`Programování` 1 pro matematiky

3. cvičení, 14-10-2024

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 2 příklady a nashromáždilo se docela hodně řesení.
    - Individuálně se vyskytly problémy s ReCodExem, zejména s pochopením toho, že vše, co napíšete na výstup, považuje ReCodEx za součást vašeho řešení.
        - Důkladně čtěte zadání
        - Napište na výstup jenom odpověď v požadovaném tvaru a nic víc.
    - Vyskytla se část příliš defenzivních řešení.
        - Pořádně si přečtěte zadání a neřešte co se stane, když se na vstupu objeví něco jiného než co uvádí zadání.
        - A když už to řešíte, alespoň to řešte správně.
    - Vyskytlo se docela dost úloh, používajících pokročilejší struktury Pythonu, např. funkce.
        - v principu nenamítám, ale myslete prosím na spolužáky, kteří stejný úkkol vyřeší za pomoci jednodušších prostředků. Je váš kód natolik elegantní, že to omlouvá používání těžkých zbraní?

    K domácím úkolům a ReCodExu se ještě obširněji vrátím.
3. **V následujících dvou týdnech nám odpadá výuka.**. Abychom ztrátu částečně nahradili, dostanete v každém týdnu jeden domácí úkol a krátké poučení o něčem novém v Pythonu, které vám s úkolem pomůže.

Opakování

... a co se posledně nevešlo:

`if-elif-else`:

```
if podmínka1:
   příkazy1
elif podmínka2:
   příkazy2
else:
   příkazy3
```

Prázdný příkaz `pass`

Tento příkaz používáme tam, kde potřebujeme mít příkaz, ale vlastně nechceme nic udělat. Často například v `if`, když se podmínka píše jednodušeji než její negace:

```
if podmínka:
    pass
else:
    příkazy
```

Příkaz `match case`

```
match term:
    case pattern-1:
         action-1
    case pattern-2:
         action-2
    case pattern-3:
         action-3
    case _:
        action-default
```

Tento příkaz nám umožňuje nahradit strukturu `if-elif-elif-...-else` v případech, kdy vybíráme z většího množství voleb.

Podstatná výhoda `match case` je, že umí přiřazovat složitější vzory:

```
point = (1, 2)

match point:
    case (0, 0):
        result = "Origin"
    case (x, 0):
        result = f"X-axis at {x}"
    case (0, y):
        result = f"Y-axis at {y}"
    case (x, y):
        result = f"Point at {x}, {y}"
```

V Pythonu existují ještě jiné způsoby implementace mnohonásobného větvení, např. pomocí slovníku - to je velice typický Pythonský postup:

```
messages = {
    "JavaScript":"You can become a web developer.",     "Python": "You can become a Data Scientist",
    "PHP": "You can become a backend developer"
    ...
}
lang = input("What's the programming language you want to learn? ")
print(messages[lang])
```

Cyklus `while`:

```
while podmínka:
   příkazy
```

Příkazy pro kontrolu běhu cyklu:

`break` - v tomto místě opustit cyklus a pokračovat příkazem, následujícím za cyklem

`continue` - v tomto místě přejít na další iteraci cyklu (tedy na testování podmínky)

Větev `else`:

```
while podmínka:
   příkazy
else
   příkazy-jen-jestli-proběhl-celý-cyklus
```

Příkaz print

```
print(a, b, c, ..., sep=<řetězec>, end=<řetězec>)
```

Default: `sep = " "`, `end = "\n"`

f-řetězce

```
a = 5
b = 10.27
s = f"Mám {a} tyčí s délkou {b:.1f}m"
print(s)
---------
Mám 5 tyčí s délkou 10.3m
```

Domácí úkoly

Prostředky Pythonu

Hodně z vás zná z Pythonu víc, než jsme zatím probrali. Přijímám i řešení, která používají prostředky jazyka, které jsme zatím neprobírali,

- pokud to není v rozporu s účelem zadání - tedy když máte sami naprogramovat něco, na co existuje v Pythonu knihovní funkce - například budeme programovat GCD(a,b), zatímco máme funkci `math.gcd()`, která to udělá za vás, nebo když máte zjistit počet číslic celého čísla bez použití zkratky `len(str(n))`.
- pokud to vede k čistšímu a efektivnímu kódu

Cílem tohoto cvičení je naučit vás psát nejen správný, ale i čistý a dobře čitelný kód. Toho dosáhnete i účelným využíváním rozmanitých prostředků jazyka.

