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

Miha Moškon

10. april 2020

Osnutek

# Kazalo



Osnutek

Osnutek

Osnutek

# 1 Spoznavanje z okoljem

Osnutek

Osnutek

# 2 Pogojni stavek

## 2.1 Zakaj pogojni stavki?

Vsi programi, ki smo jih do zdaj napisali, 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 1** *Napiši program, ki od uporabnika prebere telesno težo in višino in izpiše uporabnikov indeks telesne mase.*

**Rešitev 1** *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␣visino␣[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

Osnutek

(25 < 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.

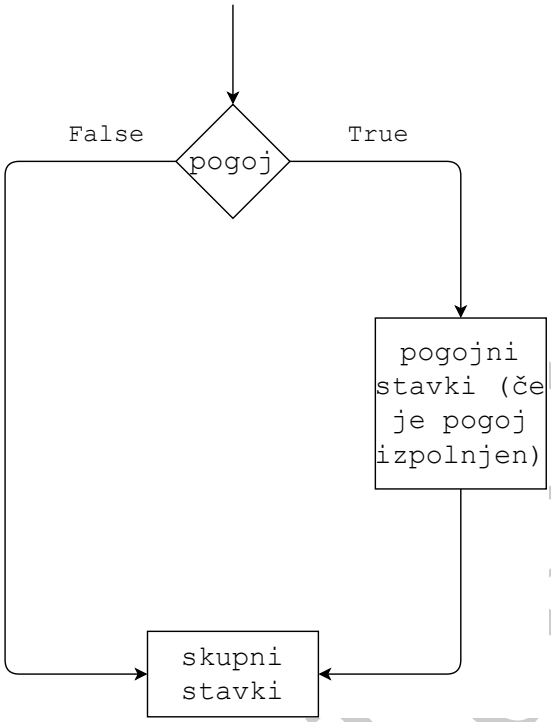
## 2.3 Kaj je pogoj?

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 poglejmo 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:

Osnutek




**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)
```

Osnutek

```
<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 2** *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 2** *Program od prej bomo dopolnili s pogojnim 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 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 if ITM > 25:
8     print("Tvoja teza 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

Osnutek

(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 čertem 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 >= 18.5` in `ITM <= 25`). Končen pogoj za izvedbo izpisa `Tvoja teza je ustrezna`, moramo torej

Osnutek

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 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** *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 manjpa ali enaka od 25, kar lahko zapišemo s pogojem `17.5 <= ITM and ITM <= 25`.*

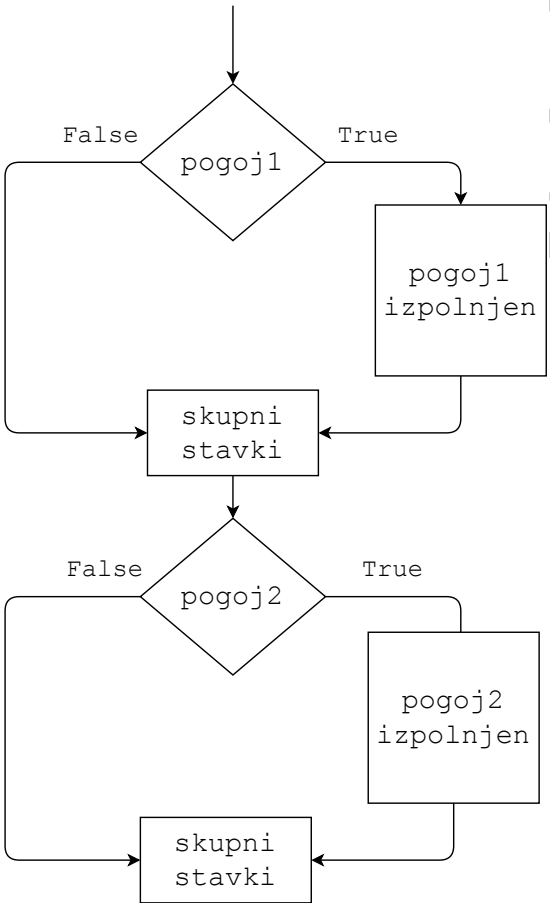
```
1  teza = float(input("Vpisi svojo tezo [kg]: "))
2  visina = float(input("Vpisi svojo visino [m]: "))
3  itm = teza/visina**2
4  print("Tvoj ITM je", itm)
5  if ITM < 17.5:
6      print("Tvoja teza je premajhna")
7  if ITM > 25:
8      print("Tvoja teza je prevelika")
9  if 17.5 <= ITM and ITM 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

Osnutek

## 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.



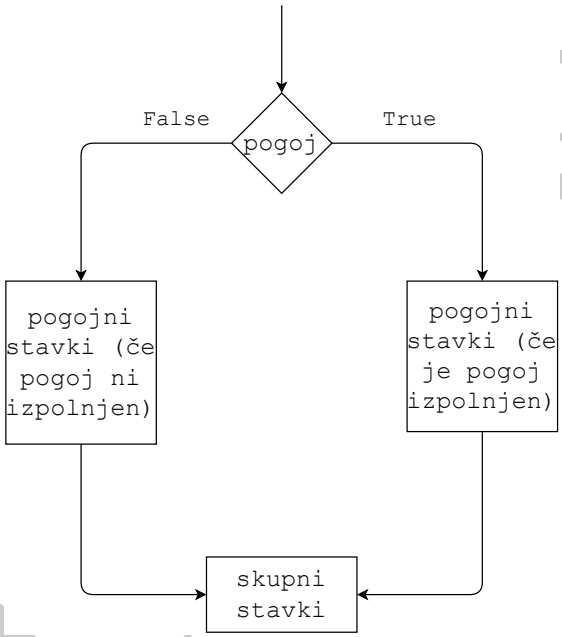

**Slika 2.2** Izpolnjenost pogoja `pogoj2` se preverja ne glede na izpolnjenost pogoja `pogoj1`.

Do sedaj smo v primeru neizpolnjenosti pogoja vedno skočili na del `skupni stavki`, torej na del, ki se izvede neodvisno od izpolnjenost 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:
```

```
    # 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 4** *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 4** *Tokrat bomo preverjali le pogoj o ustreznosti uporabnikove teže.*

