

## Day 2: Password Phyloposhy

Povezava na nalogo

Podatki so v obliki

```
1-3 a: abcde
1-3 b: cdefg
2-9 c: cccccccc
```

Vsaka vrstica vsebuje dve števili, črko in niz. Prvi del pravi, da je geslo (zadnji niz) veljavno, če je število ponovitev podanega znaka znotraj meja, ki jih opisujeta prvi številki. Drugi del pa spremeni pravilo tako, da mora biti podani znak na enem (in le enem) izmed mest, ki ga določata števili; v prvem primeru, recimo, hočemo **a** na prvem ali tretjem mestu.

V obeh delih je naloga, da izpišemo število veljavnih gesel.

### Branje podatkov

Tole je predvsem naloga iz branja podatkov, zato se poigrajmo s tem.

#### Pridna šolarka

Sorry za stereotip. Ampak, pač, tole bo zgledna, netvegana rešitev.

Gremo prek vseh vrstic. Vsako razsekamo glede na presledke in to razpakiramo v terke. Prvi element razbijemo še glede na -. Vse skupaj zložimo v terko, pri čemer prva niza pretvorimo v števili, od znaka pa odbijemo odvečni :.

```
passwords = []
for line in open("example.txt"):
    interval, character, password = line.split()
    fr, to = interval.split("-")
    passwords.append((int(fr), int(to), character.strip(":"), password))
```

Lepota te rešitve je, da smo števili pretvorili v `int`. V rokohitrskem branju spodaj bosta ostala niza, tako da ju bo potrebno pretvarjati kasneje, sproti.

```
passwords
[(1, 3, 'a', 'abcde'), (1, 3, 'b', 'cdefg'), (2, 9, 'c', 'cccccccc')]
```

#### Cenen hek

Če zamenjamo - s presledkom in : z ničemer, se niz spremeni v, na primer, "1 3 a abcde". Odtod potrebujemo le še najobičajnejši `split`, pa dobimo štiri elemente, ki jih potrebujemo.

```
passwords = [line.replace("-", " ").replace(":", "").split() for line in open("example.txt")]
passwords
```

```
[('1', '3', 'a', 'abcde'),
 ('1', '3', 'b', 'cdefg'),
 ('2', '9', 'c', 'cccccccc')]
```

Prva dva sta zdaj očitno niza in ne števili. Not good, ampak ne bo hudega.

## Regularni izrazi

Refleks vsakega resnega programerja, ki pred seboj zagleda takole reč, so regularni izrazi. V redno snov Programiranja 1 le-ti ne sodijo, vendar jih mnogi študenti že poznajo iz kakega drugega vica. Tu jih na hitro predstavimo in uporabimo.

Nizi, ki jih imamo se začnejo z nekaj števki. Števko predstavimo z `\d`; če hočemo povedati, da se števke ponavljajo, in to vsaj enkrat, dodamo `+`, torej `\d+`. Sledil bo znak `-` in potem spet števke. Torej imamo `\d+-\d+`. Sledi presledek in potem poljubna črka med a in z, kar opišemo z `[a-z]`. Sledi dvopičje in potem poljubno število ponovitev poljubnih črk, `[a-z]+`. Vse skupaj je torej `\d+-\d+[a-z]+`.

Tisti, ki to vidite prvič: temu rečemo regularni izraz. Z njim opišemo obliko niza. Regularne izraze je pogosto težko brati; tale je pravzaprav izjemno berljiv.

Na tem mestu se od vseh, ki to vidite prvič, najbrž poslavljamo; spet se vidimo v naslednjem razdelku.

V regularnem izrazu bomo morali še označiti skupine, ki jih želimo razbrati. Skupine bi bilo možno tudi poimenovati, ta izraz pa je tako preprost, da tega ne bomo storili, niti to ni potrebno, saj bomo vedno potrebovali preprosto vse štiri skupine v njihovem originalnem vrstnem redu. Naš izraz je torej `(\d+)-(\d+)([a-z]):([a-z]+)`.

Z njim bomo takole prebrali datoteko.

```
import re
```

```
passwords = [re.match(r"(\d+)-(\d+) ([a-z]): ([a-z]+)", line).groups() for line in open("ex
```

```
passwords
```

```
[('1', '3', 'a', 'abcde'),
 ('1', '3', 'b', 'cdefg'),
 ('2', '9', 'c', 'cccccccc')]
```

Niz vsebuje vzvratne poševnice, `\`. Te moramo seveda podvojiti (`\\n` je v resnici poševnica in `n`, medtem kot je `\n` znak za novo vrstico). Regularni izrazi so navadno polni vzvratnih poševnic in so dovolj neberljivi že brez njihovega podvojevanja, hvala. Zato v Pythonu radi uporabimo r-nize (*r-string*, *raw string*). Pred narekovaj damo `r` in Python bo pustil poševnice pri miru.

