Podloge za stručno usavršavanje učitelja osnovnih škola
za domenu
Računalno razmišljanje i programiranje

06

Zbirke podataka, liste (`list`) i n-torke (`tuple`)

Uz dozvolu izdavača korišteni su sadržaji iz priručnika:

Leo Budin Predrag Brođanac Zlatka Markučič
Smiljana Perić Dejan Škvorc Magdalena Babić

Računalno razmišljanje i programiranje u Pythonu
Element, Zagreb, 2017

Zbirka podataka tipa `list`

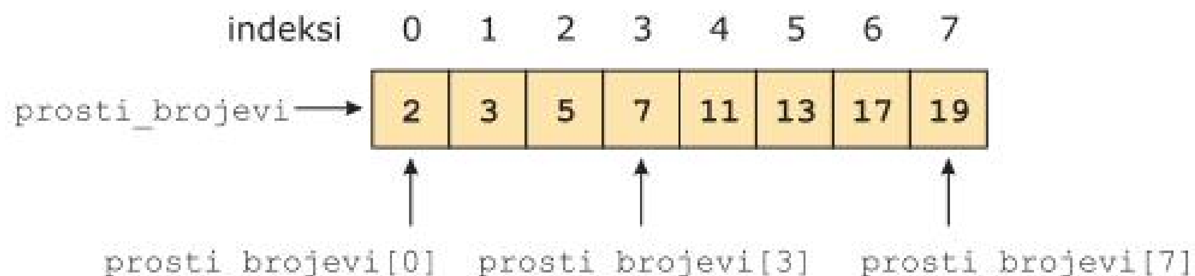
Pri rješavanju različitih problema programiranjem često je potrebno pohraniti više srodnih ili povezanih podataka. Bilo bi vrlo nezgodno za svaku vrijednost uvesti novu varijablu. Primjerice, ako bismo željeli pohraniti sve proste brojeve manje od 20, morali bismo uvesti osam varijabli i pridijeliti im pripadne vrijednosti. To bi moglo izgledati ovako:

```
>>> prosti_broj_0 = 2
>>> prosti_broj_1 = 3
>>> prosti_broj_2 = 5
>>> prosti_broj_3 = 7
>>> prosti_broj_4 = 11
>>> prosti_broj_5 = 13
>>> prosti_broj_6 = 17
>>> prosti_broj_7 = 19
```

Kao što smo već naučili, brojiti počinjemo od nula pa smo zbog toga prvi po redu prosti broj 2 pohranili u varijablu `prosti_broj_0`, a osmi po redu, 19, u varijablu `prosti_broj_7`.

U *Pythonu* možemo više vrijednosti smjestiti u popis sa zajedničkim imenom tako da ih napiše-mo unutar uglatih zagrada jedan iza drugoga i odvojimo zarezima. Tako i naše proste brojeve možemo zapisati:

```
>>> prosti_brojevi = [2, 3, 4, 5, 7, 11, 13, 17, 19]
```



Slika 6.1. prikazuje način smještanja elemenata liste u spremniku računala. Svaki broj napisan u listi je **element** liste, a redni broj elementa unutar liste je **indeks**. Još jednom treba naglasiti da je indeks prvog elementa jednak `0` i zbog toga je indeks zadnjeg elementa u listi za jedan manji od duljine liste.

Kao i kod ostalih tipova podataka u interaktivnom sučelju možemo ispitati kako je lista pohranjena i kako će biti ispisana funkcijom `print()`:

```
>>> prosti_brojevi
[2, 3, 5, 7, 11, 13, 17, 19]
>>> print(prosti_brojevi)
[2, 3, 5, 7, 11, 13, 17, 19]
```

Kada u interaktivnom sučelju napišemo ime varijable `prosti_brojevi`, prikazat će nam se cijela lista omeđena uglatim zagradama. Jednako tako, kada u funkciji `print()` kao argument napišemo ime liste, obavit će se ispis cijele liste.

Do pojedinog elementa u listi možemo doći tako da napišemo ime liste i iza njega u uglatoj zagradi njegov indeks. Tako ćemo elemente liste prikazane na slici 6.1. moći dohvatiti na sljedeći način:

```
>>> prosti_brojevi[0]
2
>>> prosti_brojevi[3]
7
>>> prosti_brojevi[7]
19
```

Prisjetimo se petlje `for` koju smo upoznali u 3. poglavlju. Ta petlja dolazi do punog izražaja pri radu s listama. Tako bismo sve pojedinačne elemente iz naše liste mogli jednostavno ispisati na sljedeći način:

```
>>> for i in range(8):
        print(prosti_brojevi[i], end=' ')

2 3 5 7 11 13 17 19
```

Funkcija `range(8)` generirat će brojeve od 0 do 7 te će varijabla `i` redom poprimati te vrijednosti. U ovom konkretnom slučaju, vrijednosti varijable `i` bit će indeksi elemenata liste. Elementi liste ne moraju biti samo brojevi. Svi tipovi koje smo dosad upoznali, pa i same liste, mogu biti elementi lista. Štoviše, elementi jedne liste ne moraju svi biti istog tipa.

Tako u listu možemo pohraniti imena učenika i ispisati pozdrave:

```
>>> imena = ['Marko', 'Ana', 'Marija', 'Pero']
>>> for i in range(4):
        print('Lijep pozdrav, {}!'.format(imena[i]))

Lijep pozdrav, Marko!
Lijep pozdrav, Ana!
Lijep pozdrav, Marija!
Lijep pozdrav, Pero!
```

Negativni indeksi

Elementi liste mogu se dohvatiti i s negativnim indeksima. Naime, istodobno s uobičajenim indeksima postoje i negativni indeksi koji su prikazani slikom 6.2.

indeksi	0	1	2	3	4	5	6	7
prosti_brojevi →	2	3	5	7	11	13	17	19
negativni indeksi	-8	-7	-6	-5	-4	-3	-2	-1

Indeks prvog elementa ima vrijednost 0, ali i negativnu vrijednost `-8`, što je u skladu s brojanjem indeksa od kraja liste koji počinje indeksom `-1`. Negativni su indeksi posebno prikladni za dohvat elemenata s kraja liste jer do njih možemo jednostavno doći i bez poznavanja njezine duljine – indeks zadnjeg elementa liste je uvijek jednak `-1`, a predzadnjeg `-2`.

Promjena vrijednosti elemenata liste

Pojedinim elementima liste možemo promijeniti vrijednost naredbom pridruživanja. Pogledajmo kako u listi imena možemo promijeniti vrijednost elementa s indeksom `2`:

```
>>> imena
['Marko', 'Ana', 'Marija', 'Pero']
>>> imena[2] = 'Marko'
>>> imena
['Marko', 'Ana', 'Marko', 'Pero']
```

U izvorno zadanoj listi `imena` promijenili smo vrijednost elementu s indeksom `2`. Uočimo da dva, ali i više elemenata liste mogu imati jednake vrijednosti.

Izdvajanje dijelova liste i kopiranje cijele liste

Umjesto pojedinačnih elemenata mogu nam zatrebati neki dijelovi liste.

