

# Osnove programiranja v jeziku Python za študente Fakultete za kemijo in kemijsko tehnologijo

Miha Moškon

21. april 2020

Osnutek

# Kazalo

<b>1</b>	<b>Spoznavanje z okoljem</b>	<b>1</b>
1.1	Izbira in namestitvev okolja . . . . .	1
1.2	Okolje IDLE . . . . .	1
1.3	Ukazna vrstica . . . . .	2
1.4	Podatkovni tipi . . . . .	3
1.5	Funkcije . . . . .	4
1.6	Spremenljivke . . . . .	5
1.7	Pisanje programov . . . . .	8
1.8	Funkcija <code>print</code> . . . . .	9
1.9	Funkcija <code>input</code> . . . . .	10
1.10	Pretvarjanje med podatkovnimi tipi . . . . .	10
1.11	Pisanje komentarjev . . . . .	12
<b>2</b>	<b>Pogojni stavek</b>	<b>15</b>
2.1	Zakaj pogojni stavki? . . . . .	15
2.2	Osnovna oblika stavka <code>if</code> . . . . .	16
2.3	Kaj je pogoj? . . . . .	16
2.3.1	Primerjalni operatorji in podatkovni tip <code>bool</code> . . . . .	16
2.3.2	Operatorja vsebovanosti . . . . .	19
2.3.3	Združevanje rezultatov primerjanja . . . . .	19
2.4	Veja <code>else</code> . . . . .	21
2.5	Veja <code>elif</code> in gnezdenje stavkov <code>if</code> . . . . .	23
<b>3</b>	<b>Zanka <code>while</code></b>	<b>27</b>
3.1	Kaj so zanke? . . . . .	27
3.2	Zanka <code>while</code> . . . . .	27
3.3	Štetje z zanko <code>while</code> . . . . .	29
3.4	Stavek <code>+=</code> . . . . .	29
3.5	Mrtva zanka . . . . .	30
3.6	Stavek <code>break</code> . . . . .	30
3.7	Veja <code>else</code> . . . . .	33

<b>4</b>	<b>Seznami in metode</b>	<b>37</b>
4.1	Sekvenčni podatkovni tipi . . . . .	37
4.2	Kaj so sezname? . . . . .	37
4.3	Indeksiranje seznamov . . . . .	38
4.4	Operatorji nad seznamami . . . . .	39
4.5	Spreminjanje in brisanje elementov seznama . . . . .	41
4.6	Vgrajene funkcije nad seznamami . . . . .	41
4.7	Metode . . . . .	41
4.8	Dodajanje elementov . . . . .	42
4.9	Branje seznamov iz ukazne vrstice . . . . .	44
4.10	Sortiranje seznamov . . . . .	45
4.11	Seznami seznamov . . . . .	46
4.12	Generiranje seznamov s funkcijo <code>range</code> . . . . .	48
4.13	Režine . . . . .	52
4.14	Indeksiranje nizov . . . . .	53
4.15	Sprehajanje čez sezname . . . . .	55
<b>5</b>	<b>Zanka <code>for</code></b>	<b>57</b>
5.1	Sprehajanje čez sezname z zanko <code>for</code> . . . . .	57
5.2	Sprehajanje s funkcijo <code>range</code> in sprehajanje čez indekse . . . . .	59
5.3	Sprehajanje čez elemente ali čez indekse? . . . . .	59
5.4	Spreminjanje elementov seznama z zanko <code>for</code> . . . . .	61
5.5	Zanka <code>for</code> ali zanka <code>while</code> ? . . . . .	62
5.6	Stavek <code>break</code> . . . . .	62
5.7	Veja <code>else</code> . . . . .	63
5.8	Gnezdenje zank . . . . .	64
5.9	Izbirni argumenti funkcij in izbirni argumenti funkcije <code>print</code> . . . . .	67
<b>6</b>	<b>Uporaba in pisanje funkcij</b>	<b>69</b>
6.1	Kaj so funkcije in zakaj so uporabne? . . . . .	69
6.2	Kako definiramo funkcijo? . . . . .	70
6.3	Globalni imenski prostor . . . . .	71
6.4	Kaj se zgodi ob klicu funkcije in lokalni imenski prostor . . . . .	71
6.5	Vsaka funkcija vrača rezultat . . . . .	76
6.6	Izbirni argumenti . . . . .	79
<b>7</b>	<b>Uporaba in pisanje modulov</b>	<b>83</b>
7.1	Kaj so moduli? . . . . .	83
7.2	Uporaba modulov . . . . .	83
7.3	Definicija in uporaba lastnih modulov . . . . .	85
7.4	Nameščanje novih modulov . . . . .	85

<b>8 Spremenljivost podatkovnih tipov in terke</b>	<b>87</b>
8.1 Kaj je spremenljivost? . . . . .	87
8.2 Kaj se zgodi ob prirejanju spremenljivk? . . . . .	88
8.3 Kaj se zgodi ob spreminjanju vrednosti spremenljivk? . . . . .	88
8.4 Ali funkcije spreminjajo vrednosti svojim argumentom? . . . . .	90
8.5 Terke . . . . .	92
8.6 Uporaba terk . . . . .	92
8.7 Sezname terk in razpakiranje elementov terk . . . . .	93
8.8 Pakiranje seznamov v seznime terk . . . . .	95
8.9 Zahteva po nespremenljivosti . . . . .	96
<b>9 Slovarji</b>	<b>99</b>
9.1 Zakaj slovarji? . . . . .	99
9.2 Kako uporabljamo slovarje? . . . . .	100
9.3 Iskanje vrednosti . . . . .	100
9.4 Dodajanje in spreminjanje vrednosti . . . . .	101
9.5 Brisanje vrednosti . . . . .	103
9.6 Ključi in vrednosti . . . . .	104
9.7 Imenski prostor in slovarji . . . . .	105
<b>10 Množice</b>	<b>107</b>
10.1 In še množice . . . . .	107
10.2 Uporaba množic . . . . .	107
10.3 Omejitve pri uporabi množic . . . . .	108
10.4 Osnovne operacije nad množicami . . . . .	108
10.5 Presek, unija in razlika . . . . .	109
10.6 Metode množic: dodajanje in brisanje elementov . . . . .	110
10.7 Zgled uporabe množic . . . . .	111

# 1 Spoznavanje z okoljem

## 1.1 Izbira in namestitev okolja

Razlaga in zgledi v knjigi temeljijo na okolju *Python 3*. Osnovno okolje, ki bo za spoznavanje osnov programiranja čisto dovolj, lahko dobite na Pythonovi domači strani (<https://www.python.org/>), kjer si izberete še svoj operacijski sistem in verzijo, ki jo želite namestiti (izberite si distribucijo Python 3...). Če uporabljate operacijski sistem Linux ali OS X, imate Python po vsej verjetnosti že nameščen. Pazite le na to, da uporabljate Python 3, saj večina zgledov v okolju Python 2 ne bo delovala ali pa bo celo delovala nekoliko drugače. Ob nameščanju okolja Python je koristno, če na začetku obkljukate izbiro `Add Python 3... to PATH`, saj si boste s tem mogoče v prihodnosti prihranili kakšno težavo. Če ste ambiciozni in pričakujete, da boste Python v prihodnosti bolj resno uporabljali, si mogoče že zdaj namestite distribucijo *Anaconda* (še ena kača), ki vključuje malo več knjižnic za bolj napredno računanje, analizo podatkov in risanje grafov. Dobite jo na strani <https://www.anaconda.com/>.

## 1.2 Okolje IDLE

Tekom razvoja enostavnejših ali kompleksnih programov ponavadi uporabljamo različna razvojna okolja (angl. *Integrated Development Environment*, IDE), ki združujejo orodja za pisanje, poganjanje in razhroščevanje (angl. *debugging*) programov. Ti pripomočki nam nekoliko olajšajo delo in naredijo razvoj kompleksnejših (ali pa tudi enostavnejših) programov nekoliko bolj pregleden. Z namestitvijo osnovnega okolja Python smo na svoj računalnik namestili tudi okolje IDLE, ki predstavlja enostavno razvojno okolje za jezik Python in katerega bomo uporabljali pri spoznavanju osnov programiranja. Zaženimo ga in začnimo (v operacijskem sistemu Windows boste IDLE zagnali tako, da v Start meni napišete `idle` in pritisnete tipko `Enter`).

### 1.3 Ukazna vrstica

Pred nami se je pojavila *ukazna* oziroma *pozivna vrstica*, s pomočjo katere lahko začnemo naš pogovor s tolmačem jezika Python (angl. *Python interpreter*). Mi mu bomo v obliki stavkov podajali ukaze, on pa jih bo izvršil in nam ponavadi tudi vrnil ali pa izpisal nek rezultat. Ukaz napišemo in tolmaču pošljemo tako, da pritisnemo tipko **Enter**.

Poskusimo nekaj osnovnih stvari.

```
>>> 1 + 2
3
>>> 1 - 2
-1
>>> 1 * 2
2
>>> 1 - 3 * 4
-11
```

Tolmaču torej lahko podam nek izraz sestavljen iz operandov (v tem primeru števil) in operatorjev (npr. +, \* in -) in on mi bo vrnil rezultat. Iz zadnjega izraza vidimo tudi, da ima množenje prednost pred odštevanjem. Uporabim lahko tudi oklepaje, s katerimi določim vrstni red računanja:

```
>>> (1 - 3) * 4
-8
```

Kaj pa deljenje?

```
>>> 5 / 4
1.25
>>> 10 / 5
2.0
```

Uporabim lahko tudi celoštevilsko deljenje (//) in ostanek pri deljenju oziroma operacijo modulo (%):

```
>>> 5 // 4
1
>>> 10 // 5
2.0
>>> 10 % 4
2
```

## 1.4 Podatkovni tipi

Celoštevilsko deljenje očitno vedno vrne celo število, običajno deljenje pa vrne decimalno število, tudi če je rezultat celo število. Operaciji sta pač definirani tako, da vsakič vrneta določen **podatkovni tip**, ki je poleg tega odvisen tudi od podatkovnega tipa vhodov. Podatkovni tip? Vse vrednosti, ki smo jih tolmaču podali, so imele določen podatkovni tip, in sicer smo v vseh primerih zgoraj podajali cela števila (angl. *integer*), ki jim Python pravi **int**. Rezultati izvedbe gornjih stavkov so v določenih primerih prav tako predstavljali cela števila, pri deljenju pa smo dobili decimalna števila oziroma *števila v plavajoči vejici* (angl. *floating point number*), ki jim Python pravi **float**. Decimalna števila lahko tolmaču podamo tudi kot vhodne podatke:

```
>>> 5.4 + 4.6
10.0
>>> 6.3 / 2
3.15
>>> 4 ** 0.5
2.0
```

Mimogrede, operator **\*\*** določa potenciranje, potenciranje z vrednostjo 0.5 pa je enako kvadratnemu korenu.

Python zna poleg s števili delati tudi z drugimi podatkovnimi tipi, npr. nizi (angl. *string*), oziroma po njegovo s podatkovnim tipom **str**. Nizi vsebujejo sekvenco poljubnih znakov (črke, števila, ločila, posebni simboli ipd.), zapisujemo pa jih znotraj dvojnih (") ali enojnih navednic ('). Poskusimo:

```
>>> "niz1"
'niz1'
>>> 'niz2'
'niz2'
```

Python očitno ne loči med dvojnimi in enojnimi navednicami. Mi jih lahko uporabljamo čisto po navdihu. Kako pa znotraj niza zapisati enojne navednice? Tako da jih ovijemo v dvojne navednice. In obratno. Poskusimo:

```
>>> 'Rekel_je:"Daj_mi_mir!'"
'Rekel_je:"Daj_mi_mir!'"
>>> "'Znamo'_programirati."
"'Znamo'_programirati."
```

Ali lahko tudi nad nizi izvajamo operacije od prej. Poskusimo:

```
>>> 'niz1' + 'niz2'
'niz1niz2'
>>> 'niz1' + '_' + 'niz2'
```

## 4 Poglavlje 1 Spoznavanje z okoljem

```
'niz1_niz2'
>>> 'niz1' * 3
'niz1niz1niz1'
>>> 'niz1' * 'niz2'
Traceback (most recent call last):
  File "<pyshell#36>", line 1, in <module>
    'niz1' * 'niz2'
TypeError: can't multiply sequence by non-int of type 'str'
```

Python nam je v zadnjem primeru vrnil napako, kar pomeni, da operacija, ki smo jo hoteli izvesti ni podprta. Ko dobimo napako, se je ne ustrašimo, ampak jo preberimo, saj nam sporoča kaj je narobe. Python torej množenja dveh nizov med seboj ne podpira. Tudi seštevanje in množenje je nad nizi definirano nekoliko drugače kot nad števili. Kaj pa če niz vsebuje samo števila?

```
>>> '3' + '5'
'35'
>>> '3' * '5'
Traceback (most recent call last):
  File "<pyshell#41>", line 1, in <module>
    '3' * '5'
TypeError: can't multiply sequence by non-int of type 'str'
```

Rezultat je podoben kot prej. Podatkovni tip operanda oziroma podatka torej določa kaj in kako lahko s posameznim podatkom počnemo. Števila lahko npr. seštevamo, če pa poskusimo sešteti dva niza, izvajamo operacijo lepljenja nizov oziroma njihovo *konkatenacijo*.

## 1.5 Funkcije

Poleg osnovnih operatorjev so v osnovnem Python okolju vgrajene tudi določene *funkcije*. Podobno kot v matematiki tudi pri programiranju funkcije sprejemajo določene vhode oziroma *argumente*. Funkcijo pokličemo tako, da podamo njeno ime in argumente oziroma vhode funkciji zapišemo znotraj oklepajev. Takole:

```
ime_funkcije(argument_1, argument_2, ..., argument_n)
```

Po klicu se bo funkcija izvedla in nekaj koristnega naredila in / ali vrnila nek uporaben rezultat. Če bi želeli izvedeti, kateremu podatkovnem tipu pripada nek podatek, bi lahko npr. poklicali funkcijo `type`:

```
>>> type(1)
<class 'int'>
>>> type(1.0)
<class 'float'>
```



```
>>> type('niz')
<class 'str'>
```

Poglejmo si še funkcijo `abs`, ki izračuna absolutno vrednost podanega argumenta:

```
>>> abs(-1)
1
>>> abs(-1.4)
1.4
>>> abs(5)
5
>>> abs('niz')
Traceback (most recent call last):
  File "<pyshell#57>", line 1, in <module>
    abs('niz')
TypeError: bad operand type for abs(): 'str'
```

Tudi argumenti funkcij so torej omejeni na določene podatkovne tipe, kar je smiselno. Funkcija `abs` je na primer omejena zgolj na števila (`int` ali `float`), saj absolutne vrednosti niza pač ne moremo izračunati.

Python ima vgrajenih še kar nekaj funkcij, ki pa jih bomo večinoma spoznavali sproti.

## 1.6 Spremenljivke

Do zdaj smo v ukazno vrstico pisali izraze sestavljene in operatorjev in operandov (podatkov). Python je po vsakem pritisku tipke **Enter** podan izraz pognal in vrnil rezultat, ki pa ga je takoj zatem pozabil. Do dobljenega rezultata tako ne moremo več priti drugače, kot da še enkrat podamo enak izraz, ki ga bo Python ponovno ovrednotil in vrnil enak rezultat. Pogosto pa si želimo rezultate izvedenih stavkov zapomniti oziroma želimo, da jih Python spravi za kasneje. Na ta način lahko sestavljamo kompleksnejše izraze (brez da bi pisali dolge kače čez več vrstic), izračunan podatek uporabimo večkrat in z njim delamo različne stvari (npr. uporaba v drugih izrazih, izpis na zaslon, shranjevanje na trdi disk itd.). Python omogoča, da posameznemu podatku dodelimo ime, preko katerega bomo lahko do tega podatka dostopali še kasneje. Takole:

```
>>> x = 1
>>> y = 2 - 3.5
>>> niz = 'abc'
```

Zdaj Python ni ničesar izpisal, je pa vrednost na desni strani *prireditvenega stavka* priredil imenu na levi strani prireditvenega stavka. Izvedli smo torej prireditev vrednosti na desni imenu na levi, ki mu pravimo tudi **spremenljivka**. Pri tem

## 6      Poglavje 1      Spoznavanje z okoljem

smo uporabili prireditveni operator `=`. Pozor: to ni operator enakosti, saj vedno deluje samo v eno smer, in sicer tistemu, kar napišemo na levi strani, priredi tisto, kar napišemo na desni strani. Če npr. napišemo

```
>>> x = x + 2
```

to ni nerešljiva enačba (kot bi bila v primeru, ko enačaj obravnavamo kot operator enakosti), ampak zgolj pomeni, da vzamemo vrednost, ki stoji za imenom `x`, to vrednost povečamo za 2 in priredimo imenu `x`. Zgornji stavek torej vrednost v spremenljivki `x` poveča za 2. Kako pa dostopamo do vrednosti posamezne spremenljivke oziroma do vrednosti, ki stoji za določenim imenom? To smo naredili že zgoraj – tako da podamo ime spremenljivke. Ko smo zgoraj napisal ime `x` na desni strani prireditvenega stavka, je Python pogledal kaj za tem imenom stoji in ime zamenjal z vrednostjo za njim. Drugače je, če ime uporabimo na levi strani prireditvenega stavka. S tem namreč imenu priredimo novo vrednost, če pa imena pred tem še nismo definirali, s tem ustvarimo tudi novo ime. Temu rečemo **definicija spremenljivke**.

Do vrednosti spremenljivke `x` ali `y` bi zdaj lahko dostopali tako, da podamo njeno ime:

```
>>> x
3
>>> y
-1.5
```

Ime spremenljivke lahko uporabimo tudi kot argument funkcije, npr. takole:

```
>>> abs(x)
3
>>> abs(y)
1.5
```

Prav tako lahko izhod funkcije priredimo novi (ali obstoječi) spremenljivki:

```
>>> z = abs(y)
```

Zdaj Python ni izpisal ničesar, je pa ustvaril novo spremenljivko, do katere lahko dostopamo:

```
>>> z
1.5
```

Kaj pa bi se zgodilo, če pokličemo ime spremenljivke, ki je še nimamo:

```
>>> novo_ime
Traceback (most recent call last):
  File "<pyshell#63>", line 1, in <module>
    novo_ime
NameError: name 'novo_ime' is not defined
```

Ker tega imena Python ne pozna, saj ga še nismo definirali, javi napako. Dostopamo lahko torej le do imen, ki smo jih bodisi definirali mi ali pa so že definirana (kot npr. `abs`).

Kaj pa bi se zgodilo, če bi imenu vgrajene funkcije priredili neko vrednost? Če bi npr. izvedli prireditvev

```
>>> abs = 5
```

bo Python to brez pritoževanja tudi izvedel. Poskusimo zdaj še enkrat izračunati absolutno vrednost tistega, kar se skriva za spremenljivko `y`:

```
Traceback (most recent call last):
  File "<pyshell#68>", line 1, in <module>
    abs(y)
TypeError: 'int' object is not callable
```

Seveda bo prišlo do napake, saj smo si funkcijo za izračun absolutne vrednosti *povozili* z vrednostjo, ki pripada podatkovnemu tipu `int`. Za imenom `abs` po novem Python nima več shranjene funkcije za izračun absolutne vrednosti, ampak število 5. Nerodno. Stvar lahko rešimo tako, da okolje IDLE resetiramo (`Shell` → `Restart Shell` oziroma kombinacija tipk `Ctrl + F6`).

V splošnem velja, da se moramo pri poimenovanju spremenljivk držati določenih pravil. Kot smo videli prej, uporaba imen, ki so že rezervirana oziroma uporabljena, ni priporočena. Prireditvev vrednosti rezerviranim imenom ni zgolj slaba, ampak celo vrne napako:

```
>>> if = 5
SyntaxError: invalid syntax
```

Zgoraj vidimo, da je besedica `if` t.i. rezervirano ime, saj jo IDLE obarva drugače kot ostale besede oziroma jo odebeli. Njeno uporabo bomo spoznali prav kmalu. Pri imenih spremenljivk nam Python poleg tega ne bo pustil uporabe presledkov:

```
>>> moje ime = 'Miha'
SyntaxError: invalid syntax
```

Stvar lahko rešimo tako, da presledke zamenjamo s podčrtaji (`_`):

```
>>> moje_ime = 'Miha'
>>> moje_ime
'Miha'
```

Poleg tega se je pri poimenovanju spremenljivk dobro držati še določenih priporočil. Navedimo jih nekaj:

- spremenljivke naj imajo smiselna imena, ki programerju sporočajo pomen spremenljivke;

- pri poimenovanju spremenljivk se izogibajmo črkam, ki ne nastopajo v osnovni angleški abecedi (izogibamo se npr. črkam č, ž in š);
- imena so lahko sestavljena iz več besed, pri čemer te ločimo s podčrtaji.

## 1.7 Pisanje programov

S tolmačem smo se do zdaj pogovarjali preko ukazne vrstice, čemur bi pa težko rekli programiranje. V ukazno vrstico bi lahko sicer napisali zaporedje stavkov, s katerim bi nekaj izračunali, oziroma s katerim bi rešili nek problem. Če pa bi hoteli to zaporedje stavkov pognati še enkrat (mogoče na drugih podatkih), bi morali v ukazno vrstico stavke v enakem vrstnem redu napisati ponovno. Očitno je, da to ni najbolj priročen način programiranja.

V splošnem zaporedja stavkov zapisujemo v tekstovne datoteke – v programe, ki jih potem v poganjanje predamo našemu tolmaču. Tolmač bo ukaze v datoteki izvedel po vrsti na podoben način, kot če bi te zapisali v ukazno vrstico. Kako lahko za pisanje takih programov uporabimo orodje IDLE? Najprej bomo ustvarili novo datoteko – program, in sicer z izbiro menija **File** → **New File** oziroma s kombinacijo tipk **Ctrl** + **N**. V primeru, da datoteka z nekim programom že obstaja, lahko to odpremo preko menija **File** → **Open** oziroma s kombinacijo tipk **Ctrl** + **O**. V obeh primerih s tem odpremo tudi IDLE-ov urejevalnik teksta, s pomočjo katerega lahko napišemo program, tega shranimo in na koncu poženemo oziroma v poganjanje damo tolmaču. Kot ste verjetno že vajeni, končnica datotek podaja tudi njihovo vsebino. Datoteke, v katerih je shranjen program v jeziku Python, prepoznamo preko končnice **.py**. Programe, ki jih bomo pisali, bomo torej tudi mi opremili s tako končnico.

Napišimo krajši program, ki temperaturo v stopinjah Celzija pretvori v temperaturo v stopinjah Fahrenheit z upoštevanjem enačbe  $T_F = T_C * 1.8 + 32$ , pri čemer  $T_C$  predstavlja temperaturo v stopinjah Celzija,  $T_F$  pa temperaturo v stopinjah Fahrenheit. Pri tem zaenkrat predpostavljajmo, da je  $T_C$ , ki ga želimo pretvoriti, enak 20. Program bo sledeč<sup>1</sup>:

```
1 T_C = 20
2 T_F = T_C * 1.8 + 32
```

Program lahko zdaj shranimo, npr. v datoteko z imenom **temperatura.py** in poženemo z izbiro menija **Run** → **Run Module** oziroma s pritiskom na tipko **F5**. Izvršil se je preklon na ukazno vrstico, nikjer pa ni vidnega rezultata izvršitve našega programa. Ali se je program res izvedel? Preverimo lahko tako, da v ukazno vrstico napišemo imeni spremenljivk, ki smo jih v programu definirali. Če se program ni pognal, bo Python vrnil napako.

<sup>1</sup>Kadar bomo v zgledih kode vrstice označili z njihovimi števkami, bo to pomenilo, da gre za program.

```
>>> T_C
20
>>> T_F
68.0
```

Program se je očitno izvedel, saj sta spremenljivki definirani. Program pa pred tem ni ničesar izpisal. Program ne izpiše ničesar, tudi če ga dopolnimo z zgornjima vrsticama:

```
1 T_C = 20
2 T_F = T_C * 1.8 + 32
3 T_C
4 T_F
```

Zakaj ne? Ko svojo kodo zapakiramo v programe, ti izpisujejo vrednosti samo takrat, ko to od njih eksplicitno zahtevamo. Kako? S funkcijo `print`.

## 1.8 Funkcija `print`

Funkcija `print` nam omogoča izpisovanje vrednosti znotraj programov. Pokličemo jo tako, da ji kot argumente naštejemo vrednosti, ki jih želimo izpisati in funkcija `print` bo vrednosti pač izpisala, vmes bo dala presledke, na koncu izpisa pa bo skočila v novo vrstico. Dopolnimo zgornji program, tako da izpiše obe temperaturi:

```
1 T_C = 20
2 T_F = T_C * 1.8 + 32
3 print(T_C)
4 print(T_F)
```

Funkciji smo podali ime spremenljivke, izpisala pa je vrednost, ki stoji za imenom:

```
20
68.0
```

Funkcija `print` je na koncu vsakega izpisa avtomatsko skočila v novo vrstico. Obe temperaturi bi lahko izpisali tudi v isti vrstici, in sicer takole:

```
print(T_C, T_F)
```

V tem primeru je izpis sledeč:

```
20 68.0
```

Funkcija `print` je torej podani vrednosti avtomatsko ločila s presledkom. Poskusimo zdaj izpis še malo olepšati. Poleg imen spremenljivk lahko kot argumenti nastopajo tudi fiksne vrednosti. Lahko bi v izpis dodali še niz, ki bi naredil vse skupaj malo bolj informativno. Takole

```
print(T_C, "stopinj_Celzija_je_enako", T_F, "stopinj_Fahrenheit.")
```

V tem primeru je izpis sledeč: »> input("Vnesi svoje ime: ") Imena spremenljivk je Python torej zamenjal z njihovimi vrednostmi, nize pa je izpisal kakor so bili pač podani.

## 1.9 Funkcija input

Še malo pa bomo napisal naš prvi pravi program. Pretvarjanje iz stopinje Celzija v stopinje Fahrenheit sicer deluje, malo pa je moteče to, da lahko pretvarjamo samo eno vrednost, ki je že vnaprej določena. Program bi bil bistveno bolj uporaben, če bi lahko vrednost, ki jo želimo pretvoriti, podal kar uporabnik ob zagonu programa. To nam omogoča funkcija `input`.

Funkcija `input` prav tako kot `print` na zaslon izpiše podan argument. Za razliko od funkcije `print`, funkcija `input` sprejema samo en argument tipa `str`. Preko tega argumenta bomo funkciji `input` podali navodilo za uporabnika. Na primer takole:

```
>>> input("Vnesi svoje ime: ")
```

Funkcija `input` čaka na uporabnikov vnos in pritisk tipke **Enter**. Če smo to funkcijo pognali iz ukazne vrstice, bo uporabnikov vnos ponovila. Lahko pa uporabnikov vnos shranimo v spremenljivko in ga kasneje uporabimo. Takole

```
>>> ime = input("Vnesi svoje ime: ")
```

Rezultat funkcije smo torej priredili spremenljivki `ime`. Zdaj lahko do tistega, kar je uporabnik vnesel, dostopamo preko imena spremenljivke. Uporabimo to na našem programu za pretvarjanje med temperaturami. Temperaturo v stopinjah bomo zdaj prebrali od uporabnika preko funkcije `input`.

```
1 T_C = input("Vnesi temperaturo v stopinjah Celzija: ")
2 T_F = T_C * 1.8 + 32
3 print(T_C, "stopinj Celzija je enako", T_F, "stopinj Fahrenheit.")
```

Program pa v tej obliki žal še ne bo deloval. Funkcija `input` namreč vedno vrne niz, saj je to podatkovni tip, v katerega lahko zapiše karkoli bo pač uporabnik vnesel. Tudi če bo uporabnik vnesel število, bo to predstavljeno kot niz oziroma podatkovni tip `str`. Kakor se spomnimo od prej pa nizov ne moremo množiti z decimalnimi števili pa tudi seštevanje je denifinirano tokrat za nas nekoliko neugodno. Prebrano število, ki je zapisano kot niz, moramo torej pred nadaljnjo obdelavo pretvoriti v nekaj, s čimer lahko računamo, npr. `float`.

## 1.10 Pretvarjanje med podatkovnimi tipi

Pretvorbo podatka v posamezen podatkovni tip lahko izvedemo z vgrajenimi funkcijami, ki nosijo enako ime, kot podatkovni tip, v katerega želimo pretvarjati.

Če bi želeli nekaj pretvoriti v niz, bi torej uporabili funkcijo `str`, v celo število funkcijo `int`, v decimalno pa funkcijo `float`. Poskusimo:

```
>>> niz = str(20)
>>> niz
'20'
>>> int(niz)
20
>>> float(niz)
20.0
>>> x = 5.4
>>> str(x)
'5.4'
>>> int(x)
5
```

Rezultat izvedbe posamezne funkcije je torej zapis podatka v željenem podatkovnem tipu. Pri tem so seveda upoštevane omejitve posameznega podatkovnega tipa. Ko npr. pretvarjamo v podatkovni tip `int`, funkcija `int` poreže decimalke za decimalno piko (brez zaokroževanja).

```
>>> int(6.9)
6
```

Drug primer take omejitve je, da nize lahko pretvarjamo v števila le v primeru, ko vsebujejo zgolj in samo števila (lahko tudi decimalno piko):

```
>>> float("6.9")
6.9
>>> int("6a")
Traceback (most recent call last):
  File "<pyshell#19>", line 1, in <module>
    int("6a")
ValueError: invalid literal for int() with base 10: '6a'
>>> float("stevilo")
Traceback (most recent call last):
  File "<pyshell#20>", line 1, in <module>
    float("stevilo")
ValueError: could not convert string to float: 'stevilo'
```

Dokončajmo zdaj naš prvi program.

**Zgled 1** *Napiši program, ki od uporabnika prebere temperaturo v stopinjah Fahrenheit in to pretvori v stopinje Celzija ter poda izpis obeh temperatur.*

**Rešitev 1** *Rešitev naloge praktično že imamo. Malenkost jo moramo le še dopolniti.*

```

1 niz = input("Vnesi_temperaturo_v_stopinaj_Celzija:_")
2 T_C = float(niz)
3 T_F = T_C * 1.8 + 32
4 print(T_C, "stopinj_Celzija_je_enako", T_F, "stopinj_Fahrenheit.")

```

Zgornji dve vrstici bi lahko združili tudi v eno samo:

```

1 T_C = float(input("Vnesi_temperaturo_v_stopinaj_Celzija:_"))
2 T_F = T_C * 1.8 + 32
3 print(T_C, "stopinj_Celzija_je_enako", T_F, "stopinj_Fahrenheit.")

```

Program zdaj deluje kakor bi želeli. Do problema pride samo takrat, ko uporabnik vnese kakšno neumnost.

## 1.11 Pisanje komentarjev

Ponavadi je dobro, da programe pišemo na tak način, da jih bodo razumeli tudi drugi in da jih bomo mogoče razumeli sami, ko jih bomo čez par mesecev ponovno pregledovali. Pri tem pomaga že to, da se trudimo pisati lepo in pregledno kodo ter spremenljivke poimenujemo tako, da vsaj približno vemo kaj predstavljajo ( $T_F$  in  $T_C$ ). Dodatno pa k razumevanju napisanih programov pripomorejo komentarji. Komentarji služijo opombam, ki jih sebi ali drugim pišemo znotraj naših programov, niso pa namenjeni izvajanju. Pythonu moramo torej nekako povedati, da naj komentarjev ne poganja. To mu lahko sporočimo z uporabo določenih znakov.

Znak `#` označuje vrstični komentar, in sicer bo Python preskočil vso kodo od začetka znaka `#` do konca vrstice, v kateri se ta znak nahaja. Primer uporabe je sledeč:

```

>>> # napisemo lahko karkoli
>>> x = 1 # prireditev bo pognal, komentarja pa ne
>>> x
1

```

Včasih si želimo napisati daljši (večvrstični komentar). Tega začnemo s tremi enojnimi ali tremi dvojnimi narekovaji in ga končamo s tremi enojnimi ali tremi dvojnimi narekovaji (s tistimi pač, s katerimi smo komentar začeli).

