resitev

January 28, 2024

Tokratna naloga je kratka: če ne štejemo glav funkcij (`def ...`), morate napisati morate samo 10 vrstic, pa še eno za dodatno nalogo, če želite. In niti vrstice več.

Nekaj pravil in dovoljenih predpostavk v zvezi s to domačo nalogo.

- Tokrat morajo vse funkcije vsebovati le `return`. Za nekatere je potrebno uporabiti generatorske izraze oz. izpeljane sezname, množice, ali slovarje, nekatere pa je možno napisati še preprosteje.
- Pisanje dodatnih funkcij je tokrat (izjemoma) prepovedano. Izdelek sme vsebovati le zahtevane funkcije. Dovoljena pa je uporaba dodatnih modulov, vendar le teh, ki jih dobimo s Pythonom, ne dodanih knjižnic,
- V tokratni nalogi je `zemljevid` slovar, ki že vsebuje povezave v obe smeri (razen v primeru, da je kolesarska dejansko enosmerna), ključi pa so že množice. Z drugimi besedami: klic `obrnjen_zemljevid` ni več potreben.
- V vseh funkcijah, razen `nove_povezave` in `izvedljiva`, smeš predpostaviti, da vse povezave na poti obstajajo.
- Vsaka podana pot vsebuje vsaj eno povezavo.

0.1 Obvezna naloga

Napiši naslednje funkcije:

0.1.1 1. `nove_povezave`

`nove_povezave(pot, zemljevid)` vrne množico povezav, ki so na podani poti, vendar na zemljevidu ne obstajajo.

Klic `nove_povezave("ABVCDFE")` vrne `{(V, C), (C, D), (F, E)}`, saj te tri povezave ne obstajajo.

(MOL potrebuje takšno funkcijo, ker je 1. 4. 2023 namreč objavil novico o načrtovani gradnji novih kolesarskih povezav, pri kateri so se - po dolgi in napeti debati - odločili ravnati po potrebah kolesarjev.)

Rešitev Na prvi pogled moramo vrniti množico povezav, ki so na dani poti (`pairwise(pot)`), vendar povezavo dodamo le, če ne nastopa kot ključ v `zemljevid`.

[1]:
```
from itertools import pairwise

def nove_povezave(pot, zemljevid):
    return {povezava for povezava in pairwise(pot) if povezava not in zemljevid}
```

Na drugi pogled: imamo množico povezav na poti in zanima nas, katere od teh ne nastopajo v množici povezav na zemljevidu.

[2]:
```
def nove_povezave(pot, zemljevid):
    return set(pairwise(pot)) - set(zemljevid)
```

0.1.2 2. `obiskane_tocke`

`obiskane_tocke(pot)` vrne množico vseh točk, ki se pojavijo na poti.

Klic `obiskane_tocke("ABVURURC")` vrne `{A, B, C, R, U, V}`.

Rešitev Tu pa res ni kaj.

[3]:
```
def obiskane_tocke(pot):
    return set(pot)
```

0.1.3 3. `popravljena_pot`

`popravljena_pot(pot)` popravi napačno zapisano pot. Nekateri pot, sestavljeno iz, recimo odsekov A-B, B-C, C-R, R-I, I-E, namreč opišejo z `"ABBCCRRIIE"`. Funkcijo mora to spraviti v običajni format.

Klic `popravi_pot("ABBCCRRIIE")` vrne `"ABCRIE"`.

(Nekateri = Angelca. Itak.)

Rešitev Pobrati je potrebno vsako drugo točko (`pot[::2]`); tako bomo dobili prvo točko in drugo ponovitev vsake od točk na poti. Na koncu dodamo še zadnjo, ‘pot[-1].

[4]:
```
def popravljena_pot(pot):
    return pot[::2] + pot[-1]
```

To ne bi delovalo, če bi bila pot lahko samo `A`, vendar naloga zagotavlja, da bo vsaka pot dolga vsaj eno povezavo. Najkrajša možna pot je torej `AB`; torej bo `pot[::2]` enak `"A"`, `pot[-1]` pa `"B"` - oboje skupaj je `"AB"`, kar bo pravilen rezultat.

0.1.4 4. `povezave_z_vescino`

`povezave_z_vescino(pot, zemljevid, vescina)` vrne seznam vseh povezav na podani poti, ki zahtevajo podano veščino. Vrstni red elementov mora biti enak vrstnemu redu na poti. Če se ista povezava pojavi večkrat na poti, mora biti tudi večkrat v seznamu.

Klic `povezave_z_vescino("RUVRUTS", zemljevid, "pešci")` vrne `[('R', 'U'), ('V', 'R'), ('R', 'U')]`.

