Uvod u programski jezik Python

Aleksandar Stojanović • Željko Kovačević

Informatičko-računarski odjel
Tehničko veleučilište u Zagrebu
© 2022.

UDŽBENICI TEHNIČKOG VELEUČILIŠTA U ZAGREBU
MANUALIA POLYTECHNICI STUDIORUM ZAGRABIENSIS

Aleksandar Stojanović • *Željko Kovačević*

UVOD U PROGRAMSKI JEZIK PYTHON

Udžbenik na kolegiju "Programiranje za forenziku" koji se održava u sklopu nastave na diplomskom stručnom studiju informatike Tehničkog veleučilišta u Zagrebu.

Zagreb, 2022.

izdavač	Tehničko veleučilište u Zagrebu Vrbik 8, Zagreb
za izdavača	izv. prof. dr. sc. Jana Žiljak Gršić
autori	dr. sc. Aleksandar Stojanović dr. sc. Željko Kovačević
recenzenti	prof. dr. sc. Danijel Radošević dr. sc. Frane Urem
vrsta djela	udžbenik Objavljivanje je odobrilo Stručno vijeće Tehničkog veleučilišta u Zagrebu odlukom broj 4932-6/22 od 15. ožujka 2022.
ISBN	978-953-7048-99-0
CIP zapis	Dostupan u računalnom katalogu Nacionalne i sveučilišne knjižnice u Zagrebu pod brojem 001133159.

Sadržaj

1 Uvod

1.1 Neke osobine Pythona

Za one koji su programirali u jezicima kao što su C/C++, Java, C# i sličnim programskim jezicima, tri su ključne razlike između tih jezika i Pythona:

- U Pythonu varijable nemaju oznaku tipa iako pojam tipa podatka postoji. To znači da nekoj varijabli možemo pridružiti vrijednost bilo kojeg tipa. Nadalje, varijabla u Pythonu ne mora uvijek sadržavati vrijednost istog tipa, to jest, u jednom trenutku može sadržavati, recimo, znakovni niz, a u drugom cijeli broj.

- U Pythonu memorija se oslobađa automatski. Drugim riječima, ne postoji naredba kojom se eksplicitno uklanja ili dealocira objekt iz memorije. To je omogućeno *sakupljačem smeća* (engl. *garbage collector*) sustavom [1], sustavom za automatsko upravljanje memorijom koji je dio Pythonovog izvršnog sustava. To znači da programer ne mora voditi računa o tome na kojem mjestu u programu treba neki objekt obrisati, niti mora misliti na potencijalno „curenje memorije“ (engl. *memory leak*), što je jedan od čestih problema s jezicima koji nemaju automatsko upravljanje memorijom. U Pythonu automatsko upravljanje memorijom radi na sličan način kao i u jezicima Java i C#.

- Program pisan u Pythonu ne prevodi se na *prirodni kôd* (engl. *native code*) dotičnog računala nego se izvršava pomoću drugog programa koji se zove *interpreter* [1, 2]. Iz tog razloga programi pisani u Pythonu izvršavaju se sporije od onih pisanih u jezicima koji izvorni kôd prevode na prirodni, kao što su C i C++. Zbog toga i zbog načina izvršavanja Python nije idealan jezik za izradu programa kod kojih je brzina izvršavanja važan aspekt, kao što su sistemski programi (na primjer, operacijski sustavi), programi za upravljanje hardverskim komponentama (engl. *driver*), razni kontroleri i sl. Međutim, u zadnje vrijeme Python se sve više upotrebljava za programiranje *ugrađenih sustava* (engl. *embedded systems*) [3], čime je primjena tog jezika ušla u prostor koji je nekad bio uglavnom rezerviran za jezike C i C++.

