. [3, 3], [3, 3]
4. Něco jiného.

To samé pro řetězce:

```
>>> x = y = "Python"
>>> x += " rocks!"
>>> print(x, y)
>>> ???
```

Co se vypíše?

Řetězce sice mají některé prvky API podobné jako seznamy, ale je to fundamentálně odlišný objekt.

Funkce `id(objekt)` vrací adresu objektu, a operátor `is` porovnává adresy dvou objektů.

Domácí úkoly

Několik užitečných postupů:

Negativní podmínky s ukončením

Okrajové případy chceme vyřešit najednou a úplně, abychom se jimi už nemuseli zabývat. To samé platí pro filtrování množin čísel (jako v případu Pythagorejské trojice): chceme co nejdřív odstranit co nejvíc neperspektivních kandidátů.

Možnost 1 V cyklu `for`

Negativní podmínka + `continue`:

```
...
for a in range(1, n):
    for b in range(1, n):
        if math.gcd(a, b) != 1:
            continue
        ...
```

Možnost 2 Ve funkci

Funkci můžeme pohodlně kdykoli opustit pomocí `return`

```
def fib(n: int) -> int:
    if n < 2:
        return 1
    ...

```

Možnost 3 V hlavním skriptu

Základní knihovna Pythonu obsahuje funkci `exit()`, která se ale chová odlišně na různých platformách. Pro spolehlivé ukončení skriptu použijeme raději `sys.exit()`:

```
import sys

n = int(input())
if n == 0:
    print(0)
    sys.exit() -------
...
```

Vstup s více čísly na řádku

Pokud vstup vypadá takto:

```
17 32
```

načte `input()` celý řádek a ten musíme rozdělit, abychom se dostali k jednotlivým číslům.

Standardní postup

1. Použijeme funkci `str.split()`, která rozdělí řetězec podle dělícího znaku. Default je libovolný bílý znak (mezera, tabelátor atd.), takže pokud nemáte jasný jiný záměr, je lepší ponechat `split()` bez argumentů.
2. Pak ještě musíme jednotlivé kousky zkonvertovat na požadovaný typ, např. `int`.

Varianta 1: funkce `map`:

```
x, y = map(int, input().split())
```

`map(fun, col)` aplikuje první argument `fun` (funkci nebo podobný volatelný objekt) na každý prvek druhého seznamu, tedy pro `col[i]` volá `fun(col[i])` a výsledek vloží do výsledného seznamu.

Varianta 2 list comprehension

V Pythonu můžete do hranatých závorek po anglicky napsat, jak chcete vytvořit nový seznam:

```
x, y = [int(s) for s in input().split()]
```

V obou variantách se seznam na levé straně automaticky zkonvertuje na tuple a odpovídající položky se přiřadí položkám tuplu na levé straně. Pokud počet položek neodpovídá, dojde k chybě.

Tisk seznamu

Jak vytisknu seznam `denominators = [1, 3, 15]` ?

Nepohodlná varianta: Smontuju si výsledný řetězec a ten pak vytisknu:

```
output = " ".join(map(str, denominators))
print(output)
```

`"spoj".join(["A", "B", "C"])` dá `"AspojBspojC"` a dělá opak funkce `split`.

Pohodlná varianta: Použiju hvězdičku:

```
print(*denimanators)
```

Hvězdička promění položky seznamu `denominators` na argumenty `print`. Podle potřeby můžeme nastavit argumenty `sep` a `end` ve funkci print. Konverzi položek na `str` zajistí funkce `print`.

Poznámka 1

Můžeme samozřejmě vytisknout i celý seznam denominators, ale pak se kolem něj vytisknou hranaté závorky a nemůžeme výstup upravit pomocí `sep` .

```
>> denominators = [1, 3, 15]
>> print(denominators)
[1, 3, 15]
>> print(*denominators)
1 3 15
>> print(*denominators, sep = ", ")
1, 3, 15

```

Poznámka 2

Dvě hvězdičky `**` promění slovník v seznam pojmenovaných argumentů:

```
>> denominators = [1, 3, 15]
>> print_settings = {"sep": ", ", "end": "!\n"}
>> print(*denominators, **print_settings)
1, 3, 15!
```

Poznámka 3

Funkce `map` je významný nástroj pro urychlení kódu, protože namísto pomalé smyčky `for` "zapouzdří" cyklus do kódu v C, který je velice rychlý.

Funkce

O funkcích v Pythonu budeme později mluvit mnohem podrobněji, ale protože je stejně většina z vás používá, několik poznámek.

