Programování 2

2. cvičení, 24-2-2026

Farní oznamy

1. Tento text a kódy ke cvičení najdete v repozitáři cvičení na https://github.com/PKvasnick/Programovani-2.
2. **Domácí úkoly**:
    - Na první várku domácích úkolů přišlo docela dost řešení
    - Byly to méně snadné úkoly, ale poradili jste si dobře.
    - O nejčastějších problémech promluvíme podrobněji

Obsah:

- Kvíz
- Domácí úkoly:
    - Načítání posloupností
    - Kdy lze posloupnost zpracovat průběžně a jak to udělat?
- Resty z minula: sečítání čísel po číslicích

Na zahřátí

> Pokud to funguje, nedotýkej se toho.

Dobrá, nebo špatná rada?

- nepromyšlené zásahy do kódu mohou mít nečekané důsledky
- to neznamená, že bychom se neměli snažit vylepšit ošklivý kód nebo doplnit nové funkce
- srozumitelný kód se modifikuje mnohem lépe

Co dělá tento kód

```
jmena = {"Jana", "Pavel", "Pepa", "Franta"}
dalsi = {"Eva", "Pepa", "Katka", "Standa"}
jmena & dalsi
????
jmena and dalsi
jmena - dalsi
```

Seřaďte podle data narození

```
lide = [
    ("Jana", "Nováková", 1964),
    ("Kateřina", "Kocourová", 1962),
    ("Jozef", "Winkler", 1952),
    ("Petr", "Suchý", 1968),
    ("Jan", "Michal", 1951)
]
```

- lambda
- `operator.itemgetter`

Domácí úkoly

Dvě poznámky.

Načítání z konzole: váš laptop a ReCodEx

Někteří z vás s tímto docela zápasili (a připojili se k dlouhé řadě zápasníků z minulých semestrů):

```
lines = []
while True:
    line = input()
    if line == "---": # Toto funguje na vašem laptopu, ale ne v ReCodExu!
        break
    lines.append(line)
```

- Pokud zadáváte vstup z klávesnice, ukončuje vstup a `input()` vrátí to, co jste napsali před stiskem
- Když ReCodEx vyhodnocuje vaše domácí úkoly, nečte z konzole, ale na jeho vstup jsou přesměrována data ze vstupního textového souboru. V tomto případě `input()` vrátí řádek textu včetně koncového `\n`.

Tedy to, co čte váš program, se odlišuje podle toho, zda program spouštíte na svém notebooku a vstup zadáváte z klávesnice, nebo zda program běží v ReCodExu.

> ⚠ **Warning**
>
> Neměli byste se nikdy spoléhat, že program načte přesně požadovaný řetězec.

Jak jste viděli, načtený řádek může nebo nemusí obsahovat `\n`, nebo se někde může přichomítnout mezera, anebo tam, kde vidíte mezeru, je ve skutečnosti tabulátor anebo jiný "bílý" znak.

Defenzivní načítání:

```
1. možnost
...
line = input().strip()  # Odstraníme případné nadbytečné znaky
if line == "---":
    break
---
2. možnost
line = input()
if "---" in line:       # Ptáme se, zda řádek obsahuje požadovaný řetězec
    break
```

Datové struktury a jejich modifikace

Toto se týká úlohy o sloučení dvou uspořádaných seznamů.

Několik řešení pracovalo tak, že přesypávalo data z jednoho seznamu na správná místa v druhém seznamu pomocí `list.insert`. Takováto řešení mají kvadratickou složitost, protože `list.insert` má lineární složitost O(n): *když vkládáte novou hodnotu, musí se celý seznam v paměti přeuspořádat*. Tím platíte za možnost rychlého přístupu k prvkům přes index.

> **Important**
>
> Pokud chcete rychlý algoritmus, snažte zbytečně neměnit datové struktury.

Pro změnu struktur dat v paměti musí program komunikovat se správou paměti v Pythonu nebo dokonce v systému, a takovéto operace mohou z času na čas vyžadovat nepředvídatelné prodlevy.