Jedna, možda i najznačajnija prednost Pythona u odnosu na druge popularne programske jezike je ta da za njega postoji, pored njegovih standardnih biblioteka [4], i

velik broj biblioteka za mnoga područja primjene, kao što su analiza podataka [5], web [6], strojno učenje i umjetna inteligencija [7, 8, 9, 10, 11], mrežno programiranje [12, 13] i mnoge druge. Ovo posljednje, umjetna inteligencija, područje je u kojem je Python trenutno najpopularniji programski jezik [14]. Nadalje, s obzirom da je Python programski jezik otvorenog koda (engl. *open source*) [15] za njega postoje nebrojeni izvori informacija, od knjiga i časopisa, do online izvora kao što su YouTube i mnoge web stranice. Službena mrežna stranica za Python je `http://www.python.org`.

1.2 Standardno razvojno okruženje Pythona

Python se isporučuje sa standardnim razvojnim okruženjem koje se zove *IDLE* (Integrated Development and Learning Environment). Nakon pokretanja IDLE-a dobije se komandna linija u koju se može unositi programski kôd. Nakon unosa, po pritisku tipke Enter ili Return ispod same linije, dobije se rezultat unesenog kôda.

```
>>> 2 + 3
5
```

U nastavku ćemo koristiti znak '#' za komentare u kodu, gdje će komentar biti bilo koji tekst koji se proteže od znaka '#' do kraja fizičkog reda. Na primjer,

```
>>> 2 + 3    # kod
5
```

U Pythonu postoje i komentari koji obuhvaćaju više redova i koji se stavljaju između jednostrukih ili dvostrukih navodnika.

1.2.1 Glavni modul

Iako je rad u komandnoj liniji koristan za kratke programe te razna isprobavanja i testiranja, za veće programe on je nepraktičan, a i to mu nije glavna namjena. U IDLE-u možemo odabrati stavku izbornika *File / New File*, nakon čega će se otvoriti uređivač teksta. Tako uneseni program možemo pokrenuti stavkom izbornika *Run / Run Module*. Prije pokretanja programa taj program moramo pohraniti u neku datoteku s nastavkom *.py*. Svaka datoteka s kodom pisanim u Pythonu mora imati nastavak *.py* i naziva se *modulom* [16]. To znači da se svaki program pisan u Pythonu mora sastojati od barem jednog modula. Modul iz kojeg pokrećemo program glavni je modul i njegova datoteka često se zove *main.py*, iako to nije obavezno. O modulima će biti puno više riječi kasnije.

1.2.2 Komandna linija

Python ne moramo nužno koristiti pomoću okruženja IDLE već ga možemo koristiti i preko komandne linije operacijskog sustava na kojem radimo. U sustavu Windows, na primjer, upisivanjem naredbe `py` (pod uvjetom da je Python instaliran i da je putanja u varijabli PATH ispravno postavljena) dobijemo sličnu komandnu liniju kao i u IDLE-u:

```
C:\Users\...\test> py
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 22:22:05) ...
Type "help", "copyright", "credits" or "license" for ...
>>>
```

Iz ovoga možemo izaći tipkama Ctrl-Z + Enter (ili Return) ili naredbom `exit()`.

Ako želimo pokrenuti program izrađen u Pythonu koji se nalazi u nekom modulu, ime tog modula možemo dodijeliti naredbi `py` kao parametar, čime će ga Pythonov interpreter izvršiti. Pretpostavimo da se u modulu `proba.py` nalazi sljedeći jednostavni program:

```
1 print(2*7)
```

Taj program možemo pokrenuti na sljedeći način:

```
C:\Users\...\test> py proba.py
14
```

Još jedan način pokretanja Python programa preko komandne linije je taj da se sâm izvorni kôd zada kao parametar naredbi `py` pomoću oznake `-c`:

```
C:\Users\...\test> py -c 'print(2*7)'
14
```

Ovdje smo jednostavan program za ispisivanje rezultata umnoška `2*7` dali kao parametar naredbi `py`, čime ga je Pythonov interpreter trenutačno izvršio. Ovaj način pokretanja programa dobar je za vrlo kratke programe koji se sastoje od nekoliko naredbi.