Definice funkce:

```
def print(par1, par2, ..., sep = " ", end = "\n", file = None)
```

Argumenty funkce:

```
- nepojmenované (poziční) - hodnota argumentu se přiřazuje na základě pořadí v seznamu nepojmenovaných parametrů
- pojmenované: hodnota argumentu se přiřazuje podle jména
```

Kromě toho může mít argument přiřazenou defaultní hodnotu.

Návratová hodnota funkce

je hodnota výrazu, uvedeného za klíčovým slovem `return`. Pokud se `return` ve funkci nenachází, návratová hodnota je `None`.

Čisté funkce

Čistá funkce vypočte výsledek z hodnot argumentů a vrátí ho bez toho, aby změnila stav nadřazeného jmenného prostoru. Tedy nic netiskne, nemění globální proměnné, nepíše do souboru a pod. Výhoda je, že takovéto funkce lze vyjmout z programu a nezávisle je otestovat, a také je lehčí odhadnout chování kódu.

Ne vždy se můžeme omezit na čisté funkce, někdy potřebujeme tisknout nebo se dívat na globální proměnnou.

Funkce main

Abychom uchránili globální prostor od proménných ve svém skriptu, je dobré skript uzavřít do funkce `main()`:

```
def fun1():
    # Funkce


def fun2():
    # jiná funkce


def main():
    # hlavní kód skriptu


# Kód se vykoná, pouze pokud je aktuální soubor spuštěn samostatně,
# tedy není importovaný jako modul.
if __name__ == "__main__":
    main()
```

Anotace typů

```
def fibn(n: int) -> list[int]:
    """Vrátí seznam prvních n Fibonacciho čísel"""
    ...
```

Anotace nejsou v Pythonu vynucovány, ale poskytují dobrou dokumentaci kódu a různá IDE je dokážou kontrolovat a upozornit na případnou chybu.

Programujeme

Euklidův algoritmus

Základní verze s odečítáním: $x > y : \gcd(x, y) = \gcd(x - y, y)$

```
#!/usr/bin/env python3
# Největší společný dělitel: Euklidův algoritmus s odčítáním

x = int(input())
y = int(input())

while x != y:
    if x > y:
        x -= y
```

```
    else:
        y -= x

print(x)

```

Ladící výpis: Pokud chceme vidět, jak si vedou čísla x a y v cyklu while, přidáme za rádek s `while` příkaz

```
print(f"{x=} {y=}")
```

Pokud je jedno z čísel o hodně menší než druhé, možná budeme opakovaně odečítat, a to nás spomaluje (náročnost algoritmu je lineární v n). Je proto lepší v jednom kroku odečítat kolikrát to jde: *odečítání nahradíme operací modulo*:

```
#!/usr/bin/env python3
# Největší společný dělitel: Euklidův algoritmus s modulem

x = int(input())
y = int(input())

while x > 0 and y > 0:
    if x > y:
        x %= y
    else:
        y %= x

if x > 0:
    print(x)
else:
    print(y)
```

Protože `x % y < y`, po každé operaci modulo víme, jaká je vzájemná velikost x a y. Kód tedy můžeme výrazně zdokonalit:

```
#!/usr/bin/env python3
# Největší společný dělitel: Euklidův algoritmus s pár triky navíc

x = int(input())
y = int(input())

while y > 0:
    x, y = y, x%y

print(x)
```

Tady si všimneme přiřazení `x, y = y, x%y`. Je to dvojí přiřazení, ale nelze jej rozdělit na dvě přiřazení `x=y` a `y=x%y`, protože druhé přiřazení se po prvním změnilo na `y=y%y` a tedy y bude přiřazena 0.

1. Můžeme se ptát, proč to funguje (protože z dvojice na pravé straně se před přiřazením vytvoří neměnná - konstantní dvojice - *tuple* - a ten se při přiřazení "rozbalí" do x a y).

2. Jak byste takovéto přiřazení rozepsali na jednoduchá přiřazení, aby to fungovalo?

Toto je už celkem výkonný algoritmus, početní náročnost je $\sim \log n$ Teď můžeme dělat víc věcí, například spočíst Eulerovu funkci pro prvních milión čísel a podobně.

Fibonacciova čísla

$$
\begin{aligned}
Fib(0) &= 1, \\
Fib(1) &= 1, \\
Fib(n) &= Fib(n-1) + Fib(n-2), \quad = 2, 3, \dots
\end{aligned}
$$

Úloha: Vypište prvních n Fibonacciových čísel.