(Ker želi MOL pripraviti tematske poti za turiste.)

Rešitev Zanimajo nas torej povezave na poti (`for povezava in pairwise(pot)`), vendar le tiste, ki zahtevajo to veščino (`if vescina in zemljevid[povezava]`).

[5]:
```
def povezave_z_vescino(pot, zemljevid, vescina):
    return [povezava for povezava in pairwise(pot) if vescina in␣
 ↪zemljevid[povezava]]
```

0.1.5 5. `dolgocasna_pot`

`dolgocasna_pot(pot, zemljevid)` vrne `True`, če pot vsebuje vsaj eno povezava, ki ne zahteva nobene veščine (in `False` sicer).

Klic `dolgocasna_pot("ABVUR")` vrne `True` (zaradi povezave B-V ("brez veze")), klic `dolgocasna_pot("AVUR")` pa vrne `False`.

(S funkcijo bo MOL lažje identificiral mesta, kjer je potrebno zvišati robnike, pustiti, da kolesarsko zaraste trava ali kaj podobnega. Kolesar, ki zeha, je zaspan in nepozoren, kar ogroža njegova varnost.)

Rešitev Da `povezava` ne zahteva nobenih veščin, preverimo kar z `not zemljevid[povezava]`. Prazne stvari so neresnične, torej bo `not zemljevid[povezava]` enak `True`, kadar je `zemljevid[povezava]` prazna množica. Ali za kako (`any`) od množic velja, da je prazna, preverimo z

[6]:
```
def dolgocasna_pot(pot, zemljevid):
    return any(not zemljevid[povezava] for povezava in pairwise(pot))
```

Lahko pa obrnemo: preverimo, ali so vse (`all`) neprazne. Če je tako, potem pot *ni* dolgočasna. Pot *je* dolgočasna, kadar ni res, da *ni* dolgočasna.

[7]:
```
def dolgocasna_pot(pot, zemljevid):
    return not all(zemljevid[povezava] for povezava in pairwise(pot))
```

Nekatere to spomni na de Morganovo pravilo: `any(not ...)` smo zamenjali z `not any(...)`.

Kdor hoče reč rešiti s funkcijami, pa napiše

[8]:
```
def dolgocasna_pot(pot, zemljevid):
    return not all(map(zemljevid.get, pairwise(pot)))
```

0.1.6 6. `dobra_pot`

`dobra_pot(pot, zemljevid)` vrne `True`, če zahtevajo vse povezave na poti vsaj dve veščini.

Klic `dobra_pot("ABCR")` vrne `True`, klic `dobra_pot("ABCRDF")` pa `False` (ker zahtega R-D samo eno veščino.)

(MOL potrebuje to in naslednjo funkcijo, da najde primerne poti za organizacijo dogodka "Kolesarimo po Ljubljani".)

Rešitev Ta je podobna, le da tu za vse (`all`) `povezava` preverjamo, ali velja `len(zemljevid(povezava)) > 2`:

[9]:
```
def dobra_pot(pot, zemljevid):
    return all(len(zemljevid[povezava]) >= 2 for povezava in pairwise(pot))
```

0.1.7 7. `zahtevnost_poti`

`zahtevnost_poti(pot, zemljevid)` vrne zahtevnost poti. Ta je enaka največjemu številu veščin, ki jih zahteva nek odsek na poti.

Klic `zahtevnost_poti("ABCRU")` vrne 3, ker povezava C-R zahteva tri različne veščine - kar je največ, kar najdemo na tej pestri poti.

Rešitev Očitno bo to `max`. Česa? `max` od `len(zemljevid[povezava])` za vse povezave na poti.

[10]:
```
def zahtevnost_poti(pot, zemljevid):
    return max(len(zemljevid[povezava]) for povezava in pairwise(pot))
```

0.1.8 8. `izvedljiva`

`izvedljiva(pot, zemljevid, zivljenj)` vrne `True`, če je pot izvedljiva za kolesarja s podanim številom življenj in `False`, če ni. Kolesar izgubi življenje, če vozi po povezavi, ki je ni na zemljevidu. Če kolesar konča pot z 0 življenji, je mrtev, torej pot zanj ni izvedljiva.

Klic `izvedljiva("AVUSPRDEI", zemljevid, 3)` vrne 3, ker kolesar izgubi vsa tri življenja na U-S, P-R in D-E.

(MOL potrebuje funkcijo za oblikovanje strokovnega priporočila o potrebnem številu življenj, ki naj jih ima kolesar, preden gre na pot v Ljubljani.)

Rešitev Prešteti je potrebno povezave, ki jih ni na zemljevidu. Če je to število večje ali enako številu življenj, je kolesar kaput.

