MATIJA LOKAR

NALOGE ZA PREHOD

IZ PROGRAMA Praktična matematika
V NOVI PROGRAM Aplikativna matematika

Predmet PROGRAMIRANJE 1

2025

O gradivu

Študenti, ki so opravili predmet Programiranje 1 na opuščenem programu Praktična matematika (pred letom 2024/25) in se želijo prepisati na novi program Aplikativna matematika morajo opraviti seminarsko nalogo, ki

V tem gradivu je naveden primer nalog, ki pokrivajo dodatno snov, k jo morajo študenti osvojiti. Naloge so razdeljene v dva dela:

- Naloge brez testnega okolja
- Naloge s testi pravilnosti, na voljo v predmetu na sistemu Projekt TOMO

Določene naloge izvirajo iz obstoječih nalog na sistemu ProjektTOMO, določene pa so originalne, oziroma prilagojene.

Naloge s samostojno pripravo testov

Tu so zbrane naloge, ki pokrivajo 4 glavne tematike

- Osnove
- Izjeme
- Datoteke
- Osnove objektnega programiranja

Ob imenu naloge je navedena tema. Pričakuje se, da študent za vsako nalogo pripravi:

- zapis ideje algoritma rešitve naloge
- ustrezno komentirano kodo rešitve
- testni program

Zoprna sedmica (osnove)

Napiši metodo `PoisciVII(tabela)`, ki za dano tabelo celih števil vrne tabelo z dvema elementoma (število števil, ki so deljiva s 7, in vsoto preostalih števil), iz dane tabele pa pobriše vsa števila, ki so deljiva s 7.

Datoteka z ulomki (datoteke)

Na tekstovni datoteki so zapisana cela števila in ulomki, vsako v svoji vrsti, kot je prikazano spodaj na primeru. Sestavi metodo, ki za dano ime datoteke prebere števila z datoteke, jih zmnoži in rezultat vrne v obliki neokrajšanega ulomka. Za datoteko iz primera bi program izpisal ulomek 2912/1050. V primeru, če so v datoteki vrstice, ki niso ustrezne, sproži izjemo, ki v sporočilu vrne število napačnih vrstic.

stevila.txt
-2 4/35 52 7/2 -1/15

Množimo se … (datoteke)

Na tekstovni datoteki so zapisana cela števila in ulomki, vsako v svoji vrsti, kot je prikazano spodaj na primeru. Sestavi program, ki prebere števila z datoteke, jih zmnoži in izpiše rezultat v obliki (lahko neokrajšanega) ulomka. Za datoteko iz primera bi program izpisal ulomek 2912/1050 ali pa 208/75 ali pa 416/150 ali pa 1456/525.

stevila.txt
-2 4/35 52 7/2 -1/15

Stave (datoteke)

Dandanes je možno staviti na vse mogoče. Janezek ima najrajši stave, ko je treba napovedati skupno število golov, doseženih na rokometni tekmi in na razliko v golih po rednem delu tekme. Na datoteko si je zapisal stave. Sestavi program, ki bo najprej prebral ime te datoteke ter ime izhodne datoteke. Če vhodna datoteka ne obstaja, naj program sprašuje toliko časa, dokler uporabnik ne vnese imena obstoječe datoteke. Nato naj za vsako stavo na izhodno datoteko izpiše, kakšen mora biti rezultat, da bo Janezek stavo dobil. Če so podatki napačni, naj napiše »napačni podatki«. Če pa so podatki načeloma smiselni, a določiti rezultata ni možno, pa naj izpiše “nemogoče”. Npr. Če datoteka vsebuje podatke

```
40 20
20 40
20.5 15
26 45 88
27 3

24 0
26 1
```

naj se na izhodno datoteko izpiše

```
30 10
Nemogoče
Napačni podatki
Napačni podatki
15 12
Napačni podatki
12 12
Nemogoče
```

Iz CSV v asociativni seznam (datoteke, izjeme)

Napiši metodo, ki za vhod sprejme imeni vhodne in izhodne datoteke. Vhodna datoteka vsebuje podatke v formatu CSV. Vsaka vrstica vsebuje en primer, lastnosti primerov pa so ločene z vejicami. Prva vrstica vsebuje imena lastnosti, prav tako ločena z vejicami. Metoda naj podatke iz vhodne datoteke prepiše v izhodno datoteko v obliki asociativnih seznamov – seznamov urejenih parov (lastnost, vrednost). Za pomoč glej spodnji primer. Metoda naj deluje za poljubno število primerov in lastnosti v vhodni datoteki. Če vhodne datoteke ni, sproži ustrezno izjemo. Prav tako sproži izjemo, če podatki na vhodni datoteki niso ustrezni. V sporočilu izjeme sporoči vrstico, v kateri se je pojavila napaka, in samo vhodno vrstico.

Vhodna datoteka

```
Ime, Starost, Teza
Marko, 28, 91
Neža, 25, 57
Vito, 35, 71
```

Izhodna datoteka

```
[(Ime, Marko), (Starost, 28), (Teza, 91)]
[(Ime, Neža), (Starost, 25), (Teza, 57)]
[(Ime, Vito), (Starost, 35), (Teza, 71)]
```

Enomestna števila (Osnove)

Sestavi funkcijo `EnomestnaStevila`, ki ugotovi, koliko enomestnih števil lahko najdemo v danem nizu. Tako je v nizu "12. 3. 2020" le eno enomestno število (3), v "D12" ni nobenega, v "a1b2c3d9" pa so kar 4.

Sestavi tudi testni program, s katerim preveriš delovanje funkcije! Pri tem moraš seveda dobro premisliti, da pri testiranju zajameš vse možnosti!

Ruchtzwertzovo zaporedje (osnove)

Naj bo $n > 1$ naravno število. Končno zaporedje celih števil $a_1, a_2, \ldots, a_k$ imenujemo *Ruchtzwertzovo zaporedje* reda n, če se poljubna dva sosednja člena tega zaporedja razlikujeta za $0, 1$ ali n. Na primer, $3, 4, 4, 3, 3, 10, 10, 9, 8$ je Ruchtzwertzovo zaporedje reda 7.

Sestavi funkcijo, ki vrne True, kadar je v dani tabeli zapisano Ruchtzwertzovo zaporedje reda n. Tudi n je parameter. V nasprotnem primeru naj metoda vrne False

Sestavi tudi testni program, s katerim preveriš delovanje funkcije!

Na datoteko brez oklepajev (datoteke)

Sestavi metodo

```
BrezOklepajev(imeVhod, imeIzhod)
```

ki tekstovno datoteko, katere ime je v `imeVhod` prepiše na novo datoteko z imenom, kot ga določa `imeIzhod`. Pri tem naj iz vrstic, ki vsebujejo izraze v oklepajih, pobriše te izraze skupaj z oklepaji. V vsaki vrstici se lahko pojavi največ en par pravilno postavljenih oklepajev. V primeru, da vrstica ni ustrezno sestavljena, jo v izhodni datoteki nadomesti z vrstico `"NAPAKA"`. Primer:

Vhodna datoteka	*Izhodna datoteka*
`Peter (spet) zamuja.`	`Peter  zamuja.`
`Jaz sem (napisana narobe`	`NAPAKA`
`To ni račun.`	`To ni račun.`
`2*2+(2-7)=8`	`2*2+=8`
`To )( ni prav.`	`NAPAKA`
`[(2)]`	`[]`
`Konec(.).`	`Konec.`

Nasvet: napiši primerne pomožne funkcije.

Pripravi tudi ustrezni testni program!

Kvadratna enačba (osnova, izjeme)

Sestavi funkcijo `ničleKvEnačbe(a, b, c)`, ki vrne obe ničli (kot tabelo velikosti 2) kvadratne enačbe $a x^2 + b x + c = 0$! V primeru, da enačba nima dveh realnih rešitev, sproži izjemo z ustreznim obvestilom. Pri proženju napak loči matematično različne situacije (ni rešitve, neskončno rešitev, kompleksne ...)

Pripravi tudi ustrezni testni program! Pri tem moraš seveda dobro premisliti, da pri testiranju zajameš vse možnosti! Testni program mora biti napisan tako, da vedno izvede vse teste!

Družabne številke (osnove)

Sestavi funkcijo `DruzabneStevke`, ki prešteje družabne številke v danem nizu. Številka je družabna, če je soseda kake druge številke. V nizu "bal123i4e7e12bed42" je sedem družabnih številk: '1', '2', '3', '1', '2', '4' in '2'.

Sestavi tudi testni program, s katerim preveriš delovanje funkcije!

Obračamo, obračamo, -omačarbo (datoteke)

Sestavi metodo

```
obrni(imeVhod, imeIzhod)
```

ki tekstovno datoteko, katere ime je v `imeVhod` prepiše na novo datoteko z imenom, kot ga določa `imeIzhod`. Pri tem vsako besedo, ki se začne z -, zapiše v obratnem vrstnem redu (brez znaka -) in odstrani odvečne presledke. Besede so med seboj ločene s presledki (enim ali več), odvečni presledki se lahko pojavijo tudi na začetku ali koncu vrstice. Če vhodne datoteke ni, na izhodno datoteko napiši 42 vrstic z besedilom "Tudi pri težjih problemih smo že odpovedali!".
Primer:

Vhodna datoteka	*Izhodna datoteka*

`Peter  -spet zamuja.`
`-Lep    dan.`
`Konec`
	`Peter  tesp zamuja.`
	`peL dan.`
	`Konec`

Iz CSV v HTML (datoteke)

Datoteke CSV (Comma Separated Values) so znakovne datoteke, na katerih hranimo podatke zapisane v več vrsticah, v posamezni vrstici pa so ti ločeni z vejicami. Sestavi metodo, ki datoteko CSV prepiše na novo znakovno datoteko, pri tem pa podatke oblikuje kot tabelo v HTML-ju (glej primer). Vrstice, ki niso ustrezne (torej sestavljene iz treh nizov, ločenih z vejicami), enostavno spusti.

Metoda naj za parametra dobi imeni obeh datotek.

Vhodna datoteka	*Izhodna datoteka*

`1000,Ljubljana,4.15`
 	
`2000,  Maribor,7.128`
`4000,8.89`
`1234,Mengeš,1.23`
 	
`200`
	`<table>`
	`  <tr>`
	`  <td>1000</td>`
	`  <td>Ljubljana</td>`
	`  <td>4.15</td>`
	`  </tr>`
	`  <tr>`
	`  <td>2000</td>`
	`  <td>  Maribor</td>`
	`  <td>7.128</td>`
	`  </tr>`
	`  <tr>`
	`  <td>1234</td>`
	`  <td>Mengeš</td>`
	`  <td>1.23</td>`
	`  </tr>`
	`</table>`

Števke so delitelji (osnove)

Napiši metodo, ki sprejme cela števila *a*, *b* in *k* ter poišče vsa cela števila iz intervala [*a*, *b*], v katerih vsaka neničelna števka deli število *k*. Metoda naj vrne niz v eni od naslednjih dveh oblik.

```
Cela števila iz intervala [10, 20], v katerih vsaka neničelna
števka deli 666, so: 10, 11, 12, 13, 16, 19, 20.

V intervalu [20, 40] ni celih števil, v katerih vsaka neničelna
števka deli 5.
```

Za šumnike, pravilno rabo predlogov s/z ter dvojine/množine ni treba skrbeti.

Prepis (datoteke)

Sestavi metodo

```
prepis(imeVhod, imeIzhod)
```

ki tekstovno datoteko, katere ime je v `imeVhod` prepiše na novo datoteko z imenom, kot ga določa `imeIzhod`. Pri tem prazne vrstice izpusti, preostale pa oštevilči. Primer:

Vhodna datoteka	*Izhodna datoteka*
`Danes je lep dan.`	`1: Danes je lep dan.`
`Druga vrstica.`	**`2: Druga vrstica.`**
	`3: Ta ni prazna.`
`Ta ni prazna.`	**`4: Zadnja!`**
`Zadnja!`	

Če imena izhodne datoteke ne podamo, naj bo to »izhod.txt«. Če izhodna datoteka obstaja že od prej, naj metoda takoj sproži izjemo (in ne prepisuje ničesar!)

Sestavi tudi ustrezni testni program in testne datoteke!

Osamljeni šumniki (osnove)

Sestavi funkcijo `OsamljeniSumniki`, ki prešteje osamljene šumnike v danem nizu. Šumniki so male ali velike črke č, š, ž. Šumnik je osamljen, če ni sosed kakega drugega šumnika.

Palindromska števila (osnove)

Sestavi metodo, ki izpiše vsa palindromska števila med a in b (parametra). Število je palindromsko, če je enako svojemu obratu. Npr. 123 in 59 nista palindromski števili, 838 in 1551 pa sta palindromski. Program naj rezultat izpiše v eni od naslednjih oblik:

```
Palindromska števila med 90 in 130 so 99, 101, 111, 121.

Med 12 in 21 ni palindromskih števil.
```

Družabne števke (osnove)

Sestavi funkcijo `DruzabneStevke`, ki prešteje družabne števke v danem nizu. Števka je družabna, če je soseda kake druge števke. V nizu "bal123i4e7e12bed42" je sedem družabnih števk: '1', '2', '3', '1', '2', '4' in '2'.

Sestavi tudi testni program, s katerim preveriš delovanje funkcije!

Masa (OOP)

Maso stvari želimo predstaviti z razredom, v katerem podamo podatek o količini mase in merski enoti,v kateri je podana ta količina mase. Masa je **celo** število, dovoljene merske enote pa so nizi "kg", "dag", "funt" ali "g". Sestavi razred `Masa`, ki se obnaša tako, kot je predpisano:

1. Ima lastnost `enota`. Enota mora biti obvezno ena od dovoljenih. Če enoto spremenimo (na eno od dovoljenih), se mora količina mase seveda (do zaokrožitvene napake) ohraniti.
2. Ima lastnost `koliko`. Če jo poskusimo nastaviti na negativno vrednost, naj se sproži izjema.
3. Vsebuje metodo `faktor`, ki za dano mersko enoto vrne, koliko je ta enota izražena v gramih.
4. Opis v obliki niza je oblike `"12.3 dag"` (ne glede na enoto, ki je uporabljena za objekt)
5. Objekte tipa `Masa` lahko seštevamo. Pri tem dobimo enoto levega operanda.
6. Objekte tipa `Masa` lahko množimo z leve ali desne s pozitivnim celim številom. Če jih množimo z 0 ali neg. številom sproži ustrezno izjemo!
7. Vsebuje metodo `najvecja_masa`, ki vrne tabelo vseh največjih mas iz dane tabele objektov tipa `Masa`. Pozor, če katero od vrnjenih mas potem spremenimo, se ustrezna masa v dani tabeli ne sme spremeniti.
8. Vsebuje metodo `sestej`, ki vrne novo maso, ki je vsota vseh mas dane tabele objektov tipa `Masa`, izražena v najbolj pogosti enoti.

Sestavi tudi program, kjer

- ustvariš tabelo mas velikosti 20 in jo v celoti simetrično napolniš z naključnimi masami naključnih mer tako, da so enaki prvi in zadnji element, drugi in predzadnji, …
- izpišeš maso, ki je vsota mas te dane tabele
- Izvedi metodo `najvecja_masa` in za 3x povečaj prvi element te tabele
- Ponovno izpišeš maso, ki je vsota mas te dane tabele

Trojisko (OOP)

Sestavi razred `Trojisko`, v katerem hranimo števila v trojiškem zapisu kot nize z obveznim predznakom. Tako število 13 zapišemo kot niz '+111', število -5 pa kot niz '-12'. Razred naj vsebuje:

- konstruktor `Trojisko()`, ki ustvari nov objekt, v katerem hranimo število z vrednostjo 3.
- konstruktor `Trojisko(vrednost)`, ki ustvari nov objekt, ki predstavlja število `vrednost`.
- konstruktor `Trojisko(a)`, ki ustvari kopijo trojiškega objekta `a`.
- metodo `vrednost()`, ki vrne vrednost tega objekta kot število v desetiškem zapisu.
- metodo `zmnozi(a)`, ki zmnoži trenutni objekt z objektom `a` in vrne rezultat.
- metodo`,` ki vrne niz, v katerem je zapisana desetiška vrednost števila.

Kvader (OOP)

Sestavi razred `Kvader`. Objekt razreda Kvader naj ima lastnosti `visina`, `globina` in `širina`. Seveda morajo biti vsi trije podatki pozitivni! Globina naj bo nespremenljiva, torej je naknadno ne moremo spreminjati, ostali dve količini pa lahko. Če poskusimo kateri koli podatek nastaviti na negativno število, sproži izjemo!

Razredu dodaj:

- Metodo `povrsina`, ki vrne površino objekta (kvadra),
- metodo `volumen`, ki vrne volumen kvadra, podanega kot parameter
- Bralno lastnost `ploščina`, ki pove ploščino osnovne ploskve kvadra (globina x širina),
- Možnost množenja s **pozitivnim** celoštevilskim faktorjem (z leve in desne). Pri tem dobimo nov kvader, ki ima za faktor večje vse tri mere.
- Ustrezno metodo za pretvarjanje v niz
- Metodo `najnizji_kvader`, ki poišče in vrne kvaderj z najmanjšo višino v tabeli objektov tipa `Kvader`. Če je več najnižjih kvadrov, naj vrne tak kvader, kot je tisti, ki se v tabeli pojavi **zadnji**. Pozor, če vrnjeni kvader potem spremenimo, se ustrezni kvader v tabeli ne sme spremeniti.

VALJ (OOP)

Sestavi razred `Valj`. Objekt razreda `Valj` naj ima lastnosti `visina` in `polmer`. Seveda mora biti oba podatka pozitivna! Polmer naj bo nespremenljiv, torej ga naknadno ne moremo spreminjati, višino pa lahko. Če poskusimo kateri koli podatek nastaviti na negativno število, sproži izjemo!

Razredu dodaj:

- Metodo `povrsina`, ki vrne površino objekta (valja),
- Statično metodo `volumen`, ki vrne volumen valja, podanega kot parameter
- Bralno lastnost `obseg`, ki pove obseg osnovnega kroga valja,
- Možnost množenja s **pozitivnim** celoštevilskim faktorjem (z leve in desne). Pri tem dobimo nov valj, ki ima enak osnovni krog in za faktor večjo višino.
- Ustrezno metodo za pretvarjanje v niz
- Metodo `najnizji_valj`, ki poišče in vrne valj z najmanjšo višino v tabeli objektov tipa `Valj`. Če je več najnižjih valjev, naj vrne tak valj, kot je tisti, ki se v tabeli pojavi **zadnji**. Pozor, če vrnjeni valj potem spremenimo, se ustrezni valj v tabeli ne sme spremeniti.

Sestavi ustrezno okolje testiranja iskanja najnižjega valja.

Dolžina (OOP)

Dolžino želimo predstaviti z razredom, v katerem podamo podatek o količini in merski enoti te razdalje. Količina je celo število, merska enota pa niz "m", "dm", "cm" ali "mm". Sestavi razred `Dolzina`, ki se obnaša tako, kot je predpisano:

1. Ima lastnost `enota`, ki jo lahko le beremo. Enota mora biti obvezno ena od dovoljenih, torej niz "m", "dm", "cm" ali "mm" (druge enote niso dovoljene).
2. Ima lastnost `koliko`. Če jo poskusimo nastaviti na negativno vrednost, naj se sproži izjema.
3. Opis v obliki niza je oblike `"12.3 cm"` (ne glede na enoto)
4. Objekte tipa `Razdalja` lahko z leve ali desne množimo s celimi števili. Če jih množimo z negativnim številom, naj se sproži izjema.
5. Vsebuje metodo `najkrajsa_razdalja`, ki poišče in vrne najkrajšo razdaljo iz tabele objektov tipa `Razdalja`. Če je najkrajših razdalj več, naj vrne prvo izmed njih. Če vrnjeno razdaljo spremenimo, se ustrezna razdalja v tabeli ne sme spremeniti.

Sestavi tudi testni program za metodo `najkrajsa_razdalja`!

Naloge s sistema ProjektTOMO

Tabele II

Drugi največji

1. podnaloga

Napiši funkcijo `drugi_najvecji(tab)`, ki vrne drugi največji element v tabeli `tab`. V primeru, da je tabela prazna, vrni None. Če pa ima tabela samo en element, prav tako vrni None. Predvidevaš lahko, da so elementi v tabeli različni.

Nalogo reši tako, da ne spremeniš tabele. Prav tako ne smeš uporabiti funkcije max!

2. podnaloga

Napiši funkcijo `drugi_najvecji1(tab)`, ki vrne drugi največji element v tabeli `tab`. Kot prej, v primeru, da je tabela prazna, ali če ima tabela samo element, vrni None. Predvidevaš lahko, da so elementi v tabele različni.

Nalogo reši z uporabo metod na tabelah in funkcije `max`, a brez uporabe zank.

Popolna števila

1. podnaloga

Število je popolno, če je enako vsoti vseh svojih pravih deliteljev (razen samega sebe, ampak vključno z 1). Primer popolnega števila je 28, ki ga delijo 1, 2, 4, 7 in 14. Če števila seštejemo, spet dobimo število 28. Recimo, da 1 **ni** popolno število.

Napiši funkcijo `popolna(sez_stevil)`, ki sprejme tabelo števil in vrne tabelo števil, ki vsebuje samo takšna števila iz `sez_stevil`, ki so popolna.

---

aN ebor

Na Najboljši TV je prava zmešnjava (kar za najnovejšo TV v Sloveniji niti ni tako nenavadno). Le še 5 minut do začetka oddaje *Izbiramo najlepšega bika Slovenije*. Težav pa cel kup.

1. podnaloga

Prva težava je, da je na vseh tabelah naveden napačni vrstni red imen bikov. Režiser na vsak način poskuša prepričati voditeljski par, da ni nič narobe, le od spodaj navzgor naj bereta razvrstitev. A kljub temu, da sta (po njunem mnenju) daleč najbolj inteligentna TV voditelja v Sloveniji, se bojita, da bo to zanju pretežka naloga.

Zato moraš hitro napisati funkcijo `obrni_tabelo(tabela_bikov)`, ki bo vrnila novo tabelo teh imen v obratnem vrstnem redu.

Primer:

```
>>> obrni_tabelo(['Bukso', 'Hitri', 'Šeko', 'Lisko'])
['Lisko', 'Šeko', 'Hitri', 'Bukso']
```

Pozor, ker moraš vrniti novo tabelo, uporaba metode `reverse` ne bo primerna! Prav tako ni dovoljena uporaba rezanja (podtabel).

2. podnaloga

Tabela bikov je sedaj urejena. A kaj, ko so se na tabelo prikradle tudi krave (na TV sumijo, da gre za podtalno delovanje tet iz ospredja). Zato se bodo pritožili na MCSodišče. A ura je neizprosna, ustrezno tabelo potrebujejo takoj. Na srečo je najomiljeni voditelj z vso svojo avtoriteto ugotovil, da je krave zlahka prepoznati, saj se vsa njihova imena končajo z 'a'. Ker želijo ohraniti dokaze, morajo tabelo s pomešanimi kravami ohraniti. Zato sestavi funkcijo `odstrani_krave(tabela)`, ki vrnila novo tabelo teh imen brez krav. Biki morajo ohraniti svoj vrstni red.

Primera:

```
    >>> odstrani_krave(['Liska', 'Šeko', 'Hitri', 'Buksa'])
    ['Šeko', 'Hitri']
    >>> odstrani_krave(['Bukso', 'Hitri', 'Šeko', 'Lisko'])
    ['Bukso', 'Hitri', 'Šeko', 'Lisko']
```

3. podnaloga

Težav pa še ni konec. Vse to prekladanje tabel je povzročilo, da so sedaj določena imena v seznamu navedena večkrat. Zato napiši funkcijo `odstrani_ponovljene(tabela)`, ki vrne novo tabelo, ki ne vsebuje ponovitev imen. Pri ponovljenih imenih ohrani le prvo pojavitev.

Primer:

```
    >>> odstrani_ponovljene(['Lisko', 'Šeko', 'Šeko', 'Šeko', 'Hitri', 'Bukso', 'Lisko'
])
    ['Lisko', 'Šeko', 'Hitri', 'Bukso']
```

Namig: Kaj vrne `'bla' in ['blu', 'ble', 'bla', 'blu']`?

Čete

Četa je najdaljše možno strnjeno podzaporedje v dani tabeli števil z določeno lastnostjo. Če je zaporedje naraščajoče (vsak naslednji element podzaporedja je *večji*), govorimo o naraščajočih četah, če je nepadajoče (torej je vsak naslednji element večji ali enak) o nepadajočih četah, če je vsak element manjši, imamo padajočo četo, kadar je vsak naslednji element manjši ali enak pa nenaraščajočo četo ...

Celotna tabela je tako sestavljena iz več zaporednih čet. Predpostavili bomo, da bomo vedno upoštevali čete iste vrste. Tako v tabeli `[2,5,7,1,45,7,7,15]` najdemo kar 4 naraščajoče čete ((`2,5,7`), (`1,45`), (`7`) in (`7,15`)), tabela `[3,5,8,10,12]` pa je sestavljena iz ene same naraščajoče čete.

Po drugi strani pa v tabeli `[2,5,7,1,45,7,7,15]` najdemo kar 6 padajočih čet ((`2`), (`5`), (`7, 1`), (`45, 7`), (`7`) in (`15`)) ali 5 nenaraščajočih čet ((`2`), (`5`), (`7, 1`), (`45, 7, 7`) in (`15`))

1. podnaloga

Sestavite funkcijo `narascajoc(tabela)`, ki preveri, ali elementi tabele `tabela` tvorijo naraščajoče zaporedje. Pomagajte si s primeri:

```
>>> narascajoc([])
True
```

```
>>> narascajoc([1, 2, 5, 8, 12, 35])
True
>>> narascajoc([3, 5, 5])
False
>>> narascajoc([2, 6, 4, 8, 9, 6])
False
```

2. podnaloga

Sestavi funkcijo `st_nepadajocih_cet(tabela)`, ki v dani številski tabeli prešteje, iz koliko nepadajočih čet je sestavljena. Nepadajoča četa je najdaljša možna nepadajoče urejena podtabela, ali še enostavneje: dokler se zaporedni elementi tabele ne zmanjšujejo, so v isti četi.

```
>>> st_nepadajocih_cet([])
0
>>> st_nepadajocih_cet([6])
1
>>> st_nepadajocih_cet([1, 3, 4, 7, 23, 56, 81])
1
>>> st_nepadajocih_cet([1, 2, 2, 6, 2, 3, 7, 5, 2, 3, 8])
4
```

3. podnaloga

Sestavi funkcijo `st_narascajocih_cet(tabela)`, ki v dani tabeli števil prešteje, iz koliko naraščajočih čet je sestavljena. Naraščajoča četa je najdaljši možen naraščajoče urejen podtabela, ali še enostavneje: dokler zaporedni elementi tabele naraščajo, so v isti četi.

```
>>> st_narascajocih_cet([])
0
>>> st_narascajocih_cet([6])
1
>>> st_narascajocih_cet([1, 3, 4, 7, 23, 56, 81])
1
>>> st_narascajocih_cet([1, 2, 2, 6, 2, 3, 7, 5, 2, 3, 8])
5
```

4. podnaloga

Sestavite funkcijo `dolzina_najdaljse_narascajoce_cete(tabela)`, ki ugotovi, kakšna je dolžina najdaljše čete v tabeli `tabela`.

```
>>> dolzina_najdaljse_narascajoce_cete([])
0
>>> dolzina_najdaljse_narascajoce_cete([2, 5, 7, 10])
4
```

```
>>> dolzina_najdaljse_narascajoce_cete([1, 3, 6, 3, 8, 8, 10, 12])
3
```

5. podnaloga

Sestavite funkcijo `narascajoce_cete(tabela)`, ki vrne tabelo vseh naraščajočih čet, ki tvorijo tabelo `tabela`.

```
>>> narascajoce_cete([1, 3, 6, 3, 8, 8, 10, 12])
[[1, 3, 6], [3, 8], [8, 10, 12]]
```

Polžje dirke

Butalci vsako leto priredijo polžje dirke. Pomagajte jim pri organizaciji.

1. podnaloga

Tekmovalnega polža opišemo s tremi parametri: starostjo, težo in velikostjo. Faktor njegove tekmovalne sposobnosti opisuje takšna formula:
`najboljsi(polzi)`, ki sprejme tabelo opisov polžev `polzi` (tabela trojčkov `[starost, teža, velikost]`) in vrne faktor za najboljšega polža, na dve decimalni mesti natančno.

```
>>> najboljsi([[5, 6, 72], [4, 17, 77], [14, 21, 22], [17, 36, 64], [13, 29, 23]])
16.39
```

2. podnaloga

Butalski polži se premikajo samo v štirih smereh: sever, jug, vzhod in zahod, kar v koordinatnem sistemu pomeni gor, dol, desno in levo. V vsakem koraku se premaknejo za eno enoto, lahko pa tudi mirujejo. Njihove premike zato lahko opišemo s tabelo, ki vsebujejo poljubno zaporedje črk `'S'`, `'J'`, `'V'`, `'Z'` in `'M'`.

Napišite funkcijo `pot(premiki)`, ki sprejme tabelo, ki opisuje premike polža in pove, kako dolgo pot je opravil polž polž in kako daleč (zračne razdalje) je prilezel. Rezultat naj bo dan v obliki para celo in dec. število, kjer je slednje zaokroženo na dve decimalni mesti.

```
>>> pot(['M', 'J', 'V', 'S', 'S', 'M'])
(4, 1.41)
```

3. podnaloga

Po končanem tekmovanju so znane poti vseh polžev. Velja, da je zmagovalec tisti polž, ki je prilezel najdlje od začetka (seveda pa si lahko prvo mesto deli več zmagovalcev, če so prelezli isto razdaljo). Pri tem seveda doseženo razdaljo računamo zaokroženo, kot pri zgornji nalogi! Napišite funkcijo `zmagovalci(poti)`, ki sprejme tabelo poti vseh polžev `poti` ter izračuna, koliko polžev je osvojilo prvo nagrado.

```
>>> zmagovalci([['S', 'Z', 'S'], ['M', 'Z', 'V'], ['J', 'J', 'Z'], ['J', 'J', 'Z'], ['S
', 'S', 'M']])
3
```

Namig: pomagaj si s funkcijo iz prejšnje naloge.

Bančni račun

Pri reševanju te naloge (razen prvih dveh) uporabljamo tudi "rezanje" (slicing). Npr. s tab[1:] dobimo tabelo, ki je enaka tabeli tab, le da nima prvega elementa.

1. podnaloga

V tabeli imamo podatke o prilivih in odlivih s tekočega računa. Pozitivna števila predstavljajo priliv (polog denarja), negativna pa dvig. Vsak element tabele predstavlja en dan. Če na nek dan ni prilivov ali dvigov, je vrednost v tabeli 0. Privzemite, da je na začetku stanje na računu 0.

Sestavite funkcijo `koncno_stanje(spremembe)`, ki iz dane tabele prilivov in odlivov izračuna končno stanje.

Pri rešitvi ne uporabi zanke neposredno v kodi te funkcije.

2. podnaloga

Sestavite funkcijo `stanja(spremembe)`, ki iz dane tabele prilivov in odlivov ustvari tabelo vmesnih stanj na računu. Privzemite, da je na začetku stanje na računu 0. To naj bo prva "sprememba" v tabeli

```
>>> stanja([10, -5, 20, -6])
[0, 10, 5, 25, 19]
```

3. podnaloga

Sestavite funkcijo `ekstrema(spremembe)`, ki pri danih spremembah stanja poišče vrednosti, ko je bilo stanje na računu najnižje oziroma najvišje. Pri tem seveda zanemari začetno stanje 0 (uporabi rezanje) in prepostavi, da je prišlo do vsaj ene spremembe

Stanji naj vrne v obliki nabora.

```
>>> ekstrema([10, -5, 20, -6])
(5, 25)
```

Rešitev poišči brez uporabe zank neposredno v kodi te funkcije. *Namig*: Pomagaj si s funkcijami iz prejšnjih nalog.

4. podnaloga

Sestavite funkcijo `kdaj_ekstrema(spremembe)`, ki pri danih spremembah stanja poišče število dni od začetka, ko je bilo stanje na računu najnižje oziroma najvišje. Rezultat naj vrne v obliki nabora.

```
>>> kdaj_ekstrema([10, -5, 20, -6])
(2, 3)
```

Rešitev poišči brez uporabe zank neposredno v kodi te funkcije. *Namig*: Pomagaj si s funkcijami iz prejšnjih nalog ter z metodo `index`.

5. podnaloga

Sestavite funkcijo `razlika(spremembe, i, j)`, ki poišče razliko stanj med dnevoma z indeksom `i` in `j`. Na dan 0 je bilo seveda stanje 0.

```
>>> razlika([10, -5, 20, -6], 1, 3)
```

```
15
>>> razlika([10, -5, 20, -6], 1, 2)
-5
```

Predpostavite lahko, da je `i` manjši ali enak `j`. Rešitev poišči brez uporabe zank neposredno v kodi te funkcije. Tu potrebuješ rezanje!

6. podnaloga

Sestavite funkcijo `najvecja_razlika(spremembe)`, ki poišče največji relativni priliv. Z drugimi besedami, poišče največjo vrednost, ki jo zavzame vsota poljubne strnjene podtabele. Tako na primer `najvecja_razlika([10, -13, 3, 20, -2, 5])` vrne 3 + 20 - 2 + 5 = 26. Če je tabela `spremembe` prazna, naj funkcija vrne `None`.

Namig: verjetno bo potrebno preveriti vse možne podtabele!

Pascalov trikotnik

1. podnaloga

Napišite funkcijo `pascal(n)`, ki vrne tabelo prvih `n` vrstic Pascalovega trikotnika. Vrstice naj bodo dane kot tabele. Če je n enak 0, naj funkcija vrne prazno tabelo.

```
>>> pascal(6)
[[1], [1, 1], [1, 2, 1], [1, 3, 3, 1], [1, 4, 6, 4, 1], [1, 5, 10, 10, 5, 1]]
```

Kamionisti

V Republiki Banana zaradi utrujenosti in pomanjkanja koncentracije kamionisti (slovensko - *vozniki kamionov* Zato pet minut za učenje slovenčine) vse pogosteje povzročajo nesreče. Na vladi so se odločili, da bodo sprejeli zakone, ki bodo uredili problematiko preutrujenih kamionistov. Najprej pa morajo analitiki preučiti njihove navade. V ta namen so pridobili podatke o kamionistih. Za vsakega imajo podano tabelo, v kateri so shranjene prevožene razdalje po posameznih dnevih. Število 0 pomeni, da je kamionist tisti dan počival.

