

1. Statistika

Senzor na kolesarski stezi beleži podatke o številu kolesarjev, ki ga prevozijo. Podatke zbira v seznam s 24×60 elementi: element z indeksom i pove število prehodov v i -ti minuti dneva. Napišite naslednje funkcije:

- `po_urah(a)` prejme opisani seznam in vrne seznam s 24 elementi, ki vsebujejo število prehodov po urah dneva. (Element z indeksom 2 vsebuje število prehodov v minutarh od 120 do (vključno) 179.)
- `naj_ura(a)` vrne uro z največ prehodi (npr. 5, če je največ prehodov med peto in šesto uro).
- `brez_prehodov(a)` vrne število minut, ko ni bilo kolesarjev (torej število ničelnih elementov podanega seznama).

Za polne točke naj bosta dve od gornjih funkcij napisani v eni vrstici, z uporabo izpeljanega seznama ali generatorja.

Rešitev

Nalogo je mogoče rešiti na kup načinov.

po_urah Prvo funkcijo najpreprosteje rešimo z rezinami: število prehodov v minutah prve ure je v `a[:60]`, prehodi v drugi uri so v `a[60:120]`, v tretji v `a[120:180]` ... in v i -ti v `a[60 * i:60 * (i + 1)]`.

Spomnimo se še na funkcijo `sum`, ki sešteje elemente seznama, pa dobimo

```
def po_urah(a):
    ure = []
    for i in range(24):
        ure.append(sum(a[i * 60:(i + 1) * 60]))
    return ure
```

ali, krajše,

```
def po_urah(a):
    return [sum(a[i * 60:(i + 1) * 60]) for i in range(24)]
```

naj_ura Naj uro poiščemo tako, da s prvo funkcijo izračunamo število prehodov po urah, potem pa poiščemo indeks ure z največjim številom prehodov.

```
def naj_ura(a):
    ure = po_urah(a)
    naj_i = 0
    for i in range(24):
        if ure[i] > ure[naj_i]:
            naj_i = i
    return naj_i
```

Malo težja rešitev Naloga pravi, naj bosta dve (in ne nujno tri) funkcije rešene z izpeljanim seznamom oziroma generatorskim izrazom. Razlog je ta funkcija: da bi rešili to nalogo v eni vrstici, bi morali uporabiti lambda funkcije ali kaj še hujšega (vendar lepšega).

Kar sledi tule, je torej nekoliko napredno - tega od programerjev-začetnikov nekakor ne zahtevamo. Ena možnost je tale:

```
def naj_ura(a):  
    return max(range(24), key=lambda i: po_urah(a)[i])
```

Ta rešitev je slaba, ker 24-krat pokliče funkcijo `po_urah`. Temu se lahko izognemo tako, da jo shranimo v argument s privzeto vrednostjo.

```
def naj_ura(a):  
    return max(range(24), key=lambda i, ure=po_urah(a): ure[i])
```

To je boljše, vendar je packarija. Prava rešitev uporablja `operator.getitem` in `functools.partial`. Poglejmo, kaj počneta.

`getitem` je funkcija, ki ji podamo neko reč (recimo seznam) in indeks, pa vrne element s tem indeksom.

```
from operator import getitem
```

```
a = ["Ana", "Berta", "Cilka", "Dani"]
```

```
getitem(a, 2)
```

```
'Cilka'
```

```
getitem(a, 0)
```

```
'Ana'
```

`partial` je funkcija, ki ji podamo neko funkcijo in nekaj argumentov. Vrne novo funkcijo, ki je enaka stari, vendar tako, da so podani argumenti že "pribiti". Tule jo bomo uporabili tako:

```
from functools import partial
```

```
get_a = partial(getitem, a)
```

```
get_a(2)
```

```
'Cilka'
```

```
get_a(0)
```

```
'Ana'
```

`get_a` je torej funkcija, ki počne isto kot `getitem`, le da je prvi argument že "fiksiran", namreč `a`.

