resitev

January 28, 2024

1 Day 3: Rucksack Reorganization

(izvirnik naloge)

1.1 Vrednosti črk

Podatki vsebujejo seznam nizov. V prvem delu je potrebno za vsak niz ugotoviti, kateri znak se pojavi tako v prvi kot v drugi polovici. V drugem delu pa je potrebno gledati po tri zaporedne vrstice in ugotoviti, kateri znak se pojavi v vseh. V obeh primerih je iskani rezultat vsota vrednost teh znakov. Vrednosti znakov so a=1, b=2, c=3 in tako naprej do z=26, potem pa A=27, B=28 in tako do Z.

Za začetek poglejmo, kako dobiti vrednosti znakov, če imamo nek znak `c`, na primer

[1]:
```
c = "R"
```

1.1.1 Kode ASCII

Pomagamo si lahko s kodo ASCII, ki jo vrne funkcija `ord`:

[2]:
```
ord("a")
```

[2]: 97

[3]:
```
ord("b")
```

[3]: 98

[4]:
```
ord("c")
```

[4]: 99

[5]:
```
ord("A")
```

[5]: 65

[6]:
```
ord("B")
```

[6]: 66

Vrednost znaka je enaka kodi minus 96, če gre za malo črko, oz. kodi -64 + 26, če gre za veliko. Torej:

[7]:
```
ord(c) - (96 if c >= "a" else (64 - 26))
```

[7]:
```
44
```

Ali, malenkost kompaktneje in zavedajoč se, da je `True` isto kot 1 in `False` isto kot 0:

[8]:
```
ord(c) - [64 - 26, 96][c >= "a"]
```

[8]:
```
44
```

1.1.2 Seznam črk

Modul `string` ima spremenljivko `ascii_letters`, ki vsebuje vse črke angleške abecede.

[10]:
```
import string

string.ascii_letters
```

[10]:
```
'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

Vrednost črke je enaka indeksu v ta niz + 1:

[11]:
```
string.ascii_letters.index(c) + 1
```

[11]:
```
44
```

Če bi radi malo skrajšali program, podtaknemo nizu na začetek še en znak:

[12]:
```
import string

letters = " " + string.ascii_letters
```

In zdaj

[13]:
```
letters.index(c)
```

[13]:
```
44
```

1.1.3 Slovar

Seznami so, vsaj teoretično, počasni, zato je bolj prav uporabiti slovar. Kako sestaviti slovar, katerega ključi bodo črke, vrednosti pa vrednosti teh črk?

Načinov je kolikor hočemo.

V vsakem primeru se nam splača iti prek znakov v `string.ascii_letters`. Razlike bodo v tem, kako pridelati vrednosti.

Uporabimo lahko gornje trike, na primer `ord`

[18]:
```
vrednosti = {c: ord(c) - [64 - 26, 96][c >= "a"] for c in string.ascii_letters}
```

ali `index`

[19]:
```
vrednosti = {c: string.ascii_letters.index(c)  + 1 for c in string.
 ↪ascii_letters}
```

Vrednosti pa lahko dobimo tudi s štetjem:

[23]:
```
vrednosti = {c: i for i, c in enumerate(string.ascii_letters, start=1)}
```

`enumerate` ima uporaben argument `start`. Preživeli pa bi tudi brez, le namesto `c: i` bi pisali `c: i + 1`.

Še bolj luštno je, če se spomnimo, da lahko `dict` sestavimo tudi tako, da mu podamo seznam parov:

[25]:
```
dict([("Ana", 5), ("Berta", 9)])
```

[25]:
```
{'Ana': 5, 'Berta': 9}
```

Tega seveda ne počne noben normalen človek, saj je lažje `{"Ana": 5, "Berta": 9}` ali kar `dict(Ana=5, Berta=9)`. Pač pa bomo slovar sestavili na ta način, kadar seznam takšnih parov slučajno že imamo ali pa si ga lahko pridelamo s kakim `zip`. Konkretno, tule sestavljamo seznam iz črk, ki jih priskrbi `string.ascii_letters` in števil od 1 naprej, ki jih dobimo s funkcijo `count` iz modula `itertools`.

[26]:
```
import string
from itertools import count