Primer: Tabela

```
[350, 542.5, 0, 602, 452.5, 590.5, 0, 248]
```

pomeni, da je kamionist prvi dan prevozil 350 km, drugi dan je prevozil 542.5 km, tretji dan je počival itd.

1. podnaloga

Sestavite funkcijo `pocitek_in_povprecje(voznja)`, ki kot argument dobi zgoraj opisano tabelo (prevožene razdalje po posameznih dnevih) in vrne par števil. Prvo število naj bo število dni, ko je kamionist počival. Drugo število naj bo na eno decimalko zaokrožena (`round(x, 1)`) povprečna dnevna prevožena razdalja, pri čemer upoštevamo samo tiste dneve, ko ni počival. Predpostavite lahko, da kamionist vsaj en dan ni počival. Primer:

```
>>> pocitek_in_povprecje([200, 300, 0, 100])
(1, 200.0)
```

Kamionist je torej enkrat počival. Ob dnevih, ko ni počival, pa je v povprečju prevozil 200 km.

2. podnaloga

Napišite funkcijo `primerjaj(prvi, drugi)`, ki primerja vožnjo dveh kamionistov. Funkcija kot argumenta dobi dve enako dolgi tabeli `prvi` in `drugi`, ki opisujeta vožnjo dveh kamionistov, ki sta se podala na isto pot. Funkcije naj sestavi in vrne novo tabelo, v kateri je za vsak dan zapisano, kdo je **do tega dne skupaj** prevozil večjo razdaljo: `1`, če je bil to 1. kamionist; `2`, če je bil to 2. kamionist in `0`, če sta oba prevozila enako razdaljo. Primer:

```
>>> primerjaj([200, 300, 100], [200, 200, 300])
[0, 1, 2]
>>> primerjaj([500, 100, 100], [100, 200, 200])
[1, 1, 1]
```

3. podnaloga

Vlada je uzakonila naslednja pravila: * Povprečna prevožena razdalja v treh zaporednih dneh ne sme biti več kot 500 km. (Štejemo tudi dneve, ko je kamionist počival.) * Kamionist ne sme brez počitka voziti več kot 5 dni v zaporedju.

Ker so ugotovili, da so kamionisti prevečkrat svoje zaporedje skrajšali na dva dni in se s tem dejansko izognili pravilu, so uzakonili še: * Zadnji dan ne sme voziti več kot 500 km. * Povprečje zadnjih dveh dni ne sme biti več kot 500 km.

Sestavite funkcijo `po_pravilih(voznja)`, ki vrne `True`, če se je kamionist držal predpisov, in `False`, če se jih ni.

```
>>> po_pravilih([50, 200, 300, 20, 100, 60])
False
>>> po_pravilih([600, 200, 0, 300, 300, 0, 600, 600, 400])
False
>>> po_pravilih([600, 600, 0, 600, 600])
False
>>> po_pravilih([600, 600, 0, 200, 600])
False
```

4. podnaloga

Podatke za več kamionistov so analitiki združili v eno samo tabelo. Zgled take tabele:

```
vsi_skupaj = [
    [50, 200, 300, 20, 100, 60],
    [600, 600, 0, 600, 600, 0, 500, 500],
    [600, 200, 0, 300, 300, 0, 600, 600, 400]
]
```

Sestavite funkcijo `preveri_vse(seznam_vozenj)`, ki dobi kot argument tabelo kot je zgoraj in vrne tabelo logičnih vrednosti, ki za vsakega kamionista pove, če je vozil po predpisih. Primer:

```
>>> preveri_vse(vsi_skupaj)
```

```
[False, True, False]
```

5. podnaloga

Ko so vožnje vseh kamionistov že tako lepo združene, so se analitiki spravili računati še to, kakšno je navečje število voženj, ki jih je nekdo opravil, kakšno je največje število dni počitka ter kakšno je največje in kakšno najmanjše povprečno število dnevno prevoženih kilometrov (povprečje tako, kot pri prvi nalogi - štejemo le dneve vožnje). Pri tem upoštevamo samo tiste kamioniste, ki so vozili po pravilih! Sestavite funkcijo `analitika(seznam_vozenj)`, ki dobi kot argument ustrezno tabelo in vrne nabor 4 vrednosti: največ voženj, največ dni počitka, maksimalno povprečno dnevno prevoženih in minimalno povprečnodnevno prevoženih kilometrov. Predpostavi, da imamo vsaj enega kamionista, ki je vozil po predpisih! Primer (podatki iz prejšnje naloge):

```
>>> analitika(vsi_skupaj)
(6, 2, 566.7, 566.7)
```

Tabele in nizi

Ukleščeni nizi

1. podnaloga

Sestavite funkcijo `prestej(niz)`, ki prešteje, koliko zašiljenih oklepajev tj. `'<'` in `'>'` je v nizu `niz`. Primer:

```
>>> prestej('<abc>>')
3
>>> prestej('a>>  x<Y>x  <<b')
6
>>> prestej('Koliko oklepajev > in < je v tem nizu?')
2
```

Pri tem ne smeš uporabiti vgrajene metode `count`

2. podnaloga

Niz imenujemo *ukleščen*, če se začne z predklepajem `'<'`, konča z zaklepajem `'>'`, vmes pa ni nobenega znaka za zašiljene oklepaje. Na primer, niz `'<Zunaj sije sonce.>'` je ukleščen niz.

Sestavite funkcijo `uklescen(niz)`, ki preveri, ali je niz `niz` ukleščen. Primer:

```
>>> uklescen('<Zunaj sije sonce.>')
True
>>> uklescen('<   <3    >')
False
>>> uklescen('Zunaj sije sonce.')
False
```

Znaš nalogo rešiti brez zanke v telesu funkcije `uklescen`? Seveda pa si lahko pomagaš s funkcijo prestej iz prejšnje podnaloge.

3. podnaloga

Sestavite funkcijo `sklop(niz1, niz2)`, ki prejme dva ukleščena niza in vrne ukleščeni niz, ki predstavlja njun sklop, ki je tudi ukleščen niz.

Primer:

```
>>> sklop('<123>', '<456>')
'<123456>'
>>> sklop('<muca>', '<copatarica>')
'<mucacopatarica>'
```

4. podnaloga

Sestavite funkcijo `razlomi(niz, s)`, ki kot argumenta prejme ukleščeni niz `niz` in tabelo nenegativnih celih števil `s` ter vrne tabelo ukleščenih nizov, ki imajo dolžine, naštete v tabele `s`, in skupaj tvorijo niz `niz`.

Primer:

```
>>> razlomi('<Zunaj sije sonce.>', [2, 7, 8])
['<Zu>', '<naj sij>', '<e sonce.>']
>>> razlomi('<muca copatarica>', [4, 0, 11])
['<muca>', '<>', '< copatarica>']
```

Predpostavite lahko, da bo vsota števil v tabeli `s` enaka dolžini ukleščenega niza `niz` (če ne štejemo zašiljenih oklepajev).

Sprehodi

1. podnaloga

Sestavite funkcijo `celostevilski(sprehod)`, ki sprejme niz, ki predstavlja sprehod po celih številih, in vrne število, v katerem se sprehod konča.

Sprehod po celih številih se začne v številu 0, predstavimo pa ga z nizem, sestavljenim iz znakov `+` in `-`, ki pomenita pomik za ena v desno in ena v levo na številski osi. Na ostale znake v nizu se ne oziramo. Zgled:

```
>>> celostevilski('++--#---@++--+-++')
-1
```

2. podnaloga

Sestavite funkcijo `ravninski(sprehod)`, ki sprejme niz, ki predstavlja zaporedje korakov v ravnini, in vrne točko (par dveh števil), v kateri se sprehod konča.

Sprehod po ravnini se začne v izhodišču, predstavimo pa ga z nizem, sestavljenim iz črk `S`, `J`, `V` ali `Z`, ki predstavljajo smeri korakov (sever, jug, vzhod, zahod). Na ostale znake v nizu se ne oziramo. Zgled:

```
>>> ravninski('#ZZZS#')
(-3, 1)
```

3. podnaloga

Sestavite funkcijo `hitri(tek)`, ki sprejme niz, ki predstavlja zaporedje korakov in skokov v ravnini, in vrne točko, v kateri se tek konča.

Tek po ravnini se začne v izhodišču, predstavimo pa ga, tako kot sprehod, z nizem, sestavljenim iz črk `S`, `J`, `V` ali `Z`, ki predstavljajo smeri korakov (sever, jug, vzhod, zahod).

Poleg tega **lahko** tek vsebuje tudi števke od `1` do `9`, ki povedo, koliko dolg naj bo naslednji korak. Tako niz `5S` pomeni skok na sever, dolg 5 korakov. Privzamete lahko, da zaporednih števk v nizu ni, ter da se na ostale znake v nizu ne oziramo. Zgled:

```
>>> hitri('3S4ZJ')
```

```
(-4, 2)
```

Stopnice

1. podnaloga

V laboratoriju FMF-P1 jim je uspel tehnološki presežek. Sestavili so robota, ki zlahka hodi po stopnicah. Konkurenca na vsak način poskuša diskreditirati ta dosežek. Zato sestavljajo različne čudne kombinacije stopnic in čakajo, kdaj bo robot pri hoji padel, saj lahko robot prehodi le stopnico, ki je visoka največ 20 cm. Ampak spretni študenti praktiki so robota opremili z merilnim sistemom, ki zmeri višino posamezne stopnice, in napisali funkcijo (opaziš, da je lepo skrbno komentirana, kot vsa koda, ki jo pišejo v tem laboratoriju).

```
def koliko_stopnic(stopnice):
    """Vrne število stopnic, ki jih lahko prehodi robot."""
    katera = 0  # indeks stopnice
    while katera < len(stopnice):
        v = stopnice[katera]            # za vsako stopnico vzamemo njeno višino
        if v > 20:   # ce je visina stopnice previsoka
            return katera  # koliko stopnic lahko prehodim (ker začnemo z 0, bo OK!)
        katera += 1   # naslednja stopnica
    # prehodili smo vse!
    return len(stopnice)
```

Na osnovi te kode sestavi funkcijo `kako_visoko_pridem(stopnice)`, ki kot argument prejme tabelo višin stopnic `stopnice` rezultat, ki ga vrne, pa pove, na kakšni višini bo robot po koncu hoje.

2. podnaloga

Na FRI-P1 pa jim je uspelo izdelati napravo, ki zmoti merilni sistem robota tako, da meri višine stopnic, merjene od tal (**ne od prejšnje stopnice**). Ampak praktiki s FMF se ne dajo. Poleg tega, da robota izpopolnijo, da ima sedaj nastavljivo maksimalno višino koraka (a žal se ta med hojo ne da spreminjati), na osnovi prejšnje naloge napišejo funkcijo `kako_visoko(stopnice, korak_robota)`, ki kot argument prejme tabelo višin stopnic `stopnice`, merjenih na "FRI način" in kako visoko se lahko robot premakne v enem koraku, rezultat, ki ga vrne, pa spet pove, kako visoko bo robot priplezal.

Primer:

```
>>> kako_visoko([5, 25, 45, 50, 76, 80, 81], 20)
50
```

Škatle

Lojzku je dolgčas, zato se igra z velikimi praznimi škatlami, ki so v skladišču, v katerem dela. Dimenzije škatel so shranjene v tabeli trojic. Na primer, trojica `(50, 100, 100)` predstavlja škatlo, ki je visoka 50 cm ter široka in dolga 100 cm.

S trojicami delamo tako kot s tabelami, le spreminjati ne moremo elementov. Torej nam `skatla[0]` da prvi element iz trojice `skatla`. Seveda pa lahko trojko tudi "razpakiramo"

```
vis, sir, dol = skatla
```

če je `skatla` neka trojka.

1. podnaloga

Sestavite funkcijo `stolp(skatle)`, ki vrne višino najvišjega stolpa, ki ga lahko sestavimo iz škatel, ne da bi jih obračali. Pri tem ni treba paziti na stabilnost stolpa.

Primer:

```
>>>stolp([(50, 100, 100), (60, 30, 50), (40, 40, 40), (10, 30, 10)])
160
```

2. podnaloga

Sestavite funkcijo `najvisji_stolp(skatle)`, ki vrne višino najvišjega stolpa, ki ga lahko sestavimo iz škatel, če jih lahko obračamo. Pri tem še vedno ni treba paziti na stabilnost stolpa.

Primer:

```
>>> najvisji_stolp([(50, 100, 100), (60, 50, 50), (40, 40, 40)])
200
```

(prva škatla ima širino in dolžino večjo od širine, zato jo prevrnemo na bok.)

3. podnaloga

Sestavite funkcijo `gre_notri(skatla1, skatla2)`, ki vrne `True`, če škatlo `skatla1` lahko obrnemo tako, da gre v škatlo `skatla2`, torej da so dimenzije prve škatle strogo manjše od dimenzij druge škatle.

Primer:

```
>>> gre_notri((30, 40, 50), (40, 50, 60))
True
>>> gre_notri((30, 50, 40), (40, 50, 60))
True
>>> gre_notri((30, 60, 60), (40, 50, 60))
False
```

Stroški

1. podnaloga

Prevzeli smo podjetje `Vsi Enaki`. V skladu z imenom podjetja bi radi poskrbeli, da bodo imeli vsi delavci enako plačo. Ker plače seveda ne moremo jemati (prepir, jok, izsiljevanje in te reči), je naša naloga napisati funkcijo `enake_place(tab_plac)`, ki kot argument dobi tabelo s plačami v gljubih (saj vam je jasno, da se to ne dogaja na našem planetu), kot rezultat pa vrne, koliko gljubov bo potrebno dodatno zagotoviti.

Primer:

```
>>> enake_place([5, 8, 6, 4])
9
```

Znaš nalogo rešiti z vgrajenimi python funkcijami brez uporabe zank?

2. podnaloga

Sedaj vemo, koliko sredstev bomo potrebovali. A koliko se bo povečala plača posameznika? V tabeli imamo poleg plače še ime posameznega delavca.

Napiši funkcijo `imena_s_placami(place)`, ki sprejme tabelo tabel prikazano na spodnjem primeru in vrne tabelo tabel s podatki o tem, za koliko se posameznemu delavcu zviša plača. V izhodni tabeli naj bodo samo delavci, ki se jim plača zviša.

Primer:

```
>>> imena_s_placami([[5,"Brane"], [8,"Bine"], [6,"Janez"], [4,"Albert"]])
[[3,"Brane"],[2,"Janez"],[4,"Albert"]]
```

Tipkanje

Z RTK 2001.1.1

1. podnaloga

Predpostavimo, da lahko vse znake, ki jih želimo natipkati, razdelimo v dve skupini: nekatere tipkamo vedno z levo roko, druge pa vedno z desno.

Napisana je funkcija `dolzina_najdaljsega_podniza(vhod)`. Dopolni jo na mestih, označenih z ###, da bo iz danega niza izračunala in vrnila dolžino takega najdaljšega podniza, ki ga je mogoče v celoti natipkati z eno samo roko.

```
def dolzina_najdaljsega_podniza(vhod):
    """Izračuna dolžino najdaljšega podniza v vhodnem nizu."""
    trenutna_roka = None
    max_z_eno = 0
    st_s_trenutno = 0
    for crka in vhod:
        nova_roka = ###
        if nova_roka == trenutna_roka:
            # Z isto roko kot doslej natipkamo še en znak več.
            st_s_trenutno += 1
        elif st_s_trenutno > ###:
            # Zamenjamo roko. Mogoče smo z dosedanjo dosegli nov rekord.
            max_z_eno = st_s_trenutno
            st_s_trenutno = 1
        else:
```

```
            ###
        trenutna_roka = nova_roka
    ###
```

Na voljo imaš funkcijo `s_katero_roko(crka)`, ki zna za vsak znak povedati, v katero od navedenih dveh skupin spada. Če ga napišemo z levo, nam vrne niz `'leva'`, sicer pa niz `'desna'`.

Primeri:

Če se *a* in *e* tipkata vedno z levo roko, *i*, *o* in *u* pa vedno z desno, je pri nizu *aeiou* pravilni odgovor:

```
>>> dolzina_najdaljsega_podniza('aeiou')
3
```

Podniz *iou* lahko namreč natipkamo z desno roko.

Poglejmo si še nekaj primerov:

```
>>> dolzina_najdaljsega_podniza('aaeaiaoaua')
4
```

Podniz *aaea* lahko namreč natipkamo z levo roko.

```
>>> dolzina_najdaljsega_podniza('ouaeauo')
3
```

Podniz *aea* lahko namreč natipkamo z levo roko.

```
>>> dolzina_najdaljsega_podniza('uaoei')
1
```

Nobenih dveh zaporednih znakov ne moremo natipkati z isto roko.

```
>>> dolzina_najdaljsega_podniza('iieeoo')
2
```

V tem primeru lahko niz *ii* v celoti natipkamo z desno ali pa *ee* z levo ali pa *oo* z desno.

Lepšanje in šifriranje

1. podnaloga

Klodovik (papiga) in ne Klodvik (frankofonski kralj)/ bi rad zašifriral svoja besedila, da jih nepoklicane osebe ne bodo mogle prebrati. To stori tako, da najprej v besedilu vse male črke spremeni v velike in odstrani vse znake, ki niso črke. Klodvik vsa pomembna besedila piše v angleščini, zato bomo uporabljali angleško abecedo.

Na primer iz besedila `'Attack at dawn!'` dobi besedilo `'ATTACKATDAWN'`. Nato ga zapiše cik-cak v treh vrsticah, kot prikazuje primer:

```
A...C...D...
.T.A.K.T.A.N
..T...A...W.
```

Sestavite funkcijo `cik_cak(niz)`, ki vrne trojico nizov (torej `tuple`) in sicer prvo, drugo in tretjo vrstico v tem zapisu. Primer:

```
>>> cik_cak('Attack at dawn!')
('A...C...D...', '.T.A.K.T.A.N', '..T...A...W.')
```

2. podnaloga

Zašifrirano besedilo dobi tako, da najprej prepiše vse znake iz prve vrstice, nato vse znake iz druge vrstice in na koncu še vse znake iz tretje vrstice. V zgornjem primeru bi tako dobil `'ACDTAKTANTAW'`. Sestavite funkcijo `cik_cak_sifra(niz)`, ki dobi kot argument niz in vrne zašifrirano besedilo. Primer:

```
>>> cik_cak_sifra('Attack at dawn!')
'ACDTAKTANTAW'
```

3. podnaloga

Klodovik se zelo razjezi, ko dobi elektronsko pošto v takšni obliki:

```
Kar sva  si obljubljala    že leta,  si   želiva potrditi tudi   pred prijatelji in   c
elo

žlahto. Vabiva te na


     poročno slovesnost,        ki bo

   10.   maja 2016 ob    15.    uri na gradu Otočec.   Prijetno   druženje bomo

nadaljevali v    hotelu   Mons.   Tjaša in  Pavle
```

Nepopisno mu gre na živce, da je med besedami po več presledkov. Še bolj pa ga nervira, ker so nekatere vrstice precej daljše od drugih. Ker je Klodvik vaš dober prijatelj, mu boste pomagali in napisali funkcije, s katerimi bo lahko olepšal besedila.

Najprej napišite funkcijo `razrez(niz)`, ki kot argument dobi niz in vrne tabelo besed v tem nizu. Besede so med seboj ločene z enim ali večimi praznimi znaki: `' '` (presledek), `'\t'` (tabulator) in `'\n'` (skok v novo vrstico). Pri tej nalogi ločilo obravnavamo kot del besede. Primer:

```
>>> razrez('   Kakršen\t pastir, \n\ntakšna  čreda. ')
['Kakršen', 'pastir,', 'takšna', 'čreda.']
```

4. podnaloga

Sedaj, ko že imate funkcijo `razrez`, bo lažje napisati tisto funckijo, ki jo Klodovik zares potrebuje. To je funkcija `olepsaj_besedilo(niz, sirina)`, ki kot argumenta dobi niz in naravno število `sirina`. Funkcija vrne olepšano besedilo, kar pomeni naslednje:

- Funkcija naj odstrani odvečne prazne znake.
- Vsaka vrstica naj bo kar se le da dolga.
- Nobena vrstica naj ne vsebuje več kot `sirina` znakov (pri čemer znaka `'\n'` na koncu vrstice ne štejemo).
- Besede znotraj iste vrstice naj bodo ločene s po enim presledkom (ne glede na to s katerimi in koliko praznimi znaki so ločene v originalnem besedilu).

Predpostavite, da dolžina nobene besede ni več kot `sirina` in da je niz neprazen. Primer:

```
>>> lepo_besedilo = olepsaj_besedilo('  Jasno in   svetlo \t\tna sveti \t\n\nvečer,  do
bre\t\t letine je dost, če pa je\t  oblačno in   temno,        žita ne bo.', 20)
>>> print(lepo_besedilo)
Jasno in svetlo na
sveti večer, dobre
letine je dost, če
pa je oblačno in
temno, žita ne bo.
```

Odstavki

1. podnaloga

V nekem besedilu so odstavki ločeni s praznimi vrsticami. Popravi funkcijo `odstavki(besedilo)`, ki naj prebere besedilo v nizu in ga vrne kot niz, pri tem pa, če se kdaj pojavi več zaporednih praznih vrstic, takšno skupino praznih vrstic nadomesti z eno samo prazno vrstico. Za prazne vrstice štejemo le tiste vrstice, ki res ne vsebujejo nobenega znaka, niti presledkov. Novo vrstico označuje `'\n'`.

```
def odstavki(besedilo):
    """Skupine praznih vrstic v besedilu nadomesti z eno samo prazno vrstico."""
    vrstice = besedilo.split('')
    izpis = ''
    prejsnja_prazna = False
    for vrstica in vrstice:
        if vrstica == '' and not prejsnja_prazna:
            continue
        elif vrstica == '' and prejsnja_prazna:
            prejsnja_prazna = True
            izpis += vrstica + '\n'
        else:
            prejsnja_prazna = False
            izpis += vrstica + '\n'
    return izpis
```

Primer

```
>>> odstavki('Danes je lepo, sončno vreme.\n\n\n\nZunaj pojejo ptički.')
'Danes je lepo, sončno vreme.\n\nZunaj pojejo ptički.'
```

Utrjevanje

MiniLogo

MiniLogo je okrnjena izvedba programskega jezika Logo, ki premore le štiri ukaze: `F`, `B`, `R` in `L`. S temi štirimi ukazi upravljamo želvo. Želva se nahaja v ravnini, ki je opremljena z običajnim koordinatnim sistemom (prva koordinata narašča proti desni, druga koordinata pa narašča navzgor). Želva se na začetku nahaja v točki in gleda navzgor (proti "severu"). Z ukazom `F` želvi povemo, naj naredi en korak naprej (v tisti smeri, kamor je trenutno obrnjena). Z ukazom `B` povemo, naj naredi želva korak nazaj. Ukaz `R` pomeni, naj se želva zavrti v desno (tj. v smeri urinega kazalca) za 90°. Ukaz `L` pa pomeni, naj se želva zavrti za 90° v levo (tj. v nasprotni smeri urinega kazalca). Program, ki upravlja z želvo, je torej niz v katerem nastopajo zgornji štirje znaki. Primer: `'FFLFRFRFFFLBRBLBBLF'`.

1. podnaloga

Sestavite funkcijo `kam_pa_kam(ukazi)`, ki kot argument dobi niz `ukazi`. Ta niz vsebuje ukaze za želvo, kot je opisano zgoraj. Funkcija naj izračuna in vrne koordinati tiste točke, kjer se želva ustavi.

```
>>> kam_pa_kam('LFF')
(-2, 0)
>>> kam_pa_kam('FRFLFRFLFRFLFRF')
(4, 4)
```

2. podnaloga

Sestavite funkcijo `pot_zelve(ukazi)`, ki kot argument dobi niz `ukazi`. Funkcija naj vrne tabelo točk (tj. urejenih parov koordinat), ki naj predstavlja pot, ki jo prepotuje želva.

```
>>> pot_zelve('LFF')
[(0, 0), (-1, 0), (-2, 0)]
>>> pot_zelve('RFRFRFRF')
[(0, 0), (1, 0), (1, -1), (0, -1), (0, 0)]
```

Želva vedno začne svoj pohod v točki obrnjena proti "severu". Pot (tj. tabela, ki jo funkcija vrne) bo vedno vsebovala en element več, kot je skupno število znakov `'F'` in `'B'` v nizu ukaz.

3. podnaloga

Sestavite funkcijo sled_zelve(ukazi), ki kot argument dobi niz `ukazi`. Funkcija naj vrne tabelo `sled`, ki predstavlja sled želve (tj. tabela vseh koordinat, ki jih želva vsaj enkrat obišče na svoji poti). Sled se od poti razlikuje v tem, da se nobena točka v tabeli `sled` ne sme pojaviti več kot enkrat. Pozor: točke, ki jih želva obišče prej, se morajo tudi v tabeli `sled` pojaviti prej.

```
>>> sled_zelve('FFFBBFFFBBFFFBB')
[(0, 0), (0, 1), (0, 2), (0, 3), (0, 4), (0, 5)]
>>> sled_zelve('FBRFBRFBRFBRFBFB')
[(0, 0), (0, 1), (1, 0), (0, -1), (-1, 0)]
```

4. podnaloga

Želve, ki živijo v neposredni bližini Černobila, lahko korakajo samo *nazaj* in se obračajo samo v *levo*. Kljub temu pa se izkaže, da lahko tudi "handicapirana" želva prehodi čisto vsako pot, ki jo lahko prehodijo običajne želve. Sestavite funkcijo `mutant(pot)`, ki kot argument dobi tabelo `pot`, ki predstavlja *veljavno* pot želve, ki se začne v točki `ukazi`, ki sme vsebovati le znaka 'L' in 'B'. Želva, ki bo začela svojo pot v točki obrnjena proti "severu" in bo sledila tem ukazom, bo prehodila natančno tisto pot, ki je podana kot argument funkcije.

```
>>> mutant([(0, 0), (-1, 0), (-2, 0)])
'LLLBB'
>>> mutant([(0, 0), (1, 0), (1, -1), (0, -1)])
'LBLLLBLLLB'
```

Niz, ki ga vrne funkcija `mutant`, ne sme vsebovati odvečnih rotacij.

Polinomi

Polinome predstavimo s tabelo koeficientov, pri čemer element z indeksom predstavlja koeficient ob (med drugim torej prazna tabela predstavlja ničelni polinom).

1. podnaloga

Napišite funkcijo `odstrani_odvecne_nicle(koeficienti)`, ki vrne tabelo `koeficienti`, v katerem na koncu odstrani odvečne ničle. Namreč polinom `2x - 3` bi lahko prestavili kot `[-3, 2]` ali pa `[-3, 2, 0]` (torej `0 x^2 + 2x - 3`) ali pa `[-3, 2, 0, 0, 0, 0]`.

```
>>> odstrani_odvecne_nicle([0, 1, 2])
[0, 1, 2]
>>> odstrani_odvecne_nicle([2, 1, 0])
[2, 1]
>>> odstrani_odvecne_nicle([0, 0, 0, 0, 0, 0])
[]
```

2. podnaloga

Napišite funkcijo `odvod_polinoma(polinom, n)`, ki za dani polinom vrne njegov `n`-ti odvod. Privzeta vrednost za `n` naj bo 1.

```
>>> odvod_polinoma([1, 2, 3])
[2, 6],
>>> odvod_polinoma([4, -1, 2, 0, 1], n=2)
[4, 0, 12],
>>> odvod_polinoma([1])
[]
```

3. podnaloga

Napišite funkcijo `vsota_polinomov(polinom1, polinom2)`, ki vrne vsoto danih polinomov. Nastali polinom naj ne vsebuje odvečnih ničel.

```
>>> vsota_polinomov([1, 2, 3], [5, 6, 7, 8])
[6, 8, 10, 8]
>>> vsota_polinomov([1, 2, 3], [-1, -2, -3])
[]
```

Sporočilo iz vesolja

Naloga s tekmovanja RTK 1999

1. podnaloga

Iz vesolja pričakuješ signal nezemeljske civilizacije. Utemeljeno lahko pričakuješ, da bo sporočilo poslano v jeziku, ki uporablja angleško abecedo 26 malih črk in zadošča naslednjim pravilom:

- Sporočilo je sestavljeno zgolj iz zaporedja znakov a, b, c, d, . . . , z, ki sestavljajo besede, in iz presledkov med njimi (brez ločil).
- Zaporedni znaki se vselej razlikujejo (ni podvojenih znakov).
- Samoglasniki a, e, i, o, u se ne smejo stikati.
- Znak x je lahko le na koncu besede.

Naloga

Preuredi vrstice funkcije `pravo(niz)` tako, da bo ugotovila, ali sporočilo ustreza tem pogojem.

```
if znak not in dovoljeni_znaki:
prejsnji_znak = znak
return False
def pravo(niz):
return False
return False
return True
dovoljeni_znaki = 'abcdefghijklmnopqrstuvwxyz '
for znak in niz:
if znak == prejsnji_znak:
prejsnji_znak = ''
samoglasniki = ['a','e','i','o','u']
if znak in samoglasniki and prejsnji_znak in samoglasniki:
'''Vrne True, če besedilo v niz ustreza vseh pogojem in False sicer.'''
return False
if prejsnji_znak == 'x' and znak != ' ':
```

Vhodni podatki

Sporočilo v obliki niza.

Izhodni podatki

`True`, če niz ustreza vsem pravilom in `False` sicer.

Primer

```
>>> pravo('we live in same universe')
`True`


>>> pravo('can you hear us')
`False`
```

Šumniki

1. podnaloga

Sestavite funkcijo `sumniki(niz)`, ki kot argument dobi niz `niz`, vrne pa niz, v katerem so vsi šumniki `'č'`, `'š'`, `'ž'`, `'Č'`, `'Š'` in `'Ž'` zamenjani s pripadajočimi znaki brez strešic `'c'`, `'s'`, `'z'`, `'C'`, `'S'` in `'Z'`. Zgled:

```
>>> sumniki('Špela nabira rožice.')
'Spela nabira rozice.'
```

NATO

Za prenos kritičnih informacij in za zmanjšanje napak pri prenosu je mednarodna organizacija za civilno letalstvo (ang. *International Civil Aviation Organization - ICAO*) uvedla mednarodno abecedo za radio-telefonsko črkovanje.

To pomeni, da je vsaki izmed 26 črk angleške abecede priredila besedo, ki se začne s to črko: alfa, bravo, charlie, delta, echo, foxtrot, golf, hotel, india, juliett, kilo, lima, mike, november, oscar, papa, quebec, romeo, sierra, tango, uniform, victor, whiskey, x-ray, yankee, zulu.

Kasneje so to abecedo prevzele NATO in nekatere druge organizacije.

1. podnaloga

Sestavite funkcijo `crkujNATO(besedilo)`, ki vrne niz, v katerem je vsaka mala črka iz niza `besedilo` zamenjana z ustrezno besedo iz zgornje abecede. Če znak ni mala tiskana črka, naj ostane nespremenjen. Med vsako besedo oz. nespremenjenim znakom naj bo presledek. Na koncu niza naj ne bo presledka.

Za pretvorbo uporabite naslednjo tabelo:

```
['alpha', 'bravo', 'charlie', 'delta', 'echo', 'foxtrot', 'golf',
 'hotel', 'india', 'juliett', 'kilo', 'lima', 'mike', 'november',
 'oscar', 'papa', 'quebec', 'romeo', 'sierra', 'tango', 'uniform',
 'victor', 'whiskey', 'x-ray', 'yankee', 'zulu']
```

Na primer:

```
>>> crkujNATO('test')

tango echo sierra tango

>>> crkujNATO('star sem 18 let.')

sierra tango alpha romeo   sierra echo mike   1 8   lima echo tango .
```

2. podnaloga

Sestavite funkcijo `razberiNATO(nizNATO)`, ki sprejme niz `nizNATO` črkovan po NATO standardu in ga pretvori v navadno besedilo. Vsaka črka (torej beseda) je v `nizNATO` ločena z enim presledkom.

Predpostavi, da v besedilu ni presledkov in da je besedilo zagotovo pravilno črkovano! Na primer:

```
>>> razberiNATO('tango echo sierra tango')

test

>>> razberiNATO('sierra tango alpha romeo - sierra echo mike - 1 8 - lima echo tango .'
)

star-sem-18-let.
```

3. podnaloga

Sestavite funkcijo `prav_razberiNATO(nizNATO)`, ki naredi isto kot funkcija `razberiNATO`, le da tokrat dopuščamo presledke med besedami. Če je v originalnem besedilu bil presledek, se seveda v `nizNATO` pojavijo trije zaporedni presledki. Na primer:

```
>>> prav_razberiNATO('tango echo sierra tango')

test

>>> prav_razberiNATO('sierra tango alpha romeo - sierra echo mike - 1 8 - lima echo tan
go .')

star-sem-18-let.

>>> prav_razberiNATO('sierra tango alpha romeo   sierra echo mike   1 8   lima echo tan
go .')

star sem 18 let.
```

4. podnaloga

Sestavite funkcijo `razberiNATODeLux(nizNATO)`, ki naredi isto kot funkcija `prav_razberiNATO`, le da tokrat dopuščamo, da je besedilo napačno črkovano. Če je besedilo napačno črkovano, naj funkcija vrne niz `NAPAKA!`

Na primer:

```
>>> razberiNATODeLux('tango echo sierra tango')

test

>>> razberiNATODeLux('tangice echo sierra tango')

NAPAKA!

>>> razberiNATODeLux('sierra tango alpha romeo - sierra echo mike - 1 8 - lima echo tan
go .')

star-sem-18-let.
```

Kronogrami

Kronogram je napis, ki ga običajno najdemo na spomenikih, v katerem je zakodirana letnica dogodka, ki ga spomenik obeležuje. Letnico v poenostavljenem kronogramu dobimo tako, da seštejemo vse vrednosti tistih črk, ki lahko nastopajo v rimskih številkah. Vrednosti črk:

• I … 1 • V … 5 • X … 10 • L … 50 • C … 100 • D … 500 • M … 1000

1. podnaloga

Sestavite funkcijo `kronogram(napis)`, ki za poljuben napis izračuna letnico, ki se skriva v njem. Letnico izračunajte tako, da pogledate vsako črko in če je enaka črki iz zgornjega seznama, letnico povečajte za vrednost, ki jo črka predstavlja.