2 Osnovni elementi Pythona

2.1 Izrazi, naredbe, varijable i vrijednosti

Prije nego što pređemo na sâm programski jezik reći ćemo nešto o pojmu izraza, odnosno razlici između izraza i naredbe. To je važno da bismo razumijeli neke osnovne konstrukte u Pythonu ali i drugim programskim jezicima.

> Izraz je bilo koji konstrukt programskog jezika čije izvršavanje vraća neku vrijednost kao rezultat.

Na primjer, *2+3* je izraz. Ako je nešto izraz to znači da ono može biti dio nekog većeg izraza. Izraz *2+3* može biti dio izraza *5*(2+3)-1*.

```
>>> 2 + 3
5    # svaki izraz ima rezultat
```

Isto tako, $a > b$ je (logički) izraz. Na primjer,

```
>>> 5 > 8   # rezultat ovog izraza je True ili False
True
```

S druge strane, naredbe nemaju definiran rezultat. Primjerice, naredba pridruživanja $x = 2 + 3$ nema definiran rezultat, što znači da ona ne može biti dio nekog većeg izraza. To možemo vidjeti i u IDLE-u, gdje se nakon unosa gornje naredbe ispod komandne linije ne ispisuje ništa:

```
>>> x = 2 + 3
>>>
```

Naredba pridruživanja jednostavno pridružuje rezultat izraza s desne strane varijabli s lijeve strane. U ovom primjeru, ako želimo vidjeti što je pridruženo varijabli x možemo unijeti izraz x:

```
>>> x    # trivijalni izraz (ali i dalje izraz)
5
```

Iako se unos same varijable x ne čini kao izraz, i to je izraz, ali trivijalan, kao i da smo unijeli ovo:

```
>>> 5
5
```

I ovdje se možemo pitati može li sama varijabla x biti dio nekog većeg izraza, na što je odgovor sigurno potvrdan:

```
>>> x * 2    # x je dio veceg izraza
10
```

To znači da programski kôd koji se sastoji samo od varijable x sačinjava izraz.

Kako općenito možemo znati što je izraz, a što naredba? To ovisi isključivo o programskom jeziku. Na primjer, u nekim jezicima i pridruživanje je izraz, odnosno ima definiran rezultat i može biti dio nekog većeg izraza, dok u drugim jezicima (kao što je Python) to nije slučaj.

U prethodnom dijelu nakratko smo spomenuli varijable i pridruživanje. Općenito, sintaksu pridruživanja možemo prikazati kao

```
<ime> = <izraz>
```

> Kod naredbe pridruživanja prvo se odredi vrijednost izraza s desne strane, a zatim se ta vrijednost pridruži imenu s lijeve strane.

Oznaka <...> znači da se na tom mjestu treba nalaziti ono što je naznačeno unutar < i >. Na primjer, ako za <ime> postavimo x, a za <izraz> 2+3, onda je

```
x = 2 + 3
```

ispravno napisana naredba pridruživanja (razmaci između broja i operatora '+' nisu bitni, kao ni razmaci prije i poslije znaka '=').

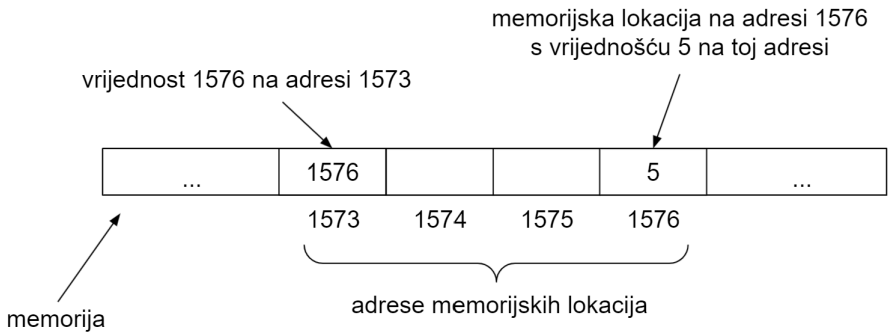
Iako naredba pridruživanja izgleda više kao izjava *x je jednak 2+3* ona se u stvari odvija u dva koraka. Prvo se dobije rezultat izraza s desne strane znaka "=", a zatim se taj rezultat pridruži imenu odnosno varijabli s lijeve strane. U Pythonu, svakoj varijabli mora biti pridružena neka vrijednost.