```
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 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

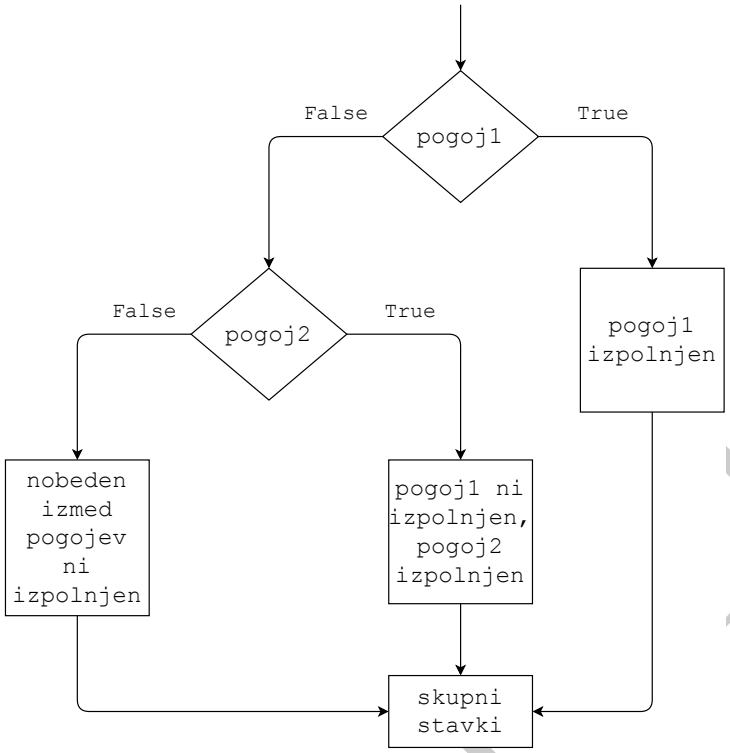
Uporabnik zdaj ve, če je njegova teža ustrezna. Če njegova teža ni ustrezna, informacije o tem ali je pretežak ali prelahek nima (verjetno se mu to sicer dozdeva). Zgornji primer bi torej radi dopolnili tako, da znotraj veje `else` izvedemo dodatno primerjanje, na podlagi katerega bomo lahko ugotovili 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 5** *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.*

Osnutek




**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 5** *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 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. Uporabi ugnezden stavek* ***if***

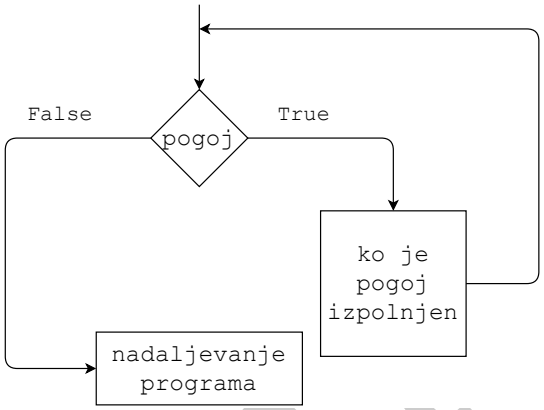
**Rešitev 6** *Potek programa bo podoben kot prej za razliko od gnezdenje stavka* ***if****.*

```
1  teza = float(input("Vpisi␣svojo␣tezo␣[kg]:␣"))
2  visina = float(input("Vpisi␣svojo␣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  else
8      if ITM > 25:
9          print("Tvoja␣teza␣je␣prevelika")
10     else:
11         print("Tvoja␣teza␣je␣ustrezna")
```