Takže:

- `list.insert` jenom v nutných případech (např. když máte záruku, že seznam má omezenou velikost)
- Předem alokovat a používat index namísto `list.append`.
- Namísto odstraňování "spotřebovaných" položek nechte dealokaci seznamu na úklidovou službu (*garbage collector*)

```
a = list(map(int, input().split()))
b = list(map(int, input().split()))

c = [0] * (len(a) + len(b)) # alokuji předem, protože znám velikost

index_a = 0
index_b = 0
index_c = 0 # ukazovátka na aktuální polohu namísto list.append()

...
```

Posloupnosti

1. Načtení posloupnosti z konzole a její zpracování

Načtení dat z řádku

Pokud máme na standardním vstupu data ve tvaru `1 2 3 5 8 ...`, typický způsob načtení je

`a = [int(s) for s in input().split(5)]` (list comprehension), toto dává seznam (`list`).

nebo

`a = map(int, input().split())` toto dává iterátor, pokud potřebujete, můžete ho zkovertovat na seznam, `list(a)`.

Načtená data lze rovnou rozbalit do příslušných proměnných:

`m, n = [int(s) for s in input().strip()]`

Toto je standardní metoda ne proto, že by nějak hezky vypadala, ale proto, že dělá načtení a konverzi v jediném kroku a tedy není potřebné zřizovat dočasná pole.

Načtení dat ze sloupce

Toto jste u domácích úkolů dělali běžně:

> Vstupní posloupnost načtete z konzoly číslo po čísle, každé číslo na novém řádku. Posloupnost je ukončená řádkou s −1, která nepatří do posloupnosti.

Generická verze:

```
def read_from_console() -> list[float]:
    a = []
    while True:
        line = input()
        if "-1" in line:
            break
        a.append(float(line))
    return a
```

Poznámky:

1. Kód `-> list[float]` oznamuje Pythonu, že výstupem funkce je seznam desetinných čísel. Podobně můžeme oznámit i typy parametrů, a analyzátor kódu ve vašem IDE pak zahlásí chybu, pokud někde použijeme nesprávný typ.
2. Logika: Máme nekonečný cyklus a vyskakujeme z něj, pokud narazíme na znak konce posloupnosti.
3. Testujeme `"-1" in line` namísto `line == "-1"`

Ze souboru:

```
def read_from_file(filename: str) -> list[float]:
    a = []
    with open(filename, "r") as infile:
        for lines in infile:
            if "-1" in line:
                break
            a.append(float(line))
    return a
```

Načíst do paměti nebo nenačíst?

U některých úkolů dokážeme čísla zpracovávat postupně a nepotřebujeme mít celou posloupnost uloženou v paměti.

Příklad: : nalezení maximální hodnoty, výpočet průměru a standardní odchylky

V těchto případech má algoritmus formu propagování nějakého vnitřního stavu přes vstupná data: Začneme počátečním stavem (např. nejmenší možnou hodnotou typu `float`, tedy `float('-inf')`) a jak přicházejí data, aktualizujeme ji (např. se u každé vstupní hodnoty ptáme: Je toto největší hodnota, jakou jsem doposud viděl? a pokud je, nastavíme vnitřní stav na tuto hodnotu). Na konci máme výsledný stav (například maximální hodnotu posloupnosti) a umíme dokázat, jaké má vlastnosti (například že je to opravdu maximum posloupnosti).

Příklady:

- maximum
- k-tý největší prvek (k << n)
- střední hodnota a standardní odchylka

Naopak, u jiných úloh potřebujeme mít posloupnost v paměti celou nebo její podstatnou část

Příklady:

- medián
- dělící bod posloupnosti

Načítání pomocí generátoru

Pokud chceme hledat například maximum posloupnosti a nechceme ji celou načítat, musíme kód pro načítání a hledání promíchat. To je nešťastné, pokud chceme pro zpracování posloupnosti použít stejný kód pro načítání ze standardního vstupu nebo souboru.

```
m = float("-inf")
while "-1" not in (line := input()):
    number = float(line)
    if number > m:
        m = number
```

Poznámky:

- Inicializace: `float("-inf")` je nejmenší číslo v plovoucí desetinné čárce. Jaké je nejmenší celé číslo?