> Kada varijabli pridružimo neku vrijednost, ona joj ostaje pridružena sve dok toj varijabli ne pridružimo neku drugu vrijednost.

Varijablom pohranjujemo podatke. Ona služi kao ime za mjesto u memoriji na kojem se nalazi vrijednost koja joj je pridružena (više detalja o ovome nalazi se u dijelu 2.7).

Tehnički detalji

Python se smatra programskim jezikom više razine [17] što obično znači da programer ne mora misliti na detalje kao što su upravljanje memorijom, način pristupa vrijednostima kroz varijable, optimizaciju koda i sl. Međutim, poznavanje nekih osnovnih načela organizacije memorije i načina na koji Python smješta podatke u nju može pomoći u razjašnjavanju nekih problema koji se javljaju u programiranju s njim. U ovom dijelu ta su načela ukratko opisana.



Slika 2.1: Primjer sadržaja memorije nakon naredbe `x = 5` u kojem se varijabla x nalazi na adresi 1573 i sadrži vrijednost 1576 (adresa vrijednosti 5).

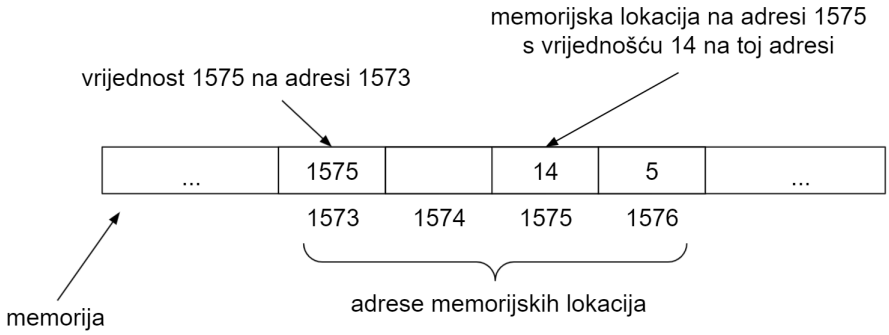
Svaka je vrijednost smještena negdje u memoriji računala. Memoriju računala možemo zamisliti kao niz "praznih" mjesta u koja možemo smjestiti podatke. Svako to mjesto, koje se zove *memorijska lokacija*, ima jedinstveni numerički identifikator koji se zove *adresa* memorijske lokacije. Memoriju računala, dakle, možemo zamisliti kako je prikazano na Slici 2.1. Prema tome, svaka se vrijednost nalazi na nekoj adresi u memoriji računala. Adrese su vrijednosti numeričkog tipa i obično su puno veći brojevi od onih na Slici 2.1 kao, na primjer, 140735839036176[1]. Kada varijabli pridružimo neku vrijednost, Pythonov interpreter mora tu varijablu smjestiti negdje u memoriju i od tog trenutka voditi evidenciju gdje je ona smještena. Pretpostavimo sada da je naredbom `x = 5` Pythonov intepreter smjestio varijablu x an adresu 1573, a vrijednost 5 na adresu 1576. Ovdje možemo uočiti dvije razlike u vrijednostima na adresama 1573 i 1576. Vrijednost na adresi 1573 je *adresa* na kojoj se nalazi vrijednost 5, dok je vrijednost 5 ono što smo pridružili varijabli x. Ako adresa 1573 predstavlja varijablu x onda vidimo da je vrijednost te varijable u stvari adresa broja 5, a ne sâm broj 5.

Kada varijabli pridružimo neku vrijednost u Pythonu ta varijabla sadrži adresu memorijske lokacije na kojoj se ta vrijednost nalazi.