Rešitev naloge je potem

```
def naj_ura(a):
    return max(range(24), key=partial(getitem, po_urah(a)))
```

Malo prelahka rešitev Že med izpitom sem opazil študenta (med ocenjevanjem pa jih bom še več), ki je nalogo rešil tako:

```
def naj_ura(a):
    return po_urah(a).index(max(po_urah(a)))
```

Ta očitna, kratka rešitev mi, priznam, pri sestavljanju izpita sploh ni prišla na misel. Zato, ker tako nikoli ne programiram: ta funkcija mora dvakrat prek seznama `a`. Prava, zaresna, profesionalna rešitev uporablja `max`. Je pa ta rešitev seveda popolnoma pravilna in primerna za prvi letnik študija, tako da jo bom hočem-nočem priznal. :)

brez__prehodov Tako:

```
def brez_prehodov(a):
    nicel = 0
    for x in a:
        if x == 0:
            nicel += 1
    return nicel
```

Ali tako

```
def brez_prehodov(a):
    nicel = 0
    for x in a:
        nicel += (x == 0)
    return nicel
```

Kar nas pripelje do tako:

```
def brez_prehodov(a):
    return sum(x == 0 for x in a)
```

Čistuni pa rečejo

```
def brez_prehodov(a):
    return sum(1 if x == 0 else 0 for x in a)
```

Javascriptovci napišejo

```
def brez_prehodov(a):
    return sum(not x for x in a)
```

Še kdo drug pa še kako drugače.

Nič pa ni narobe, če kdo napiše

```
def brez_prehodov(a):
    return a.count(0)

:)
```

2. Brez kolesarjev

Napišite funkcijo `obdobje_brez(a)`, ki prejme takšen argument kot funkcije iz prejšnje naloge, in vrne začetek in konec najdaljšega obdobja brez prehodov. Če so vsi elementi na indeksih od, na primer, 150 do (vključno) 180 enaki 0 in je to tudi najdaljše zaporedje ničel, mora funkcija vrniti (150, 180). Da bo reševanje lažje, so v testih tudi trije primeri s tabelami, ki nimajo 24×60 temveč le 13 števil.

Rešitev

Prva naloga je bila zelo lahka, praktično podarjena. No, ta je precej težja.

Ena možna rešitev je

```
def obdobje_brez(a):
    naj_zac = 1
    naj_kon = 0
    zac = None
    for i, x in enumerate(a):
        if x == 0:
            if zac is None:
                zac = i
            if i - zac > naj_kon - naj_zac:
                naj_zac, naj_kon = zac, i
        else:
            zac = None
    return naj_zac, naj_kon
```

Gremo prek seznama. `zac` hrani indeks začetka trenutnega zaporedja ničel. Če trenutno nismo v zaporedju ničel, je `zac` enak `None`.

Če je trenutni element enak 0, pogledamo, ali je trenutno zaporedje daljše od najdaljšega doslej. Če, potem si zapomnimo njegove meje:

```
if i - zac > naj_kon - naj_zac:
    naj_zac, naj_kon = zac, i
```

Če trenutni element ni enak 0, pa le postavimo `zac` na `None`.

Ostane le še malo administracije. V začetku se delamo, da se je najdaljše zaporedje začelo z 1 in končalo z 0. Tako je dolgo -1 in bo že prva ničla, na katero bomo naleteli, predstavljala daljše zaporedje.

Drugi košček administracije je

```

if zac is None:
    zac = i

```

Če naletimo na 0 in doslej še nismo bili v zaporedju ničel, si zapomnimo trenutni indeks kot začetek trenutnega zaporedja.

Nalogo je možno rešiti še na veliko načinov. Gornji morda ni najbolj učinkovit, ker prepogosto nastavlja `naj_zac` in `naj_kon`. Načelno bi bilo boljše, če bi vsakič, ko se zaporedje konča, preverili, ali je bilo to zaporedje daljše od najdaljšega doslej. To bi povzročilo nekaj sitnosti, če je najdaljše zaporedje ravno na koncu ...

Tule je zabavnejša rešitev.

```

def obdobje_brez(a):
    s = ""
    for x in a:
        if x == 0:
            s += "x"
        else:
            s += " "
    naj = max(s.split())
    zac = s.index(naj)
    return zac, zac + len(naj) - 1

```

zaporedje števil pretvorimo v niz, sestavljen iz x-ov in presledkov: ničle spremenimo v x in ne-ničle v presledke. Nato na tem nizu pokličemo `split`. Dobimo "besede"; vsaka je sestavljena iz toliko x-ov, kolikor je bilo zaporednih ničel. "Maksimalna" beseda je beseda, ki je zadnja po abecedi; vse besede so sestavljene iz enakih črk (x-ov); v tem primeru je kasneje po abecedi tista, ki je daljša. `naj` bo torej najdaljša beseda. Pogledamo, kje v nizu se nahaja, pa imamo začetni indeks zaporedja; končni indeks dobimo tako, da k začetnemu prištejemo dolžino besede.