Takav dio liste možemo izdvojiti tako da uz ime liste u uglatoj zagradi napišemo dva indeksa razdvojena dvotočkom `lista_a[početak : kraj]`. Ovakva naredba izdvojit će isječak u kojem će na prvom mjestu biti element `lista_a[početak]`, a posljednji `lista_a[kraj - 1]`. Pogledajmo to u interaktivnom sučelju:

```
>>> prosti_brojevi = [2, 3, 5, 7, 11, 13, 17, 19]          #1
>>> prosti_brojevi[1 : 7]                                  #2
[3, 5, 7, 11, 13, 17]
>>> prosti_brojevi[1 : -1]                                 #3
[3, 5, 7, 11, 13, 17]
>>> prosti_brojevi[2 : 6]                                  #4
[5, 7, 11, 13]
>>> prosti_brojevi[2 : -2]                                 #5
[5, 7, 11, 13]
>>> prosti_brojevi[3 : 5]                                  #6
[7, 11]
```

Naredbom (#2) dobili smo isječak u kojem nedostaju prvi i zadnji element liste (#1). Isti učinak ima i naredba (#3) u kojoj smo zadnji element označili indeksom `-1`. U naredbama (#4) i (#5) smo u isječku izostavili prva dva i zadnja dva elementa, dok isječak (#6) sadrži samo dva srednja elementa početne liste. Dobiveni isječci prikazani su slikom 6.3.

indeksi	0	1	2	3	4	5	6	7
prosti_brojevi →	2	3	5	7	11	13	17	19
prosti_brojevi[1:7] →	2	3	5	7	11	13	17	19
prosti_brojevi[2:6] →	2	3	5	7	11	13	17	19
prosti_brojevi[3:5] →	2	3	5	7	11	13	17	19

Na sličan način mogli bi možda izdvojiti i cijelu listu. Pokušajmo to učiniti:

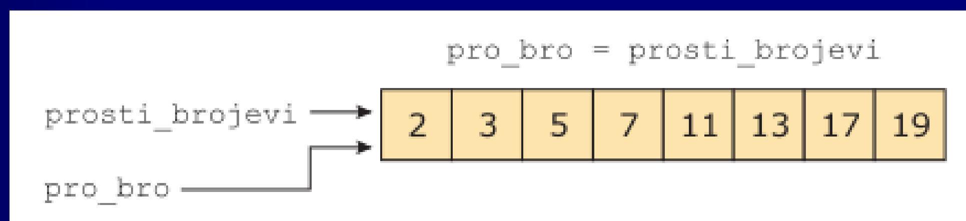
```
>>> prosti_brojevi[0 : 7]                                    #7
[2, 3, 5, 7, 11, 13, 17]
>>> prosti_brojevi[0 : 8]                                    #8
[2, 3, 5, 7, 11, 13, 17, 19]
>>> prosti_brojevi[0 : ]                                     #9
[2, 3, 5, 7, 11, 13, 17, 19]
>>> prosti_brojevi[ : ]                                      #10
[2, 3, 5, 7, 11, 13, 17, 19]
```

U naredbi (#7) nismo uspjeli obuhvati cijelu listu. Indeks 0 kao početni indeks dohvaća prvi element liste, ali s indeksom 7 zadnji element nije obuhvaćan isječkom. U naredbi (#8) taj smo indeks povećali za jedan i tako dohvatili i posljednji element. No, iz naredbe (#9) je vidljivo da umjesto toga možemo jednostavno iza dvotočke ostaviti prazno mjesto. Na taj način ne moramo voditi računa o duljini liste jer će ostavljanje praznog mjesta iza dvotočke automatski u isječak preuzeti sve elemente do kraja liste. Konačno, pri kopiranju cijele liste možemo i umjesto početnog indeksa 0 ostaviti prazno mjesto tako da se kopiranje može obaviti i naredbom (#10) u kojoj unutar uglatih zagrada pišemo samo znak dvotočke. Prema tome, ostavljanjem praznog mjesta ispred dvotočke u isječak se preuzimaju elementi od početka liste.

U radu s listama moramo biti oprezni !

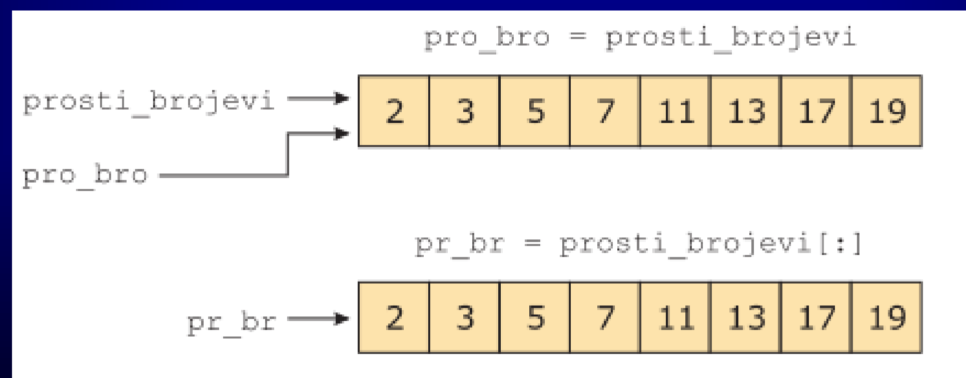
```
>>> pro_bro = prosti_brojevi                                    #11
>>> pro_bro
[2, 3, 5, 7, 11, 13, 17, 19]
```

Naredba `#11` dodijeljuje listi `prosti_brojevi` novo ime.




```
>>> pr_br = prosti_brojevi[ : ]                                 #12
>>> pr_br
[2, 3, 5, 7, 11, 13, 17, 19]
```

Naredba `#12` pridružuje drugoj varijabli isječak cijele liste



Ako nešto mijenjamo u listi s imenom `pro_bro`, promjena će biti vidljiva i pod imenom `prosti_brojevi`, dok promjena u listi `pr_br` ne utječe na sadržaj izvorne liste. Pogledajmo to u interaktivnom sučelju:

```
>>> pr_br[3] = 0                                            #13
>>> pr_br                                                   #14
[2, 3, 5, 0, 11, 13, 17, 19]
>>> prosti_brojevi                                          #15
[2, 3, 5, 7, 11, 13, 17, 19]
>>> pro_bro[3] = 0                                          #16
>>> pro_bro                                                 #17
[2, 3, 5, 0, 11, 13, 17, 19]
>>> prosti_brojevi                                          #18
[2, 3, 5, 0, 11, 13, 17, 19]
```

U naredbi (**#13**) pridijelili smo vrijednost `0` elementu s indeksom 3. Pogled na listu naredbom (**#14**) vidimo da se sadržaj liste promijenio. Lista `pr_br` je prava kopija liste `prosti_brojevi` tako da ta promjena ne utječe na njezin sadržaj kao što vidimo nakon izvođenja naredbe (**#15**). Međutim, `pro_bro` je samo drugo ime za listu `prosti_brojevi` tako da je promjena sadržaja (**#16**) vidljiva u toj listi (**#17**) i (**#18**).

Osnovni operatori, funkcije i metode za rad s listama

Operatori nadovezivanja (+) i uvišestručavanja (*)

Operatori djeluju slično kao i za stringove:

```
>>> brojevi_0 = [1, 2]
>>> brojevi_1 = [3, 4]
>>> brojevi = brojevi_0 + brojevi_1                    #1
>>> print(brojevi)                                     #2
[1, 2, 3, 4]
>>> imena_0 = ['Ana', 'Marija']
>>> imena_1 = ['Marko', 'Pero']
>>> imena = imena_0 + imena_1                          #3
>>> print(imena)
['Ana', 'Marija', 'Marko', 'Pero']
```

Naredbom (#1) elemente druge liste nadovezujemo na prvu listu. Ispisom (#2) vidimo da smo uistinu dobili listu u kojoj su prvoj listi dodani (nadovezani) elementi druge liste. Naredbom (#3) ćemo nadovezati liste `imena_0` i `imena_1` u jednu listu imena.