vrednosti = dict(zip(string.ascii_letters, count(1)))
```

Ta seveda zmaga.

1.2 Prvi del: Samo preseki

Ni kaj: začetniška naloga iz presekov. Iz niza potegnemo prvo in drugo polovico ter ju spremenimo v množici. Sestavimo presek; ta bo, če je verjeti sestavljalcu naloge, vseboval točno en element, in dobimo ga s `pop()`.

[34]:
```
vsota = 0
for vrstica in open("input.txt"):
    sredina = len(vrstica) // 2
    prva = set(vrstica[:sredina])
    druga = set(vrstica[sredina:])
    skupni = (prva & druga).pop()
    vsota += vrednosti[skupni]

print(vsota)
```

```
8085
```

Če tole bere kdo, ki ni prav izkušen v Pythonu, ga opozorimo na par detajlov.

Prvi je celoštevilsko deljenje. Rezultat deljenja z `/` je v Pythonu vedno `float`, `float`-ov pa ne moremo uporabiti za indekse. Zato delimo celoštevilsko, z `//`. Nizi v nalogi so vedno sode dolžine, tako da smo lahko brzz skrbi.

Funkcija (no, v resnici *konstruktur tipa*) `set` kot argument prejme poljubno reč, prek katere je možno iterirati. Torej mu damo preprosto niz, pa imamo množico.

Nato sestavimo presek in `pop`-nemo iz njega prvi element. Preseka ni potrebno spravljati v novo množico, pač pa seveda ne smemo izpustiti oklepajev.

Gornja rešitev je napisana lepo, počasi, po korakih. Šlo pa bi tudi brez vmesnih spremenljivk.

[36]:
```
vsota = 0
for vrstica in open("input.txt"):
    vsota += vrednosti[(set(vrstica[:len(vrstica) //2]) &␣
 ↪set(vrstica[len(vrstica) // 2:])).pop()]

print(vsota)
```

```
8085
```

A je to boljše? Niti slučajno. Je hitrejše? (Dvomim. Vsaj ne bistveno. Čemu potem to kažem? Samo zato, da pridemo do (med nekaterimi cenjene) rešitve v eni vrstici. :)

[41]:
```
print(sum(
    vrednosti[(set(vrstica[:len(vrstica) //2])
               & set(vrstica[len(vrstica) // 2:])
              ).pop()]
    for vrstica in open("input.txt")))
```

```
8085
```

Če se zdi komu goljufija, da smo že poprej napisali vrstico, v kateri sestavimo slovar `vrednosti`, pa ga lahko nadomesti s katero od drugih različic, torej `ord` ali `indeks`.

1.3 Drugi del: kosi po tri

Drugi del je zelo podoben: računamo preseke po treh zaporednih vrstic. Zabavno je vprašanje, ko priti do treh vrstic.

1.3.1 Preberemo vrstico in še dve

Odpremo datoteko, a tokrat shranimo njeno ime v spremenljivko. Z zanko `for` beremo vrstice, a po vsaki prebrani vrstici z `readline()` preberemo še dve. Vse tri pretvorimo v množice in izračunamo presek.

[47]:
```
vsota = 0
```

```
with open("input.txt") as f:
    for vrstica in f:
        presek = set(vrstica.strip()) & set(f.readline()) & set(f.readline())
        vsota += vrednosti[presek.pop()]

print(vsota)
```

```
2515
```

Čemu `strip()`? In čemu le enkrat? Vse tri vrstice se končajo z `\n`, torej se bo poleg iskanega znaka v vseh treh vrsticah pojavil tudi ta znak. Moramo se ga torej znebiti; zadošča pa, da se ga znebimo enkrat.

1.3.2 Generatorji

Bolj zabavno je uporabljati generator. Kot vsi objekti, čez katere je mogoče iterirati, tudi datoteka vrne generator, če pokličemo funkcijo `iter`. Priskrbimo si torej iterator, potem pa ga trikrat podtaknemo `zip`-u. Zip ne ve - oziroma ga, točneje, nič ne briga - da je trikrat dobil isti iterator. Ko poženemo zanko čez `zip`, bo pač pridobil po en element od vsakega iteratorja in ker gre za en in isti iterator, bomo dobili tri zaporedne elemente.

Kako to deluje, si za začetek raje oglejmo na preprostem seznamu.

[48]:
```
s = [4, 2, 6, 9, 7, 5, 2, 4, 7]
it = iter(s)
for x in zip(it, it, it):
    print(x)
```