2. podnaloga

Dana je funkcija `krajse_kronogram(napis)`, ki za poljuben napis izračuna letnico, ki se skriva v njem, s pomočjo tabele, v katero zapišemo črke in njim pripadajoče vrednosti (npr. [("I, 1"), ...]).

Žal pa ima nekaj napak. Odpravi jih!

Hammingova razdalja

1. podnaloga

Naj bosta in niza dolžine in (recimo jima binarna niza). Hammingova razdalja med binarnima nizoma in je število pozicij v katerih se razlikujeta. Sestavite funkcijo `hammingova_razdalja(a, b)`, ki vrne Hammingovo razdaljo med `a` in `b`. Če niza nista enakih dožin, naj funkcija vrne `None`. Na primer:

```
>>> hammingova_razdalja('00100', '10000')
2
>>> hammingova_razdalja('11100', '00011')
5
>>> hammingova_razdalja('11100000', '00011')
None
```

2. podnaloga

Recimo, da imamo tri binarne nize dolžine je binarni niz dolžine `mediana(a, b, c)`, ki vrne mediano nizov `a`, `b` in `c`. Na primer:

```
>>> mediana('0000', '1100', '1010')
'1000'
>>> mediana('10101', '11010','00011')
'10011'
```

3. podnaloga

Naj bo tabela različnih binarnih nizov dolžine tudi njihova mediana v `je_medianska(s)`, ki vrne `True`, če je tabela `s` medianska in `False` v nasprotnem primeru. Na primer:

```
>>> je_medianska(['111','110','100','000'])
```

```
True
>>> je_medianska(['010', '100','001'])
False
```

4. podnaloga

Recimo, da opazite na majhnem seznamu binarnih nizov, da za poljubno trojico nizov in njihovo mediano velja: Hammingova razdalja med in `ali_velja(s)`, ki prejme seznam `s` binarnih nizov enake dolžine in preveri ali zgornja trditev velja za vsako trojico nizov iz tega seznama. Na primer:

```
>>> ali_velja(['000','001','011', '111'])
True
```

Številski sestavi

Za naravno število `n` dobimo števke na sledeči način: enice so ostanek pri deljenju števila `n` z 10, desetice so ostanek pri deljenju celoštevilskega količnika `n//10` z 10, stotice so ostanek pri deljenju celoštevilskega količnika `n//100` z 10 itd. Enaka ideja deluje, če želimo dobiti števke števila `n` v kakšnem drugem številskem sestavu. Na primer, pri osnovi 3 so 'enice' ostanek pri deljenju števila `n` s 3, 'trojice' ostanek pri deljenju celoštevilskega količnika `n//3` s 3, 'devetice' ostanek pri deljenju celoštevilskega količnika `n//9` s 3 itd.

1. podnaloga

Napišite funkcijo `pretvori_iz_desetiskega(n, b)`, ki v obliki niza vrne zapis števila `n` v sestavu z osnovo `b`. Predpostaviti smete, da je `b` naravno število med 2 in 10.

Primer:

```
>>> pretvori_iz_desetiskega(14, 2)
'1110'
>>> pretvori_iz_desetiskega(14, 7)
'20'
```

2. podnaloga

Možne so tudi osnove, večje od 10, le da potrebujemo več števk. Običaj je, da se števke, večje od 9, označujejo z angleškimi velikimi tiskanimi črkami; tako A predstavlja 10, B predstavlja 11 itd. Na primer, šestnajstiški sistem vsebuje števke 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F. Z običajnimi števkami in črkami skupaj tako dobimo 36 možnih števk.

Napišite funkcijo `boljsi_pretvori_iz_desetiskega(n, b)`, ki deluje za vse osnove od 2 do 36.

Primer:

```
>>> boljsi_pretvori_iz_desetiskega(200, 16)
'C8'
```

3. podnaloga

Napišite še funkcijo `pretvori(n, a, b)`, kjer `n` predstavlja zapis števila v osnovi `a`, rezultat pa je isto število, zapisano v osnovi `b`. Parameter `n` je lahko dan kot število ali kot niz, rezultat pa naj bo v vsakem primeru niz. Privzeta vrednost za `a` in `b` naj bo 10 (dovoljene so od 2 do 36).

Primer:

```
>>> pretvori('A1', a=16, b=9)
'188'
>>> pretvori(100, b=8)
'144'
```

Namig: tu si lahko prebereš nekaj malega o privzetih vrednostih argumentov v pythonu.

Lepšanje in šifriranje

1. podnaloga

Klodovik (papiga) in ne Klodvik (frankofonski kralj)/ bi rad zašifriral svoja besedila, da jih nepoklicane osebe ne bodo mogle prebrati. To stori tako, da najprej v besedilu vse male črke spremeni v velike in odstrani vse znake, ki niso črke. Klodvik vsa pomembna besedila piše v angleščini, zato bomo uporabljali angleško abecedo.

Na primer iz besedila `'Attack at dawn!'` dobi besedilo `'ATTACKATDAWN'`. Nato ga zapiše cik-cak v treh vrsticah, kot prikazuje primer:

```
A...C...D...
.T.A.K.T.A.N
..T...A...W.
```

Sestavite funkcijo `cik_cak(niz)`, ki vrne trojico nizov (torej `tuple`) in sicer prvo, drugo in tretjo vrstico v tem zapisu. Primer:

```
>>> cik_cak('Attack at dawn!')
('A...C...D...', '.T.A.K.T.A.N', '..T...A...W.')
```

2. podnaloga

Zašifrirano besedilo dobi tako, da najprej prepiše vse znake iz prve vrstice, nato vse znake iz druge vrstice in na koncu še vse znake iz tretje vrstice. V zgornjem primeru bi tako dobil `'ACDTAKTANTAW'`. Sestavite funkcijo `cik_cak_sifra(niz)`, ki dobi kot argument niz in vrne zašifrirano besedilo. Primer:

```
>>> cik_cak_sifra('Attack at dawn!')
'ACDTAKTANTAW'
```

3. podnaloga

Klodovik se zelo razjezi, ko dobi elektronsko pošto v takšni obliki:

```
Kar sva  si obljubljala    že leta,  si   želiva potrditi tudi   pred prijatelji in   c
elo

žlahto. Vabiva te na


    poročno slovesnost,        ki bo

  10.   maja 2016 ob    15.    uri na gradu Otočec.   Prijetno   druženje bomo
```

```
nadaljevali v    hotelu   Mons.   Tjaša in  Pavle
```

Nepopisno mu gre na živce, da je med besedami po več presledkov. Še bolj pa ga nervira, ker so nekatere vrstice precej daljše od drugih. Ker je Klodvik vaš dober prijatelj, mu boste pomagali in napisali funkcije, s katerimi bo lahko olepšal besedila.

Najprej napišite funkcijo `razrez(niz)`, ki kot argument dobi niz in vrne tabelo besed v tem nizu. Besede so med seboj ločene z enim ali večimi praznimi znaki: `' '` (presledek), `'\t'` (tabulator) in `'\n'` (skok v novo vrstico). Pri tej nalogi ločilo obravnavamo kot del besede. Primer:

```
>>> razrez('   Kakršen\t pastir, \n\ntakšna  čreda. ')
['Kakršen', 'pastir,', 'takšna', 'čreda.']
```

4. podnaloga

Sedaj, ko že imate funkcijo `razrez`, bo lažje napisati tisto funckijo, ki jo Klodovik zares potrebuje. To je funkcija `olepsaj_besedilo(niz, sirina)`, ki kot argumenta dobi niz in naravno število `sirina`. Funkcija vrne olepšano besedilo, kar pomeni naslednje:

- Funkcija naj odstrani odvečne prazne znake.
- Vsaka vrstica naj bo kar se le da dolga.
- Nobena vrstica naj ne vsebuje več kot `sirina` znakov (pri čemer znaka `'\n'` na koncu vrstice ne štejemo).
- Besede znotraj iste vrstice naj bodo ločene s po enim presledkom (ne glede na to s katerimi in koliko praznimi znaki so ločene v originalnem besedilu).

Predpostavite, da dolžina nobene besede ni več kot `sirina` in da je niz neprazen. Primer:

```
>>> lepo_besedilo = olepsaj_besedilo('  Jasno in   svetlo \t\tna sveti \t\n\nvečer,  do
bre\t\t letine je dost, če pa je\t  oblačno in   temno,        žita ne bo.', 20)
>>> print(lepo_besedilo)
Jasno in svetlo na
sveti večer, dobre
letine je dost, če
pa je oblačno in
temno, žita ne bo.
```

Slovarji III

Zbiranje osebnih podatkov

V nekem podjetju zbirajo podatke o osebah, ki brskajo po medmrežju. Podatki so shranjeni v slovarjih (en slovar za vsako osebo). Ključ v takem slovarju je *lastnost*, vrednosti pa *podatek* o tej lastnosti za določeno osebo. Primeri takšnih slovarjev:

```
oseba1 = {'ime': 'Božidar', 'telefon': '031918211',
          'obiskane spletne strani': ['facebook.com', 'google.com']}
oseba2 = {'ime': 'Pikica', 'naslov': 'Dunajska 105', 'številka noge': 42,
          'prijatelji': ['Marko', 'Ana']}
```

1. podnaloga

Sestavite funkcijo `podatek(oseba, lastnost)`, ki vrne podatek o lastnosti `lastnost`, ki ga imamo v slovarju `oseba`. Funkcija naj vrne `None`, če se ta podatek v slovarju ne nahaja. Primer:

```
>>> podatek({'ime': 'Božidar', 'naslov': 'Dunajska 105'}, 'naslov')
'Dunajska 105'
>>> podatek({'ime': 'Božidar', 'naslov': 'Dunajska 105'}, 'prijatelji')
None
```

Pri tem ne smeš uporabiti get!

2. podnaloga

Sestavite funkcijo `pridobi_podatek(oseba, lastnost)`, ki vrne podatek o lastnosti `lastnost`, ki ga imamo v slovarju `oseba`. Funkcija naj vrne `None`, če se ta podatek v slovarju ne nahaja. Primer:

```
>>> pridobi_podatek({'ime': 'Božidar', 'naslov': 'Dunajska 105'}, 'naslov')
'Dunajska 105'
>>> pridobi_podatek({'ime': 'Božidar', 'naslov': 'Dunajska 105'}, 'prijatelji')
None
```

Pri tem si nujno pomagajte z metodo `get`

3. podnaloga

Za osebi `oseba1` in `oseba2` pravimo, da se *ujemata* v lastnosti `lastnost`, če za obe osebi poznamo podatka o tej lastnosti in sta podatka enaka. Če pa za obe osebi poznamo podatka in sta podatka različna, pa pravimo, da se osebi *razlikujeta* v lastnosti `lastnost`. Na primer, osebi

```
oseba1 = {'ime': 'Janez', 'priimek': 'Novak'}
oseba2 = {'ime': 'Jože', 'priimek': 'Novak', 'starost': 20}
```

se ujemata v lastnosti `'priimek'` in razlikujeta v lastnosti `'ime'`. (V lastnosti `'starost'` se niti ne ujemata niti ne razlikujeta.)

Sestavite funkcijo `ujemanje(oseba1, oseba2)`, ki pove, v koliko lastnostih se osebi `oseba1` in `oseba2` ujemata in v koliko lastnostih se razlikujeta. Rezultat naj funkcija vrne kot nabor z dvema elementoma. Primer:

```
>>> ujemanje({'ime': 'Janez', 'priimek': 'Novak'},
             {'ime': 'Jože', 'priimek': 'Novak', 'starost': 20})
(1, 1)
```

4. podnaloga

Dva različna slovarja `oseba1` in `oseba2` lahko predstavljata isto osebo. To se zgodi, če se slovarja ne razlikujeta v več kot 1 lastnosti in se ujemata vsaj v 3 lastnostih ali če sta v obeh slovarjih lastnosti `ime` in `priimek` enaki. Na primer, slovarja

```
oseba1 = {'ime': 'Janez', 'priimek': 'Novak', 'telefon': '031123234',
          'starost': 90}
oseba2 = {'ime': 'Janez', 'priimek': 'Novak', 'davčna': '43424140'}
```

predstavljata isto osebo, prav tako slovarja

```
oseba1 = {'ime': 'Janez', 'priimek': 'Novak', 'telefon': '031123234',
          'starost': 90, davčna': '43424140'}
oseba2 = {'ime': 'Luka', 'priimek': 'Novak', 'starost': 90,
          'davčna': '43424140', 'telefon': '031123234'}
```

Sestavite funkcijo `ista(oseba1, oseba2)`, ki preveri, ali slovarja predstavljata isto osebo. Na primer, za zgornja primera mora funkcija vrniti `True`.

5. podnaloga

V seznamu slovarjev, ki predstavljajo osebe, se lahko zgodi, da več slovarjev predstavlja isto osebo (isto v smislu prejšnje podnaloge). V takem primeru pravimo, da so ti slovarji *podvojeni*.

Sestavite funkcijo `podvojeni(seznam_oseb)`, ki vrne seznam vseh podvojenih slovarjev iz seznama `seznam_oseb`. Slovarji naj bodo razvrščeni v istem vrstnem redu kot v seznamu `seznam_oseb`. Primer:

```
>>> podvojeni([{'ime': 'Jan', 'priimek': 'Dan', 'naslov': 'Jadranska 21'},
               {'ime': 'Jan', 'priimek': 'Dan', 'naslov': 'Jadranska 19'},
               {'ime': 'Žan', 'priimek': 'Dan', 'naslov': 'Jadranska 21'},
               {'ime': 'Žan', 'priimek': 'Noč', 'naslov': 'Jamova 25'}])
[{'ime': 'Jan', 'priimek': 'Dan', 'naslov': 'Jadranska 21'},
 {'ime': 'Jan', 'priimek': 'Dan', 'naslov': 'Jadranska 19'}]
```

Šifriranje

Babica Metka in njen vnuk Boštjan sta se odločila, da si bosta pošiljala skrivna sporočilca zakodirana s substitucijsko šifro.

Substitucijska šifra je enostaven način šifriranja, pri katerem vsako črko iz dane abecede zamenjamo z neko drugo črko. Tako šifro predstavimo s slovarjem, ki ima za ključe vse črke iz abecede, pripadajoče vrednosti pa so črke, s katerimi jih zašifriramo.

Tako slovar `{'A': 'B', 'C': 'C', 'B': 'D', 'D': 'A'}` pove, da se `'A'` zašifrira v `'B'`, `'B'` v `'D'`, `'D'` v `'A'`, `'C'` pa se ne spremeni.

1. podnaloga

Sestavite funkcijo `sifriraj(sifra, beseda)`, ki vrne besedo, zašifrirano z dano šifro. Predpostavite lahko, da vse črke v besedi nastopajo v šifri. Zgled:

```
>>> sifriraj(nasa_sifra, "IGRICA")
'BPLBZO'
```

2. podnaloga

Sestavite funkcijo `je_sifra(slovar)`, ki ugotovi, ali `slovar` predstavlja šifro, torej ali je bijekcija črk na neki abecedi (bijektivna preslikava abecede samo vase). Zgled:

```
>>> je_sifra(nasa_sifra)
True
```

Namig: pri bijektivni preslikavi je poljuben element iz množice B slika točno enega elementa množice A.

3. podnaloga

Sestavite funkcijo `inverz(sifra)`, ki vrne inverz dane šifre, če ta obstaja. V nasprotnem primeru funkcija vrne `None`. Zgled:

```
>>> inverz({'A': 'D', 'B': 'A', 'D': 'J', 'J': 'B'})
{'A': 'B', 'B': 'J', 'J': 'D', 'D': 'A'}
```

Namig: inverz obstaja samo kadar je slovar, ki predstavlja šifro, bijekcija črk na neki abecedi.

4. podnaloga

Sestavite funkcijo `odsifriraj(sifra, beseda)`, ki sprejme šifro in zašifrirano besedilo, vrne pa odšifrirano besedilo. Če slovar ni bijekcija (in se torej besedilo ne da nujno odšifrirati), naj funkcija vrne `None`. Zgled:

```
>>> odsifriraj(nasa_sifra, 'BPLBZO')
'IGRICA'
```

Permutacije - lažje

Babica Metka bi rada svojemu vnuku Boštjanu zelo rada pomagala do boljše ocene pri matematiki. Ker trenutno pri pouku obravnavajo permutacije, o katerih sama ne ve prav dosti, ji priskočite na pomoč.

Permutacijo naravnih števil od do lahko predstavimo s slovarjem. Na primer permutacijo, ki slika v v pa pusti pri miru, predstavimo s slovarjem `{1: 3, 2: 2, 3: 1}`.

1. podnaloga

Sestavite funkcijo `slika(permutacija, x)`, ki kot argumenta dobi slovar `permutacija`, ki predstavlja permutacijo, in število `x`. Funkcija naj vrne sliko števila `x` s podano permutacijo. Zgled:

```
>>> slika({1: 3, 2: 4, 3: 2, 4: 1}, 4)
1
```

2. podnaloga

Sestavite funkcijo `je_permutacija(slovar)`, ki vrne `True`, če dan slovar predstavlja permutacijo, in `False` sicer. Zgled:

```
>>> je_permutacija({1: 3, 2: 5, 5: 1})
False
```

3. podnaloga

Sestavite funkcijo `slike(permutacija, x, n)`, ki vrne zaporedje slik, ki ga dobimo, če začnemo s številom `x` in na njem `n`-krat uporabimo permutacijo `permutacija`. Zgled:

```
>>> slike({1: 3, 2: 4, 3: 2, 4: 1}, 1, 2)
[1, 3, 2]
```

Davki

Pri nas trenutno obstajajo tri davčne stopnje: višja, nižja in oproščena. Davčne stopnje so podane v slovarju, kjer so ključi vedno taki, kot prikazuje zgled davcne_stopnje = {'V': 22, 'N': 9.5, 'O': 0}

Lastnik popularne trgovinice (ki ima celo lastno pekarno), vas je najel, da mu napišete program za sestavljanje računov.

Nakupi so shranjeni v slovarju `kosarice`, katerega ključi so imena kupcev, vrednosti pa slovarji predmetov in količin. Zgled:

```
kosarice = {'Janez': {'banana': 5, 'jogurt': 7},
            'Mojca': {'francoska štrucka': 7}}
```

1. podnaloga

V tej trgovinici so francoske štručke zelo popularne. Vsak kupec vedno kupi vsaj dve, tudi če tega nima zapisanega na nakupovalnem listku. Sestavite funkcijo `dodaj_strucke(kosarice)`, ki sestavi in vrne nov slovar, v katerem ima vsak kupec poleg vsega, kar že ima, še vsaj po dve francoski štručki. Zgled:

```
>>> kosarice = {'Janez': {'banana': 5, 'jogurt': 7}, 'Mojca': {'francoska štručka': 7}}
>>> dodaj_strucke(kosarice)
{'Janez': {'jogurt': 7, 'francoska štručka': 2, 'banana': 5},
 'Mojca': {'francoska štručka': 7}}
```

2. podnaloga

Šef trgovine je pripravil cenik vseh izdelkov, ki jih trgovinica prodaja. Vsak izdelek ima par podatkov in sicer bruto ceno in davčno stopnjo. Napišite funkcijo `neto_in_davek(cenik, davcne_stopnje)`, ki sestavi in vrne nov slovar, v katerem bo par podatkov in sicer: neto cena izdelka in davek, ki ga mora trgovinica plačati državi, če so trenutno veljavne podane davčne stopnje. Pri tem vse cene in davke zaokroži na 5 decimalnih mest. Zgled:

```
>>> cenik = {'voda': (0.25, 'N'), 'sok': (1.20, 'V'), 'znamka A': (0.23, 'O')}
>>> neto_in_davek(cenik, {'V': 20, 'N': 8.5, 'O': 0})
{'sok': (1.0, 0.2), 'voda': (0.23041, 0.01959), 'znamka A': (0.23, 0.0)}
```

3. podnaloga

Sestavite funkcijo `cene_z_ddv(kosarice, cenik, davcne_stopnje)`, ki dobi slovar nakupov in cenik (v originalni obliki), ter sestavi nov slovar nakupov, ki za vsak nakup vsebuje peterico: `(kolicina, neto_na_enoto, neto_skupaj, davcna_stopnja, davek_skupaj)` (Ne pozabite upoštevati, da vsak kupi vsaj dve štručki.)

Pri tem vse cene in davke zaokroži na 5 decimalnih mest. Zgled:

```
>>> cenik = {'voda': (0.25, 'N'), 'sok': (1.25, 'V'), 'znamka A': (0.23, 'O'),
             'francoska štručka': (1, 'N')}
>>> kosarice = {'Janez': {'voda': 5, 'sok': 7},
                'Mojca': {'francoska štručka': 7}}
>>> davcne_stopnje = {'V': 20, 'N': 8.5, 'O': 0}
>>> cene_z_ddv(kosarice, cenik, davcne_stopnje)
{'Janez': {'voda': (5, 0.23041, 1.15205, 'N', 0.09795),
           'sok': (7, 1.04167, 7.29169, 'V', 1.45831),
           'francoska štručka': (2, 0.92166, 1.84332, 'N', 0.15668)},
 'Mojca': {'francoska štručka': (7, 0.92166, 6.45162, 'N', 0.54838)}}
```

Slovarji IV (rekurzija)

Kraji in reči

V nalogi se bomo igrali s slovarji, ki opisujejo, kje se nahaja kakšen kraj. Slovar je lahko recimo, tak

```
svet = {
    "Evropa": {
        "Slovenija": {
            "Gorenjska": {
                "Kranj": {},
                "Radovljica": {},
                "Zali log": {},
            },
            "Štajerska": {
                "Maribor": {},
                "Celje": {}
            },
            "Osrednja": {
                "Ljubljana": {
                    "Vič": {
                        "FMF": {
                            "2.02": {
                                "tretja vrsta desno": {
                                    "peti stol z desne": {
                                        "Benjamin": {}
                                    }
                                }
                            }
                        }
                    },
                    "Šiška": {}
                }
            }
        },
        "Nemčija": {
            "Bavarska": {
```

```
                "Munchen": {}
            },
            "Berlin": {}
        },
    },
    "Amerika": {
        "ZDA": {
            "Teksas": {
                "Houston": {},
                "Austin": {}
            },
            "Kalifornija": {
                "San Francisco": {}
            },
            "Anchorage": {}
        },
        "Kanada": {}
    },
    "Azija": {
        "Osaka": {}
    }
   }
```

Ključi slovarja so torej opisi, vsebine pa so slovarji, ki so lahko tudi prazni. Namig: Oglejte si kako preveriti, ali je neka struktura prazna,

1. podnaloga

Napiši funkcijo `vrni_vse(slovar_opisov)`, ki kot argument dobi slovar, kakršen je gornji, in vrne po abecedi urejeno tabelo vseh opisov v njem.

Primer:

```
>>> prvo_nadstropje = {"Prvo nadstropje": {"soba": {"računalnik": {}, "jaz": {}},
                       "otroška soba": {"Martin": {"ostali piškoti": {}}}}}
>>> vrni_vse(prvo_nadstropje)
['Martin', 'Prvo nadstropje', 'jaz', 'ostali piškoti', 'otroška soba', 'računalnik',
'soba']
```

2. podnaloga

Napiši funkcijo `prestej(slovar_opisov)`, ki vrne število vseh opisov, ki se pojavijo v podanem slovarju.

Primer:

```
>>> prvo_nadstropje = {"Prvo nadstropje": {"soba": {"računalnik": {}, "jaz": {}},
                       "otroška soba": {"Martin": {"ostali piškoti": {}}}}}
>>> prestej(prvo_nadstropje)
7
```

3. podnaloga

Denimo, da imamo slovar (kot sta `kraji` in `svet`), ki kot vrednosti spet vsebuje slovarje. Sestavi funkcijo `naj_slovar(slovar)`, ki vrne tisti slovar, ki vsebuje največ ključev.

Predpostaviš lahko, da je samo en tak slovar (torej ima le en slovar to največje število ključev.

Primer:

```
>>> prvo_nadstropje = {"Prvo nadstropje": {
                       "soba": {"računalnik": {}, "jaz": {}, "miza": {}, "tipkovnica
": {}},
                       "otroška soba": {"Martin": {"ostali piškoti": {}}},
                       "kopalnica": {"milo": {}}}}
>>> naj_slovar(prvo_nadstropje)
{'računalnik': {}, 'jaz': {}, 'miza': {}, 'tipkovnica': {}}
```

4. podnaloga

Sestavi funkcijo `je_v_slovarju(slovar, nekaj)`, ki ugotovi, ali je v slovarju `slovar` tudi niz `nekaj` (bodisi kot ključ, bodisi v vrednosti, ki je spet slovar).

Primer:

```
>>> prvo_nadstropje = {"Prvo nadstropje": {"soba": {"računalnik": {}, "jaz": {}},
                       "otroška soba": {"Martin": {"ostali piškoti": {}}}}}
>>> je_v_slovarju(prvo_nadstropje, "računalnik")
True
>>> je_v_slovarju(prvo_nadstropje, "Martin")
True
>>> je_v_slovarju(prvo_nadstropje, "kremna rezina")
False
```

5. podnaloga

Sestavi funkcijo `vrni_vrednost(slovar, nekaj)`, ki vrne tisto vrednost iz slovarja slovarjev `slovar`, ki pripada ključu `nekaj` Če pa v slovarju slovarjev `nekaj` ni ključ, naj vrne None.

Primer:

```
>>> prvo_nadstropje = {"Prvo nadstropje": {"soba": {"računalnik": {}, "jaz": {}},
                       "otroška soba": {"Martin": {"ostali piškoti": {}}}}}
>>> vrni_vrednost(prvo_nadstropje, "Martin")
{"ostali piškoti": {}}
```

```
>>> vrni_vrednost(prvo_nadstropje, "jaz")
{}
>>> vrni_vrednost(prvo_nadstropje, "kopalnica")
None
```

Veliki šef in njegovi podrejeni

V nekem uspešnem podjetju ima skoraj vsak zaposleni svojega šefa. Seveda imajo tudi šefi svoje šefe in ti spet svoje šefe itn. Dobili smo podatke o hierarhiji v podjetju in ugotovili, da velja naslednje:

- Vsaka oseba ima kvečjemu enega šefa.
- En šef ima lahko pod seboj več zaposlenih.
- Vsem, ki so nad nami (naš šef, šef našega šefa itd.), pravimo *nadrejeni*.
- Vsem, ki so pod nami (sami smo njihov šef, ali pa smo šef njihovega šefa itd.), pravimo *podrejeni*.
- Nihče ni sam svoj šef in nihče ni samemu sebi podrejen.

Napisali bomo nekaj funkcij, s katerimi bomo preučili razmere v podjetju.

1. podnaloga

Podatke smo dobili v obliki tabele parov oblike `(uslužbenec, šef)`. Vsi uslužbenci so predstavljeni z nizi (ki so običajno njihova imena oz. priimki). Zgled:

```
[('Mojca', 'Tilen'), ('Andrej', 'Tilen'), ('Tilen', 'Zoran')]
```

Komentar: Tilen ima pod seboj dva podrejena (Andreja in Mojco), njegovi šef pa je Zoran. Zoran nima šefa, vsi ostali pa so njegovi podrejeni.

Napišite funkcijo `slovar_sefov(tabela)`, ki bo iz zgoraj opisane tabele zgradila slovar šefov. Ključi v slovarju naj bodo uslužbenci, vrednosti pa njihovi šefi. Zgled:

```
>>> slovar_sefov([('Mojca', 'Tilen'), ('Andrej', 'Tilen'), ('Tilen', 'Zoran')])
{'Andrej': 'Tilen', 'Mojca': 'Tilen', 'Tilen': 'Zoran'}
```

2. podnaloga

Napišite funkcijo `neposredno_podrejeni(tabela)`, ki bo iz zgoraj opisane tabele sestavila slovar neposredno podrejenih. Vrednost pri vsakem ključu naj bo *množica* tistih uslužbencev, ki imajo le-ta ključ za svojega šefa. Zgled:

```
>>> neposredno_podrejeni([('Mojca', 'Tilen'), ('Andrej', 'Tilen'), ('Tilen', 'Zoran')])
{'Zoran': {'Tilen'}, 'Tilen': {'Andrej', 'Mojca'}}
```

Zoranu je torej neposredno podrejen le Tilen, Tilnu pa sta podrejena tako Andrej kot Mojca.

3. podnaloga

Sestavite funkcijo `veriga_nadrejenih(usluzbenec, slovar)`, ki kot prvi argument dobi niz, ki predstavlja nekega uslužbenca, kot drugi argument pa dobi slovar šefov (v obliki kot ga vrne funkcija `slovar_sefov`). Funkcija naj sestavi tabelo, ki po vrsti vsebuje: šefa osebe `usluzbenec`, šefa od njegovega šefa itn. (dokler končno ne pridemo do osebe, ki nima šefa). Zgled:

```
>>> veriga_nadrejenih('Mojca', {'Andrej': 'Tilen', 'Mojca': 'Tilen', 'Tilen': 'Zoran'})
['Tilen', 'Zoran']
```

Mojci je nadrejen Tilen, kateremu je nadrejen Zoran, torej sta Mojčina šefa Tilen in Zoran.

4. podnaloga

Sestavite funkcijo `mnozica_podrejenih(usluzbenec, slovar)`, ki kot prvi argument dobi ime uslužbenca, kot drugi argument pa slovar neposredno podrejenih (v obliki kot ga vrne funkcija `neposredno_podrejeni`). Funkcija naj sestavi in vrne *množico* vseh tistih oseb, ki so (posredno ali neposredno) podrejeni osebi `usluzbenec`. Zgled:

```
>>> mnozica_podrejenih('Zoran', {'Zoran': {'Tilen'}, 'Tilen': {'Andrej', 'Mojca'}, 'Mojca' : {'Urša'}})
{'Andrej', 'Mojca', 'Tilen', 'Urša'}
```

Zoranju je neposredno podrejen Tilen, torej sta mu podrejena tudi Tilnova podrejena Andrej in Mojca, ter Urša, ker je ta podrejena Mojci.

5. podnaloga

Tistemu uslužbencu, za katerega velja:

- da nima nadrejenih;
- je zadnji v verigi nadrejenih vseh ostalih uslužbencev (razen seveda samega sebe);

pravimo *big boss*. Sestavite funkcijo `big_boss(slovar)`, ki kot argument dobi slovar nadrejenih in vrne ime osebe, ki je big boss v podjetju oz. vrednost `None`, če to podjetje nima big boss-a. Zgled:

```
>>> big_boss({'Andrej': 'Tilen', 'Mojca': 'Tilen', 'Tilen': 'Zoran'})
'Zoran'
```

Permutacije in cikli

Se še spomnimo Metke in Boštjana in permutacij? Če ne, poišči nalogo Permutacije - lažje, saj bomo pri reševanju uporabili tudi tam razvite funkcije.

Samo da se spomnimo:

Permutacijo naravnih števil od do lahko predstavimo s slovarjem. Na primer permutacijo, ki slika v v pa pusti pri miru, predstavimo s slovarjem `{1: 3, 2: 2, 3: 1}`.

1. podnaloga

Sestavite funkcijo `cikel(permutacija, x)`, ki vrne celoten cikel, ki se začne s številom `x`. Zgled:

```
>>> cikel({1: 3, 2: 2, 3: 1}, 1)
[1, 3]
>>> cikel({1: 3, 2: 2, 3: 1}, 2)
[2]
```

2. podnaloga

Sestavite funkcijo `cikli(permutacija)`, ki vrne seznam disjunktnih ciklov dane permutacije. Vsak cikel naj se začne z najmanjšim številom v ciklu, cikli pa naj bodo urejeni po začetnem številu. Zgled:

```
>>> cikli({1: 3, 2: 5, 3: 1, 4: 2, 5: 4})
[[1, 3], [2, 5, 4]]
>>> cikli({1: 6, 2: 5, 3: 3, 4: 2, 5: 4, 6: 1})
[[1, 6], [2, 5, 4], [3]]
```

Gensko spremenjena hrana

Upravljalec restavracije te prosi za pomoč. Ljudje so ozaveščeni in skrbi jih gensko spremenjena hrana. Moram mu pomagati, da bo izračunal količino GSOja, ki bo zapisana v novem jedilniku.

1. podnaloga

Kuhar je že sestavil slovar vseh jedi na jedilniku skupaj z njihovimi sestavinami. Sestavine so tudi podane v obliki slovarja z vnosi oblike `sestavina: količina`. Enota pri količini ni podadana (lahko je to teža, volumen, število kosov, …). Ker se je zelo mudilo, je pri jedeh, kjer je cela jed ena sama sestavina, pisal le prazno množico. Sestavite funkcijo `odpravi_okrajsave(recepti)`, ki sestavi in vrne nov slovar, ki ne vsebuje okrajšav. Zgled:

```
>>> recepti = {'pica': {'moka': 80, 'sir': 30}, 'solata': dict()}
>>> odpravi_okrajsave(recepti)
{'solata': {'solata': 1}, 'pica': {'sir': 30, 'moka': 80}}
```

2. podnaloga

V tistih množicah, ki predstavljajo recepte, je kuhar količine sestavin podal v gramih. Da bomo lažje računali GSO, jih je treba "normalizirat", da bo vse v procentih. Napišite funkcijo `normaliziraj_kolicine(recepti)`, ki kot argument dobi slovar receptov in vrne "normaliziran" slovar receptov (tj. vsota vseh količin v vsakem receptu naj bo točno 100). Zgled:

```
>>> recepti = {'pica': {'moka': 80, 'sir': 30} 'solata': dict()}
>>> normaliziraj_kolicine(recepti)
{'solata': {'solata': 100.0},
 'pica': {'sir': 27.272727272727273, 'moka': 72.72727272727273}}
```

3. podnaloga

Upravljalec restavracije vam je priskrbel še seznam vseh elementarnih sestavin skupaj z njihovim deležem GSO. Sestavite funkcijo `delez_gso(recepti, elementarne)`, ki dobi slovar z recepti in slovar z deleži GSO v elementarnih sestavinah in poračuna deleže GSO v jedeh. V slovarju elementarnih sestavin so vsaj vse elementarne sestavine, ki se nahajajo v receptih, lahko pa jih je tudi več. Zgled:

```
>>> recepti = {'pica': {'moka': 80, 'sir': 30}, 'solata': dict()}
>>> elementarne = {'moka': 70, 'sir': 10, 'solata': 0}
>>> delez_gso(recepti, elementarne)
```

```
{'solata': 0.0, 'pica': 53.63636363636364}
```

4. podnaloga

Poleg običajnega imajo v restavraciji tudi specialni jedilnik, ki vsebuje eksotične jedi. Te vsebujejo tudi sestavine, ki prihajajo iz "sumljivih" dežel in deleža GSO ni mogoče ugotoviti. Napišite še funkcijo `delez_gso_special(recepti, elementarne)`, ki je podobna kot v prejšni nalogi, le da vsaki jedi priredi "interval" (tj. par `(min_gso, max_gso)`). V minimalnem primeru predpostavimo povsod 0 % GSO, v maximalnem primeru pa povsod 100 % GSO. Zgled:

```
>>> recepti = {'pica_special': {'moka': 80, 'sir': 30, 'namaz': 10}, 'solata_special':
dict()}

>>> elementarne = {'moka': 70, 'sir': 10, 'solata': 5}

>>> delez_gso_special(recepti, elementarne)

{'pica_special': (49.16666666666667, 57.5), 'solata_special': (0.0, 100.0)}
```

Usmerjeni grafi

Kadar nas zanima, kam se lahko iz danega kraja odpravimo, lahko odpremo zemljevid in pogledamo, v katere kraje nas iz danega kraja vodijo ceste. Recimo iz Celja se lahko odpravimo proti Velenju, Laškem ali Rogaški Slatini, iz Logatca se lahko odpravimo proti Idriji, Postojni in Vrhniki, iz Vrhinike se lahko odpravimo nazaj proti Logatcu ali proti Ljubljani.