- `:=` je “walrus”, čili mroží operátor, pomocí kterého můžeme ukrást hodnotu, kterou načte `input()` do proměnné `line`. To je šikovné a řeší to malý problém s logikou načítacího cyklu: protože v cyklu načítáme, musíme kód zpřeházet, pokud chceme testovat v hlavičce `while` anebo testovat uvnitř cyklu a použít `break`

```
def read_from_console():
    while "-1" not in (line := input()):
        yield float(line)
    return

m = float("-inf")
for number in read_from_console():
    if number > m:
        m = number
print(m)
```

Napište kód, který takto nalezne maximum při načítání posloupnosti ze souboru.

Reduce

Uměli bychom uzavřít otevřený cyklus `while`, resp. `for`, který máme v tomto kódu?

Můžeme použít funkci `functools.reduce`, která dělá přibližně toto:

```
def reduce(function, iterable, initializer=None):
    it = iter(iterable)
    if initializer is None:
        value = next(it)
    else:
        value = initializer
    for element in it:
        value = function(value, element)
    return value
```

Tedy funkce `reduce` propaguje a aktualizuje nějaký stav přes posloupnost. Na rozdíl od `itertools.accumulate` ale nevrací průběžné stavy, ale pouze konečný stav.

```
from functools import reduce

def read_from_console():
    while "-1" not in (line := input()):
        yield float(line)
    return

maximum = reduce(max, read_from_console, float("-inf"))
print(maximum)
```

Takovýto kód bude rychlý, protože cyklus se vykonává uvnitř funkce, a tedy běží v C a ne v Pythonu.

Podobné úlohy

- rozhodnout, zda je posloupnost čísel monotonní a jak (konstantní, rostoucí, neklesající, klesající, nerostoucí)
- v posloupnosti čísel nalézt druhou největší hodnotu a počet jejích výskytů
- v posloupnosti čísel určit délku nejdelšího souvislého rostoucího úseku
- v posloupnosti čísel určit počet různých hodnot
- v posloupnosti čísel nalézt souvislý úsek se součtem K (pro zadanou hodnotu K)
- v posloupnosti kladných čísel nalézt souvislý úsek se součtem K (pro zadanou hodnotu K)
- v posloupnosti čísel nalézt souvislý úsek s maximálním součtem.

Z minula:

Součet dvou čísel

Na vstupu načteme dvě celá čísla jako znakové řetězce. Na výstupu má váš kód vypsat jejich součet, ale máte povolenou sečítat pouze číslice 0-9, ne celá čísla. Můžete si představit, že pro sčítání máte k dispozici tabulku typu

	0	**1**	**...**	**8**	**9**
0	(0, 0)				
1	(1, 0)	(2, 0)			
2	(2, 0)	(3, 0)			
...					
8	(8, 0)	(9, 0)		(6, 1)	

	0	1	...	8	9
9	(9, 0)	(0, 1)		(7, 1)	(8, 1)

kde první číslo jsou jednotky součtu a druhé číslo je "přenos".

Stalo se, že jako řešení jsem ukazoval takovýto kód:

```
from itertools import zip_longest

# Sestrojíme slovník pro sčítání číslic
# Klíče: dvojice číslic (0-9). Hodnoty: dvojice (přenos, součet % 10
add_table = {(i, j) : divmod(i + j, 10) for i in range(10) for j in range(10)}

# Načteme dvě čísla jako seznamy číslic
a = [int(x) for x in input()]
b = [int(x) for x in input()]

result = []
carry = 0
for da, db in zip_longest(reversed(a), reversed(b), fillvalue=0):
    carry, digit_sum = add_table[(da, db)] # <<< Toto je špatně, nepřičítáme přenos!!!
    result.append(digit_sum)
if carry > 0:
    result.append(carry)

a_print = "".join(map(str, a))
b_print = "".join(map(str, b))
result_print = "".join(map(str, reversed(result)))
print(f"{a_print} + {b_print} = {result_print}")
```

Tento kód je špatný a děkuji kolegovi Danu Ransdorfovi za upozornění a dokonce *pull request* s návrhem opravy.