Rešitev lahko skrajšamo v

```

def obdobje_brez(a):
    s = "".join(" x"[x == 0] for x in a)
    naj = max(s.split())
    zac = s.index(naj)
    return zac, zac + len(naj) - 1

```

Prva rešitev, tista z zankami in `naj_zac` in tako naprej je bila tipičen primer programa v C-ju. Tam bi nalogo rešili natančno tako, kot smo ga v Pythonu. Druga rešitev je bila malo kavbojska - zna pa biti, da je v resnici najhitrejše. Bolj Pythonovska bi bila tale:

```

def obdobje_brez(a):
    naj_zac = naj_kon = 0
    zac = 0

```

```

for k, group in groupby(a):
    group = list(group)
    if k == 0 and len(group) > naj_kon - naj_zac + 1:
        naj_zac, naj_kon = zac, zac + len(group) - 1
    zac += len(group)
return naj_zac, naj_kon

```

Kdor jo hoče razumeti, naj pogleda dokumentacijo funkcije `itertools.groupby`. Sam pa priznam, da to funkcijo uporabljam redko. Če bi šlo zares, bi nalogo najbrž rešil po kavbojsko.

Najbrž najlepše pa je poiskati vse začetke in konce intervalov ničel. Potem zipnemo oba seznama (ali generatorja, če hočemo biti še bolj *fancy*) skupaj, in vrnemo par z največjo razliko.

```

def obdobje_brez(a):
    zacetki = (i for i in range(len(a))
               if a[i] == 0 and (i == 0 or a[i - 1] != 0))
    konci = (i for i in range(len(a))
             if a[i] == 0 and (i == len(a) - 1 or a[i + 1] != 0))
    return max(zip(zacetki, konci), key=lambda x: x[1] - x[0])

```

To gre potem čisto lepo celo v enovrstični izraz (čeprav je večvrstični natančno enako učinkovit, hkrati pa preglednejši).

```

def obdobje_brez(a):
    zacetki = (i for i in range(len(a))
               if a[i] == 0 and (i == 0 or a[i - 1] != 0))
    konci = (i for i in range(len(a))
             if a[i] == 0 and (i == len(a) - 1 or a[i + 1] != 0))
    return max(zip(zacetki, konci), key=lambda x: x[1] - x[0])

```

Če povem po pravici: ta rešitev mi ob sestavljanju izpita ni prišla na misel. Idejo sem dobil od študenta,

3. Obremenitve

Kolesarskih stez s senzorji je v resnici več. Recimo, da imamo tri ulice, imenujmo jih Anina, Bertina, Cilkina. Podatki o številu kolesarjev so zbrani v enem samem seznamu: prvi trije elementi se nanašajo na število prehodov čez Anin, Bertin in Cilkin senzor (v tem vrstnem redu) v niči minuti dneva. Naslednji trije se nanašajo na prehode čez te tri senzorje v prvi minuti dneva, naslednji trije na drugo minuto in tako naprej. Celotna dolžina tabele je torej enaka $24 \times 60 \times$ število senzorjev (ki seveda ni nujno vedno tri!)

Napišite funkcijo `obremenitve(imena, porocila)`, ki prejme imena senzorjev in število kolesarjev (kot ga opisujemo zgoraj). Vrniti mora ime najbolj obremenjenega senzorja. Če je teh več, lahko vrne ime poljubnega med njimi.

Rešitev

Tole je spet ena preprostejša naloga. Rešitev bi bila lahko

```
def obremenitve(imena, porocila):
    n = len(imena)
    kolesarjev = [0] * n
    for i, x in enumerate(porocila):
        kolesarjev[i % n] += x
    return imena[kolesarjev.index(max(kolesarjev))]
```

Lahko pa, podobno kot v prvi nalogi, uporabimo rezine

```
def obremenitve(imena, porocila):
    naj_ime = None
    naj_ob = -1
    for i, ime in enumerate(imena):
        ob = sum(porocila[i::len(imena)])
        if ob > naj_ob:
            naj_ob = ob
            naj_ime = ime
    return naj_ime
```

Kdor hoče, pa se lahko poglobi v rešitev, ki jo dobimo, če malo predelamo tole, drugo različico.

```
def obremenitve(imena, porocila):
    return imena[max(
        range(len(imena)),
        key=lambda i: sum(porocila[i::len(imena)])])
```

Tak način programiranja bi bil tipičen za Javascript in, recimo, Kotlin.

4. Minuta za Angelco

Angelca dela doktorat iz kolesarske politike. V okviru doktorata je definirala koncept "zlate minute". Ta se nanaša na obremenjenost kolesarskih stez.