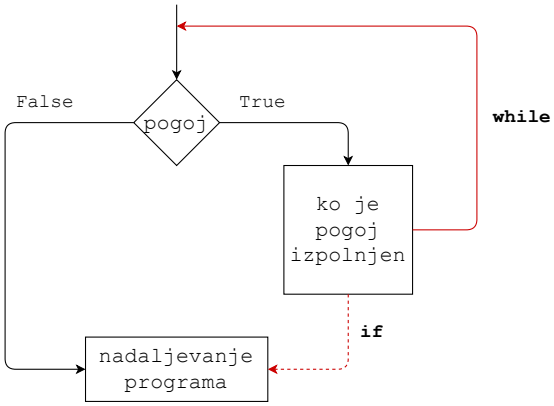
Osnutek

Osnutek

# 3 Zanka `while`

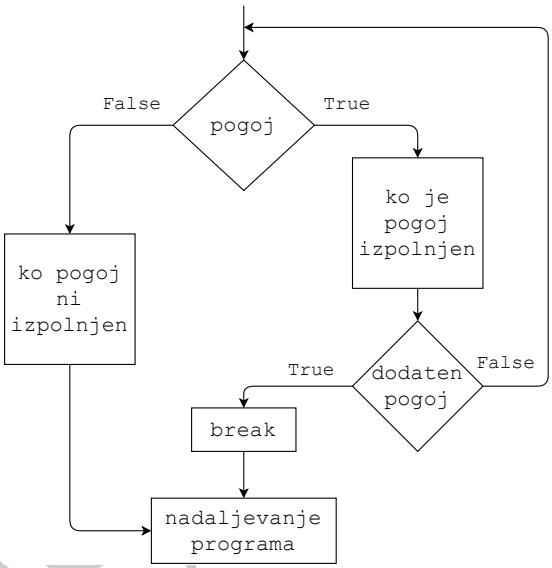



**Slika 3.1** ...




**Slika 3.2** ...

Osnutek




**Slika 3.3** ...

Osnutek

# 4 Seznami 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 obravnati ločeno, kar nas bo pripeljalo do ogrome 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 obravnati. 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 seznami?

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 seznami. 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,

Osnutek

tega v seznam dodamo.
Sezname 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 sezname shranjujejmo 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 definiranega seznama `dolzine` gredo torej indeksi od 0 do 3, saj seznam vsebuje 4 elemente:

| **indeksi** | | 0 | 1 | 2 | 3 |
|---|---|---|---|---|---|
| `dolzine` | = | `[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
```

Osnutek

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 seznami

Nad seznami 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 konkatenacija 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):
```

Osnutek

```
  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 seznami 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 seznami 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 seznami 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 seznama, 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

Osnutek

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 indeksiranje, le da tokrat pred indeksiranjem uporabimo besedico **`del`**:

```
del seznam[indeks]
```

## 4.6 Vgrajene funkcije nad seznami

Srečali smo že funkcijo `len`, s pomočjo katere lahko ugotovimo kakšna je dolžina seznama. Nad seznami 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 seznami 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 seznami 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 seznami. 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 seznami objekti (pravzaprav

Osnutek

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 seznami. 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 dvoje 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

Osnutek

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 7** *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 7** *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))
```

Osnutek

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 8** *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 8** *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.*

Osnutek

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.

Osnutek

**Zgled 9** *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 9** *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]
```

Sezname (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 zelimo 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

Osnutek

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 seznami bo za sortiranje uporabljena relacija `<`, ki smo jo v tem poglavju v povezavi s primerjanjem seznamov že srečali. Rekli smo, da relacija manjše sezname med seboj primerja leksikografsko. Najprej primerja ničti element prvega seznam z ničtim drugega. Če sta enaka, primerja prvi element prvega seznama s prvim elementom drugega seznama in tako naprej. Če bomo torej v vgnezdene sezname 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:

Osnutek

```
>>> 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 10** *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 10** *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 podseznam shranimo dolžino skoka, saj želimo podseznam sortirati po dolžini skokov. Na koncu skoke sortiramo in izpišemo zmagovalce in dolžine zmagovalnih skokov.*

```
1  skoki = [] # na zacetku 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 Generiraje seznamov s funkcijo range

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

Osnutek

V določenih primerih želimo imeti sezname s števili v podanem razponu. Praktično uporabo takih seznamov bomo podrobneje spoznali v naslednjem poglavju, zaenkrat pa si poglejmo, 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 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`. Poglejmo 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 pomnlniškim prostorom, saj dolgi seznami zasedejo veliko prostora. Še vedno pa lahko nad rezultatom funkcije `range` delamo podobne stvari, kot nad seznami. Lahko jih npr. indeksiramo:

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

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

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

Osnutek

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 sprašujetete 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 `argument` 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)
```

Osnutek

```
>>> 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 opcijski, 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 narediili 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.

Osnutek

## 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 nevključenostjo 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]
```

Osnutek

Vse zgoraj naštete kombinacije lahko seveda po mili volji kombiniramo. Če bi npr. hoteli vzeti vsak drug element seznama v razponu od 3 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 specificirali 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 11** *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 11** *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 12** *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*

Osnutek

**Rešitev 12** *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 prerezemo gen 1
5  i1 = len(gen1)//2 # celostevilsko deljenje z 2
6  # kje prerezemo 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 13** *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 13** *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
```

Osnutek

```
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. Pogoj 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 seznami je ena izmed najpogostejši operacij sprehajanje nad seznami. Kako narediti tak sprehod? Zgoraj smo se z zanko `while` sprehajali nad indeksi niza. Podoben sprehod bi lahko naredili tudi nad seznami. 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 sprehajanju nad seznami in seznamu podobnimi podatki v Pythonu namenjena posebna zanka, zanka `for`.