Opravené řešení bude vypadat nějak takto:

```
from itertools import zip_longest

# Sestrojíme slovník pro sčítání číslic
# Klíče: dvojice číslic (0-9). Hodnoty: dvojice (přenos, součet % 10
add_table = {(i, j) : divmod(i + j, 10) for i in range(10) for j in range(10)}

# Načteme dvě čísla jako seznamy číslic
a = [int(x) for x in input()]
b = [int(x) for x in input()]

reversed_result = []    # Dobré jméno (C) Dan Ransdorf
carry = 0
for da, db in zip_longest(reversed(a), reversed(b), fillvalue=0):
    carry_1, da = add_table[(da, carry)]
    carry_2, digit_sum = add_table[(da, db)]
```

```
    _, carry = add_table[(carry_1, carry_2)]
    reversed_result.append(digit_sum)
if carry > 0:
    reversed_result.append(carry)

a_print = "".join(map(str, a))
b_print = "".join(map(str, b))
result_print = "".join(map(str, reversed(reversed_result)))
print(f"{a_print} + {b_print} = {result_print}")
```

Vidíte, že řešení je poněkud překomplikované, pokud trváme na použití sčítací tabulky: Kvůli započtení přenosu nám přibydou další dvě sčítání a kromě toho tento postup není dobře zobecnitelný na součet mnoha čísel.

Naštěstí alespoň kód s funkcí `itertools.accumulate` byl a je správný.

```
from itertools import accumulate, zip_longest


def add_and_carry(accumulator: tuple[int, int], digits: tuple[int, int]) -> tuple[int, int]:
    """
    Přičte součet dvou dalších číslic a přenosu, vrací aktualizovaný součet a přenos.
    :param accumulator: tuple of the running sum and carry
    :param digits: tuple of two new digits to add
    :return: tuple of updated sum and carry
    """
    carry, old_sum = accumulator
    a, b = digits
    new_sum = a + b + carry
    return divmod(new_sum, 10)  # returns (carry, digit_sum)


def sum_by_digits(a: list[int], b: list[int]) -> list[int]:
    """
    Sums two integers given as lists of digits.
    :param a0: list of digits of the first number
    :param b0: list of digits of the second number
    :return: list of digits of the sum
    """
    tuples = list(accumulate(zip_longest(a, b, fillvalue=0), func=add_and_carry, initial=
(0,0)))
    tuples.reverse()
    tuples.pop()    # remove initial state
    carry = tuples[0][0]
    result = [carry] if carry != 0 else []
    result += [v for u, v in tuples]
    return result


def main() -> None:
    """
    Demonstrates the sum_by_digits function.
    """
    a = [int(c) for c in input()]
```

```
    b = [int(c) for c in input()]
    print("First number: ", *a, sep = "")
    print("Second number: ",*b, sep = "")
    digit_sum = sum_by_digits(a, b)
    print("Sum: ", *digit_sum, sep="")
    print("Check:")
    a_int = int("".join([str(d) for d in a]))
    b_int = int("".join([str(d) for d in b]))
    print("Int addition: ", a_int + b_int)


if __name__ == "__main__":
    main()
```

Je toto užitečný kód? Sice jsme ukryli cyklus do `accumulate`, ale drobná práce s obracením pořadí stejně zůstává, protože to je vlastnost sčítání.

Domácí úkoly

Dnes dostanete další úkoly, které volně souvisí s tříděním. U jedné hledáme, jestli je posloupnost částečně setříděná, zatímco druhá je zjednodušenou variantou důležité úlohy z dynamického programování.

1. **Dělící bod posloupnosti** - najít v posloupnosti čísel takový index, že položky vlevo jsou menší a položky vpravo jsou větší než hodnota na tomto indexu.
2. **Minimální a maximální součet** - najít v posloupnosti n čísel k čísel s minimálním a k čísel s maximálním součtem.