Povadimo zdaj oboje na našem prvem programu.

```

1 """
2 Program, ki pretvori prebrano temperaturo
3 iz stopinj Celzija v stopinje Fahrenheit
4 """
5
6 # branje in pretvorba v decimalno stevilo
7 T_C = float(input("Vnesi_temperaturo_v_stopinaj_Celzija:_"))

```



```
8
9 T_F = T_C * 1.8 + 32 # enacba pretvorbe
10
11 # izpis na zaslon
12 print(T_C, "stopinj_Celzija_je_enako", T_F, "stopinj_Fahrenheit.")
```

Osnutek

Osnutek

## 2 Pogojni stavek

### 2.1 Zakaj pogojni stavki?

Vsi programi, ki smo jih do zdaj napisali (je bil mogoče samo eden?), so se izvedli po vnaprej določenem zaporedju. Od zgoraj navzdol so se namreč lepo po vrsti izvedli vsi stavki v programu. V določenih primerih pa bi posamezne stavke radi izvedli samo ob izpolnjenosti (ali pa neizpolnjenosti) izbranega pogoja. Poglejmo si spodnji primer:

**Zgled 2** *Napiši program, ki od uporabnika prebere telesno težo in višino in izpiše uporabnikov indeks telesne mase.*

**Rešitev 2** *Preko funkcije `input` bomo torej prebrali težo in višino. Ker funkcija vrača niz, bomo obe vrednosti pretvorili v decimalno število (`float`). Potem bomo uporabili enačbo za izračun indeksa telesne mase:  $itm = \frac{teza}{visina^2}$ . Pri tem mora biti teža podana v kilogramih, višina pa v metrih.*

```
1 teza = float(input("Vpisi svojo tezo [kg]: "))
2 visina = float(input("Vpisi svojo višino [m]: "))
3 itm = teza/visina**2
4 print("Tvoj ITM je", itm)
```

Zgornji program je popolnoma pravilen, bi ga pa radi še malo dopolnili. Veliko uporabnikov verjetno ne ve kaj posamezna vrednost indeksa telesne mase (ITM) pomeni. Ali mora shujšati, je njegova teža ustrezna, ali bi se moral malo zrediti? Če malo poenostavimo, lahko ljudi razdelimo v tri skupine, in sicer tiste s premajhno težo, tiste z ustrezno težo in tiste s preveliko težo:

pogoj	skupina
$ITM < 18.5$	podhranjenost
$18.5 \leq ITM \leq 25$	normalna teža
$25 < ITM$	debelost

Naš program bi torej radi dopolnili tako, da bo uporabniku izpisal tudi informacijo o tem, v katero skupino spada. Z drugimi besedami, če bo izpolnjen prvi pogoj ( $ITM < 18.5$ ), bi radi izpisali, da je uporabnik podhranjen, če bo izpolnjen drugi pogoj ( $18.5 \leq ITM \leq 25$ ), da je njegova teža ustrezna in če bo izpolnjen tretji pogoj

( $25 < \text{ITM}$ ), da je pretežak. Radi bi torej, da se določeni deli našega programam (v konkretnem primeru različni izpisi) izvedejo v odvisnosti od vrednosti ITM.

## 2.2 Osnovna oblika stavka `if`

Za pisanje pogojnih stavkov bomo uporabili Pythonov stavek `if`. Njegova osnovna oblika je sledeča:

```
if pogoj:
    # pogojni_stavki so zamaknjeni
    # pogojni stavki
    # ce je pogoj izpolnjen
    ...
# skupni stavki
# ni vec zamaknjeno
...
```

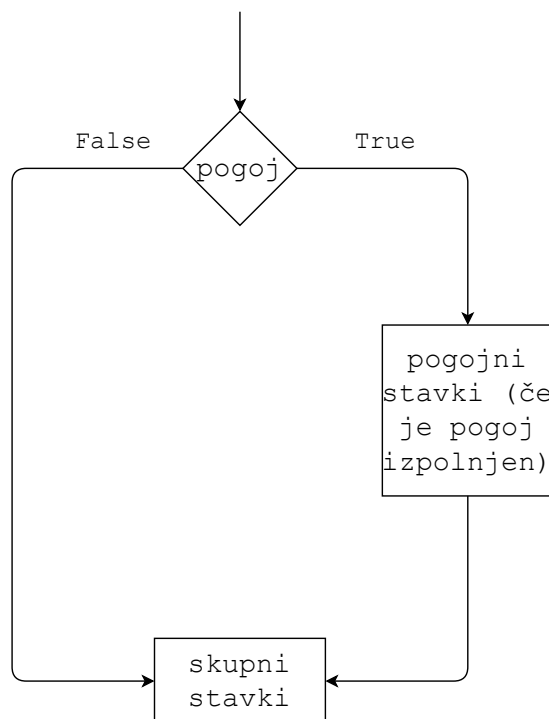
Začnemo torej z rezervirano besedico `if`, ki ji sledi pogoj. Temu sledi dvopičje (`:`), s katerim povemo, da je konec pogoja. Potem sledi pogojni del, ki se bo izvedel samo v primeru, da je podan pogoj izpolnjen (glej sliko 2.1). Pogojnemu delu sledijo stavki, ki se bodo izvedli v vsakem primeru, ne glede na izpolnjenost pogoja. Kako pa Pythonu povemo kje je konec pogojnega dela? Z zamikom (angl. *indent*). Zgoraj smo tiste stavke, ki se izvedejo samo v primeru izpolnjenosti pogoja zamaknili, tako da smo pred njih vstavili tabulator (tipka *Tab*) ali par presledkov (če smo pikolovski, se držimo konvencije štirih presledkov). Pogojni del smo zaključili tako, da smo enostano nehali zamikati. Izvedbo zgornje kode prikazuje diagram poteka na sliki 2.1.

## 2.3 Kaj je pogoj?

Kaj pravzaprav predstavlja pogoj za izvedbo pogojnega dela stavka? Kot je razvido iz slike 2.1 je pogoj nekaj, kar je lahko resnično (angl. *true*) ali neresnično (angl. *false*). Pogoj moramo torej zastaviti kot vprašanje, na katerega lahko odgovorimo bodisi z odgovorom *da* ali z odgovorom *ne*. Pri formiranju vprašanja oziroma pogoja bomo torej večinoma uporabljali operatorje, ki vračajo take odgovore. Tem operatorjem pravimo *pogojni operatorji* pa si pogledjmo nekaj njihovih primerov.

### 2.3.1 Primerjalni operatorji in podatkovni tip `bool`

Prva skupina operatorjev, ki jih bomo uporabljali pri pisanju pogojev so t.i. *primerjalni operatorji*, ki jih poznamo že iz matematike. To so npr. operator enakosti `==` (ker je enojni enačaj uporabljen za prirejanje, moramo za primerjanje uporabiti dvojni enačaj), operator neenakosti `!=`, večji `>`, večji ali enak `>=` itd. Poskusimo:



**Slika 2.1** Diagram poteka osnovne oblike stavka `if`. V primeru, če je pogoj izpolnjen, se izvede pogojni del.

```

>>> 1 == 1
True
>>> 1 == 2
False
>>> 1 != 2
True
>>> 1 > 2
False
>>> 1 <= 2
True
>>> 1 == 2.5
False
>>> 1 == 1.0
True

```

S primerjalnimi operatorji torej lahko primerjamo med seboj dva podatka, rezultat primerjanja pa je bodisi vrednost `True` ali `False`. Rezultat primerjanja je podatek, ki lahko zavzame samo te dve vrednosti. Kakšen je podatkovni tip tega podatka?

```

>>> type(True)

```

```
<class 'bool'>
>>> type(False)
<class 'bool'>
```

Za oblikovanje pogojev imamo torej na voljo poseben podatkovni tip, tj. `bool`, oziroma po angleško *boolean*, ki lahko zavzame samo dve vrednosti, tj. `True` ali `False`.

Poskusimo uporaba primerjalnih operatorjev uporabiti na zgledu iz začetka poglavja.

**Zgled 3** *Napiši program, ki od uporabnika prebere telesno težo in višino in izpiše uporabnikov indeks telesne mase (ITM). Poleg tega program uporabniku pove, v katero skupino spada.*

**Rešitev 3** *Program od prej bomo dopolnili s pogojnimi stavkom. Če je ITM manjši od 17.5, lahko program izpiše, da je uporabnikova teža premajhna. Če je ITM večji od 25, lahko program izpiše, da je uporabnikova teža prevelika. Kaj pa vmes? Tega pa zaenkrat še ne znamo.*

```
1 teza = float(input("Vpisi svojo tezo [kg]: "))
2 visina = float(input("Vpisi svojo višino [m]: "))
3 itm = teza/visina**2
4 print("Tvoj ITM je", itm)
5 if itm < 17.5:
6     print("Tvoja teža je premajhna")
7 if itm > 25:
8     print("Tvoja teža je prevelika")
```

Primerjalne operatorje lahko uporabimo tudi nad podatki, ki niso števila

```
>>> "abc" == "abc"
True
>>> "abc" == "ABC"
False
>>> "abc" < "b"
True
>>> "abc" < "abc"
False
>>> "abc" < "abd"
True
```

Iz zgornjih zgledov vidimo npr., da Python loči med velikimi in malimi črkami in da so določeni nizi manjši od drugih. Kako pa primerjanje dveh nizov poteka? Na enak način, kot primerjamo nize, ko poskušamo besede sortirati po abecedi

(npr. v slovarju ali telefonskem imeniku). Dve besedi primerjamo znak po znaku od začetka do konca, dokler ne pridemo do dveh znakov, ki se razlikujeta ali do konca ene izmed besed. Če se besedi ujemata po vseh znakih in je ena beseda krajša, je krajša beseda zagotovo manjša. Npr., beseda "Beda" je manjša od besede "Bedarija" (v slovarju bo verjetno Beda nastopala pred Bedarijo):

```
>>> "Beda" < "Bedarija"
True
```

Beseda "Bedno" pa ni manjša od besede "Bedarija", čeprav je od nje krajša. Zakaj ne? Zato, ker se besedi razlikujeta v črtem primerjanju na znakih "n" in "a" in ker "n" pač ni manjši od znaka "a".

```
>>> "Bedno" < "Bedarija"
False
```

Takemu primerjanju pravimo *leksikografsko* primerjanje.

### 2.3.2 Operatorja vsebovanosti

Ko smo ravno pri nizih, lahko nad njimi definiramo še *operatorja vsebovanosti*, ki preverjata ali nekaj je (*in*) oziroma ni (*not in*) v posameznem nizu vsebovano. Operatorja bomo uporabljali tudi na drugih podatkovnih tipih, ki podobno kot nizi, vsebujejo druge podatke – nizi so sestavljeni iz več znakov oziroma podnizov. Če je nek niz *podniz* vsebovan v nekem nizu *niz*, lahko preverim takole:

```
>>> podniz in niz
```

Povadimo:

```
>>> "Beda" in "Bedarija"
True
>>> "beda" in "Bedarija"
False
>>> "ana" in "anakonda"
True
>>> "a" in "abeceda"
True
```

Spet vidimo, da je znak "b" nekaj drugega kot znak "B" in da Python loči med malimi in velikimi črkami.

### 2.3.3 Združevanje rezultatov primerjanja

Pri reševanju naloge z izpisovanjem podatkov o ITM imamo še vedno težave s primerom, kjer morata biti izpolnjena dva pogoja hkrati ( $ITM \geq 18.5$  in  $ITM \leq 25$ ). Končen pogoj za izvedbo izpisa *Tvoja teza je ustrezna*, moramo torej

sestaviti iz dveh pogojev. Za ta namen lahko uporabimo t.i. *logične operatorje*, ki omogočajo medsebojno združevanje več spremenljivk tipa `bool`. Osnovna logična operatorja, ki ju bomo uporabljali v takem primeru sta operator `and` in operator `or`. Njuno delovanje lahko ponazorimo s spodnjo tabelo:

pogoj1	pogoj2	pogoj1 and pogoj2	pogoj1 or pogoj2
False	False	False	False
False	True	False	True
True	False	False	True
True	True	True	True

V primeru, da morata biti izpolnjena oba vhodna pogoja, torej uporabimo operator `and`. Če je dovolj, da je izpolnjen samo eden izmed vhodnih pogojev, uporabimo operator `or`. Pogosto uporabljen logični operator je še operator `not`, ki `True` spremeni v `False` in obratno:

```
>>> not True
False
>>> not False
True
```

Zdaj lahko dokončamo zgled z izpisovanjem podatkov o ITM.

**Zgled 4** *Napiši program, ki od uporabnika prebere telesno težo in višino in izpiše uporabnikov indeks telesne mase (ITM). Poleg tega program uporabniku pove, v katero skupino spada.*

**Rešitev 4** *Zdaj lahko dodamo še pogojni stavek, pri katerem bo pogoj sestavljen iz dveh delov. V tem primeru mora biti vrednost spremenljivke `ITM` večja ali enaka od 17.5 in manjša ali enaka od 25, kar lahko zapišemo s pogojem `17.5 <= ITM and ITM <= 25`.*

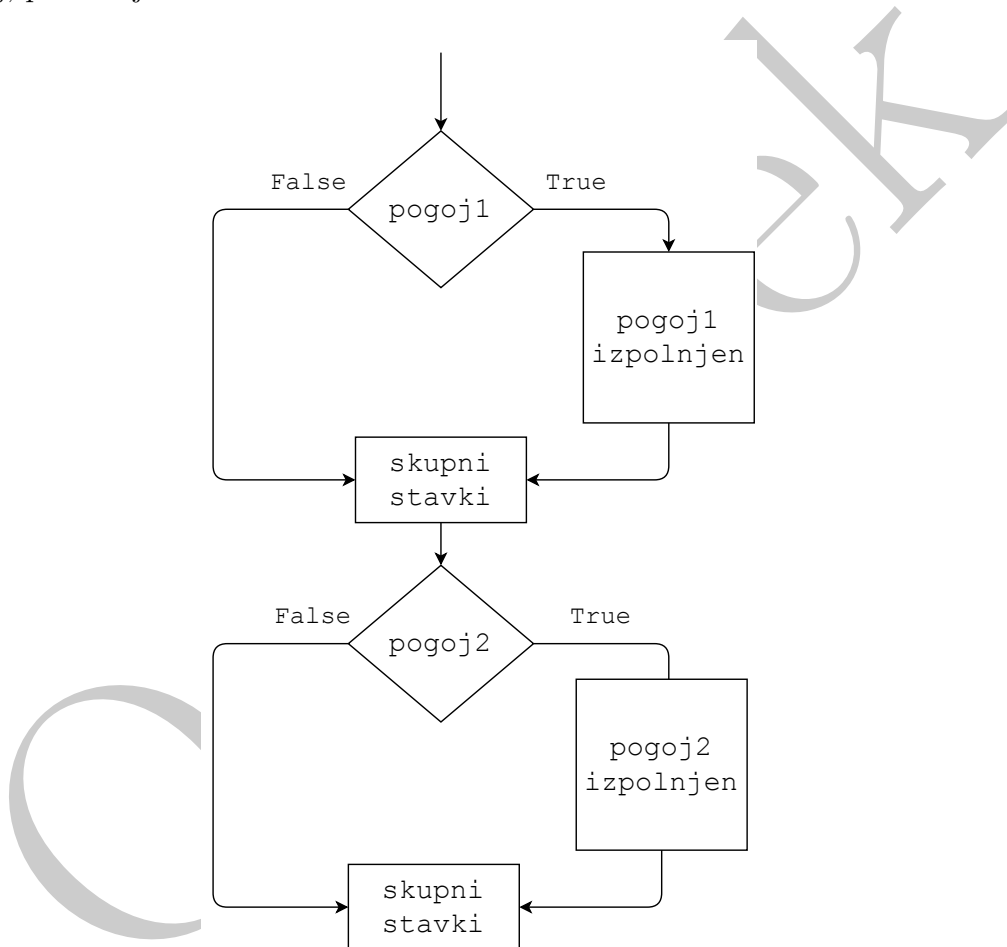
```
1 teza = float(input("Vpisi svojo tezo [kg]: "))
2 visina = float(input("Vpisi svojo višino [m]: "))
3 itm = teza/visina**2
4 print("Tvoj ITM je", itm)
5 if ITM < 17.5:
6     print("Tvoja teza je premajhna")
7 if ITM > 25:
8     print("Tvoja teza je prevelika")
9 if 17.5 <= ITM and ITM <= 25:
10    print("Tvoja teza je ustrezna")
```

Programiranja se učimo v jeziku Python med drugim tudi zato, ker omogoča kar nekaj uporabnih funkcionalnosti, ki jih drugi jeziki ne podpirajo. Sestavljen pogoj `17.5 <= ITM and ITM <= 25` lahko v tem jeziku zapišemo tudi malo krajše, in sicer takole: `17.5 <= ITM <= 25`.



## 2.4 Veja else

Zgornji program je sicer pravilen, ni pa najlepši. V primeru, da je npr. izpolnjen prvi pogoj, tj.  $ITM < 17.5$ , ni nobene potrebe po tem, da preverjamo še izpolnjenost drugega in tretjega pogoja. To sicer v tem primeru ni narobe (lahko bi bilo), je pa nepotrebno in po eni strani naredi našo kodo manj pregledno, po drugi strani pa trati dragocen procesorski čas, saj preverja, če je določen pogoj izpolnjen, kljub temu, da vemo, da zagotovo ne more biti. Potek programa, ki smo ga napisali zgoraj, ponazarja slika 2.2.



**Slika 2.2** Izpolnjenost pogoja `pogoj2` se preverja ne glede na izpolnjenost pogoja `pogoj1`.

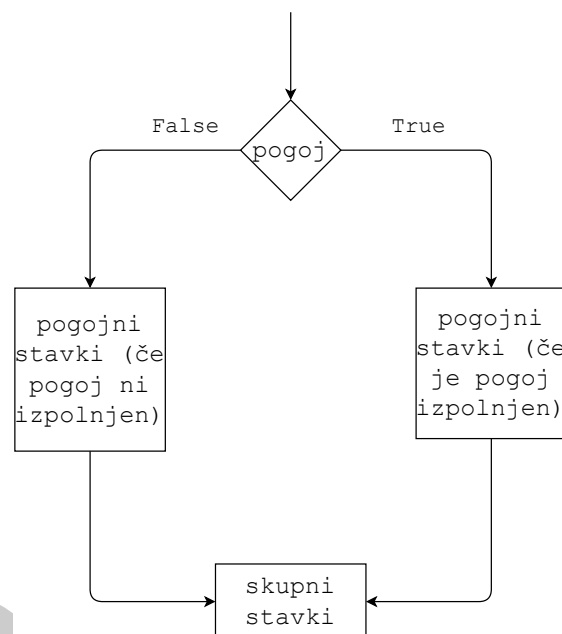
Do sedaj smo v primeru neizpolnjenosti pogoja vedno skočili na del `skupni stavki`, torej na del, ki se izvede neodvisno od izpolnjenosti pogoja. V splošnem pa stavek `if` omogoča, da del kode izvedemo samo takrat, ko pogoj **ni** izpolnjen. To kodo podamo v veji `else` stavka `if`:

`if` pogoj:

## 22      Poglavje 2      Pogojni stavek

```
# pogojni stavki
# ce je pogoj izpolnjen
...
else:
    # pogojni stavki
    # ce pogoj ni izpolnjen
    ...
# skupni stavki
...
```

Potek izvajanja zgornje kode prikazuje slika 2.3.



**Slika 2.3** Dopolnitev stavka `if` z vejo `else`. Veja `else` se izvede samo v primeru, ko pogoj ni izpolnjen.

Uporabimo zgornji stavek za poenostavljeno rešitev naloge z izpisovanjem podatkov o ITM.

**Zgled 5** *Napiši program, ki od uporabnika prebere telesno težo in višino in izpiše uporabnikov indeks telesne mase (ITM). Poleg tega program uporabniku pove, če je njegova teža ustrezna ali ne.*

**Rešitev 5** *Tokrat bomo preverjali le pogoj o ustreznosti uporabnikove teže.*

- 1 `teza = float(input("Vpisi svojo tezo [kg]: "))`
- 2 `visina = float(input("Vpisi svojo višino [m]: "))`

```

3 itm = teza/visina**2
4 print("Tvoj ITM je", itm)
5 if 17.5 <= ITM and ITM <= 25:
6     print("Tvoja teza je ustrezna")
7 else:
8     print("Tvoja teza ni ustrezna")

```

## 2.5 Veja `elif` in gnezdenje stavkov `if`

Uporabnik zdaj ve, če je njegova teža ustrezna. Če njegova teža ni ustrezna, informacije o tem ali je pretežak ali prelahek nima (verjetno se mu to sicer dozdeva). Zgornji primer bi torej radi dopolnili tako, da znotraj veje `else` izvedemo dodatno primerjanje, na podlagi katerega bomo lahko ugotovili ali je ITM prevelik ali premajhen.

To lahko naredimo na dva načina. Elegantnejši način je uporaba stavka `elif`, ki omogoča preverjanje dodatnega pogoja znotraj veje `else`. Celoten stavek `if` z vejo `elif` zapišemo takole:

```

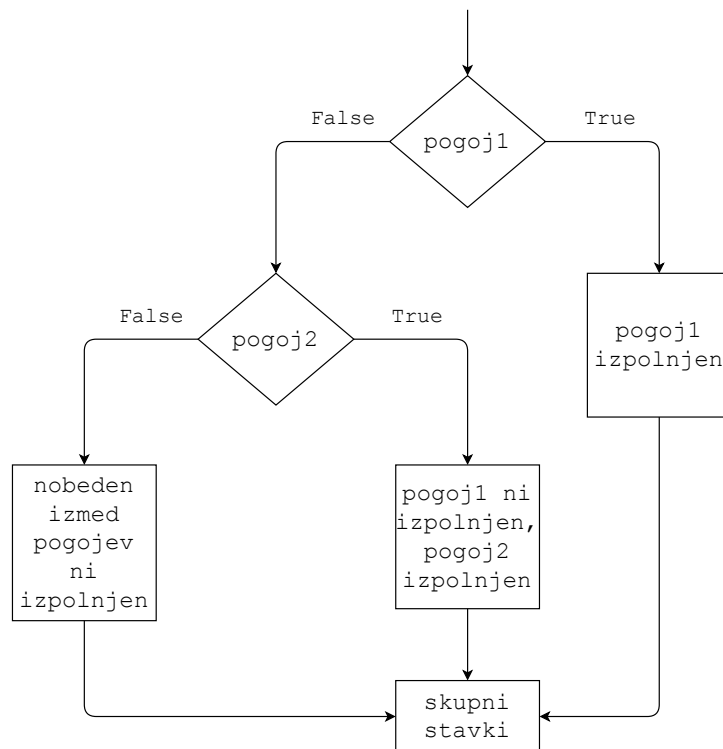
if pogoj1:
    # pogojni stavki
    # pogoj1 izpolnjen
    ...
elif pogoj2:
    # pogojni stavki
    # pogoj1 ni izpolnjen
    # pogoj2 izpolnjen
    ...
else:
    # nobeden izmed pogojev
    # ni izpolnjen
# skupni stavki
...

```

V tem primeru se izpolnjenost pogoja `pogoj2` preverja samo, če pogoj `pogoj1` ni izpolnjen, veja `else` pa se izvede samo v primeru, ko ni bil izpolnjen nobeden izmed prejšnjih pogojev. Potek programa prikazuje slika 2.4.

Zdaj lahko končno podamo lepšo rešitev zgleda z izpisovanjem podatkov o ITM.

**Zgled 6** *Napiši program, ki od uporabnika prebere telesno težo in višino in izpiše uporabnikov indeks telesne mase (ITM). Poleg tega program uporabniku pove, v katero skupino spada.*



**Slika 2.4** Dopolnitev stavka `if` z vejama `elif` in `else`. Veja `elif` se izvede samo v primeru, ko pogoj `pogoj1` ni izpolnjen, pogoj `pogoj2` pa je.

**Rešitev 6** Najprej bomo preverili enega izmed pogojev, npr. če je ITM manjši od 17.5. V primeru, da je izpolnjen, izpišemo, da je teža premajhna. V veji `elif` lahko preverimo naslednji pogoj, npr. če je ITM večji od 25. V primeru, da je izpolnjen, izpišemo, da je teža prevelika. Če ni izpolnjen nobeden izmed obeh pogojev, lahko izpišemo, da je teža ustrezna.

```

1 teza = float(input("Vpisi svojo tezo [kg]: "))
2 visina = float(input("Vpisi svojo visino [m]: "))
3 itm = teza/visina**2
4 print("Tvoj ITM je", itm)
5 if ITM < 17.5:
6     print("Tvoja teza je premajhna")
7 elif ITM > 25:
8     print("Tvoja teza je prevelika")
9 else:
10    print("Tvoja teza je ustrezna")
  
```

Drug pristop k reševanju enakega problema je uporaba dodatnega stavka `if` znotraj

veje `else`. Temu rečemo tudi *gnezdenje* ali *ugnezdeni stavek if*. Kodo bi napisali takole:

```
if pogoj1:
    # pogojni stavki
    # pogoj1 izpolnjen
    ...
else
    if pogoj2:
        # tukaj so uporabljeni dvojni zamiki
        # pogojni stavki
        # pogoj1 ni izpolnjen
        # pogoj2 izpolnjen
        ...
    else:
        # tukaj so uporabljeni dvojni zamiki
        # nobeden izmed pogojev
        # ni izpolnjen
# skupni stavki
...
```

Začetek vgenzdenega stavka `if` je zamaknjen enkrat, s čimer povemo, da naj se izvede samo v primeru, ko pogoj `pogoj1` ni izpolnjen. Vsebino ugnezdenega stavka moramo zamakniti dvakrat. Izvedba zgornje kode bo enaka kot v primeru z uporabo veje `elif` in jo prikazuje slika 2.4.

Uporabimo ugnezden stavek `if` še pri reševanju naše naloge.

**Zgled 7** *Napiši program, ki od uporabnika prebere telesno težo in višino in izpiše uporabnikov indeks telesne mase (ITM). Poleg tega program uporabniku pove, v katero skupino spada. Uporabi ugnezden stavek `if`*

**Rešitev 7** *Potek programa bo podoben kot prej za razliko od gnezdenje stavka `if`.*

```
1 teza = float(input("Vpisi svojo tezo [kg]: "))
2 visina = float(input("Vpisi svojo višino [m]: "))
3 itm = teza/visina**2
4 print("Tvoj ITM je", itm)
5 if ITM < 17.5:
6     print("Tvoja teza je premajhna")
7 else
8     if ITM > 25:
9         print("Tvoja teza je prevelika")
10    else:
11        print("Tvoja teza je ustrezna")
```

Osnutek

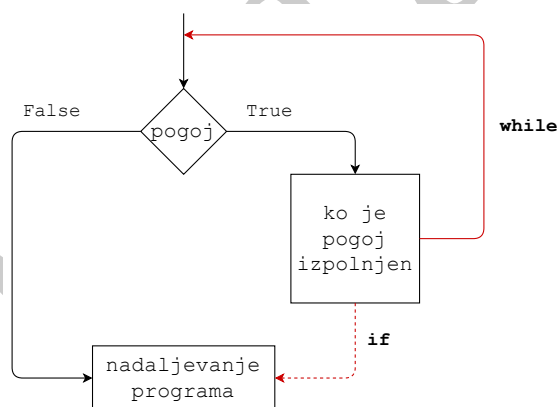
## 3 Zanka while

### 3.1 Kaj so zanke?

Z uporabo stavka `if` lahko torej izbrane stavke izvedemo samo v primeru, ko je nek pogoj izpolnjen. Včasih pa bi želeli izbrane stavke izvajati vse **dokler** (angl. *while*) je nek pogoj izpolnjen. To omogočajo **zanke**. V sledečem poglavju si bomo podrobneje pogledali zanko `while`.

### 3.2 Zanka while

Razliko med izvedbo pogojnega stavka `if` in zanko `while` prikazuje slika 3.1.



**Slika 3.1** Razlika med izvedbo pogojnega stavka `if` (črtkana linija rdeče barve) in zanko `while` (polna linija rdeče barve).

Po izvedbi pogojnega dela stavka `if` se izvajanje programa nadaljuje v delu, ki sledi pogojnemu stavku. Po drugi strani se po izvedbi pogojnega dela zanke `while`, recimo tem stavkom raje *telo zanke*, izpolnjenost pogoja v *glavi zanke* ponovno preveri. Telo (vsebina) zanke se bo torej izvajalo vse dokler bo pogoj izpolnjen. Zanko `while` lahko zapišemo takole:

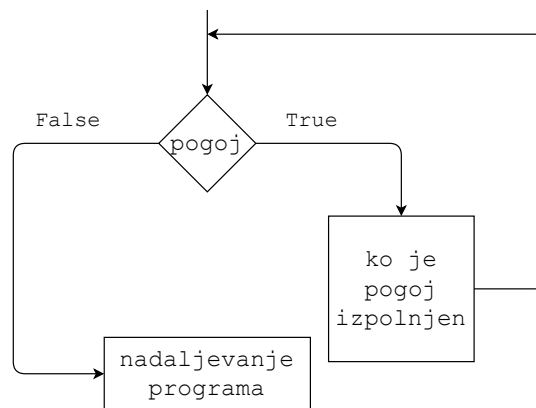
```
while pogoj: # glava zanke
```

```

    # telo zanke
    ...
# nadaljevanje programa
...

```

Zapis zanke `while` je torej zelo podoben zapisu stavka `if`. Glavi zanke sledi telo zanke oziroma njena vsebina, ki se izvaja vse dokler je pogoj izpolnjen. Enemu obhodu zanke pravimo tudi *iteracija* zanke. Pogoje za izvedbo nove iteracije zanke lahko zapisujemo na popolnoma enak način kot pri stavku `if`. Prav tako kot pri stavku `if` vsebino zanke definiramo tako, da stavke znotraj telesa zanke zamikamo. Izvajanje zanke `while` ponazarja diagram poteka na sliki 3.2.



Slika 3.2 Potek izvajanja zanke `while`.

Zanke torej lahko uporabljamo takrat, ko želimo nekaj ponavljati, dokler je določen pogoj izpolnjen. Npr., dokler uporabnik ne poda veljavnega vnosa, dokler ni števec dosegel določene vrednosti ali pa dokler sta števili različni. Končno lahko rešimo program, ki išče največji skupni delitelj dveh števil.

**Zgled 8** *Napiši program, ki od uporabnika prebere dve celi števili in poišče največji skupni delitelj teh dveh števil z uporabo Evklidovega algoritma.*

**Rešitev 8** Program bo manjše število odšteval od večjega, dokler sta števili različni. Uporabili bomo torej zanko `while` (dokler sta števili različni). Znotraj zanke bomo uporabili še stavek `if`, s pomočjo katerega bomo ugotovili, katero število je manjše.

```

1 st1 = int(input("Vnesi prvo število: "))
2 st2 = int(input("Vnesi drugo število: "))
3
4 while st1 != st2: # dokler sta števili različni
5     if st1 < st2: # drugo število je večje
6         st2 = st2 - st1

```



```

7     else: # prvo stevilo je veje
8         st1 = st1 - st2
9
10 # konec vsebine zanke
11 print(st1) # stevili sta tu enaki, zato je vseeno katero izpisem

```

*Opomba: vsebina stavka if je zamaknjena dvakrat, saj je zapisana tako znotraj stavka if kot tudi znotraj zanke while!*

### 3.3 Štetje z zanko while

Zanko while bi lahko uporabili tudi za štetje. Za ta namen je sicer boljša zanka for, ki jo bomo spoznali malo kasneje. Poglejmo si zgled.

**Zgled 9** *Napiši program, ki šteje od 0 do števila, ki ga je vnesel uporabnik. Vsa števila naj program tudi izpiše*

**Rešitev 9** Število, do katerega smo prešteli, si bomo morali nekam zabeležiti, npr. v spremenljivko z imenom *i*. Šteti bomo glede na navodila začeli s številom 0. Torej bomo spremenljivko *i* na začetku postavili na 0. Končali bomo s številom, ki ga je vnesel uporabnik, recimo *n*. Pogoji za štetje naprej bo torej  $i \leq n$ . Znotraj zanke while bomo trenutno število (*i*) izpisali, poleg tega moramo trenutno število tudi povečati, saj bo sicer pogoj za vedno izpolnjen.

```

1 n = int(input("Vpisi stevilo do katerega želiš šteti: "))
2 i = 0 # stevec s katerim bomo steli
3 while i <= n: # smo že presteli?
4     print(i)
5     i = i + 1 # povečanje stevca za 1

```

### 3.4 Stavek +=

Znotraj zanke smo števec povečali za 1 z izvedbo prireditvenega stavka

```
i = i + 1 # povečanje stevca za 1
```

Ker je tak način povečevanja vrednosti zelo pogost, v jeziku Python obstaja bližnjica

```
i += 1 # povečanje stevca za 1
```

Bistvo zgornjega stavka je, da izvedemo aritmetično operacijo seštevanja in rezultat priredimo spremenljivki, nad katero smo operacijo izvedli. Na podoben način lahko operator prirejanja = kombiniramo z drugimi aritmetičnimi operatorji in števili:

```

>>> x = 10
>>> x += 1 # povecaj za 1
>>> x
11
>>> x -= 2 # zmanjsaj za 2
>>> x
9
>>> x *= 5 # pomnozi s 5
>>> x
45
>>> x /= 9 # deli z 9
>>> x
5.0
>>> x **= 2 # potenciraj na 2
>>> x
25.0

```

### 3.5 Mrtva zanka

Kaj pa bi se zgodilo, če bi števec v prejšnjem zgledu znotraj zanke pozabili povečati? Spremenljivka *i* (oziroma števec) bi ostala na vrednosti 0 ne glede na to koliko iteracij zanke bi se izvedlo. To pomeni, da bi bil pogoj za vedno izpolnjen (**True**). Kdaj bi se taka zanka končala? Ker je pogoj vedno resničen, se taka zanka nikoli ne konča in tak program je potrebno končati na silo (v okolju Python je temu namenjena kombinacija tipk **ctrl + c**). Na take stvari moramo torej pri programiranju z zanko **while** paziti. Zanki, ki se nikoli ne konča, pravimo *mrtva zanka* (angl. *deadlock*).