Osnutek

Osnutek

# 5 Zanka `for`

Osnutek

Osnutek

# 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 objavio na internetu. In svet je postal še malenkost

Osnutek

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 sprejala, 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 pommnilniku (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 našteli 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,

Osnutek

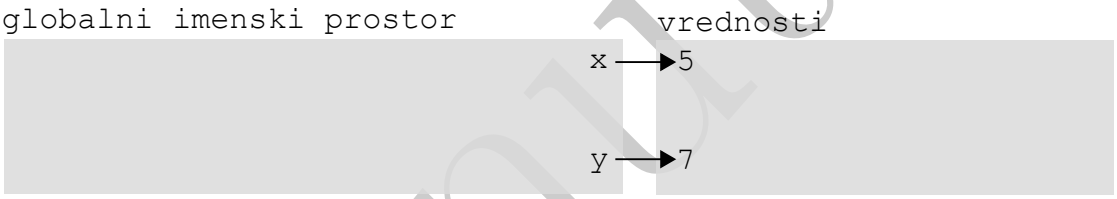
kot če smo nekemu imenu – spremememnljivki, priredili neko vrednost). Dejanska izvedba stavkov znotraj funkcije pa se je izvršila šele, ko smo funkcijo poklicali. Mimogrede, če bi v funkciji imeli kakšno sintaktično napako, kot je npr. uporaba nedefinirane spremenljivke, bi jo Python našel šele ob klicu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



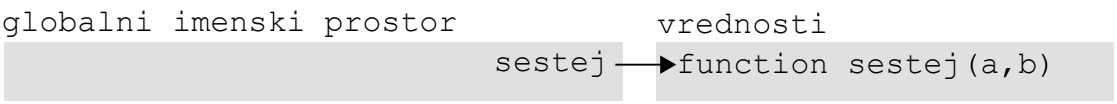

**Slika 6.1** Imena v imenskem prostoru kažejo na konkretne vrednosti v pomnilniku.

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.

Osnutek




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

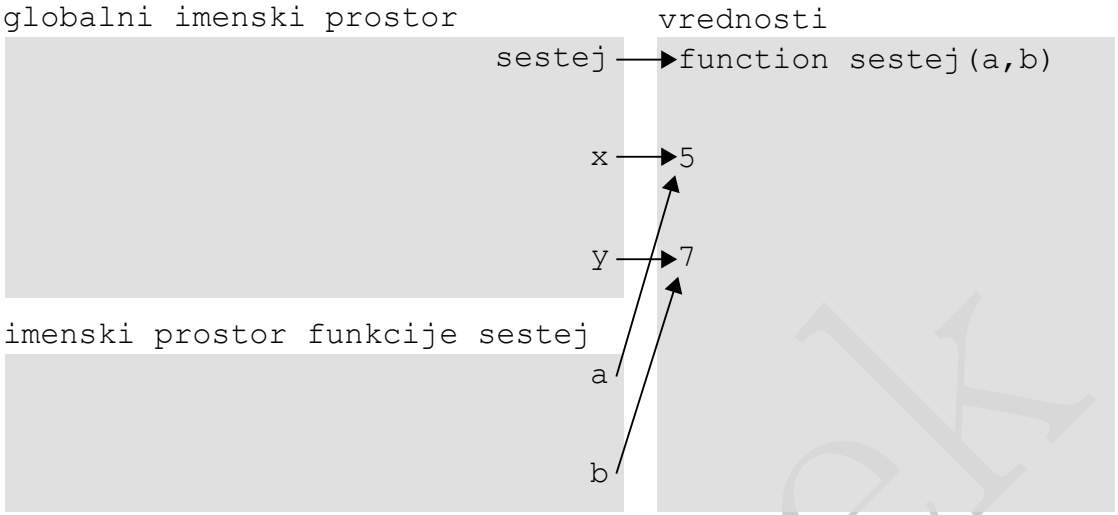
```
1 def sestej(a, b): # definicija funkcije
2     c = a + b
3     print(c)
4 x = 5
5 y = 7
6 sestej(x,y) # klic funkcije
```

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 prireditev

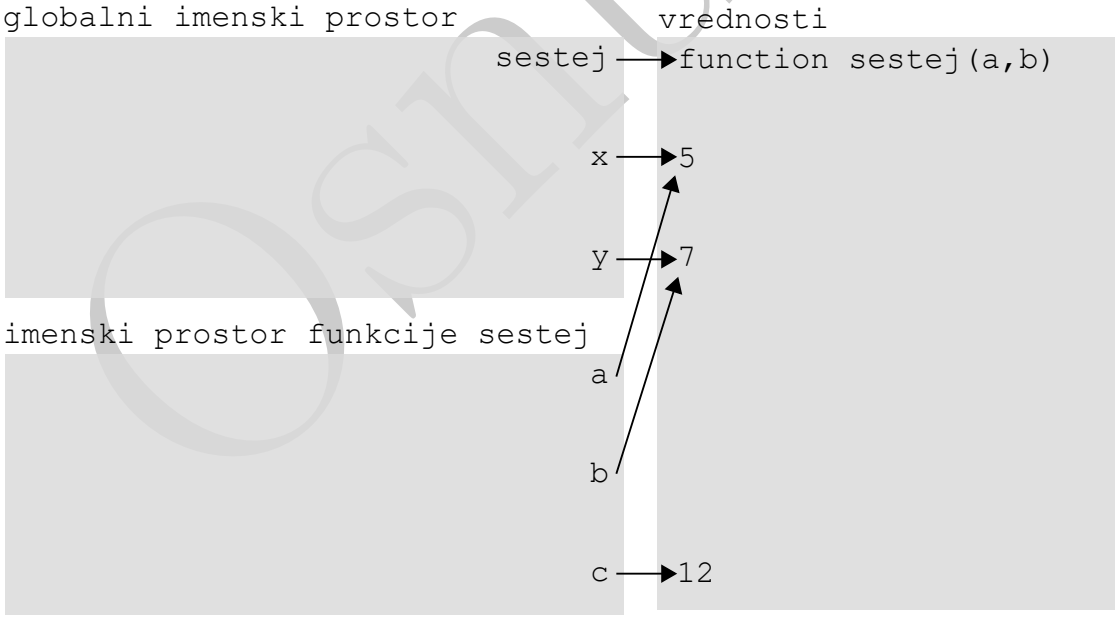
```
>>> a = x
>>> b = y
```

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.