```
(4, 2, 6)
(9, 7, 5)
(2, 4, 7)
```

Razred `set` ima metodo `intersection`, ki ji podamo poljubno število poljubnih objektov in vrnila bo njihov presek. Le prvi element mora biti nujno množica.

[57]:
```
set.intersection({3, 6, 7}, (1, 2, 6), [6, 3, 1])
```

[57]:
```
{6}
```

[58]:
```
set.intersection(set("peter"), "jože", "janez")
```

[58]:
```
{'e'}
```

Tako pridemo do te rešitve:

[64]:
```
it = iter(open("input.txt"))
print(sum(
    vrednosti[(set.intersection(*map(set, triplet)) - {"\n"}).pop()]
for triplet in zip(it, it, it)))
```

2515

Z `for triplet in zip(it, it, it)` dobivamo trojke.

Zdaj pa moramo brati od znotraj: `map(set, triplet)` jih bo spremenil v množice (ker moramo spremiti eno, je vseeno, če spremenimo vse tri). Te tri množice moramo podati kot argument funkciji `set.intersection`; ker gre tu za tri reči, mi pa potrebujemo tri argumente, je potrebna zvezdica. Iz preseka odstranimo `\n` in s `pop` poberemo preostali element. Potem pa vzmemo vrednost in seštevamo.

Če bi radi to spravili v eno vrstico, zamenjamo `it, it, it` z `*[iter(open("input.txt"))] * 3`, torej

[66]:
```
it = iter(open("input.txt"))
print(sum(
    vrednosti[(set.intersection(*map(set, triplet)) - {"\n"}).pop()]
for triplet in zip(*[iter(open("input.txt"))] * 3)))
```

2515

[68]:
```
[iter(open("input.txt"))] * 3
```

[68]:
```
[<_io.TextIOWrapper name='input.txt' mode='r' encoding='UTF-8'>,
 <_io.TextIOWrapper name='input.txt' mode='r' encoding='UTF-8'>,
 <_io.TextIOWrapper name='input.txt' mode='r' encoding='UTF-8'>]
```

vsebuje tri kopije istega iteratorja, natančno isto kot `[it, it, it]`). Vendar ne moremo klicati

[69]:
```
zip([iter(open("input.txt"))] * 3)
```

[69]:
```
<zip at 0x7f8b30371d80>
```

saj bi `zip` v tem primeru kot argument dobil en sam seznam. Kot argument mu hočemo dati vse elemente tega seznama, zato torej

[70]:
```
zip(*[iter(open("input.txt"))] * 3)
```

[70]:
```
<zip at 0x7f8b3036d300>
```

Mimogrede pa opazimo: `iter(f)`, kjer je `f` neka datoteka, vrne kar to datoteko:

[73]:
```
f = open("input.txt")

f is iter(f)
```

[73]:
```
True
```

Torej niti ni potrebno klicati `iter`, temveč lahko napišemo kar

[75]:
```
print(sum(
    vrednosti[(set.intersection(*map(set, triplet)) - {"\n"}).pop()]
for triplet in zip(*[open("input.txt")] * 3)))
```

```
2515
```

1.4 Kotlin

Pa dajmo še v Kotlinu. :)

```
import java.io.File

println(
    File("input.txt")
    .readLines()
    .map { it.trim() }
    .map { it.take(it.length / 2).toSet()
            .intersect(it.takeLast(it.length / 2).toSet()).first() }
    .sumOf { it.code -  (if (it >= 'a') 96 else 64 - 26) }
)

println(
    File("input.txt")
    .readLines()
    .map { it.trim() }
    .chunked(3)
    .map {
        it.map { it.toSet }
        .reduce { acc, s -> acc.intersect(s) }
        .first()
    }
    .sumOf { it.code -  (if (it >= 'a') 96 else 64 - 26) }

)
```

Ideja je enaka: naredimo množici, presek in potem seštejemo vrednosti znakov; dobimo jih z `it.code`, ki je ekvivalenten Pythonovemu `ord(it)`.

Bolj zanimiv je drugi del: Kotlinovi seznami imajo `chunked(n)`, ki vrne seznam seznamov po `n` elementov. Te potem spremenimo v množice, z `reduce` izračunamo njihov presek in vzamemo prvi element.