### 3.6 Stavek break

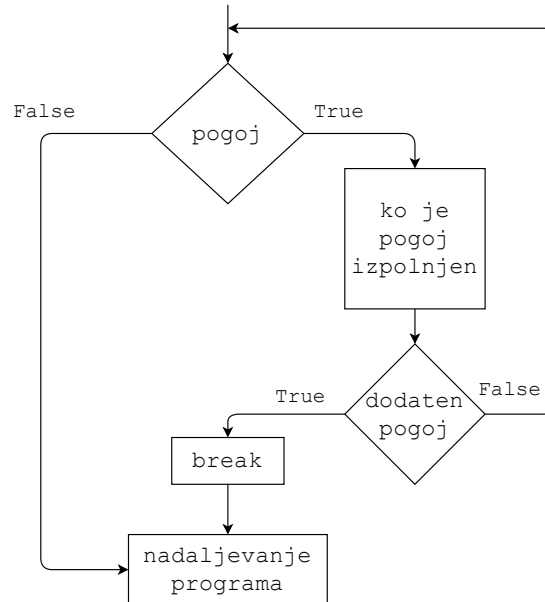
Zanko pa lahko prekinemo tudi drugače kot z neizpolnjenostjo pogoja v glavi zanke. Uporabimo lahko namreč stavek **break**, ki prekine izvajanje zanke brez preverjanja pogoja v glavi zanke. Primer uporabe stavka **break** ponazarja spodnja koda

```

while pogoj:
    # telo zanke
    ...
    if dodaten_pogoj:
        break # prekine izvajanje zanke
# nadaljevanje programa
...

```

Izvedbo primera prikazuje slika 3.3.



Slika 3.3 Primer uporabe stavka `break` znotraj zanke `while`.

Stavek `break` ponavadi uporabljamo v kombinaciji z dodatnim pogojem. V primeru, da je slednji izpolnjen, se izvajanje zanke prekine predčasno. Poglejmo si še en zgled, na katerem bomo potrenirali uporabo stavka `break`.

**Zgled 10** *Napiši program, ki od uporabnika prebere dve celi števili in izpiše, če sta števili tuji. Števili sta tuji, če nimata nobenega skupnega delitelja, ki je večji od 1.*

**Rešitev 10** *Nalogo bi lahko rešili tako, da bi poiskali največji skupni delitelj podanih števil (naj bosta to števili `st1` in `st2`) in pogledali, če je ta večji od 1. Tokrat se bomo rešitve lotili na malo drugačen način. Preverili bomo, če med kandidati za skupne delitelje obstaja kakšno število, ki deli obe števili, pri čemer bodo kandidati v razponu od števila 2 do manjšega števila, torej do vrednosti `min(st1, st2)`. V primeru, da med kandidati najdemo eno število, ki deli obe podani števili (ostanek po deljenju posameznega števila s kandidatom je enak 0 (`st % delitelj == 0`)), lahko takoj izpišemo, da števili nista tuji.*

```

1 st1 = int(input("Vnesi prvo število: "))
2 st2 = int(input("Vnesi drugo število: "))
3
4 delitelj = 2 # zacetna vrednost kandidata
5
6 while delitelj <= min(st1, st2): # kandidat gre do manjšega

```

```

7      # ali kandidat deli obe stevili?
8      if st1 % delitelj == 0 and st2 % delitelj == 0:
9          print("Stevili_nista_tuji")
10     delitelj += 1

```

Rešitev je še nepopolna, saj izpis poda samo v primeru, ko števili nista tuji. Kako bi lahko program dopolnili, tako da bi izpis podal tudi v primeru, ko sta števili tuji. Tak izpis lahko podamo samo v primeru, ko smo pregledali vse kandidate in nismo našli nobenega, ki deli obe števili. Pomagamo si lahko s pomožno spremenljivko tipa `bool`, v katero bomo shranili informacijo o tem, ali smo že našli kašnega delitelja. Pri tem bomo na začetku predpostavljali, da delitelja ni. Če ga bomo našli, bomo predpostavko popravili. Na koncu bomo preverili, če smo kakšnega delitelja našli. Če bo odgovor ne (`nasli == False`), bomo izpisali, da sta si števili tuji.

```

1  st1 = int(input("Vnesi_prvo_stevilo:_"))
2  st2 = int(input("Vnesi_drugo_stevilo:_"))
3
4  delitelj = 2 # zacetna vrednost kandidata
5  # predpostavljamo, da skupnega delitelja se nismo nasli:
6  nasli = False
7
8  while delitelj <= min(st1, st2): # kandidat gre do manjšega
9      # ali kandidat deli obe stevili?
10     if st1 % delitelj == 0 and st2 % delitelj == 0:
11         print("Stevili_nista_tuji")
12         nasli = True # popravimo predpostavko
13     delitelj += 1
14
15 # ce do tu skupnega delitelja nismo nasli, potem ga ni
16 if nasli == False:
17     print("Stevili_sta_tuji")

```

Program sicer deluje pravilno, je pa njegov izpis moteč, v primeru, da najdemo več skupnih deliteljev dveh števil. Vsakič, ko najdemo skupnega delitelja, namreč izpišemo, da smo ga našli. Poleg tega bi lahko izvajanje zanke `while` prekinili takoj, ko smo našli enega skupnega delitelja, saj je to zadosten pogoj, da si števili nista tuji. Uporabimo lahko torej stavek `break`. Končna rešitev bo sledeča.

```

1  st1 = int(input("Vnesi_prvo_stevilo:_"))
2  st2 = int(input("Vnesi_drugo_stevilo:_"))
3
4  delitelj = 2 # zacetna vrednost kandidata
5  # predpostavljamo, da skupnega delitelja se nismo nasli:
6  nasli = False

```

```

7
8 while delitelj <= min(st1, st2): # kandidat gre do manjšega
9     # ali kandidat deli obe stevili?
10    if st1 % delitelj == 0 and st2 % delitelj == 0:
11        print("Stevili_nista_tuji")
12        nasli = True # popravimo predpostavko
13        break # lahko prenehamo z iskanjem
14    delitelj += 1
15
16 # ce do tu skupnega delitelja nismo nasli, potem ga ni
17 if nasli == False:
18    print("Stevili_sta_tuji")

```

### 3.7 Veja else

Ena izmed posebnosti jezika Python je tudi to, da lahko vejo **else** uporabljamo tudi v kombinaciji z zanko **while**. Takole:

```

while pogoj:
    # telo zanke
    ...
else:
    # ko pogoj ni vec izpolnjen
    ...
# nadaljevanje programa
...

```

Veja **else** se torej izvede, ko pogoj ni več izpolnjen. Vprašanje pa je ali se veja **else** izvede vsakič, ko se izvajanje zanke zaključí? Kakšna je razlika med stavki, ki sledijo zanki **while**, in stavki znotraj veje **else** zanke **while**?

Do bistvene razlike med vejo **else** in običajnimi stavki, ki sledijo zanki **while**, pride, kadar zanko prekinemo s stavkom **break**. V tem primeru namreč skočimo iz zanke, s čimer preskočimo tudi njeno **else** vejo. Slednja se izvede samo v primeru, ko smo zanko prekinili po *običajni* poti, tj. z neizpolnjenostjo pogoja v glavi zanke.

```

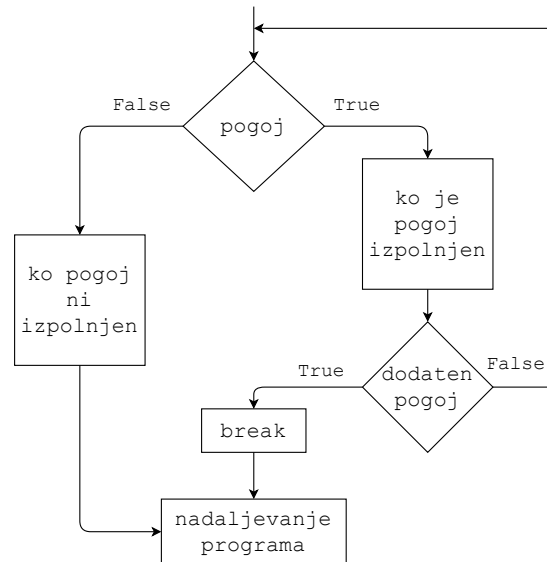
while pogoj:
    # telo zanke
    ...
    if dodaten_pogoj:
        break # prekini izvajanje zanke
else: # samo v primeru, ko zanka ni bila prekinjena z break
    # ko pogoj ni vec izpolnjen
    ...

```

*# nadaljevanje programa*

...

Delovanje zgornjega programa ponazarja slika 3.4. Z vejo `else` lahko določene



**Slika 3.4** Primer uporabe stavka `break` v kombinaciji z vejo `else` za zanko `while`.

stavke po zaključku zanke `while` izvedemo samo v primeru, ko zanka ni bila prekinjena s stavkom `break`. Povadimo na prejšnjem zgledu še tole.

**Zgled 11** *Napiši program, ki od uporabnika prebere dve celi števili in izpiše, če sta števili tuji. Števili sta tuji, če nimata nobenega skupnega delitelja, ki je večji od 1.*

**Rešitev 11** *Del programa, kjer izpisujemo, da si števili nista tuji, bo ostal bolj ali manj nespremenjen. Skrajšamo pa lahko tiste dela programa, ki jih potrebujemo za izpis, da sta si števili tuji. Števili sta si tuji, ko nismo našli nobenega skupnega delitelja. To se zgodi takrat, ko se je zanka `while` odvirtela do konca in nismo našli nobenega skupnega delitelja, torej je posledično nismo prekinili s stavkom `break`. Če zanko `while` dopolnimo z vejo `else`, se bo ta izvedla natanko takrat, ko zanka ni bila prekinjena s stavkom `break`, torej takrat, ko nismo našli nobenega skupnega delitelja. Znotraj veje `else` lahko torej zapišemo, da sta si števili tuji. Na tak način se lahko znebimo spremenljivke `nasli` in naredimo program krajši in bistveno bolj pregleden.*

```

1 st1 = int(input("Vnesi prvo število: "))
2 st2 = int(input("Vnesi drugo število: "))
3
4 delitelj = 2 # zacetna vrednost kandidata

```

```
5
6 while delitelj <= min(st1, st2): # kandidat gre do manjsega
7     # ali kandidat deli obe stevili?
8     if st1 % delitelj == 0 and st2 % delitelj == 0:
9         print("Stevili_nista_tuji")
10        break # lahko prenehamo z iskanjem
11    delitelj += 1
12 else: # ali se je zanka odvrtela do konca
13     # zanke nismo prekinili s stavkom break
14     print("Stevili_sta_tuji")
```

Osnutek



## 4 Sezname in metode

### 4.1 Sekvenčni podatkovni tipi

Podatkovni tipi, ki smo jih srečali do sedaj, so bili večinoma namenjeni temu, da vanje shranimo posamezen (en) podatek. V spremenljivko, ki je pripadala podatkovnemu tipu `int`, smo npr. lahko shranili eno število. V določenih primerih pa se srečamo z veliko količino med seboj podobnih podatkov, nad katerimi želimo izvajati enake ali podobne operacije. V praksi bi to lahko pomenilo, da izvajamo ponavljajoče meritve enake količine, npr. dolžine skoka smučarjev skakalcev. Kaj narediti v takem primeru? Na podlagi našega dosedanjega znanja bi lahko za vsakega skakalca definirali svojo spremenljivko, kar pa ne bi bila ravno najboljša rešitev. Prvi problem tega pristopa bi bil, da je lahko skakalcev zelo veliko. V primeru skakalcev bi bila stvar mogoče še lahko obvladljiva, kaj pa če npr. merimo prisotnost transkriptov genov v celici, ki ima par tisoč genov? Drugi problem je ta, da moramo vsako izmed spremenljivk obravnavati ločeno, kar nas bo pripeljalo do ogromne količine nepregledne *copy-paste* kode. Tretji problem tega pristopa je, da včasih ne vemo čisto točno koliko meritev bomo imeli in koliko spremenljivk bomo imeli (koliko bo skakalcev, koliko je genov v opazovani celici) in zato težko povemo koliko spremenljivk moramo posebej obravnavati. K sreči pa obstajajo t.i. *sekvenčni podatkovni tipi*, v katere lahko shranjujemo večjo količino podatkov oziroma več kot en podatek. Dodatna prednost sekvenčnih podatkovnih tipov je ta, da lahko podatke sproti dodajamo in ne potrebujemo vnaprej definirati števila podatkov, ki jih bomo na koncu imeli. Mimogrede, tudi nizi so sekvenčni podatkovni tipi, saj lahko vanje shranjujemo večjo količino podatkov, ki v tem primeru predstavljajo znake oziroma enočrkovne nize.

### 4.2 Kaj so sezname?

Drug predstavnik sekvenčnih podatkovnih tipov je seznam oziroma `list`. Za razliko od nizov lahko vanj shranimo poljubne podatke, kot so npr. števila, nizi in tudi drugi sezname. Dodatna prednost uporabe seznamov je ta, da lahko elemente v seznamu dodajamo sproti, zato dolžine seznama ni treba vnaprej definirati. Lahko torej začnemo s praznim seznamom in vsakič, ko dobimo podatek o novi meritvi,

tega v seznam dodamo.

Seznane definiramo z oglatimi oklepaji [ in ], znotraj katerih naštejemo elemente. Prazen seznam bi naredili takole

```
>>> prazen_seznam = []
```

Seznam, ki vsebuje približno naključne dolžine skokov smučarjev skakalcev pa takole

```
>>> dolzine = [121.4, 143.1, 105.2, 99.3]
```

V isti seznam bi lahko zapisali tudi različne podatkovne tipe, npr. 3 cela števila, 1 decimalko, 5 nizov itd., čeprav v praksi tega ne srečamo pogosto. Ponavadi v seznane shranjujemo podatke, ki pripadajo istemu podatkovnemu tipu, saj se ti podatke nanašajo na ponavljajoče izvajanje npr. določene meritve. Na koncu lahko zato z uporabo seznamov izvedemo določene statistike, npr. kdo je skočil najdlje, kakšna je povprečna dolžina skoka, koliko ljudi je skočilo itd.

### 4.3 Indeksiranje seznamov

Seznami so urejeni. To pomeni, da je vrstni red, v katerem naštejemo elemente seznama, pomemben. Vsak element v seznamu ima namreč svoj *indeks*. Pri tem se indeksiranje začne z najmanjšim pozitivnim številom, ki v računalništvu ni 1, ampak 0. Indeksi bodo torej šli od števila 0 do dolžine seznama – 1. V primeru zgoraj definirane seznama *dolzine* gredo torej indeksi od 0 do 3, saj seznam vsebuje 4 elemente:

<b>indeksi</b>	0	1	2	3	
<b>dolzine</b>	=	[121.4,	143.1,	105.2,	99.3]

Do elementa na določenem indeksu lahko pridemo z indeksiranjem, ki ga izvedemo tako, da za imenom spremenljivke indeks zapišemo v oglatih oklepajih:

```
ime_seznama[indeks]
```

Do dolžine skoka 0-tega skakalca bi torej prišli takole:

```
>>> dolzine[0]
121.4
```

Kaj pa do zadnjega skakalca? Do dolžine seznama lahko pridemo preko vgrajene funkcije *len*:

```
>>> len(dolzine)
4
```

Funkcijo lahko torej uporabimo pri indeksiranju, kadar ne vemo točno, koliko elementov ima seznam. Do zadnjega elementa torej pridemo takole:

```
>>> dolzine[len(dolzine)-1]
99.3
```

Zakaj moramo od dolžine seznama odšteti 1? Ker smo začeli šteti s številom 0, bo zadnji indeks enak dolžini seznama – 1. Kaj pa če vseeno poskusimo indeksirati po indeksu, ki ga v seznamu ni? V tem primeru seveda dobimo napako:

```
>>> dolzine[len(dolzine)]
Traceback (most recent call last):
  File "<pyshell#16>", line 1, in <module>
    dolzine[len(dolzine)]
IndexError: list index out of range
```

Kot smo do zdaj že večkrat videli ima Python veliko lepih lastnosti. Ena izmed njih je tudi ta, da lahko uporabljamo negativno indeksiranje, pri čemer indeks -1 predstavlja zadnji element, indeks -2 predzadnji in tako naprej. Dolžine skokov imajo torej tudi negativne indekse:

indeksi	-4	-3	-2	-1	
dolzine	=	[121.4,	143.1,	105.2,	99.3]

Prednost takega načina indeksiranja je v tem, da lahko na zelo enostaven način pridemo do zadnjega elementa seznama (brez funkcije `len`):

```
>>> dolzine[-1]
99.3
```

Mimogrede, podobno kot lahko indeksiramo elemente seznamov, lahko indeksiramo tudi elemente nizov. Prav tako lahko dolžino niza preverimo s funkcijo `len`.

```
>>> niz = "banana"
>>> niz[0]
"b"
>>> niz[-1]
"a"
>>> len(niz)
6
```

## 4.4 Operatorji nad seznamami

Nad seznamami lahko uporabimo različne operatorje, ki smo jih do zdaj uporabljali že npr. nad nizi. Nize smo npr. lahko med seboj seštevali (temu smo sicer rekli konkatencija oziroma lepljenje). Med seboj lahko seštevamo tudi sezname:

```
>>> [1,2,3] + [4,5,6]
[1,2,3,4,5,6]
```

Ne moremo pa seznamom prišteti nečesa, kar ni seznam, npr. števila:

```
>>> [1,2,3]+4
Traceback (most recent call last):
```

```
File "<pyshell#20>", line 1, in <module>
    [1,2,3]+4
TypeError: can only concatenate list (not "int") to list
```

Lahko pa sezname množimo s celimi števili:

```
>>> [1,2,3]*4
[1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]
```

S čim drugim jih ni smiselno množiti, zato Python tega ne podpira:

```
>>> [1,2,3]*[4,5,6]
Traceback (most recent call last):
  File "<pyshell#22>", line 1, in <module>
    [1,2,3]*[4,5,6]
TypeError: can't multiply sequence by non-int of type 'list'
```

Nad seznamami lahko uporabimo tudi operatorja vsebovanosti `in` in `not in`, ki vrneta `True` ali `False` v odvisnosti od tega ali je nekaj v seznamu vsebovano ali ne:

```
>>> 1 in [1,2,3]
True
>>> 1 not in [1,2,3]
False
```

Sezname lahko primerjamo z drugimi seznamami z uporabo primerjalnih operatorjev. Takole preverjamo enakost oziroma neenakost dveh seznamov:

```
>>> [1,2,3] == [1,2,3]
True
>>> [1,2,3] != [1,2,3]
False
```

Lahko tudi ugotavljamo, če je prvi seznam manjši od drugega (besedico manjši bi lahko zamenjali tudi z večji, manjši ali enak ter večji ali enak):

```
>>> [1,2,3] < [1,2,3,4]
True
>>> [1,3,3] < [1,2,3]
False
```

Primerjalni operatorji nad seznamami delujejo podobno kot nad nizi, in sicer se gre za leksikografsko primerjanje. Leksikografsko primerjanje je npr. uporabljeno pri sortiranju besed v slovarju. Deluje tako, da med seboj primerjamo istoležne elemente seznamov, dokler ne pridemo do neenakosti oziroma do konca enega izmed obeh seznamov. V zgornjem primeru smo prišli do konca prvega seznama. Ker je nekaj kar ne obstaja načeloma manjše kot nekaj kar obstaja, je Python vrnil, da je prvi seznam manjši od drugega. V drugem primeru se je primerjanje ustavilo pri

elementih na indeksu 1, saj sta elementa na tem indeksu različna. Ker 3 ni manjše od 2, je Python ugotovil, da prvi seznam ni manjši od drugega in vrnil rezultat `False`.

## 4.5 Spreminjanje in brisanje elementov seznama

Videli smo že, da lahko do elementov seznama dostopamo preko indeksiranja. Preko indeksiranja pa lahko vrednosti v seznamih tudi spreminjamo. Kako? Tako, da indeksiranje seznama dopolnimo s prireditvenim stavkom:

```
seznam[indeks] = nova_vrednost
```

Tudi brisanje elementov iz seznama lahko izvajamo s pomočjo indeksiranja, le da tokrat pred indeksiranjem uporabimo besedico **del**:

```
del seznam[indeks]
```

## 4.6 Vgrajene funkcije nad seznamami

Srečali smo že funkcijo `len`, s pomočjo katere lahko ugotovimo kakšna je dolžina seznama. Nad seznamami pogosto uporabljamo še druge vgrajene funkcije, izmed katerih so pogosto uporabljene `min`, `max` in `sum`.

Funkcija `min` vrne najmanjši, funkcija `max` pa največji element v seznamu glede na relacijo `<`. Zdaj lahko končno ugotovimo kakšna je bila dolžina najdaljšega skoka:

```
>>> max(dolzine)
143.1
```

Izračunamo lahko tudi povprečno dožino skoka

```
>>> sum(dolzine)/len(dolzine)
117.25
```

Nad seznamami lahko uporabimo še druge vgrajene funkcije. Nekatere izmed njih bomo srečali kasneje, druge pa boste zagotovo našli, če se bo takšna potreba pokazala.

## 4.7 Metode

Nad seznamami lahko torej uporabljamo vgrajene funkcije, ki so pač v Pythonu na voljo. Te funkcije lahko sicer uporabimo na poljubnem podatku, ki ni nujno seznam. Obstaja poseben nabor funkcij, ki pa jih lahko uporabljamo samo nad seznamami. Tem funkcijam pravimo *metode seznamov*. V splošnem se izraz *metode* uporablja za posebno družino funkcij, ki pripadajo določenemu *objektu*. Kaj je objekt zaenkrat ne bomo podrobneje razlagali. Lahko pa povemo, da so seznamski objekti (pravzaprav

je skoraj vse v Pythonu objekt). Kakorkoli že metode so tiste funkcije, ki pripadajo določenemu objektu. Do posamezne metode seznama lahko pridemo s spodnjim klicem:

```
ime_seznama.ime_metode(argumenti)
```

Klic metode je torej podoben klicu običajne funkcije, le da moramo pred imenom metode podati ime objekta, preko katerega oziroma nad katerim metodo kličemo, imeni pa ločimo s piko (.).

Če delamo v okolju IDLE ali v kakšnem še pametnejšem okolju, nam bo to po izpisu imena seznama in pikice podalo nabor metod, ki jih imamo na razpolago. Ko v okolju IDLE npr. napišemo

```
>>> dolzine.
```

se po nekaj sekundah pojavijo imena metod, ki jih lahko nad seznamom uporabimo: `append`, `copy`, `clear` itd.

Metode torej razširjajo vgrajene funkcije okolja Python in so vezane na točko določen podatkovni tip. Če bi npr. enako stvar kot zgoraj poskusili z nizom, bi dobili drug seznam metod, ki jih lahko poženemo nad nizom. Metodam kot argument za razliko od vgrajenih funkcij ni potrebno podati seznama (ali pa niza) nad katerim jih želimo izvesti, saj smo seznam (ali pa niz) podali že pred piko – že s samim klicom smo povedali nad čim želimo metodo pognati. Metode vseeno velikokrat vsebujejo določene argumente, ki pač določijo kaj in kako naj metoda nad objektom naredi.

Prav tako kot obstaja kar veliko vgrajenih funkcij, obstaja tudi veliko metod nad seznamami. Natančneje si bomo v nadaljevanju tega poglavja pogledali tiste, ki jih uporabljamo pogosteje.

## 4.8 Dodajanje elementov

Dodajanje elementov v seznam je pogosta operacija, zato jo lahko izvedemo na več načinov. Enega smo pravzaprav že srečali, saj lahko za dodajanje elementov v seznam uporabimo kar operator `+`, ki omogoča lepljenje seznamov. Če želimo element seznamu dodati, bomo obstoječemu seznamu prišteli seznam, ki vsebuje ta element. Takole:

```
seznam = seznam + [element]
```

oziroma malo lepše:

```
seznam += [element]
```

Tole dvojce sicer ni popolnoma enako, ampak zaenkrat recimo, da je bolje uporabiti spodnjo različico.

Elemente lahko v sezname dodajamo tudi preko metode `append` in metode `extend`. Obe metodi bosta dodajali na koncu seznama. Razlika je v tem, da v primeru

metode `append` dodajamo en element, zato ta metoda kot argument prejme element, ki ga bomo v seznam dodali. Dodajanje bi torej izvedli takole:

```
seznam.append(element)
```

Metoda sicer ne bo ničesar vrnila, bo pa naš seznam spremenila. Primer uporabe je sledeč:

```
>>> seznam = [1,2,3]
>>> seznam.append(4)
>>> seznam
[1,2,3,4]
```

Podobno lahko uporabimo metodo `extend`, ki v seznam dodaja drug seznam. Kot argument moramo torej metodi `extend` podati seznam, ki ga želimo v obstoječ seznam dodati. Takole:

```
seznam.extend(seznam2)
```

Oziroma na prejšnjem zgledu takole:

```
>>> seznam = [1,2,3]
>>> seznam.extend([4])
>>> seznam
[1,2,3,4]
```

Povadimo dodajanje elementov v seznam na praktičnem zgledu.

**Zgled 12** *Napiši program, ki ga bo lahko uporabil sodnik smučarskih skokov. Program naj sodnika sprašuje po dolžini skoka. V primeru, da sodnik vnese število večje od 0, naj program to število doda v seznam in sodnika vpraša po novi dolžini. Če sodnik vpiše številko 0, naj program izpiše dolžino najdaljšega skoka in povprečno dolžino skoka.*

**Rešitev 12** *Sodnikova števila lahko beremo preko funkcije `input`, katere rezultat moramo pretvoriti v podatkovni tip `float`, saj so dolžine decimalna števila. Beremo dokler sodnik ne vnese števila 0, medtem pa dolžine dodajamo v seznam. Na koncu izračunamo povprečno dolžino skoka, poleg tega pa izpišemo tudi najdaljši skok. Program je sledeč:*

```
1 dolzine = [] # na zacetku ni nobene dolzine
2 d = float(input("Vpisi_dolzino:_")) # prvo branje
3 while d > 0: # dokler je dolzina veljavna
4     dolzine.append(d) # dodaj dolzino
5     d = float(input("Vpisi_dolzino:_")) # ponovno branje
6 print("Najdaljsi_skok:", max(dolzine))
7 print("Povprecna_dolzina:", sum(dolzine)/len(dolzine))
```

Prednost zgornjega programa je v tem, da deluje ne glede na to koliko skokov je v seznamu. Vse dokler sodnik ne vnese kakšne neumnosti.

## 4.9 Branje seznamov iz ukazne vrstice

Včasih pa bi si želeli celoten seznam prebrati z enim samim uporabnikovim vnosom. Torej bomo spet uporabili funkcijo `input`. Spomnimo se, da funkcija `input` uporabnikov vnos vedno zapiše v niz oziroma v podatkovni tip `str`, ne glede na to kaj je uporabnik vnesel. Tak niz smo v prejšnjih primerih s funkcijo `int` pretvorili v celo število ali pa s funkcijo `float` v decimalno, če smo želeli podan vnos v nadaljevanju obravnavati kot število. Kaj pa če bi želeli niz, ki ga je vnesel uporabnik, pretvoriti v seznam? Na prvo žogo bi lahko rekli, da pač uporabimo funkcijo `list`. Poskusimo:

```
>>> seznam = list(input("Vnesi seznam: "))
Vnesi seznam: [1,2,3]
>>> seznam
['[', '1', ',', '2', ',', '3', ']']
```

To torej ni tisto, kar smo želeli. Dobili smo namreč seznam vseh znakov, ki v podanem nizu nastopajo (vključno z vejicami in oklepaji).

Kaj bi pravzaprav radi dosegli? To, da se niz, ki ga uporabnik poda funkciji `input` obravnava na enak način, kot če bi isti niz vpisal v ukazno vrstico. Temu je namenjena funkcija `eval`, ki kot argument sprejme niz in ga izvede kot Python kodo. Poskusimo še to:

```
>>> seznam = eval(input("Vnesi seznam: "))
Vnesi seznam: [1,2,3]
>>> seznam
[1,2,3]
```

V tem primeru stvar deluje, kot bi si želeli. Povadimo še na zgledu.

**Zgled 13** *Napiši program, ki ga bo lahko uporabil sodnik smučarskih skokov. Programu naj sodnik vnese seznam dolžin smučarskih skokov, program pa naj izpiše dolžino najdaljšega skoka in povprečno dolžino skoka.*

**Rešitev 13** *Rešitev bo podobna kot prej, le da tokrat ne potrebujemo zanke.*

```
1 dolzine = eval(input("Vnesi dolzine: "))
2 print("Najdaljsi skok:", max(dolzine))
3 print("Povprecna dolzina:", sum(dolzine)/len(dolzine))
```

*Slabost programa je ta, da mora sodnik zdaj vse dolžine vnesti naenkrat.*



Uporaba funkcije `eval` je sicer lahko v določenih primerih nevarna (če imamo zlobne uporabnike), saj bo slepo izvedla kodo, ki jo bo uporabnik preko vnosa podal.

## 4.10 Sortiranje seznamov

Zaenkrat znamo določiti najdaljši skok, ne znamo pa določiti najdaljših treh skokov. Najdaljše tri skoke bi lahko poiskali tako, da seznam uredimo (posortiramo), tako da vsebuje skoke od najdaljšega do najkrajšega. Potem izpišemo skoke na indeksih 0, 1 in 2.

Sortiranje seznamov lahko izvedemo z metodo `sort`:

```
>>> dolzine.sort()
>>> dolzine
[99.3, 105.2, 121.4, 143.1]
```

Metoda `sort` torej sortira seznam, nad katerim smo jo poklicali, in ničesar ne vrača. Opazimo tudi, da je seznam sortirala od najmanjše vrednosti do največje. Najboljše skoke bi torej lahko izpisali tako, da bi izpisali zadnje tri dolžine iz seznama. Lahko pa seznam sortiramo v obratnem vrstnem redu, tako da metodi `sort` nastavimo opcijski (izbirni) argument `reverse` na vrednost `True`. Do dokumentacije metode `sort` lahko pridemo preko funkcije `help`:

```
>>> help(list.sort)
Help on method_descriptor:

sort(self, /, *, key=None, reverse=False)
    Stable sort *IN PLACE*.
```

Dokumentacija ni nič kaj preveč izčrpna, vidimo pa lahko, da metoda `sort` sprejema tudi dva opcijska argumenta, in sicer `key`, ki je privzeto enak vrednosti `None` in `reverse`, ki je privzeto enak vrednosti `False`. Opcijski argumenti so tisti argumenti, ki imajo nastavljene privzete vrednosti. Če ob klicu ne specificiramo drugačnih vrednosti, bodo pač uporabljene privzete vrednosti. Privzete vrednosti pa lahko *povozimo*, tako da specificiramo drugačne vrednosti. Vrstni red urejanja bi lahko spremenili tako, da bi argument `reverse` nastavili na vrednost `True`. V našem primeru takole:

```
>>> dolzine.sort(reverse=True)
>>> dolzine
[143.1, 121.4, 105.2, 99.3]]
```

Rešimo zdaj celoten zgled od začetka do konca.