Osnutek

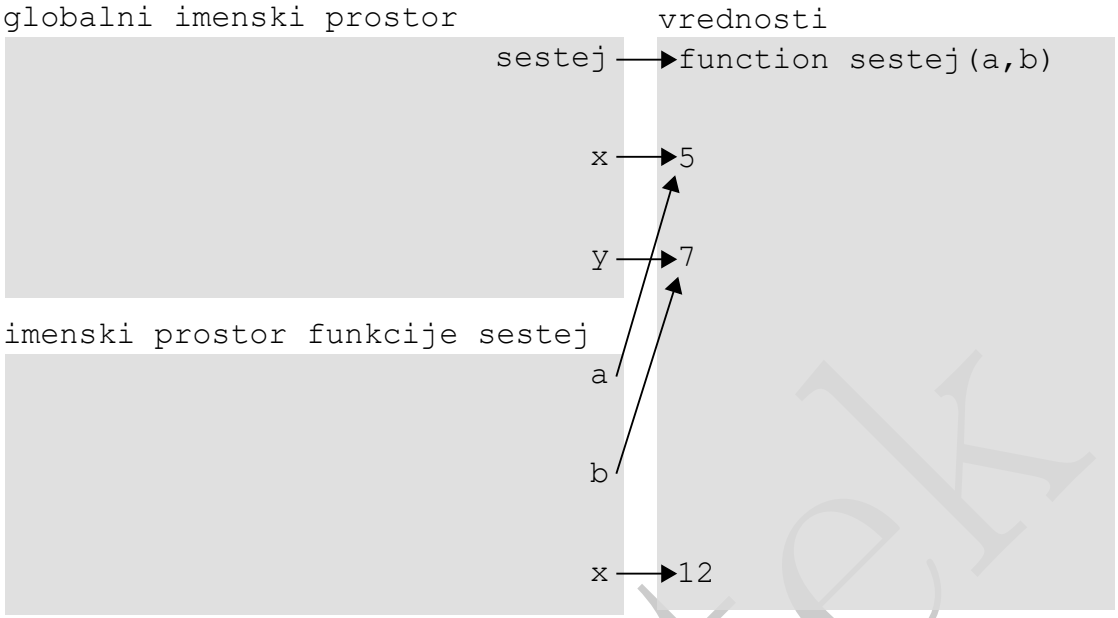



**Slika 6.3** Ob klicu funkcije se ustvari njen lokalni imenski prostor znotraj katerega se dodatno ustvarijo plitve kopije vrednosti, s katerimi smo funkcijo poklicali.




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

Osnutek



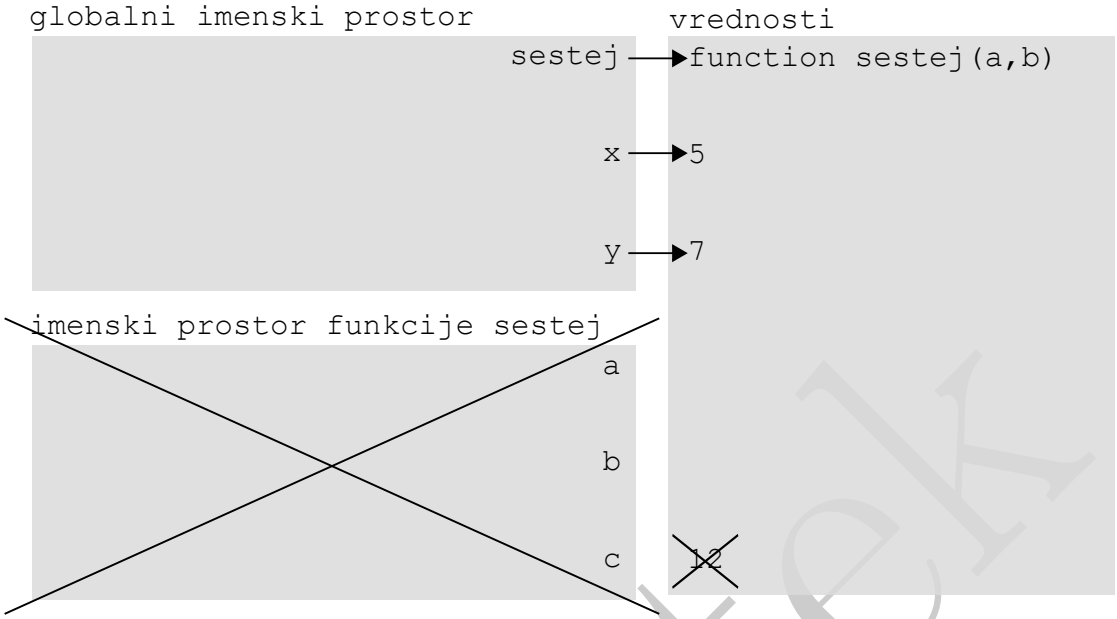

**Slika 6.5** Znotraj funkcije lahko uporabljamo enaka imena spremenljivk kot izven funkcije in s tem ne vplivamo na globalne spremenljivke.