Funkciji `re.match` podatmo regularni izraz in niz; kot rezultat vrne `MatchObject` ali `None`. Mi se delamo, da bodo vse vrstice oblikovane pravilno, zato lepo

pokličemo metodo `groups()`, ki vrne vsebino vseh skupin.

### Produksijska verzija

Če bi vas najel kot programerje, bi si želel, da mi napišete tole.

```
import re
from typing import NamedTuple
```

```
class PassData(NamedTuple):
    fr: int
    to: int
    character: str
    password: str

    @classmethod
    def from_seq(cls, s):
        return cls(int(s[0]), int(s[1]), s[2], s[3])
```

```
re_pass = re.compile(r"(\d+)-(\d+) ([a-z]): ([a-z]+)")
passwords = [PassData.from_seq(re_pass.match(line).groups()) for line in open("example.txt")]
```

Spet imamo regularni izraz. To bo najbrž hitreje in pravilneje od `split-ov`. Spet tudi predpostavimo, da je datoteka oblikovana pravilno. Če bi šlo čisto čisto zares, bi morali seveda preveriti še to in če naletite na napako, to seveda povedati na primeren način.

Novost je, da podatke shranjujemo v `NamedTuple`. Ta ima polja `fr` in `to`, ki sta lepo zgledno `int`-a, ter `character` in `password`, ki sta niza.

Dodali smo tudi poimenovan konstruktor, `from_seq`, ki kot argument prejme zaporedje elementov iz vrstice in jih lepo zloži v novi objekt. Kaj je `@classmethod` se pri tem predmetu ne bomo učili; to je bolj pythonovska zadeva. To kodo torej kažem samo zato, da malo nakažem, kaj se še da.

Zdaj so podatki sestavljeni res zgledno. Vedejo se kot terke, obenem pa lahko do njih dostopamo tudi po imenih.

```
passwords
```

```
[PassData(fr=1, to=3, character='a', password='abcde'),
 PassData(fr=1, to=3, character='b', password='cdefg'),
 PassData(fr=2, to=9, character='c', password='cccccccc')]
```

```
prvi = passwords[0]
```

```
prvi.fr
```

```

1
prvi.password
'abcde'
prvi[-1]
'abcde'

```

## Prvi del

V rešitvah bomo predpostavili, da so podatki prebrati v najlepšo, zadnjo obliko, saj bodo rešitve tako najčitljivejše. Če ni tako, boste, recimo, `pwd.character` zamenjali s `pwd[2]`, `pwd.to` pa z `int(pwd[1])`.

V prejšnjih primerih smo brali le datoteko s tremi primeri (`example.txt`); zdaj preberimo prave podatke.

```

passwords = [PassData.from_seq(re_pass.match(line).groups()) for line in open("input.txt")]
valid = 0
for pwd in passwords:
    if pwd.fr <= pwd.password.count(pwd.character) <= pwd.to:
        valid += 1
print(valid)

```

569

Ni posebne umetnosti. Omembe vreden je le pogoj v `if`: ali je nek `x` znotraj intervala `[a, b]`, izvemo z `a <= x <= b` in ne `x >= a and x <= b`, ki bi v gornjem primeru zahteval, da dvakrat štejemo `pwd.password.count(pwd.character)` (ali pa posebej za to naredimo ločeno spremenljivko).

Ker gre v bistvu za vsoto, jo lahko izračunamo s `sum`. Tipično študenti napišejo:

```

sum(1 for pwd in passwords if pwd.fr <= pwd.password.count(pwd.character) <= pwd.to)

```

569

Kdor se spomni, da je `True` isto kot `1` in `False` isto kot `0`, pa ne sešteva enic.

```

sum(pwd.fr <= pwd.password.count(pwd.character) <= pwd.to
    for pwd in passwords)

```

569

## Drugi del

Kako preveriti, ali je nek znak `c` na `i`-tem ali `j`-tem mestu niza `s`, ne pa na obeh? Recimo tako: pogledamo `s[i] == c in s[j] == c`. Eden od teh dveh mora biti `False`, eden `True`. Se pravi: biti morata različna: `(s[i] == c) != (s[j] ==`

c). Oklepaji so potrebni: brez njih bi Python preveril ali veljajo vse naštete relacije, nas pa zanima, ali je resničnost prve različna od resničnosti druge.

Tako dobimo

```
valid = 0
for pwd in passwords:
    if (pwd.password[pwd.fr - 1] == pwd.character) != (pwd.password[pwd.to - 1] == pwd.character):
        valid += 1
print(valid)
```

346

ali

```
sum((pwd.password[pwd.fr - 1] == pwd.character) != (pwd.password[pwd.to - 1] == pwd.character)
    for pwd in passwords)
```

346