Osnovni operatori, funkcije i metode za rad s listama

Operatori nadovezivanja (+) i uvišestručavanja (*)

Operatori djeluju slično kao i za stringove:

```
>>> brojevi_0 = [1, 2]
>>> brojevi_1 = [3, 4]
>>> brojevi = brojevi_0 + brojevi_1                    #1
>>> print(brojevi)                                      #2
[1, 2, 3, 4]
>>> imena_0 = ['Ana', 'Marija']
>>> imena_1 = ['Marko', 'Pero']
>>> imena = imena_0 + imena_1                          #3
>>> print(imena)
['Ana', 'Marija', 'Marko', 'Pero']
```

Naredbom (#1) elemente druge liste nadovezujemo na prvu listu. Ispisom (#2) vidimo da smo uistinu dobili listu u kojoj su prvoj listi dodani (nadovezani) elementi druge liste. Naredbom (#3) ćemo nadovezati liste `imena_0` i `imena_1` u jednu listu imena.

Operator uvišestručenja liste (*) koristi se kako bi se elementi izvorne liste uvišestručili zadani broj puta. Njegovo je djelovanje slično operaciji nadovezivanja, ali se ovdje umjesto nadovezivanja dviju različitih lista, jedna te ista lista nadovezuje sama na sebe zadani broj puta.

```
>>> brojevi_0 * 1                                  #6
[1, 2]
>>> brojevi_0 * 2                                  #7
[1, 2, 1, 2]
>>> 3 * brojevi_0                                  #8
[1, 2, 1, 2, 1, 2]
>>> brojevi_0 * 0                                  #9
[]
>>> brojevi_0 * -2                                 #10
[]
```

Naredbe (#6), (#7) i (#8) ukazuju da ćemo uvišestručenjem dobiti liste u kojima se elementi izvorne liste ponavljaju onoliko puta koliko određuje faktor uvišestručavanja, a on je određen cijelim brojem koji je zadan u naredbi. Uočimo da je svejedno pišemo li faktor uvišestručavanja prije ili poslije znaka * (#8). Uvišestručujemo li sa 0 i bilo kojim negativnim cijelim brojem, dobit ćemo praznu listu koja se obilježava uglatim zagradama napisanim jedna uz drugu (#9), (#10).

Operator uvišestručavanja (*) ne može djelovati na dvije liste. Pokušamo li to načiniti, dobit ćemo sljedeći ispis:

```
>>> brojevi_0 * brojevi_1
Traceback (most recent call last):
  File "<pyshell#28>", line 1, in <module>
    brojevi_0 * brojevi_1
TypeError: can't multiply sequence by non-int of type 'list'
```

Primjenom navedenih operatora može se od nekih jednostavnijih lista stvarati složenije. Evo nekih primjera:

```
>>> [] + [0] + [1] + [2] + [3] + [4] + [5]                    #11
[0, 1, 2, 3, 4, 5]
>>> brojevi = []
>>> for i in range(6):
        brojevi += [i]                                        #12

>>> brojevi
[0, 1, 2, 3, 4, 5]
>>> brojevi = []
>>> for i in range(6):
        brojevi += [i * i]                                    #13

>>> brojevi
[0, 1, 4, 9, 16, 25]
```

```
>>> brojevi = []
>>> for i in range(6):
        brojevi += [i] * i                                    #14

>>> brojevi
[1, 2, 2, 3, 3, 3, 4, 4, 4, 4, 5, 5, 5, 5, 5]
```

Funkcija `len()`

Želimo li ispisivati pojedine elemente liste tada se oni moraju dohvatiti svojim indeksima. Tako bismo elemente gornjih lista (#1) i (#3) mogli ispisati na sljedeći način:

```
>>> for i in range(4):                                                    #4
        print(brojevi[i], end=' ')                                        #5

1 2 3 4
>>> for i in range(4):
        print(imena[i], end=' ')

Ana Marija Marko Pero
```

U naredbi (#4) počinje petlja `for` koja će ponoviti ispis (#5) za svaki element liste. U našem slučaju se radi o četiri elementa liste `brojevi` te smo kao argument funkcije `range()` zadali broj 4. Jednako, moramo postupati i za ispis druge liste – kako bismo znali pravilno zadati argument funkcije `range()` moramo ručno brojiti elemente liste i saznati njezinu duljinu.

Prebrojavanje liste za nas može načiniti i *Python* svojom funkcijom `len()`. Ta će funkcija vratiti duljinu liste kao broj elemenata liste. Pogledajmo:

```
>>> len(brojevi_0)
2
>>> len(brojevi_1)
2
>>> len(brojevi)
4
```

Prema tome ispis lista `brojevi` možemo obaviti i na sljedeći način:

```
>>> for i in range(len(brojevi)):
        print(brojevi[i], end=' ')

1 2 3 4
>>> for i in range(len(imena)):
        print(imena[i], end=' ')

Ana Marija Marko Pero
```

Na taj način ne moramo više ručno brojiti elemente liste što je vrlo praktično naročito kada radimo s listama promjenljive duljine.

Metoda `extend()`

Nadovezivanje lista može se obaviti i metodom `extend()`. Prisjetimo se da se metoda može na neki način smatrati funkcijom, no treba ju drukčije pozivati. Metodu pozivamo tako da prvo napišemo ime varijable, i nakon toga naziv metode (već smo kod opisa formatiranja upoznali metodu `format()` koja djeluje na string).

```
>>> br_0 = [1, 2]                    #15
>>> br_1 = [3, 4]                    #16
>>> br = []                          #17
>>> br
[]
>>> br.extend(br_0)                  #18
>>> br
[1, 2]
>>> br.extend(br_1)                  #19
>>> br
[1, 2, 3, 4]
```

Dvije liste (#15) i (#16) znamo nadovezati operatorom nadovezivanja (kao u naredbi (#3)). Metodom `extend()` to ćemo učiniti na drugi način. Prvo ćemo naredbom (#17) definirati novu praznu listu `br`, na koju ćemo naredbom (#18) nadovezati listu `br_0`, a zatim naredbom (#19) i listu i `br_1`.

Ako nam izvorna lista `br_0` kasnije nije potrebna, nadovezivanje se može obaviti neposredno na nju. Pogledajmo:

```
>>> br_0 = [1, 2]
>>> br_1 = [3, 4]
>>> br_0.extend(br_1)                #20
>>> br_0
[1, 2, 3, 4]
```

Metoda `index()`

U nekim situacijama nećemo točno znati na kojem mjestu u listi se nalazi neka vrijednost. Tada će nam dobro doći metoda `index()` koja pronalazi indeks nekog elementa u listi. Pogledajmo kako ona djeluje:

```
>>> imena = ['Ana', 'Marko', 'Marija', 'Pero']                    #1
>>> imena.index('Marija')
2
>>> imena.indeks('Igor')                                           #2
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    imena.index('Igor')
ValueError: 'Igor' is not in list
```

U naredbi (#1) definirali smo listu od četiriju elemenata. Naredbom (#2) saznali smo da je indeks elementa čija je vrijednost `'Marija'` jednak `2`. Pokušamo li naći indeks neke vrijednosti koja se ne nalazi u listi kao u naredbi (#2) *Python* će nam javiti pogrešku i poruku da tog elementa nema u listi.

Ovaj bismo problem mogli izbjeći tako da provjerimo nalazi li se neka vrijednost u listi. Tu će nam pomoći operatori `in` ili `not in`.

Ako se u listi nalazi više elemenata s istom vrijednošću metoda `index()` vratit će indeks prvoga od njih.