**Zgled 14** *Napiši program, ki ga bo lahko uporabil sodnik smučarskih skokov. Programu naj sodnik poda seznam dolžin smučarskih skokov, program pa naj izpiše najdaljše tri skoke.*

**Rešitev 14** *Zdaj bo branju seznama sledilo sortiranje in izpis zmagovalnih dolžin.*

```
1 dolzine = eval(input("Vnesi dolzine: "))
2 dolzine.sort(reverse=True)
3 print("1. mesto", dolzine[0])
4 print("2. mesto", dolzine[1])
5 print("3. mesto", dolzine[2])
```

Prej smo videli, da metoda `sort` sprejema tudi izbirni argument `key`, s katerim lahko določimo, preko katerega naj funkcija sortira. Če bi želeli npr. sortirati seznam po absolutnih vrednostih, bi argumentu `key` priredili funkcijo `abs`. Takole:

```
>>> seznam = [-100, 10, -1, -5, 50]
>>> seznam.sort(key=abs)
>>> seznam
[-1, -5, 10, 50, -100]
```

Seznami (in še kaj drugega) pa bi lahko sortirali tudi preko vgrajene funkcije `sorted`. Ta funkcija deluje na enak način kot metoda `sort`, le da podanega seznama ne sortira, ampak vrne sortiran seznam. Poglejmo si na zgledu:

```
>>> seznam = [-100, 10, -1, -5, 50]
>>> # funkciji kot argument podamo tisto kar želimo posortirati
>>> sorted(seznam)
[-1, -5, 10, 50, -100]
>>> seznam # podan seznam je ostal nespremenjen
[-100, 10, -1, -5, 50]
```

Funkcija torej vrne sortiran seznam, izhodiščni seznam pa je ostal nespremenjen. Kako bi dosegli, da se ime spremenljivke, preko katerega smo funkcijo poklicali, spremeni, tako da vsebuje sortiran seznam? Tako, da bi rezultat funkcije `sorted` priredili spremenljivki:

```
>>> seznam = [-100, 10, -1, -5, 50]
>>> seznam = sorted(seznam)
>>> seznam
[-1, -5, 10, 50, -100]
```

## 4.11 Seznami seznamov

Vemo že veliko več kot prej, še vedno pa ne vemo kdo je skočil največ in komu moramo podeliti medalje. Poleg dolžin bi si namreč v ta namen morali beležiti tudi

imena tekmovalcev skakalcev. To lahko rešimo tako, da imamo pač dva seznama, tj. seznam dolžin in seznam tekmovalcev. Na istoležnem indeksu imamo v obeh seznamih podatke o istem skakalcu. Takole:

```
>>> dolzine = [121.4, 143.1, 105.2, 99.3]
>>> skakalci = ["Andrej", "Bojan", "Cene", "Dejan"]
```

Andrej je torej skočil 121.4 metra, Dejan pa zgolj 99.3 metra. Zmagovalne tri skoke še vedno lahko dobimo tako, da sortiramo seznam dolžin:

```
>>> dolzine.sort(reverse=True)
>>> dolzine
[143.1, 121.4, 105.2, 99.3]
```

Do problema pa pride, ker zdaj ne vemo več kateremu imenu pripada posamezna dolžina, saj smo indekse v seznamu dolžin s sortiranjem premešali. Kaj lahko naredimo?

Alternativen pristop bi bil, da za beleženje podatkov o dolžinah in imenih uporabimo nov, vgnezden seznam. Torej naredimo seznam seznamov. Takole:

```
>>> skoki = [[121.4, "Andrej"],
              [143.1, "Bojan"],
              [105.2, "Cene"],
              [99.3, "Dejan"]]
```

Kasneje bomo za take primere sicer uporabljali malo drugačno strukturo, ampak zaenkrat te še ne poznamo. Naredili smo torej seznam seznamov. Kaj se nahaja v tem primeru na indeksu 0?

```
>>> skoki[0]
[121.4, "Andrej"]
```

Seznam, ki vsebuje podatke o ničtem skakalcu. Kako pa bi prišlo do njegovega imena? Z uporabo verižnega indeksiranja oziroma tako, da po indeksiranju zunanjega seznama pač še enkrat indeksiramo notranji seznam:

```
>>> skoki[0][1]
"Andrej"
```

Kaj se bo zgodilo, če tak seznam sortiramo? Nad vgnezdenimi seznamami bo za sortiranje uporabljena relacija <, ki smo jo v tem poglavju v povezavi s primerjanjem seznamov že srečali. Rekli smo, da relacija manjše seznime med seboj primerja leksikografsko. Najprej primerja ničti element prvega seznama z ničtim drugega. Če sta enaka, primerja prvi element prvega seznama s prvim elementom drugega seznama in tako naprej. Če bomo torej v vgnezdene seznime na ničto mesto dali dolžine na prvo mesto pa imena, bo sortiranje izvedeno po dolžinah. Po imenih bo sortiranje potekalo samo v primeru, če bosta dolžini pri dveh podseznamih enaki. Poskusimo:

```
>>> skoki.sort(reverse=True)
>>> skoki
[[143.1, 'Bojan'], [121.4, 'Andrej'], [105.2, 'Cene'],
[99.3, 'Dejan']]
```

Ker smo zdaj sortirali dolžine skokov skupaj z imeni tekmovalcev, informacije o tem kdo je skočil koliko nismo izgubili in lahko izpišemo zmagovalce, ki se nahajajo v prvih treh podseznamih na indeksu 1:

```
>>> skoki[0,1]
'Bojan'
>>> skoki[1,1]
'Andrej'
>>> skoki[2,1]
'Cene'
```

To so zmagovalci. Zapišimo celoten zgled.

**Zgled 15** *Napiši program, ki ga bo lahko uporabil sodnik smučarskih skokov. Program naj sodnika sprašuje po dolžini skoka in imenu tekmovalca. V primeru, da sodnik za dolžino vnese število večje od 0, naj program dolžino in ime doda v seznam. Če sodnik vpiše številko 0, naj program izpiše zmagovalce in dolžine njihovih skokov.*

**Rešitev 15** *Ponovno bomo brali dolžino po dolžino, le da bomo tokrat v primeru, da bo vnesena dolžina večja kot 0, prebrali še ime tekmovalca. Potem bomo oboje dodali v seznam skokov. Pomembno je, da na ničto mesto v podsezname shranimo dolžino skoka, saj želimo podseznime sortirati po dolžini skokov. Na koncu skoke sortiramo in izpišemo zmagovalce in dolžine zmagovalnih skokov.*

```
1 skoki = [] # na začetku ni nobenega skoka
2 d = float(input("Vpisi_dolzino:_")) # prvo branje
3 while d > 0: # dokler je dolzina veljavna
4     ime = input("Vpisi_ime_tekmovalca:_") # branje imena
5     skoki.append([d,ime]) # dodajanje podseznama
6     d = float(input("Vpisi_dolzino:_")) # ponovno branje
7 skoki.sort(reverse=True)
8 print("1._mesto:", skoki[0][1], ",_dolzina_skoka:", skoki[0][0])
9 print("2._mesto:", skoki[1][1], ",_dolzina_skoka:", skoki[1][0])
10 print("3._mesto:", skoki[2][1], ",_dolzina_skoka:", skoki[2][0])
```

## 4.12 Generiranje seznamov s funkcijo range

Do zdaj smo sezname generirali oziroma dopolnjevali na podlagi vrednosti, ki jih je podal uporabnik. Taki sezname so lahko vsebovali poljubne elemente – pač tisto, kar je uporabnik vnesel.

V določenih primerih želimo imeti sezname s števili v podanem razponu. Praktično uporabo takih seznamov bomo podrobneje spoznali v naslednjem poglavju, zaenkrat pa si pogledjmo, kako jih lahko generiramo.

Generiranje seznamov v podanem razponu omogoča vgrajena funkcija `range`. Funkcijo `range` lahko uporabljamo na tri različne načine, in sicer preko podajanja sledečih argumentov:

- **start**: določa začetek seznama (celo število),
- **stop**: določa konec seznama (celo število),
- **step**: določa korak (celo število).

Pri prvem načinu uporabe funkciji `range` podamo zgolj argument `stop`. V tem primeru bomo dobili seznam vrednosti do argumenta `stop-1` s korakom 1. Poskusimo:

```
>>> razpon = range(10)
>>> razpon
range(0, 10)
```

Tale izpis je malo čuden. Poklicali smo funkcijo `range` in dobili `range`. Pogledjmo si kakšen je podatkovni tip rezultata:

```
>>> type(razpon)
<class 'range'>
```

Rezultat funkcije `range` torej pripada razredu `range`, ki nam vrednosti iz razpona vrača sproti, ko jih pač potrebujemo. Zakaj tako? Funkcija `range` je varčna in ponavadi ni nobene potrebe po tem, da bi morali celoten razpon ustvariti naenkrat, ampak naenkrat potrebujemo samo en element iz razpona. S tem, ko funkcija `range` ustvari objekt, ki nam po potrebi vrne željeni element, varčuje tako s procesorjevim časom (hitrost), saj je generiranje dolgih seznamov zamudno, kot tudi s pomnilniškim prostorom, saj dolgi sezname zasedejo veliko prostora. Še vedno pa lahko nad rezultatom funkcije `range` delamo podobne stvari, kot nad seznamami. Lahko jih npr. indeksiramo:

```
>>> razpon[2]
2
```

Ne moremo pa nad njimi klicati metod, ki so definirane nad običajnimi seznamami:

```
>>> razpon.sort()
Traceback (most recent call last):
  File "<pyshell#98>", line 1, in <module>
    razpon.sort()
AttributeError: 'range' object has no attribute 'sort'
```

Poleg tega funkcija `print`, kot smo videli že zgoraj, ne izpiše vrednosti elementov v razponu. Če bi želeli preko funkcije `range` dobiti običajen seznam, lahko uporabimo pretvorbo v seznam preko funkcije `list`:

```
>>> razpon = range(10)
>>> seznam = list(razpon)
>>> seznam
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> type(seznam)
<class 'list'>
```

Za to pa ponavadi ni potrebe. Primerjajmo ekonomičnost funkcije `range` in generiranja običajnega seznama. To lahko poskusimo tako, da s funkcijo `range` naredimo nek relativno velik razpon števil. Npr. od 0 do  $10^8-1$ :

```
>>> razpon = range(10**8)
```

Tudi če imate počasen računalnik, bo generiranje razpona narejeno v trenutku. Zdaj pa poskusimo iz tega razpona narediti klasičen seznam:

```
>>> seznam = list(razpon)
```

Če vam Python ni javil napake `MemoryError`, je tole verjetno nekaj časa trajalo. Če ni in vas nisem prepričal, poskusite stvar ponoviti z večjim številom, npr.  $10^{10}$ . Vrnimo se k osnovni uporabi funkcije `range`. Mogoče se sprašujete zakaj razpon ne vključuje vrednosti `stop`. Razlogov za to je več. Zaenkrat podajmo najbolj očitnega. Ker funkcija `range` začne šteti z vrednostjo 0 (in ne z 1), bo razpon, ki ga bo vračala, vseboval točno `stop` elementov. Če bi funkciji `range` za argument `stop` podali dolžino nekega seznama, bi razpon vseboval vse indekse tega seznama (ena izmed možnih uporab funkcije `range` se že počasi odkriva).

S funkcijo `range` lahko generiramo razpon elementov, ki se ne začne s številom 0. V tem primeru bomo funkciji poleg argumenta `stop` podali še argument `start`. Najprej seveda navedemo `start`, potem pa `stop`:

```
>>> range(start, stop)
```

Če bi npr. želeli generirati seznam v razponu od 5 do 10, bi napisali takole

```
>>> razpon = range(5, 10)
```

Poglejmo si seznam, ki ga s takim razponom dobimo:

```
>>> list(razpon)
[5, 6, 7, 8, 9]
```

Seznam torej vsebuje argument `start`, argumenta `stop` pa ne. Podobno kot prej. Razpon od 0 do 10 (brez števila 10) bi torej lahko dobili tudi takole:

```
>>> razpon = range(0, 10)
```

```
>>> list(razpon)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Do zdaj je bil korak med sosednjima elementoma v razponu vedno enak. To lahko spremenimo tako, da podamo še argument `step`. V tem primeru bomo funkcijo poklicali takole:

```
>>> range(start, stop, step)
```

Argument `step` je opsijski, njegova privzeta vrednost pa je 1. Lahko ga nastavimo na kaj drugega, npr. na 2. Če bi hoteli zgenerirati seznam lihih števil v razponu od 0 do 100, bi to lahko naredili takole:

```
>>> razpon = range(1, 101, 2)
>>> list(razpon)
[1, 3, 5, ..., 97, 99]
```

Zakaj smo argument `start` postavili na 1? Če bi začeli šteti z 0, bi dobili seznam sodih števil.

```
>>> razpon = range(0, 101, 2)
>>> list(razpon)
[0, 2, 4, ..., 98, 100]
```

Korak lahko nastavimo tudi na negativno vrednost:

```
>>> razpon = range(0, 101, -2)
>>> list(razpon)
[]
```

Tokrat smo dobili prazen seznam. Zakaj? Negativen korak pomeni, da štejemo navzdol. Torej mora imeti argument `start` večjo vrednost kot argument `stop`:

```
>>> razpon = range(101, 0, -2)
>>> list(razpon)
[101, 99, 97, ..., 3, 1]
```

Spet smo dobili seznam lihih števil. Zakaj? Zato ker smo začeli šteti z lihim številom. Poleg tega razpon zdaj vključuje število 101, ker je argument `start` v razponu vključen. Razpon sodih števil bi dobili takole

```
>>> razpon = range(100, 0, -2)
>>> list(razpon)
[100, 98, 96, ..., 4, 2]
```

Število 0 tokrat v razponu ni vključeno, ker razpon argumenta `stop` ne vključuje.

### 4.13 Rezine

V določenih primerih želimo namesto indeksiranja enega samega elementa izvesti indeksiranje razpona elementov v seznamu. Dobimo torej kos oziroma *rezino* (angl. *slice*) seznama. Razpon seznama podamo na zelo podoben način, kot smo ga uporabljali pri funkciji `range`, in sicer preko začetka (`start`) rezine, konca (`stop`) rezine in koraka *rezinjenja* (`step`).

Podamo lahko samo začetek rezine: `seznam[start:]`. V tem primeru bo rezina odrezana do konca seznama. Če bi npr. radi dobili vse elemente seznama od vključno petega indeksa naprej, bi napisali takole:

```
>>> seznam = list(range(10))
>>> seznam[5:]
[5, 6, 7, 8, 9]
```

Izhodiščni seznam je kot pri običajnem indeksiranju ostal nespremenjen:

```
>>> seznam
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Podamo lahko samo konec rezine: `seznam[:stop]`. V tem primeru se bo začela na začetku seznama in zaključila na indeksu `stop - 1`. Podobno kot pri funkciji `range` tudi pri rezinah `stop` ni vključen v razpon. Če bi npr. radi dobili vse elemente seznama od začetka do petega indeksa (pri tem peti indeks ne bo vključen), bi napisali takole:

```
>>> seznam = list(range(10))
>>> seznam[:5]
[0, 1, 2, 3, 4]
```

Z ne vključenostjo indeksa `stop` smo zopet prišli do točno `stop` vrednosti, saj se štetje začne z indeksom 0.

Pri rezinjenju lahko podajamo tudi zgolj korak: `seznam[:step]`. V tem primeru bo rezina odrezana od začetka do konca seznama, pri čemer bo uporabljen podan korak. Če bi npr. hoteli dobiti vsak drugi element seznama, bi napisali takole:

```
>>> seznam = list(range(10))
>>> seznam[:2]
[0, 2, 4, 6, 8]
```

Korak je lahko tudi negativen. Če bi kot korak npr. napisali vrednost `-1`, bi s tem seznam obrnili. S tem smo namreč povedali, da gremo čez cel seznam s korakom `-1`, torej od konca do začetka:

```
>>> seznam = list(range(10))
>>> seznam[::-1]
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```



Vse zgoraj naštetе kombinacije lahko seveda po mili volji kombiniramo. Če bi npr. hoteli vzeti vsak tretji element seznama v razponu od 2 do 9, bi napisali takole:

```
>>> seznam = list(range(10))
>>> seznam[2:9:3]
[2, 5, 8]
```

Če je korak negativen, moramo zopet paziti na to, da ima začetek (**start**) večjo vrednost od konca (**stop**)

```
>>> seznam = list(range(10))
>>> seznam[2:9:-1]
[]
>>> seznam[9:2:-1]
[9, 8, 7, 6, 5, 4, 3]
```

Če bi želeli iti od konca, do nekega indeksa proti začetku, bi to lahko podali kot `seznam[:stop:-1]`, npr.

```
>>> seznam = list(range(10))
>>> seznam[:1:-1]
[9, 8, 7, 6, 5, 4, 3, 2]
```

**stop** tudi tokrat ni vključen.

Povadimo rezine še na enem zgledu.

**Zgled 16** *Napiši program, ki ga bo lahko uporabil sodnik smučarskih skokov. Programu naj sodnik poda seznam dolžin smučarskih skokov, program pa naj izpiše najdaljše tri skoke.*

**Rešitev 16** *Podobno kot prej bomo seznam prebrali in uredili. Zmagovalce lahko zdaj izpišemo v eni vrstici. Tokrat bo program za razliko od prej deloval tudi v primeru, če bo sodnik vnesel manj kot 3 skoke. Rezine pač režejo dokler gre in primeru da je razpon preseže indekse seznama, napake ne javljajo.*

```
1 dolzine = eval(input("Vnesi dolzine: "))
2 dolzine.sort(reverse=True)
3 print(dolzine[:3])
```

## 4.14 Indeksiranje nizov

Kot smo že omenili lahko podobno kot sezname indeksiramo tudi nize. Prav tako lahko nad nizi izvajamo rezine. Povadimo najprej rezinjenje.

**Zgled 17** *Napiši program, ki bo od uporabnika prebral dve zaporedji baz (zapisani kot niza) in med njima na sredini izvedel križanje, tako da bo sestavil dve novi zaporedji baz in jih izpisal*

**Rešitev 17** Program bo torej od uporabnika prejel dva niza. Najprej bomo določili indeksa, kjer bomo križanje naredili. To bo na polovici posameznega zaporedja. Dolžino posameznega zaporedja bomo delili z 2, pri čemer bomo uporabili celoštevilsko deljenje (`//`), saj morajo biti indeksi cela števila. Potem bomo odrezali rezine in jih med seboj sestavili (z operatorjem lepljenja `+`) ter izpisali.

```

1  gen1 = input("Vpisi prvo zaporedje baz: ")
2  gen2 = input("Vpisi drugo zaporedje baz: ")
3
4  # kje prerežemo gen 1
5  i1 = len(gen1)//2 # celostevilsko deljenje z 2
6  # kje prerežemo gen 2
7  i2 = len(gen2)//2 # celostevilsko deljenje z 2
8
9  gen11 = gen1[:i1] # prva polovica gena 1
10 gen12 = gen1[i1:] # druga polovica gena 1
11 gen21 = gen2[:i2] # prva polovica gena 2
12 gen22 = gen2[i2:] # druga polovica gena 2
13
14 # lepljenje iz izpis
15 print(gen11 + gen22)
16 print(gen21 + gen12)

```

Iz zgornjega zgleda vidimo še eno prednost tega, da `stop` v rezino ni vključen. Če prvo rezino režemo do indeksa `stop`, drugo pa od istega indeksa naprej, bosta rezini nepresečni, kar pomeni, da element na indeksu `stop` ne bo podvojen. Povadimo zdaj še običajno indeksiranje, ki ga bomo potem pohitrili z rezinami.

**Zgled 18** Palindrom je niz, ki se na enak način bere naprej kot nazaj. Napiši program, ki od uporabnika prebere niz in izpiše, če podani niz je oziroma ni palindrom.

**Rešitev 18** Prva rešitev bo temeljila na zanki `while`, s katero se bomo sprehajali od začetka proti koncu niza. Zanko `while` lahko torej ponavljamo, dokler z nekim števcem (npr. `i`) ne preštejemo do konca niza. Začeli bomo pa seveda na začetku, torej pri vrednosti 0 (`i=0`). V zanki `while` bomo primerjali enakost znaka na indeksu `i` z znakom na indeksu `-i-1`. Če se bodo ti pari ujemali **povsod**, bomo lahko sklepali, da je niz palindrom. Takoj, ko bomo našli **en** primer, kjer se par ne ujema (protiprimer), pa bomo lahko sklepali, da niz ni palindrom.

```

1  niz = input("Vpisi niz: ")
2  i = 0 # zaceli bomo na zacetku niza
3  while i < len(niz): # do konca niza
4      if niz[i] != niz[-i-1]: # protiprimer

```

```

5         print("Niz_ni_palindrom")
6         break
7     i += 1 # gremo na naslednji par
8 else: # ce smo prisli do konca brez break-a
9     print("Niz_je_palindrom")

```

Program bi sicer lahko nekoliko pohitrili, saj se nam ni treba premikati do konca niza, ampak je dovolj, da končamo, ko števec *i* pride do polovice niza. Pogoji v zanki *while* bi torej lahko spremenili v *i < len(niz)//2*.

Do bistveno lepše rešitve pa pridemo, če uporabimo rezine. Niz je palindrom, če se bere naprej enako kot nazaj. Torej mora biti naprej prebran niz (*niz*) enak nazaj prebranemu niz (*niz[::-1]*). Program je torej sledeč:

```

1 niz = input("Vpisi niz: ")
2 if niz == niz[::-1]:
3     print("Niz_je_palindrom")
4 else:
5     print("Niz_ni_palindrom")

```

## 4.15 Sprehajanje čez sezname

Do zdaj smo se temu sicer izogibali, ampak pri delu s seznamami je ena izmed najpogostejših operacij sprehajanje nad seznamami. Kako narediti tak sprehod? Zgoraj smo se z zanko *while* sprehajali nad indeksi niza. Podoben sprehod bi lahko naredili tudi nad seznamami. Posamezen element seznama bi lahko izpisali npr. takole:

```

i = 0
while i < len(seznam):
    print(seznam[i])
    i += 1

```

Sprehajamo se torej po indeksih od začetka (0) do konca seznama (*len(seznam)-1*). Zgornja koda je sicer popolnoma pravilna, ni pa najlepša, saj je sprehajanje nad seznamami in seznamu podobnimi podatki v Pythonu namenjena posebna zanka, zanka *for*.

Osnutek

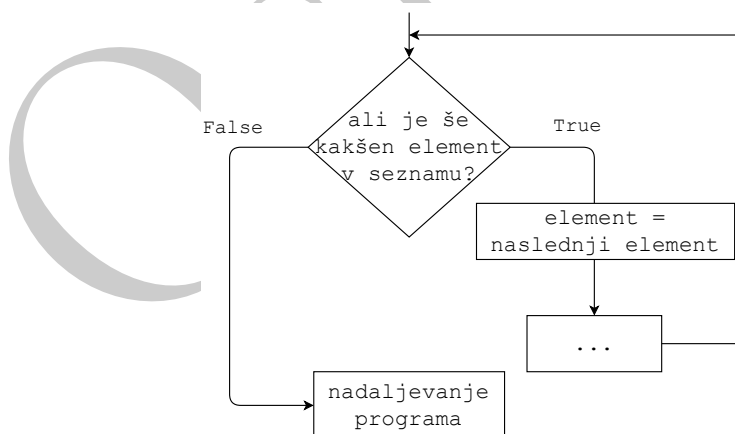
## 5 Zanka for

### 5.1 Sprehajanje čez seznane z zanko for

Kot smo videli na koncu prejšnjega poglavja, se lahko čez seznam (ali niz) sprehodimo z uporabo zanke `while`, pri čemer sprehod vršimo preko indeksov seznama (ali niza). Preko indeksov lahko potem posredno pridemo tudi do vrednosti elementov seznama (ali niza). Veliko bolj elegantno pa se čez seznam (ali pač niz) sprehodimo z uporabo zanke `for`:

```
for element in seznam:  
    # telo zanke  
    # spremenljivka element vsebuje trenutni element  
    ...  
# nadaljevanje programa  
...
```

Potek izvedbe osnovne oblike zanke `for` ponazarja slika 5.1 Izvajanje zanke torej



Slika 5.1 Potek izvedbe osnovne oblike zanke `for`.

ponavljamo vse dokler je v seznamu (ali nizu) še kakšen element, pri čemer se spremenljivka `element` pomika od začetka proti koncu seznama (ali niza). Če bi

npr. želeli izpisati vse elemente v seznamu, pri čemer bi vsak element izpisali v svoji vrstici, bi to lahko naredili takole:

```
>>> seznam = [1,2,3]
>>> for element in seznam:
    print(element)
1
2
3
```

Na podoben način bi se lahko sprehodili tudi čez niz:

```
>>> niz = "ABC"
>>> for znak in niz:
    print(znak)
A
B
C
```

V tem primeru se zanka for torej sprehaja čez znake niza. Povadimo sprehajanje še na zgledu.

**Zgled 19** *Napiši program, ki od uporabnika preko funkcije `eval` prebere seznam in izpiše najmanjši element seznama (brez uporabe funkcije `min`).*

**Rešitev 19** *Najmanjši element bomo našli tako, da se bomo z zanko `for` sprehodili čez seznam in si zapomnili element, ki je pač najmanjši. Kako pa vemo, da je nek element najmanjši, če ostalih še nismo pregledali? Težko. Vemo pa, če je nek element manjši od vseh elementov, ki smo jih pregledali preden smo do njega prišli. Nalogo lahko rešimo tako, da naredimo predpostavko, da je najmanjši ničti element v seznamu. Potem naredimo sprehod čez celoten seznam. Če bomo našli kakšen element, ki je manjši od trenutno najmanjšega, bomo trenutno najmanjši element postavili na tega, ki je očitno manjši. To bomo nadaljevali, dokler ne pridemo do konca seznama.*

```
1 seznam = eval(input("Vnesi seznam: "))
2
3 najmanjsi = seznam[0] # trenutno najmanjsi
4
5 for element in seznam: # sprehod cez elemente
6     if element < najmanjsi: # nasli manjsega?
7         najmanjsi = element # popravimo vrednost
8
9 print(najmanjsi)
```

*Program sicer deluje pravilno, ampak bi ga lahko še nekoliko optimizirali. Trenutno namreč ničti element v seznamu pregledamo dvakrat. Sprehoda z zanko `for` nam torej ne bi bilo potrebno delati čez cel seznam, ampak bi ga lahko naredili čez rezino seznama, ki se začne na indeksu 1. Torej bi zanko `for` lahko delali čez rezino `seznam[1:]`.*

## 5.2 Sprehajanje s funkcijo `range` in sprehajanje čez indekse

Z zanko `for` se lahko sprehajamo tudi čez seznane, ki jih generira funkcija `range`. Na tak način se lahko sprehajamo čez vrednosti elementov v določenem razponu. Vsa števila od 0 do vključno števila, ki ga je vnesel uporabnik, bi torej lahko izpisali takole:

```
n = int(input("Vnesi število: "))
for i in range(n+1):
    print(i)
```

Zanko `for` bi lahko na podoben način uporabili za sprehajanje po indeksih seznama. Vrednosti v seznamu na posameznih indeksih bi lahko izpisali takole:

```
for i in range(len(seznam)):
    print(i, seznam[i])
```

Tokrat funkciji `range` kot argument `stop` podamo dolžino seznama, kar pomeni, da bo funkcija zgenerirala razpon elementov v intervalu od 0 do `len(seznam)-1`, kar je ravno razpon indeksov seznama. Zato torej argument `stop` v interval ni vključen in zato funkcija `range` (tudi) deluje kakor deluje.

## 5.3 Sprehajanje čez elemente ali čez indekse?

Zgornji program bo poleg indeksa izpisal še vrednost elementa, ki se nahaja na posameznem indeksu. Ali bi lahko do indeksa elementov prišli tudi v primeru, ko se sprehajamo neposredno po elementih seznama? Težko. Zato v primeru, ko informacijo o indeksu potrebujemo, uporabljamo zanko čez indekse in ne čez elemente.

Poglejmo si spodnji primer, kjer rešitev zahteva izvedbo sprehoda čez indekse seznama.

**Zgled 20** *Napiši program, ki od uporabnika preko funkcije `eval` prebere seznam in izpiše najmanjši element seznama ter njegov indeks.*

**Rešitev 20** *Najmanjši element bomo našli na podoben način kot prej, le da si moramo tokrat zapomniti tudi njegov indeks. Ker preko direktnega sprehoda čez elemente seznama informacije o indeksih elementov nimamo, se bomo morali sprehoditi čez indekse seznama.*

```

1 seznam = eval(input("Vnesi seznam: "))
2
3 najmanjsi = seznam[0] # trenutno najmanjsi element
4 najmanjsi_i = 0 # zapomnimo si tudi njegov indeks
5
6 for i in range(len(seznam)): # sprehod cez indekse
7     element = seznam[i] # preko indeksa do elementa
8     if element < najmanjsi: # nasli manjsega?
9         najmanjsi = element # popravimo vrednost
10        najmanjsi_i = i # popravimo indeks
11
12 print(najmanjsi)
13 print(najmanjsi_i)

```

*Spet bi lahko pri sprehodu prvi element seznama izpustili, tako da bi se sprehajali čez razpon indeksov `range(1, len(seznam))`.*

Zgornja rešitev ima manjšo pomanjkljivost, in sicer ne upošteva, da se lahko enako majhen element v seznamu pojavi večkrat. V tem primeru vrne zgolj indeks njegove prve pojavitve. Naprednejšo rešitev prikazuje spodnji zgled.

**Zgled 21** *Napiši program, ki od uporabnika preko funkcije `eval` prebere seznam in izpiše najmanjši element seznama ter vse indekse njegove pojavitve.*

**Rešitev 21** *Rešitev bo podobna kot prej, le da si bomo indekse pojavitve najmanjšega elementa zabeležili kar v seznam. V primeru, da bomo našli manjši element od trenutnega, bomo naredili nov seznam, ki bo vseboval samo en indeks (trenutni indeks). V primeru, da bomo našli element, ki bo enak trenutno najmanjšemu, bomo v seznam indeksov dodali trenutni indeks. V zgornjih dveh rešitvah smo na koncu dodali boljšo rešitev, ki pri sprehodu izpusti ničti element seznama, saj smo tega upoštevali že pred zanko. Tokrat bo program brez te "optimizacije" deloval narobe. V primeru, da bo najmanjši element na ničtem mestu, bo njegov indeks v seznamu najmanjših indeksov namreč nastopal dvakrat.*

```

1 seznam = eval(input("Vnesi seznam: "))
2
3 najmanjsi = seznam[0] # trenutno najmanjsi element
4 najmanjsi_i = [0] # v seznam shranimo njegov indeks
5
6 for i in range(1, len(seznam)): # sprehod cez indekse
7     element = seznam[i] # preko indeksa do elementa
8     if element < najmanjsi: # nasli manjsega?
9         najmanjsi = element # popravimo vrednost

```



```

10     najmanjsi_i = [i] # resetiramo seznam indeksov
11     elif element == najmanjsi: # nasli enako majhnega
12         najmanjsi_i.append(i) # dodamo indeks
13
14 print(najmanjsi)
15 print(najmanjsi_i)

```

## 5.4 Spreminjanje elementov seznama z zanko for

Kaj pa v primeru da želimo seznam v zanki spremeniti, npr. da želimo vse negativne vrednosti seznama spremeniti v pozitivne (izračunati želimo absolutne vrednosti elementov seznama in seznam skladno s tem posodobiti). Poskusimo z običajnim sprehodom čez elemente seznama.