- Vse minute od ničte do (vključno) 59-te so zlate.
- Poleg tega so zlate vse minute, ko je stezo prevozil vsaj en kolesar, vendar pod pogojem, da je zlata tudi minuta ob pol manjšem času. Če gre za liho minuto, poskusimo polovico zaokrožiti navzdol in navzgor; da bo 1017 zlata, zadošča, da je zlata bodisi 508 bodisi 509 (lahko pa sta tudi obe).

Angelco so vprašali, kakšen smisel ima vse to. Odgovorila je, da je to doktorat in tega pač ne more kar vsak razumeti.

- V primerih iz testov minuta 420 ni zlata: v 420 minuti so kolesarsko stezo prevozili 4 kolesarji, vendar v polovični, 210 minuti ni bilo nikogar.
- Minuta 1017 je zlata. Razpolovimo jo lahko v 508 ali 509; obe imata kolesarje. Če izberemo 508, bi to vodilo v 254 in 127, ko ni kolesarjev. 509

pa lahko razpolovimo v 254 ali 255. Če izberemo 254, bi spet padli v 127. Če pa razpolovimo 509 v 255, lahko nadaljujemo v 127 (ni kolesarjev) ali v 128 (so), tega v 64 (so) in tega v 32 (< 60).

Napišite funkcijo `zlata_minuta(i, a)`, ki prejme številko minute (`i`) in seznam `s` prehodi kolesarjev, ter vrne `True`, če je `i`-ta minuta zlata in `False`, če ni.

Rešitev

Samo po definiciji: minuta je zlata, če je manjša od 60 ali pa je zlata polovična minuta ali pa je zlata polovična minuta, zaokrožena navzgor - pri čemer je drugo smiselno preverjati le, če je minuta liha.

```
def zlata_minuta(i, a):
    return i < 60 or (
        a[i] != 0 and (
            zlata_minuta(i // 2, a)
            or (i % 2 == 1 and zlata_minuta(i // 2 + 1, a))
        )
    )
```

Oklepaji - razen tistega za `and` - in prelomi vrstic so seveda nepotrebni.

Če malo spremenimo zaokrožanje, lahko izraz še poenostavimo.

```
def zlata_minuta(i, a):
    return i < 60 or a[i] != 0 and (zlata_minuta(i // 2, a) or zlata_minuta((i + 1) // 2, a))
```

Zdaj vedno preverimo minuti `i // 2` in `(i + 1) // 2`. Če je `i` sod, je to eno in isto. Če je lih, pa je to ravno zaokrožanje navzdol in navzgor. Ja, za sode brez potrebe kličemo dvakrat. Bog pomagaj. :)

5. Nadzorni sistem

Napišite razred `Senzor`:

- konstruktor prejme id senzorja (id neko celo število),
- `prehod(self, smer)`, pri čemer je smer lahko "+" ali "-", zabeleži, da je kolesar prevozil senzor v podani smeri,
- `prehodov(self)`, ki vrne par s številom prehodov v smer "+" in v smer "-".

Poleg tega napišite razred `NadzorniSistem`, katerega konstruktor prejme seznam senzorjev, to je, seznam objektov tipa `Senzor`. `NadzorniSistem` ima primeren konstruktor in metode:

- `prehod(self, id, smer)` objektu, ki predstavlja senzor s podanim id-jem (vedno bo šlo za enega od senzorjev, ki so bili podani konstruktorju) sporoči, da so ga prevozili v podani smeri;
- `prehodov(self, id)` vrne par s številom prehodov prek senzorja s podanim id-jem.

Pazi: NadzorniSistem naj ne shranjuje podatkov o prehodih. Podatki o prehodih so shranjeni v objektih razreda `Senzor`, `NadzorniSistem` pa shranjuje senzorje.

Rešitev

```
class Senzor:
    def __init__(self, id):
        self.id = id
        self.naprej = self.nazaj = 0

    def prehod(self, smer):
        if smer == "+":
            self.naprej += 1
        else:
            self.nazaj += 1

    def prehodov(self):
        return (self.naprej, self.nazaj)

class NadzorniSistem:
    def __init__(self, senzorji):
        self.senzorji = {senzor.id: senzor for senzor in senzorji}

    def prehod(self, id, smer):
        self.senzorji[id].prehod(smer)

    def prehodov(self, id):
        return self.senzorji[id].prehodov()
```

Razred `Senzor` preprosto hrani število prehodov, ga povečuje in vrača.

Bolj zanimiv je `NadzorniSistem`: ta mora shraniti senzorje. Lahko bi shranil kar podani seznam, bolj praktično pa je, da senzorje shrani v slovar, katerega ključi so id-ji senzorjev. To namreč bistveno poenostavi njegovi funkciji `prehod` in `prehodov`.