Tak zemljevid lahko predstavimo tudi z usmerjenim grafom. Usmerjen graf sestavljata množica vozlišč in množica usmerjenih povezav

```
= {'Logatec', 'Vrhnika', 'Ljubljana', 'Postojna', 'Idrija'}

A = [('Logatec', 'Vrhnika'), ('Logatec', 'Postojna'), ('Logatec', 'Idrija'), ('Vrhnika'
, 'Logatec'), ('Vrhnika', 'Ljubljana')]
```

Lahko pa ga podamo v obliki slovarja naslednikov:

```
{'Logatec': ['Vrhnika', 'Postojna', 'Idrija'], 'Vrhnika': ['Logatec', 'Ljubljana']}
```

(V zgornjem primeru smo predpostavili, da se iz Postojne, Idrije in Ljubljane, ne da priti nikamor (kar seveda ni res))

Ključi v tem slovarju so vozlišča, vrednost pri posameznem ključu `u` pa je seznam vseh vozlišč, ki so nasledniki vozlišča `u`. (Vozlišče je *naslednik* vozlišča
izoliranih vozlišč (to so vozlišča, ki niso niti začetek niti konec katere od povezav).

1. podnaloga

Napišite funkcijo `slovar_naslednikov(seznam_povezav)`, ki kot argument dobi seznam povezav digrafa, sestavi in vrne pa naj pripadajoči slovar naslednikov. Zgled:

```
>>> slovar_naslednikov([('a', 'b'), ('c', 'b'), ('c', 'd'), ('d', 'a'), ('a', 'c')])

{'a': ['b', 'c'], 'c': ['b', 'd'], 'd': ['a']}
```

Seznami naslednikov naj bodo *urejeni*.

2. podnaloga

Sestavite funkcijo `seznam_povezav(digraf)`, ki kot argument dobi usmerjen graf, ki je podan kot slovar naslednikov. Funkcija naj sestavi in vrne seznam usmerjenih povezav. (Torej, funkcija `seznam_povezav` naj naredi ravno obratno kot funkcija `slovar_naslednikov`.) Zgled:

```
>>> seznam_povezav({'a': ['b', 'c'], 'c': ['b', 'd'], 'd': ['a']})
[('a', 'b'), ('a', 'c'), ('c', 'b'), ('c', 'd'), ('d', 'a')]
```

Seznam povezav, ki ga vrne funkcija, naj bo *urejen*.

3. podnaloga

Napišite funkcijo `nasprotni_graf(digraf)`, ki dobi usmerjen graf v obliki slovarja naslednikov, sestavi in vrne pa naj nasprotni graf (tudi v obliki slovarja naslednikov). *Nasprotni graf* grafa ima enako množico vozlišč kot

```
 nasprotni_graf({'a': ['b', 'c'], 'c': ['b', 'd'], 'd': ['a']})
{'a': ['d'], 'c': ['a'], 'b': ['a', 'c'], 'd': ['c']}
```

Seznami naslednikov naj bodo *urejeni*.

4. podnaloga

Usmerjene grafe lahko sestavimo iz večih usmerjenih podgrafov. Vsak posamezen podgraf je predstavljen s slovarjem, za katerega velja, da ima vsako vozlišče le enega naslednika.

Sestavite funkcijo `sestavljen_graf(tabela_podgrafov)`, ki bo iz dane tabele usmerjenih podgrafov sestavila slovar, ki bo predstavljal graf v obliki slovarja naslednjikov. Vrednosti naj bodo v tabelah zapisane v enakem vrstnem redu, kot nastopajo v tabeli podgrafov. Zgled:

```
>>> sestavljen_graf([{'a': 'b', 'c': 'b'}, {'a': 'c', 'c': 'd', 'd': 'a'}])
{'a': ['b', 'c'], 'c': ['b', 'd'], 'd': ['a']}
```

Rodovniki II

Spet se bomo ukvarjali z rodovniki (Celjskih grofov in drugih). Rodovnik imamo podan kot slovar, kjer je ključ ime "glave rodbine" vrednost pa tabela imen otrok. Recimo:

```
rodovnik =
 {'Ulrik I.': ['Viljem'], 'Margareta': [], 'Herman I.': ['Herman II.', 'Hans'],
  'Elizabeta II.': [], 'Viljem': ['Ana Poljska'], 'Elizabeta I.': [],
  'Ana Poljska': [], 'Herman III.': ['Margareta'], 'Ana Ortenburška': [],
  'Barbara': [], 'Herman IV.': [], 'Katarina': [], 'Friderik III.': [],
  'Herman II.': ['Ludvik', 'Friderik II.', 'Herman III.', 'Elizabeta I.', 'Barbara'],
  'Ulrik II.': ['Herman IV.', 'Jurij', 'Elizabeta II.'], 'Hans': [], 'Ludvik': [],
  'Friderik I.': ['Ulrik I.', 'Katarina', 'Herman I.', 'Ana Ortenburška'],
  'Friderik II.': ['Friderik III.', 'Ulrik II.'], 'Jurij': []}
 rodovnik['Friderik II.']
```

nam torej vrne

```
['Friderik III.', 'Ulrik II.']
```

1. podnaloga

Je v rodbini?

Sestavi funkcijo `je_v_rodbini(ime, glava_rodbine, rodovnik)`, ki ugotovi, ali je oseba z imenom `ime` v rodbini osebe `glava_rodbine`. Funkcija torej vrne `True` oz. `False`.

2. podnaloga

Kdo se podpisuje najdlje časa?

Sestavi funkcijo `najdaljsi_podpis(glava_rodbine, rodovnik)`, ki ugotovi, kdo v rodbini osebe `glava_rodbine` ima najdaljše ime za podpis (torej kompletno ime).

3. podnaloga

Kdo ima najkrajše ime?

Sestavi funkcijo `najkrajse_ime(glava_rodbine, rodovnik)`, ki ugotovi, kdo v rodbini osebe `glava_rodbine` ima najkrajše ime. (šteje samo krstno ime, brez "Ortenburga" in "Celja" ter brez številk)? "Ana X" ima torej krajše ime kot "Hugo".

4. podnaloga

Globina rodbine

"Globino" rodbine definiramo tako: če nekdo nima otrok, je globina njegove rodbine 1. Če ima otroka, ta pa nima vnukov (ali celo več otrok, ti pa nimajo vnukov), je globina rodbine 2. Če nekdo ima vnuke, vendar nobenega pravnuka, je globina njegove rodbine 3.

Sestavi funkcijo `globina(glava_rodbine, rodovnik)`, ki vrne globino rodbine osebe `glava_rodbine` v rodovniku `rodovnik`

Rekurzija III

Dvobarvne ogrlice

Takrat, ko je upokojenki Tini dolgčas, vzame svoji dve posodici z belimi in rdečimi kroglicami ter začne nizati ogrlice. Te ogrlice bomo predstavili z nizi, sestavljenimi iz znakov `B` in `R`.

Na primer: `'BBRBBRB'` in `'RRBBBBB'` sta dve izmed 21 možnih ogrlic, sestavljenih iz petih belih in dveh rdečih kroglic.

1. podnaloga

Sestavite funkcijo `je_ogrlica(niz, b, r)`, ki preveri, ali `niz` predstavlja ogrlico iz `b` belih in `r` rdečih kroglic. Na primer:

```
>>> je_ogrlica('BBRBBRB', 5, 2)
True
>>> je_ogrlica('RRBBBBB', 5, 2)
True
>>> je_ogrlica('BBRBBRB', 2, 5)
False
>>> je_ogrlica('BBRBBRBBB', 5, 2)
False
>>> je_ogrlica('BBRBBRBXY', 5, 2)
False
```

2. podnaloga

Z označimo število različnih ogrlic, sestavljenih iz natanko belih in rdečih kroglic. Če je eno od števil ali enako nič, potem je `'BBBBB'`.
V nasprotnem primeru pa velja belih in rdečih kroglic:

- bodisi začne z belo kroglico, preostalih belih in rdečih kroglic pa lahko sestavimo na načinov,
- bodisi začne z rdečo kroglico, preostalih belih in rdečih kroglic pa lahko sestavimo na načinov.

Sestavite funkcijo `stevilo_ogrlic(b, r)`, ki izračuna število vseh možnih ogrlic, sestavljenih iz natanko `b` belih in `r` rdečih kroglic. Na primer:

```
>>> stevilo_ogrlic(5, 0)
1
>>> stevilo_ogrlic(5, 2)
21
>>> stevilo_ogrlic(4, 2)
15
>>> stevilo_ogrlic(5, 1)
```

```
6
```

3. podnaloga

Sestavite funkcijo `ogrlice(b, r)`, ki generira vse nize, ki predstavljajo ogrlice, sestavljene iz `b` belih in `r` rdečih kroglic. Funkcija naj nize vrača v abecednem vrstnem redu.

Primer:

```
>>> ogrlice(2, 2)
['BBRR', 'BRBR', 'BRRB', 'RBBR', 'RBRB', 'RRBB']
```

Dolžina poleta

1. podnaloga

Letala letalske družbe SLED pogosto opravljajo precej dolge polete z vmesnimi postanki. V kontroli ob vsakem postanku preverijo, ali bodo letala lahko prispela do cilja s trenutno količino goriva.

Sestavite funkcijo `dovolj_goriva(gorivo, pot)`, kjer je gorivo količina goriva v letalu na začetku potovanja, pot pa je tabela s potrebnimi količinami goriva do naslednjega letališča. Funkcija naj vrne `True`, če je goriva dovolj za celo pot in `False`, če bo letalu goriva na poti zmanjkalo. Če je v letalu ravno prav goriva, naj funkcija vrne `True`.

```
>>> dovolj_goriva(200, [25, 60, 43, 28])
True
>>> dovolj_goriva(10, [30, 40])
False
```

Najprej rešite nalogo z uporabo zanke for ali while.

2. podnaloga

Sestavite enako funkcijo, (poimenujte jo `dovolj_goriva_rek(gorivo, pot)`) le da tokrat ne smete uporabiti zanke for ali while, pač pa rekurzivni klic.

Pri tej nalogi dobite dva ustavitvena pogoja - enega, če zmanjka goriva in drugega, če ga je dovolj.

SSCŠ se vrača

Že pri prejšnjih nalogah smo spoznali SSCŠ. Ponovimo še enkrat - seznam seznamov celih števil (SSCŠ) je seznam, katerega elementi so bodisi cela števila bodisi seznami seznamov celih števil. Da bo enostavneje, recimo, da je tudi prazen seznam SSCŠ. Nekaj primerov:

```
[-1, 2, -3]
[1, [2], 3, [2, 3, 4]]
[[[[0]], 1], [2, [[-3], [4]]]]
```

1. podnaloga

Sestavi funkcijo `loci_stevila(sscs)`, ki vrne trojico `(poz, nic, neg)`, ki pove, koliko je v sscs pozitivnih, ničelnih in koliko negativnih celih števil. Za primere od zgoraj so rezultati:

```
(1, 0, 2)
(6, 0, 0)
(3, 1, 1)
```

Predpostavi, da je parameter zagotovo SSCS.

2. podnaloga

Sestavi funkcijo `je_sscs(sscs)`, ki preveri (vrne True oz. False), če je dan seznam res seznam seznamov celih števil.

POZOR: Python pravi, da je `isinstance(True, int)` enako `True`, pa tudi `isinstance(False, int)` enako `True`, zato se "znebi" napačnih logičnih vrednosti z `isinstance(True, bool)`

Lahko pa uporabiš tudi type(nekaj) is int, ki pa deluje "prav" tudi, če je v x logična vrednost

3. podnaloga

Sestavi funkcijo `koliko_napacnih(sscs)`, ki prešteje, koliko je v seznamu seznamov celih števil napačnih podatkov, torej elementov, ki niso ne cela števila, ne SSCŠ. Če argument ni seznam, vrni `None`

```
>>>koliko_napacnih([1, 'bla', 2])
1
>>>koliko_napacnih([True, 2, False])
2
>>>koliko_napacnih([1, [['bla', 'blu'], False], 2])
1
>>>koliko_napacnih([[1.8], [2, 3, [4]], [False, 4, [True]]])
2
>>>koliko_napacnih(12)
None
>>>koliko_napacnih(set())
None
```

Podrejeni

Rekurzija se lahko pojavi tudi brez "eksplicitnega" ustavitvenega pogoja. Npr. takrat, ko je rekurzivni klic del zanke in se lahko zgodi, da so podatki taki, da se zanka ne izvede nikoli.

1. podnaloga

V podjetju SLED vodijo evidenco svojih zaposlenih tako, da se vedno ve, kdo je glavni. Za vsakega zaposlenega v tabelo `zaposleni` vpišejo vse njegove neposredno podrejene. Če bi imeli štiri zaposlene, kjer bi bil Janez šef, Liza in Katja njegovi podrejeni, poleg tega pa bi Luka bil podrejen Lizi, bi tabela `zaposleni` izgledala tako:

```
  [('Janez', 'Liza'),('Janez', 'Katja'), ('Liza', 'Luka')]
```

Vsak zaposleni je podrejen kvečjemu eni osebi.

Uredite vrstice kode spodaj, da bo funkcija `podrejeni(ime, zaposleni)`, ki sprejme ime delavca `ime` in tabelo `zaposleni` in vrne število vseh podrejenih (tudi posredno podrejenih) delavcu z imenom `ime`, delovala pravilno. Poleg tega ima ena vrstica napako, ena pa je odveč.

```
      """Prešteje uslužbence, podrejene delavcu ime."""
      return podrejenih
      podrejenih = 0
      if sef == ime:
      def podrejeni(ime, zaposleni):
      for (sef, usluzbenec) in zaposleni:
      podrejenih = podrejeni(usluzbenec, zaposleni)
      return 0
```

2. podnaloga

Sestavite še funkcijo `glavni(zaposleni)`, ki vrne množico vseh glavnih šefov. Glavni šef je tisti, ki nima nobenega nadrejenega.

Naloga je rešljiva s pomočjo rekurzije, vendar je precej težka, tako da lahko rešite tudi brez rekurzije, oglejte pa si uradno rešitev z rekurzijo, tam se boste še kaj novega naučili.

Cik-cakasti sprehodi

Pravimo, da je sprehod po točkah v ravnini *cik-cakast*, če gremo iz točke vedno v točko ali pa v `je_cikcakast(sprehod)`, ki vrne `True` natanko tedaj, kadar je sprehod, podan z zaporedjem točk `sprehod`, cik-cakast. Zgled:

```
>>> je_cikcakast([(1, 1), (2, 0), (3, -1), (4, 0), (5, 1)])
True
```

2. podnaloga

Sestavite funkcijo `prestavi_v_1kvadrant(sprehod)`, ki sprehod zamakne tako, da v celoti poteka izključno v 1. kvadrantu ter da je njegova prva točka na ordinatni, njegova najnižja točka pa na abscisni osi. Zgled:

```
>>> prestavi_v_1kvadrant([(1, 1), (2, 0), (3, -1), (4, 0), (5, 1)])
[(0, 2), (1, 1), (2, 0), (3, 1), (4, 2)]
```

3. podnaloga

Sestavite funkcijo `stevilo_cikcakastih(zac, kon)`, ki vrne število vseh cik-cakastih sprehodov med točkama `zac` in `kon`. Zgled:

```
>>> stevilo_cikcakastih((0, 0), (9, 3))
84
>>> stevilo_cikcakastih((0, 0), (0, 0))
1
>>> stevilo_cikcakastih((0, 0), (-1, 0))
0
>>> stevilo_cikcakastih((1, 1), (9, 3))
56
>>> stevilo_cikcakastih((1, -1), (9, 3))
28
```

Namig: Kam gre lahko prvi korak cik-cakastega sprehoda? Dobro si oglejte zgornji zgled.

4. podnaloga

Sestavite funkcijo `narisi_sprehod(sprehod, izhod)`, ki v datoteko z imenom `izhod` z znaki `/` in `\` izpiše cik-cakast sprehod, podan z množico točk `sprehod`. Sprehod naj premakne tako, da bo najbolj levi znak v prvem stolpcu, najvišji pa v prvi vrstici datoteke. Sprehod `[(0, 0), (1, 1), (2, 0), (3, -1), (4, -2), (5, -1)]` morate tako izpisati kot:

```
/\
  \
   \/
```

Enako sliko da npr. tudi

`[(1, 1), (2, 2), (3, 1), (4, 0), (5, -1), (6, 0)]`

sprehod `[(5, -4), (6, -3), (7, -4), (8, -5), (9, -4), (10, -3), (11, -4), (12, -5)]` pa kot:

```
/\  /\
  \/  \
```

Testni primeri delovanje primerjajo na sledečih cik-cakastih sprehodih:

- `[(0, 0), (1, 1), (2, 2), (3, 3), (4, 4), (5, 5)]`,
- `[(0, 0), (1, -1), (2, -2), (3, -3), (4, -4), (5, -5)]`,
- `[(0, 0), (1, 1), (2, 0), (3, -1), (4, -2), (5, -1)]`
- `[(1, 1), (2, 2), (3, 1), (4, 0), (5, -1), (6, 0)]`
- `[(5, -4), (6, -3), (7, -4), (8, -5), (9, -4), (10, -3), (11, -4), (12, -5)]`

Namig:

Collatzovo zaporedje

Collatzovo zaporedje tvorimo na sledeč način. Začnemo z nekim naravnim številom in prište-
jemo (v tem primeru stvar ni več zanimiva, saj se začno ponavljati števila je tako
in
`naslednji_clen(n)`, ki izračuna člen, ki v Collatzovemu zaporedju sledi številu `n`.

2. podnaloga

Sestavite funkcijo `dolzina_zaporedja(n)`, ki izračuna dolžino Collatzovega zaporedja, ki se začne s številom `n`. Dolžina zaporedja je število korakov, preden zaporedje doseže 1.

Na primer, dolžina zaporedj, ki se začne s

```
>>> dolzina_zaporedja(2)
  1
  >>> dolzina_zaporedja(6)
  8
  >>> dolzina_zaporedja(5)
  5
  >>> dolzina_zaporedja(1)
  3
```

3. podnaloga

Sestavite funkcijo `najvecji_clen(n)`, ki izračuna največji člen v Collatzovem zaporedju, ki se začne s številom `n`.

V zaporedju, ki se začne s

```
>>> najvecji_clen(2):
  2
  >>> najvecji_clen(6)
  16
  >>> najvecji_clen(5)
  16
  >>> najvecji_clen(1)
  4
```

4. podnaloga

Sestavite funkcijo `najdaljse_zaporedje(m, n)`, ki vrne dolžino najdaljšega zaporedja med vsemi tistimi Collatzovimi zaporedji, ki se začnejo s števili med (vključno) `m` in `n`.

Primera:

```
  >>> najdaljse_zaporedje(5, 5)
  5
  >>> najdaljse_zaporedje(5, 8)
  16
```

Rekurzija in OS

==========

Sklop nalog, kjer pišemo funkcije, ki naredijo določena opravila z datotekami na nivoju operacijskega sistema. V nalogah bodo potrebne te funkcije:

- `os.listdir(mapa)` seznam datotek in map v mapi
- `os.path.isfile(dat)` ali je dat »navadna« datoteka
- `os.path.isdir(dat)` ali je dat imenik
- `os.path.getsize(dat)` velikost datoteke
- `os.path.split(dat)` razdeli ime datoteke
- `os.path.getmtime(dat)` čas zadnje spremembe datoteke
- `os.rmdir(pot)` odstrani mapo
- `os.path.join(mapa1, mapa2, ime_datoteke)` združi argumente v veljavno pot - zadnji je lahko ime datoteke ali mape (npr. `os.path.join("P1", "os", "resitev.py")` vrne niz `"P1\\os\\resitev.py"` na OS Windows oziroma `"P1/os/resitev.py"` na OS Linux in MacOS)

Pregled map in datotek

Trenutno preverjenje ne deluje (ker ni ustrezne datoteke) Vsaka rešitev je javljena kot pravilna! Za preverjanje nalog v imenik/mapo, kjer imate rešitve, skopirajte

ZIP datoteko in jo razširite (nastala bo mapa `PythonOsRek` s podmapami `testX` in `EURO2012`)

1. podnaloga

Sestavite funkcijo `seznam_vseh_podmap(zacetna_mapa)`, ki vrne seznam poti do vseh map (vključno z začetno), ki jih vsebuje `zacetna_mapa`. Če `zacetna_mapa` ni veljavno ime mape, naj funkcija vrne `None`.

```
>>>> seznam_vseh_podmap('.\\PythonOSRek\\test4')
['.\\PythonOSRek\\test4', '.\\PythonOSRek\\test4\\test3',
  '.\\PythonOSRek\\test4\\New folder', '.\\PythonOSRek\\test4\\test3\\New folder']
```

Namig: Ali pot predstavlja mapo oz. datoteko preverimo z `os.path.isdir`. Poti med seboj združujemo z `os.path.join`, da lahko program uporabljamo na različnih operacijskih sistemih.

2. podnaloga

Sestavite funkcijo `mape_ki_vsebujejo_datoteko(zacetna_mapa, datoteka)`, ki vrne urejen seznam vseh map, ki vsebujejo dano datoteko. Pri tem naj iskanje prične v začetni mapi.

```
>>>> mape_ki_vsebujejo_datoteko('.\\PythonOSRek\\test1', 'drevo.py')
['.\\PythonOSRek\\test1', '.\\PythonOSRek\\test1\\PodM\\TraRa']
```

Če `mapa` ne opisuje poti do mape, naj funkcija vrne `None`.

3. podnaloga

Sestavite funkcijo `koliko_datotek`, ki prešteje, koliko je vseh datotek v izbrani mapi. Pri tem naj šteje tudi datoteke v podmapah, map pa naj ne šteje. ČeCheck.feedback("Funkcija izpisi_datoteke trenutno nima testov, zato morate za testiranje rešitve poskrbeti sami.") podana pot ne opisuje mape, naj funkcija vrne `None`.

```
>>>> koliko_datotek('.\\PythonOSRek\\test1')
46
```

4. podnaloga

Sestavite funkcijo `vrni_python_datoteke`, ki vrne po abecedi urejen seznam (polnih) imen vseh datotek v izbrani mapi in njenih podmapah, ki imajo končnico `.py`.

Namig: Za pridobivanje končnice lahko uporabite metodo `os.path.splitext`.

Izpisovanje ...

Naloga nima testnega programa! V Tomu so le zato, da so "na kupu"

Vsaka oddaja bo označena kot pravilna!

1. podnaloga

Izpiši mapo

Sestavi funkcijo `izpisi_datoteke(pot_do_mape)`, ki izpiše vse datoteke v izbrani mapi in njenih podmapah. Funkcija naj izpisuje samo imena datotek, brez poti. Uporabi izpis z zamikanjem, da bo razvidno, kateri mapi pripada katera datoteka. Vedno najprej izpiši datoteke, potem mape.

```
    a.txt
    tra.py
    bla
      dat.txt
    ble
      blu
        c.dat
      xxx
      yyy
        e.dat
```

2. podnaloga

Sestavi funkcijo `izpisi_py(pot_do_mape)`, ki izpiše imena vseh datotek v izbrani mapi in njenih podmapah, ki imajo končnico .py. Izpisuje naj polna imena (skupaj s potjo).

Briši ...

1. podnaloga

Sestavi funkcijo `izbrisi_prazne_mape(pot_do_mape)`, ki v izbrani mapi (in njenih podmapah) pobriše vse prazne datoteke (tiste z velikostjo 0) in vse prazne mape (tiste, ki ne vsebujejo nobenih datotek ali podmap). Funkcija naj vrne par, ki ga sestavljata število pobrisanih datotek in število pobrisanih

map, oziroma `None`, če pot_do_mape ne opisuje imenika. Pozor: ko zbrišemo prazno datoteko, se lahko sprazni tudi mapa.

TRENUTNO TESTIRANJE NE DELA! Pred testiranjem vedno skopirajte *to datoteko ZIP* (staro mapo `PythonOsRek` pobrišite!) in jo razširite (nastala bo mapa `PythonOsRek` s podmapami `testX` in `EURO2012`)

Iskanje

Trenutno preverjenje ne deluje (ker ni ustrezne datoteke) Vsaka rešitev je javljena kot pravilna! Za preverjanje nalog v imenik/mapo, kjer imate rešitve, skopirajte

Za preverjanje nalog v imenik/mapo, kjer imate rešitve, skopirajte *ZIP datoteko* in jo razširite (v mapo, v kateri se nahaja pythonova datoteka z nalogami).

Namig: naloge se vam splača reševati tako, da najprej definirate delovanje funkcij v trenutni mapi, potem pa jih rekurzino pokličete še na vseh podmapah. Uporabljajte relativne poti.

1. podnaloga

Sestavi funkcijo `poisci_velike(pot_do_mape, velikost)`, ki vrne seznam polnih imen datotek v dani mapi in njenih podmapah, katerih velikost je večja ali enaka dani vrednosti. Če je mapa prazna, naj funkcija vrne prazen seznam.

```
    >>> poisci_velike('mapa_1', 2000)
    ['mapa_1\\mapa_1_1\\jovo_datoteka.txt', 'mapa_1\\mapa_1_3_abc\\bn_datoteka_556.txt'
]
```

2. podnaloga

Sestavi funkcijo `poisci_poti(pot_do_mape,ime_datoteke)`, ki vrne seznam vseh poti, ki vodijo do dane datoteke.

```
     >>> poisci_poti('mapa_1', 'z_datoteka_1_2_e.txt')
     ['mapa_1\\z_datoteka_1_2_e.txt', 'mapa_1\\mapa_1_1\\z_datoteka_1_2_e.txt']
```

3. podnaloga

Sestavi funkcijo `najstevilcnejsa(pot_do_mape)`, ki v izbrani mapi (in njenih podmapah) poišče mapo, ki vsebuje največ datotek (štejemo samo datoteke v mapi, ne tudi v podmapah, saj je drugače naloga nesmiselna). Funkcija naj vrne ime iskane mape (skupaj s potjo) ter število datotek v tej mapi. Če je mapa prazna, naj funkcija vrne `(None, 0)`.

```
  >>> najstevilcnejsa('mapa_1')
  ('mapa_1\\mapa_1_1', 4)
  >>> najstevilcnejsa('mapa_1\\mapa_1_2')
  (None, 0)
```

4. podnaloga

Poišči vse najnovejše datoteke.

Sestavi funkcijo `poisci_najnovejse(pot_do_mape)`, ki v izbrani mapi (in njenih podmapah) poišče datoteke, ki so bile zadnje spremenjene. Funkcija naj vrne tabelo imen teh datotek (skupaj s potjo).

```
>>> poisci_najnovejse('mapa_1')
['mapa_1\\z_datoteka_1_2_e.txt', 'mapa_1\\mapa_1_1\\z_datoteka_1_2_e.txt', 'mapa_1\
\mapa_1_3_abc\\arrested.txt']
```

5. podnaloga

Poišči najnovejšo datoteko.

Sestavi funkcijo `poisci_najnovejso_abeceda(pot_do_mape)`, ki v izbrani mapi (in njenih podmapah) poišče datoteko, ki je bila zadnja spremenjena, in vrne njeno polno ime (skupaj s potjo). Če je takih datotek več, naj funkcija vrne ime datoteke, ki je prva po abecedi.

```
>>> poisci_najnovejso_abeceda('mapa_1')
'mapa_1\\mapa_1_1\\z_datoteka_1_2_e.txt'
```

6. podnaloga

Poišči največjo datoteko.

Sestavi funkcijo `najvecja(pot_do_mape)`, ki v izbrani mapi (in njenih podmapah) poišče največjo datoteko. Funkcija naj vrne par, ki ga sestavljata po abecedi zadnje ime take datoteke (skupaj s potjo) in njena velikost.

```
>>> najvecja('mapa_1')
('mapa_1\\mapa_1_3_abc\\bn_datoteka_556.txt', 6120)
```

Iskanje II

Naloge nimajo testnega programa! V Tomu so le zato, da so "na kupu". Vsaka oddaja bo označena kot pravilna!

Če uporabite datotečno strukturo iz datoteke ZIP, kot je opisana pri nalogi *Pregled map in datotek*, naj se vaše funkcije obnašajo, kot kažejo zgledi!

Naloga je namenoma ZELO PODOBNA nalogi Iskanje. Le testna ZIP datoteka je drugačna ter sem in tja spremenjena kakšna malenkost. Poskusite nalogi rešiti na različne načine.

1. podnaloga

Sestavi funkcijo `izpisi_velike(pot_do_mape, velikost)`, ki vrne seznam polnih imen datotek v dani mapi in njenih podmapah, katerih velikost je večja ali enaka dani vrednosti.

2. podnaloga

Sestavi funkcijo `izpisi_mapet(pot_do_mape,imeDatoteke)`, ki izpiše vse mape, ki vsebujejo dano datoteko.

```
>>> izpisi_mapet('.\\PythonOSrek', 'drevo.py')
.\PythonOSrek\test1\drevo.py
.\PythonOSrek\test1\podM\TraRa\drevo.py
.\PythonOSrek\test1a\drevo.py
.\PythonOSrek\test1a\podM\TraRa\drevo.py
```

3. podnaloga

Sestavi funkcijo `najstevilcnejsa(pot_do_mape)`, ki v izbrani mapi (in njenih podmapah) poišče mapo, ki vsebuje največ datotek (štejemo samo datoteke v mapi, ne tudi v podmapah, saj je drugače naloga nesmiselna). Funkcija naj vrne ime iskane mape (skupaj s potjo) ter število datotek v tej mapi.

```
>>> najstevilcnejsa('.\\PythonOSrek')
('.\\PythonOSrek\\test1', 7)
```

4. podnaloga

Poišči najnovejšo datoteko.

Sestavi funkcijo `poisci_najnovejse(pot_do_mape)`, ki v izbrani mapi (in njenih podmapah) poišče datoteke, ki so bile zadnje spremenjene. Funkcija naj vrne tabelo imen teh datotek (skupaj s potjo).

```
>>> poisci_najnovejse('.\\PythonOSrek')
['.\\PythonOSrek\\EURO2012\\euro database.py']
```

5. podnaloga

Poišči najnovejšo datoteko.

Sestavi funkcijo `poisci_najnovejse(pot_do_mape)`, ki v izbrani mapi (in njenih podmapah) poišče datoteko, ki so bile zadnja spremenjena. Funkcija naj vrne po abecedi najmanjše ime izmed teh datotek (skupaj s potjo).

```
>>> poisci_najnovejse('.\\PythonOSrek')
'.\\PythonOSrek\\EURO2012\\euro database.py'
```

6. podnaloga

Poišči največjo datoteko.

Sestavi funkcijo `najvecja(pot_do_mape)`, ki v izbrani mapi (in njenih podmapah) poišče največjo datoteko. Funkcija naj vrne par, ki ga sestavljata po abecedi zadnje ime take datoteke (skupaj s potjo) in njena velikost.

```
>>> najvecja('.\\PythonOSrek')
('.\\PythonOSrek\\test1\\Tips.PNG', 94535)
```

Izpisovanje II

Trenutno testi ne delujejo. Vse naloge bodo označene kot pravilne!

Tomo bo pravilnost vaših rešitev preverjal na datotekah, ki jih najdete stisnjene v tej *ZIP datoteki*. Stisnjeno datoteko razširite v isto mapo, kjer se nahaja vaš program.

1. podnaloga

Sestavi funkcijo `poisci_datoteke(pot_do_mape)`, ki vrne seznam vseh datotek (njihova imena brez poti) v izbrani mapi in njenih podmapah. Za vsako podmapo naj funkcija seznamu doda par (ime

podmape, [seznam datotek]). Če podmapa vsebuje še dodatne podmape, naj se par globje podmape vgnezdi v seznam datotek prvotne podmape. Funkcija naj v seznam vedno najprej doda imena datotek, šele potem pare s podmapami.

```
>>> poisci_datoteke('mapa_2')

['datoteka_1_1_abcd.txt', 'dolzina_vektorja.py', 'z_datoteka_1_2_e.txt', ('mapa_1_1'
, ['32_datoteka.txt', 'ar_datoteka_jk.txt', 'jovo_datoteka.txt', 'program_445.py', 'z_d
atoteka_1_2_e.txt', ('gh_mapa_zadnja', ['nova_datoteka.txt', 'test_datoteka.txt', 'zivi
_program.py'])]), ('mapa_1_2', []), ('mapa_1_3_abc', ['arrested.txt', 'bn_datoteka_556.
txt', 'ddatoteka.txt', 'zeta_funkcija.py'])]
```

2. podnaloga

Sestavi funkcijo `izpisi_datoteke()`, ki vpraša po imenu (oz. relativni poti) mape in izpiše vse datoteke v izbrani mapi in njenih podmapah. Funkcija naj izpisuje samo imena datotek, brez poti. Izpiše naj tudi imena podmap, imena datotek v njih pa naj izpiše z zamikom, da bo izpis bolj pregleden. Za vsak nov nivo podmape naj izpis zamakne za dodatna dva presledka. Vedno najprej izpiši imena datotek, šele potem imena podmap. Če si rešil zgornjo nalogo, lahko zgolj predelaš njeno rešitev.