```

>>> seznam = [-1, 10, -5, 15, 0, -3]
>>> for element in seznam:
        element = abs(element)
        print(element)
1
10
5
15
0
3
>>> print(seznam)
[-1, 10, -5, 15, 0, -3]

```

Elemente smo torej uspešno postavili na njihove absolutne vrednosti, na kar nakazujejo izpisi, ki smo jih izvedli v telesu zanke. Kot pa vidimo iz izpisa, ki je sledil zanki, se seznam ni spremenil, saj še vedno vsebuje negativne elemente. Spreminjali smo torej vrednosti elementov, ne pa samega seznama. Če bi želeli spreminjati seznam, bi to lahko naredili preko indeksiranja:

```

>>> seznam = [-1, 10, -5, 15, 0, -3]
>>> for i in range(len(seznam)):
        seznam[i] = abs(seznam[i])
        print(seznam[i])
1
10
5
15
0
3

```

```
>>> print(seznam)
[1, 10, 5, 15, 0, 3]
```

Zdaj se je seznam seveda spremenil, saj smo absolutne vrednosti direktno prirejali seznamu na posameznem indeksu.

## 5.5 Zanka `for` ali zanka `while`?

Vidimo, da so naši programi z uporabo zanke `for` v določenih veliko krajši in lepši kot v primeru uporabe zanke `while`. Poleg tega nam pri uporabi zanke `for` ni potrebno skrbeti, da bo program za vedno obtičal v zanki (mrtva zanka). Zakaj bi torej sploh uporabljali zanko `while`? Izkaže se, da je zanka `while` bolj splošna kot zanka `for` in da lahko z njo rešimo določene probleme, ki jih z zanko `for` ne moremo. Kako bi npr. z zanko `for` od sodnika smučarskih skokov brali dolžine skokov, dokler sodnik ne vnese števila 0? Koliko ponovitev bi morali narediti? Kako bi z zanko `for` odštevali manjše število od večjega, dokler števili ne bi postali enaki? Odgovor je enostaven. Težko.

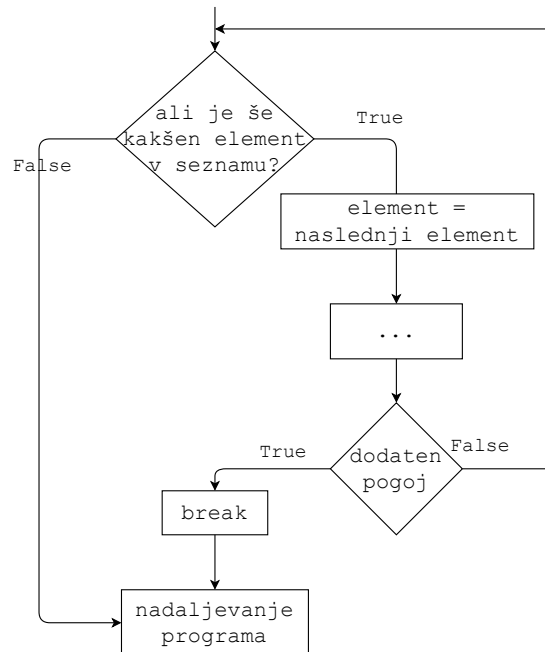
Vprašajmo se, kaj je skupnega primerom, kjer zanka `for` odpove. V obeh zgornjih primerih je število ponovitev, ki jih bo morala zanka narediti, vnaprej težko predvidljivo. V splošnem velja, da zanko `while` uporabljamo, kadar število ponovitev zanke težko podamo vnaprej, lahko pa oblikujemo pogoj, ki bo določil, do kdaj naj se zanka izvaja. V primeru, da je število ponovitev predvidljivo (npr. podan je razpon štetja ali pa seznam s fiksno dolžino, čez katerega se sprehajao) pa je kot nalašč zanka `for`.

## 5.6 Stavek `break`

V kombinaciji z zanko `for` lahko prav tako kot pri zanki `while` uporabljamo stavek `break`. Ta izvajanje zanke prekine, kljub temu, da ta še ni prišla do konca seznama (ali česa drugega). Primer uporabe stavka `break` znotraj zanke `while` ponazarja spodnja koda:

```
for element in seznam:
    # telo zanke
    ...
    if dodaten_pogoj:
        break # prekine izvajanje zanke
# nadaljevanje programa
...
```

Potek izvedbe kode iz primera prikazuje slika 5.2.



Slika 5.2 Potek izvedbe zanke for v kombinaciji s stavkom break.

## 5.7 Veja else

Podobno kot lahko vejo else kombiniramo z zanko while, jo lahko kombiniramo tudi z zanko for:

```

for element in seznam:
    # telo zanke
    ...
    if dodaten_pogoj:
        break # prekini izvajanje zanke
else: # samo v primeru, ko zanka ni bila prekinjena z break
    # konec seznama
    ...
# nadaljevanje programa
...
  
```

Veja else se bo kot pri zanki while izvedla samo v primeru, ko zanka ni bila prekinjena s stavkom break. Demonstrirajmo uporabo koncepta na zgledu s tujimi števili.

**Zgled 22** *Napiši program, ki od uporabnika prebere dve celi števili in izpiše, če sta števili tuji. Števili sta tuji, če nimata nobenega skupnega delitelja, ki je večji od 1.*

**Rešitev 22** Program bo strukturiran zelo podoben kot v primeru zanke *while*, le da bomo tokrat razpon števil, čez katera se sprehaja kandidat, ustvarili z uporabo funkcije *range*.

```

1  st1 = int(input("Vnesi prvo stevilo: "))
2  st2 = int(input("Vnesi drugo stevilo: "))
3
4  # sprehod od 2 do manjšega od obeh števil
5  # desni del intervala naj bo vključen, zato pristevamo 1
6  for delitelj in range(2, min(st1, st2)+1):
7      # ali delitelj deli obe števili?
8      if st1 % delitelj == 0 and st2 % delitelj == 0:
9          print("Števili nista tuji")
10         break # lahko prenehamo z iskanjem
11 else: # ali se je zanka odvirtela do konca
12     # zanke nismo prekinili s stavkom break
13     print("Števili sta tuji")

```

## 5.8 Gnezdenje zank

Podobno kot smo gnezдили stavke *if* lahko gnezidmo tudi zanke. To pomeni, da bomo zanko izvajali znotraj druge zanke. Primer gnezdenja zanke *for* prikazuje spodnji izsek kode:

```

>>> for i in range(5):
      for j in range(5):
          print(i,j)
0 0
0 1
0 2
0 3
0 4
1 0
1 1
...
3 3
3 4
4 0
4 1
4 2
4 3
4 4

```

Notraja zanka `for` torej za vsako iteracijo zunanje zanke izvede enako število ponovitev.

Potrenirajmo na zgledu.

**Zgled 23** *Napiši program, ki od uporabnika prebere celo število in izpiše poštrevanko števil od 1 do vključno podanega števila.*

**Rešitev 23** Števila od 1 do podanega števila  $n$  bomo najprej množili z 1, potem z 2, potem s 3 in tako naprej, dokler ne pridemo do števila  $n$ . To lahko enostavno rešimo z uporabo ugnezdene zanke.

```
n = int(input("Vnesi število: "))

for i in range(1, n+1): # zunanja zanka
    for j in range(1, n+1): # notranja zanka
        print(i*j) # izpis produkta
    print() #nova vrstica
```

V primeru, da uporabnik vpiše število 3, bo izpis sledeč:

```
1
2
3

2
4
6

3
6
9
```

V zgornjih primerih je bila notranja zanka neodvisna od tega, kako daleč se je že odvila zunanja zanka. Ponavadi pa temu ni tako. Primer ugnezdene zanke, pri kateri je razpon notranje zanke odvisen od števila izvedenih iteracij zunanje zanke, prikazuje spodnji izsek kode:

```
>>> for i in range(5):
        for j in range(i,5):
            print(i,j)

0 0
0 1
0 2
0 3
```

```

0 4
1 1
1 2
1 3
1 4
2 2
2 3
2 4
3 3
3 4
4 4

```

V prvi iteraciji zunanje zanke, se torej notranja zanka izvede petkrat, v drugi štirikrat, v tretji trikrat, v četrti dvakrat, v peti pa zgolj enkrat. Kasnejša kot je iteracija zunanje zanke, manjše je število ponovitev ugnezdene zanke.

Povadimo tako gnezdenje še na primeru s tujimi števili.

**Zgled 24** *Napiši program, ki od uporabnika prebere celo število in izpiše vsa števila, ki so podanemu številu tuja in so od njega manjša.*

**Rešitev 24** *Kandidati, ki jih moramo torej obravnavati, se gibljejo v razponu od števila 1 (ki je vsem številom tuje število) do števila  $n-1$ , pri čemer je  $n$  število, ki ga je vnesel uporabnik. Kako za posameznega kandidata preverimo, če je tuj številu  $n$ ? Podobno kot prej – tako da se sprehodimo od števila 2, do manjšega od obeh števil. Če smo našli kakšnega delitelja, si števili očitno nista tuji.*

```

1  n = int(input("Vnesi število: "))
2
3  for kandidat in range(1, n): # razpon cez kandidate
4      # kandidat ne sme imeti nobenega skupnega delitelja
5      # ugnezdimo kodo iz prejsnjih zgledov
6      st1 = n
7      st2 = kandidat
8
9      # sprehod od 2 do manjšega od obeh števil
10     # desni del intervala naj bo vključen, zato pristevamo 1
11     for delitelj in range(2, min(st1, st2)+1):
12         # ali delitelj deli obe števili?
13         if st1 % delitelj == 0 and st2 % delitelj == 0:
14             break # lahko prekinemo ugnezdeno zanko
15     else: # ali se je ugnezdena zanka odvirtela do konca
16         # ugnezdene zanke nismo prekinili s stavkom break
17         print(kandidat)

```

Ugnezdjena zanka je v tem primeru odvisna od tega kako daleč je naš program prišel z zunanjo zanko. Mimogrede, na podoben način bi lahko gnezdili tudi zanko `while`.

## 5.9 Izbirni argumenti funkcij in izbirni argumenti funkcije `print`

Tole sicer ni neposredno povezano z zanko `for`, bo pa služilo kot osnova za dopolnitev zgleда s poštevanko.

Povedali smo že, da funkcije sprejemajo argumente, ki jih ob klicu pač podamo. V določenih primerih pa imajo funkcije tudi t.i. *izbirne* ali *opcijske* argumente, za katere velja da imajo (pred)nastavljeno *privzeto* vrednost. V primeru, da vrednosti teh argumentov eksplicitno ne podamo, bodo ti nastavljeni na njihove privzete vrednosti. V primeru, da vrednosti tem argumentom podamo, bomo s tem *povozili* privzete vrednosti in uporabljene bodo podane (naše) vrednosti.

Poglejmo si dva izbirna argumenta funkcije `print` in primer njune uporabe. Do dokumentacije funkcije `print` lahko pridemo preko funkcije `help`:

```
>>> help(print)
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout,
          flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file: a file-like object (stream); defaults to the current
    sys.stdout.
    sep: string inserted between values, default a space.
    end: string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.
```

Zaenkrat nas bosta zanimala predvsem argumenta `sep` in `end`. Funkcija `print` deluje tako, da sprejme poljubno število števil, nizov in še česa drugega, to med seboj združi in izpiše na zaslon. Pri tem argument `sep` določa s čim naj podane številke, nize in še kaj drugega med seboj združi. Privzeto je ta argument postavljen na vrednost ' ', kar vidimo iz zgleда klica funkcije (`sep=' '`). To pomeni, da bo izpis narejen tako, da bodo med podanimi argumenti za izpis vstavljeni presledki. Povadimo:

```
>>> print(1,2,3) # privzeta vrednost argumenta
1 2 3
```

```
>>> print(1,2,3, sep='') # brez presledka
123
>>> print(1,2,3, sep='+++') # poljuben niz kot locilo
1+++2+++3
```

Izbirni argument `end` podaja niz, ki naj se vstavi na koncu izpisa. Privzeto je argument `end` nastavljen na znak `'\n'` (`end='\n'`), ki predstavlja znak za novo vrstico (angl. *line feed*). Tudi tega lahko postavimo na kakšno drugo vrednost. Povadimo nastavljanje opsijskih argumentov na zgledu v kombinaciji z ugnezdjeno zanko `for`.

**Zgled 25** *Napiši program, ki od uporabnika prebere celo število in izpiše poštrevanko števil od 1 do vključno podanega števila. Pri tem naj bo poštrevanka s posameznim številom podana v svoji vrstici, števila pa naj bodo ločena s presledki.*

**Rešitev 25** *Rešitev bo podobna kot prej, le da se tokrat ne bomo pomikali v novo vrstico po vsakem izpisu. To lahko naredimo tako, da opsijski argument `end` nastavimo na znak `' '`.*

```
n = int(input("Vnesi število:"))

for i in range(1, n+1): # zunanja zanka
    for j in range(1, n+1): # notranja zanka
        print(i*j, end = ' ') # izpis produkta brez nove vrstice
    print() #nova vrstica
```

V primeru, da uporabnik vpiše število 3, bo tokrat izpis sledeč:

```
1 2 3
2 4 6
3 6 9
```



## 6 Uporaba in pisanje funkcij

### 6.1 Kaj so funkcije in zakaj so uporabne?

Kot že vemo, funkcije predstavljajo del kode, ki jo lahko izvedemo tako, da funkcijo pač pokličemo.

Uporaba funkcij ima veliko prednosti. Govorili smo že o tem, da je glavno vodilo programiranja razdelitev problemov na obvladljive podprobleme. Določanje algoritma, ki ga potem samo še prenesemo v programsko kodo, je podobno določanju recepta, ki ga potem prenesemo v okusno jed. Prav tako, kot se moramo pri kuhanju zavedati sestavin, ki jih imamo na razpolago, se moramo tudi pri programiranju zavedati gradnikov programskega jezika, ki jih lahko pri pisanju algoritma uporabimo.

Funkcije nam omogočajo, da osnovne korake za reševanje programa vgradimo v enostavnejše funkcije, enostavnejše funkcije v kompleksnejše in tako naprej. Podobno, kot če bi pri peki torte lahko uporabili že vnaprej pripravljeno testo, preliv in kar se pač pri torti še uporabi, namesto da moramo torto sestaviti iz enostavnejših (nižjenivojskih) sestavin, kot so jajca, mleko in sladkor. Tako, kot bi lahko tudi pri peki seveda šli v drug ekstrem in se lotili reje kokoši, bi na veliko nižji nivo lahko šli tudi pri programiranju, ampak pustimo to za kdaj drugič. S pisanjem svojih funkcij se lahko torej najprej lotimo enostavnejših korakov, ki predstavljajo del rešitve izbranega problema. Potem lahko vmesne rešitve (velikokrat na enostaven način) združimo v končno rešitev. Če bi npr. želeli najti vsa praštevila v določenem razponu števil, bi lahko najprej napisali funkcijo, ki za podano število preveri, če je praštevilo. Vse kar bi morali narediti potem bi bil zgolj klic te funkcije za vsako število z intervala.

Zgled s praštevili pa nam je posredno razodel še eno veliko prednost uporabe funkcij. Isto kodo, tj. preverjanje ali je neko število praštevilo, smo poklicali večkrat, vsakič seveda z drugim argumentom, tj. številom, ki je kandidat za praštevilo. Funkcije nam torej omogočajo tudi to, da lahko isti kos kode večkrat pokličemo brez tega, da bi jo vključevali v zanke ali pa kopirali v vse dele programa, kjer jo potrebujemo. To kodo bi lahko delili tudi z drugimi programerji. Če smo npr. napisali zelo dobro funkcijo za iskanje praštevil in smo nad njo nadvse navdušeni, hkrati pa vemo, da bi bila lahko koristna tudi za druge iskalce praštevil, lahko funkcijo enostavno zapakiramo v t.i. modul, ki ga objavimo na internetu. In svet je postal še malenkost

boljši.

## 6.2 Kako definiramo funkcijo?

Vsaki funkciji, ki jo želimo v naših programih ponovno uporabiti, moramo dati seveda neko ime, preko katerega jo bomo lahko po potrebi poklicali. Skupaj s seznamom argumentov, ki jih bo naša funkcija sprejela, to podamo v definiciji funkcije. Definicijo funkcije začnemo z rezervirano besedo **def** in končamo z dvopičjem:

```
def ime_funkcije(argument_1, argument_2,..., argument_n):
```

Definiciji funkcije sledi njena vsebina. Stavke, ki so v funkciji vsebovani tudi tokrat določimo z zamikanjem na začetku vrstice (podobno kot pri pogojnemu stavku in zankah). Ko želimo Pythonu sporočiti, da koda ni več del funkcije, enostavno nehamo zamikati.

Spodnji primer predstavlja definicijo enostavne funkcije, ki sešteje vrednosti dveh spremenljivk (a in b) v novo spremenljivko (c) in rezultat seštevanja izpiše.

```
1 def sestej(a, b):
2     c = a + b
3     print(c)
4     # tale komentar je se del funkcije
5 # tale komentar ni vec del funkcije
```

Kaj pa se zgodi, ko program s tole definicijo poženemo. Navidez se ne zgodi nič, če pa v ukazno vrstico napišemo ime pravkar definirane funkcije, bi moral Python izpisati nekaj podobnega temu:

```
<function sestej at 0x000001C24E1481E0>
```

Kaj to pomeni? To pomeni, da se je v našem imenskem prostoru (pojem bomo razložili v kratkem) pojavilo ime **sestej**, ki ima v ozadju funkcijo, ta pa je shranjena nekje v pomnilniku (natančneje na pomnilniškem naslovu 0x000001C24E1481E0). Ko smo izvedli zgornjo kodo, smo torej dobili definicijo funkcije **sestej**, ki jo zdaj lahko pokličemo.

Do zdaj smo funkcije vedno klicali tako, da so imenu funkcije sledili oklepaji, znotraj katerih smo našli vrednosti argumentov, nad katerimi smo želeli funkcijo poklicati. In seveda je tako tudi v primeru funkcij, ki jih definiramo sami. Če bi torej želeli izpisati vsoto števil 5 in 7, bi lahko izvedli klic

```
>>>sestej(5,7)
12
```

Ko smo funkcijo definirali se torej koda znotraj funkcije sploh ni izvedla. Izvedla se je zgolj njena definicija, ki nam je njeno ime umestila v imenski prostor (podobno,

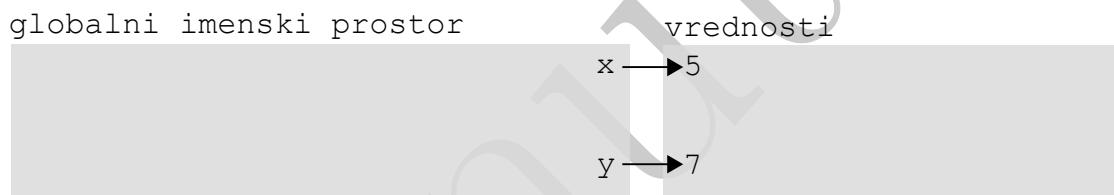
kot če smo nekemu imenu – spremenljivki, priredili neko vrednost). Dejanska izvedba stavkov znotraj funkcije pa se je izvršila šele, ko smo funkcijo poklicali. Mimogrede, če bi v funkciji imeli kakšno sintaktično napako, kot je npr. uporaba nedefinirane spremenljivke, bi jo Python našel šele ob klicu funkcije.

### 6.3 Globalni imenski prostor

Vsakič, ko v Pythonu definiramo novo spremenljivko, se ime, preko katerega bomo dostopali do vrednosti te spremenljivke shrani v t.i. *imenski prostor*. Podobno se zgodi ob definiciji funkcije, le da se v tem primeru za imenom funkcije skriva vsebina funkcije, ki se bo izvedla, ko jo bomo poklicali. Ko npr. definiramo spremenljivki `x` in `y` z uporabo kode

```
>>> x = 5
>>> y = 7
```

se v imenskem prostoru pojavita imeni `x` in `y`, za katerimi se skrivata podani vrednosti, kot prikazuje slika 6.1. Preko imen `x` in `y` lahko zdaj dostopamo do



**Slika 6.1** Imena v imenskem prostoru kažejo na konkretne vrednosti v pomnilniku.

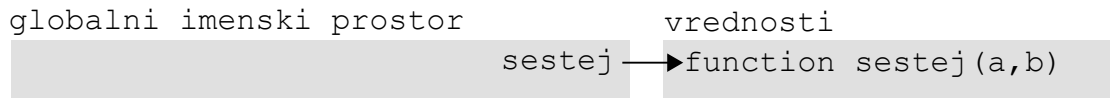
vrednosti, ki se skrivajo v ozadju, ne da bi se morali zavedati kje konkretno v pomnilniku so te vrednosti shranjene, kar nam bistveno olajša življenje.

Definicije novih imen, pa naj gre za imena spremenljivk ali funkcij, ki jih ustvarimo izven funkcij, se shranijo v t.i. *globalni imenski prostor*. Zato tem imenom pogosto rečemo kar globalna imena, spremenljivkam pa *globalne spremenljivke*. Če obstaja globalni imenski prostor pa bo verjetno obstajal tudi lokalni. Poglejmo si, kaj se zgodi, ko funkcijo pokličemo.

### 6.4 Kaj se zgodi ob klicu funkcije in lokalni imenski prostor

Kot smo že omenili, se ob definiciji funkcije v globalnem imenskem prostoru ustvari novo ime, ki je enako imenu funkcije. To kaže na samo funkcijo, tako da bomo lahko le-to kasneje preko imena tudi poklicali. Situacijo po definiciji funkcije `sestej` prikazuje slika 6.2.

Dopolnimo program, v katerem smo napisali funkcijo `sestej`, še z njenim klicem.



**Slika 6.2** Ob definiciji funkcije v imenskem prostoru dobimo novo ime, ki je enako imenu funkcije. Za tem imenom se skriva naša funkcija.

```

1 def sestej(a, b): # definicija funkcije
2     c = a + b
3     print(c)
4 x = 5
5 y = 7
6 sestej(x,y) # klic funkcije
  
```

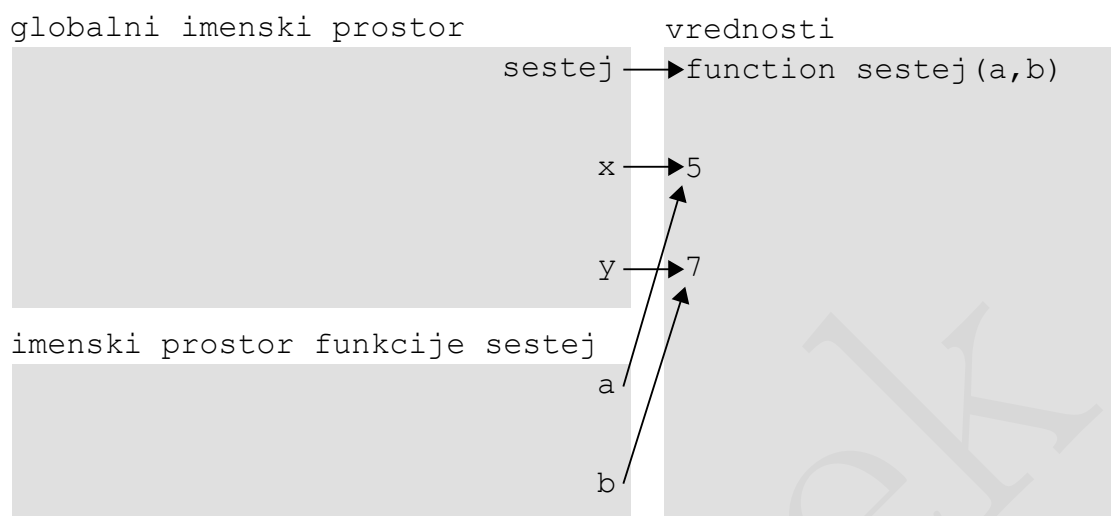
Vrstice programa od 1, 4 in 5 bi morale biti zdaj že popolnoma jasne. Kaj pa se zgodi, ko program pride do vrstice 6? Ustvari se lokalni imenski prostor funkcije **sestej**, znotraj katerega bo funkcija ustvarila svoje lokalne spremenljivke. V lokalnem imenskem se najprej ustvarita lokalni spremenljivki z imeni **a** in **b**, ki predstavljata *plitvi* kopiji spremenljivk **x** in **y**, tj. spremenljivk, s katerimi smo funkcijo poklicali. Enako posledico bi imela prireditvev

```

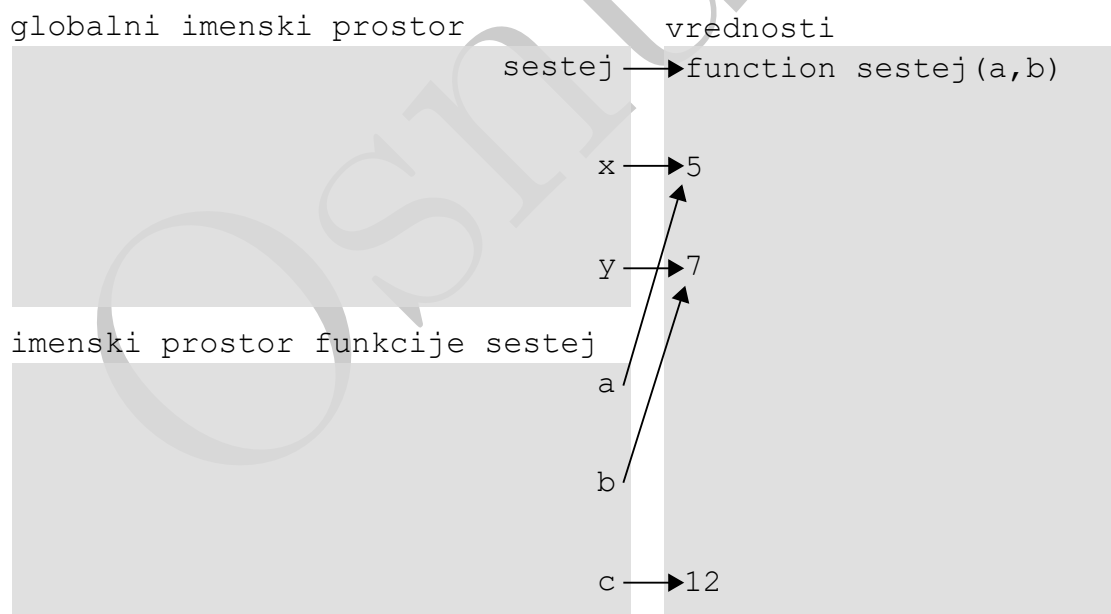
>>> a = x
>>> b = y
  
```

s to razliko, da bi se imeni **a** in **b** ustvarili v globalnem imenskem prostoru.

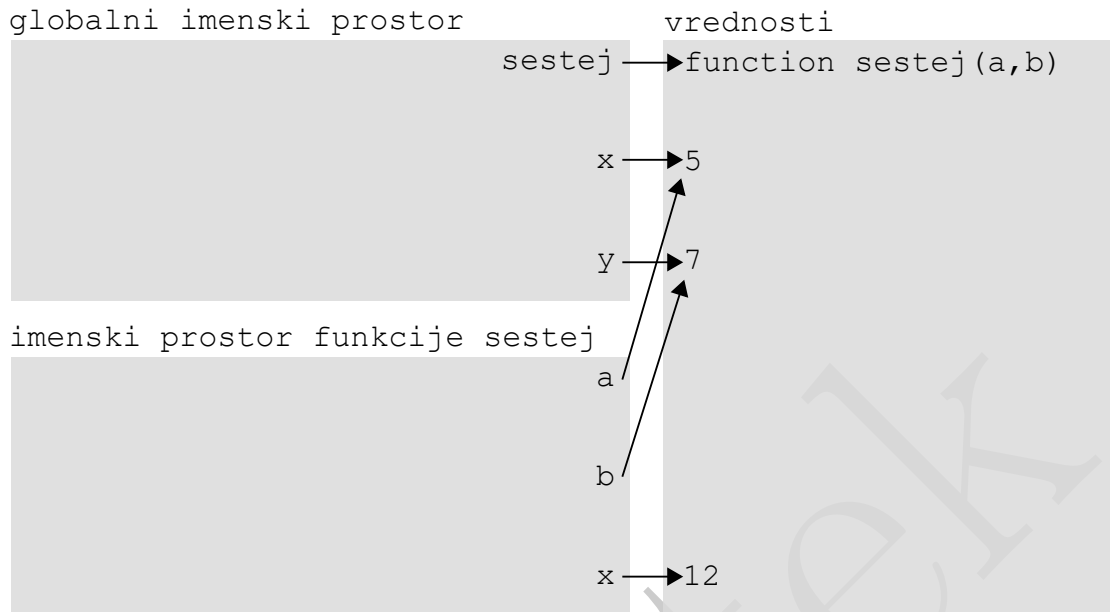
Plitva kopija pomeni, da vrednost, ki je shranjena v pomnilniku dobi dodatno ime, brez da bi se dejansko kopirala (to bi bila globoka kopija), s čimer smo s pomnilniškim prostorom veliko bolj varčni. O tem bomo še govorili, zaenkrat pa se vrnimo k naši funkciji in njenem lokalnem imenskem prostoru. Situacijo ob klicu funkcije prikazuje slika 6.3. Vsa imena, ki jih bomo v nadaljevanju definirali znotraj funkcije, bodo ustvarjena v lokalnem imenskem prostoru funkcije. Ko naš program na primer izvede vrstico 2 (ta se je ob definiciji funkcije preskočila in se izvede šele ob njenem klicu), bo prišlo do situacije, kot jo prikazuje slika 6.4 Kaj pa bi se zgodilo, če bi znotraj funkcije definirali ime, ki obstaja že v globalnem imenskem prostoru. Nič posebnega. Spremenljivka s tem imenom bi se ustvarila v lokalnem imenskem prostoru funkcije in to na globalno spremenljivko ne bi vplivalo. Zgodilo bi se nekaj takega kot prikazuje slika 6.5. Zakaj je tak način delovanja dober? Če bi morali znotraj funkcij paziti, da ne uporabljamo enakih imen, kot so že definirana izven funkcij, potem bi morali že vnaprej predvideti kakšna imena bodo pri programiranju uporabljali vsi bodoči uporabniki naših funkcij. Prav tako bi morali biti zelo pazljivi, ko bi obstoječe funkcije uporabljali mi. Vedeti bi morali katere spremenljivke za izpis nečesa na zaslon na primer uporablja funkcija **print**. Tem imenom bi se morali izogibati, kar pa bi bilo skrajno nerodno in nesmiselno.



**Slika 6.3** Ob klicu funkcije se ustvari njen lokalni imenski prostor znotraj katerega se dodatno ustvariijo plitve kopije vrednosti, s katerimi smo funkcijo poklicali.



**Slika 6.4** Vsa imena, ki jih definiramo znotraj funkcije, se ustvariijo zgolj v lokalnem imenskem prostoru te funkcije.

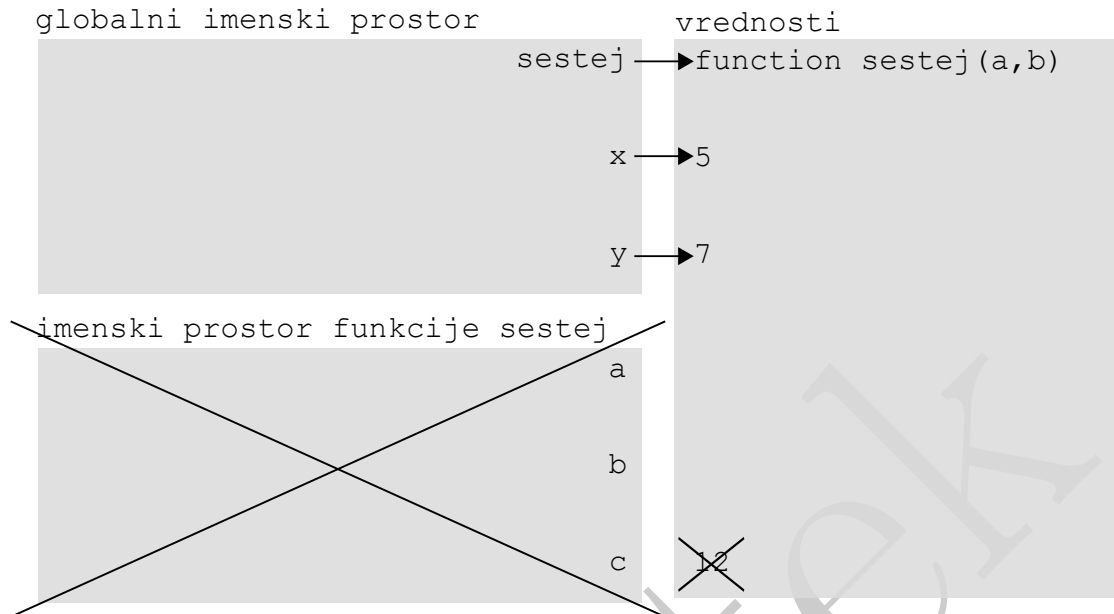


**Slika 6.5** Znotraj funkcije lahko uporabljamo enaka imena spremenljivk kot izven funkcije in s tem ne vplivamo na globalne spremenljivke.

Vprašanje, na katerega moramo še odgovoriti je, kaj se zgodi, ko se funkcija izvede do konca. V našem primeru se funkcija konča po izpisu vrednosti spremenljivke `c` (vrstica 3). Ko se funkcija konča, njenega lokalnega imenskega prostora ne potrebujemo več. Če bomo funkcijo še enkrat poklicali, bo Python pač ustvaril nov lokalni imenski prostor. Iz tega razloga, po končanju izvedbe funkcije, lokalni imenski prostor funkcije izgine. V našem konkretnem primeru torej imena `a`, `b` in `c` izginejo. Kaj pa vrednosti? Do vrednosti 12 ne moremo več dostopati preko nobene spremenljivke, zato se lahko izbriše tudi ta. Vrednosti 5 in 7 po drugi strani останeta, saj nanju še vedno kažeta imeni `x` in `y`. To prikazuje slika 6.6.

Iz globalnega imenskega prostora do lokalnih imenskih prostorov uporabljenih funkcij torej ne moremo dostopati, saj se po zaključku izvajanja funkcij (ko izvedba programa preide spet v globalni imenski prostor), lokalni imenski prostori izbrišejo. Kaj pa obratno? Iz lokalnega imenskega prostora funkcije, lahko dostopamo do globalnega (tudi zato se mu reče globalni), kar pomeni, da lahko dostopamo do vrednosti globalnih spremenljivk. Še pomembneje pa je to, da lahko iz lokalnega imenskega prostora funkcij, dostopamo do imen globalno definiranih funkcij. To pomeni, da lahko iz posamezne funkcije pokličemo druge funkcije (gnezdenje funkcij) ali pa tudi samo sebe. Slednjemu se reče *rekurzija*, ampak pustimo to za kdaj drugič.

Napišimo malo razširjen program, ki bo seštel vrednosti dveh seznamov. Pri tem si bomo pomagali z definicijo dveh funkcij.



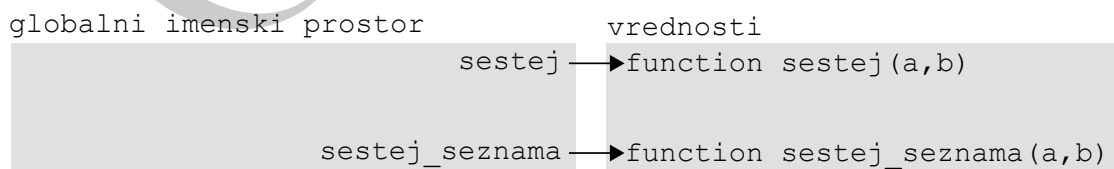
Slika 6.6 Po izvedbi klica funkcije, se njen imenski prostor izbriše.