Operatori `in` i `not in`

Operatori `in` i `not in` djeluju slično kao kao operatori usporebe:

```
>>> 'Marija' in imena
True
>>> 'Igor' in imena
False
>>> 'Marija' not in imena
False
>>> 'Igor' not in imena
True
```

Izrazi u naredbama vraćaju logičke vrijednosti `True` i `False`. Na taj način prije traženja indeksa možemo saznati nalazi li se dotični element u listi.

Uporabu operatora `in` ilustrira sljedeći program:

```
#Prva inačica rješenja - program_6_1_1.py

def indeks(element, lista):
    if element in lista:                                           #3
        return lista.index(element)                                #4
    else:
        return 'Nema tog imena u listi'                            #5

imena = ['Ana', 'Marko', 'Marij', 'Pero']
ime1 = input('Upiši ime: ')
print(indeks(ime1, imena))
ime2 = input('Upiši ime: ')
print(indeks(ime1, imena))
```

Nakon izvođenja programa mogl bi dobiti ispis:

```
Upiši ime: Marija
2
Upiši ime: Filip
Nema tog imena u listi
```

U funkciji se najprije naredbom (#3) ispituje nalazi li se traženi element u listi. Tek nakon potvrdnog odgovora, traži se i vraća njegov indeks naredbom (#4). Ako element nije u listi, naredbom (#5) vraća se tekst `Nema tog imena u listi`. U nekom drugom programu ta bi vraćena vrijednost mogla poslužiti za donošenje odluke o daljnjem postupanju. Postavlja se pitanje je li ta povratna vrijednost dobro odabrana i mora li to biti baš neki tekst.

Posebna vrijednost None

Za lakše razrješavanje takvih nedoumica u *Pythonu* postoji posebna vrijednost koja se zove None (engl. *none* – ništa, nitko, nijedan). Kada funkcija ne vraća neku određenu vrijednost, tj. kada iza ključne riječi return ne piše ništa, tada zapravo ona vraća vrijednost None. Uz malu promjenu u definiciji funkcije indeks() naš će program dobiti sljedeći oblik:

```
#Druga inačica rješenja - program_6_1_2.py

def indeks(element, lista):
    if element in lista:
        return lista.index(element)
    return

imena = ['Ana', 'Marko', 'Marija', 'Pero']
print(indeks('Marija', imena))
print(indeks('Igor', imena))
if indeks('Igor', imena) == None:
    print('Igora nema na popisu')
```

Metoda `append()`

Metodom `append()` jednostavno se dodaje novi element na kraj liste. Ta metoda djeluje neposredno na postojeću listu tako da nam nije potrebna naredba pridruživanja (na engleskom se kaže da metoda djeluje *in place* – na mjestu). Osim toga, novi element ne moramo pisati unutar uglatih zagrada pa dodavanje izgleda ovako:

```
>>> imena = ['Ana', 'Marija', 'Marko']
>>> imena.append('Pero')
>>> imena
['Ana', 'Marija', 'Marko', 'Pero']
```

Ubuduće ćemo nove elemente na kraj liste dodavati na ovaj način.

Pogledajmo neke primjere:

```
>>> niz_vrijednosti_i = []                                    #6
>>> for i in range(10):                                       #7
        niz_vrijednosti_i.append(i)

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

Na praznu listu (#5) će se u svakom prolazu petlje `for` dodati trenutačna vrijednost varijable petlje `i`. Prema tome, mi smo generatorskom funkcijom (#7) stvorili listu brojeva u intevalu [0, 9]. Vidjet ćemo kasnije da se takva lista može načiniti i jednostavnije.

```
>>> kvadrati = []
>>> for i in range(10):
        kvadrati.append(i * i)

>>> kvadrati
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> trostruke_vrijednosti = []
>>> for i in range(10):
        trostruke_vrijednosti.append(3 * i)

>>> trostruke_vrijednosti
[0, 3, 6, 9, 12, 15, 18, 21, 24, 27]
```

Listu nasumičnih brojeva iz intervala [1,100] možemo generirati ovako:

```
>>> from random import randint                              #8
>>> nasumični_brojevi = []                                  #9
>>> for i in range(10):                                     #10
        nasumični_brojevi.append(randint(1,100))

>>> nasumični_brojevi
[43, 74, 30, 58, 45, 59, 80, 56, 77, 20]
```

Metoda `insert()`

Druga metoda za umetanje elemenata u listu je metoda `insert(indeks, element)`. Ona za razliku od metode `append()` koja dodaje novi element na kraj postojeće liste umeće novi element na mjesto prije navedenog indeksa. Elementima iza umetnutog elementa se time indeks povećava za jedan. Pogledajmo:

```
>>> imena = ['Ana', 'Marko', 'Pero']                    #11
>>> imena.insert(1, 'Marija')                           #12
>>> imena
['Ana', 'Marija', 'Marko', 'Pero']
```

Početna lista zadana je naredbom (#11). Element `'Marija'` smjestit će se naredbom (#12) ispred elementa koji je u početnoj listi imao indeks `1`. Njegov se indeks povećao. To možemo lako ustanoviti:

```
>>> imena.index('Marko')
2
```

Metoda `remove()`

Želimo li izbaciti neki element s poznatom vrijednošću, tada je praktično rabiti metodu `remove()`. Pogledajmo kako ćemo izbaciti vrijednost `'Marko'` iz postojeće liste imena:

```
>>> imena = ['Ana', 'Marija', 'Marko', 'Pero', 'Zvonko']
>>> imena.remove('Marko')
>>> imena
['Ana', 'Marija', 'Pero', 'Zvonko']
```

Kada u listi ima više imena s jednakom vrijednošću, bit će izbačen samo onaj koji je u listi prvi po redu (s najmanjim indeksom).

Naredba `del`

Ako iz liste želimo izbaciti element s poznatim indeksom, tada je praktičnije rabiti naredbu `del`. Tako možemo pisati:

```
>>> del (imena[2])                                             #1
>>> imena
['Ana', 'Marija', 'Zvonko']
```

Djelovanjem funkcije (#1) izbacili smo iz liste element s indeksom `2`, tj. element s vrijednošću `'Pero'` tako da su u listi preostala samo tri elementa.

Metoda pop()

Brisanje metodom remove() i naredbom del gubi se potpuno vrijednost elementa jer ju prije brisanja ne pohranjujemo.

Međutim metoda pop() vraća vrijednost koju skida iz liste. Ona može uzeti element s bilo kojim indeksom:

```
>>> imena = ['Ana', 'Marija', 'Marko', 'Pero', 'Zvonko']
>>> ime = imena.pop(2)                                      #3
>>> ime
'Marko'
>>> imena
['Ana', 'Marija', 'Pero', 'Zvonko']
```

Kada na zadanu listu primijenimo metodu pop() kao u naredbi (#3) varijabla ime poprimit će vrijednost 'Marko' i ova će vrijednost istodobno biti izbačena iz liste.

Najčešće se iz liste uzimaju ili zadnji element sa pop() (pri čemu ne treba pisati indeks zadnjeg elementa) ili prvi element sa pop(0).

Kao što metoda append() "zna" dodavati elemente na kraj liste, tako i metoda pop() "zna" uzimati elemente s kraja liste bez obzira na njezinu duljinu.

Metoda pop()

Brisanje metodom remove() i naredbom del gubi se potpuno vrijednost elementa jer ju prije brisanja ne pohranjujemo.

Međutim metoda pop() vraća vrijednost koju skida iz liste. Ona može uzeti element s bilo kojim indeksom:

```
>>> imena = ['Ana', 'Marija', 'Marko', 'Pero', 'Zvonko']
>>> ime = imena.pop(2)                                        #3
>>> ime
'Marko'
>>> imena
['Ana', 'Marija', 'Pero', 'Zvonko']
```

Kada na zadanu listu primijenimo metodu pop() kao u naredbi (#3) varijabla ime poprimit će vrijednost 'Marko' i ova će vrijednost istodobno biti izbačena iz liste.

Najčešće se iz liste uzimaju ili zadnji element sa pop() (pri čemu ne treba pisati indeks zadnjeg elementa) ili prvi element sa pop(0).

Kao što metoda append() "zna" dodavati elemente na kraj liste, tako i metoda pop() "zna" uzimati elemente s kraja liste bez obzira na njezinu duljinu.

Detaljniji opisi funkcija `range()` *i* `list()`

Naučili smo da postoji funkcija `range(n)` koja nam u svakom prolazu petlje `for` daje redom vrijednosti `0, 1, 2, ..., n-1`. Te vrijednosti obično pridružujemo varijabli koju zovemo varijablom petlje. Pogledajmo to još jednom:

```
>>> for i in range(10):
        print(i, end=' ')

0 1 2 3 4 5 6 7 8 9
```

Postavlja se pitanje možemo li unaprijed znati koje će nam vrijednosti redom vraćati funkcija `range()`?

U tome nam može pomoći funkcija `list()` koja i inače služi za stvaranje lista. Ta će nam funkcija načiniti listu svih vrijednosti koje će generirati `range()` te ih možemo vidjeti sve odjednom:

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(20))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

Funkcija `range()` se može pozvati i trima parametrima `range(start,stop,korak)`, gdje je:

- `start` početna vrijednost niza
- `stop` gornja granica niza (pri čemu će najveći član niza uvijek biti za jedan manji od `stop`)
- korak razlika između dvaju uzastopnih članova niza.

Primjenom funkcije `list()` možemo vidjeti koje vrijednosti nam daje funkcija `range()`:

```
>>> list(range(1, 20))                                                   #1
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
>>> list(range(10, 20))                                                  #2
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
>>> list(range(2, 20, 2))                                                #3
[2, 4, 6, 8, 10, 12, 14, 16, 18]
>>> list(range(1, 20, 2))                                                #4
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
```

Ako se u pozivu funkcije `range()` pojavljuju dva parametra, smatra se da su to parametri `start` i `stop`. Naredbom (#1) generirat ćemo niz prirodnih brojeva od `1` do `19`. Naredbom (#2) generirat će se niz koji počinje brojem `10` i opet završava sa `19`. Od ranije znamo da početni broj možemo izostaviti kada želimo da niz započne sa `0`. U naredbama (#3) i (#4) pojavljuju se sva tri parametra. Naredbom (#3) dobili smo niz parnih brojeva, a naredbom (#4) neparne brojeve.

Funkcija `list()` može se primijeniti i na neke druge tipove podataka. Tako se ona može primijeniti i na tip `string`. Pogledajmo to u interaktivnom sučelju:

```
>>> list('abcdef')                                                    #5
['a', 'b', 'c', 'd', 'e', 'f']
>>> list(123456)                                                      #6
Traceback (most recent call last):
  File "<pyshell#70>", line 1, in <module>
    list(123456)
TypeError: 'int' object is not iterable
>>> list(str(1234569))                                                #7
['1', '2', '3', '4', '5', '6', '9']
```

Naredbom (#5) string će se rastaviti tako da svaki znak stringa postaje string jedinične duljine, a nakon toga se smješta u listu. Pokušaj pretvaranja broja u listu znamenaka u naredbi (#6) ne uspijeva jer se funkcija `list()` ne može primijeniti na tip `int`.

No, znamo da se tip `int` može pretvoriti u tip `string` funkcijom `str`, pa smo naredbom (#7) uspjeli dobiti listu znamenaka broja u obliku stringova. Sada treba još samo stringove znamenaka prevesti u tip `int`.

Primjer:

Napišimo funkciju koja će vraćati listu znamenaka višezamenkastog broja. Nadalje napišimo program koji će unositi višeznamenkasti prirodan broj i ispisivati listu njegovih znamenaka.

```
#Prvo rješenje - program_6_2_1.py
def znamenke(broj):
    z = list(str(broj))                    #8
    znam = []                              #9
    for i in range(len(z)):                #10
        znam.append(int(z[i]))
    return znam

broj = int(input('unesi višeznamenkasti prirodan broj: '))
print(znamenke(broj))
```

Naredba (#8) načinit će listu znamenaka kao i naredba (#7) u prethodnom primjeru. Praznu listu `znam` koju smo uveli naredbom (#9) ćemo na već poznati način u petlji (#10) nadopunjavati tako da redom sve elemente liste `z` pretvaramo iz tipa `str` u tip `int`.

Nakon izvođenja ovog programa ispis će biti:

```
unesi višeznamenkasti prirodan broj: 123456
[1, 2, 3, 4, 5, 6]
```

Ostvarenje petlje `for` uz pomoć liste i funkcije `enumerate()`

Dosad smo sve petlje `for` ostvarivali uz pomoć funkcije `range(n)`. Funkcija `range(n)` za svaki prolaz generira jednu vrijednost iz niza `0, 1, 2, …, n-1` pri čemu te vrijednosti nisu prethodno pohranjene u spremniku računala. Nova se vrijednost određuje na početku svakog prolaza (svake iteracije). To je posebno važno ako je `n` veliki broj.

```
>>> for i in range(10):                                    #1
        print(i, end=' ')
0 1 2 3 4 5 6 7 8 9
>>> for i in range(1, 1000):                               #2
        if i % 73 == 0:
            print(i, end=' ')

73 146 219 292 365 438 511 584 657 730 803 876 949
```

U petlji (#1) varijabla `i` poprimit će samo deset vrijednosti, dok će u petlji (#2) ona tijekom izvođenja petlje poprimati ukupno tisuću vrijednosti (petlja ispisuje višekratnike broja 73 koji su manji od 1000). Jako je važno da tih tisuću vrijednosti koje ispitujemo ne moramo prethodno pohraniti.

U *Pythonu* se može odrediti i petlja u kojoj nam nije potrebna funkcija `range()` već se može tražiti da se blok naredbi ponavlja s vrijednošću svakog elementa neke liste. Tako možemo pisati:

```
>>> imena = ['Ana', 'Marija', 'Marko', 'Pero',]                    #3
>>> for ime in imena:
        print('Lijep pozdrav {}!'.format(ime))                     #4

Lijep pozdrav Ana!
Lijep pozdrav Marija!
Lijep pozdrav Marko!
Lijep pozdrav Pero!
```

Za listu od četiriju elemenata zadanu naredbom (#3) petlja će se ponavljati četiri puta jer je toliko elemenata u listi. Varijabla kojoj smo odabrali naziv `ime` u svakom će prolazu poprimiti vrijednost jednog od tih četiriju elemenata liste `imena`. Kao što vidimo, u naredbi (#4) ne pojavljuje se indeks kojim se dohvaća pojedini element. Isto tako, ne spominje se ni duljina liste.

No, ako nam je zbog nekog razloga potreban i indeks elementa, postoji mogućnost da se pri obilasku liste primijeni funkcija `enumerate()` koja uz vrijednost elemenata liste vraća i njegov indeks. Pogledajmo to u interaktivnom sučelju:

```
>>> for i, ime in enumerate(imena):                                          #5
        print('{} ima indeks {}'.format(ime, i))                              #6

