

Programování 2

2. cvičení, 23-2-2023

tags: Programování 2, čtvrtek 1, čtvrtek 2

Farní oznamy

1. Tento text a kódy ke cvičení najdete v repozitáři cvičení na <https://github.com/PKvasnick/Programovani-2>.

2. **Domácí úkoly:**

- Prosím zaregistrujte se do své skupiny v ReCodExu, pokud jste tak dosud neučinili
- Na první várku domácích úkolů přišlo docela dost řešení
- Dva nebo tři příklady?

Obsah:

- Kvíz
- Velká a malá písmena: slovníky a množiny
- ChatGPT a domácí úkoly
- Načítání a průběžné zpracování posloupností (maximum, 2 největší prvek a pod.)

Na zahřátí

Pokud to funguje, nedotýkej se toho.



Dobrá, nebo špatná rada?

Co dělá tento kód

```
1 jmena = {"Jana", "Pavel", "Pepa", "Franta"}
2 dalsi = {"Eva", "Pepa", "Katka", "Standa"}
3 jmena & dalsi
4 ???
5 jmena and dalsi
6 jmena - dalsi
```

Seřadte podle data narození

```
1 lide = [
2     ("Jana", "Nováková", 1964),
3     ("Kateřina", "Kocourová", 1962),
4     ("Jozef", "Winkler", 1952),
5     ("Petr", "Suchý", 1968),
6     ("Jan", "Michal", 1951)
7 ]
```

- `operator.itemgetter`

- lambda

O velkých a malých písmenech

```
1  uzivatele = [  
2      "johndoe@gmail.com",  
3      "JANICKA@seznam.cz",  
4      "PEPA@GMAIL.COM",  
5  ]  
6  
7  def is_new_email(address):  
8      for user in users:  
9          if ...:  
10             return False  
11     return True
```

Chceme doplnit `...` tak, aby kód fungoval - tedy např. aby fungovali následující testy:

```
1  assert is_new_email("john@gmail.com")  
2  assert is_new_email("example@gmail.com")  
3  
4  assert not is_new_email("JOHNdoe@gmail.com")  
5  assert not is_new_email("JOHNDOE@GMAIL.COM")  
6  assert not is_new_email("Janicka@Seznam.cz")  
7  assert not is_new_email("pepa@gmail.com")
```

`str.lower` / `str.upper` není dobré řešení

Zřejmé řešení:

```
1  def is_new_email(address):  
2      for user in users:  
3          if user.lower() == address.lower():  
4              return False  
5      return True
```

Nemáme problém pro 26 písmen anglické abecedy:

```
1  >>> import string  
2  >>> string.ascii_uppercase  
3  'ABCDEFGHIJKLMNOPQRSTUVWXYZ'  
4  >>> string.ascii_lowercase  
5  'abcdefghijklmnopqrstuvwxyz'
```

Ale: Německé ostré s, ß:

```

1 >>> "ß".lower()
2 'ß'
3 >>> "ß".upper()
4 'SS'
5 >>> "SS".lower()
6 ???

```

Existuje ještě několik dalších problematických znaků:

```

1 for i in range(65535):
2     c = chr(i)
3     if c.lower() == c and c.upper().lower() != c: print(i, c)
4
5 """
6 Dostaneme znaky jako:
7 181 µ
8 223 ß
9 305 ı
10 329 ħ
11 383 f
12 496 ĵ
13 837 `
14 # a další.
15 """

```

Samozřejmě můžeme omezit množinu znaků povolených v e-mailových adresách, např. na malá a velká písmena ze `string.ascii_lowercase`, ale ne vždy jde řešit problémy omezením možností.