```

1 def sestej(a, b): # sestej in izpisi
2     c = a + b
3     print(c)
4
5 def sestej_seznam(a,b): # sestej istolezne elemente
6     for i in range(len(a)):
7         sestej(a[i], b[i])
8
9 sestej_seznam([1,2,3],[4,5,6]) # klic funkcije

```

Iz funkcije `sestej_seznam` torej kličemo funkcijo `sestej`. Ali je to dovoljeno? Seveda. Imeni `sestej` in `sestej_seznam` bomo po izvedbi vrstic 1 in 5 imeli v globalnem imenskem prostoru, kot prikazuje slika 6.7.

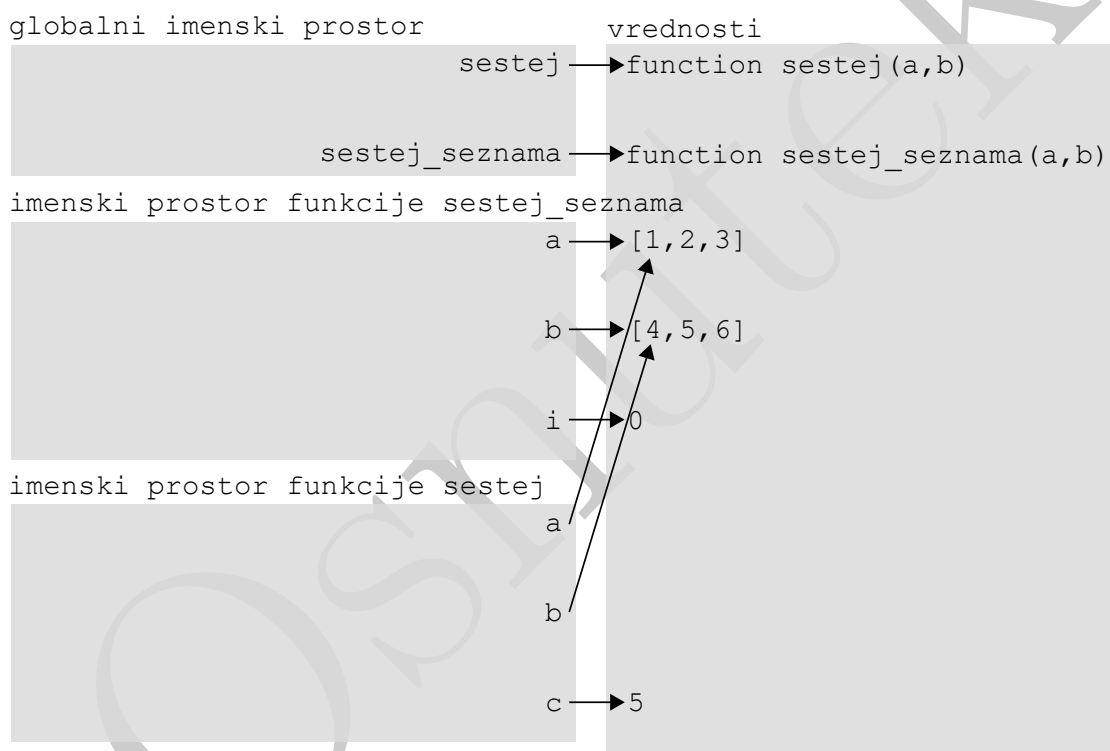


Slika 6.7 Imeni definiranih funkcij sta shranjeni v globalnem imenskem prostoru, zato jih lahko pokličemo od kjerkoli.

Ker je globalni imenski prostor viden tudi iz lokalnih imenskih prostorov posameznih

funkcij, jih lahko od tam tudi pokličemo. V sled temu je zgornji program popolnoma pravilen.

Če program pogledamo podrobneje, lahko vidimo, da obe funkciji uporabljata enaka imena spremenljivk. Tudi to ne bo povzročalo nobenih težav, saj bo vsaka funkcija dobila svoj lasten lokalni imenski prostor. Ko bomo poklicali funkcijo `sestej_seznam`, bo ta dobila lokalni imenski prostor. Ko bomo iz te funkcije poklicali funkcijo `sestej`, bo ta dobila svoj imenski prostor, ki se s prostorom funkcije `sestej_seznam` ne bo prekrival. Lokalne imenske prostore si torej lahko predstavljamo kot ločene mehurčke, ki se med seboj ne prekrivajo. Stanje našega programa ob prvi izvedbi funkcije `sestej` do vključno vrstice 2 prikazuje slika 6.8. Vprašanje za razmislek – zakaj se znotraj funkcije `sestej` ne ustvarita novi



**Slika 6.8** Lokalni imenski prostori funkcij so med seboj ločeni.

vrednosti, na kateri bosta kazali imeni `a` in `b`?

## 6.5 Vsaka funkcija vrača rezultat

V splošnem pri programiranju ločimo dva tipa funkcij, in sicer tiste, ki nekaj uporabnega vrnejo in tiste, ki nekaj uporabnega naredijo (vrnejo pa nič). V določenih programskih jezikih ti dve skupini nosijo celo posebna imena in so tudi



drugače definirane. Kaj pa v jeziku Python? V skupino funkcij, ki nekaj uporabnega vračajo bi lahko uvrstili npr. funkcijo `input`, ki prebere uporabnikov vnos in tega vrne kot podatkovni tip `str`. V skupino funkcij, ki ne vračajo nič kaj preveč uporabnega, je pa uporabno tisto, kar naredijo, pa spada funkcija `print`. Dejstvo je, da v Pythonu vsaka funkcija nekaj vrne pa tudi, če to ni čisto nič uporabnega. Poglejmo si kaj vrne funkcija `print`. Kako? Rezultat funkcije `print` bomo shranili v spremenljivko in vrednost te spremenljivke izpisali.

```
>>>a = print("testni_izpis")
testni izpis
>>>print(a)
None
```

Kaj se torej skriva v rezultatu funkcije `print`? Dobesedno nič oziroma `None`. Funkcija nekaj vrne, in sicer vrne nič. Preverimo lahko tudi njegov podatkovni tip.

```
>>>print(type(a))
<class 'NoneType'>
```

Nič oziroma `None` je torej poseben podatek, ki pripada podatkovnemu tipu nič oziroma `NoneType`. Ni sicer veliko, ampak nekaj pa je. Enak rezultat vračajo funkcije, ki smo jih definirali v prejšnjem razdelku. Lahko preverite sami.

Kaj pa če bi želeli, da naša funkcija vrne nekaj uporabnega? V tem primeru moramo od nje to eksplicitno zahtevati, in sicer s stavkom `return`.

Spremenimo funkcijo `sestaj`, tako da bo vsoto dveh števil vračala in ne izpisovala.

```
1 def sestaj(a, b): # sestaj in izpisi
2     c = a + b
3     return c
```

Z uporabo stavka `return` smo torej povedali, da želimo, da naša funkcija vrne vrednost spremenljivke `c`. Ali nismo tega naredili že prej? Ne. V prejšnji različici je funkcija vrednost spremenljivke `c` zgolj izpisovala. Ko se je funkcija končala, je njen lokalni imenski prostor izginil in z njim tudi vrednost spremenljivke `c`. Pogosto pa želimo rezultate funkcij uporabiti tudi v drugih delih naših programov (npr. ko uporabljamo funkcijo `input` želimo z uporabnikovim vnosom ponavadi nekaj uporabnega narediti in ga ne zgolj izpisati na zaslon). To lahko dosežemo s stavkom `return`. Kaj se zgodi, če funkcijo v našem programu zdaj še pokličemo. Razširimo program na sledeč način.

```
1 def sestaj(a, b): # sestaj in vrni
2     c = a + b
3     return c
4 sestaj(4,5)
```

Program tokrat ne izpiše ničesar. Zakaj ne? Ker tega od njega nismo nikjer zahtevali. Kaj torej naredi klic funkcije `sestaj`. V konkretnem primeru nič

uporabnega, saj izračuna vsoto števil 4 in 5, rezultat shrani v spremenljivko `c` in ko se funkcija zaključi, le-ta izgine, saj nanj ne kaže nobeno ime več. Kako pa bi lahko dobljeno vrednost uporabili še kje druge v našem programu? Podobno kot pri uporabi funkcije `input` – tako, da bi rezultat funkcije priredili spremenljivki.

```
1 def sestej(a, b): # sestej in vrni
2     c = a + b
3     return c
4 rezultat = sestej(4,5)
5 print(rezultat)
```

V zgornjem primeru bomo rezultat izpisali, lahko pa bi z njim naredili tudi karkoli drugega.

Stavek `return` ima dvojno vlogo. Ob njegovem klicu funkcija vrne rezultat, poleg tega pa se njeno izvajanje prekine (podobno, kot če uporabimo stavek `break` v kombinaciji z zanko `while` ali `for`).

Povadimo zdaj to na iskanju praštevil. Najprej poskusimo napisati funkcijo, ki uporabniku informacijo o tem, ali število je praštevilo ali ne, zgolj izpiše.

**Zgled 26** *Napiši funkcijo, ki kot argument prejme celo število in izpiše, če je podano število praštevilo ali ne.*

#### Rešitev 26

```
1 def prastevilo(stevilo):
2     for i in range(2, stevilo): # razpon preiskovanja
3         if stevilo % i == 0:
4             print(stevilo, "ni prastevilo")
5             break # dovolj je, da najdemo enega delitelja
6     else: # ce se je zanka odvirtela do konca
7         print(stevilo, "je prastevilo")
```

Podoben program smo napisali že, ko smo se srečali z zanko `for`, tako da ga posebej ne bomo komentirali. Poskusimo zdaj program spremeniti, tako da ne bo ničesar izpisoval, ampak bo uporabniku podal povratno informacijo o tem, če je število praštevilo ali ne.

**Zgled 27** *Napiši funkcijo, ki kot argument prejme celo število in vrne vrednost `True`, če je to število praštevilo, sicer pa vrne vrednost `False`.*

#### Rešitev 27

```
1 def prastevilo(stevilo):
2     for i in range(2, stevilo): # razpon preiskovanja
3         if stevilo % i == 0:
```

```

4         return False # prekine funkcijo in vrne False
5     return True # for se je odvirtel do konca

```

Ta rešitev je bistveno lepša in enostavnejša. Iz nje vidimo dodatno prednost stavka **return**, ki poleg vračanja rezultata prekine izvajanje funkcije. Ko smo v zanki **for** našli prvega delitelja, smo prekinili izvajanje funkcije in vrnili rezultat **False**. S prekinitvijo izvajanja funkcije se je prekinila tudi zanka, zato **break** ni več potreben. Če je program prišel do vrstice številka 5, zagotovo nismo našli nobenega delitelja, saj bi sicer funkcija že vrnila **False** in se nehala izvajati. Zato lahko v vrstici 5 brezpogojno vrnemo vrednost **True**. Tako tukaj ne potrebujemo niti stavka **if**. Dodatna prednost te rešitve je tudi to, da lahko zdaj rezultat preverjanja uporabimo tudi kje drugje. Lahko na primer napišemo funkcijo, ki izpiše vsa praštevila v določenem razponu.

**Zgled 28** *Napiši funkcijo, ki kot argument prejme celo število in izpiše vsa praštevila do vključno podanega števila.*

#### Rešitev 28

```

1 def prastevila(stevilo):
2     for kandidat in range(2, stevilo+1): # kandidati
3         if prastevilo(kandidat): # ali je prastevilo
4             print(kandidat)

```

Rešitev je izjemno enostavna. Sprehodili smo se čez vse možne kandidate za praštevila in za vsakega preverili, če je prastevilo. Kako? Tako, da smo poklicali funkcijo, ki vrne **True**, če je podano število prastevilo. Klic te funkcije smo samo še vstavili v stavek **if**, ki je izpisal število v primeru izpolnjenosti pogoja.

## 6.6 Izbirni argumenti

V določenih primerih želimo, da imajo določeni argumenti funkcije svoje vrednosti že vnaprej določene, razen v primeru, da uporabnik želi za te argumente uporabiti druge vrednosti. Če torej uporabnik vrednosti argumentov ne bo podal, bodo uporabljene njihove privzete vrednosti. V nasprotnem primeru bodo uporabljene uporabnikove vrednosti. Tak primer uporabe funkcij smo srečali že pri funkciji **print**, ki ima kar nekaj izbirnih argumentov. Privzeto gre funkcija **print** po vsakem klicu v novo vrstico (argument **end** je privzeto enak znaku za novo vrstico

**n**), v primeru več podanih vrednosti pa te izpišejo tako, da se med njih vrine presledke (argument **sep** je privzeto enak presledku). Njune privzete vrednosti lahko povežemo, tako da jih specificiramo ob klicu, npr. kot

```
>>> print(1,2,3,sep='+',end='␣')
1+2+3
```

Podobno lahko specificiramo izbirne argumente in njihove vrednosti pri definiciji svojih funkcij.

```
def ime_funkcije(arg1, arg2, ..., opc1=v1, opc2=v2, ...):
```

Paziti moramo samo na to, da so tisti argumenti, ki nimajo privzetih vrednosti vedno podani pred tistimi, ki privzete vrednosti imajo.

Povadimo to na malo bolj splošnem Fibonaccijevem zaporedju. Najprej poskusimo brez uporabe opsijskih argumentov.

**Zgled 29** *Napiši funkcijo, ki vrne Fibonaccijevo zaporedje števil, pri čemer naj uporabik poda dolžino zaporedja in prvi dve števili v zaporedju.*

#### Rešitev 29

```
1 def fibonacci(n, a, b):
2     if n == 1:
3         return [a]
4     f = [a,b]
5     for i in range(3,n+1):
6         f.append(f[-1]+f[-2])
7     return f
```

V primeru, da uporabnik želi imeti zaporedje dolžine 1, funkcija vrne zaporedje z enim elementom, in sicer *a*. V nasprotnem primeru naredi začetno zaporedje z elementoma *a* in *b* in potem v zanki doda ustrezno število dodatnih elementov, ki vsakič predstavljajo vsoto zadnjih dveh elementov zaporedja. S to rešitvijo sicer ni nič narobe, je pa dejstvo to, da si kot Fibonaccijevo zaporedje ponavadi predstavljamo zaporedje števil, ki se začne z vrednostima 1, 1. Smiselno bi torej bilo, da se privzeto naše zaporedje začne s števili 1, 1, razen če uporabnik tega ne specificira drugače.

**Zgled 30** *Napiši funkcijo, vrne Fibonaccijevo zaporedje števil, pri čemer naj uporabik poda dolžino zaporedja. Uporabnik lahko poda tudi prvi dve števili v zaporedju, ki sta privzeto enaki 1.*

#### Rešitev 30

```
1 def fibonacci(n, a=1, b=1):
2     if n == 1:
3         return [a]
4     f = [a,b]
```

```
5     for i in range(3,n+1):  
6         f.append(f[-1]+f[-2])  
7     return f
```

Zgornjo funkcijo lahko torej pokličemo tudi tako, da podamo samo dolžino zaporedja. V tem primeru bosta prvi dve števili v zaporedju enaki 1, 1. V primeru, da jo bomo poklicali tako, da podamo še vrednosti za argumenta **a** in **b** pa bosta za začetna elementa uporabljeni ti vrednosti.

Osnutek

Osnutek

## 7 Uporaba in pisanje modulov

### 7.1 Kaj so moduli?

Moduli predstavljajo Pythonove datoteke, ki vsebujejo implementacijo določenih funkcij, spremenljivk in razredov (angl. *classes*). Module lahko vključimo v svoje programe in na ta način razširimo osnovne funkcionalnosti jezika Python. Primeri že vgrajenih modulov, ki jih ni potrebno posebej namestiti, so modul `math`, v katerem so definirane določene matematične funkcije in konstante, modul `time` za vračanje podatkov o času in tvorjenje zakasnitev ter modul `random` za delo s (psevdo)naključnimi števili.

### 7.2 Uporaba modulov

Module lahko v svoje programe vključimo na različne načine, v vseh primerih pa uporabljamo rezervirano besedo `import`. Če želimo npr. v naš program uvoziti celoten modul `math`, lahko to naredimo s sledečo vrstico:

```
import math
```

oziroma v splošnem

```
import ime_modula
```

Najprej lahko preverimo kaj uvožen modul dejansko ponuja. Če je pisec modula bil priden in napisal tudi dokumentacijo, bomo za to lahko uporabili funkcijo `help`

```
help(ime_modula)
```

Funkcija nam bo izpisala nekaj osnovnih informacij o modulu, tako da bo uporaba lažja. Seveda pa lahko informacije o modulu poiščemo tudi na internetu, ki pa včasih ni na voljo (npr. v času pisanja kolokvijev in izpitov), zato se je dobro navaditi tudi uporabe zgoraj omenjene funkcije.

Če bi zdaj želeli dostopati do posamezne funkcije, ki je v uvoženem modulu definirana, bi to naredili na sledeč način

```
ime_modula.ime_funkcije(argumenti)
```

Če bi npr. želeli izračunati sinus števila shranjenega v spremenljivki `x` in rezultat shraniti v spremenljivko `y`, bi za to uporabil funkcijo `sin`, ki je vsebovana v modulu `math`. Poklicali bi jo takole

```
y=math.sin(x)
```

Včasih imajo moduli zelo dolga in težko berljiva imena. Če želimo modul uvoziti pod drugačnim imenom (lahko bi rekli psevdonimom), uvoz dopolnimo z `as` stavkom:

```
import dolgo_ime_modula as psevdonim
```

Tako lahko pri klicanju funkcij (ali pa česarkoli že) modula podajamo le kratko ime modula. V prejšnjem primeru bi kodo lahko spremenili na sledeč način:

```
import math as m
y=m.sin(x)
```

V določenih primerih pa želimo iz modula uvoziti le določeno funkcijo (spremenljivko, razred). Takrat lahko uporabimo rezervirano besedo `from`, in sicer takole

```
from ime_modula import ime_funkcije
```

S takim načinom uvažanja smo uvozili le tisto kar potrebujemo, poleg tega pa zdaj pri klicu funkcije imena modula ni potrebno več podajati. Primer sinusa bi se spremenil v sledečo kodo

```
from math import sin
y=sin(x)
```

Zdaj smo iz modula uvozili zgolj funkcijo `sin` – če bi želeli imeti še kakšno drugo funkcijo, npr. kosinus, bi jo morali uvoziti ločeno oziroma hkrati s funkcijo `sin`. To bi naredili takole:

```
from math import sin,cos
```

Lahko pa naredimo še nekaj, kar ponavadi ni priporočljivo. Uvozimo lahko vse, kar je v modulu definirano, in sicer namesto imena funkcije podamo `*`, ki se v računalništvu velikokrat uporablja kot simbol za *vse*. Rekli bomo torej *iz modula uvozi vse*:

```
from ime_modula import *
```

V primeru modula `math` bomo zapisali takole:

```
from math import *
```

Zakaj tak način uvažanja ni priporočljiv? Vse funkcije, spremenljivke in razrede modula smo zdaj dobili v naš globalni imenski prostor. Pri tem je velika verjetnost, da smo si s tem povozili kakšno od spremenljivk, ki jo tam že uporabljamo in ima enako ime kot kakšna izmed funkcij ali spremenljivk definiranih v modulu. Zato se takemu načinu uvažanja modulov izogibamo.



## 7.3 Definicija in uporaba lastnih modulov

Vsi programi, ki smo jih do zdaj napisali, predstavljajo module, ki jih lahko uvozimo v druge programe. To pomeni, da pri reševanju nekega (bolj kompleksnega) problema ni potrebno vse kode napisati v isti datoteki, ampak lahko datoteko razdelimo po smiselnih modulih, pri čemer lahko funkcije prvega modula uporabljamo v drugem in obratno. Pri tem lahko uporabimo kodo opisano v prejšnjem razdelku. Paziti moramo le na to, da se modul, ki ga uvažamo nahaja v isti mapi kot modul, v katerega kodo uvažamo. V nasprotnem primeru moramo pri uvažanju modula do drugega modula podati še pot do njega. Če se modul, ki ga želimo uvoziti, npr. nahaja v podmapi mapi `podmapa`, ga bomo uvozili na sledeč način:

```
import podmapa.ime_modula
```

Če v tem primeru ne želimo, da modul vsakič posebej kličemo z imenom `podmapa.ime_modula`, ga je smiselno uvoziti pod krajšim imenom takole

```
import podmapa.ime_modula as ime_modula
```

Mogoče se sprašujete zakaj poti ni bilo potrebno podajati pri uvažanju modula `math`. Dejstvo je, da so določeni moduli v Python že vgrajeni, sicer pa Python module, poleg v trenutni delovni mapi, išče tudi v mapi `lib\site-packages`, kamor se shranijo namestitve vseh modulov, ki jih bomo v prihodnosti potencialno še namestili.

## 7.4 Nameščanje novih modulov

Na spletu obstaja veliko modulov, ki so jih razvili programerji pred nami. Te lahko uporabimo, kadar želimo pri reševanju določenega problema uporabiti višjenivojske sestavine. Če želimo npr. narisati graf povprečne mesečne plače v Sloveniji, nam ni potrebno študirati, kako se lotiti kakršnegakoli risanja v jeziku Python, ampak enostavno uporabimo paket (paket ni nič drugega kot zbirka modulov) `matplotlib` in njegove funkcije za risanje grafov. Problem, s katerim se srečamo, je, da tovrstni paketi v osnovni različici Pythona še niso nameščeni (razen, če si nismo namestili distribucije Anaconda <sup>1</sup>) Pred uporabo jih moramo torej namestiti. Problem nameščanja tovrstnih paketov je, poleg včasih mukotrpnega procesa iskanja ustreznih namestitvenih datotek in ročne namestitve, tudi v tem, da za svoje delovanje večina paketov uporablja druge pakete, ti paketi spet druge in tako naprej. Temu rečemo odvisnost med paketi (angl. *package dependency*). Da pa se s tem običajnemu uporabniku Pythona ni potrebno ukvarjati, Python okolje vsebuje orodje `pip` (angl. *pip installs packages*), ki preko repozitorija PyPI (angl.

<sup>1</sup>Anaconda je distribucija Pythona za znanstveno računanje, ki ima nameščenih že večino paketov, ki jih za tako računanje potrebujemo. Dostopna je na povezavi <https://www.anaconda.com/>.

*Python package index*) poišče in namesti ustrezne pakete avtomatsko. Nameščen je že skupaj z osnovno distribucijo Pythona. Vse kar poleg tega potrebujemo je še internetna povezava in ime paketa, ki ga želimo namestiti. Orodje `pip` bomo pognali iz sistemske ukazne vrstice (v operacijskem sistemu Windows jo zaženemo tako, da v start meni vpišemo `cmd`). V primeru, da smo ob namestitvi Pythona obkljukali opcijo *Add Python to path*, lahko orodje `pip` poženemo iz poljubne lokacije. V nasprotnem primeru se moramo premakniti v mapo, kjer `pip` nameščen (podmapa `Scripts` mape, kjer je nameščen Python). Paket z imenom `ime_paketa` zdaj namestimo s sledečim ukazom

```
>>> pip install ime_paketa
```

in `pip` bo poskrbel za vse ostalo.

## 8 Spremenljivost podatkovnih tipov in terke

### 8.1 Kaj je spremenljivost?

Določeni podatkovni tipi v Pythonu so spremenljivi (angl. *mutable*), določeni pa ne. Kaj to pomeni? Če je nek podatek nespremenljiv (angl. *immutable*), to pomeni, da ga po tistem, ko je enkrat ustvarjen, ne moremo več spreminjati. Lahko pa naredimo nov podatek, ki odraža spremembo, ki jo želimo nad podatkom narediti. Primeri nespremenljivih podatkovnih tipov so števila tipa `int` in `float`, niz oziroma `str` in `bool` (spremenljivost osnovnih podatkovnih tipov v jeziku Python prikazuje tabela 8.1). To so torej skoraj vsi podatkovni tipi, ki smo jih do sedaj spoznali. Če je določen podatek spremenljiv, potem ga lahko spreminjamo tudi kasneje. Primer spremenljivega podatkovnega tipa je seznam oziroma `list`. Spremenljivost podatkovnih tipov na videz izgleda kot nekaj, s čimer se nam pri osnovah programiranja niti ne bi bilo potrebno ukvarjati. Žal pa ima veliko posledic, ki jih brez razumevanja spremenljivosti težko razumemo, zato je smiselno, da si celoten koncept podrobneje pogledamo.

**Tabela 8.1** Spremenljivost osnovnih podatkovnih tipov v jeziku Python

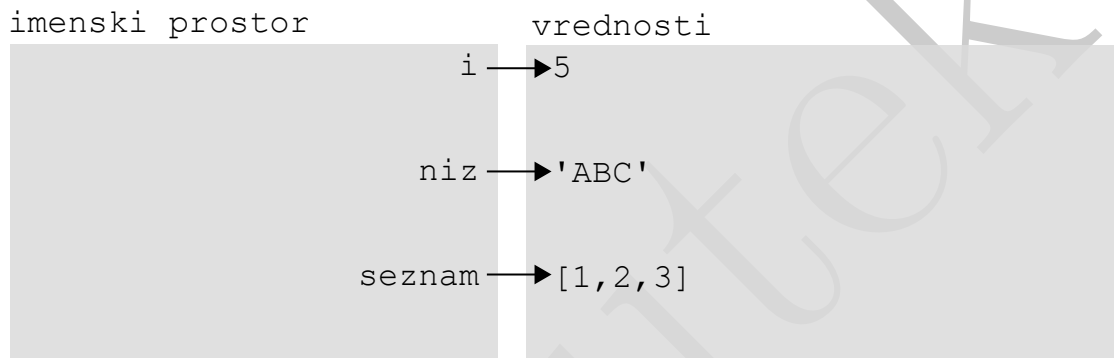
podatkovni tip	opis	spremenljiv
<code>bool</code>	<i>boolean</i> ( <code>True</code> , <code>False</code> )	Ne
<code>int</code>	<i>integer</i> (celo število)	Ne
<code>float</code>	<i>floating-point</i> (decimalno število)	Ne
<code>str</code>	<i>string</i> (niz)	Ne
<code>list</code>	<i>list</i> (seznam)	Da
<code>tuple</code>	<i>tuple</i> (terka)	Ne
<code>dict</code>	<i>dictionary</i> (slovar)	Da
<code>set</code>	<i>set</i> (množica)	Da
<code>frozenset</code>	<i>frozenset</i> (nespremenljiva množica)	Ne

## 8.2 Kaj se zgodi ob prirejanju spremenljivk?

Kaj se zgodi, ko spremenljivki priredimo neko vrednost že vemo. V imenskem prostoru, kjer spremenljivko definiramo, se pojavijo imena, ki smo jih dodelili spremenljivkam, v pomnilniku pa se ustvarijo vrednosti, na katere ta imena kažejo. Zapooredje prireditvenih stavkov

```
>>> i = 1
>>> niz = 'ABC'
>>> seznam = [1,2,3]
```

lahko ponazorimo s sliko 8.1. Kaj pa se zgodi, če spremenljivko priredimo drugi imenski prostor



**Slika 8.1** Ob prireditvi se imenu spremenljivke priredi podana vrednost.

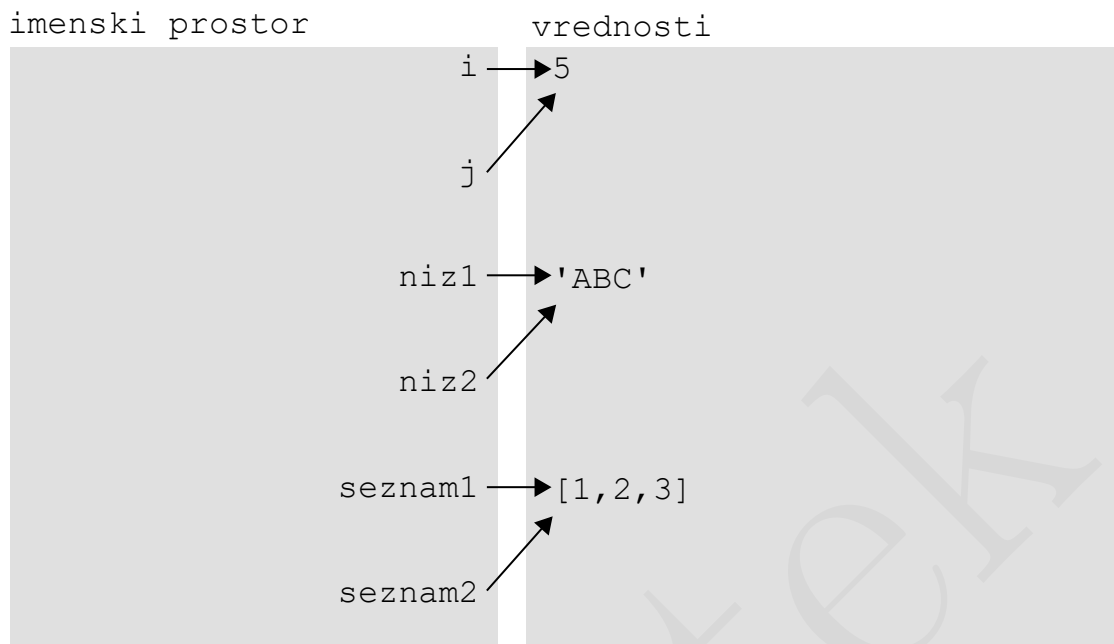
spremenljivki, na primer takole:

```
>>> i = 1
>>> j = i
>>> niz1 = 'ABC'
>>> niz2 = niz1
>>> seznam1 = [1,2,3]
>>> seznam2 = seznam1
```

Nekaj podobnega smo srečali že pri klicu funkcije. Spomnimo se, da se v Pythonu v takem primeru naredi t.i. *plitva kopija* spremenljivke. To pomeni, da vrednost v pomnilniku dobi novo ime. Do dejanskega (*globokega*) kopiranja vrednosti v tem primeru ne pride. To lahko ponazorimo s sliko 8.2. Na tak način je delovanje tako s časovnega stališča (hitrost) kot tudi s prostorskega stališča (poraba pomnilnika) bolj varčno.

## 8.3 Kaj se zgodi ob spreminjanju vrednosti spremenljivk?

Kaj pa se zgodi, če vrednost nove (ali pa stare) spremenljivke spremenimo? Vse skupaj zavisi od tega ali je podatek, ki ga spreminjamo spremenljiv ali ne. Spo-



**Slika 8.2** Ob prireditvi spremenljivke drugi spremenljivki se ustvari plitva kopija spremenljivke.

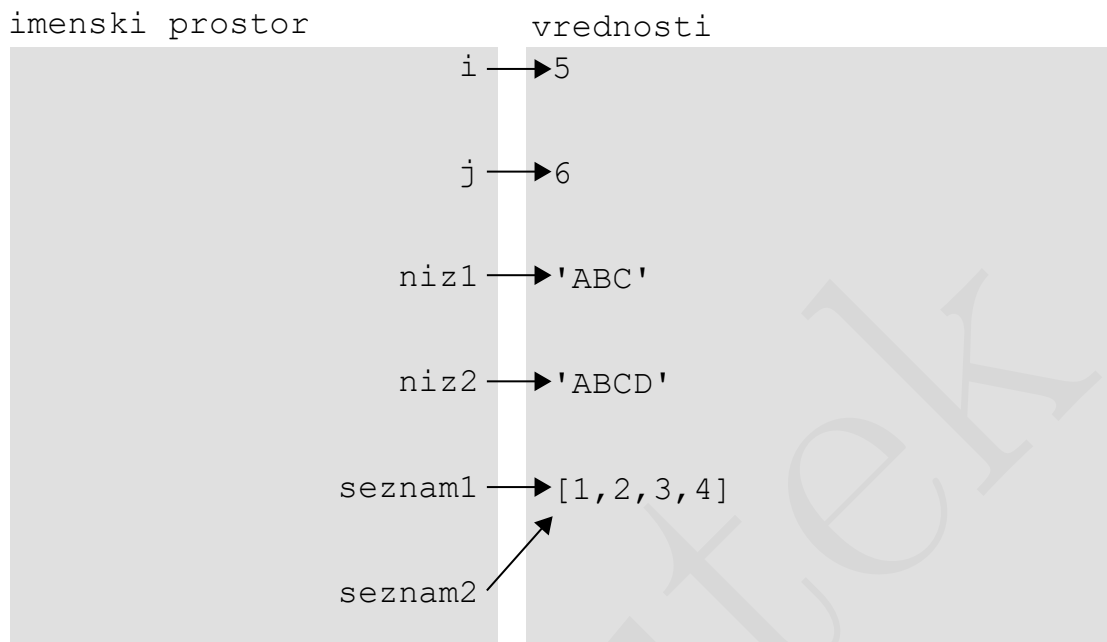
mnimo se. Spremenljiv podatek lahko spreminjamo, ko pa poskusimo spremeniti nespremenljiv podatek, se ustvari njegova kopija (globoka), ki odraža narejeno spremembo. Če npr. uporabimo operator `+=`, bo v primeru spremenljivega podatka spremenjen obstoječ podatek, v primeru nespremenljivega podatka pa bo ustvarjen nov podatek, ki bo odražal narejeno spremembo.

Kaj pa se zgodi v primeru, da na podatek kaže več imen, kot v scenariju zgoraj. Ali se bo po izvedbi spodnje kode sprememba odražala tudi preko drugih imen podatka? Poglejmo si spodnjo kodo.