Po definiciji funkcije jo še pokliči brez argumentov (da bo Tomo lahko preveril rešitev; ta vrstica je že dodana).

Namig: Tomo bo med preverjanjem klical funkcijo brez argumentov. To pa še ne pomeni, da funkcija ne sme sprejemati argumentov. Lahko jih, dokler so ti argumenti neobvezni (to pomeni, da imajo predpisano privzeto vrednost).

```
>>> izpisi_datoteke()
Pot do mape: mapa_2
datoteka_1_1_abcd.txt
dolzina_vektorja.py
z_datoteka_1_2_e.txt
mapa_1_1
  32_datoteka.txt
  ar_datoteka_jk.txt
  jovo_datoteka.txt
  program_445.py
  z_datoteka_1_2_e.txt
  gh_mapa_zadnja
    nova_datoteka.txt
    test_datoteka.txt
    zivi_program.py
mapa_1_2
mapa_1_3_abc
  arrested.txt
  bn_datoteka_556.txt
  ddatoteka.txt
  zeta_funkcija.py
```

```
>>>
```

3. podnaloga

Sestavi funkcijo `poisci_datoteke(pot_do_mape, koncnica)`, ki vrne seznam vseh datotek (njihova imena skupaj s potjo) v izbrani mapi in njenih podmapah, ki imajo končnico `koncnica` (npr. `.txt`, `.xlsx`, ..). Privzeta končnica naj bo `.txt`.

```
>>> poisci_datoteke('mapa_2', '.py')
['mapa_2\\dolzina_vektorja.py', 'mapa_2\\mapa_1_1\\program_445.py', 'mapa_2\\mapa_1_1\\gh_mapa_zadnja\\zivi_program.py', 'mapa_2\\mapa_1_3_abc\\zeta_funkcija.py']
```

Datoteke II

Golf

Na vsakem zanimivem igrišču za golf so ovire: jezero, pesek, ...

`Jezero` je krog, podan kot trojica `(x,y,r)`. `x` in `y` sta koordinati središča, `r` pa polmer. `Pesek` je pravokotnik, podan kot četverica`(x1,y1,x2,y2)`. Predpostaviš lahko, da so to po vrsti koordinate levega spodnjega in desnega zgornjega oglišča. Vse koordinate računamo na tri decimalke.

1. podnaloga

Na datoteki imamo zapisane podatke o posameznih udarcih v obliki polarnih koordinat (kot je podan v stopinjah). V vsaki vrstici sta zapisani celi števili `r` in `fi`, ločeni s presledkom. Napišite funkcijo `datoteka_polozajev(vhod, izhod)`, ki naj prebere datoteko `vhod` in ustvari novo datoteko `izhod` tako, da je v vsaki vrstici zapisan trenutni položaj žogice (v obliki decimalnih števil, zaokroženih na 3 decimalna mesta) in ločenih s presledkom. V ta namen uporabite formatiranje z f-nizi. Začetni položaj žogice naj bo v točki `(0,0)`.

2. podnaloga

Podan je seznam položajev žogic po posameznem udarcu in seznam, katerega vsak element je nabor, ki podaja jezero ali pesek. Napišite metodo `se_izogne(pot, ovire)`, ki pove, ali se pot v celoti izogne oviram. Pazi: jezero je podano z naborom treh, pesek pa z naborom štirih števil.

Najprej napišite metodi `je_v_jezeru(zogica, jezero)` in `je_v_pesku(zogica, pesek)`, ki povesta, ali je žogica v jezeru ali v pesku. Žogica je podana kot par `(x,y)`, torej "nima dimenzije".

3. podnaloga

Napišite metodo `kje_je_zogica(datoteka, zacetek, ovire)`, ki vrne vektor od začenega do končnega položaja ali None, če žogica kdaj vmes pade v oviro. Vektor naj bo zaokrožen na 3 decimalna mesta. Posamezni udarci so v polarnih koordinatah zapisani na datoteko, začetek pa je podan kot par `(x,y)`.

Delnice

V datoteki imamo zapisane podatke o vrednosti neke delnice. V vsaki vrstici je zapisan podatek v obliki

```
YYYY-MM-DD,vrednost
```

kjer je prvi podatek dan, drugi pa vrednost delnice na ta dan.

1. podnaloga

Sestavite funkcijo `preberi(ime_datoteke)`, ki kot parameter sprejme ime datoteke, vrne par nabor dveh seznamov, v prvem naj bodo datumi (kot nizi), v drugem pa vrednosti delnice (kot realna števila).

2. podnaloga

Logaritemski povratek delnice je definiran kot logaritem kvocienta vrednosti delnice za dva zaporedna dneva trgovanja:

Sestavite funkcijo `povratek(ime_dat)`, ki kot parameter sprejme ime datoteke z vrednostmi delnice. in vrne seznam logaritemskih povratkov. Če je podana datoteka prazna ali pa vsebuje le en podatek, naj funkcija vrne prazen seznam.

3. podnaloga

Iz logaritemskih povratkov lahko razberemo, ali je vrednost delnice naraščala ali padala. Sestavite funkcijo `trend(povratki)`, ki sprejme seznam logaritemskih povratkov in vrne niz `pozitiven trend`, če je v seznamu več pozitivnih vrednosti kot negativnih, sicer pa naj vrne `negativen trend`. Vrednosti 0 štejte k negativnim.

4. podnaloga

Letna volatilnost delnice (Volatilnost ali nihajnost označuje, koliko je statistično verjetno, da cena delnice v kratkem času močneje zraste ali pade) je definirana kot večkratnik standardnega odklona logaritemskega povratka:

povprečna vrednost logaritemskih povratkov, so posamezni logaritemski povratki, je število logaritemskih povratkov, 252 pa predstavlja število trgovalnih dni v letu. Sestavite funkcijo `volatilnost(ime_datoteke)`, ki iz datoteke prebere vrednosti delnice in vrne njeno letno volatilnost

---

`html2txt(vhod, izhod)`, ki bo vsebino datoteke z imenom `vhod` prepisala v datoteko z imenom `izhod`, pri tem pa odstranila vse značke. Vemo, da je datoteka HTML sestavljena prav!

Značke se začnejo z znakom `<` in končajo z znakom `>`. Pozor: Začetek in konec značke nista nujno v isti vrstici. Takrat se vrstica nadaljuje! Prav tako ima lahko značka lastnosti, npr. značka `a` ima lastnost `href`

```
<a href = "kk.htm">
```

Na primer, če je v datoteki `vreme.html` zapisano:

```
<h1>Napoved vremena</h1>
<p>Jutri bo <i><b>lepo</b></i> vreme.
Več o vremenu preberite <a
href="napoved.html">tukaj</a>.</p>
```

bo po klicu `html2txt('vreme.html', 'vreme.txt')` v datoteki `vreme.txt` zapisano (pozor na tretjo vrstico!):

```
Napoved vremena
Jutri bo lepo vreme.
Več o vremenu preberite tukaj.
```

2. podnaloga

Sestavite funkcijo `tabela(vhod, izhod)`, ki bo podatke iz vhodne datoteke zapisala v obliki HTML tabele v izhodno datoteko.

V vhodni datoteki so podatki shranjeni po vrsticah ter ločeni z vejicami. Na primer, če je v datoteki `tabela.txt` zapisano:

```
ena,dva,tri
17,52,49.4,6
abc,xyz
```

bo po klicu `tabela('tabela.txt', 'tabela.html')` v datoteki `tabela.html`:

```
<table>
  <tr>
    <td>ena</td>
    <td>dva</td>
    <td>tri</td>
  </tr>
  <tr>
    <td>17</td>
    <td>52</td>
    <td>49.4</td>
    <td>6</td>
  </tr>
  <tr>
    <td>abc</td>
    <td>xyz</td>
  </tr>
</table>
```

Pozor: Pazi na zamik (število presledkov na začetku vrstic) v izhodni datoteki.

3. podnaloga

Sestavite funkcijo `seznami(vhod, izhod)`, ki bo podatke iz vhodne datoteke zapisala v izhodno datoteko v obliki neurejenega seznama. V vhodni datoteki se vrstice seznamov začnejo z zvezdico. Temu lahko sledi nekaj presledkov, nato pa element seznama

Na primer, če je v datoteki seznami.txt zapisano:

```
V trgovini moram kupiti:
*jajca,
*       kruh,
* moko.
Na poti nazaj moram:
* obiskati sosedo.
```

bo po klicu `seznami('seznami.txt', 'seznami.html')` v datoteki `seznami.html`:

```
V trgovini moram kupiti:
```

```
<ul>
  <li>jajca,</li>
  <li>kruh,</li>
  <li>moko.</li>
</ul>
Na poti nazaj moram:
<ul>
  <li>obiskati sosedo.</li>
</ul>
```

4. podnaloga

Sestavite funkcijo `gnezdeni_seznami(vhod, izhod)`, ki bo podatke iz vhodne datoteke zapisala v izhodno datoteko v obliki neurejenega gnezdenega seznama. V vhodni datoteki je vsak element seznama v svoji vrstici, zamik pred elementom pa določa, kako globoko je element gnezden.

Na primer, če je v datoteki `seznami.txt` zapisano:

```
živali
  sesalci
    slon
  ptiči
    sinička
rastline
  sobne rastline
    difenbahija
```

bo po klicu `gnezdeni_seznami('seznami.txt', 'seznami.html')` v datoteki `seznami.html` zapisano:

```
<ul>
  <li>živali
    <ul>
      <li>sesalci
        <ul>
          <li>slon
        </ul>
      <li>ptiči
        <ul>
          <li>sinička
        </ul>
    </ul>
  <li>rastline
    <ul>
```

```
      <li>sobne rastline
        <ul>
          <li>difenbahija
        </ul>
    </ul>
</ul>
```

Značk `<li>` ne zapirajte.

Kolesarski maraton Franja

1. podnaloga

Napišite funkcijo `cas_maratona(niz)`, ki sprejme niz z doseženim časom na kolesarskem maratonu v obliki `'ure:minute:sekunde'` in vrne ta čas v obliki tabele s tremi elementi: `[ure, minute, sekunde]`. Predpostavite, da so vrednosti v nizu smiselne. Primer:

```
>>> cas_maratona('4:13:59')
[4, 13, 59]
```

2. podnaloga

Napišite funkcijo `vsi_casi(ime_datoteke)`, ki prebere vhodno datoteko, v kateri so shranjeni časi tekmovalcev in vrne tabelo trojčkov s temi časi. Vsaka vrstica vsebuje čas enega tekmovalca. Uporabite rešitev prve podnaloge!

Primer datoteke (maraton.txt):

```
4:13:59
5:45:12
3:17:25
8:4:35
2:58:26
3:1:15
```

Primer uporabe za zgornjo datoteko:

```
>>> vsi_casi('maraton.txt')
[[4, 13, 59], [5, 45, 12], [3, 17, 25], [8, 4, 35], [2, 58, 26], [3, 1, 15]]
```

3. podnaloga

Napišite funkcijo `nagrajeni(ime_datoteke, ura, minuta, sekunda)`, ki prebere čase tekmovalcev iz datoteke ter vrne število nagrajenih tekmovalcev (to so tisti, ki so dosegli enak ali boljši čas od podanega). Vhodna datoteka ima takšno obliko, kot je opisano v drugi podnalogi. Uporabite rešitev druge podnaloge!

Primer uporabe za datoteko iz druge podnaloge:

```
>>> nagrajeni('maraton.txt', 4, 0, 0)
```

```
3
```

4. podnaloga

Napišite funkcijo `urejeni_casi(ime_vhodne, ime_izhodne)`, ki prebere čase iz vhodne datoteke, jih uredi v naraščajočem vrstnem redu ter urejene zapiše v izhodno datoteko. Vhodna in izhodna datoteka imata enako obliko, tako kot je opisano v drugi podnalogi. Uporabite rešitev druge podnaloge!

Primer vhodne datoteke (maraton.txt):

```
4:13:59
5:45:12
3:17:25
8:4:35
2:58:26
3:1:15
```

Po klicu funkcije `urejeni_casi('maraton.txt', 'urejeno.txt')` mora imeti izhodna datoteka (urejeno.txt) naslednjo vsebino:

```
2:58:26
3:1:15
3:17:25
4:13:59
5:45:12
8:4:35
```

Nasvet 1: Uporabite vgrajeno funkcijo `sort`, ki zna sortirati tudi gnezdene tabele (v vašem primeru tabele trojčkov).

Nasvet 2: Pri oblikovanju vrstic izhodne datoteke si lahko pomagate s funkcijo `join`!

Dnevnik

S tekmovanja RTK 2014

1. podnaloga

V računalniškem sistemu imamo zabeležene dogodke in bi jih radi v strnjeni obliki shranjevali v seznam nizov. Vsak dogodek (kot na primer prijava ali odjava uporabnika, razne napake) je predstavljen z nizom znakov (kratko besedilo/vrstica), dolžine največ `strni_izpis(datot)`, ki naj bi brala dogodke iz vhodne datoteke in jih shranjevala v seznam nizov. Vendar pa je v programu nekaj napak, zaradi katerih ne deluje. Popravi napake v kodi.

```
def strni_izpis(datot):
    prejsnja = ""
    n = 0
    dat = with open(datot,'w', encoding='utf-8')
        dat = dat.readlines()
```

```
        sez_strnjenih = []
        for dogodek in dat:
            if n > 0 and prejsnja != dogodek:
                n += 2
                continue
            if n == 1:
                sez_strnjenih.append("%s" % prejsnja[:-1])
            elif n > 2:
                sez_strnjenih.append("ponovljeno se %d-krat" % (n))
            prejsnja = dogodek
            n = 1
            if dogodek:
                sez_strnjenih.append(dogodek[0])
            return sez_strnjenih
```

Vhodni podatki

Tekstovna datoteka `dogodki.txt` z zaporedjem dogodkov. Datoteka je kodirana v utf-8. Primer:

```
aaa
aaa
bbbbb
ccc
ccc
ccc
dd
dd
aaa
aaa
aaa
aaa
```

Izhodni podatki

Funkcija naj vrne seznam nizov, v katerem so strnjeni dogodki.

```
>>> strni_izpis(dogodki.txt)
['aaa', 'aaa', 'bbbbb', 'ccc', 'ponovljeno se 2-krat', 'dd', 'dd', 'aaa', 'ponovljeno s
e 3-krat']
```

Kolera

S tekmovanja RTK 2013

1. podnaloga

V devetnajstem stoletju se še ni kaj dosti vedelo o načinu prenašanja nalezljivih bolezni. Leta so imeli v Londonu velik izbruh črevesne bolezni kolere. Zdravnik John Snow je takrat s svojo analizo pokazal na vzročno zvezo med lokacijo londonskih vodnjakov in lokacijo domovanja obolelih. Izkazalo se je, da so žrtve pile vodo iz istega okuženega vodnjaka na ulici Broad Street. Zdravnik si je narisal zemljevid Londona, vanj vrisal lokacije vodnjakov in za vsak vodnjak z drugo barvo pobarval območje zemljevida, ki je temu vodnjaku najbližje. Ko je vrisal še točke domovanja obolelih, je postala vzročna zveza precej očitna, saj so prebivalci večinoma hodili po vodo k njim najbližjemu vodnjaku.
Nekoliko si poenostavimo zemljevid velemesta in denimo, da nas zanima le območje, ki ga razdelimo na kvadratno mrežo celic. V nekaterih celicah te mreže se nahajajo vodnjaki. Vseh vodnjakov je deset, njihove koordinate preberemo iz vhodne datoteke: v vsaki vrstici sta dve celi števili, in (obe sta z območja od do
`povrsine_vodnjakov(koordinate_vodnjakov)`, ki bo prebrala koordinate vodnjakov, potem pa vrnila seznam površin posameznih vodnjakov, tj. seznam s številom celic, za katere velja, da jim je v tej pravokotni mreži ta vodnjak najbližji.

```
def povrsine_vodnjakov(koordinate_vodnjakov):
    """Za vodnjake v vhodni datoteki izračuna, kolikšna površina pripada vsakemu vodnja
ku in vrne seznam površin."""
    vodnjaki = []
    with open(koordinate_vodnjakov, 'r') as vhod:
        # Shranimo koordinate vodnjakov
        for vrstica in vhod:
            (x, y) = vrstica.###.split(' ')
            vodnjaki.append((int(x), int(y)))
    povrsine = len(vodnjaki) * [0]
    # Za vsako celico poiščemo najbližji vodnjak
    for y in range():
        for x in range(1, 51):
            najkrajsa = 50 * 50 + 50 * 50   # vsak kvadrat razdalje dveh celic bo manjš
i od tega
            naj_vodnjak = 0
            for i in range(len(vodnjaki)):
                vodnjak = vodnjaki[i]
                razdalja = ###
                # Če je celica bližje temu vodnjaku od doslej najbližjega vodnjaka, si
to zapomnimo
                if vodnjak == vodnjaki[0] or razdalja < najkrajsa:
                    najkrajsa = razdalja
                    naj_vodnjak = i
        ### += 1
    return povrsine
```

Za merjenje razdalje med dvema celicama uporabimo Pitagorov izrek: kvadrat razdalje med in je `kolera.txt`.

Vhodni podatki

Funkciji kot parameter podamo datoteko `kolera.txt`, v kateri so koordinate vodnjakov.

Izhodni podatki

Seznam dolg kolikor je vodnjakov, v katerem je na mestu število celic, ki pripadajo

```
 5
33 8
1 27
23 43
25 45
12 8
10 29
3 5
23 6
12 32
```

Primer izhodnega podatka glede na zgornje vhodne podatke:

```
>>> povrsine_vodnjakov('kolera.txt')
[240, 420, 115, 325, 452, 206, 158, 97, 184, 303]
```

Računi

V datoteki imamo zapisane račune, vsak se začne z vrstico oblike `'RačunNNNNN'`, kjer je NNNNN zaporedna številka računa. Kupljeni izdelki so zapisani v naslednjih vrsticah v obliki `"izdelek:kolicina:cena_enega"`.

1. podnaloga

Sestavite funkcijo `preberi(datoteka)`, ki iz datoteke prebere račune in jih kot slovarje shrani v tabelo. Slovarji naj imajo za ključe izdelke, za vrednosti pa pare `(kolicina, cena_za_vse)`.

Če je v datoteki zapisano:

```
Račun23421
vino:2:10
voda:1:2
Račun64529
parfum:1:30
voda:5:2
čips:2:2.5
```

Naj funkcija vrne:

```
>>> preberi(datoteka)
```

```
  [{'vino': (2, 20), 'voda':(1, 2)}, {'parfum': (1, 30), 'voda': (5, 10), 'čips': (2, 5
)}]
```

2. podnaloga

Sestavite funkcijo `prodaja(ime_datoteke)`, ki bo iz datoteke kot prej prebrala račune in vrnila par: (zasluzek, prodano), kjer je `zasluzek` vsota zasluženega denarja, `prodano` pa nov slovar, v katerem so ključi izdelki, vrednosti pa število prodanih tovrstih izdelkov.

Podnapisi

Franci zelo rad gleda filme. Ker ne razume tujih jezikov, si s strani podnapisi.net sname podnapise. Datoteke s podnapisi (ki imajo končnico `.srt`) so takšne oblike:

```
1

00:08:51,520 --> 00:08:53,317

Pa je šla nevesta!


2

00:14:03,920 --> 00:14:08,357

Jebi ga, jaz bi ti pomagal,

samo šofer, Slovenec...


3

01:18:07,640 --> 01:18:12,589

Ima ta tvoja Špela na desni

joški materino znamenje? Ima?!


4

01:18:17,040 --> 01:18:22,194

Jebi ga, Božo.

Umiri malo žogo.
```

Za nas bo beseda *podnapisi* pomenila zaporedje manjših enot, ki jim bomo rekli *napisi*. Napisi so med seboj ločeni s po eno prazno vrstico. Vsak napis je sestavljen iz treh ali več vrstic. V prvi vrstici je zaporedna številka napisa, v drugi čas trajanja, od tretje dalje pa je besedilo.

Ker podnapise delajo amaterji, se včasih zgodi, da na neki točki začnejo podnapisi zaostajati ali pa prehitevati. To gre Franciju zelo na k[redacted]. Če boste sestavili program, ki bo sposoben podnapise popraviti, se vam obeta gajba …

Opomba: Zgornji zgled je iz filma Kajmak in marmelada.

1. podnaloga

Sestavite funkcijo `pretvori(h, m, s, milis)`, ki dobi čas v urah, minutah, sekundah in milisekundah ter ga preračuna v milisekunde. Zgled:

```
>>> pretvori(1, 18, 7, 640)
4687640
```

2. podnaloga

Sestavite še funkcijo `pretvori_nazaj(milis)`, ki naredi ravno obratno kot funkcija `pretvori`, tj. dobi čas v milisekundah in ga preračuna v ure, minute, sekunde in milisekunde. Primer:

```
>>> pretvori_nazaj(4687640)
(1, 18, 7, 640)
```

3. podnaloga

Sestavite funkcijo `popravi(vhod, izhod, t, d)`, ki kot argumente dobi:

- ime vhodne datoteke;
- item ime izhodne datoteke;
- čas (v milisekundah), kjer se zgodi časovni preskok, in
- dolžino časovnega preskoka (v milisekundah).

Funkcija naj prebere podnapise iz datoteke z imenom `vhod` in naj na datoteko z imenom `izhod` izpiše popravljene podnapise, tako da vse čase, ki so kasnejši od `t`, premakne za `d` milisekund naprej. Če je slučajno `d` negativno število, se časi seveda premaknejo nazaj. Predpostavite lahko, da se preskok nikoli ne zgodi sredi napisa, ampak vedno v *vmesnem času*.

Obe datoteki sta kodirani v UTF-8. Pri odpiranju datotek uporabite imenovani argument `encoding`, s katerim eksplicitno navedete kodno tabelo, takole: `open(vhod, encoding='utf-8')`.

Primer: Če je na datoteki kajmak.srt primer z začetka opisa naloge, naj bo po klicu funkcije

```
>>> popravi('kajmak.srt', 'popravljeno.srt', 3600000, 120000)
```

v datoteki popravljeno.srt besedilo:

```
1
00:08:51,520 --> 00:08:53,317
Pa je šla nevesta!


2
00:14:03,920 --> 00:14:08,357
Jebi ga, jaz bi ti pomagal,
samo šofer, Slovenec...


3
01:20:07,640 --> 01:20:12,589
Ima ta tvoja Špela na desni
joški materino znamenje? Ima?!


4
01:20:17,040 --> 01:20:22,194
```

```
Jebi ga, Božo.

Umiri malo žogo.
```

OOP I - lastnosti

lastnosti /property/

Uporaba dekoratorja 'property'

Pri preprostih razredih pogosto upravljamo direktno z lastnostmi objektov. Če želimo točko s koordinatama `x` in `y` premakniti za vektor `(1, 0)` to storimo tako, da popravimo koordinati. V mnogih primerih pa imajo lastnosti objektov omejitve, ali pa preprosto želimo abstrakcijo implementacije.

V teh primerih do lastnosti ne dostopamo direktno (takšne lastnosti predznačimo z _), temveč za njih spišem posebne metode za dostop. Za lažje delo v Pythonu uporabljamo dekorator `@property`, ki ga bomo spoznali v naslednjih podnalogah.

1. podnaloga

Točko v polarnih koordinatah podamo s kotom in razdaljo od izhodišča. Pri tem je razdalja nujno pozitivna, kot pa leži na intervalu
`SlabaPolarnaTocka`, za katero definirajte metodo `__init__(self, r, phi)`, ki ustvari. Če je razdalja `r` negativna, jo nastavimo na 0, na kot `phi` pa gledamo modulo

```
 a = SlabaPolarnaTocka(-10, 4)

>>> a.r

0

>>> a.phi
```

2. podnaloga

Napišite primer `pokvarjena_tocka`, ki pripada razredu `SlabaPolarnaTocka` ampak ima razdaljo `r` negativno.

Namig: Ustvarite poljubno točko, in jo pokvarite.

```
>>> pokvarjena_tocka.r < 0

True
```

3. podnaloga

Napišite razred `BoljsaPolarnaTocka`, ki ima namesto lastnosti `r` lastnost `_r` (podrčtaj označuje, da je uporabnik ne sme direktno spreminjati). Za dostop do razdalje definirajte metodi `nastavi_r(self, r)` in `vrednost_r(self)`, ki pravilno upravljata z lastnostjo `_r`. Metoda `__init__` naj že uporablja ti dve metodi za kreacijo točke.

Opomba: Za kot točke se trenutno ne zmenimo (lahko ga tudi izpustite).

```
>>> a = BoljsaPolarnaTocka(-10, 0)

0

>>> a.nastavi_r(5)

>>> a.vrednost_r()

5
```

4. podnaloga

Uporaba dodatnih metod za dostop do lastnosti je naporna, zato problem rešimo z dekoratorjem `@property`. Razred `PolarnaTocka` ze uporablja `@property` za razdaljo, vaša naloga pa je, da ta pristop uporabite še za kot.

Če lastnosti definiramo na takšen način, jih lahko uporabljamo kot običajno.

Opomba: Metoda za prikaz vrednosti lastnosti uporablja dekorator `@property`, metoda za nastavljanje vrednosti pa `@ime_lastnosti.setter`.

```
>>> a = PolarnaTocka(-5, 10)
>>> a.phi
0.5752220392306207
>>> a.phi = -2
>>> a.phi
1.1415926535897931
```

Kompleksna števila z lastnostmi

1. podnaloga

Sestavljen je razred KompleksnoStevilo z metodama `__init__(self, re, im)`, lastnostima `im` in `re` ter `__repr__(self)`. Realni del kompleksnega števila je shranjen v spremenljivki `_re`, imaginarni pa v `_im`. Metoda `__repr__(self)` predstavi kompleksno število z nizom oblike `KompleksnoStevilo(re, im)`.

Zgled:

```
>>> u = KompleksnoStevilo(3, 4)
>>> u
KompleksnoStevilo(3, 4)
```

Žal so se vrstice v kodi zamešale. uredi jih v pravilni vrstni red. Razen spreminanja vrstnega reda ne naredi nobene druge spremembe

---

Ulomki z lastnostmi

Razred Ulomki (osnova) iz sklopa Uvod v OOP bomo nadgradili z lastnostmi. Skopirajte svojo implementacijo razreda `Ulomek` iz sklopa Uvod v OOP in jo dopolnite po navodilih.

1. podnaloga

Sestavite razred `Ulomek`, s katerim predstavimo ulomek. Števec in imenovalec sta celi števili, pri čemer je imenovalec vedno pozitiven. Ulomki naj bodo vedno okrajšani.

Do števca lahko pridemo preko lastnosti `st`, do imenovalca pa preko lastnosti `im`. Če poskusimo spremeniti števec ali imenovalec (torej lastnosti `st` ali `im`) naj koda sproži napako z obvestilom "Obstoječega ulomka ne moremo spreminjati".

Namig: veljavnost argumentov preverite v metodi `__init__`, setter metodi (recimo `st.setter`) pa naj poskrbita, da vrednosti ne moremo pokvariti.

Tudi metodi `__str__` in `__repr__` popravite tako, da bosta do vrednosti dostopali preko lastnosti (property) in ne direktno preko "privatnih" atributov.

Zgled:

```
>>> u = Ulomek(5, -20)
>>> u.st
-1
>>> u.im
4
>>> u.st = 4
... Exception("Obstoječega ulomka ne moremo spreminjati")
```

Polinomi z lastnostmi

Kot pri nalogi Polinomi v sklopu Uvod v OOP sestavite razred `Polinom`, s katerim predstavimo polinom v spremenljivki
predstavimo s `Polinom([7, 2, 0, 1])`. Razmislite, kaj predstavlja `Polinom([])`. Zadnji koeficient v tabeli mora biti neničelen.
Pomembno: Pri tej nalogi poskrbite, da bo tabela koeficientov lastnost. Torej uporabite dekorator:

```
@property
def koef(self):
    # vrni vrednost
```

Če ne boste pravilno uporabili dekoratorja, boste dobili takole izjemo: `AttributeError: type object 'Polinom' has no attribute 'koef'`

1. podnaloga

Podan imate osnutek razreda s konstruktorjem `__init__(self, koef_pol)`, ki priredi vrednost spremenljivke koef_pol lastnosti `koef` objekta. Če kasneje spremenimo seznam, ki smo ga kot argument podali konstruktorju, se koeficienti polinoma ne bodo spremenili.

```
class Polinom:


    def __init__(self, koef_pol):
        # Znebimo se vseh ničelnih vodilnih koeficientov polinoma:
        zadnji = len(koef_pol) # vodilne nicle so kvecjemu zadaj
        while zadnji > 0 and koef_pol[zadnji - 1] == 0:
            zadnji -= 1
        self._koef = koef_pol[:zadnji]  # Rezina naredi kopijo seznama.
```

Sestavite lastnost `koef` kot zgoraj, ki bo vrnila koeficiente polinoma tako, da jih kasneje ne bo mogoče po pomoti spremeniti. Prav tako naj se koeficientov polinoma kasneje ne sme spremnijati (polinom torej ustvarimo ob inicializaciji!)

```
>>> p = Polinom([1, 2, 3])
>>> l = p.koef
>>> l.append(4)
>>> p.koef
[1, 2, 3]
>>> l
[1, 2, 3, 4]
```

Bančni račun

Naloga Bančni račun je nadgradnja naloge Bitni cekini iz sklopa `Uvod v OOP`. Če ste že rešili nalogo Bitni cekini, jo uporabite kot osnovo za reševanje te naloge. Če je še niste in imate z OOP še težave, vam priporočam, da jo rešite za vajo preden se lotite te naloge.

Razred BancniRacun bo poleg podatka o stanju vseboval tudi podatek o lastniku računa in bo implementiran z uporabo lastnosti (@property). Metode `__str__`, `dvig` in `polog` naj do vrednosti dostopajo preko lastnosti (metod z oznako `@property` oziroma `@nekaj.setter`).

1. podnaloga

Sestavite razred `BancniRacun` s konstruktorjem `__init__(self, stranka, začetno_stanje)`, ki sprejme ime lastnika in začetno stanje na računu. Lastnost, v katerega shranite lastnika, naj bo `lastnik` in naj se naknadno ne da spreminjati.

Objekti tipa `BancniRacun` naj imajo tudi lastnost `stanje`, ki mora biti vedno nenegativno decimalno število.

Parameter konstruktorja `začetno_stanje` naj bo neobvezen in v primeru, ko ni podan, naj bo začetno stanje enako nič.

2. podnaloga

Sestavite metodo `__str__(self)`, ki predstavi stanje na računu v obliki: `'Lastnik: <ime_lastnika>, stanje: <stanje_na_racunu>'`

Primer:

```
>>> racun = BancniRacun("Marko", 6)
>>> print(racun)
Lastnik: Marko, stanje: 6
```

3. podnaloga

Sestavite metodi `dvig(self, koliko)` in `polog(self, koliko)`, ki dvigneta oz. položita ustrezno količino bitnih cekinov na račun. Predpostavimo, da bo vrednost argumenta `koliko` vedno nenegativno celo število (ni potrebno preverjati).

Pri metodi `dvig` upoštevajte, da stanje na računu ne sme biti negativno. V takšnem primeru se dvig ne sme izvesti.

Metoda `dvig` naj vrne `True`, če je dvig uspel in `False`, če ni. Metoda `polog` naj spremeni stanje in vrne stanje na računu po pologu.

4. podnaloga

Sestavite funkcijo `prenesi(racun1, racun2, koliko)`, ki iz računa `racun1` prenese `koliko` cekinov na račun `racun2`. Funkcija `prenesi` naj ne bo znotraj razreda `BancniRacun`, saj ni objektna metoda, ampak je čisto običajna funkcija. Spremenljivki `racun1` in `racun2` sta seveda objekta tipa `BancniRacun`, kar ni potrebno preverjati!

Če na računu `racun1` ni dovolj denarja, se transakcija ne sme izvršiti, torej mora stanje na obeh računih ostati nespremenjeno. Funkcija naj vrne uspešnost transakcije (`True`, če je transakcija uspela, in `False`, če ni).

Kvadratni polinomi

1. podnaloga

Sestavimo razred `KvPol`, ki ga bomo uporabili za predstavitev kvadratnih polinomov, torej za "zadeve" oblike
`koef_a`, `koef_b`, `koef_c`. Polinom bomo ustvarili s klicem `KvPol(2, -3, 10)`
V razredu naj bosta še osnovni metodi za prikaz polinoma v obliki niza, ki se obnašata kot kaže zgled (ni najlepše, a to ni najpomembnejši del vaje)

```
>>> kv = KvPol(3, -3, 10)
>>> kv.koef_a
3
>>> kv.koef_a = 2
>>> repr(kv)
'KvPol(2, -3, 10)'
>>> str(kv)
2x**2 + -3x + 10
```

Namig: koeficiente lahko hranite kar v tabeli, a ta naj bo "privatna" (označena s _ na začetku imena).

2. podnaloga

Razred `KvPol` dopolni z metodo `vrednost`, ki izračuna vrednost polinoma za dani x.

```
>>> pol1 = KvPol(2, 1, 3)
>>> pol1.vrednost(0)
3
>>> pol1.vrednost(1)
6
```

Zajec

Jože goji zajce. V zadnjih letih so se tako namnožili, da si Jože enostavno ne more več zapomniti vseh. Zato potrebuje primeren informacijski sistem. V pomoč mu sestavite razred, ki bo vseboval vse potrebne podatke o vsakem zajcu.

1. podnaloga

Sestavite razred `Zajec` s konstruktorjem `__init__(self, teza, starost)`, ki predstavlja zajca z dano težo in starostjo. Lastnosti `teza` in `starost` naj bodo definirane kot `property`, saj zajci z negativno težo in starostjo ne obstajajo. Takšnim zajcom nastavite težo oz. starost na 0 (in se pretvarjajte, da obstajajo zajci brez teže).

2. podnaloga

Sestavite metodo `nahrani(self, hrana)`, kjer je argument `hrana` teža hrane, ki jo damo zajcu. Pri hranjenju se teža zajca poveča za 30 % teže hrane, ki jo zajec poje. Zgled:

```
>>> z = Zajec(5, 2)
>>> z.nahrani(2)
>>> z.teza
5.6
```

3. podnaloga

Sestavite metodo `__str__(self)`, ki vrne predstavitev razreda `Zajec` z nizom oblike `'Zajec težak X kg, star Y let.'`.