Ze stejných důvodů nejde použít ani `str.upper()`:

```

1 for i in range(65535):
2     c = chr(i)
3     if c.upper() == c and c.lower().upper() != c: print(i, c)
4
5 """
6 Dostaneme znaky jako:
7 304 İ
8 1012 Ø
9 7838 ß
10 8486 Ω
11 8490 Κ
12 8491 Å
13 """

```

`str.casefold()`

dělá pro běžná písmena to, co `str.lower()`, ale chová se regulárně i pro problematická písmena:

```

1 >>> sentence = "THE QUICK brown Fox jumps OVER the LaZy dog."
2 >>> sentence.lower()
3 'the quick brown fox jumps over the lazy dog.'
4 >>> sentence.casefold()
5 'the quick brown fox jumps over the lazy dog.'
6 ---
7 >>> word = "straße"
8 >>> word.lower()
9 'straße'
10 >>> word.casefold()
11 'strasse'
12

```

Takže `str.casefold()` je správný nástroj pro porovnání řetězců necitlivé k velikosti písmen:

```

1 def addresses_match(new, old):
2     return new.casefold() == old.casefold()
3
4 address_in_database = "Imaginary Straße, 27"
5 new_address = "IMAGINARY STRASSE, 27"
6
7 print(addresses_match(new_address, address_in_database)) # True

```

Slovník nerozlišující velikost písmen

Chtěli bychom něco takového:

```

1 class CaseInsensitiveDict(dict):
2     ...
3
4 d = CaseInsensitiveDict()
5 print(d) # {}
6
7 d["Rodrigo"] = "Rodrigo"
8 print(d["RODRIGO"]) # Rodrigo
9 del d["roDRiGo"]
10 print(d) # {}
11
12 d["straße"] = "street"
13 d["STRASSE"] = "STREET"
14 print(d) # {'strasse': "STREET"}

```

Slovníky používají pro přístup ke klíčům, pro nastavování hodnot pro nový klíč, a pro odstraňování klíčů ze slovníku [dunder metody](#).

- `__setitem__(key, value)` se volá, když chceme nastavit hodnotu pro klíč, například příkazem `d[key] = value`;
- `__getitem__(key)` se volá, když chceme ze slovníku získat hodnotu pro daný klíč, např. příkazem `print(d[key])`;

- `__delitem__(key)` se volá, když chceme ze slovníku odstranit ze slovníku klíč a odpovídající hodnotu, např. příkazem `del d[key]`.

Takže potřebujeme přepsat tyto metody tak, aby se před použitím klíčů tyto normalizovali pomocí `str.casefold`. Abychom si ulehčili práci, odvodíme naši třídu `CaseInsensitiveDict` od základního slovníku `dict`. Tak nám zůstane starost, aby se pro normalizaci klíčů použila funkce `str.casefold`, ale jinak chování slovníku převezmeme z mateřské třídy `dict`.

```
1 class CaseInsensitiveDict(dict):
2     """Case-insensitive dictionary implementation."""
3
4     def __getitem__(self, key):
5         return dict.__getitem__(self, key.casefold())
6
7     def __setitem__(self, key, value):
8         dict.__setitem__(self, key.casefold(), value)
9
10    def __delitem__(self, key):
11        dict.__delitem__(self, key.casefold())
```

Zavedli jsme malé úpravy slovníku, ale složitou funkcionalitu přebíráme od `dict`. Naše implementace pracuje takto:

```
1 d = CaseInsensitiveDict()
2 print(d) # {}
3
4 d["Rodrigo"] = "Rodrigo"
5 print(d["RODRIGO"]) # Rodrigo
6 del d["roDRiGo"]
7 print(d) # {}
8
9 d["straße"] = "street"
10 d["STRASSE"] = "STREET"
11 print(d) # {'strasse': 'STREET'}
```

Množina, nerozlišující velikost písmen

Aby množina nerozlišovala velikost písmen v prvcích, musíme použít `str.casefold` při všech manipulacích s prvky množiny, např.

- při přidání nového prvku, `set.add`;
- při odstraňování prvků z množiny, `set.remove` / `set.discard` (jak se liší?);
- při kontrole, zda se prvek nachází v množině, `value in set`.