```
>>> i = 1
>>> j = i
>>> j += 1
>>> niz1 = 'ABC'
>>> niz2 = niz1
>>> niz2 += 'D'
>>> seznam1 = [1,2,3]
>>> seznam2 = seznam1
>>> seznam2 += [4]
```

Zanima nas ali se po spreminjanju spremenljivk `j`, `niz2` in `seznam2` spremembe odražajo tudi na spremenljivkah `i`, `niz1` in `seznam1`. Odgovor ni enostaven da ali ne. Odgovor je namreč odvisen od spremenljivosti podatka, ki ga spreminjamo.

Situacijo po spreminjanju podatka z operatorjem `+=` prikazuje slika 8.3. V primeru,



**Slika 8.3** Ob spreminjanju spremenljivih podatkov se spremenijo vse plitve kopije podatka.

da je podatek spremenljiv, se torej sprememba odraža na vseh spremenljivkah, ki predstavljajo plitve kopije tega podatka. S tem ko v zgornjem zgledu spreminjamo spremenljivko `seznam2`, spreminjamo tudi spremenljivko `seznam1`. Po drugi strani spreminjanje spremenljivk `j` in `niz2` ustvari globoko kopijo spremenljivk `j` in `niz2`, ki odraža narejeno spremembo. Globoka kopija predstavlja nov podatek, tj. podatek ki se razlikuje od tistega, na katerega kažeta imeni `i` in `niz1`. Posledica tega je, da spreminjanje vrednosti spremenljivk `j` in `niz2` na vrednostih spremenljivk `i` in `niz1` ne vplivajo, saj pripadajo nespremenljivim podatkovnim tipom.

## 8.4 Ali funkcije spreminjajo vrednosti svojim argumentom?

Spomnimo se, da se ob klicu funkcije ustvari lokalni imenski prostor funkcije. V lokalnem imenskem prostoru se ob klicu spremenljivkam, ki nastopajo kot argumenti funkcije, priredi vrednosti, s katerimi smo funkcijo poklicali. V primeru, da smo funkcijo poklicali z globalnimi spremenljivkami, se argumentom funkcije priredi plitva kopija teh spremenljivk. Vprašanje pa je ali se bodo ob spreminjanju argumentov funkcije spremembe odražale tudi izven funkcije, torej po tem, ko se bo funkcija že končala. Vprašanje lahko ponazorimo s spodnjim zgledom.

**Zgled 31** Kakšna je vrednost spremenljivk *st1*, *niz1* in *seznam1* po izvedbi spodnje kode in kakšen bo izpis programa?

```

1 def spremeni(a, b):
2     a += b
3
4     i = 1
5     j = 2
6     spremeni(i, j)
7     print(i)
8
9     niz1 = "ABC"
10    niz2 = "D"
11    spremeni(niz1, niz2)
12    print(niz1)
13
14    seznam1 = [1, 2, 3]
15    seznam2 = [4, 5, 6]
16    spremeni(seznam1, seznam2)
17    print(seznam1)

```

**Rešitev 31** Ob klicu funkcije spremenljivki, s katerima funkcijo pokličemo, dobimo plitvi kopiji z imeni *a* in *b* v lokalnem imenskem prostoru funkcije. Znotraj funkcije plitvo kopijo z imenom *a* spreminjamo. V primeru, da je spremenljivka spremenljivega podatkovnega tipa (npr. seznam) se spreminja obstoječ podatek, na katerega kaže tudi globalna spremenljivka, kar pomeni, da se bo sprememba odražala tudi izven funkcije. V primeru, da je spremenljivka nespremenljivega podatkovnega tipa, se ustvari globoka kopija podatka, ki bo odražala narejeno spremembo. Spremeni se torej zgolj spremenljivka, ki je definirana znotraj funkcije, ta sprememba pa izven funkcije ne bo vidna.

Spremenljivki *i* in *niz1* se torej po klicu funkcije ne bosta spremenili, spremenljivka *seznam1* pa se bo spremenila. Po izvedbi programa bo izpis sledeč:

```

1 # nespremenjena vrednost
ABC # nespremenjena vrednost
[1, 2, 3, 4, 5, 6] # spremenjena vrednost

```

Funkcije torej lahko spreminjajo vrednosti svojim argumentom, tako da so spremembe vidne tudi izven funkcij, ampak samo v primeru, ko so podani argumenti spremenljivega podatkovnega tipa.

## 8.5 Terke

Zdaj, ko vemo, kaj je to spremenljivost, lahko razložimo tudi, kaj so terke (angl. *tuples*). Terka oziroma `tuple` predstavlja sekvenčen podatkovni tip, ki je nespremenljiv. Ker terke zelo spominjajo na sezname, bi jim lahko rekli tudi nespremenljivi sezname. Načeloma bi lahko pri programiranju shajali tudi brez njih (tako kot bi lahko shajali tudi brez zanke `for`), ampak njihova uporaba v veliko primerih naredi naše programe lepše in boljše (tako kot uporaba zanke `for`).

## 8.6 Uporaba terk

Terko definiramo z navadnimi oklepaji, tj. `( in )`, znotraj katerih naštejemo elemente. Terko treh elementov, bi lahko definirali na primer takole:

```
>>> terka=("Janez", 1.8, 75)
```

Za terko je pa mogoče bolj kot oklepaji bistveno naštevanje elementov, zato bi tudi, če bi oklepaje izpustili, dobili terko. Takole:

```
>>> terka="Janez", 1.8, 75
>>> terka
("Janez", 1.8, 75)
>>> type(terka)
<class 'tuple'>
```

Python je ob naštevanju elementov z vejicami ugotovil, da želimo imeti terko in jo naredil. Python ima nekoliko težav, ko želimo narediti terko dolžine 1, saj si v tem primeru oklepaje razlaga kot operator, ki določa prioriteto. V primeru, da znotraj oklepajev damo zgolj eno npr. celo število, bomo torej dobili podatek, ki pripada podatkovnemu tipu `int` in ne `tuple`:

```
>>> terka=(1)
>>> terka
1
>>> type(terka)
<class 'int'>
```

Že prej smo omenili to, da je bistvena lasntnost terk naštevanje elementov, ki jih ločimo z vejicami. Kako v primeru enega elementa povemo, da gre za naštevanje? Tako, da za njim napišemo vejico: `tuple`:

```
>>> terka=(1,)
>>> terka
(1,)
>>> type(terka)
<class 'tuple'>
```



Gre pa seveda tudi brez oklepajev: `tuple`:

```
>>> terka=1,
>>> terka
(1,)
>>> type(terka)
<class 'tuple'>
```

Kaj lahko s terkami počnemo? Podobno kot sezname lahko elemente terke indeksiramo, lahko delamo rezine, lahko preverjamo vsebovanost elementov, z zanko `for` se lahko čez elemente terke sprehajamo itd. Z njimi lahko delamo torej skoraj vse, kar smo delali s seznamami. Skoraj vse? Ker so terke nespremenljive, jih seveda ne moremo spreminjati, tako kot lahko spreminjamo sezname. Poskusimo:

```
>>> terka = ("Janez", 1.8, 75)
>>> terka[0] = "Marko"
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    terka[0] = "Marko"
TypeError: 'tuple' object does not support item assignment
```

Očitno res ne gre. Seveda ne, saj so nespremenljive. Zakaj bi terke potem sploh uporabljali? Nekaj primerov, pri katerih je uporaba terk smiselna, je podanih v nadaljevanju poglavja.

## 8.7 Seznami terk in razpakiranje elementov terk

Nenapisano pravilo (ki ga seveda lahko kršimo) je, da v sezname shranjujemo homogene podatke, torej podatke, ki se nanašajo npr. na isto spremenljivko. To pomeni, da vsak element seznama obravnavamo na enak način, saj se nanaša na isto količino. Terke se pogosto uporabljajo za shranjevanje heterogenih podatkov, tj. podatkov različnih tipov, ki pa pripadajo isti entiteti, kot je npr. oseba ali meritev. Če si torej želimo pri določeni entiteti zabeležiti več podatkov, lahko uporabimo seznam terk. Na primer, če imamo vzorec oseb, pri čemer za vsako osebo beležimo ime, višino in težo, potem lahko uporabimo seznam terk, pri čemer vsaka izmed terk vsebuje ime, višino in težo dotične osebe. Primer takega seznama bi bil

```
meritve = [("Janez", 1.8, 75),
           ("Ana", 1.65, 60),
           ("Nika", 1.66, 55)]
```

Elementi seznama so torej homogeni, kar pomeni, da bomo vsakega obravnavali enako. Elementi seznama so namreč terke, ki imajo vsakič enako obliko. Na 0-tem indeksu je shranjeno ime osebe, na indeksu 1 višina v metrih in na indeksu 2

teža osebe v kilogramih. Elementi posamezne terke pa očitno pripadajo različnim spremenljivkam.

Tak način predstavitve podatkov bomo srečali velikokrat. Kako pa lahko tako shranjene podatke uporabimo pri nadaljnji analizi. Na primer pri izračunu in izpisu indeksa telesnih mas posamezne osebe v seznamu. Tako, da se čez seznam sprehodimo z zanko `for` in v vsaki iteraciji zanke tekočo terko *razpakiramo* in tako pridemo do konkretnih vrednosti. To lahko naredimo na sledeč način:

```
for meritev in meritve:
    ime = meritev[0]
    visina = meritev[1]
    teza = meritev[2]
    itm = teza/visina**2
    print("ITM_osebe", ime, "je", itm)
```

Do posameznih elementov terke smo torej prišli z njihovim indeksiranjem. Terke pa lahko razpakiramo veliko hitreje, in sicer tako, da terko priredimo drugi terki, ki vsebuje imena spremenljivk, v katere želimo vrednosti shraniti oziroma razpakirati. Takole:

```
(spremenljivka1, spremenljivka2,...) = terka
```

Paziti moramo le na to, da terka na levi strani vsebuje enako število elementov kot terka na desni strani prireditvenega stavka. Kaj smo v zgornjem stavku pravzaprav naredili? Naredili smo terko spremenljivk, ki smo ji priredili terko na desni strani. Ker terki spremenljivk nismo dali nobenega imena, je v imenski prostor nismo shranili in zato tudi ni shranjena nikjer v pomnilniku. So pa v pomnilniku ostale spremenljivke, v katere smo razpakirali terko na desni. Kot smo videli že zgoraj pa lahko oklepaje pri definiciji tudi izpustimo. Torej lahko napišemo tudi nekaj takega

```
spremenljivka1, spremenljivka2,... = terka
```

Mimogrede, tako razpakiranje elementov bi delovalo tudi, če bi imeli na desni strani prireditvenega stavka seznam.

Tak način razpakiranja elementov lahko uporabimo v našem zgledu z računanjem indeksa telesnih mas, s čimer se koda občutna skrajša:

```
for meritev in meritve:
    ime, visina, teza = meritev
    itm = teza/visina**2
    print("ITM_osebe", ime, "je", itm)
```

Kodo lahko še dodatno skrajšamo, če razpakiranje naredimo kar v glavi zanke `for`:

```
for ime, visina, teza in meritve:
    itm = teza/visina**2
    print("ITM_osebe", ime, "je", itm)
```

## 8.8 Pakiranje seznamov v seznane terk

Seznami terk, kot smo jih srečali zgoraj, torej predstavljajo lep način zapisovanja podatkov, ko želimo pri posamezni entiteti imeti več podatkov. Dejstvo pa je, da velikokrat podatkov ne dobimo v taki obliki, ampak dobimo za vsako količino svoj seznam. Pri tem so sezname med seboj poravnani, kar pomeni, da istoležni elementi v vseh seznamih pripadajo isti entiteti. Elementi na indeksu 0 torej pripadajo entiteti 0, elementi na indeksu 1 entiteti 1 itd. V primeru imen, višin in tež, bi torej imeli tri sezname v obliki

```
imena = ["Janez", "Ana", "Nika"]
visine = [1.8, 1.65, 1.66]
teze = [75, 60, 55]
```

Elementi vseh treh seznamov torej na indeksu 0 pripadajo Janezu, na indeksu 1 Ani in na indeksu 2 Niki. Kaj imajo ti sezname skupnega? Indekse! Čez take podatke bi se torej lahko sprehodili tako, da se sprehajamo po indeksih in ne direktno po elementih. Naredimo torej sprehod z znako `for` od indeksa 0 do dolžine seznama - 1. Dolžine katerega seznama? Ni važno, saj so vsi enko dolgi (oziroma vsaj smiselno bi bilo, da so). To bi lahko naredili takole:

```
for i in range(len(imena)):
    ime = imena[i]
    visina = visine[i]
    teza = teze[i]
    itm = teza/visina**2
    print("ITM_osebe", ime, "je", itm)
```

Kako pa bi lahko iz treh seznamov naredili seznam terk, s katerim smo delali zgoraj. Izkaže se, da se s takim problemom srečamo relativno pogosto, zato nam Python za *zapakiranje* več seznamov v seznam terk ponuja vgrajeno funkcijo `zip`. Funkcija `zip` iz zgornjih treh seznamov naredi točno to, kar bi si želeli:

```
>>> meritve = zip(imena, visine, teze)
>>> meritve
<zip object at 0x0000019A2A865D48>
```

Tale izpis je malo čuden, ampak ni z njim nič narobe. Funkcija `zip` je t.i. *iterator*, ki dejanski seznam elementov vrne, šele ko ga potrebujemo oziroma posamezne elemente seznama vrača sproti. Če bi želeli imeti lepši izpis, bi lahko do njega prišli tako, da rezultat funkcije `zip` eksplicitno pretvorimo v seznam s funkcijo `list`:

```
>>> meritve = list(zip(imena, visine, teze))
>>> meritve
[('Janez', 1.8, 75), ('Ana', 1.65, 60), ('Nika', 1.66, 55)]
```

Sprehod čez sezname lahko torej naredimo na podoben način kot v prejšnjem poglavju, le da prej sezname zapakiramo v seznam terk:

```
for ime, visina, teza in zip(imena, visine, teze):
    itm = teza/visina**2
    print("ITM_osebe", ime, "je", itm)
```

## 8.9 Zahteva po nespremenljivosti

V določenih primerih Python zahteva uporabo nespremenljivih podatkovnih tipov. Nespremenljive podatkovne tipe moramo uporabiti, kadar želimo podatke shranjevati v množico (`set`) in kadar želimo nek podatek uporabiti kot ključ (`key`) slovarja (`dict`). Če želimo v takem primeru uporabiti več elementov, moramo namesto po seznamu poseči po terki. V teh primerih je torej uporaba terk obvezna. Več o množicah in slovarjih bomo izvedeli prav kmalu.

Drug primer, v katerem bi nespremenljivost bila zaželeno (ne pa obvezna), je, ko ne želimo, da funkcija spremeni vrednosti posanega argumenta. Kot smo videli lahko funkcije spreminjajo vrednosti svojih argumentov, tako da bodo spremembe vidne tudi izven funkcij. Če bi radi zagotovilo, da se podan argument izven funkcije zagotovo ne bo spremenil, namesto spremenljivega seznama enostavno uporabimo nespremenljivo terko. V tem kontekstu si pogledajmo spodnji zgled.

**Zgled 32** *Kakšna je vrednost spremenljivk `seznam1` in `terka1` po izvedbi spodnje kode in kakšen bo izpis programa?*

```
1 def spremeni(a, b):
2     a += b
3
4 seznam1 = [1,2,3]
5 seznam2 = [4,5,6]
6 spremeni(seznam1, seznam2)
7 print(seznam1)
8
9 terka1 = (1,2,3)
10 terka2 = (4,5,6)
11 spremeni(terka1, terka2)
12 print(terka1)
```

**Rešitev 32** *Kot smo videli že prej, se sprememba, ki smo jo nad plitvo kopijo spremenljivke `seznam1` naredili znotraj funkcije, odraža tudi izven funkcije, saj je seznam spremenljiv podatkovni tip. Ko torej spreminjamo njegovo plitvo kopijo, s tem spreminjamo vse spremenljivke, ki nanj kažejo.*

Kaj pa se zgodi, ko funkcijo pokličemo s terko. Najprej se ustvari plitva kopija terke v lokalnem imenskem prostoru funkcije (spremenljivka *a*). Ker je terka nespremenljiv podatkovni tip, je ne moremo spreminjati. Ob njenem spreminjanju se zato ustvari globoka kopija, torej nov podatek v pomnilniku, ki odraža narejeno spremembo. Na ta podatek pa kaže zgolj lokalna spremenljivka *a*. Ko se funkcija zaključi, njen lokalni imenski prostor skupaj z lokalno spremenljivko *a* izgine (tako kot tudi spremenjena terka – globoka kopija terke, s katero smo funkcijo poklicali). V sled temu sprememba, ki smo jo naredili znotraj funkcije, izven funkcije ni vidna. Po izvedbi funkcije *spremeni* ima spremenljivka *seznam1* spremenjeno vrednost (*[1,2,3,4,5,6]*), spremenljivka *terka1* pa ostane taka, kot je bila pred klicem funkcije (*(1,2,3)*). Izpis programa je torej

```
[1,2,3,4,5,6] # spremenjena vrednost
(1,2,3) # nespremenjena vrednost
```

Funkcijam, ki kot argumente sprejemajo spremenljive podatkovne tipe, spremenjenih vhodnih argumentov ni potrebno vračati, saj se bodo spremembe odražale tudi izven funkcije. Funkcije, ki kot argumente sprejemajo nespremenljive podatkovne tipe, morajo spremenjene vhodne argumente eksplicitno vrniti, saj so spremenjene vrednosti sicer za vedno izgubljene. Oba načina si pogledjmo v spodnjih zgledih. Najprej si pogledjmo kako se lotiti pisanja in uporabe funkcije, ki sprejema nespremenljive podatke.

**Zgled 33** Napiši funkcijo *dodaj\_AT\_niz*, ki sprejme dve nukleotidini zaporedji zapisani kot niza in v prvo zaporedje doda vse ponovitve baz *A* in *T* v enakem zaporedju kot nastopajo v drugem nizu. Funkcijo uporabi na zaporedjih *'ATCG'* in *'AATGGAATGG'*, tako da bo prvo zaporedje po njeni izvedbi spremenjeno.

**Rešitev 33** Funkcija sprejema in spreminja podatke tipa *str*, ki je nespremenljiv podatkovni tip. Če bomo vhodne argumente spreminjali znotraj funkcije, se te spremembe izven funkcije ne bodo odražale, kar pomeni, da mora funkcija vračati spremenjen niz. Napišimo jo.

```
def dodaj_AT_niz(zaporedje1, zaporedje2):
    for baza in zaporedje2:
        if baza in 'AT':
            zaporedje1 += baza
    return zaporedje1
```

Na koncu torej vrnemo spremenjeno zaporedje1. Kljub temu, da smo znotraj funkcije to spremenljivko spreminjali, spremembe izven funkcije ne bodo vidne.

Klic funkcije moramo izvesti na tak način, da bo spremenila vrednost prve spremenljive, s katero funkcijo kličemo. Kako to doseči? Enostavno tako, da rezultat funkcije priredimo vrednosti te spremenljivke. Takole:

```
>>> zaporedje1 = 'ATCG'
>>> zaporedje2 = 'AATGGAATGG'
>>> zaporedje1 = dodaj_AT_niz(zaporedje1, zaporedje2)
```

Na tak način smo vrednost spremenljivke `zaporedje1` spremenili.

Poglejmo si še kakšne so razlike pri delu s spremenljivimi podatki.

**Zgled 34** Napiši funkcijo `dodaj_AT_seznam`, ki sprejme dve nukleotidini zaporedji zapisani kot seznama enoznakovnih nizov (baz) in v prvo zaporedje doda vse ponovitve baz `A` in `T` v enakem zaporedju kot nastopajo v drugem nizu. Funkcijo uporabi na zaporedjih `['A', 'T', 'C', 'G']` in `'A', 'A', 'T', 'G', 'G', 'A', 'A', 'T', 'G', 'G'`, tako da bo prvo zaporedje po njeni izvedbi spremenjeno.

**Rešitev 34** Navodilo naloge je praktično enako kot prej, le da tokrat namesto nespremenljivih podatkovnih tipov uporabljamo spremenljive. To pomeni, da ni potrebe po tem, da funkcija vrača spremenjen rezultat, saj se bodo spremembe odražale že preko podanega argumenta. Koda je torej sledeča:

```
def dodaj_AT_seznam(zaporedje1, zaporedje2):
    for baza in zaporedje2:
        if baza in 'AT':
            zaporedje1.append(baza)
```

Tokrat funkcija ne vrača ničesar uporabnega, zato njenega rezultata nima smisla ničemur prirejati. Vse kar potrebujemo je klic funkcije z ustreznimi argumenti:

```
>>> zaporedje1 = ['A', 'T', 'C', 'G']
>>> zaporedje2 = ['A', 'A', 'T', 'G', 'G', 'A', 'A', 'T', 'G', 'G']
>>> dodaj_AT_seznam(zaporedje1, zaporedje2)
```

Prepričajmo se, če je vrednost spremenljivke `zaporedje1` res spremenjena:

```
>>> zaporedje1
['A', 'T', 'C', 'G', 'A', 'A', 'T', 'A', 'A', 'T']
```

## 9 Slovarji

### 9.1 Zakaj slovarji?

Zdaj smo se že поблиže spoznali z različnimi sekvenčnimi podatkovnimi tipi, med katere smo uvrstili nize, sezname in terke. V spremenljivke, ki so pripadale tem tipom, smo lahko shranili več podatkov, pri čemer je bil podatek vedno povezan z določenim indeksom. Če se osredotočimo na sezname (kar bomo povedali velja sicer tudi za nize in terke), lahko rečemo, da so podatki v njih na nek način urejeni, ko smo v sezname dodajali nove podatke, smo te ponavadi dodajali na konec seznama, iskanje pa je potekalo tako, da smo se morali z zanko `for` sprehoditi čez celoten seznam in iskati dokler elementa nismo našli. Sezname smo lahko tudi sortirali po nekem ključu (recimo glede na relacijo  $<$ ). Pri tem je bil rezultat sortiranja tak, da smo na manjših indeksih dobili elemente, ki so imeli manjšo vrednost od tistih na večjem indeksu, pri čemer je bila upoštevana uporabljena relacija.

V določenih primerih pa je bolj priročno, če lahko do vrednosti v spremenljivki dostopamo še preko česa drugega kot njenega indeksa. Pomislimo npr. na telefonski imenik števil, kjer lahko do telefonske številke neke osebe, dostopamo preko imena te osebe. Rekli bi lahko, da je ime osebe *ključ* preko katerega dostopamo do telefonske številke oziroma *vrednosti*. Na podoben način iščemo gesla v Slovarju slovenskega knjižnjega jezika (ali pač kakšnem drugem slovarju), kjer kot ključ nastopa iskana beseda oziroma geslo, kot vrednost pa razlaga iskane besede. Za razlago določene besede nam torej ni potrebno preiskati celotnega slovarja, ampak njeno razlago poiščemo preko gesla, ki ga je v slovarju načeloma enostavno (in predvsem hitro) najti.

V takih primerih lahko uporabimo v Python že vgrajen podatkovni tip `dict` (angl. *dictionary*) oziroma po slovensko kar *slovar*. V primeru slovarjev posamezna *vrednost* (angl. *value*) ni vezana na določen indeks, ampak je povezana na *ključ* (angl. *key*). Elemente slovarja torej vedno podajamo kot pare *ključ:vrednost*. Kakšna je prednost takega načina shranjevanja podatkov? Uporabna je takrat, ko ključ, po katerih shranjujemo in kasneje iščemo vrednosti, poznamo. V tem primeru je iskanje zelo hitro, saj nam ni potrebo preiskati celotnega slovarja, ampak slovarju zgolj podamo ključ in do vrednosti pridemo takoj. Operacija iskanja je torej zelo hitra v primerjavi s seznamami ali terkami.

## 9.2 Kako uporabljamo slovarje?

Slovarje zapisujemo z zavitimi oklepaji, torej { in }, pri čemer elemente naštejemo kot pare ključ in vrednost ločene z dvopičjem (:). Prazen slovar bi naredili takole

```
>>> prazen_slovar = {}
```

Slovar, ki vsebuje enostaven telefonski imenik bi lahko zapisali kot

```
>>> imenik = {"Janez": "083455544",
              "Ana": "084566732",
              "Nika": "099563123"}
```

V tem primeru so ključi slovarja nizi, ki predstavljajo imena oseb, preko katerih lahko pridemo vrednosti, ki v tem primeru predstavljajo telefonske številke zapisane kot nizi.

Mimogrede, slovar je spremenljiv podatkovni tip, kot smo zapisali že v tabeli 8.1.

## 9.3 Iskanje vrednosti

Telefonsko številko osebe lahko torej najdemo tako, da podamo njeno ime, kar je smiselno, saj imena svojih prijateljev ponavadi poznamo na pamet, njihovih telefonskih števil pa ne. Telefonsko številko od Janeza lahko npr. dobimo tako, da slovar indeksiramo po ključu "Janez":

```
>>> imenik["Janez"]
083455544
```

Indeksiranje se torej v primeru slovarjev namesto po indeksih izvaja po ključih. Kaj pa če bi želeli poiskati telefonsko številko od Dejana. Potem bi slovar indeksirali po ključu "Dejan", kar pa nam v konkretnem primeru vrne napako:

```
>>> imenik["Dejan"]
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    imenik["Dejan"]
KeyError: 'Dejan'
```

Problem je v tem, da ključa "Dejan" v našem slovarju (še) ni, zato preko njega slovarja ne moremo indeksirati (tako kot nismo mogli indeksirati seznam z indeksom, ki ga v seznamu ni bilo). Obstoja ključa v slovarju lahko preverimo z operatorjem in:

```
>>> "Dejan" in imenik
False
>>> "Nika" in imenik
True
```



Preden do določenega ključa v slovarju dostopamo je torej smiselno, da preverimo, če ključ v slovarju sploh obstaja.

**Zgled 35** *Napiši funkcijo **poisci**, ki kot argument sprejme imenik in ime osebe. Če imena ni v imeniku naj funkcija vrne vrednost **False**, sicer pa njeno telefonsko številko.*

**Rešitev 35** *Rešitev mora pred indeksiranjem po podanem ključu, preveriti, če ključ v slovarju obstaja. V nasprotnem primeru vrne vrednost **False**.*

```
def poisci(imenik, ime):
    if ime in imenik:
        return imenik[ime]
    else:
        return False
```

## 9.4 Dodajanje in spreminjanje vrednosti

Indeksiranje po ključih, ki jih v slovarju ni, pa je v določenih primerih dovoljeno, in sicer takrat, ko želimo v slovar dodati nov par ključ, vrednost. Dodajanje novega elementa lahko torej izvedemo tako, da slovar indeksiramo po novem (neobstoječem) ključu in takemu indeksiranjju priredimo vrednost, ki jo želimo s tem ključem povezati. Če bi npr. želeli v slovar dodati Dejana in z njim povezati neko telefonsko številko, npr. 089543678, bi to naredili takole:

```
>>> imenik["Dejan"] = "089543678"
```

V tem primeru napake nismo dobili, v slovarju pa se je ponavil nov par ključ, vrednost.

```
>>> imenik
{'Janez': '083455544', 'Ana': '084566732',
 'Nika': '099563123', 'Dejan': '089543678'}
```

Kaj pa se zgodi, če poskusimo v slovar dodati še enega Dejana? Ker v slovarju do vrednosti dostopamo preko ključev, morajo biti ključi enolični, kar pomeni, da se posamezen ključ lahko v slovarju pojavi največ enkrat. Če torej naredimo prireditev vrednosti preko ključa, ki v slovarju že obstaja, bomo s tem izvedli spreminjanje vrednosti, ki je povezana s tem ključem. Prireditev

```
>>> imenik["Dejan"] = "000000000"
```

bo torej spremenila telefonsko številko Dejana na 000000000.

```
>>> imenik
{'Janez': '083455544', 'Ana': '084566732',
 'Nika': '099563123', 'Dejan': '000000000'}
```

Če bi se želeli torej omejiti samo na dodajanje elementov, bi lahko prej preverili, če ključ preko katerega dodajamo v slovarju že obstaja in dodajanje izvedli le, če takega ključa še ni.

**Zgled 36** *Napiši funkcijo `dodaj`, ki kot argumente prejme imenik, ime osebe in telefonsko številko osebe. Osebo in njeno telefonsko številko naj v slovar doda samo v primeru, če te osebe še ni v slovarju. Sicer naj izpiše, da ta oseba že obstaja.*

**Rešitev 36** *V funkciji bomo tokrat izvedli prirejanje vrednosti, samo če podanega ključa še ni v slovarju. Poraja pa se dodatno vprašanje. Ali mora funkcija spremenjen slovar vračati? Odgovor je ne, saj je slovar spremenljiv podatkovni tip in spremembe slovarja, ki jih bomo v funkciji naredili in ki ga je funkcija prejela kot argument, se bodo odražale tudi izven funkcije.*

```
def dodaj(imenik, ime, stevilka):
    if ime not in imenik:
        imenik[ime] = stevilka
    else:
        print("Ta oseba že obstaja")
```

Če bi se želeli samo na spreminjanje elementov, bi morali spremeniti pogoj v stavku `if`, tako da številko prirejamo samo v primeru, če je ključ v slovarju že vsebovan.

**Zgled 37** *Napiši funkcijo `spremeni`, ki kot argumente prejme imenik, ime osebe in telefonsko številko osebe. Telefonsko številko naj spremeni samo v primeru, če je oseba v imeniku že vsebovana. Sicer naj izpiše, da te osebe v imeniku ni.*

**Rešitev 37** *Tokrat bomo v funkciji izvedli prirejanje vrednosti samo v primeru, če podan ključ v slovarju je. V nasprtnem primeru ne bi izvajali spreminjanja vrednosti za ključem, ampak bi izvajali dodajanje novega para ključ, vrednost.*

```
def spremeni(imenik, ime, stevilka):
    if ime in imenik:
        imenik[ime] = stevilka
    else:
        print("Te osebe ni v imeniku")
```

V določenih primerih, npr. pri štetju pojavitev nečesa, moramo v slovar dodati nov ključ, če tega ključa v slovarju še ni, ali spreminjati nanj vezano vrednost, če ta ključ v slovarju že obstaja. Poglejmo si sledeč primer:

**Zgled 38** *Napiši funkcijo `prestěj_baze`, ki kot argument prejme nukleotidno zaporedje baz zapisano kot niz. Funkcija naj vrne slovar, ki za posamezno bazo vsebuje število ponovitev.*

**Rešitev 38** Za razliko od prej bo funkcija slovar vračala, saj ji ga kot argument nismo podali. V programu bi lahko predpostavljali, da delamo samo z bazami A, T, C in G. Tako bi si na začetku naredili slovar, v katerem kot ključi nastopajo oznake baz, nanje pa so vezane vrednosti 0. Takole:

```
baze = {'A':0, 'T':0, 'C':0, 'G':0}
```

Slabost takega pristopa je ta, da smo se omejili samo na ključe A, T, C in G. Kaj pa če kot vhod dobimo RNA zaporedje? Ali pa če se v zaporedju pojavi še kakšna oznaka, ki je nismo predvideli?

Boljši pristop bi bil, da na začetku naredimo prazen slovar in ko v nizu najdemo bazo, ki je v slovarju še ni, nanjo vežemo vrednost 1 (če smo jo našli v nizu prvič, potem to pomeni, da smo jo našli enkrat). V primeru, da v nizu najdemo bazo, ki v slovarju že obstaja, število njenih pojavitev povečamo za 1.