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 lokalen 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 ostaneta, 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.

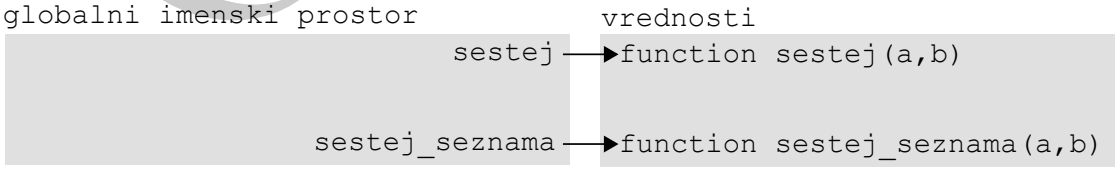
Osnutek




**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_seznama(a,b): # sestej istolezne elemente
6     for i in range(len(a)):
7         sestej(a[i], b[i])
8
9 sestej_seznama([1,2,3],[4,5,6]) # klic funkcije
```

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

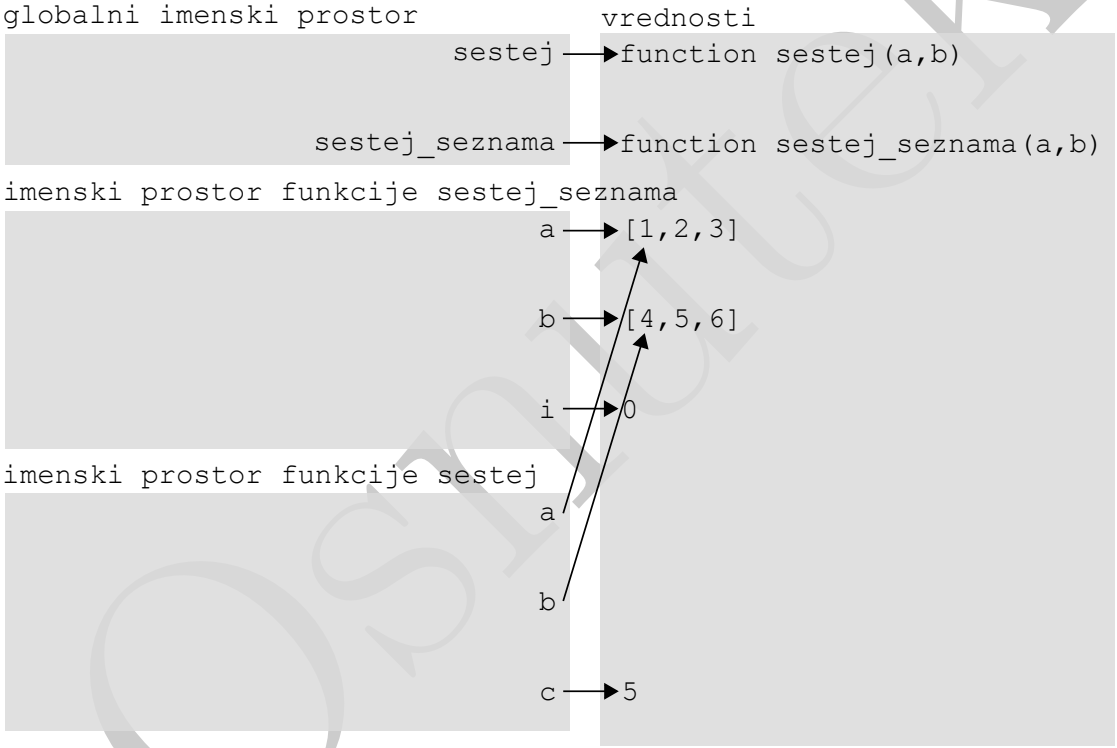



**Slika 6.7** Imeni definiranih funkcij sta shranjeni v globalnem imenskem prostoru, zato jih lahko pokličemo od kjerkoli.

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

Osnutek

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_seznama`, bo ta dobila lokalen imenski prostor. Ko bomo iz te funkcije poklicali funkcijo `sestej`, bo ta dobila svoj imenski prostor, ki se s prostorom funkcije `sestej_seznama` 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 funkcijo, 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 `sestej`, tako da bo vsoto dveh števil vračala in ne izpisovala.

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

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

Osnutek

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

**Rešitev 14**

```
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 odvrtela 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 15** *Napiši funkcijo, ki kot argument prejme celo število in* ***vrne*** *vrednost* `True`*, če je to število praštevilo, sicer pa* ***vrne*** *vrednost* `False`*.*

**Rešitev 15**

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

Osnutek

```
4             return False # prekine funkcijo in vrne False
5     return True # for se je odvrtel 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 16** *Napiši funkcijo, ki kot argument prejme celo število in izpiše vsa praštevila do vključno podanega števila.*

**Rešitev 16**

```
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 praštevilo. Kako? Tako, da smo poklicali funkcijo, ki vrne `True`, če je podano število praštevilo. 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 povozimo, tako da jih specificiramo ob klicu, npr. kot

Osnutek

```
>>> 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 opcijskih argumentov.

**Zgled 17** *Napiši funkcijo, ki vrne Fibonaccijevo zaporedje števil, pri čemer naj uporabik poda dolžino zaporedja in prvi dve števili v zaporedju.*

**Rešitev 17**

```
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 18** *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 18**