Pro ilustraci si vytvoříme třídu `CaseInsensitiveSet`, do kterého můžeme přidávat prvky, odebírat je a zjišťovat, zda se prvek nachází v množině bez ohledu na velikost písmen.

Chceme, aby třída dělala přibližně toto:

```
1 class CaseInsensitiveSet(set):
```

```

2      # ...
3
4      s = CaseInsensitiveSet()
5
6      s.add("Rodrigo")
7      s.add("mathspp")
8      s.add("RODRIGO")
9
10     print(s) # CaseInsensitiveSet({'rodrigo', 'mathspp'})
11     print("RODRIGO" in s) # True
12
13     s.discard("MaThSpP") # Try to remove "mathspp"
14     print(s) # CaseInsensitiveSet({'rodrigo'})
15
16     s.discard("mathspp") # Try to remove "mathspp"
17     print(s) # CaseInsensitiveSet({'rodrigo'})
18
19     s.add("mathspp")
20     s.remove("rodrigo") # Remove "rodrigo" and error if not present
21     print(s) # CaseInsensitiveSet({'mathspp'})

```

Použijeme stejný postup jako u slovníku. Pro zjištění, zda se hodnota nachází v množině, používáme *dunder* funkci `__contains__`:

```

1 class CaseInsensitiveSet(set):
2     def add(self, value):
3         set.add(self, value.casefold())
4
5     def discard(self, value):
6         set.discard(self, value.casefold())
7
8     def remove(self, value):
9         set.remove(self, value.casefold())
10
11     def __contains__(self, value):
12         return set.__contains__(self, value.casefold())

```

Takováto implementace není úplná - potřebovali bychom upravit ještě několik dalších *dunder* metod, např. `set.__update__()`.

ChatGPT a domácí úkol

Chtěl jsem si vyzkoušet, jak ChatGPT zvládne vyřešit jeden z domácích úkolů. Následující je záznam z chatu:

Prosím, vygeneruj kód v jazyce Python, který bude řešit následující úlohu:

Samozřejmě, rád vám s tím pomohu! Prosím, popište mi co nejpodrobněji, co by váš kód měl dělat, abych vám mohl připravit co nejlepší řešení.

Na vstupu je číslo N, pak N řádků se slovy slovníku, pak číslo M a nakonec M řádků s dotazy (slovy). Pro každý dotaz najdete ve slovníku všechny jeho přesmyčky, tedy slova, která jsou složena ze stejných písmen, jen možná v jiném pořadí (každé slovo je tedy svou vlastní přesmyčkou). Přesmyčky ke každému dotazu vypíšete na samostatný řádek v abecedním (lexikografickém) pořadí, oddělené mezerami. Pokud k dotazu není nalezena žádná přesmyčka, vypíšete prázdný řádek.

Jistě, zde je kód v jazyce Python, který vyhovuje vašim požadavkům:

python

```
1  from collections import Counter
2
3  # načtení vstupních hodnot
4  n = int(input())
5  dictionary = [input().strip() for _ in range(n)]
6  m = int(input())
7  queries = [input().strip() for _ in range(m)]
8
9  # vytvoření seznamu přesmyček pro každý dotaz
10 anagrams = {}
11 for query in queries:
12     query_counts = Counter(query)
13     anagram_list = [word for word in dictionary if Counter(word) ==
14 query_counts]
15     anagram_list.sort()
16     anagrams[query] = anagram_list
17
18 # vypsání výsledků
19 for query in queries:
20     for anagram in anagrams[query]:
21         print(anagram, end=' ')
22     print()
```

Kód nejprve načte vstupní hodnoty a uloží je do proměnných `n`, `dictionary`, `m` a `queries`. Poté vytvoří seznam přesmyček pro každý dotaz a uloží je do slovníku `anagrams`, kde klíče jsou dotazy a hodnoty jsou seznamy přesmyček odpovídajících dotazu. Seznam přesmyček se vytvoří tak, že se pro každé slovo ze slovníku spočítají jeho písmena a porovnájí se s písmeny dotazu. Pokud se shodují, slovo je přidáno do seznamu přesmyček. Seznam je seřazen lexikograficky.