```
def prestej_baze(zaporedje):
    baze = {} # prazen slovar
    for baza in zaporedje:
        if baza not in baze: # baza v slovarju?
            baze[baza] = 1 # ne
        else:
            baze[baza] += 1 # da
    return baze
```

## 9.5 Brisanje vrednosti

V določenih primerih želimo elemente iz slovarjev tudi brisati. To lahko naredimo z besedico `del`, ki smo jo srečali že pri brisanju elementov iz seznama. Podobno kot pri seznamih tudi iz slovarjev brišemo z indeksiranjem vrednosti, ki jo želimo izbrisati:

```
del slovar[kljuc]
```

Tudi tokrat je smiselno, da pred brisanjem preverimo, če podan ključ v slovarju obstaja (brisanje po neobstoječem ključu bo spet vrnilo napako).

**Zgled 39** Napiši funkcijo *izbrisi*, ki kot argumente prejme imenik in ime osebe, ki jo želimo iz imenika izbrisati. Brisanje naj se izvede samo v primeru, ko oseba v imeniku obstaja. Sicer naj funkcija izpiše, da te osebe v imeniku ni.

**Rešitev 39** Spet bomo najprej preverili obstoj ključa, nato pa izvedli brisanje.

```
def izbrisi(imenik, ime):
    if ime in imenik:
        del imenik[ime]
```

```

else:
    print("Te osebe ni v imeniku")

```

## 9.6 Ključi in vrednosti

Posamezen ključ se lahko v slovarju pojavi največ enkrat. Ključi so torej enolični identifikatorji, preko katerih pridemo do posamezne vrednosti. Za ključe pa velja tudi to, da morajo biti nespremenljivi. Zakaj? Vrednost, ki je vezana na posamezen ključ, se zapiše na lokacijo v pomnilniku, ki je določena s preslikavo ključa (angl. *hash function*). Če ključ spreminjamo, se bo spremenila tudi vrednost preslikave. Za pravilno delovanje torej ključev ne smemo spreminjati (ko so enkrat shranjeni v slovarju). Zato je smiselno, da so ključi nespremenljivi podatki. Za vrednosti ni nobene omejitve – uporabimo lahko poljuben podatkovni tip vključno z drugim, vgnezdenim, slovarjem.

Sicer lahko do vseh ključev v slovarju pridemo z uporabo metode `keys`, do vseh vrednosti z uporabo metode `values`, do vseh parov pa z uporabo metode `items`.

```

>>> imenik.keys()
dict_keys(['Janez', 'Ana', 'Nika', 'Dejan'])
>>> imenik.values()
dict_values(['083455544', '084566732', '099563123',
'089543678'])
>>> imenik.items()
dict_items([('Janez', '083455544'), ('Ana', '084566732'),
('Nika', '099563123'), ('Dejan', '089543678')])

```

Metode vračajo nekaj kar je zelo podobno seznamom. Pri metodi `keys` tako dobimo seznam ključev, pri metodi `values` seznam vrednosti, pri metodi `items` pa seznam terk. Nad rezultatom, ki ga posamezna metoda vrne, se lahko sprehajamo z zanko `for`. Zanko `for` pa lahko izvedemo tudi direktno nad slovarjem, pri čemer se bomo tako sprehajali čez ključe slovarja:

```

>>> for k in imenik():
    print(k)
Janez
Ana
Nika
Dejan

```

Preko ključev seveda lahko dostopamo tudi do vrednosti.

**Zgled 40** *Napiši funkcijo `izpisi_imenik`, ki kot argument prejme imenik. Funkcija naj izpiše vsebino imenika, tako da se vsak vnos nahaja v svoji vrstici, ime in telefonska številka pa naj bosta ločena z dvopičjem.*

**Rešitev 40** V zanki `for` se lahko sprehajamo direktno čez slovar (sprehod čez ključe), čez ključe preko metode `keys` ali pa čez pare preko metode `items`. Uporabimo tokrat zadnjo možnost.

```
def izpisi_imenik(imenik):
    for ime, stevilka in slovar.items():
        print(ime, ":", stevilka)
```

Podoben sprehod bi lahko naredili tudi kadar npr. iščemo največjo vrednost v slovarju.

**Zgled 41** Napiši funkcijo `naj_baza`, ki kot argument sprejme zaporedje baz, vrne pa ime baze, ki se v zaporedju pojavi največkrat. Pri tem si pomagaj s funkcijo `prestej_baze`.

**Rešitev 41** Najprej bomo poklicali funkcijo `prestej_baze`, ki bo iz niza naredila slovar pojavitev baz. V naslednjem koraku moramo najti bazo, ki ima največ pojavitev. To bomo naredili na podoben način, kot smo izvedli iskanje največjega (ali najmanjšega) elementa v seznamu – s sprehodom z uporabo zanke `for`.

```
def naj_baza(zaporedje):
    baze = prestej_baze(zaporedje)

    naj_B = "" # naj baza
    M = 0 # stevilo pojavitev
    for baza in baze:
        pojavitev = baze[baza]
        if pojavitev > M:
            naj_B = baza
            M = pojavitev
    return naj_B
```

## 9.7 Imenski prostor in slovarji

Poleg tega, da slovarje lahko navadni smrtniki uporabljamo za reševanje svojih programerskih problemov, slovarje za svoje delovanja uporablja tudi Python sam. Omenili smo že, da Python v svojem imenskem prostoru vsebuje vsa imena, preko katerih lahko dostopamo do vrednosti spremenljivk, funkcij in še česa drugega. Do imenskega prostora lahko pridemo s funkcijo `globals`. Poglejmo si, kaj ta funkcija vrne:

```
>>> globals()
{'__name__': '__main__', '__doc__': None, '__package__': None,
```

```
'__loader__': <class '_frozen_importlib.BuiltinImporter'>,
'__spec__': None, '__annotations__': {}, '__builtins__':
<module 'builtins' (built-in)>}
```

Funkcija vrne imena v imenskem prostoru in vrednosti, ki se za imeni skrivajo. Definirajmo novo spremenljivko in pokličimo gornjo funkcijo še enkrat.

```
>>> x = 1
>>> globals()
{'__name__': '__main__', '__doc__': None, '__package__': None,
'__loader__': <class '_frozen_importlib.BuiltinImporter'>,
'__spec__': None, '__annotations__': {}, '__builtins__':
<module 'builtins' (built-in)>, 'x': 1}
```

V imenskem prostoru se je zdaj očitno pojavilo ime `x`, za katerem se skriva vrednost 1. Če pogledamo bolj natančno, lahko vidimo, da je funkcija `globals` vrnila slovar.

```
>>> type(globals())
<class 'dict'>
```

V tem slovarju so ključi imena spremenljivk, funkcij itd., vrednosti pa tisto, kar se za imeni skriva. Ko v ukazno vrstico torej napišem `x` bo Python v svojem slovarju, ki predstavlja imenski prostor, pogledal, če tam obstaja ime `x` in vrnil vrednost, ki se za tem imenom skriva. To bi lahko naredili tudi takole:

```
>>> globals()['x']
1
```

Vsakič, ko definiram novo funkcijo ali spremenljivko, se njeno ime in vrednost doda v Pythonov slovar, podobno kot smo prej ključe in vrednosti v slovarje dodajali mi. Kaj se zgodi, ko določeni spremenljivki vrednost *povežimo* - vrednost za ključem z imenom te spremenljivke se enostavno poveži, saj lahko posamezen ključ v slovarju nastopa največ enkrat.

## 10 Množice

### 10.1 In še množice

Zadnji izmed vgrajenih podatkovnih tipov, ki si ga moramo pogledati, so množice. Množice za predstavitev podatkov uporabljamo takrat, ko želimo, da posamezen podatek obravnavamo največ enkrat. Ker nam Python poleg tega nad množicami omogoča izvedbo osnovnih operacij, kot so unija, presek in razlika, množice zelo spominjajo na matematične množice.

### 10.2 Uporaba množic

Množice podobno kot slovarje zapisujemo v zavite oklepaje, znotraj katerih naštejemo elemente. Množico elementov 1, 2 in 3, bi torej naredili takole:

```
>>> mnozica = {1,2,3}
>>> mnozica
{1, 2, 3}
>>> type(mnozica)
<class 'set'>
```

Kljub temu, da množice uporabljajo podoben zapis kot slovarji, ju Python med seboj brez problemov loči, saj slovarji za razliko od množic vsebujejo pare ključ: vrednost. Do težave pride le takrat, ko je množica oziroma slovar prazen. Do praznega slovarja pridemo tako, da podamo zavite oklepaje brez elementov:

```
>>> slovar = {}
>>> slovar
{}
>>> type(slovar)
<class 'dict'>
```

Do prazne množice pridemo z uporabo funkcije `set`, ki jo pokličemo brez argumentov:

```
>>> mnozica = set()
>>> mnozica
```

```
set()
>>> type(mnozica)
<class 'set'>
```

### 10.3 Omejitve pri uporabi množic

Elementi množic imajo zelo podobne omejitve kot ključi slovarji, za katere prav tako velja, da lahko posamezno vrednost vsebujejo največ enkrat. Prav tako kot za ključe slovarjev tudi za množice velja, da lahko vsebujejo le nespremenljive podatkovne tipe.

Množice predstavljajo neurejeno strukturo, kar pomeni, da vrstni red elementov množice ni pomemben in tudi ni določen. Elementi torej niso vezani na indekse, zato množic ne moremo indeksirati in nad njimi delati rezine:

```
>>> mnozica = {1,2,3}
>>> mnozica[0]
TypeError: 'set' object is not subscriptable
```

Prav tako nad množicami ne moremo izvajati aritmetičnih operacij, kot smo jih npr. lahko izvajali nad nizi:

```
>>> {1,2,3}*3
TypeError: unsupported operand type(s) for *: 'set' and 'int'
>>> {1,2,3} + {4,5,6}
TypeError: unsupported operand type(s) for +: 'set' and 'set'
```

### 10.4 Osnovne operacije nad množicami

Kaj pa pravzaprav potem z množicami sploh lahko počnemo. Ko množico enkrat imamo se lahko čez njo sprehajamo z zanko `for`:

```
>>> mnozica = {1,2,3}
>>> for element in mnozica:
    print(element)
1
2
3
```

Poleg tega lahko preverjamo ali množica določen element vsebuje (ali ne) z operatorjema vsebovanosti:

```
>>> mnozica = {1,2,3}
>>> 1 in mnozica
True
```



```
>>> 2 not in mnozica
False
>>> 4 in mnozica
False
```

Številu elementov v množici pravimo tudi moč množice, do katere pridemo enostavno s klicem vgrajene funkcije `len`

```
>>> len({1,2,3})
3
```

**Zgled 42** *Napiši funkcijo `razlicni`, ki kot argument sprejme niz, kot rezultat pa vrne število različnih znakov, ki v nizu nastopajo*

**Rešitev 42** *Najprej bomo niz pretvorili v množico s funkcijo `set`. Ker množica posamezen element vsebuje največ enkrat, se bomo s tem znebili vseh potencialnih ponovitev znakov. Potem samo še izračunamo in vrnemo moč množice.*

```
1 def razlicni(niz):
2     mnozica = set(niz) # odstranitev duplikatov
3     return len(mnozica) # moc mnozice
```

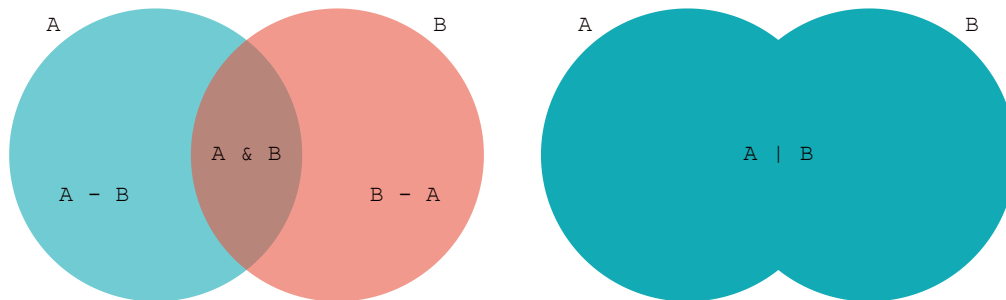
Tudi primerjalni operatorji so zdaj prilagojeni matematični interpretaciji množic. Ali je prva množica podmnožica druge, lahko npr. ugotovimo z operatorjem `<=`:

```
>>> {1,2,3} <= {1,2,3,4}
True
```

## 10.5 Presek, unija in razlika

Primeri najbolj tipičnih operacij, ki jih prikazujejo diagrami na sliki 10.1, so seveda presek ( $A \& B$ ), unija ( $A \mid B$ ) in razlika ( $A - B$ ). Primer uporabe gornjih operacij je sledeč:

```
>>> {1,2,3} & {3,4,5} # presek
{3}
>>> {1,2,3} | {3,4,5} # unija
{1,2,4,5}
>>> {1,2,3} - {3,4,5} # razlika
{1,2}
```



**Slika 10.1** Vennova diagrama, ki ponazarjata osnovne operacije nad množicami. Slika na levi prikazuje razliko ( $A-B$  in  $B-A$ ) in presek ( $A \& B$ ) med množicama A in B. Slika na desni prikazuje unijo ( $A | B$ ) med množicama A in B

## 10.6 Metode množic: dodajanje in brisanje elementov

Dodajanje elementa v množico lahko izvedemo z uporabo metode `add`, ki kot argument sprejme element, ki ga želimo dodati:

```
>>> mnozica = {1,2,3}
>>> mnozica.add(4)
>>> mnozica
{1, 2, 3, 4}
```

V primeru, da dodajamo element, ki v množici že obstaja, metoda ne naredi ničesar. Dodajanje bi lahko izvedli tudi z operatorjem `|=`, ki naredi unijo množice z neko drugo množico. V tem primeru lahko dodamo več elementov naenkrat:

```
>>> mnozica = {1,2,3}
>>> mnozica |= {4,5}
>>> mnozica
{1, 2, 3, 4, 5}
```

Brisanje elementov iz množice vršimo z metodo `remove`. Ta v primeru neobstoja elementa vrne napako. Uporabimo lahko tudi metodo `discard`, ki v primeru neobstoja napake ne vrne, sicer pa deluje enako kot metode `remove`:

```
>>> mnozica = {1,2,3}
>>> mnozica.remove(3)
>>> mnozica
{1, 2}
>>> mnozica.remove(3)
KeyError: 3
>>> mnozica.discard(2)
>>> mnozica
{1}
```

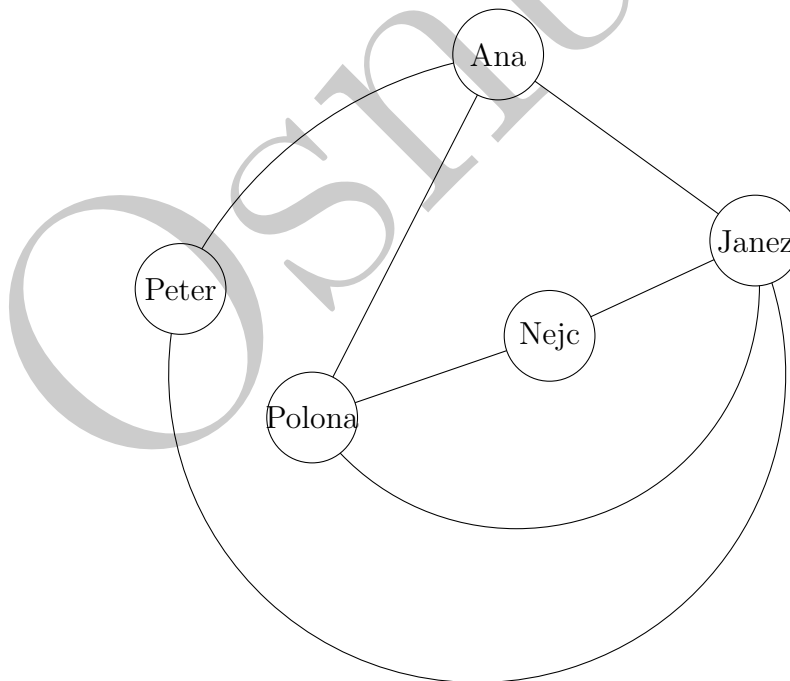
```
>>> mnozica.discard(2)
```

Podobno kot prej, lahko uporabimo tudi operator `-=`, ki naredi razliko med množico in neko drugo množico in to priredi izhodiščni množici:

```
>>> mnozica = {1,2,3}
>>> mnozica -= {2,3}
>>> mnozica
{1}
```

## 10.7 Zgled uporabe množic

Ker so bili sprotni zgledi v tem poglavju relativno skopi, bomo to nadoknadili z malo daljšim zgledom, s katerim bomo povadili tudi slovarje in mogoče še kaj. Zamislimo si, da bi radi opazovali omrežje prijateljev oziroma povezanost ljudi preko relacije *prijateljstvo*, bodisi v resničnem ali pa virtualnem življenju. Tako omrežje lahko predstavimo z *neusmerjenim grafom*, v katerem vozlišča predstavljajo imena oseb, povezave med vozlišči pa prijateljstva med osebami. Te povezave so neusmerjene, saj prijateljstvo deluje v obe smeri: če je oseba 1 prijatelj osebe 2, je tudi oseba 2 prijatelj osebe 1. Primer grafa prijateljstev prikazuje slika 10.2.



**Slika 10.2** Primer grafa omrežja prijateljstev.

Da se bomo lahko lotili nadaljnjih analiz omrežja prijateljstev, moramo tega najprej predstaviti v obliko, ki jo bomo lahko zapisali v Pythonu. V nadaljevanju bomo napisali par funkcij, ki jih bomo lahko uporabili pri posodabljanju omrežja in njegovi analizi.

**Zgled 43** *Izberi si podatkovno strukturo, ki jo boš lahko uporabil pri predstavitvi in analizi omrežja prijateljstev ter z njo predstavi omrežje na sliki 10.2. Pri izbiri strukture upoštevaj to, da želiš nad omrežjem vršiti funkcije, kot so dodajanje prijateljev, brisanje prijateljev, iskanje skupnih prijateljev, iskanje osebe z največ prijatelji ipd.*

**Rešitev 43** *Omrežje bi lahko predstavili na različne načine, a vendar težimo k tisti predstavitvi, ki nam bo pri nadaljnjih operacijah nad omrežjem prihranila največ dela, hkrati pa bodo operacije nad omrežjem potekale kar se da hitro. Začnimo z mogoče najbolj intuitivno, a ne preveč dobro predstavitvijo, in sicer s seznamom terk. V tem primeru vsaka terka v seznamu vsebuje ime osebe in seznam njenih prijateljev. Omrežje s slike 10.2 bi na tak način zapisali kot:*

```
prijatelji = [('Ana', ['Janez', 'Peter', 'Polona']),
              ('Janez', ['Ana', 'Nejc', 'Peter', 'Polona']),
              ('Nejc', ['Janez', 'Polona']),
              ('Peter', ['Ana', 'Janez']),
              ('Polona', ['Ana', 'Janez', 'Nejc'])]
```

*Zakaj predstavitev ni najboljša? Zatakne se nam že pri izpisovanju vseh prijateljev podane osebe, saj že ta zahteva iskanje podane osebe s sprehodom čez (v najslabšem primeru) celoten seznam prijateljev. Če je ta seznam dolg, je lahko ta operacija časovno zelo potratna. Stvar bi lahko izboljšali, če bi namesto seznama terk uporabili kar slovar, v katerem so ključi imena oseb, vrednosti pa seznamami prijateljev. Takole:*

```
prijatelji = {'Ana': ['Janez', 'Peter', 'Polona'],
              'Janez': ['Ana', 'Nejc', 'Peter', 'Polona'],
              'Nejc': ['Janez', 'Polona'],
              'Peter': ['Ana', 'Janez'],
              'Polona': ['Ana', 'Janez', 'Nejc']}
```

*S tem smo rešili problem iskanja prijateljev podane osebe, saj do teh pridemo z enostavnim indeksiranjem po imenu osebe, ki nas zanima. Še vedno pa je problematično npr. iskanje skupnih prijateljev dveh oseb, saj zahteva ugnezdено zanko po seznamih prijateljev teh dveh oseb. Spet je ta operacija lahko časovno potratna, če so ti seznammi dolgi, kar v socialnih omrežjih zagotovo ni izključeno. Temu problemu bi se lahko izognili tako, da prijatelje podane osebe predstavimo z množicami, nad katerimi so operacije kot je presek bolj učinkovite. Poleg tega glede na podan opis problema vrstni red prijateljev ni pomemben, posamezna oseba pa kot prijatelj druge osebe nastopa največ enkrat. Predstavitev bi bila torej sledeča:*

```
prijatelji = {'Ana': {'Janez', 'Peter', 'Polona'},
              'Janez': {'Ana', 'Nejc', 'Peter', 'Polona'},
              'Nejc': {'Janez', 'Polona'},
              'Peter': {'Ana', 'Janez'},
              'Polona': {'Ana', 'Janez', 'Nejc'}}
```

**Zgled 44** Napiši funkcijo `prijatelji_od`, ki kot argument sprejme omrežje prijateljstev in ime osebe in izpiše vse prijatelje podane osebe. V primeru, da podane osebe ni v omrežju, naj funkcija to izpiše.

**Rešitev 44** Kot smo videli že prej, je zaradi izbrane predstavitve omrežja, iskanje prijateljev podane osebe enostavno. Preveriti moramo samo, če oseba v omrežju obstaja.

```
1 def prijatelji_od(prijatelji, oseba):
2     if oseba not in prijatelji: # ali je oseba v omrežju?
3         print("Ta oseba ne obstaja!")
4     else:
5         print(prijatelji[oseba])
```

**Zgled 45** Napiši funkcijo `dodaj_osebo`, ki kot argument sprejme omrežje prijateljstev in ime osebe ter podano osebo doda v omrežje prijateljstev s prazno množico prijateljev.

**Rešitev 45** V omrežje bomo torej dodali novo osebo (nov ključ), na katerega bomo vezali prazno množico. Smiselno je tudi, da preverimo, če oseba v omrežju že obstaja. Ali mora funkcija vračati spremenjen slovar? Odgovor je seveda ne, saj je slovar spremenljiv podatkovni tip in se bo spreminjanje tega znotraj funkcije odražalo tudi izven funkcije.

```
1 def dodaj_osebo(prijatelji, oseba):
2     if oseba not in prijatelji: # dodaj samo, ce je se ni
3         prijatelji[oseba] = set() # prazna mnozica
```

**Zgled 46** Napiši funkcijo `spoprijatelji`, ki kot argument sprejme omrežje prijateljstev in imena dveh oseb, ki sta se spoprijateljili. Če sta osebi že v prijateljstvu, naj funkcija to izpiše. Če katerekoli izmed oseb še ni v omrežju, naj to osebo doda preko funkcije `dodaj_osebo`.

**Rešitev 46** Spoprijateljjevanje bomo naredili tako, da bomo prvo osebo dodali v množico prijateljev druge osebe in obratno. Pri tem lahko uporabimo metodo `add`. Še prej bomo po potrebi posamezno osebo dodali v omrežje, če je tam še ni.

```

1 def spoprijatelji(prijatelji, oseba1, oseba2):
2     if oseba1 not in prijatelji:
3         dodaj_osebo(prijatelji, oseba1)
4     if oseba2 not in prijatelji:
5         dodaj_osebo(prijatelji, oseba2)
6
7     # spoprijatelji samo, ce se nista prijatelja
8     if oseba2 not in prijatelji[oseba1]:
9         # dodaj oseba2 med prijatelje oseba1
10        prijatelji[oseba1].add(oseba2)
11        # dodaj oseba1 med prijatelje oseba2
12        prijatelji[oseba2].add(oseba1)
13    else:
14        print("Osebi sta ze v prijateljstvu")

```

Primer klika:

```

>>> spoprijatelji(prijatelji, 'Ana', 'Nejc')
>>> prijatelji
{'Ana': {'Janez', 'Peter', 'Nejc', 'Polona'},
 'Janez': {'Peter', 'Ana', 'Nejc', 'Polona'},
 'Nejc': {'Janez', 'Ana', 'Polona'},
 'Peter': {'Janez', 'Ana', 'Polona'},
 'Polona': {'Janez', 'Peter', 'Ana', 'Nejc'}}
>>> spoprijatelji(prijatelji, 'Ana', 'Nejc')
Osebi sta ze v prijateljstvu

```

**Zgled 47** Napiši funkcijo *skregaj*, ki kot argument sprejme omrežje prijateljstev in imena dveh oseb, ki sta se skregali. Če osebi nista v prijateljstvu, naj funkcija to izpiše. Če katerekoli izmed oseb še ni v omrežju, naj funkcija to izpiše.

**Rešitev 47** Pri skreganju bomo prvo osebo odstranili iz množice prijateljev druge osebe in obratno. Pri tem lahko uporabimo metodo *remove*. Še prej bomo preverili, če osebi sploh sta v omrežju in če sta v relaciji prijateljstva.

```

1 def skregaj(prijatelji, oseba1, oseba2):
2     if oseba1 not in prijatelji or oseba2 not in prijatelji:
3         print("Ene izmed oseb ni v omrežju!")
4         return # koncaj in ne vrni nic
5     # ce oseba1 ni prijatelj oseba2, velja tudi obratno
6     if oseba1 not in prijatelji[oseba2]:
7         print("Osebi nista prijatelja!")
8         return # koncaj in ne vrni nic
9

```

```

10     # odstrani prijateljstvo
11     prijatelji[oseba1].remove(oseba2)
12     prijatelji[oseba2].remove(oseba1)

```

Primer klica:

```

>>> skregaj(prijatelji, 'Ana', 'Nejc')
{'Ana': {'Janez', 'Peter', 'Polona'},
 'Janez': {'Peter', 'Ana', 'Nejc', 'Polona'},
 'Nejc': {'Janez', 'Polona'},
 'Peter': {'Janez', 'Ana', 'Polona'},
 'Polona': {'Janez', 'Peter', 'Ana', 'Nejc'}}
>>> skregaj(prijatelji, 'Ana', 'Nejc')
Osebi nista prijatelja!

```

Zdaj pa se lotimo še analize omrežja.

**Zgled 48** *Napiši funkcijo `najbolj_popularni`, ki kot argument sprejme omrežje prijateljstev in vrne ime osebe, ki ima največ prijateljev.*

**Rešitev 48** *Spet rešujemo nalogo z iskanjem največjega elementa v slovarju, tokrat glede na dolžino množice, ki se skriva za posameznim ključem. Rešitev se ne bo dosti razlikovala od programa, ki je iskal oznako nukleotida, ki se v nukleotidnem zaporedju pojavi največkrat.*

```

1 def najbolj_popularni(prijatelji):
2     naj_oseba = ""
3     naj_prijateljev = 0
4
5     for oseba in prijatelji: # sprehod cez kljuce
6         prijateljev = len(prijatelji[oseba]) # stevilo prijateljev
7         if prijateljev > naj_prijateljev:
8             naj_oseba = oseba
9             naj_prijateljev = prijateljev
10    return naj_oseba

```

Primer klica:

```

>>> najbolj_popularni(prijatelji)
'Janez'

```

Rešitev predpostavlja, da je enako najbolj popularna samo ena oseba, kar lahko rešimo z manjšo dopolnitvijo, tako da si najbolj popularne osebe shranimo v seznam.

```

1 def najbolj_popularni(prijatelji):
2     naj_osebe = []

```

```

3     naj_prijateljev = 0
4
5     for oseba in prijatelji: # sprehod cez kljuce
6         prijateljev = len(prijatelji[oseba]) # stevilo prijateljev
7         if prijateljev > naj_prijateljev:
8             naj_osebe = [oseba]
9             naj_prijateljev = prijateljev
10        elif prijateljev == naj_prijateljev:
11            naj_osebe.append(oseba)
12    return naj_osebe

```

*Primer klica:*

```

>>> spoprijatelji(prijatelji, 'Polona', 'Peter')
>>> najbolj_popularni(prijatelji)
['Janez', 'Polona']

```

**Zgled 49** Napiši funkcije *skupni\_prijatelji*, *vsaj\_od\_enega* in *brez\_njegovih*.

Vse tri funkcije naj prejmejo omrežje prijateljstev in dve osebi ter izpišejo njune skupne prijatelje, prijatelje vsaj od ene izmed podanih oseb ter tiste, ki so prijatelji prve osebe ne pa prijatelji druge. V primeru, da v omrežju ni obeh podanih oseb, funkcije izpišejo, da vsaj ene podane osebe v omrežju ni.

**Rešitev 49** V tej rešitvi bomo povadili uporabo preseka (skupni prijatelji), unije (vsaj od enega) ter razlike (brez njegovih).

```

1 def skupni(prijatelji, oseba1, oseba2):
2     if oseba1 not in prijatelji or oseba2 not in prijatelji:
3         print("Vsaj_ene_osebe_ni_v_omrezju!")
4     else:
5         # presek
6         print(prijatelji[oseba1] & prijatelji[oseba2])
7
8 def vsaj_od_enega(prijatelji, oseba1, oseba2):
9     if oseba1 not in prijatelji or oseba2 not in prijatelji:
10        print("Vsaj_ene_osebe_ni_v_omrezju!")
11    else:
12        # unija
13        print(prijatelji[oseba1] | prijatelji[oseba2])
14
15 def brez_njegovih(prijatelji, oseba1, oseba2):
16     if oseba1 not in prijatelji or oseba2 not in prijatelji:
17        print("Vsaj_ene_osebe_ni_v_omrezju!")
18    else:

```



```

19      # razlika
20      print(prijatelji[oseba1] - prijatelji[oseba2])

```

*Primer klica:*

```

>>> skupni(prijatelji, 'Ana', 'Janez')
{'Peter', 'Polona'}
>>> skupni(prijatelji, 'Ana', 'Jan')
Vsaj ene osebe ni v omrezju!
>>> vsaj_od_enega(prijatelji, 'Ana', 'Janez')
{'Janez', 'Peter', 'Ana', 'Nejc', 'Polona'}
>>> brez_njegovih(prijatelji, 'Ana', 'Janez')
{'Janez'}

```

**Zgled 50** Napiši funkcijo `prijatelji_prijateljev`, ki kot argumenta sprejme omrezje prijateljev in ime osebe ter izpiše prijatelje prijateljev podane osebe, pri čemer naj iz izpisa izpusti prijatelje podane osebe ter podano osebo samo.

**Rešitev 50** Tokrat se bomo sprehodili čez prijatelje podane osebe z uporabo zanke `for` ter s prijatelji prijateljev dopolnili na začetku prazno množico. Potem bomo iz te množice izločili prijatelje podane osebe in osebo samo.

```

1 def prijatelji_prijateljev(prijatelji, oseba):
2     vsi_pp = set() #prijatelji prijateljev je na zacetku prazna
3
4     # sprehod cez prijatelje osebe
5     for prijatelj in prijatelji[oseba]:
6         pp = prijatelji[prijatelj]
7         vsi_pp |= pp # z unijo dodam prijatelje prijatelja
8     # odstranim osebo
9     vsi_pp.remove(oseba)
10    # odstranim neposredne prijatelje osebe
11    vsi_pp -= prijatelji[oseba]
12
13    print(vsi_pp)

```

*Primer klica:*

```

>>> prijatelji_prijateljev(prijatelji, 'Nejc')
{'Ana', 'Peter'}

```