Ana ima indeks 0
Marija ima indeks 1
Marko ima indeks 2
Pero ima indeks 3
```

U naredbi (#5) pojavljuju se dvije varijable. Prva od njih poprimat će vrijednost indeksa elementa, a druga vrijednost samog elementa. Naredbom (#6) ispisat ćemo u svakom prolazu dobivene vrijednosti.

Pogledajmo još jedan primjer u kojem želimo u jednu listu pohraniti sve višekratnike broja 73 manje od tisuću. U naredbi (#2) na početku ovog odjeljka ispisali smo sve te brojeve. Umjesto ispisivanja pohranit ćemo ih u neku listu na sljedeći način:

```
>>> brojevi_73 = []
>>> for i in range(1, 1000):
        if i % 73 == 0:
            brojevi_73.append(i)

>>> brojevi_73
[73, 146, 219, 292, 365, 438, 511, 584, 657, 730, 803, 876, 949]
```

Listu `brojevi_73` dobili smo na već poznati način. Želimo li pogledati koji su od tih brojeva djeljivi još i s brojem 3, onda nema potrebe ponovno pregledavati svih 999 brojeva već je dovoljno ispitati jesu li brojevi iz liste `brojevi_73` djeljivi sa 3. Prema tome, možemo napisati petlju koja će obići sve elemente te liste i ispitati njihovu djeljivost brojem 3:

```
>>> brojevi_73_3 = []
>>> for x in brojevi_73:                                                    #9
        if x % 3 == 0:
            brojevi_73_3.append(x)
```

Nakon što smo definirali praznu listu `brojevi_73_3`, u petlji (**#9**) koja obilazi sve elemente liste `brojevi_73` postupno ćemo popunjavati listu `brojevi_73_3` samo onim elementima izvorne liste koji su djeljivi brojem 3.

```
>>> brojevi_73_3
[219, 438, 657, 876]
```

Konstruiranje lista

Već smo dosad mogli primijetiti da su liste vrlo korisne pri rješavanju programskih zadataka. Zbog toga u *Pythonu* postoji posebni postupak za konstruiranje lista. U engleskom jeziku se taj postupak zove *list comprehensions*. Ovakav način konstruiranja lista ne mora se nužno koristiti u prvim pokušajima pisanja programa, ali se pokazalo da malo iskusniji programeri mogu na taj način pripremiti vrlo sažeta i pregledna programska rješenja.

Dosad smo naučili da se nova lista može stvoriti tako da se iz neke postojeće liste uzimaju vrijednosti elemenata (ili se vrijednosti generiraju funkcijom `range()`) i na temelju tih vrijednosti određuju vrijednosti elemenata nove liste koji se zatim redom stavljaju u novu listu. Uz to se nekada ispituju i dodatni uvjeti koji određuju hoće li element biti stavljen u novu listu ili neće.

Konstruiranje se može provesti operacijom konstrukcije:

```
>>> kvadrati = [i * i for i in range(10)]                          #1
>>> kvadrati
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> trostruke_vrijednosti = [3 * i for i in range(10)]             #2
>>> trostruke_vrijednosti
[0, 3, 6, 9, 12, 15, 18, 21, 24, 27]
>>> from random import randint
>>> nasumični_brojevi = [randint(10, 99) for i in range(10)]       #3
>>> nasumični_brojevi
[11, 84, 56, 28, 27, 71, 53, 56, 36, 17]
```

U naredbama (#1) i (#2) vidimo da se lista konstruira tako da unutar uglatih zagrada prvo napišemo kako će se odrediti (izračunati) vrijednost pojedinog elementa i zatim se u obliku svojevrsne petlje određuje dobavljanje varijable i uz pomoć funkcije `range()`. U naredbi (#3) vrijednost elementa ne ovisi o varijabli petlje, a funkcija `range(10)` određuje samo koliko će elemenata biti u novoj listi.

Vještim konstruiranjem možemo znatno pojednostavniti programe. Tako umjesto funkcije `znamenke(broj)` iz `program_6_2_1.py` možemo dobiti listu znamenki nekog broja mnogo jednostavnije:

```
>>> broj = 123456
>>> znam = [int(x) for x in list(str(broj))]                    #4
>>> znam
[1, 2, 3, 4, 5, 6]
```

Izraz (#4) kojim smo konstruirali listu `znam` je u neku ruku nastao prepisivanjem naredbi iz definicije funkcije `znamenke()`. Iz tog se izraza vidi da varijablu `x` uzimamo iz liste u kojoj su elementi stringovi pojedinih znamenaka. Na svaki taj element primjenjuje se funkcija `int()` koja će string znamenke prevesti u broj.

Funkcija `znamenke()` može se sada napisati na jednostavniji način:

```
>>> def znamenke(broj):
        return [int(x) for x in list (str(broj))]

>>> znamenke(123456)
[1, 2, 3, 4, 5, 6]
>>> znamenke(478)
[4, 7, 8]
```

Prema tome, za konstruiranje lista postoje različite mogućnosti. Tako možemo pisati:

```
>>> x = list(range(1, 11))                                   #5
>>> x
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> def dva_puta(x):
        return [2 * a for a in x]                            #6

>>> y = dva_puta(x)
>>> y
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

Naredbom (#5) smo generirali početnu listu `x` koja se sastoji od deset elemenata. Funkcija `dva_puta()` vraća konstruiranu listu u kojoj je svaki element izvorne liste pomnožen s dva (#6). Pri pozivu funkcije `dva_puta(x)` ona vraća listu s udvostručenim vrijednostima. Jednako tako, mogli bismo napisati i funkciju koja utrostručuje elemente početne liste:

```
>>> def tri_puta(x):
        return [3 * a for a in x]

>>> y = tri_puta(x)
>>> y
[3, 6, 9, 12, 15, 18, 21, 24, 27, 30]
```

Bilo bi razumno napisati jednu funkciju kojoj se uz izvornu listu kao parametar prenosi i faktor s kojim ju treba pomnožiti. To bi moglo izgledati ovako:

```
>>> def n_puta (x, n):                                    #7
        return [n * a for a in x]                         #8

>>> y = n_puta(x, 5)                                      #9
>>> y
[5, 10, 15, 20, 25, 30, 35, 40, 45, 50]
>>> print(n_puta(x, 8))                                   #10
[8, 16, 24, 32, 40, 48, 56, 64, 72, 80]
```

Funkcija definirana sa (#7) i (#8) vratit će listu u kojoj su elementi *n* puta veći od elemenata izvorne liste `x`. Naredbe (#9) i (#10) pokazuju djelovanje te funkcije.

```
>>> def tri_puta(x):
        return [3 * a for a in x]

>>> y = tri_puta(x)
>>> y
[3, 6, 9, 12, 15, 18, 21, 24, 27, 30]
```

Bilo bi razumno napisati jednu funkciju kojoj se uz izvornu listu kao parametar prenosi i faktor s kojim ju treba pomnožiti. To bi moglo izgledati ovako:

```
>>> def n_puta (x, n):                                    #7
        return [n * a for a in x]                         #8

>>> y = n_puta(x, 5)                                      #9
>>> y
[5, 10, 15, 20, 25, 30, 35, 40, 45, 50]
>>> print(n_puta(x, 8))                                   #10
[8, 16, 24, 32, 40, 48, 56, 64, 72, 80]
```

Funkcija definirana sa (#7) i (#8) vratit će listu u kojoj su elementi *n* puta veći od elemenata izvorne liste `x`. Naredbe (#9) i (#10) pokazuju djelovanje te funkcije.