Nakonec kód projde všechny dotazy a vypíše seznam přesmyček pro každý dotaz. Pokud nebyly nalezeny žádné přesmyčky, vypíše se prázdný řádek.

Toto je dost pomalý kód, protože pro každý dotaz musí projít celým seznamem slov a u každého slova porovnat písmena, z nichž pozůstává. Nešlo by to vyhledávání implementovat efektivněji?

Ano, máte pravdu, že tento kód může být pomalý pro velká vstupní data. Efektivnější řešení by mohlo být použít slovník, kde klíčem by byla hash hodnota převedeného slova a hodnotou by byl seznam všech slov se stejnou hash hodnotou. Potom by stačilo porovnávat hash hodnoty namísto písmen pro každé slovo v seznamu, což by mělo být mnohem rychlejší.

Zde je kód s tímto vylepšením:

python

```
1  from collections import Counter
2
3  # načtení vstupních hodnot
4  n = int(input())
5  dictionary = [input().strip() for _ in range(n)]
6  m = int(input())
7  queries = [input().strip() for _ in range(m)]
8
9  # vytvoření slovníku hash hodnot a seznamů přesmyček pro každý dotaz
10 anagrams = {}
11 for word in dictionary:
12     word_hash = hash(frozenset(Counter(word).items()))
13     if word_hash in anagrams:
14         anagrams[word_hash].append(word)
15     else:
16         anagrams[word_hash] = [word]
17
18 # vypsání výsledků
19 for query in queries:
20     query_hash = hash(frozenset(Counter(query).items()))
21     if query_hash in anagrams:
22         anagram_list = sorted(anagrams[query_hash])
23         print(' '.join(anagram_list))
24     else:
25         print()
```

Kód funguje téměř stejně jako původní kód, ale namísto porovnávání písmen pro každé slovo se používá hash hodnota vytvořená z počtu výskytů jednotlivých písmen v daném slově pomocí třídy `Counter`. Slovník `anagrams` obsahuje seznamy přesmyček pro každou hash hodnotu, což by mělo být rychlejší než hledání přesmyček v celém seznamu slov.

Pozor, tato implementace má ale jednu výraznou nevýhodu. Pokud nějaká dva různá slova mají stejnou hash hodnotu, budou přidána do stejného seznamu přesmyček, i když ve skutečnosti nejsou přesmyčkami. Tento problém lze řešit pomocí nějakého dalšího mechanismu na kontrolu, zda jsou slova skutečně přesmyčkami.

Abychom předešli konfliktům hash hodnot, nemohli bychom namísto hashe vytvořit pro každé slovo nové slovo tak, že uspořádáme písmena slova podle abecedy? Takovéto abecedně seřazené slovo bychom pak mohli používat namísto hashe. Co si o tom myslíš?

Ano, použití abecedně seřazených slov místo hashovací funkce je další způsob, jak řešit konflikty hash hodnot, a mělo by to fungovat dobře. Toto řešení je sice jednodušší, ale může být náročnější na paměť, protože pro každé slovo musíme uložit jeho abecedně seřazenou verzi a porovnávat ji s abecedně seřazenými verzemi v seznamu při vyhledávání.