Programujeme - pokračování od minula

Test prvočísel

Chceme otestovat, zda je číslo n ze vstupu prvočíslo.

Metoda: U všech čísel $d < n$ prověřím, zda jsou děliteli n.

```
# Otestuje, zda číslo je prvočíslem

n = int(input())
d = 2
mam_delitele = False

while d < n:
    if n%d == 0:
        print("Číslo", n, "je dělitelné", d)
        mam_delitele = True
        break
    d += 1

if not mam_delitele:
    print("Číslo", n, "je prvočíslo")
```

To není nijak zvlášť efektivní metoda, ale to nám nevadí, my jsme celí rádi, že umíme napsat něco, co v zásadě funguje.

Pojďme opatrně vylepšovat. Zásadní vylepšení kódu by bylo, kdybychom "nahý" cyklus `while` uměli celý zapouzdřit do jediného příkazu.

😎 *Pokročilé kolegy poprosím o tvar onoho jediného příkazu.*

(Co udělá toto? `all(n % d for d in range(2, n))`)

Asi první věc, která nám vadí, je stavová proměnná `mam_delitele`. A té se v prvním kroku zbavíme za použití větve `else`:

```
# Otestuje, zda číslo je prvočíslem (2. pokus)

n = int(input())
d = 2

while d < n:
    if n%d == 0:
        print("Číslo", n, "je dělitelné", d)
        break
    d += 1
else:
    print("Číslo", n, "je prvočíslo")
```

Jak bychom mohli dál vylepšit náš test?

Popřemýšlíme, a zatím vymyslíme, jak bychom vypsali všechna provčísla menší nebo rovná n. Nejjednodušší metoda bude projít všechna čísla od 2 do n, u každého rozhodnout, zda je prvočíslem, a jestli ano, vypsat ho.

```
# Vypíše všechna prvočísla od 1 do n

n = int(input())

x = 2
while x <= n:       # vnější cyklus: čísla od 2 do n
    d = 2
    while d < x-1:  # x určitě dělí x, nemusíme testovat
        if x%d == 0:
            break
        d += 1
    else:
        print(x)
    x += 1
```

Optimalizace je v tomto případě ještě více nasnadě, jenomže si zatím neumíme pamatovat věci - například všechna prvočísla, které jsme dosud nalezli.

🤓 *Pokročilé kolegy* poprosím o optimalizovaný algoritmus, např. Erastothenovo síto.

My ale začneme s jinou věcí, a to je *organizace kódu*. Požadavek zjistit, zda je nějaké číslo prvočíslem, je dobře izolovatelný: potřebujeme jenom ono číslo a už nepotřebujeme znát nic jiného. Proto můžeme kód, zjišťující, zda je číslo prvočíslem, izolovat do *funkce*.

```
def is_prime(n):
    d = 2
    while d < n:
        if n%d == 0:
            return False
        d += 1
    return True

n = int(input())
if is_prime(n):
    print(f"{n} je prvočíslo")
else:
    print(f"{n} není prvočíslo")
```

Funkce v Pythonu:

```
def jméno_funkce(parametr1, parametr2, ...):
    příkazy
```

Návratová hodnota funkce: `return výraz`.

Později budeme o funkcích mluvit podrobněji, a že bude o čem mluvit, prozatím nám stačí tato jednoduchá koncepce.

Součet posloupnosti čísel

```
# Načteme ze vstupu posloupnost čísel, ukončenou -1.
# Vypíšeme jejich součet.

s = 0
while True:
    n = int(input())
    if n == -1:
        break
    s += n
print(s)
```

Proč nemůžeme na konci jenom stisknout Enter a nezadat nic?

😎 *Pokročilé kolegy poprosím*

- o variantu se stiskem Enter (`sys.stdin`)
- a o vypsání aritmetického průměru a standardní odchylky.

Mroží operátor

```
>>> a = (b := 10)**2
>>> a
100
>>> b
10
>>>
```

Přiřazení, které vrací hodnotu. *Používat uměřeně.*

Euklidův algoritmus

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

```
# 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:

```
# 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.