Če imamo seznam `s` in želimo preveriti, za koliko njegovih elementov velja `pogoj`, lahko napišemo `len([x for x in s if pogoj(x)])`, preprosteje in hitreje (če vemo, da je `True` enak `1` in `False` enak `0`) pa je `sum(pogoj(x) for x in s)`. Torej

[11]:
```
def izvedljiva(pot, zemljevid, zivljenj):
    return sum(povezava not in zemljevid for povezava in pairwise(pot)) <␣
 ↪zivljenj
```

Če je kdo poskusil narediti zanko, v kateri bi kolesarju odšteval življenja - torej nekaj takega kot

[12]:
```
def izvedljiva(pot, zemljevid, zivljenj):
    for povezava in pairwise(pot):
        if povezava not in zemljevid:
            zivljenj -= 1
            if zivljenj == 0:
                return False
    return True
```

je ugotovil, da ne gre. Generatorski izraz je zelo težko z našim znanjem napisati tako, da bi bil naslednji korak odvisen od rezultata prejšnjega. V starejših Pythonih bi si pomagali z `reduce`, v novejših z mrožem (operator `:=`) … vendar te stvari navadno niso prav elegantne in čitljive.

0.1.9 9. `enosmerne`

`enosmerne(zemljevid)` vrne množico povezav, ki so samo enosmerne.

V običajnem zemljevidu takih povezav ni, zato klic `enosmerne(zemljevid)` vrne prazno množico. Klic

```
  enosmerne({(A, B): {"robnik", "bolt"},
             (B, A): {"robnik", "bolt"},
             (A, C): {"bolt", "rodeo", "pešci"},
             (C, D): set(),
             (D, C): set(),
             (V, B): {"avtocesta"}
 })
```

pa vrne `{(A, C), (V, B)}`, ker gresta povezavi A-C in V-B le v eno smer.

Rešitev

[13]:
```
def enosmerne(zemljevid):
    return {(od, do) for (od, do) in zemljevid if (do, od) not in zemljevid}
```

Ali pa brez razpakiranja:

[14]:
```
def enosmerne(zemljevid):
    return {povezava for povezava in zemljevid if povezava[::-1] not in␣
 ↪zemljevid}
```

0.1.10 10. `dvosmerne`

`dvosmerne(zemljevid)` vrne nov slovar z zemljevidom, ki vsebuje le povezave, ki so dvosmerne.

Klic `dvosmerne(zemljevid)` vrne enak zemljevid, kot ga je prejel. Klic `dvosmerne` z gornjim zmeljevidom pa vrne

```python
{(A, B): {"robnik", "bolt"},
 (B, A): {"robnik", "bolt"},
 (C, D): set(),
 (D, C): set()}
```

Rešitev Ta je podobna prejšnji, le da zahteva, da sestavimo slovar. Torej bomo šli čez `zemljevid.items()`, da dobimo še pripadajoče vrednosti.

[15]:
```
def dvosmerne(zemljevid):
    return {povezava: vescine
            for povezava, vescine in zemljevid.items()
            if povezava[::-1] in zemljevid}
```

Povezavo lahko tudi razpakiramo - če znamo.

[16]:
```
def dvosmerne(zemljevid):
    return {(od, do): vescine
            for (od, do), vescine in zemljevid.items()
            if (do, od) in zemljevid}
```

0.2 Dodatna naloga

Samo ena funkcija, ki zahteva malo več (če nismo spretni, pa je malo bolj zapletena).

Napiši funkcijo `najzahtevnejsi_odsek(pot, zemljevid)`, ki vrne tisti odsek na poti, ki zahteva največ veščin. Če je takih odsekov več, vrne tistega, ki se na poti pojavi prej.

Rešitev Ta je lahko zapletena - lahko pa znamo poklicati `max` z argumentom `key` in kot `key` podati lambda-funkcijo.

[17]:
```
def najzahtevnejsi_odsek(pot, zemljevid):
    return max(pairwise(pot), key=lambda povezava: len(zemljevid[povezava]))
```

Če `max`-u z argumentom `key` podamo funkcijo, ne bo primerjal objektov temveč tisto, kar za posamične objekte vrne podana funkcija. Funkcija je lahko običajna funkcija, ki bi jo morali definirati posebej, ker naloga ne pusti dodatnih funkcij, pa jo definiramo kar tu, na licu mesta kot anonimno, lamda-funkcijo. `lambda povezava: len(zemljevid[povezava])` je funkcija, ki prejme en argument (`povezava`) in kot rezultat vrne `len(zemljevid[povezava])` - torej število veščin, kar je točno tisto, po čemer želimo primerjati povezave.