Primjer:

Pogledajmo još jedan primjer uporabe konstrukcije lista. Pripremit ćemo funkciju koja će nam ispisati tablicu množenja prirodnih brojeva. Pogledajmo:

```
>>> def tablica_množenja(n):
        x = list(range(1, n + 1))                          #11
        for i in range(1, n + 1):
            z = [a * i for a in x]                         #12
            for j in range(n):                             #13
                print('{:3d}'.format(z[j]), end=' ')       #14
print()                                                    #15
```

Funkcijom `list()` u naredbi (#11), definiramo listu prirodnih brojeva iz intervala `[1, n]`. Nakon toga, naredbom (#12) konstruirat ćemo listu `z` u kojoj su elementi liste `x` pomnoženi s vrijednošću varijable petlje `i`. Elemente kreirane liste ispisujemo jedan po jedan u petlji u istom redu naredbama (#13) i (#14). Naredbom (#15) prelazimo nakon ispisa jednog reda tablice množenja u novi red. Pozivanjem te funkcije dobivamo ovakve ispise tablice množenja:

Primjer:

Pogledajmo još jedan primjer uporabe konstrukcije lista. Pripremit ćemo funkciju koja će nam ispisati tablicu množenja prirodnih brojeva. Pogledajmo:

```
>>> def tablica_množenja(n):
        x = list(range(1, n + 1))                              #11
        for i in range(1, n + 1):
            z = [a * i for a in x]                             #12
            for j in range(n):                                 #13
                print('{:3d}'.format(z[j]), end=' ')           #14
            print()                                            #15
```

Funkcijom `list()` u naredbi (#11), definiramo listu prirodnih brojeva iz intervala `[1, n]`. Nakon toga, naredbom (#12) konstruirat ćemo listu `z` u kojoj su elementi liste `x` pomnoženi s vrijednošću varijable petlje `i`. Elemente kreirane liste ispisujemo jedan po jedan u petlji u istom redu naredbama (#13) i (#14). Naredbom (#15) prelazimo nakon ispisa jednog reda tablice množenja u novi red. Pozivanjem te funkcije dobivamo ovakve ispise tablice množenja:

```
>>> tablica_množenja(5)
  1   2   3   4   5
  2   4   6   8  10
  3   6   9  12  15
  4   8  12  16  20
  5  10  15  20  25
```

```
>>> tablica_množenja(15)
   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15
   2   4   6   8  10  12  14  16  18  20  22  24  26  28  30
   3   6   9  12  15  18  21  24  27  30  33  36  39  42  45
   4   8  12  16  20  24  28  32  36  40  44  48  52  56  60
   5  10  15  20  25  30  35  40  45  50  55  60  65  70  75
   6  12  18  24  30  36  42  48  54  60  66  72  78  84  90
   7  14  21  28  35  42  49  56  63  70  77  84  91  98 105
   8  16  24  32  40  48  56  64  72  80  88  96 104 112 120
   9  18  27  36  45  54  63  72  81  90  99 108 117 126 135
  10  20  30  40  50  60  70  80  90 100 110 120 130 140 150
  11  22  33  44  55  66  77  88  99 110 121 132 143 154 165
  12  24  36  48  60  72  84  96 108 120 132 144 156 168 180
  13  26  39  52  65  78  91 104 117 130 143 156 169 182 195
  14  28  42  56  70  84  98 112 126 140 154 168 182 196 210
  15  30  45  60  75  90 105 120 135 150 165 180 195 210 225
```

Zbirka podataka `tuple`

U *Pythonu* postoji zbirka podataka koja se zove `tuple`. U hrvatskom jeziku taj tip podataka možemo zvati *n*-torka (i u engleskom jeziku se također susreće pojam *n-tuple*). Tip `tuple` je u neku ruku sličan tipu `list`. Zadavanje *n*-torki obavljamo slično kao i liste samo što umjesto uglatih zagrada rabimo okrugle zagrade ili naprosto pišemo nizove elemenata odvojene zarezima. Pogledajmo:

```
>>> prosti_brojevi = (2, 3, 5, 7, 11, 13, 17, 19)        #1
>>> neparni_brojevi = 1, 3, 5, 7, 9, 11, 13, 15, 17, 19  #2
```

Elemente *n*-torki dohvaćamo na isti način kao i elemente lista. Tako se do pojedinog člana *n*--torke može doći uporabom indeksa. Pogledajmo:

```
>>> for i in range(len(prosti_brojevi)):                  #3
        print(prosti_brojevi[i], end=' ')                 #4

2 3 5 7 11 13 17 19
>>> for x in neparni_brojevi:                             #5
        print(x, end=' ')                                 #6

1 3 5 7 9 11 13 15 17 19
```

Naredbama (#1) i (#2) zadali smo *n*-torke koje se sastoje od prvih osam prostih brojeva odnosno od prostih brojeva manjih od 20. Iz naredbe (#3) vidljivo je da na *n*-torke djeluje funkcija `len()`, a u naredbi (#4) do pojedinih elemenata *n*-torke dolazimo s pomoću indeksa koje kao i kod lista pišemo unutar uglatih zagrada. Isto tako, petlja `for` može se ostvariti *n*-torkom na isti način kao i s listom što pokazuju naredbe (#5) i (#6).

Za *n*-torku (tip `tuple`) vrijede i drugi operatori i funkcije koje rabimo za liste, ali samo oni koji ne mijenjaju elemente *n*-torke. U tome i jest osnovna razlika između *n*-torki i lista. Pokušaj promjene *n*-torke nije dozvoljen:

```
>>> prosti_brojevi[4] = 23                                                    #7
Traceback (most recent call last):
  File "<pyshell#177>", line 1, in <module>
    prosti_brojevi[4] = 23
TypeError: 'tuple' object does not support item assignment
>>> prosti_brojevi.append(23)                                                 #8
Traceback (most recent call last):
  File "<pyshell#178>", line 1, in <module>
    prosti_brojevi.append(23)
AttributeError: 'tuple' object has no attribute 'append'
```

Naredba (#7) u kojoj jednom elementu *n*-torke pokušavamo promijeniti vrijednost ne uspijeva kao ni dodavanje novog elementa naredbom (#8).

Prema tome, zaključimo da su sadržaji *n*-torki nepromijenljivi, što u neku ruku zaštićuje sadržaje pohranjene u njih od neželjenih promjena.

Međutim, ako baš namjerno želimo mijenjati sadržaj *n*-torke i za to postoji mogućnost. Naime, funkcijom `list()` možemo *n*-torku pretvoriti u listu, u toj listi načiniti promjenu i nakon toga funkcijom `tuple()` tu promijenjenu listu opet pretvoriti u *n*-torku. Evo primjera:

```
>>> lista_a = list(prosti_brojevi)                    #9
>>> lista_a
[2, 3, 5, 7, 11, 13, 17, 19]
>>> lista_a.append(23)                                #10
>>> lista_a.append(29)                                #11
>>> lista_a
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
>>> prosti_brojevi = tuple(lista_a)                   #12
>>> prosti_brojevi
(2, 3, 5, 7, 11, 13, 17, 19, 23, 29)
```

Nakon što naredbom (#9) *n*-torku pretvorimo u listu, možemo u njoj načiniti željene promjene. Naredbama (#10) i (#11) dodali smo u nju još dva prosta broja i na kraju naredbom (#12) pretvorili tu listu u promijenjenu *n*-torku. Dakle, promjena sadržaja *n*-torki je na ovakav zaobilazni način moguća, ali je mogućnost nenamjerne pogreške bitno smanjena jer je postupak promjene sadržaja puno složeniji nego kod lista. Osim toga, prikazani način rada s *n*-torkama najčešće nema smisla. Naime, ako nam je u programu potrebna struktura podataka nad kojom trebamo često raditi izmjene sadržaja, onda treba umjesto *n*-torki rabiti liste. Primjena *n*-torki ima smisla za one podatke čiji se sadržaj ne mijenja tijekom izvođenja programa.