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[::-1]
[5, 4, 3, 2, 1]
>>> cisla.append(6) # Přidání nového prvku do seznamu
>>> cisla
```

```
[1, 2, 3, 4, 5, 6]
>>> cisla.pop()
6
>>> cisla
[1, 2, 3, 4, 5]
```

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)     # nevznikne nový seznam
<list_reverseiterator at 0x25cdadbffa0>
>>> list(reversed(c))
[4, 3, 2, 1]
>>> c.reverse()     # vytvoří nový seznam
[4, 3, 2, 1]
```

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-
```

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
```

`range` respektuje číslovací konvence Pythonu a podporuje ještě další argumenty: začátek sekvence a krok:

```
>>> for i in range(2,10,3):
...     print(i)
...
2
5
8
```

Argumenty nemusí být kladná čísla, takže můžeme lehko iterovat pozpátku:

```
>>> for i in range(9,-1,-1):
---    print(i, end = ' ')
---
9 8 7 6 5 4 3 2 1 0 >>>
```

Tedy chceme-li iterovat pozpátku přes `range(n)`, použijeme `range(n-1,-1, -1)`.Abychom se nespletli, nabízí Python elegantnější řešení:

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

4 3 2 1 0
```

a tuto funkci můžeme použít na libovolný seznam:

```
>>> cisla = ["jedna", "dvě", "tři", "čtyři"]
>>> for cislo in reversed(cisla):
    print(cislo, end = " ")

čtyři tři dvě jedna
```

`range(n)` není seznam, i když podporuje přístup k polžkám přes index:

```
>>> range(10)
range(0, 10)
>>> type(range(10))
<class 'range'>
>>> range(10)[-1]
9
>>> list(range(10))
[0,1,2,3,4,5,6,7,8,9]
>>> [i for i in range(10)]
[0,1,2,3,4,5,6,7,8,9]
```

Také `reversed/seznam)` není seznam, ale reverzní iterátor:

```
>>> reversed(cisla)
<list_reverseiterator object at 0x000002AE55D45370>
>>> [s for s in reversed(cisla)]
['čtyři', 'tři', 'dvě', 'jedna']
```

ale `sorted(seznam)` je seznam:

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

Fibonacciho čísla

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

Úloha: Vypište prvních n Fibonacciho čísel.

Úvahy:

- potřebujeme vůbec seznam?

```
# Vypsat prvnich n Fibonacciho cisel
n = int(input())
a = 1
print(a, end = ", ")
b = 1
print(b, end = ", ")
for k in range(3,n+1):
    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 Fibonacciho cisel
n = int(input())
fibs = [1,1]
while len(fibs)<n:
    fibs.append(fibs[-1] + fibs[-2])
print(fibs)
```

- Nikdy nechcete alokovat paměť po malých kouscích. Když předem víme délku seznamu, je nejlepší zřídit ho celý a jenom ho naplnit.

Příklad: Erastothénovo síto

Úloha: Najděte všechna prvočísla menší nebo rovná n.

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.

```
# vypiš všechna prvočísla menší nebo rovná n

n = int(input())

prvocisla = [True]*(n+1) # včetně nuly 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 = ', ')
```

Další úkoly

- Najděte číslo zapsané samými jedničkami (v desítkové soustavě), které je dělitelné zadaným K. Jak se včas zastavit, když neexistuje?
- Najděte číslo mezi 1 a N s co nejvíce děliteli.

Domácí úkoly na příští týden:

- Na vstupu kladné celé n. Vypsat počet Pytagorejských trojic s čísly menšími než n. U této úlohy prohledáváte prostor trojic přirozených čísel a je užitečné zvolit promyšleně pořadí filtrů.
- Převést zlomek v tvaru $\frac{x}{y}$ na tvar $\frac{1}{a} + \frac{1}{b} + \frac{1}{c} + \ldots$ (Fibonacciův algoritmus)

U obou úloh je rozumné posadit se na chvíli s tužkou a papírem, než začnete psát kód.

Pomůcka 1

```
>>> "1 2 3".split()
["1", "2", "3"]
>>> [int(s) for s in "1 2 3".split()]
[1, 2, 3]
>>>
```

Pomůcka 2

Náhodná čísla pro testování domácího úkolu:

```
from random import randint

low = 0
high = 10
n = 10

print([randint(low, high) for i in range(10)])
```