Primer:

```
>>> z = Zajec(5, 2)
>>> print(z)
'Zajec težak 5 kg, star 2 let.'
```

Opomba: Funkcija `print` na svojem argumentu pokliče metodo `__str__` in izpiše niz, ki ga ta metoda vrne. Metoda `__str__` običajno vrne razumljiv opis objekta, ki naj bi ga razumeli tudi ne-programerji.

4. podnaloga

Sestavite še metodo `__repr__`, ki vrne predstavitev razreda `Zajec` kot niz oblike `'Zajec(X, Y)'`, kjer je `X` teža, `Y` pa starost zajca.

Primer:

```
>>> z = Zajec(5, 2)
>>> z
Zajec(5, 2)
```

Opomba: Če v interaktivni konzoli pokličemo nek objekt, se izpiše niz, ki ga vrne klic metode `__repr__` na tem objektu. Priporočilo je, da je niz, ki ga vrne metoda `__repr__`, veljavna programska koda v Pythonu, ki ustvari identično kopijo objekta.

5. podnaloga

Sestavite metodo `__lt__(self, drugi)`, ki dva zajca primerja med sabo. Metoda naj vrne `True`, če je prvi zajec manjši od drugega in `False` sicer.

Manjši zajec je tisti, ki je lažji. Če pa imata zajca enako maso, je manjši tisti, ki je mlajši (tj. ima manjše število let).

```
>>> Zajec(5, 3) < Zajec(6, 2)
```

```
True
>>> Zajec(3, 1) < Zajec(2, 2)
False
>>> Zajec(4, 3) < Zajec(4, 2)
False
```

6. podnaloga

Sestavite funkcijo `uredi(teze, starosti)`. Argumenta `teze` in `starosti` sta enako dolga seznama števil, kjer `uredi` naj ne bo znotraj razreda `Zajec`, saj ni objektna metoda, ampak je čisto običajna funkcija.
Funkcija naj ustvari seznam ustreznih primerkov razreda `Zajec`, ga uredi po velikosti glede na zgoraj opisano relacijo in ta seznam vrne kot rezultat.

Namig: metodo `__lt__` ste že definirali, torej jo Python lahko uporablja v svojih funkcijah.

```
>>> l = uredi([5, 4, 4], [3, 2, 3])
>>> for z in l:
...     print(z)
...
Zajec težak 4 kg, star 2 let.
Zajec težak 4 kg, star 3 let.
Zajec težak 5 kg, star 3 let.
```

Datumi

Koledar, ki ga trenutno uporabljamo v zahodnem svetu, se imenuje gregorijanski koledar. Pri tej nalogi bomo implementirali razred `Datum`, ki bo omogočal predstavitev datumov v gregorijanskem koledarju in računanje z njimi.

1. podnaloga

Sestavite funkcijo `je_prestopno(leto)`, ki preveri, ali je dano `leto` prestopno (po gregorijanskem koledarju). Zgled:

```
>>> je_prestopno(2004)
True
>>> je_prestopno(1900)
False
```

2. podnaloga

Sestavite funkcijo `stevilo_dni(leto)`, ki vrne število dni v danem letu. Zgled:

```
>>> stevilo_dni(2015)
365
>>> stevilo_dni(2016)
```

```
366
```

Nasvet: Uporabite funkcijo `je_prestopno`.

3. podnaloga

Sestavite funkcijo `dolzine_mesecev(leto)`, ki vrne seznam dolžine 12, ki ima za elemente števila dni po posameznih mesecih v danem letu. Zgled:

```
>>> dolzine_mesecev(2015)
[31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
```

4. podnaloga

Sestavite metodo `je_veljaven(d, m, l)`, ki preveri, ali d. m. l predstavlja veljaven datum. Pri tem predpostavi, da so d, m in l zagotovo cela števila Zgled:

```
>>> je_veljaven(8, 2, 1849)
True
>>> je_veljaven(5, 14, 2014)
False
```

5. podnaloga

Definirajte razred `Datum`, s katerim predstavimo datum. Najprej sestavite konstruktor `__init__(self, dan, mesec, leto)`. Nastali objekt naj ima *bralne* lastnosti `dan`, `mesec` in `leto`. Zgled:

```
>>> d = Datum(8, 2, 1849)
>>> d.dan
8
>>> d.mesec
2
>>> d.leto
1849


>>> d.dan = 12
Traceback (most recent call last):
   ...
AttributeError: can't set attribute
```

V primeru, da dan, mesec in leto ne predstavljajo veljavnega datuma, ustvari datum 1. 1. 2020

6. podnaloga

Sestavite metodo `__str__(self)`, ki predstavi datum v obliki `'dan. mesec. leto'`. Zgled:

```
>>> d = Datum(8, 2, 1849)
>>> print(d)
8. 2. 1849
```

7. podnaloga

Sestavite metodo `spremeni(self, dan=None, mesec=None, leto=None)`, ki nastavi datum na `dan. mesec. leto` in vrne `True`. Če je vrednost parametra None, uporabi obstoječi dan (oz. mesec, leto). Če datum `dan. mesec. leto` ni pravilen, naj metoda ne naredi nič le vrne `False`.

```
>>> d = Datum(8, 2, 1849)
>>> print(d)
8. 2. 1849
>>> d.spremeni(30, 4, 1849)
```

8. podnaloga

Sestavite še metodo `__repr__(self)`, ki vrne niz oblike `'Datum(dan, mesec, leto)'`. Zgled:

```
>>> d = Datum(8, 2, 1849)
>>> d
Datum(8, 2, 1849)
```

9. podnaloga

Sestavite metodo `dan_v_letu(self)`, ki izračuna, koliko dni je minilo od začetka leta do danega datuma. Zgled:

```
>>> d = Datum(1, 11, 2014)
>>> d.dan_v_letu()
305
```

Nasvet: Ali si lahko kako pomagate s funkcijo `dolzine_mesecev`?

10. podnaloga

Sestavite metodo `razlika(self, other)`, ki kot argument dobi še en objekt razreda `Datum` in vrne število dni med datumoma. Dobra rešitev deluje *brez* uporabe zanke po vseh letih med danima datuma (ima časovno zahtevnost

```
 datum1 = Datum(1, 11, 2014)
>>> datum2 = Datum(14, 10, 2014)
>>> datum1.razlika(datum2)
18
```

Namig: Najprej sestavite pomožno metodo `dni_od_zacetka(self)`, ki izračuna, koliko dni je minilo od "začetka štetja" (`leto==0`, `mesec=1`, `dan=1`). Razliko v dnevih med `self` in `other` lahko nato preprosto izračunate z metodo `dni_od_zacetka`.

Opomba: Gregorijanski koledar seveda ni bil v veljavi leta 0, ampak za potrebe računanja se lahko pretvarjamo, da ga je uporabljal tudi Julij Cezar.

11. podnaloga

Sestavite metodo `dan_v_tednu(self)`, ki vrne številko dneva v tednu (1 = ponedeljek, 2 = torek, …, 7 = nedelja). Lahko si pomagate z Zellerjevim obrazcem. Druga možnost je, da izračunate razliko med datumom `self` in nekim fiksnim datumom, za katerega že poznate dan v tednu. Zgled:

```
>>> d = Datum(1, 11, 2014)
```

```
>>> d.dan_v_tednu()
6
```

12. podnaloga

Sestavite *funkcijo* `datum_iz_dneva_v_letu(leto, dan)`, ki za parametra dobi leto in zaporedno številko dneva v letu ter sestavi in vrne ustrezen datum.

Zgled:

```
>>> datum_iz_dneva_v_letu(2014, 305)
Datum(1, 11, 2014)
```

Študenti

Sestavili bomo elektronski indeks študenta.

1. podnaloga

Ustvari razred `Ocena`, ki ima naslednje lastnosti: niz `predmet`, trojico treh števil `datum`, predstavljeno z naborom oz. "tuplom", niz `tip`, ki je bodisi `"ustna"` bodisi `"pisna"`, `stevilcna_ocena`, ki je celo število od 1 do 10 vključno z njima in `utez`, ki je poljubno realno število od 0 do 1 vključno z njima. Vse lastnosti naj se nastavijo v konstruktorju. Le `datum` in `stevilcna_ocena` naj bosta nastavljivi lastnosti, ostale naj bodo zgolj bralne. Ustreznost vseh tipov moraš preveriti!

Datum bo podan v obliki niza "dd. mm. llll", kjer sta za prvi januar 2000 ustrezna oba naslednja zapisa: "01. 01. 2000" in "1. 1. 2000". Leto mora biti celo število od 1919 do trenutnega leta vključno z njima. Če je datum pravilne oblike, ti ni treba preverjati, če res obstaja. Recimo "30. 02. 1970" je z vidika naloge ustrezen datum. Koda, ki preverja ustreznost datuma je že napisana, a ima nekaj napak, ki jih moraš popraviti. Ostalo kodo moraš napisati sam/a.

Lastnost `utez` pove, kolikšen delež k skupni oceni pri nekem predmetu prispeva pisna in koliko ustna ocena. Lahko predpostaviš, da bo vsota uteži za posamezni predmet vedno 1.

Neobvezno: dodaj še metodo `__repr__`, ki vrne oceno v zapisu, s katerim lahko ustvarimo kopijo ocene (torej standardno delovanje). Bodi pozoren na zadnjo vrstico zgleda: prvi trije argumenti znotraj oklepajev morajo biti nizi!

Zgled (zadnji dve vrstici prikazujeta delovanje metode `__repr__`):

```
>>> ocena = Ocena('Matematika 1', '05. 06. 2007', 'pisna', 8, 0.5)
>>> ocena.predmet
'Matematika 1'
>>> ocena.datum
(5, 6, 2007)
>>> Ocena('Numerične metode 1', (5, 7, 2005), 'pisna', 8, 0.5)
Traceback (most recent call last):
 ...
AttributeError: 'tuple' object has no attribute 'split'


During handling of the above exception, another exception occurred:
```

```
...
TypeError
>>> Ocena('Programiranje 3', '8. 7. 2011', 'ustna', 7, 13)
...
ValueError
>>>
>>> ocena
Ocena("Matematika 1", "5. 6. 2007", "pisna", 8, 0.5)
>>>
```

2. podnaloga

Znotraj razreda `Ocena` ustvari metodo `popravi_oceno(self, nova_ocena)`, ki lastnost `stevilcna_ocena` nastavi na vrednost `nova_ocena`, datum ocene pa nastavi na današnji dan.

Zgled:

```
>>> ocena = Ocena('Algebra in diskretna matematika', '31. 1. 2013', 'pisna', 6, 0.8)
>>> ocena.stevilcna_ocena
6
>>> ocena.datum
(31, 1, 2013)
>>> ocena.popravi_oceno(9)
>>> ocena
Ocena(Algebra in diskretna matematika, 11. 5. 2020, pisna, 9, 0.8)
>>> ocena.stevilcna_ocena
9
>>> ocena.datum
(11, 5, 2020)
>>> ocena.tip
'pisna'
```

3. podnaloga

Ustvari razred `Student`, ki ima naslednje lastnosti: niz `ime`, `spol`, ki je bodisi "Ž" bodisi "M", `leto_rojstva`, ki je celo število od 1800 do trenutnega leta vključno z njima in tabelo `tabela_ocen`. Le zadnja naj bo nastavljiva, vse ostale pa zgolj bralne. Preveriti moraš ustreznost vseh tipov! Za podano tabelo ocen moraš preveriti, če je res tabela in če so vsi njeni elementi tipa `Ocena`. Zadnje storiš takole: `isinstance(element_tabele, Ocena)`.

Zgled:

```
>>> tabela_ocen = [Ocena('Matematika 1', '05. 06. 2011', 'pisna', 8, 0.5),
                   Ocena('Matematika 1', '1. 7. 2011', 'ustna', 7, 0.5),
                   Ocena('Računalniški praktikum', '3. 2. 2011', 'pisna', 9, 1)]
>>> studentka = Student('Sabina Sabinovič', 'Ž', 1991, tabela_ocen)
```

```
>>> studentka.ime
'Sabina Sabinovič'
>>> studentka.tabela_ocen
[Ocena(Matematika 1, 5. 6. 2011, pisna, 8, 0.5), Ocena(Matematika 1, 1. 7. 2011, ustna,
 7, 0.5), Ocena(Računalniški praktikum, 3. 2. 2011, pisna, 9, 1)]
>>> studentka.tabela_ocen[1].datum
(1, 7, 2011)
>>> studentka = Student('Sabina Sabinovič', 'ženska', 1991, tabela_ocen)
Traceback (most recent call last):
  ...
TypeError
>>> studentka = Student('Sabina Sabinovič', 'Ž', 1991, [10, 7])
Traceback (most recent call last):
  ...
TypeError
>>>
```

4. podnaloga

Znotraj razreda `Student` ustvari metodo `dodaj_oceno(self, ocena)`, ki v tabelo ocen študenta doda oceno. Metoda naj preveri, če je argument ocena res tipa `Ocena`. Če ni, naj sproži napako.

Zgled:

```
>>> ocena1 = Ocena('Matematika 1', '05. 06. 2007', 'pisna', 8, 0.5)
>>> ocena2 = Ocena('Matematika 1', '1. 7. 2007', 'ustna', 7, 0.5)
>>> student = Student('Simon Simonovič', 'M', 1987, [ocena1])
>>> student.dodaj_oceno(ocena2)
>>> student.tabela_ocen
[Ocena(Matematika 1, 5. 6. 2007, pisna, 8, 0.5), Ocena(Matematika 1, 1. 7. 2007, ustna,
 7, 0.5)]
>>> student.dodaj_oceno(7)
Traceback (most recent call last):
  ...
TypeError
>>>
```

5. podnaloga

V razredu `Student` ustvari metodo `ocene_v_mesecu(self, datum)`, ki za podani mesec, zapisan v obliki "mm. llll" ali "m. llll" (kot v prvi podnalogi), vrne tabelo vseh ocen, ki jih je študent pridobil v danem mesecu. Če ocen v danem mesecu ni ali če je datum neveljaven, naj metoda vrne prazen seznam. Lahko predpostaviš, da bo datum zapisan v predpisani obliki.

Zgled:

```
>>> tabela_ocen = [Ocena('Matematika 1', '05. 06. 2011', 'pisna', 8, 0.5),
                   Ocena('Matematika 1', '30. 6. 2011', 'ustna', 7, 0.5),
                   Ocena('Računalniški praktikum', '3. 2. 2011', 'pisna', 9, 1)]
>>> študent = Student('Dunja Simonova', 'Ž', 1991, tabela_ocen)
>>> študent.ocene_v_mesecu('6. 2011')
[Ocena(Matematika 1, 5. 6. 2011, pisna, 8, 0.5), Ocena(Matematika 1, 30. 6. 2011, ustna
, 7, 0.5)]
>>> študent.ocene_v_mesecu('07. 2011')
[]
>>>
```

6. podnaloga

Znotraj razreda `Student` ustvari metodo `stevilo_ocen(self, tip)`, ki vrne število vseh ocen zahtevanega tipa, ki jih je pridobil študent. Argument `tip` naj ne bo obvezen. V tem primeru naj metoda vrne število vseh ocen, ne glede na njihov tip.

Zgled:

```
>>> ocena1 = Ocena('Matematika 1', '05. 06. 2007', 'pisna', 8, 0.5)
>>> ocena2 = Ocena('Matematika 1', '1. 7. 2007', 'ustna', 7, 0.5)
>>> student = Student('Aleksej Aleksandrovič', 'M', 1990, [ocena1, ocena2])
>>> student.stevilo_ocen()
2
>>> student.stevilo_ocen('pisna')
1
>>> student.stevilo_ocen('ustna')
1
>>>
```

7. podnaloga

Znotraj razreda `Student` sestavi metodo `povprecna_ocena(self, tip)`, ki vrne povprečje vseh študentovih ocen zahtevanega tipa, zaokroženo na štiri decimalna mesta. Argument `tip` naj ne bo obvezen in naj se obnaša enako kot v metodi `stevilo_ocen`.

Zgled:

```
>>> ocena1 = Ocena('Matematika 1', '05. 06. 2007', 'pisna', 8, 0.4)
>>> ocena2 = Ocena('Matematika 1', '1. 7. 2007', 'ustna', 7, 0.6)
>>> ocena3 = Ocena('Računalniški praktikum', '3. 2. 2011', 'pisna', 9, 1)
>>> student = Student('Aleksej Aleksandrovič', 'M', 1990, [ocena1, ocena2, ocena3])
>>> student.povprecna_ocena()
8.0
>>> student.povprecna_ocena('pisna')
8.5
```

```
>>> student.povprecna_ocena('ustna')
7.0
>>>
```

8. podnaloga

V razredu `Student` ustvari metodo `ocena_pri_predmetu(self, predmet)`, ki izračuna uteženo oceno predmeta. Vsaka oceni je pripisana utež, ki pove, kolikšen delež k skupni oceni prispeva. Predmet nima nujno natanko dveh oceh, lahko pa predpostaviš, da je vsota vseh uteži ocen posameznega predmeta vedno 1.

Ker Python računa z binarnimi približki števil, lahko tudi tu dobiš kot rezultat periode, čeprav se z vidika formule ne bi smele pojaviti. Zato tudi tu rezultat zaokroži na štiri decimalna mesta.

Zgled:

```
>>> ocena1 = Ocena('Matematika 1', '05. 06. 2007', 'pisna', 8, 0.3)
>>> ocena2 = Ocena('Matematika 1', '1. 7. 2007', 'ustna', 7, 0.7)
>>> ocena3 = Ocena('Računalniški praktikum', '3. 2. 2011', 'pisna', 9, 1)
>>> student = Student('Aleksej Aleksandrovič', 'M', 1990, [ocena1, ocena2, ocena3])
>>> student.ocena_pri_predmetu('Matematika 1')
7.3
>>>
```

9. podnaloga

Znotraj razreda `Student` sestavi metodo `ocene_po_predmetih(self)`, ki vrne slovar, čigar ključi so imena predmetov, vrednosti pa tabele ocen, ki jih je študent pridobil pri ustreznem predmetu.

Zgled:

```
>>> ocena1 = Ocena('Matematika 1', '05. 06. 2007', 'pisna', 8, 0.3)
>>> ocena2 = Ocena('Matematika 1', '1. 7. 2007', 'ustna', 7, 0.7)
>>> ocena3 = Ocena('Računalniški praktikum', '3. 2. 2011', 'pisna', 9, 1)
>>> student = Student('Aleksej Aleksandrovič', 'M', 1990, [ocena1, ocena2, ocena3])
>>> student.ocene_po_predmetih()
{'Matematika 1': [Ocena(Matematika 1, 5. 6. 2007, pisna, 8, 0.3), Ocena(Matematika 1, 1
. 7. 2007, ustna, 7, 0.7)], 'Računalniški praktikum': [Ocena(Računalniški praktikum, 3.
 2. 2011, pisna, 9, 1)]}
>>>
```

10. podnaloga

Znotraj razreda `Student` ustvari metodo `neustrezni_predmeti(self)`, ki vrne dve množici: v prvi množici so imena vseh predmetov, pri katerem je vsota uteži vseh ocen manjša od 1, v drugi množici pa vsi predmeti, kjer je vsota uteži vseh ocen večja od 1. Tip ocen nas ne zanima.

Del kode je že napisan in ima natanko eno napako. Odpravi jo in dopolni kodo!

Zgled:

```
>>> ocena1 = Ocena('Matematika 1', '05. 06. 2007', 'pisna', 8, 0.3)
```

```
>>> ocena2 = Ocena('Matematika 1', '1. 7. 2007', 'ustna', 7, 0.7)
>>> ocena3 = Ocena('Računalniški praktikum', '3. 2. 2011', 'pisna', 9, 1)
>>> ocena4 = Ocena('Statistika', '5. 11. 2008', 'ustna', 6, 0.5)
>>> ocena5 = Ocena('Optimizacija', '13. 8. 1999', 'ustna', 7, 0.7)
>>> ocena6 = Ocena('Optimizacija', '4. 7. 1999', 'pisna', 7, 0.5)
>>> študent = Student('Mirko', 'M', 1970, [ocena1, ocena2, ocena3, ocena4, ocena5, ocen
a6])
>>> študent.neustrezni_predmeti()
({'Statistika'}, {'Optimizacija'})
>>>
```

OOP II - "magične" metode

"Magične" metode

Kvadratni polinomi 2

Prva in četrta podnaloga sta enaki kot v sklopu OOP I (Kvadratni polinomi). Če ste nalogo rešili v sklopu prejšnjih vaj, lahko rešitev uporabite pri tej nalogi, seveda pa ni prepovedano naloge rešiti ponovno za vajo.

1. podnaloga

Sestavimo razred `KvPol`, ki ga bomo uporabili za predstavitev kvadratnih polinomov, torej za "zadeve" oblike `koef_a`, `koef_b`, `koef_c`. Polinom bomo ustvarili s `KvPol(2, -3, 10)`
V razredu naj bosta še osnovni metodi za prikaz polinoma v obliki niza, ki se obnašata kot kaže zgled

```
>>> kv = KvPol(2, -3, 10)
>>> kv.koef_a = 3
>>> repr(kv)
'KvPol(3, -3, 10)'
>>> str(kv)
3x**2 + -3x + 10
```

2. podnaloga

Razred `KvPol` dopolni, tako, da bomo lahko seštevali polinome.

```
>>> pol1 = KvPol(2, 1, 3)
>>> pol2 = KvPol(1, -1, 2)
>>> pol1 + pol2
'KvPol(3, 0, 5)'
```

Namig: pomagajte si z metodo `__add__`, ki mora za dva vhodna polinoma vrniti nov polinom, ki predstavlja njuno vsoto.

3. podnaloga

Razred `KvPol` dopolni tako, da bomo lahko polinome množili s faktorjem

```
>>> pol1 = KvPol(2, 1, 3)
>>> pol2 = KvPol(2, 1, 2)
>>> 2 * pol1
'KvPol(4, 2, 6)'
>>> pol2 * 1.5
'KvPol(3, 1.5, 3)'
```

Namig: pomagajte si z metodo `__mul__`, ki mora za vhodni polinom ter številski faktor nov polinom, pomnožen za ta faktor.

4. podnaloga

Razred `KvPol` dopolni z metodo `vrednost`, ki izračuna vrednost polinoma za dani x.

```
>>> pol1 = KvPol(2, 1, 3)
>>> pol1.vrednost(0)
3
>>> pol1.vrednost(1)
6
```

Polinomi 3

Kot pri nalogi Polinomi v sklopu Uvod v OOP sestavite razred `Polinom`, s katerim predstavimo polinom v spremenljivki
predstavimo s `Polinom([7, 2, 0, 1])`. Razmislite, kaj predstavlja `Polinom([])`. Zadnji koeficient v tabeli mora biti neničelen.
Pomembno: Pri tej nalogi poskrbite, da bo tabela koeficientov lastnost. Torej uporabite dekorator:

```
@property
def koef(self):
    # vrni vrednost
```

Če ne boste pravilno uporabili dekoratorja, boste dobili takole izjemo: `AttributeError: type object 'Polinom' has no attribute 'koef'`

1. podnaloga

Podan imate osnutek razreda s konstruktorjem `__init__(self, koef_pol)`, ki priredi vrednost spremenljivke koef_pol lastnosti `koef` objekta. Če kasneje spremenimo seznam, ki smo ga kot argument podali konstruktorju, se koeficienti polinoma ne bodo spremenili.

```
class Polinom:


    def __init__(self, koef_pol):
        # Znebimo se vseh ničelnih vodilnih koeficientov polinoma:
        zadnji = len(koef_pol) # vodilne nicle so kvecjemu zadaj
        while zadnji > 0 and koef_pol[zadnji - 1] == 0:
            zadnji -= 1
        self._koef = koef_pol[:zadnji]  # Rezina naredi kopijo seznama.
```

Sestavite lastnost `koef` kot je opisano v navodilih naloge.
Pazite, da bo vrnila koeficiente polinoma tako, da jih kasneje ne bo mogoče po pomoti spremeniti. Prav tako se koeficientov polinoma kasneje ne sme spreminjati (polinom torej ustvarimo ob inicializaciji!)

```
>>> p = Polinom([1, 2, 3])
>>> l = p.koef
>>> l.append(4)
```

```
>>> p.koef
[1, 2, 3]
>>> l
[1, 2, 3, 4]
```

2. podnaloga

Sestavite metodo `__eq__(self, other)` za primerjanje polinomov. Zgled:

```
>>> Polinom([3, 2, 0, 1]) == Polinom([3, 2])
False
>>> Polinom([3, 2, 1, 0]) == Polinom([3, 2, 1])
True
```

3. podnaloga

Sestavite metodo `__call__(self, x)`, ki izračuna in vrne vrednost polinoma v `x`. Pri izračunu vrednosti uporabite Hornerjev algoritem. Če definiramo metodo `__call__`, objekt postane "klicljiv" (tj. lahko ga kličemo, kakor da bi bil funkcija). Zgled:

```
>>> p = Polinom([3, 2, 0, 1])
>>> p(1)
6
>>> p(-3)
-30
>>> p(0.725)
4.8310781249999994
```

4. podnaloga

Sestavite metodo `__add__(self, other)` za seštevanje polinomov. Metoda naj sestavi in vrne nov objekt razreda `Polinom`, ki bo vsota polinomov, ki sta operanda pri `+` Zgled:

```
>>> Polinom([1, 0, 1]) + Polinom([4, 2])
Polinom([5, 2, 1])
```

Pozor: Pri seštevanju se lahko zgodi, da se nekateri koeficienti pokrajšajo: `stopnja(self)`, ki vrne stopnjo polinoma. Uporabite jo, da sestavite metodo `__mul__(self, other)` za množenje polinomov. Metoda naj sestavi in vrne nov objekt razreda `Polinom`, ki bo produkt polinomov, operandov pri `*`. Zgled:

```
>>> Polinom([1, 0, 1]) * Polinom([4, 2])
Polinom([4, 2, 4, 2])
```

Kompleksna števila III

Prvi dve nalogi sta iz Kompleksna števila I v OOP I.

1. podnaloga

Sestavljen je razred KompleksnoStevilo z metodama `__init__(self, re, im)`, lastnostma `im` in `re` ter `__repr__(self)`. Realni del kompleksnega števila je shranjen v spremenljivki `_re`, imaginarni pa v `_im`. Metoda `__repr__(self)` predstavi kompleksno število z nizom oblike `KompleksnoStevilo(re, im)`.

Zgled:

```
>>> u = KompleksnoStevilo(3, 4)
>>> u
KompleksnoStevilo(3, 4)
```

Žal so se vrstice v kodi zamešale. uredi jih v pravilni vrstni red. Razen spreminanja vrstnega reda ne naredi nobene druge spremembe

2. podnaloga

Razredu dodaj metodo `__str__(self)`, ki kompleksno stevilo predstavi z nizom oblike npr. `3 + 4i` Pri tem bodi pozoren da: ◦ Če je realni ali imaginarni del števila enak `0`, naj bo v nizu njegov člen izpuščen (npr. namesto `2 + 0i` pišemo samo `2`). ◦ Če je imaginarni del števila enak `1`, namesto `1i` pišemo samo `i`. Če je enak `-1`namesto `-1i` pišemo `-i`. ◦ Če je imaginarni del števila negativen, njegov predznak zamenja `+`. Torej, namesto `2 + (-3)i` pišemo kar `2 - 3i`. Če je realni del enak `0` med predznakom `-` in nadaljevanjem ni presledka. Torej namesto `- 3i` pišemo `-3i`.

Zgled:

```
>>> u = KompleksnoStevilo(3, 4)
>>> print(u)
3 + 4i
>>> v = KompleksnoStevilo(2, 0)
>>> print(v)
2
>>> w = KompleksnoStevilo(0, -4)
>>> print(w)
-4i
>>> w = KompleksnoStevilo(0, 1)
>>> print(w)
i
>>> y = KompleksnoStevilo(-2, -6)
>>> print(y)
-2 - 6i
>>> z = KompleksnoStevilo(0, 0)
>>> print(z)
0
```

3. podnaloga

Razširite razred tako, da podpira seštevanje dveh kompleksnih števil. Zgled:

```
>>> KompleksnoStevilo(3, -2) + KompleksnoStevilo(2, 1)
KompleksnoStevilo(5, -1)
```

4. podnaloga

Razširite razred tako, da podpira seštevanje kompleksnih števil z decimalnimi in celimi števili.
Zgled:

```
>>> str(KompleksnoStevilo(3, -2) + KompleksnoStevilo(2, 1))
5 - i
>>> str(KompleksnoStevilo(3, 2) + 4.5)
7.5 + 2i
>>> str(5 + KompleksnoStevilo(3, -2))
8 - 2i
>>> str("bla" + KompleksnoStevilo(3, -2))
...TypeError('Kompleksna števila ne znamo seštevati s/z str')
```

5. podnaloga

Sestavite metodo `__mul__(self, other)`, ki vrne produkt dveh kompleksnih števil. Množenec na levi bo vedno tipa `KompleksnoStevilo`, množenec na desni pa je lahko tudi navadno število tipa `float` ali `int`.

Zgled:

```
>>> KompleksnoStevilo(3, 4) * 2
KompleksnoStevilo(6, 8)
>>> KompleksnoStevilo(3, 4) * KompleksnoStevilo(2, -1)
KompleksnoStevilo(10, 2)
```

6. podnaloga

Omogočite, da lahko med sabo poljubno množimo kompleksna, cela in decimalna števila.

Zgled:

```
>>> d1 = KompleksnoStevilo(3, 4)
>>> d2 = KompleksnoStevilo(2, -1)
>>> d1 * d2
KompleksnoStevilo(10, 2)
>>> d1 * 3
KompleksnoStevilo(9, 12)
>>> 2.2 * d2
KompleksnoStevilo(4.4, -2.2)
>>> 7 * 3
21
```

7. podnaloga

Izven razreda KompleksnoStevilo sestavite funkcijo `vsota_kompleksnih`, ki sprejme tabelo kompleksnih števil in vrne vsoto števil v tabeli.

```
>>> kompleksna = [KompleksnoStevilo(0, 0), KompleksnoStevilo(3, 4), KompleksnoStevilo(2
, 2)]
>>> vsota_kompleksnih(kompleksna)
KompleksnoStevilo(5, 6)
```

Ulomki II

Prve tri podnaloge ste že reševali pri OOPI pri nalogi Ulomki

Da ne bo težav pri testiranju, pri vseh podnalogah začnemo z

```
class Ulomek(Ulomek):
```

Seveda pa 1. podnalogo še vedno začnemo z

```
class Ulomek:
```

1. podnaloga

Sestavite razred `Ulomek`, s katerim predstavimo ulomek. Števec in imenovalec sta celi števili, pri čemer je imenovalec vedno pozitiven. Ulomki naj bodo vedno okrajšani.

Do števca lahko pridemo preko lastnosti `st`, do imenovalca pa preko lastnosti `im`. Če poskusimo spremeniti števec ali imenovalec (torej lastnosti `st` ali `im`) naj koda sproži napako z obvestilom "Obstoječega ulomka ne moremo spreminjati" Zgled:

```
>>> u = Ulomek(5, -20)
>>> u.st
-1
>>> u.im
4
>>> u.st = 4
... Exception("Obstoječega ulomka ne moremo spreminjati")
```

2. podnaloga

Trenutna implementacija razreda `Ulomek` objekt `Ulomek(5, 20)` izpiše z nizom oblike `<__main__.Ulomek object at 0x000002CB41CDD2B0>`, iz katerega ne moremo razbrati za kateri ulomek gre. Kako naj se objekt našega razreda izpiše, lahko sami določimo z metodama `__str__` in `__repr__`. Kako metodi delujeta in kakšna je razlika med njima si poglejte v videu.

Sestavite metodo `__str__(self)`, ki predstavi ulomek z nizom oblike `'st/im'`. Zgled:

```
>>> u = Ulomek(5, 20)
>>> print(u)
1/4
```

3. podnaloga

Sestavite še metodo `__repr__(self)`, ki predstavi ulomek z nizom oblike `'Ulomek(st, im)'`. Zgled:

```
>>> u = Ulomek(5, 20)
>>> u
Ulomek(1, 4)
```

4. podnaloga

Sestavite metodo `__eq__(self, other)`, ki vrne `True` če sta dva ulomka enaka, in `False` sicer. Ko definirate to metodo, lahko ulomke primerjate kar z operatorjem `==`. Zgled:

```
>>> Ulomek(1, 3) == Ulomek(2, 3)
False
>>> Ulomek(2, 3) == Ulomek(10, 15)
True
```

Lahko prepodstavite, da je drugi parameter (desni operand pri ==) zagotovo objekt iz razreda `Ulomek`.

5. podnaloga

Sestavite metodo `__add__(self, other)`, ki vrne vsoto dveh ulomkov. Ko definirate to metodo, lahko ulomke seštevate kar z operatorjem `+`. Na primer:

```
>>> Ulomek(1, 6) + Ulomek(1, 4)
Ulomek(5, 12)
```

Lahko prepodstavite, da je drugi parameter (operand pri +) zagotovo objekt iz razreda `Ulomek`.

6. podnaloga

Sestavite metodo `__sub__(self, other)`, ki vrne razliko dveh ulomkov. Ko definirate to metodo, lahko ulomke odštevate kar z operatorjem `-`. Na primer:

```
>>> Ulomek(1, 4) - Ulomek(1, 6)
Ulomek(1, 12)
```

Lahko prepodstavite, da je drugi parameter (desni operand pri -) zagotovo objekt iz razreda `Ulomek`.

7. podnaloga

Sestavite metodo `__mul__(self, other)`, ki vrne zmnožek dveh ulomkov. Ko definirate to metodo, lahko ulomke množite kar z operatorjem `*`. Na primer:

```
>>> Ulomek(1, 3) * Ulomek(1, 2)
Ulomek(1, 6)
```

Lahko prepodstavite, da je drugi parameter (desni operand pri *) zagotovo objekt iz razreda `Ulomek`.

8. podnaloga

Sestavite metodo `__truediv__(self, other)`, ki vrne kvocient dveh ulomkov. Ko definirate to metodo, lahko ulomke delite kar z operatorjem `/`. Na primer:

```
>>> Ulomek(1, 6) / Ulomek(1, 4)
Ulomek(2, 3)
```

(Videli smo nekaj posebnih metod, seznam preostalih pa si lahko pogledaš npr. tukaj.)