Pokazali smo da pojedine elemente *n*-torki možemo dohvatiti njihovim indeksom. No, ako broj elemenata nije prevelik, *n*-torke možemo *raspakirati* jednom naredbom pridruživanja tako da s lijeve strane znaka pridruživanja napišemo točno onoliko imena varijabli koliko u *n*-torki ima elemenata. Tako možemo pisati:

```
>>> četvorka = (1, 2, 3, 4)                                   #13
>>> a, b, c, d = četvorka                                     #14
>>> print(četvorka)                                           #15
(1, 2, 3, 4)
>>> print('a = {} b = {} c = {} d = {}'.format(a, b, c, d))
a = 1 b = 2 c = 3 d = 4
```

Naredbom (#13) definirali smo *n*-torku četvorka te ju u naredbi (#14) raspakirali u četiri varijable a, b, c i d. Ispis obavljen naredbom (#15) pokazuje da je to raspakiranje korektno obavljeno.

Dosad nismo spomenuli da se na jednak način mogu raspakirati i liste. Katkada to može biti korisno i pojednostavnit će pisanje programa posebice ako se radi o kraćim listama. Pogledajmo:

```
>>> lista = [1, 2, 3, 4]                                      #16
>>> a, b, c, d = lista                                        #17
>>> print(lista)
[1, 2, 3, 4]
>>> print('a = {} b = {} c = {} d = {}'.format(a, b, c, d))
a = 1 b = 2 c = 3 d = 4
```

U naredbi (#16) definirali smo listu koja je zatim u naredbi (#17) raspakirana u četiri varijable na isti način kao što je u naredbi (#14) raspakirana *n*-torka.

Tip tuple najčešće se rabi za vraćanje vrijednosti funkcija. Vrijednosti koje funkcija vraća moraju se sve napisati iza ključne riječi return i međusobno ih treba razdvojiti zarezima. Pogledajmo sljedeći primjer:

```
>>> def aritmetičke_operacije(a, b):
        zbroj = a + b
        razlika = a - b
        umnožak = a *  b
        količnik, ostatak = divmod(a, b)                              #18
        return zbroj, razlika, umnožak, količnik, ostatak             #19

>>> aritmetičke_operacije(19, 7)                                      #20
(26, 12, 133, 2, 5)
>>> aritmetičke_operacije(23, 11)                                     #21
(34, 12, 253, 2, 1)
```

Funkcija aritmetičke_operacije() vratit će rezultate pet osnovnih operacija za prirodne brojeve. Prisjetimo se da smo upoznali funkciju divmod() koja vraća *n*-torku dvaju elemenata koje smo nazivali parom vrijednosti. Taj je par u naredbi (#18) raspakiran u varijable količnik i ostatak. U naredbi (#19) smo iza ključne riječi return napisali sve varijable čije vrijednosti vraća funkcija. Kada pozovemo funkciju s konkretnim vrijednostima parametara a i b, ona će vratiti *n*-torku s pet elemenata (petorku) rezultata. To možemo provjeriti promatranjem rezultata dobivenih naredbama (#20) i (#21).

Funkciju za provođenje aritmetičkih operacija možemo pojednostavniti tako da rezultate aritmetičkih operacija ne smještamo najprije u varijable i zatim te varijable poimence pišemo iza naredbe `return` već da uz naredbu `return` napišemo izraze koji će izračunavati sve rezultate (#22).

```
>>> def aritmetičke_operacije(a, b):
        return a + b, a - b, a * b, a // b, a % b                    #22

>>> aritmetičke_operacije(19, 7)
(26, 12, 133, 2, 5)
>>> aritmetičke_operacije(23, 11)
(34, 12, 253, 2, 1)
```

Primjer:

Napišimo program koji će služiti za vježbanje pet osnovnih aritmetičkih operacija. Program sam treba nasumično odabirati aritmetičku operaciju i operande iz intervala [1, N]. Na početku programa treba zadati gornju granicu intervala. Taj bi program mogao izgledati ovako:

```python
#Program za vježbanje arimetičkih operacija - program_6_3.py

from random import randint

def aritm_oper(a, b):
    return a + b, a - b, a * b, a // b, a % b                        #1

def main():
    print('Program zadaje zadatke nasumičnim odabirom')
    print('jedne od aritmetičkih operacija: +, -, *, //, %.')
    print('Operandi se odabiru nasumično iz intervala [1, N].\n')
    n = int(input('Odaberi gornju granicu intervala N = '))
    print('Nakon upisa rezultata pritisni tipku (Unos).')
    print('Za završetak vježbanja pritisni samo tipku (Unos).')
    print('\nVježbanje počinje!\n')
    operatori = ('+', '-', '*', '//', '%')                           #2
    while True:
        op = randint(0,  4)                                           #3
        a = randint(1, n)                                             #4
        b = randint(1, n)                                             #5
        upis_rezultata = input('{} {} {} = '.format(a, operatori[op], b)) #6
        if not upis_rezultata:                                        #7
            print('Vježbanje je završeno!')
            break
        rezultat = aritm_oper(a, b)
        if int(upis_rezultata) == rezultat[op]:                       #8
            print('Odgovor je točan!')
        else:
            print('Odgovor je netočan!')

main()
```

Funkcija `aritm_oper()` pri svakom pozivu vraća rezultate svih pet aritmetičkih operacija kao petorku brojeva (#1).

U funkciji `main()` naredbom (#3) nasumično se generira indeks `op` iz intervala [0, 4]. Taj indeks će u naredbi (#9) odabrati iz vraćene petorke jedan od rezultata. Naredbom (#2) uvedena je petorka simbola pojedinih operacija. Iz te će petorke indeks `op` odabrati simbol operacije koji će u naredbi (#6) biti ispisan između brojeva `a` i `b`. Ti se brojevi nasumično generiraju naredbama (#4) i (#5).

Obratimo pažnju na to da se naredbom (#6) unosi string koji se ispituje u naredbi `if` (#7). Ako je string prazan, tj. ako smo samo pritisnulli tipku *(Unos)*, uvjet `not upis_rezultata` bit će ispunjen te će djelovati naredba `break`. Ako je upisan rezultat, on će u naredbi (#8) biti preveden u tip `int` i uspoređen s rezultatom koji je vratila funkcija `aritm_oper()`.

Ispis nakon pokretanja programa mogao bi izgledati ovako:

```
Program zadaje zadatke nasumičnim odabirom
jedne od aritmetičkih operacija: +, -, *, //, %.
Operandi se odabiru nasumično iz intervala [1, N].

Odaberi gornju granicu intervala N = 20
Nakon upisa rezultata pritisni tipku (Unos).
Za završetak vježbanja pritisni samo tipku (Unos).

Vježbanje počinje!

20 - 16 = 4
Odgovor je točan!
19 // 15 = 1
Odgovor je točan!
16 - 14 = 2
Odgovor je točan!
18 % 2 = 0
Odgovor je točan!
10 // 8 = 1
Odgovor je točan!
20 * 19 = 390
Odgovor je netočan!
14 * 4 = 56
Odgovor je točan!
16 // 7 =
Vježbanje je završeno!
```