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

Osnutek

```
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 posamezen funkcije, ki je v uvoženem modulu definirana, bi to naredili na sledeč način

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

Osnutek

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 način uvažanja modulov izogibamo.

Osnutek

## 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 lotili 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 [1]) Pred uporabo jih moramo torej namestiti. Problem nameščanja tovrstnih paketov je, poleg včasih mukotrpnega procesa iskanja ustreznih namestitvenih datotek in ročne namestitive, 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.

[1] 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/`.

Osnutek

*Python package index*) poišče in namesti ustrezne pakete avtomatsko. Nameščen je že skuppaj 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 Widnows jo zaženemo tako, da v start meni vpišemo `cmd`). V primeru, da smo ob namestitivi 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.

Osnutek

# 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.

Osnutek

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

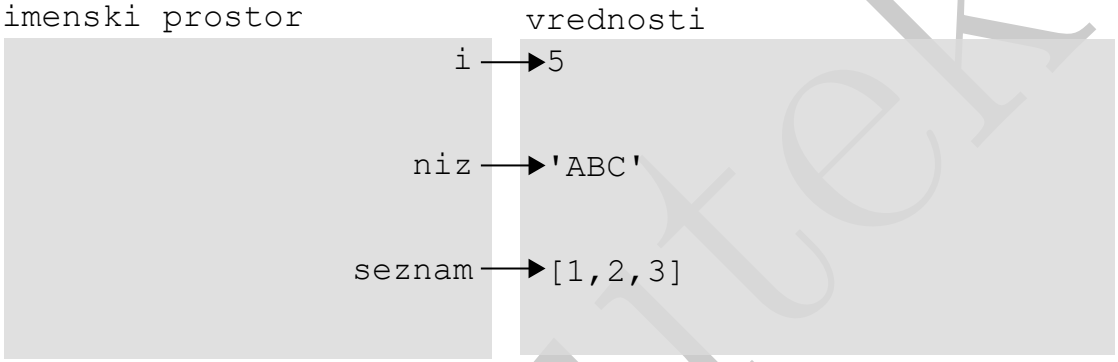
| podatkovni tip | opis | spremenljiv |
|---|---|---|
| `bool` | *boolean* (`True, False`) | Ne |
| `int` | *integer* (celo število) | Ne |
| `float` | *floating-point* (decimalno število) | Ne |
| `str` | *string* (niz) | Ne |
| `list` | *list* (seznam) | Da |
| `tuple` | *tuple* (terka) | Ne |
| `dict` | *dictionary* (slovar) | Da |
| `set` | *set* (množica) | Da |
| `frozenset` | *frozenset* (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. Zapoerdje prireditvenih stavkov

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

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




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

spremenljivki, na primer takole:

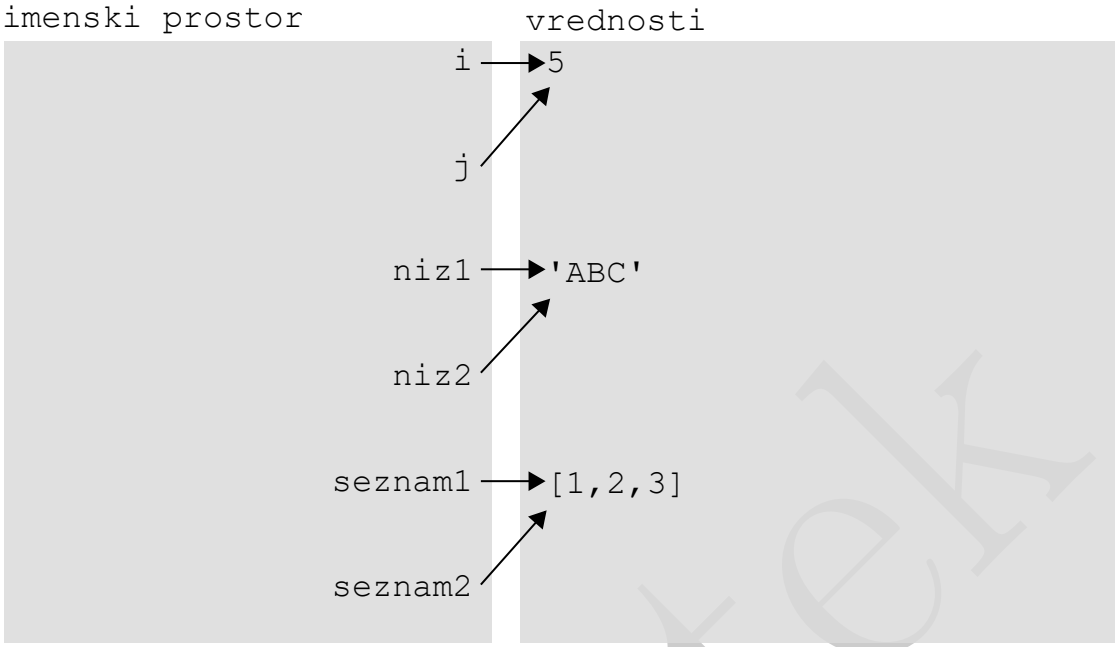
```
>>> i = 1
>>> j = i
>>> niz1 = 'ABC'
>>> niz2 = niz1
>>> seznam1 = [1,2,3]
>>> seznam2 = seznam1
```

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-

Osnutek




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

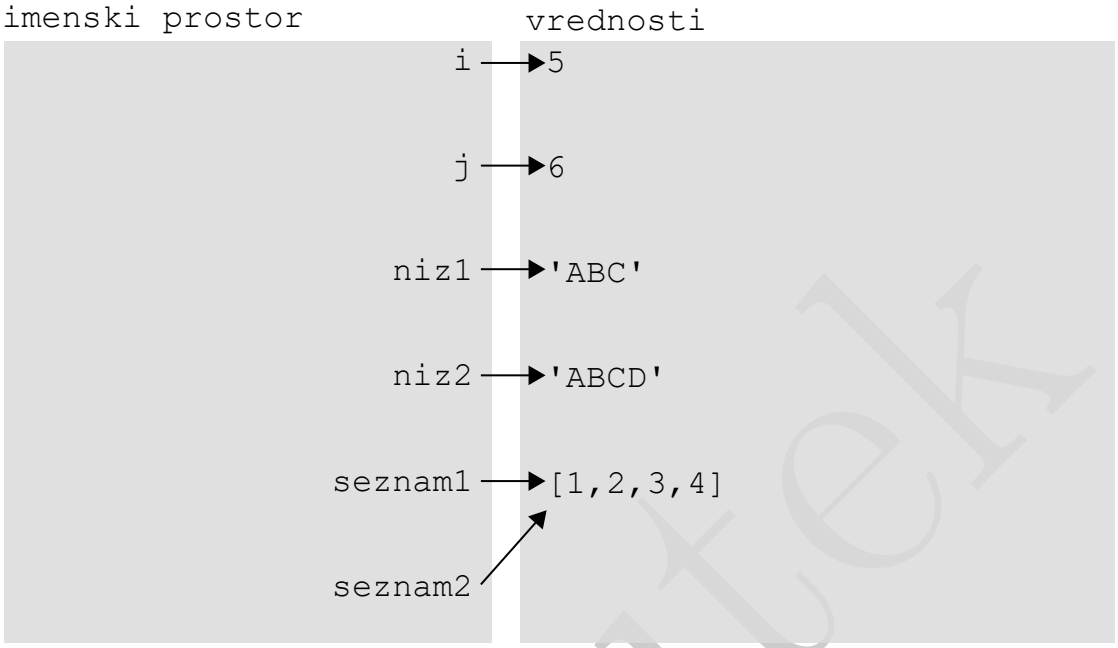
mnimo se. Spremenljiv podatek lahko spreminjamo, ko pa pokusimo spremeniti nespremenljiv podatek, se ustvari njegova kopija (globoka), ki odraža narejeno spremembo. Če npr. uporabimo operator `+=`, bo v primeru spremenljivega podatka spremenjen obstoječ podatek, v primeru nespremenljivega podatka pa bo ustvarjen nov podatek, ki bo odražal narejeno spremembno.
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 enostvane da ali ne. Odgovor je namreč odvisen od spremenljivosti podatka, ki ga spreminjamo.

Osnutek

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, priredijo 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.

Osnutek

**Zgled 19** *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 19** *Ob klicu funkcije spremenljivki, s katerima funkcijo pokličemo, dobita 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.

Osnutek

## 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 seznami. 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'>
```