9. podnaloga

Izven razreda `Ulomek` definirajte funkcijo `priblizek(n)`, ki vrne vsoto `Ulomek`. Zgled:

```
>>> priblizek(5)
Ulomek(163, 60)
```

Ali je izračunana vrednost blizu števila

`Pes` s konstruktorjem `__init__(self, ime_p, starost_p, visina_p)`. Opremite ga z lastnostmi `ime`, `starost` in `visina` in ga popravite tako, da bosta starost in višina vedno pozitivni števili: če uporabnik vnese število manjše od 0, sprožite napako vrste `ValueError`.

Pozor: Enaka napaka naj se zgodi, če poskusimo ustvariti psa z nesmiselnimi podatki.

2. podnaloga

Razred Pes dopolnite tako, da bodo v zavetišču lažje našli potencialne podvojene vnose v bazi. Sestavite metodo `__eq__(self, other)`, ki bo vrnila `True`, če imata psa enaki imeni, starosti in višini, sicer pa `False`. Ta metoda se izvede, če uporabimo operator primerjanja (glej zgled).

Podvojen vnos:

```
>>> p1 = Pes('Rex', 5, 60)
>>> p2 = Pes('Rex', 5, 60)
>>> p1 == p2
True
```

Dva različna vnosa:

```
>>> p1 = Pes('Rex', 5, 60)
>>> p2 = Pes('Fifi', 4, 60)
>>> p1 == p2
False
```

Ponavljanje II

Nadomestni upor (s testi)

Kot je dobro znano iz elektrotehnike, nadomestni upor dveh zaporedno vezanih upornikov izračunamo po formuli
`nadomestni_upor_test`, ki ti bo pomagala pri reševanju naslednje podnaloge. Vsak posamezni slovar naj bo zastavljen tako kot kaže naslednji primer:

```
{"R1": 0, "R2": 0, "vezava": 'Z', "rezultat": 0}
```

Razmisli, katere robne primere potrebuješ (pri tem si pomagaj z opisom naslednje podnaloge).

2. podnaloga

Napišite funkcijo `nadomestni_upor(r1, r2, tip_vezave)`, ki kot argumenta dobi dve števili `r1` in `r2` (tj. upor obeh upornikov) in tip vezave `tip_vezave`, ki je bodisi znak `'V'` bodisi znak `'Z'`. Funkcija naj vrne nadomestni upor vezja. Ta naj bo zaokrožen na tri decimalke. Pozor: nekateri uporniki so vezani kratkostično in imajo upor 0. (Upor dveh vzporedno vezanih upornikov, od katerih ima vsaj en upor 0, je 0.)

```
>>> nadomestni_upor(3, 5, 'Z')
8
>>> nadomestni_upor(3, 0, 'V')
0
>>> nadomestni_upor(3, 5, 'V')
1.875
```

Pri tej nalogi ti Tomo ne bo povedal kaj je narobe - pomagaj si s testi iz prve naloge.

3. podnaloga

Sestavite funkcijo `upor_vezja(niz)`, ki izračuna nadomestni upor vezja. Vezje je podano kot niz v "RPN notaciji": operandoma (tj. vrednosti dveh uporov) sledi operator (tj. način vezave, `'V'` ali `'Z'`).

Analizirajmo vezje `'3 4 1 Z Z 0 V 3 2 Z V'`: Niz `'4 1 Z'` tako pomeni, da sta zaporedno vezana upornika z upornostma 4 in 1 (ki ju lahko nadomestimo z enim upornikom z upornostjo 5). Torej je enako, če bi imeli `'3 5 Z 0 V 3 2 Z V'`. Upora 3 in 5 sta spet zaporedno vezana, torej dobimo vezje `'8 0 V 3 2 Z V'`. V tem novem vezju niz `'8 0 V'` pomeni, da sta vzporedno vezana upornika z upornostima 8 in 0. Vzporedna vezava, ki vsebuje kratkostični upornik, je kratkostična, zato je 0 nadomestni upor vezja `'8 0 V'`. Skratka, dobimo vezje `'0 3 2 Z V'`. Niz `'3 2 Z'` pomeni, da sta zaporedno vezana upornika z upornostima 3 in 2 (in torej z nadomestno upornostjo 5). Če upoštevamo še to, potem dobimo vezje `'0 5 V'`, kar na koncu da rezultat 0.

```
>>> upor_vezja('3 5 Z 0 V 3 2 Z V')
0
```

Predpostavite, da v nizu `niz` nastopajo (poleg znakov `'V'`, `'Z'` in `' '`) le nenegativna cela števila. Rezultat zaokroži na tri decimalke.

4. podnaloga

Napišite še funkcijo `sestavi_racun(niz)`, ki namesto da izračuna nadomestni upor vezja sestavi račun, ki ga je potrebno izračunati, da dobimo nadomestni upor vezja.

```
>>> sestavi_racun('3 5 V 0 Z 3 2 V Z')
'(((1/3 + 1/5)^-1 + 0) + (1/3 + 1/2)^-1)'


>>> sestavi_racun('3 5 Z 0 V')
'0'
```

Namig: Ali lahko nalogo rešite tako, da malenkost predelate rešitev prejšnje naloge?

5. podnaloga

Stari električarski mački si račun poenostavijo na sledeč način: če je upor več kot 10-krat večji kot upor in vzamejo kar
`stari_macki(niz)`, ki bo nadomestni upor vezja izračunala po "metodi starih mačkov". Rezultat zaokrožite na tri decimalke.

```
>>> stari_macki('2 30 Z 20 V 3 2 Z V')
3.529
```

Polinomi

Polinome predstavimo s tabelo koeficientov, pri čemer element z indeksom predstavlja koeficient ob (med drugim torej prazna tabela predstavlja ničelni polinom).

1. podnaloga

Napišite funkcijo `odstrani_odvecne_nicle(koeficienti)`, ki vrne tabelo `koeficienti`, v katerem na koncu odstrani odvečne ničle. Namreč polinom `2x + 3` bi lahko prestavili kot `[3, 2]` ali pa `[3, 2, 0]` (torej `0 x^2 + 2x + 3`) ali pa `[3, 2, 0, 0, 0, 0]`.

```
>>> odstrani_odvecne_nicle([0, 1, 2])
[0, 1, 2]
>>> odstrani_odvecne_nicle([2, 1, 0])
[2, 1]
>>> odstrani_odvecne_nicle([0, 0, 0, 0, 0, 0])
[]
```

2. podnaloga

Napišite funkcijo `odvod_polinoma(polinom, n)`, ki za dani polinom vrne njegov `n`-ti odvod. Privzeta vrednost za `n` naj bo 1.

```
>>> odvod_polinoma([1, 2, 3])
[2, 6],
>>> odvod_polinoma([4, -1, 2, 0, 1], n=2)
[4, 0, 12],
```

```
>>> odvod_polinoma([1])
[]
```

3. podnaloga

Napišite funkcijo `vsota_polinomov(polinom1, polinom2)`, ki vrne vsoto danih polinomov.

```
>>> vsota_polinomov([1, 2, 3], [5, 6, 7, 8])
[6, 8, 10, 8]
>>> vsota_polinomov([1, 2, 3], [-1, -2, -3])
[]
```

Kolesarska dirka

Gibanje kolesarja na dirki podamo s tabelo časov (v minutah), ki povedo, koliko časa je kolesar porabil za posamezno etapo. Dolžina tabele pove, koliko etap je kolesar prevozil.

1. podnaloga

Napiši funkcijo `statistika(casi)`, ki vrne par števil: število etap, ki jih je kolesar prevozil, in skupen čas. Primer:

```
>>> statistika([14, 9, 23])
(3, 46)
```

2. podnaloga

V spremenljivki `dirka` je zapisana tabela tabel etapnih časov kolesarjev. Napiši funkcijo `zmagovalec(dirka)`, ki vrne par: število etap, in indeks zmagovalca. Pozor: nekateri kolesarji pred zaključkom dirke odstopijo, zato ne prevozijo vseh etap.

```
>>> dirka([[2, 1], [1, 2, 3], [6, 3, 1]])
(3, 1)
```

Prepdostavite lahko, da je dirko uspešno zaključil vsaj en tekmovalec ter da na koncu rezultat ni bil izenačen.

3. podnaloga

Dani sta dve (enako dolgi) tabeli časov po etapah. Funkcija `primerjaj(prvi, drugi)` naj vrne trojico števil `(hitrejsi_prvi, enako_hitra, hitrejsi_drugi)`, kjer

- število `hitrejsi_prvi` pove, v koliko etapah je bil hitrejši prvi tekmovalec,
- število `enako_hitra` pove, v koliko etapah sta bila oba tekmovalca enako hitra,
- število `hitrejsi_drugi` pove, v koliko etapah je bil hitrejši drugi tekmovalec.

Na primer:

```
>>> primerjaj([3, 7, 4, 3], [5, 6, 3, 2])
(1, 0, 3)
```

4. podnaloga

Sestavite funkcijo `idealni_cas(dirka)`, ki za tabelo tabel časov `dirka` pove, koliko časa bi za celotno dirko potreboval idealni kolesar, torej tak, ki bi posamezno etapo prevozil tako hitro kot zmagovalec te etape.

```
>>> idealni_cas([[2, 1], [1, 2, 3], [6, 3, 1]])
3
```

Box blur

Box blur je eden od filtrov, katerih učinek je zameglitev slike. Takšna orodja najdemo v programih za obdelavo slik, kot so npr. Photoshop, GIMP in Inkscape.

V našem primeru bomo imeli opraviti le s sivinskimi slikami. Slika bo podana kot matrika. Vsak element matrike bo število med `0.0` (črna) in `1.0` (bela). To število predstavlja barvo pripadajočega piksla. Primer take sivinske slikice je naslednja matrika:

```
slika = [
    [1.00, 0.72, 0.56, 0.45],
    [0.92, 0.64, 0.48, 0.32],
    [0.80, 0.57, 0.42, 0.25],
    [0.73, 0.49, 0.35, 0.21]
]
```

1. podnaloga

Napišite funkcijo `povprecna(mat, r, c)`, ki kot argument dobi matriko `mat`, kot je opisana zgoraj ter celi števili `r` in `c`. Funkcija naj izračuna in vrne povprečno vrednost tistih elementov matrike, katerih številka vrstice se kvečjemu za 1 razlikuje od `r`, številka stolpca pa se kvečjemu za 1 razlikuje od `c`. Zgled (`slika` je matrika, kot je definirana zgoraj):

```
>>> povprecna(slika, 1, 2)
0.49
>>> povprecna(slika, 3, 0)
0.6475
```

2. podnaloga

Napišite funkcijo `box_blur(mat)`, ki sestavi in vrne novo sliko, ki jo dobi tako, da na sliki `mat` uporabi učinek *box blur*. Vrne naj torej enako veliko matriko, katere element v `povprecna(mat, i, j)`. Zgled (matrika `slika` naj bo enaka kot zgoraj):

```
>>> box_blur(slika)
[[0.82, 0.72, 0.5283333333333333, 0.4525],
 [0.775, 0.6788888888888889, 0.49, 0.41333333333333333],
 [0.6916666666666668, 0.6, 0.41444444444444445, 0.3383333333333333],
 [0.6475, 0.56, 0.38166666666666665, 0.3075]]
```

3. podnaloga

Napišite še funkcijo `box_blur_2(mat)`, ki naj deluje podobno kot funkcija `box_blur`, le da ničesar ne vrača, pač pa spremeni matriko `mat`, ki naj na koncu hrani rezultat uporabe filtra box blur.

Zgled (`slika` naj bo enaka kot zgoraj):

```
>>> box_blur_2(slika)
>>> slika
[[0.82, 0.72, 0.5283333333333333, 0.4525],
 [0.775, 0.6788888888888889, 0.49, 0.41333333333333333],
 [0.6916666666666668, 0.6, 0.41444444444444445, 0.3383333333333333],
 [0.6475, 0.56, 0.38166666666666665, 0.3075]]
```

Namig: Kaj že naredi `tabela1[:] = tabela2` ali `tabela1 = tabela2[:]`?

Stagnacija

1. podnaloga

Jure se je odločil, da se bo udeležil kolesarskega maratona, zato je pričel z dvomesečnim treningom. Treniral je vsak dan natanko eno uro in si dnevno število prevoženih kilometrov zapisoval v enodimenzionalno tabelo realnih števil. Pri tem je Jure opazil, da se njegova kondicija v glavnem izboljšuje, saj lahko skoraj vsak dan prevozi nekoliko več kilometrov. Kljub vsem prizadevanjem pa trening ni vedno pokazal želenih rezultatov, saj zaradi utrujenosti mišic Jure ni uspel vedno povečati prevožene razdalje. Pomagaj Juretu napisati funkcijo `stagnacija`, s pomočjo katere bo lahko analiziral svoje nazadovanje (stagnacijo) v svojem treningu. Funkcija `stagnacija` naj vrne število dni stagnacije.

Primeri:

```
stagnacija([26.5,27.1,27.3,27.9,26.8,26.5,27.2])
stagnacija([27.5,27.9,27.2,27.6,28.3,27.7,27.7,28.4,28.5])
```

Prva funkcija vrne vrednost `2`, ker je prišlo `5.` in `6.` dan do nazadovanja. Druga funkcija vrne vrednost `3`, ker je prišlo`3.`, `6.` in `7.` dan do nazadovanja.

2. podnaloga

Miha sledi Juretovemu postopku. Pri pogovorih ob pivu pravi, da njemu vedno uspe, da nikoli ne nazaduje, torej da vedno naslednji dan prevoz vsaj toliko kot prejšnji dan. Jure mu ne verjame in zato se dogovorita, da bo Miha naslednji dan prinesel zapiske svojega treninga.

Miha je seveda s pomočjo tvojega programa iz prejšnje naloge ugotovil, da se je prejšnji dan preveč bahal in da dejansko tudi on določeno dni nazaduje. Zato se odloči, da bo malo "popravil" svoj dnevnik. Pomagaj mu, in sestavi funkcijo `popravki(razdalje)`, ki na podlagi tabele razdalj vrne novo tabelo (z istimi podatki), a v takem vrstnem redu, da bi funkcija `stagnacija` na tej popravljeni tabeli vrnila 0!

Primeri:

```
>>>popravki([26.5,27.1,27.3,27.9,26.8,26.5,27.2])
[26.5, 26.5, 26.8, 27.1, 27.2, 27.3, 27.9]
>>>popravki([27.5,27.9,27.2,27.6,28.3,27.7,27.7,28.4,28.5])
```

```
[27.2, 27.5, 27.6, 27.7, 27.7, 27.9, 28.3, 28.4, 28.5]
```

Rezine in rekurzija

1. podnaloga

Sestavite funkcijo `filtriraj`, ki sprejme dva niza in vrne nov niz sestavljen zgolj iz znakov prvega niza, ki so hkrati tudi v drugem nizu, preostale znake pa zamenja z _ Velikost črk je nepomembna.

```
>>> filtriraj("Ne gremo še domov", "ngm")
"N__g__m_______m__"
```

2. podnaloga

Sestavite funkcijo `pretvori`, ki sprejme niz in bazo ter vrne podano število v desetiškem zapisu. Ko zmanjka števil si znaki sledijo po angleški abecedi `0123456789ABC...`. Primer vrstnega reda lahko najdete v `string.ascii_uppercase`. Lahko predpostavite, da bo baza vedno med 2 in 36.

```
>>> pretvori("10001", 2)
17
>>> pretvori("2ACBD04", 36)
4978911892
```

3. podnaloga

Sestavite funkcijo `izbrisi_podvojene`, ki sprejme niz in odstrani vse zaporedno enake znake, kjer velikost črk ni pomembna. Če se po izbrisu pojavijo nove podvojitve, naj jih funkcija ne izbriše.

```
>>> izbrisi_podvojene("aaab")
"b"
>>> izbrisi_podvojene("abaab")
"abb"
```

4. podnaloga

Sestavite funkcijo `vsak_k_ti`, ki sprejme niz in parameter `k` ter vrne nov niz, kjer iz vhodnega niza vzame vsak `k`-ti znak. Za nesmiselne parametre naj funkcija vrne prazen niz

```
>>> vsak_k_ti("abcdefghijk", 3)
"adgj"
>>> vsak_k_ti("abcdefghijk", 0)
""
```

5. podnaloga

Sestavitev funkcijo `zaporedje`, ki sprejme niz in vrne nov niz sestavljen iz znakov na indeksih 0, 1, 3, 6, 10, ... Namig: Ali razlike med indeksi sledijo kakemu preprostemu zaporedju?

```
>>> zaporedje("0123456789X")
```

```
"0136X"
```

Ugibanje

Pri teh nalogah ti bo prav prišla rekurzija. Seveda se da stvari rešiti tudi brez nje, a uradne rešitve jo uporabljajo!

1. podnaloga

Sestavite funkcijo `preberi_celo_stevilo`, ki s standardnega vhoda prebere celo število. Funkcija naj od uporabnika zahteva vnos, dokler le-ta ne vnese veljavnega celega števila, in vrne prvo veljavno celo število.

```
>>> preberi_celo_stevilo()
> Vnesi celo število: sto
Žal "sto" ni celo število, poskusi ponovno!
> Vnesi celo število: trinajst
Žal "trinajst" ni celo število, poskusi ponovno!
> Vnesi celo število: 10
10
```

2. podnaloga

S pomočjo prejšnje funkcije sestavi funkcijo `ugibaj`, ki sprejme pravilni odgovor in uporabnika sprašuje po številki, dokler le-ta ne ugane pravilne. Funkcija uporabnika obvesti, ali je njegov odgovor pravilen, ali pa je ciljno število večje/manjše. Funkcija naj ne vrača ničesar (`None`).

```
>>> ugibaj(20)
> Vnesi celo število: a
Žal "a" ni število, poskusi ponovno!
> Vnesi celo število: b
Žal "b" ni število, poskusi ponovno!
> Vnesi celo število: 14
Moje število je večje!
> Vnesi celo število: 30
Moje število je manjše!
> Vnesi celo število: x
Žal "x" ni število, poskusi ponovno!
> Vnesi celo število: 20
BRAVO! Res sem si zamislil število 20!
```

3. podnaloga

Sestavite funkcijo `racunalnik_ugiba(spodnji, zgornji)`, ki z metodo bisekcije ugiba število med `spodnji` in `zgornji`, ki si ga je uporabnik zamislil. To pomeni, da računalnik ugiba število, ki je

na sredini med `spodnji` in `zgornji`, nato pa glede na odgovor uporabnika primerno premakne interval ugibanja. Če je veljavno le še eno število, tj., zgornja meja je enaka spodnji, naj računalnik tako proslavi zmago. Funkcija naj ne vrača ničesar (`None`).

```
>>> racunalnik_ugiba(1,10)
Ali je tvoje število Enako/Večje/Manjše od 5?
E/V/M> 42
Ali lahko daš normalen odgovor?
Ali je tvoje število Enako/Večje/Manjše od 5?
E/V/M> banana
Ali lahko daš normalen odgovor?
Ali je tvoje število Enako/Večje/Manjše od 5?
E/V/M> M
Ali je tvoje število Enako/Večje/Manjše od 2?
E/V/M> V
Ali je tvoje število Enako/Večje/Manjše od 3?
E/V/M> V
Juhu, uganil sem! Zamislil si si število 4!


>>> racunalnik_ugiba(1,20)
Ali je tvoje število Enako/Večje/Manjše od 10?
E/V/M> v
Ali je tvoje število Enako/Večje/Manjše od 15?
E/V/M> m
Ali je tvoje število Enako/Večje/Manjše od 12?
E/V/M> e
Juhu, uganil sem! Zamislil si si število 12!
```

Ploščina pod valom

Z metodo Monte Carlo lahko računamo tudi ploščine. Pri tem gre v grobem za to, da naključno izbiramo točke na nekem pravokotniku in štejemo, koliko točk je takih, da "spadajo" k ploščini. Če razmerje med "zadetki" in vsemi točkami pomnožimo s ploščino pravokotnika, dobimo približek za ploščino območja.

1. podnaloga

Ploščino pod enim valom funkcije sinus (enaka je 2) lahko približno izračunamo tudi tako, da naključno izbiramo točke na pravokotniku [0, Pi] x [0, 1] in s Pi pomnožimo razmerje med točkami pod valom in vsemi točkami.

Sestavi funkcijo `ploscina_val(n)`, ki izračuna ploščino vala funkcije `sin(x)` po opisani metodi. Število naključnih točk funkcija dobi kot parameter.

Izpeljane strukture, lambde, ...

Uvod v izpeljane sezname

V tej nalogi si bomo ogledali osnove izpeljanih seznamov. Naloge bodo zato enake ali vsaj precej podobne nalogam, ki smo jih v preteklosti že reševali, a jih tokrat rešite z uporabo izpeljanih seznamov.

Včasih želimo ustvariti seznam (tabelo), ki ga lahko zapišemo s preprosto for zanko, na primer tabelo kvadratov celih števil med 1 in vključno 10.

Do sedaj smo to zapisali takole:

```
tabela = [] for i in range(1, 11): tabela.append(i**2)
```

Znamo to napisati krajše?

Izpeljan seznam sestavimo podobno kot navaden seznam (`[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]`), le da tokrat namesto konkretnih vrednosti v seznam vpišemo kar izraz, s katerim te vrednosti izračunamo.

```
tabela = [i**2 for i in range(1, 11)]
```

Kaj pa kompleksnejši izrazi? Lahko dodamo le kvadrate, ki so sodi?

V izpeljan seznam lahko dodamo tudi pogoj, pod katerim bomo to vrednost dodali v seznam. Na primer:

```
tabela = [i**2 for i in range(1, 11) if i**2 % 2 == 0]
```

1. podnaloga

Napiši funkcijo `veckratniki(k, n)`, ki vrne seznam prvih `n` večkratnikov števila `k`, kjer je

```
 veckratniki(2, 5)

[2, 4, 6, 8, 10]
```

2. podnaloga

Napiši funkcijo `kvadrati(stevila)`, ki ustvari nov seznam, ki vsebuje kvadrate vrednosti iz seznama `stevila`.

Funkcija naj seznam ustvari s pomočjo izpeljanega seznama - telo funkcije naj bo torej zapisano v eni vrstici. V telo funkcije se ne štejejo vrstični komentarji in dokumentacijski niz.

For example:

```
>>> kvadrati([2, 1, 0, -1, -2, -1, 0, 1, 2])

[4, 1, 0, 1, 4, 1, 0, 1, 4]

>>> kvadrati([])

[]
```

Namig: Namesto, da gremo s for zanko čez range (`for i in range(...)`), gremo kar čez podani seznam `stevila`.

3. podnaloga

Napiši funkcijo `odstrani_negativne(stevila)`, ki ustvari nov seznam, ki vsebuje le nenegativne vrednosti iz seznama `stevila`.

Funkcija naj seznam ustvari s pomočjo izpeljanega seznama - telo funkcije naj bo torej zapisano v eni vrstici. V telo funkcije se ne štejejo vrstični komentarji in dokumentacijski niz.

For example:

```
>>> odstrani_negativne([2, 1, 0, -1, -2, -1, 0, 1, 2])
[2, 1, 0, 0, 1, 2]
>>> odstrani_negativne([1, 2, 3, 4, 5, 6])
[1, 2, 3, 4, 5, 6]
>>> odstrani_negativne([-1, -2, -3, -4, -5, -6])
[]
>>> odstrani_negativne([])
[]
```

4. podnaloga

Napiši funkcijo `povecaj_crke(nizi)`, ki ustvari nov seznam, ki vsebuje *neprazne* nize iz seznama `nizi`, le da naj bodo ti spremenjeni v *velike tiskane črke.

Funkcija naj seznam ustvari s pomočjo izpeljanega seznama - telo funkcije naj bo torej zapisano v eni vrstici. V telo funkcije se ne štejejo vrstični komentarji in dokumentacijski niz.

For example:

```
>>> povecaj_crke(["Kokos", "kraVa", "MAČKA", "kričač"])
["KOKOS", "KRAVA", "MAČKA", "KRIČAČ"]
>>> povecaj_crke(["", "   x", "", "", " "])
["   X", " "]
```

Pretvorba v izpeljani seznam

V tej nalogi bo že podana koda, zapisana s klasično for zanko. For zanko nadomestite z izpeljanim seznamom. Poleg pravilnosti testi preverjajo število vrstic v telesu funkcije. Telo funkcije naj bo zapisano v eni vrstici (v to kvoto ne sodijo prazne vrstice ter vrstice s komentarji in dokumentacijskimi nizi).

1. podnaloga

Ugotovi kaj dela funkcija in jo pretvori v izpeljani seznam.

```
import random
def funkcija1(n, a, b):
    seznam = []
    for _ in range(n):
        seznam.append(random.randint(a, b))
```

```
        return seznam
```

2. podnaloga

Ugotovi kaj dela funkcija in jo pretvori v izpeljani seznam.

```
    def funkcija2(seznam, k):
        nov_seznam = []
        for el in seznam:
            if el % k == 0:
                nov_seznam.append(el)
        return nov_seznam
```

Ostale izpeljane strukture

Podobno kot izpeljane sezname lahko tvorimo tudi slovarje in množice.

Množico kvadratov naravnih števil od 1 do 5, na primer, ustvarimo tako:

```
  {x**2 for x in range(1, 6)},     # to ustvari množico {1, 4, 9, 16, 25}
```

slovar, ki ima za ključ števke od 1 do 6, vrednosti pa so ključi deljeni z dva, če je ključ sod, sicer pa ključ pomnožen s 3 in povečan za 1 pa takole:

```
  {x: x//2 if x%2==0 else 3*x+1 for x in range(1, 6)}
  # dobljeni slovar je {1: 4, 2: 1, 3: 10, 4: 2, 5: 16, 6: 3}
```

1. podnaloga

Napiši funkcijo `potence2(n)`, ki vrne množico potenc 2:

```
 potence2(5)
{1, 2, 4, 8, 16, 32}
```

2. podnaloga

Napiši funkcijo `mnozica_sodih(stevila)`, ki vrne množico sodih elementov seznama `stevila`.

Funkcija naj množico ustvari s pomočjo izpeljane množice - telo funkcije naj bo torej zapisano v eni vrstici. V telo funkcije se ne štejejo vrstični komentarji in dokumentacijski niz.

Na primer:

```
>>> mnozica_sodih([1, 4, 5, 2, 7, -8, 5, 0, 101, 2, 1])
{4, 2, -8, 0}
```

3. podnaloga

Napiši funkcijo `ustvari_slovar(seznam_parov)`, ki iz seznama parov oblike `[(k_i, v_i)]` ustvari slovar, ki ima za ključe

```
 ustvari_slovar([(1, "ena"), (2, "dve"), (3, "tri"), (11, "enajst")])
{1: "ena", 2: "dve", 3: "tri", 11: "enajst"}
```

4. podnaloga

Napiši funkcijo `obrni_slovar(slovar)`, ki iz podanega slovarja ustvari nov seznam, ki ima za ključe vrednosti starega slovarja, za vrednosti pa ključe starega slovarja. Predpostaviš lahko, da se posamezna vrednost pojavi pri natanko enem ključu.

Funkcija naj množico ustvari s pomočjo izpeljane množice - telo funkcije naj bo torej zapisano v eni vrstici. V telo funkcije se ne štejejo vrstični komentarji in dokumentacijski niz.

Na primer:

```
>>> obrni_slovar({1: 3, 2: 1, 3: 4, 4: 2})
{3: 1, 1: 2, 4: 3, 2: 4}
```

5. podnaloga

Napiši funkcijo `pomnozi(slovar, k)`, ki ustvari nov slovar, kjer so vrednosti slovarja pomnožene s `k`. Vključeni naj bodo le ključi, pri katerih je vrednost večja od 0.

Funkcija naj množico ustvari s pomočjo izpeljane množice - telo funkcije naj bo torej zapisano v eni vrstici. V telo funkcije se ne štejejo vrstični komentarji in dokumentacijski niz.

Na primer:

```
>>> pomnozi({'jajca': 4, 'moka': 500, 'mleko': 0}, 2)
{'jajca': 8, 'moka': 1000}
```

Lambda funkcije

Včasih pri reševanju programerskih problemov potrebujemo preproste funkcije, ki jih lahko zapišemo z enim izrazom in jih želimo uporabiti le na enem mestu. Takrat ne želimo definirati nove funkcije s ključno besedo def, ampak namesto nje ustvarimo kar anonimno (lambda) funkcijo na mestu, kjer jo potrebujemo.

Tovrstne funkcije so zelo uporabne na primer kot ključ za sortiranje, iskanje najmanjšega/največjega elementa v tabeli, podajanje funkcije kot argument drugi funkciji...

Oglejmo si nekaj preprostejših primerov:

- Lambda funkcija, ki argument `x` poveča za 1: `lambda x: x + 1`
- Lambda funkcija, ki se šteje argumenta `x` in `y`: `lambda x, y: x + y`
- Lambda funkcija, ki vrne "večji", če je `x` strogo večji od `y`, sicer pa "manjši":

  `lambda x, y: "večji" if x < y else "manjši"`

1. podnaloga

Napiši funkcijo `najkrajsa(besede)`, ki vrne najkrajšo besedo v seznamu `besede`.

Primer:

```
>>> najkrajsa(["banana", "krompir", "kisla repa", "kis"])
"kis"
```

Telo funkcije naj bo dolgo le eno vrstico (torej oblike `return izraz_ki_resi_problem`).

Namig: Uporabi funkcijo `min()`, ki sprejme seznam in opcijski argument `key`. Kot ključ uporabite lambda funkcijo, ki bo določila vrednost posameznega elementa.

```
min(seznam, key=lambda x: nek_izraz)
```

V lambda funkciji zapišemo izraz, na podlagi katerega se bo izračunala vrednost posameznega elementa seznama. To vrednost bo funkcija `min` uporabila kot ključ na podlagi katerega izbere najmanjši element.

2. podnaloga

Napiši funkcijo `uredi_po_uspesnosti(ekipe)`, ki podani seznam parov imen košarkarskih ekip in njihov odstotek zmag uredi od najuspešnejše do najmanj uspešne (torej padajoče). Funkcija naj vrne nov seznam, seznama `ekipe` naj ne spreminja.

Primer:

```
>>> uredi_po_uspesnosti([("Jazz", 0.598), ("Grizzlies", 0.683), ("Mavericks", 0.634), ("Nuggets", 0.585), ("Warriors", 0.646), ("Suns", 0.780)])

[("Suns", 0.780), ("Grizzlies", 0.683), ("Warriors", 0.646), ("Mavericks", 0.634), ("Jazz", 0.598), ("Nuggets", 0.585)]
```

Telo funkcije naj bo dolgo le eno vrstico (torej oblike `return izraz_ki_resi_problem`).

Namig: Uporabi funkcijo `sorted()`, ki sprejme seznam in opcijska argumenta `key` in `reverse`.

3. podnaloga

Napiši funkcijo `najbolj_raznolika(besede)`, ki vrne najbolj raznoliko besedo v seznamu `besede`. Najbolj raznolika beseda je tista, ki ima največ različnih soglasnikov.

Primer:

```
>>> najbolj_raznolika(["banana", "krompir", "kisla repa", "kis"])

"kisla repa"
```

Telo funkcije naj bo dolgo le eno vrstico (torej oblike `return izraz_ki_resi_problem`).

Namig: Pomagaj si z množicami.

Matrike

Matriko v Pythonu predstavimo s seznami seznamov, pri čemer predpostavimo, da ima matrika vsaj en element in da imajo vsi podseznami enako dolžino.

Naloge reši z uporabo izpeljanih seznamov.

Enako kot zanke lahko gnezdimo tudi izpeljane sezname. Tako bi na primer zanko, ki ustvari seznam seznamov enic z `n` vrsticami in `m` stolpci

```
enice = []
for i in range(n):
    vrstica = []
    for j in range(m):
        vrstica.append(1)
```

```
        enice.append(vrstica)
```

prevedli v izraz:

```
    enice = [[1 for j in range(m)] for i in range(n)]
```

1. podnaloga

Sestavite funkcijo `diagonala(matrika)`, ki vrne seznam elementov na diagonali podane matrike.

```
>>> diagonala([[1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 1, 0], [0, 0, 0, 1]])
[1, 1, 1, 1]
```

2. podnaloga

Sestavite funkcijo `vsota_elementov(matrika)`, ki vrne vsoto vseh elementov matrike.

```
>>> vsota_elementov([[1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 1, 0], [0, 0, 0, 1]])
4
```

3. podnaloga

Sestavite funkcijo `identiteta(n)`, ki vrne identično matriko dimenzij `n` × `n`.

```
>>> identiteta(3)
[[1, 0, 0], [0, 1, 0], [0, 0, 1]]
```

4. podnaloga

Sestavite funkcijo `transponiraj`, ki sestavi in vrne novo matriko in sicer transponiranko dane matrike dimenzij `n` × `m`.

```
>>> transponiraj([[1, 2], [3, 4]])
[[1, 3], [2, 4]]
```

5. podnaloga

Sestavite funkcijo `sestej`, ki sprejme dve matriki in sestavi ter vrne novo matriko, ki je vsota podanih matrik.

```
>>> sestej([[1, 0], [0, 1]], [[0, 2], [0, 0]])
[[1, 2], [0, 1]]
```

6. podnaloga

Sestavite funkcijo `splosci(matrika)`, ki matriko po vrsticah splošči v vektor.

```
>>> splosci([[1, 0], [0, 1]])
[1, 0, 0, 1]
```

Datoteke

Naslednje naloge so podobne nalogam iz prejšnjih sklopov. Tokrat naloge rešite z uporabo novo pridoboljenih znanj - izpeljanih struktur in lambda funkcij.

1. podnaloga

Sestavite funkcijo `stevilo_dolgih_vrstic(ime_datoteke, dolz)`, ki vrne število vseh vrstic v datoteki z danim imenom, ki so dolge vsaj `dolz` znakov (pri tem je `dolz`-ti lahko tudi znak `"\n"` za novo vrstico).

2. podnaloga

Sestavite funkcijo `najdaljsa_vrstica(ime_datoteke)`, ki vrne najdaljšo vrstico v datoteki z danim imenom. Če je takih vrstic več, vrni prvo!

Namig, uporabi funkcijo `max`.

OOP III - dedovanje

Naloge v tem sklopu služijo spoznavanju z dedovanjem v objektno usmerjenem programiranju.

Uvod v dedovanje

Za spoznavanje z dedovanjem si oglejmo preprost primer.