Zde je kód, který používá abecedně seřazené slovo namísto hashe:

python

```
1 # načtení vstupních hodnot
2 n = int(input())
3 dictionary = [input().strip() for _ in range(n)]
4 m = int(input())
5 queries = [input().strip() for _ in range(m)]
6
7 # vytvoření slovníku přesmyček pro každý dotaz
8 anagrams = {}
9 for word in dictionary:
10     sorted_word = ''.join(sorted(word))
11     if sorted_word in anagrams:
12         anagrams[sorted_word].append(word)
13     else:
14         anagrams[sorted_word] = [word]
15
16 # vypsání výsledků
17 for query in queries:
18     sorted_query = ''.join(sorted(query))
19     if sorted_query in anagrams:
20         anagram_list = sorted(anagrams[sorted_query])
21         print(' '.join(anagram_list))
22     else:
23         print()
```

Tento kód funguje podobně jako předchozí řešení s hashovací funkcí, ale namísto hashovací funkce používá abecedně seřazené verze slov. Pokud jsou dvě slova přesmyčky, budou mít stejnou abecedně seřazenou verzi a budou přidána do stejného seznamu přesmyček. Pokud tedy máme slova "abcd" a "dcba", abecedně seřazené verze obou slov budou "abcd", a proto budou přidána do stejného seznamu přesmyček.

Obě řešení mají své výhody a nevýhody a vhodné řešení závisí na konkrétní situaci. Pokud jsou slova vstupu relativně krátká, použití abecedně seřazených slov může být efektivnější. Na druhé straně, pokud jsou slova delší a mají velký počet různých znaků, může použití hashovací funkce být rychlejší a méně náročné na paměť.

-- konec GPT chatu --

Co si o tom myslíte?

Maximum a jiné vlastnosti posloupností

Tady si procvičíme úplně jednoduché věci, zčásti také proto, abychom si zopakovali některé postupy, které využijete pro domácí úkoly.

Načtení posloupnosti z konzole a ze souboru

Typické zadání úlohy v ReCodExu:

Načtěte ze standardního vstupu posloupnost desetinných čísel oddělených znakem nového řádku, ukončenou řádkou s číslem -1 (toto číslo do posloupnosti nepatří)

Generická verze:

```
1 def read_from_console() -> list[float]:
2     a = []
3     while True:
4         line = input()
5         if "-1" in line:
6             break
7         a.append(float(line))
8     return a
```

Poznámky:

1. Kód `-> list[float]` oznamuje Pythonu, že výstupem funkce je seznam desetinných čísel. Podobně můžeme oznámit i typy parametrů, a interpret pak zahlásí chybu, pokud někde použijeme nesprávný typ.
2. Logika: Máme nekonečný cyklus a vyskakujeme z něj, pokud narazíme na znak konce posloupnosti. Už jsme si ukazovali, že to jde udělat i šikovněji.
3. Testujeme `"-1" in line` namísto `line == "-1"` Proč? (Už vás někdy ReCodEx trýznil hláškami "End-of-file error" u testů?)
4. Načítáme do paměti celou posloupnost - to nemusí být praktické.

Ze souboru:

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

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.

```

1 m = float("-inf")
2 while "-1" not in (line := input()):
3     number = float(line)
4     if number > m:
5         m = number
6

```

Poznámky:

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

```

1 def read_from_console():
2     while "-1" not in (line := input()):
3         yield float(line)
4     return
5
6 m = float("-inf")
7 for number in read_from_console():
8     if number > m:
9         m = number
10 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:

```

1 def reduce(function, iterable, initializer=None):
2     it = iter(iterable)
3     if initializer is None:
4         value = next(it)
5     else:
6         value = initializer
7     for element in it:
8         value = function(value, element)
9     return value

```

Tedy funkce `reduce` propaguje a aktualizuje nějaký stav přes posloupnost.

```

1  from functools import reduce
2
3  def read_from_console():
4      while "-1" not in (line := input()):
5          yield float(line)
6      return
7
8  maximum = reduce(max, read_from_console, float("-inf"))
9  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.

2D úlohy

Máme matici $r \times s$, třeba s celými čísly. Chceme

- v matici nalézt maximální podmatici tvořenou kladnými čísly (podmaticí rozumíme souvislý obdélníkový výřez, maximální podmatice znamená podmatice tvořená co nejvíce prvky)
- v matici nalézt podmatici s maximálním součtem prvků.