Osnutek

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 seznami. 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 obravavali 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

Osnutek

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 prireditvnega 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)
```

Osnutek

## 8.8 Pakiranje seznamov v sezname 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 seznami 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 seznami 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)]
```

Osnutek

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 zaželena (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 poglejmo spodnji zgled.

**Zgled 20** *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 20** *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.*

Osnutek

*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 spreminjau 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 poglejmo v spodnjih zgledih.
Najprej si poglejmo kako se lotiti pisanja in uporabe funkcije, ki sprejema nespremenljive podatke.

**Zgled 21** *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 21** *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:*

Osnutek

```
>>> 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 22** *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 22** *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']
```

Osnutek

# 9 Slovarji

## 9.1 Zakaj slovarji?

Zdaj smo se že pobliže 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 čemer je bil rezultat sortiranja tak, da je manjši ključ imel manjšo vrednost glede na uporabljeno relacijo.

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 številil, 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 najti.

Takim imenikom in slovarjem ustreza podatkovna struktura `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 ključe, 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 seznami ali terkami.

Osnutek

## 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 številk 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). Obstoj ključa v slovarju lahko preverimo z operatorjem `in`:

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

Osnutek

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

**Zgled 23** *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 23** *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 indeksrianju 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'}
```

Osnutek

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 24** *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 24** *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 spremememe 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␣ze␣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 25** *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 25** *Tokrat bomo v funkciji izvedli prirejanje vrednosti samo v primeru, če podan ključ v slovarju je. V nasprtonem 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 26** *Napiši funkcijo* `prestej_baze`*, ki kot argument prejme nukleotidno zaporedje baz zapisano kot niz. Funkcija naj vrne slovar, ki za posamezno bazo vsebuje število ponovitev.*

Osnutek

**Rešitev 26** *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 slovarjrv 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 27** *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 27** *Spet bomo najprej preverili obstoj ključa, nato pa izvedli brisanje.*

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

Osnutek

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

## 9.6 Ključi in vrednosti

Posamezen ključe se lahko v slovarju pojavi največ enkrat. Ključi so torej enolični identifikatorji, preko katerih pridemo do posamezne vrednost. 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 28** *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.*

Osnutek

**Rešitev 28** *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 29** *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 29** *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
```

Osnutek