Úvahy:

- potřebujeme vůbec seznam?

```
# Vypsat prvnich n Fibonacciových cisel
n = int(input())
a = 1   # 1. číslo
print(a, end = ", ")
b = 1   # 2. číslo
print(b, end = ", ")
for k in range(3,n+1):  # 3. a další až po n
    b, a = b+a, b
    print(b, end = ", ")
```

- Můžeme začít seznamem prvních dvou čísel, a pak dopočítávat a přidávat na konec další čísla:

```
# Vypsat prvnich n Fibonacciových cisel
n = int(input())
fibs = [1,1]
while len(fibs)<n:
    fibs.append(fibs[-1] + fibs[-2])
print(fibs)
```

- Pokud možno, nechcete alokovat paměť po malých kouscích. Když předem víme délku seznamu, je nejlepší zřídit ho celý a jenom ho naplnit.
- Pokud předem nevíme, kolik paměti alokovat, nebo chceme přidávat/odebírat hodnoty na obou koncích pole, je lepší použít `collections.deque` (double-ended queue = dvouhlavá fronta).

Příklad: Erastothénovo síto

Úloha: Najděte všechna prvočísla menší nebo rovná n.

Algoritmus: Napíšeme si čísla od 2 do n. Začneme s 2. Je to prvočíslo, takže si ho zapamatujeme a pak ze seznamu vyškrtneme všechny násobky 2. Následující číslo je 3, zase prvočíslo, takže si ho zapamatujeme a ze seznamu vyškrtneme všechny násobky 3. Takto postupujeme, dokud seznam čísel nevyprázdníme.

Úvahy:

- Musíme si nějak pamatovat, která čísla jsme už vyšktrtli a která nám ještě zůstala.

- Jedno řešení je, že vezmeme seznam všech čísel od 2 do n a budeme odstraňovat ty, které nejsou prvočísly. To je ale pomalé a v proměnlivém poli se špatně iteruje - není snadné určit, kde právě v poli jsme.
- Lepší je vzít seznam logických hodnot. Index bude číslo, a hodnota bude označovat, jestli jej zatím považujeme za prvočíslo anebo už ne.

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
T	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F
T	**T**	F	T	F	T	F	F	F	T	F	T	F	F	F	T	F	T	F
T	**T**	F	**T**	F	T	F	F	F	T	F	T	F	F	F	T	F	T	F
...																		

```python
# vypiš všechna prvočísla menší nebo rovná n

n = int(input())

prvocisla = [True] * (n + 1) # včetně nuly, jedničky a n
prvocisla[0] = False
prvocisla[1] = False
for i in range(2, n + 1):
    if prvocisla[i]:
        for j in range(i * i, n + 1, i):
            prvocisla[j] = False

print("Pocet: ", sum(prvocisla))
for i in range(n + 1):
    if prvocisla[i]:
        print(i, end = ', ')
```

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]] # Hierarchický seznam.
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]
>>>
```

> ⚠ **Warning**
>
> 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")
```

> ⚠ **Warning**
>
> **Pozor** To, že `in` je krátké slovíčko neznamená, že hledání, zda se nějaký prvek nachází v seznamu, je nějak zvlášt 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**

Vyhledávání v setříděném seznamu

Proč třídíme seznamy? Protože v setříděném seznamu dokážeme vyhledávat fundamentálně rychleji než v nesetříděném.

- Abychom našli prvek v nesetříděném seznamu délky n, musíme prohlédnout řádově `n` prvků: nemůžeme dělat nic jiného než prohlížet prvky seznamu jeden po druhém, až najdeme co hledáme.
- Abychom našli prvek v setříděném seznamu délky n, stačí prohlédnout řádově `log2 n` prvků.

Mimochodem, jak zjistíme, že je seznam setříděný? Kolik prvků přitom potřebujeme vidět?

> **Important**
>
> Generické metody seznamu `index` a `count` a operátor `in` používají postup vyhledávání pro nesetříděná data a jsou tedy pro velké seznamy pomalé.

Algoritmus: Půlení intervalu (proto *binární*).

Náročnost: O(log n)

> **Tip**
>
> Pro nalezení požadovaného prvku (nebo alternativně, pro zjištění, že se v seznamu nenalézá) v setříděném seznamu s `n = 1 000 000` prvky potřebujeme vídět řádově 20 prvků.

```
#!/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.")

```

Třídění

O třídění budeme ještě obsáhle mluvit a ukážeme si podstatně efektivnější algoritmy než následující dva.

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)

```

Domácí úkol

Pascalův trojúhelník - pro dané n máte vypsat seznam seznamů, obsahující prvních n řádek Pascalova trojúhelníku.