Ovo je važno razumijeti jer je kasnije lakše objasniti neke druge strukture podataka s kojima ćemo često raditi kao što je lista, rječnik ili znakovni niz.

Promijeniti vrijednost pridruženu nekoj varijabli možemo samo tako da toj varijabli izravno pridružimo novu vrijednost naredbom pridruživanja. Ne postoji neki neizravni način (kao, primjerice, u jeziku C). Prema tome, pridruživanjem nove vrijednosti varijabli ta će varijabla sadržavati adresu te nove vrijednosti. Ovo je ilustrirano na Slici 2.2 gdje možemo zamisliti da smo nakon naredbe `x = 5` varijabli x pridružili vrijednost 14 naredbom `x = 14`. Vidimo da se varijabla x i dalje nalazi na istoj adresi i da se na adresi 1576 ništa nije promijenilo novim pridruživanjem. Razlika je na adresi 1573 -

[1] Adrese se često zapisuju u takozvanom *heksadecimalnom* obliku [18]. U tom obliku ova bi se adresa pisala kao *7fff9db16310*.



Slika 2.2: Primjer sadržaja memorije nakon naredbe `x = 14` u kojem se varijabla x nalazi na adresi 1573 i sadrži vrijednost 1575 (adresa broja 14).

umjesto prijašnje vrijednosti 1576 (adrese broja 5) sada je tu vrijednost 1575, to jest adresa broja 14.

2.2 Ispisivanje

Funkcija koju ćemo vrlo često koristiti je `print`. Ona služi za ispis rezultata svih izraza (uključujući i one trivijalne kao što su konstante ili varijable) koji su joj zadani, s lijeva na desno. Na primjer,

```
>>> print(1 + 1, 2 + 2, 3 + 3)
2 4 6
```

Ova se funkcija može upotrebljavati s različitim vrstama (tipovima) podataka:

```
>>> print('Kvadrat od 5 je', 5 * 5)
Kvadrat od 5 je 25
```

Prva vrijednost rezultat je tekstualnog izraza *'Kvadrat od 5 je'*, a druga rezultat izraza *5*5*. Funkcija `print` ima više mogućnosti koje ćemo uvoditi naknadno i po potrebi.

2.3 Primjer: program za rješavanje kvadratne jednadžbe

Sada imamo dovoljno informacija da napišemo prvi program - rješavanje kvadratne jednadžbe (Primjer 2.1). Ova se jednadžba rješava po formuli $x_{1,2} = \frac{-b \pm \sqrt{b^2-4ac}}{2a}$ gdje ćemo za $x_{1,2}$ koristiti dvije varijable, `x1` i `x2`.

U prvom redu vidimo naredbu `import math`. Ime *math* odnosi se na modul koji sadrži matematičke operacije kao što su *sqrt*, *sin*, *sqrt*, *pow* i slično. Python ima puno modula koji zajedno sadrže velik broj funkcija. Svrha modula je da se na logičan način organiziraju sve te operacije u skupine tako da su matematičke operacije na jednom mjestu, to jest u jednom modulu, operacije za rad s datotekama u drugom, operacije

Primjer 2.1: Program za izračunavanje kvadratne jednadžbe.

```
import math

a = float(input('Unesite a: '))
b = float(input('Unesite b: '))
c = float(input('Unesite c: '))

x1 = (-b + math.sqrt(b * b - 4 * a * c)) / (2 * a)
x2 = (-b - math.sqrt(b * b - 4 * a * c)) / (2 * a)

print('x1 =', x1, '   x2 =', x2)
```

za rad s računalnim mrežama u trećem, i tako dalje [16]. Izrazom `math.sqrt` koristimo operaciju *sqrt* za koju smo naveli da se nalazi u modulu *math*. Moduli općenito služe za organizaciju programa i o njima će više riječi biti u dijelu o modulima.