1. podnaloga

Napiši razred `Zival`, ki bo žival predstavil z imenom, višino in težo. Razred naj vsebuje konstruktor, ki nastavi vrednosti lastnosti (property) `ime`, `visina`, `teza`. Pri lastnostih naredi smiselne kontrole.

Razredu dodaj metodo `predstavi_se`, ki *vrne* predstavitveni niz živali kot kaže primer:

```
>>> zival = Zival("Nana", 10, 0.5)
>>> print(zival.predstavi_se())
Sem Nana, merim 10 centimetrov in tehtam 0.5 kilogramov.
```

2. podnaloga

Sedaj ustvari razred `Macka`, ki predstavlja mačko. Ker velja, da je mačka tudi žival in ima vse lastnosti živali - ime, višino in težo, lahko razred `Macka` podeduje lastnosti in metode razreda `Zival`.

Če želimo, da razred `Podrazred` deduje iz razreda `Razred`, ga ustvarimo takole

```
    class Podrazred(Razred):
```

V telo razreda zaenkrat napišimo samo ukaz `pass`.

Sestavi program, ki ustvari novo mačko v spremenljivki `macka` z imenom Tačka, višino 20 cm in težo 5kg. Nato izpiši njeno ime (`macka.ime`). Nato pokliči metodo `predstavi_se()` in izpiši vrnjeni niz.

Kot opaziš, se tudi pri mački nastavijo enake lastnosti kot pri živali, prav tako pa nad mačkami prav tako lahko uporabljamo metode iz razreda živali, čeprav jih nismo izrecno definirali. Kadar metoda v podrazredu ni definirana (v tem primeru `__init__` in `predstavi_se`), Python izvede metodo iz starševskega razreda.

3. podnaloga

Pri prejšnji podnalogi smo ugotovili, kako nam dedovanje pomaga pri tem, da imajo tudi objekti podrazredov dostop do metod nadrazreda. Vendar pa ponavadi podrazredom želimo dodati še kakšno lastnost ali metodo, da razširimo oziroma prilagodimo funkcionalnost.

Definiraj razred `Pes`, ki bo prav tako kot `Macka` dedoval iz razreda `Zival`. Objektom tipa pes bomo tokrat želeli dodati še lastnost `pasma`. Sedaj moramo torej napisati konstruktor `__init__`, ki bo poleg imena, teže in višine prejel še pasmo. Lastnost pasma torej nastavimo v konstruktorju, kot smo tega navajeni že iz preteklih srečanj z OOP. Ker so ostale lastnosti enake starševskemu razredu `Zival`, lahko za njihovo nastavitev poskrbi kar konstruktor za žival.

Konstruktor starševskega razreda, ki sprejme argumente args, pokličemo takole:

```
  def __init__(self, args):
      super().__init__(args)
```

4. podnaloga

Podobno kot smo storili s konstruktorjem, lahko v podrazredu pokličemo tudi druge metode nadrazreda. Napiši metodo `predstavi_se()`, ki bo predstavitev iz starševskega razreda dopolnila s "pasjim" pozdravom in informacijo o pasmi, kot kaže primer:

```
>>> pes = Pes("Reks", 100, 30, "nemški ovčar")
>>> print(pes.predstavi_se())
Hov! Sem Reks, merim 100 centimetrov in tehtam 30 kilogramov. Sem nemški ovčar.
```

5. podnaloga

Včasih pa bomo želeli metode starševskega razreda kar povoziti. Napiši razred `Lisica`, ki bo sprejel enake argumente kot `Zival` - uporabi konstruktor razreda `Zival`.

Definirajte metodo `predstavi_se`, ki vrne naključnega izmed nizov:

```
  nizi = [
       "Ring-ding-ding-ding-dingeringeding!",
       "Wa-pa-pa-pa-pa-pa-pow!",
       "Hatee-hatee-hatee-ho!",
       "Joff-tchoff-tchoffo-tchoffo-tchoff!"]
```

Opomba: za tiste, ki ne veste, kako se oblaša lisica: Ylvis - The Fox :D

Knjižnica

V knjižnici potrebujejo nov informacijski sistem. Pomagajte jim pripraviti ustrezne razrede za ustrezen prikaz podatkov.

1. podnaloga

Za začetek pripravite razred `Gradivo`, ki bo predstavljal knjižnično gradivo. Napišite konstruktor, ki bo nastavil lastnosti `naslov` in `avtor` gradiva, podana kot argument. Dodajte še metodo za izpis, ki bo delovala, kot kaže primer:

```
  >>> gradivo = Gradivo("Naslov gradiva", "Ime Avtorja")
  >>> gradivo.avtor
  Ime Avtorja
  >>> print(gradivo)
  [Gradivo neznanega tipa] Ime Avtorja: Naslov Gradiva
```

2. podnaloga

Ker pa želi knjižnica natančneje definirati tip gradiva, poleg tega pa za različne tipe gradiv hraniti različne lastnosti in jih izpisovati različno, bomo definirali nekaj podrazredov. Kot vsaka dobra knjižnica, je tudi pri nas najpomembnejši tip gradiva knjiga. Definirajte razred `Knjiga`, ki bo podrazred razreda `Gradivo` (torej bo dedoval iz tega razreda, kot smo se naučili pri uvodni nalogi). Poleg lastnosti `naslov` in `avtor` (uporabi naj kar konstruktor razreda `Gradivo`), naj razred Knjiga hrani še podatek o številu strani v lastnosti `st_strani`.

Dodajte še metodo za izpis, ki bo pripravila izpis oblike:

```
[Knjiga] Ime Avtorja: Naslov knjige (100 strani)
```

3. podnaloga

Ker dandanes veliko ljudi raje gleda filme, kot pa bere, se je knjižnica odločila izposojati tudi filme. Podobno razredu `Knjiga` definirajte tudi razred `Film`, le da bo imel ta poleg naslova in avtorja še lastnosti `trajanje` in `zvrst`.

```
Film('Titanic', 'J. Cameron', 'drama', 194)
```

Metoda za izpis naj sestavi takšen izpis:

```
[Film] Ime Avtorja: Naslov filma (komedija; 100 minut)
[Film] J. Cameron: Titanic (drama; 194 minut)
```

4. podnaloga

Sedaj, ko imamo pripravljene različne tipe gradiva, ki jih naša knjižnica ponuja, se lahko lotimo priprave krovnega razreda - razreda `Knjiznica`. Tako lahko delovanje našega informacijskega sistema razširimo na večje število knjižnic.

Razred knjižnica naj ima lastnosti (`@property`) `ime` in `gradiva`, ki naj ju ne bo mogoče spreminjati (če to želimo, naj se sproži napaka `AttributeError`). Ob ustvarjanju nove knjižnice, bomo podali ime, seznam gradiv pa naj bo prazen.

5. podnaloga

Razredu knjižnica dodajte metodo `dodaj_gradivo(self, gradivo)`, ki v seznam gradiv pripne novo gradivo. Če podano gradivo ni podatkovnega tipa Gradivo (oziroma njegovih podtipov), naj metoda sproži napako `TypeError`.

6. podnaloga

Za lažje delo razredu `Knjiznica` dodajmo še metodo `__str__`. Ta naj sestavi takšen izpis: V prvi vrstici naj se nahaja niz `"Knjiznica <ime_knjiznice> ponuja naslednja gradiva:"`. Sledi naj prazna vrstica, nato pa naj sledijo izpisi posameznih gradiv. Za boljšo preglednost naj bodo ločeni z vrstico poljubnega števila znakov "-".

Primer izpisa:

```
Knjiznica RadiBeremo ponuja naslednja gradiva:

[Knjiga] Karl May: Vinetou I (555 strani)
----------------------------
[Knjiga] Douglas Adams: Štoparski vodnik po Galaksiji (169 strani)
----------------------------
[Film] Theodore Melfi: Hidden Figures (biografija; 127 minut)
----------------------------
[Film] Morten Tyldum: The Immitation Game (biografija; 114 minut)
----------------------------
[Gradivo neznanega tipa] Ian Goodfellow: Generative Adversarial Networks
```

7. podnaloga

Sedaj, ko smo ustvarili vse potrebne razrede in metode, je čas, da jih uporabimo. Ustvarite novo knjižnico s poljubnim imenom. Dodajte ji vsaj pet gradiv različnih tipov. Nato knjižnico izpišite (`print`).

Preverite, da so gradiva izpisana pravilno. V poročilu oddajte posnetek zaslona izpisa.

Večkotniki

1. podnaloga

Za začetek napišite razred `Tocka`, ki predstavlja točko na ravnini. Predstavljena naj bo s koordinatama `x` in `y` (naj bosta lastnosti - property in nimata metode setter), ki sta podani konstruktorju.

Razredu dodajte še metodo `__str__`, ki točko predstavi z izpisom oblike `(x, y)` in `__repr__`, ki točko predstavi z izpisom oblike `Tocka(x, y)`

Napišite metodo `razdalja(self, druga)`, ki sprejme drugo točko in izračuna razdaljo med dvema točkama.

Ustvarite točko `tocka` s koordinatama in in stestirajte, da deluje pravilno. Izračunajte njeno razdaljo do koordinatnega izhodišča in jo shranite v spremenljivko `razdalja`.
Primer:

```
>>> t = Tocka(0, 10)
>>> t
(0, 10)
>>> round(t.razdalja(Tocka(10, 10)))
10
```

2. podnaloga

Napišite razred `Veckotnik`, ki bo hranil tabelo točk v lastnosti `tocke`. Lastnost naj ne omogoča nastavljanja. Za lažje delo z večkotniki preverite, če je zadnja točka v seznamu enaka prvi - če ni, jo dodajte.

3. podnaloga

Razredu večkotnik dodajte metodi `obseg` in `ploscina`, ki izračunata obseg in ploščino poljubnega večkotnika (lahko predpostavite, da nima samopresečišč).

Namig: pri računanju ploščine si pomagajte s formulo (shoelace formula)

`Tocka`.

Ustvarite večkotnik, določen s točkami `ob_nkot` shranite njegov obseg, v `pl_nkot` pa njegovo ploščino. Preverite, da je ploščina enaka 8,5, obseg pa približno enak 12,36.

4. podnaloga

Sestavite razred `Trikotnik`, ki naj bo izpeljan iz razreda `Veckotnik` (naj deduje iz njega), Namesto tabele točk, želimo konstruktorju trikotnika podati kar tri oglišča: `__init__(self, a, b, c)`. Konstruktor naj jih sestavi v tabelo in posreduje konstruktorju razreda `Veckotnik`.

Ustvarite nov trikotnik z oglišči in ga shranite v spremenljivko `trikotnik`. Izračunajte obseg in ploščino, ter ju shranite v spremenljivki `ob_3kot` in `pl_3kot`. Preverite, da je obseg približno 9,84, ploščina pa 4.

5. podnaloga

Sedaj napišite še razred `Pravokotnik`, ki naj prav tako deduje iz razreda `Veckotnik`. Pravokotnik želimo definirati s spodnjim levim oglišcem (torej tistim, ki ima najmanjšo in koordinato) ter širino in višino: `__init__(self, ta, sirina, visina)`.
Konstruktor naj izračuna preostale točke in jih poda konstruktorju nadrazreda. Poleg tega naj si zapomni tudi višino in širino.

Ker poznamo višino in širino pravokotnika, lahko ploščino izračunamo učinkoviteje kot za poljubni večkotnik. Dodajte metodo `ploscina`.

Ustvarite pravokotnik s točko `pravokotnik`. Preverite, da je ploščina enaka 6, obseg pa 10. Preverite še, da bi splošen večkotnik, sestavljen iz istih točk vrnil enako ploščino.

6. podnaloga

Ustvarite tabelo vsaj petih večkotnikov različnih tipov. Za vsakega izračunajte ploščino in obseg ter izpišite podatke in izračunani vrednosti. Seveda preverite tudi pravilnost izračunanih vrednosti. Pri tem si lahko pomagate na primer z GeoGebro

Izpis naj izgleda nekako takole:

```
Večkotnik [(-3, 2), (-1, 3), (0, 2), (2, 1), (0, 0), (-2, 0), (-3, 2)]:
    obseg: 12.36
    ploscina: 8.50.
Trikotnik [(2, 4), (3, 6), (6, 4), (2, 4)]:
    obseg: 9.84
    ploscina: 4.00.
Pravokotnik [(0, 1), (0, 3), (3, 3), (3, 1), (0, 1)]:
     obseg: 10.00
     ploščina: 6.00.
```

Izpis se lahko razlikuje od predlaganega, mora pa vsebovati iste informacije. Posnetek zaslona izpisa dodajte v poročilo.

Zaloga nalog

Dodatne naloge (možno, da se kakšna ponovi).

Gensko spremenjena hrana

Upravljalec restavracije te prosi za pomoč. Ljudje so ozaveščeni in skrbi jih gensko spremenjena hrana. Moram mu pomagati, da bo izračunal količino GSOja, ki bo zapisana v novem jedilniku.

1. podnaloga

Kuhar je že sestavil slovar vseh jedi na jedilniku skupaj z njihovimi sestavinami. Sestavine so tudi podane v obliki slovarja z vnosi oblike `sestavina: količina`. Enota pri količini ni podadana (lahko je to teža, volumen, število kosov, …). Ker se je zelo mudilo, je pri jedeh, kjer je cela jed ena sama sestavina, pisal le prazno množico. Sestavite funkcijo `odpravi_okrajsave(recepti)`, ki sestavi in vrne nov slovar, ki ne vsebuje okrajšav. Zgled:

```
>>> recepti = {'pica': {'moka': 80, 'sir': 30}, 'solata': dict()}
>>> odpravi_okrajsave(recepti)
{'solata': {'solata': 1}, 'pica': {'sir': 30, 'moka': 80}}
```

2. podnaloga

V tistih množicah, ki predstavljajo recepte, je kuhar količine sestavin podal v gramih. Da bomo lažje računali GSO, jih je treba "normalizirat", da bo vse v procentih. Napišite funkcijo `normaliziraj_kolicine(recepti)`, ki kot argument dobi slovar receptov in vrne "normaliziran" slovar receptov (tj. vsota vseh količin v vsakem receptu naj bo točno 100). Zgled:

```
>>> recepti = {'pica': {'moka': 80, 'sir': 30} 'solata': dict()}
>>> normaliziraj_kolicine(recepti)
{'solata': {'solata': 100.0},
 'pica': {'sir': 27.272727272727273, 'moka': 72.72727272727273}}
```

3. podnaloga

Upravljalec restavracije vam je priskrbel še seznam vseh elementarnih sestavin skupaj z njihovim deležem GSO. Sestavite funkcijo `delez_gso(recepti, elementarne)`, ki dobi slovar z recepti in slovar z deleži GSO v elementarnih sestavinah in poračuna deleže GSO v jedeh. V slovarju elementarnih sestavin so vsaj vse elementarne sestavine, ki se nahajajo v receptih, lahko pa jih je tudi več. Zgled:

```
>>> recepti = {'pica': {'moka': 80, 'sir': 30}, 'solata': dict()}
>>> elementarne = {'moka': 70, 'sir': 10, 'solata': 0}
>>> delez_gso(recepti, elementarne)
{'solata': 0.0, 'pica': 53.63636363636364}
```

4. podnaloga

Poleg običajnega imajo v restavraciji tudi specialni jedilnik, ki vsebuje eksotične jedi. Te vsebujejo tudi sestavine, ki prihajajo iz "sumljivih" dežel in deleža GSO ni mogoče ugotoviti. Napišite še funkcijo `delez_gso_special(recepti, elementarne)`, ki je podobna kot v prejšni nalogi, le da vsaki jedi

priredi "interval" (tj. par `(min_gso, max_gso)`). V minimalnem primeru predpostavimo povsod 0 % GSO, v maximalnem primeru pa povsod 100 % GSO. Zgled:

```
>>> recepti = {'pica_special': {'moka': 80, 'sir': 30, 'namaz': 10}, 'solata_special':
dict()}

>>> elementarne = {'moka': 70, 'sir': 10, 'solata': 5}

>>> delez_gso_special(recepti, elementarne)

{'pica_special': (49.16666666666667, 57.5), 'solata_special': (0.0, 100.0)}
```

Rimski imperij vrača udarec

Pri tej nalogi boste napisali funkcijo, ki bo za dano število vrnila niz z ustrezno rimsko številko. Nato boste napisali še funkcijo, ki bo rimsko številko spremenila nazaj v število.

Preden začnete, si na Wikipediji oglejte članek o rimskih številkah: Rimske številke.

1. podnaloga

Napišite funkcijo `v_rimsko(stevilo)`, ki kot argument dobi celo število med 1 in 3999 (vključno z 1 in 3999). Funkcija naj sestavi in vrne niz, ki vsebuje rimsko številko, ki predstavlja število `stevilo`. Zgled:

```
>>> v_rimsko(2013)

'MMXIII'
```

2. podnaloga

Napišite še funkcijo `v_arabsko(niz)`, ki bo dobila niz `niz`, ki predstavlja rimsko številko. Funkcija naj naredi ravno obratno kot funkcija `v_rimsko`, tj. vrne naj ustrezno število, ki ga predstavlja dana rimska številka. Če `niz` ne predstavlja veljavne rimske številke, naj funkcija vrne `None`. Zgled:

```
>>> v_arabsko('MMXIII')

2013
```

Nasvet: Najprej sestavi slovar, ki kot ključe vsebuje rimske številke, kot vrednosti pa ustrezna števila. Potem le uporabi ta slovar.

Usmerjeni grafi

Kadar nas zanima, kam se lahko iz danega kraja odpravimo, lahko odpremo zemljevid in pogledamo, v katere kraje nas iz danega kraja vodijo ceste. Recimo iz Celja se lahko odpravimo proti Velenju, Laškem ali Rogaški Slatini, iz Logatca se lahko odpravimo proti Idriji, Postojni in Vrhniki, iz Vrhinike se lahko odpravimo nazaj proti Logatcu ali proti Ljubljani.

Tak zemljevid lahko predstavimo tudi z usmerjenim grafom. Usmerjen graf sestavljata množica vozlišč in množica usmerjenih povezav

```
= {'Logatec', 'Vrhnika', 'Ljubljana', 'Postojna', 'Idrija'}

A = [('Logatec', 'Vrhnika'), ('Logatec', 'Postojna'), ('Logatec', 'Idrija'), ('Vrhnika'
, 'Logatec'), ('Vrhnika', 'Ljubljana')]
```

Lahko pa ga podamo v obliki slovarja naslednikov:

```
{'Logatec': ['Vrhnika', 'Postojna', 'Idrija'], 'Vrhnika': ['Logatec', 'Ljubljana']}
```

(V zgornjem primeru smo predpostavili, da se iz Postojne, Idrije in Ljubljane, ne da priti nikamor (kar seveda ni res))

Ključi v tem slovarju so vozlišča, vrednost pri posameznem ključu `u` pa je seznam vseh vozlišč, ki so nasledniki vozlišča `u`. (Vozlišče je *naslednik* vozlišča
izoliranih vozlišč (to so vozlišča, ki niso niti začetek niti konec katere od povezav).

1. podnaloga

Napišite funkcijo `slovar_naslednikov(seznam_povezav)`, ki kot argument dobi seznam povezav digrafa, sestavi in vrne pa naj pripadajoči slovar naslednikov. Zgled:

```
>>> slovar_naslednikov([('a', 'b'), ('c', 'b'), ('c', 'd'), ('d', 'a'), ('a', 'c')])
{'a': ['b', 'c'], 'c': ['b', 'd'], 'd': ['a']}
```

Seznami naslednikov naj bodo *urejeni*.

2. podnaloga

Sestavite funkcijo `seznam_povezav(digraf)`, ki kot argument dobi usmerjen graf, ki je podan kot slovar naslednikov. Funkcija naj sestavi in vrne seznam usmerjenih povezav. (Torej, funkcija `seznam_povezav` naj naredi ravno obratno kot funkcija `slovar_naslednikov`.) Zgled:

```
>>> seznam_povezav({'a': ['b', 'c'], 'c': ['b', 'd'], 'd': ['a']})
[('a', 'b'), ('a', 'c'), ('c', 'b'), ('c', 'd'), ('d', 'a')]
```

Seznam povezav, ki ga vrne funkcija, naj bo *urejen*.

3. podnaloga

Napišite funkcijo `nasprotni_graf(digraf)`, ki dobi usmerjen graf v obliki slovarja naslednikov, sestavi in vrne pa naj nasprotni graf (tudi v obliki slovarja naslednikov). *Nasprotni graf* grafa ima enako množico vozlišč kot

```
 nasprotni_graf({'a': ['b', 'c'], 'c': ['b', 'd'], 'd': ['a']})
{'a': ['d'], 'c': ['a'], 'b': ['a', 'c'], 'd': ['c']}
```

Seznami naslednikov naj bodo *urejeni*.

4. podnaloga

Usmerjene grafe lahko sestavimo iz večih usmerjenih podgrafov. Vsak posamezen podgraf je predstavljen s slovarjem, za katerega velja, da ima vsako vozlišče le enega naslednika.

Sestavite funkcijo `sestavljen_graf(tabela_podgrafov)`, ki bo iz dane tabele usmerjenih podgrafov sestavila slovar, ki bo predstavljal graf v obliki slovarja naslednjikov. Vrednosti naj bodo v tabelah zapisane v enakem vrstnem redu, kot nastopajo v tabeli podgrafov. Zgled:

```
>>> sestavljen_graf([{'a': 'b', 'c': 'b'}, {'a': 'c', 'c': 'd', 'd': 'a'}])
{'a': ['b', 'c'], 'c': ['b', 'd'], 'd': ['a']}
```

Rodovniki

Ukvarjali se bomo z rodovniki (Celjskih grofov in drugih). Rodovnik imamo podan kot slovar, kjer je ključ ime "glave rodbine" vrednost pa tabela imen otrok. Recimo:

```
rodovnik =
 {'Ulrik I.': ['Viljem'], 'Margareta': [], 'Herman I.': ['Herman II.', 'Hans'],
  'Elizabeta II.': [], 'Viljem': ['Ana Poljska'], 'Elizabeta I.': [],
  'Ana Poljska': [], 'Herman III.': ['Margareta'], 'Ana Ortenburška': [],
  'Barbara': [], 'Herman IV.': [], 'Katarina': [], 'Friderik III.': [],
  'Herman II.': ['Ludvik', 'Friderik II.', 'Herman III.', 'Elizabeta I.', 'Barbara'],
  'Ulrik II.': ['Herman IV.', 'Jurij', 'Elizabeta II.'], 'Hans': [], 'Ludvik': [],
  'Friderik I.': ['Ulrik I.', 'Katarina', 'Herman I.', 'Ana Ortenburška'],
  'Friderik II.': ['Friderik III.', 'Ulrik II.'], 'Jurij': []}
 rodovnik['Friderik II.']
```

nam torej vrne

```
  ['Friderik III.', 'Ulrik II.']
```

1. podnaloga

Število otrok

Sestavi funkcijo `koliko_otrok(ime, rodovnik)`, ki za dano ime in rodovnik vrne število otrok te osebe, oz None, če osebe ni v rodovniku.

2. podnaloga

Število potomcev

Sestavi funkcijo `koliko_potomcev(ime, rodovnik)`, ki za dano `ime` in `rodovnik` vrne število potomcev te osebe. Če osebe ni v rodovniku, vrni None

3. podnaloga

Je v rodbini?

Sestavi funkcijo `je_v_rodbini(ime, glava_rodbine, rodovnik)`, ki ugotovi, ali je oseba z imenom `ime` v rodbini osebe `glava_rodbine`.

4. podnaloga

Kdo se podpisuje najdlje časa?

Sestavi funkcijo `najdaljsi_podpis(glava_rodbine, rodovnik)`, ki ugotovi, kdo v rodbini osebe `glava_rodbine` ima najdaljše ime za podpis (torej kompletno ime).

5. podnaloga

Kdo ima najkrajše ime?

Sestavi funkcijo `najkrajse_ime(glava_rodbine, rodovnik)`, ki ugotovi, kdo v rodbini osebe `glava_rodbine` ima najkrajše ime. (šteje samo krstno ime, brez "Ortenburga" in "Celja" ter brez številk)?

6. podnaloga

Globina rodbine

"Globino" rodbine definiramo tako: če nekdo nima otrok, je globina njegove rodbine 1. Če ima otroka, ta pa nima vnukov (ali celo več otrok, ti pa nimajo vnukov), je globina rodbine 2. Če nekdo ima vnuke, vendar nobenega pravnuka, je globina njegove rodbine 3.

Sestavi funkcijo `globina(glava_rodbine, rodovnik)`, ki vrne globino rodbine osebe `glava_rodbine` v rodovniku `rodovnik`

Premešan vrstni red

1. podnaloga

Spodnji program naj bi pretvarjal starost psa iz človeških v pasja leta. Prvi dve leti so pasja leta enaka `10.5 * človeška leta`, nato pa je vsako nadaljnje pasje leto enako 4 človeškim letom. V programu se je premešal vrstni red vrstic (zamiki so ostali pravilni). Popravi vrstni red vrstic, da bo program smiselno deloval.

```
    if c_starost <= 2:
    print("Starost mora biti pozitivno število.")
c_starost = int(input("Vnesi starost psa v človeških letih: "))
        p_starost = 21 + (c_starost - 2) * 4
if c_starost < 0:
else:
    print("Starost psa v pasjih letih je", p_starost)
        p_starost = c_starost * 10.5
    else:
```

2. podnaloga

Spodnji program naj bi za dani mesec izpisal število dni v tem mesecu. V programu se je premešal vrstni red vrstic (zamiki so ostali pravilni). Popravi vrstni red vrstic, da bo program smiselno deloval.

```
else:
elif mesec in ("januar", "marec", "maj", "julij", "avgust", "oktober", "december"):
if mesec == "februar":
    print("Napačno poimenovanje meseca")
    print("Število dni: 30 dni")
mesec = input("Vnesi ime meseca: ")
elif mesec in ("april", "junij", "september", "november"):
    print("Število dni: 31 dni")
    print("Število dni: 28/29 dni")
```

Popravi kodo II

1. podnaloga

Spodnji program izračuna vsoto in povprečje celih števil, ki jih uporabnik vnese. V kodi manjkajo zamiki. Popravi kodo, da bo smiselno delovala.

```
print("Vnesi nekaj celih števil, da bom lahko izračunal njihovo vsoto in povprečje. Vne
si 0, če si zaključil z vnašanjem števil.")
stevec = 0
vsota = 0.0
stevilo = 1
while stevilo != 0:
stevilo = int(input(""))
vsota = vsota + stevilo
stevec += 1
if stevec == 1:
print("Nisi vnesel števil.")
else:
print("Povprečje in vsota vnešenih števil sta:", vsota / (stevec-1), vsota)
```

2. podnaloga

Spodnji program izbere naravno število med 1 in 10. Uporabnika pozove, da ugane izbrano število. Uporabnik ugiba, dokler ne najde pravega števila. A program ne deluje pravilno, ker so se vrstice med seboj premešale in se naključno zamaknile. Popravi program, da bo pravilno deloval.

```
       elif iskano_stevilo < ugibanje:
ugibanje = int(input("Ugibaj število med 1 in 10, dokler ne najdeš pravega števila: "))
    print("Našel si iskano število!")
   print("Iščeš večje število.")
    while iskano_stevilo != ugibanje:
           if iskano_stevilo > ugibanje:
   print("Iščeš manjše število.")
iskano_stevilo, ugibanje = random.randint(1, 10), 0
        import random
```

3. podnaloga

Spodnja funkcija naj bi računala največji skupni delitelj dveh števil z uporabo Evklidovega algoritma, ne deluje pravilno, ker so vrstice med seboj premešane. Poleg tega so se v funkcijo vrinile nekatere nepotrebne vrstice. Popravi funkcijo, da bo pravilno delovala.

```
b = a % b
a, b = b, a % b
def gcd(a, b):
```

```
    a = b
return a
a, b = b - a, a
while b > 0:
''' izracuna najvecji skupni delitelj (ang. greatest common divisor) stevil a in b '''
```

Hatebook

Za razliko od običajnih družabnih omrežij, deluje nedružabno omrežje Hatebook tako, da si v omrežje vsak dodaja svoje sovražnike. Omrežje predstavimo s slovarjem, pri čemer so ključi osebe, vrednosti pa množice oseb, ki jih te osebe sovražijo.

Primer omrežja:

```
>>> omrezje = {'Žiga': {'Zoran', 'Vesna', 'Ugo'},
               'Zoran': {'Vesna'},
               'Vesna': set(),
               'Ugo': {'Zoran', 'Žiga', 'Ugo'},
               'Tina': {'Vesna'}}
```

1. podnaloga

Sestavite funkcijo `se_sovrazita(omrezje, oseba1, oseba2)`, ki vrne `True`, kadar osebi sovražita druga drugo, in `False` sicer.

Primer:

```
>>> se_sovrazita(omrezje, "Ugo", "Žiga")
True
>>> se_sovrazita(omrezje, "Ugo", "Zoran")
False
```

2. podnaloga

Sestavite funkcijo `kdo_sovrazi(omrezje, oseba)`, ki vrne množico oseb, ki v danem omrežju sovražijo dano osebo.

Primer:

```
>>> kdo_sovrazi(omrezje, "Vesna")
{"Žiga", "Zoran", "Tina"}
```

3. podnaloga

Sestavite funkcijo `nesrecniki(omrezje)`, ki vrne množico oseb, ki sovražijo same sebe.

Primer:

```
>>> nesrecniki(omrezje)
"Ugo"
```

4. podnaloga

Sestavite funkcijo `najbolj_zadrti(omrezje)`, ki vrne množico vseh oseb, ki v danem omrežju sovražijo največ oseb.

Primer:

```
>>> najbolj_zadrti(omrezje)
{"Žiga", "Ugo"}
```

Kompleksna števila II

1. podnaloga

Sestavljen je razred KompleksnoStevilo z metodama `__init__(self, re, im)`, lastnostima `im` in`re` ter**repr**(self). `Realni del kompleksnega števila je shranjen v spremenljivki_re, imaginarni pa v_im. Metoda`**repr**(self)`predstavi kompleksno število z nizom oblike`KompleksnoStevilo(re, im)`.

Zgled:

```
>>> u = KompleksnoStevilo(3, 4)
>>> u
KompleksnoStevilo(3, 4)
```

Žal so se vrstice v kodi zamešale. uredi jih v pravilni vrstni red. Razen spreminanja vrstnega reda ne naredi nobene druge spremembe

2. podnaloga

Razredu dodaj metodo `__str__(self)`, ki kompleksno stevilo predstavi z nizom oblike npr. `3 + 4i` Pri tem bodi pozoren da: ◦ Če je realni ali imaginarni del števila enak `0`, naj bo v nizu njegov člen izpuščen (npr. namesto `2 + 0i` pišemo samo `2`). ◦ Če je imaginarni del števila enak `1`, namesto `1i` pišemo samo `i`. Če je enak `-1`namesto `-1i` pišemo `-i`. ◦ Če je imaginarni del števila negativen, njegov predznak zamenja `+`. Torej, namesto `2 + (-3)i` pišemo kar `2 - 3i`. Če je realni del enak `0` med predznakom `-` in nadaljevanjem ni presledka. Torej namesto `- 3i` pišemo `-3i`.

Zgled:

```
>>> u = KompleksnoStevilo(3, 4)
>>> print(u)
3 + 4i
>>> v = KompleksnoStevilo(2, 0)
>>> print(v)
2
>>> w = KompleksnoStevilo(0, -4)
>>> print(w)
-4i
>>> w = KompleksnoStevilo(0, 1)
>>> print(w)
i
```

```
>>> y = KompleksnoStevilo(-2, -6)
>>> print(y)
-2 - 6i
>>> z = KompleksnoStevilo(0, 0)
>>> print(z)
0
```

Opozorilo: To in vse ostale podnaloge začnite s `class KompleksnoStevilo(KompleksnoStevilo)`

Ulomki

Da ne bo težav pri testiranju, pri vseh podnalogah začnemo z

```
class Ulomek(Ulomek):
```

Seveda pa 1. podnalogo še vedno začnemo z

```
class Ulomek:
```

1. podnaloga

Sestavite razred `Ulomek`, s katerim predstavimo ulomek. Števec in imenovalec sta celi števili, pri čemer je imenovalec vedno pozitiven. Ulomki naj bodo vedno okrajšani.

Do števca lahko pridemo preko lastnosti `st`, do imenovalca pa preko lastnosti `im`. Če poskusimo spremeniti števec ali imenovalec (torej lastnosti `st` ali `im`) naj koda sproži napako z obvestilom "Obstoječega ulomka ne moremo spreminjati" Zgled:

```
>>> u = Ulomek(5, -20)
>>> u.st
-1
>>> u.im
4
>>> u.st = 4
... Exception("Obstoječega ulomka ne moremo spreminjati")
```

2. podnaloga

Trenutna implementacija razreda `Ulomek` objekt `Ulomek(5, 20)` izpiše z nizom oblike `<__main__.Ulomek object at 0x000002CB41CDD2B0>`, iz katerega ne moremo razbrati za kateri ulomek gre. Kako naj se objekt našega razreda izpiše, lahko sami določimo z metodama `__str__` in `__repr__`. Kako metodi delujeta in kakšna je razlika med njima si poglejte v videu.

Sestavite metodo `__str__(self)`, ki predstavi ulomek z nizom oblike `'st/im'`. Zgled:

```
>>> u = Ulomek(5, 20)
>>> print(u)
1/4
```

3. podnaloga

Sestavite še metodo `__repr__(self)`, ki predstavi ulomek z nizom oblike `'Ulomek(st, im)'`. Zgled:

```
>>> u = Ulomek(5, 20)
>>> u
Ulomek(1, 4)
```

Ploščina pod valom

Z metodo Monte Carlo lahko računamo tudi ploščine. Pri tem gre v grobem za to, da naključno izbiramo točke na nekem pravokotniku in štejemo, koliko točk je takih, da "spadajo" k ploščini. Če razmerje med "zadetki" in vsemi točkami pomnožimo s ploščino pravokotnika, dobimo približek za ploščino območja.

1. podnaloga

Ploščino pod enim valom funkcije sinus (enaka je 2) lahko približno izračunamo tudi tako, da naključno izbiramo točke na pravokotniku [0, Pi] x [0, 1] in s Pi pomnožimo razmerje med točkami pod valom in vsemi točkami.

Sestavi funkcijo `ploscina_val(n)`, ki izračuna ploščino vala funkcije `sin(x)` po opisani metodi. Število naključnih točk funkcija dobi kot parameter.