Sada možemo preći na sam program. Svaki se program izršava redoslijedom kojim su napisani njegovi izrazi, operacije i sve ostalo, počevši od prvog (u dijelu o kontroli toka izvršavanja programa vidjet ćemo kako se redoslijedom izvršavanja može manipulirati na određene načine). Prva tri reda služe za unos konstanti a, b i c, nakon čega slijedi izračunavanje dvaju rješenja, x_1 i x_2, to jest korijena ove jednadžbe. U zadnjem redu jednostavno ispisujemo rezultat. Na primjer,

```
Unesite a: 1
Unesite b: -1
Unesite c: -2
x1 = 2.0    x2 = -1.0
```

Iako je ovaj program vrlo jednostavan, već možemo uočiti moguća poboljšanja. Pogledajmo ova dva izraza:

```
(-b + math.sqrt(b * b - 4 * a * c)) / (2 * a)
(-b - math.sqrt(b * b - 4 * a * c)) / (2 * a)
```

Jedina razlika između njih je u jednoj operaciji - kod prvoga broju `-b` dodajemo korijen, a kod drugoga ga oduzimamo; sve ostalo je identično u oba izraza. Jedna od osnovnih pravila u programiranju je izbjegavanje dupliciranja koda (gdje god je to moguće i opravdano), što je u našem programu lako. Kao prvo, izraz za korijen možemo izdvojiti i njegov rezultat pridružiti nekoj varijabli. Kao drugo, izraz za nazivnik, `2 * a`, također možemo izdvojiti i pridružiti nekoj varijabli. Prema tome, poboljšana inačica našeg programa može izgledati kao u Primjeru 2.2.

Svrha ovog poboljšanja nije u tome da ono što se ponavlja na više mjesta u programu jednostavno stavimo na jedno mjesto, već da program učinimo razumljivijim. Ako želimo biti sigurni da ovaj program točno računa rješenja kvadratne jednadžbe onda u ovoj poboljšanoj inačici imamo manje izraza koje moramo provjeriti - jedan izraz za korijen,

Primjer 2.2: Poboljšani program za izračunavanje kvadratne jednadžbe.

```
import math

a = float(input('Unesite a: '))
b = float(input('Unesite b: '))
c = float(input('Unesite c: '))

korijen = math.sqrt(b * b - 4 * a * c)
nazivnik = 2 * a

x1 = (-b + korijen) / nazivnik
x2 = (-b - korijen) / nazivnik

print('x1 =', x1, '   x2 =', x2)
```

jedan za nazivnik te dva za x_1 i x_2 - sveukupno četiri izraza, dok ih je u prvoj inačici bilo šest. Naravno, kod ovako kratkog programa ovo poboljšanje nije od velikog značaja, ali programi koji se danas koriste u industriji imaju desetke ili stotine tisuća, a neki i milijune redova. Kod takvih programa ovakva i mnoga druga poboljšanja koja olakšavaju razumijevanje, a time i razne modifikacije i nadogradnje, često mogu značiti razliku između uspjeha i propasti softverskog sustava i projekta čiji je on bio cilj.

Kod programa u Primjerima 2.1 i 2.2 možemo uočiti razmake između redova, kao što je onaj između *import* i pridruživanja varijabli *a*. Ti razmaci ne utječu na izvršavanje programa nego nam samo olakšavaju njegovo čitanje jer njima vizualno grupiramo dva ili više izraza koji imaju (po našem mišljenju) nekakvu sličnu funkciju, kao što je pridruživanje varijablama *x1* i *x2*. Isto tako, razmaci između zareza ili operatora, kao što je `print(a, b)` ili `x + y`, imaju istu svrhu i ne utječu na izvršavanje programa.

2.4 Osnove funkcija

Sintaksni oblik kao što je `print(...)` označava *poziv funkcije*. *Funkcija* je niz naredbi kojima se izvodi neki računalni postupak. Neka postoji funkcija F kojom je definiran sljedeći niz naredbi:

```
funkcija F:
    naredba 1
    naredba 2
    naredba 3
```

Nadalje, neka postoji sljedeći niz naredbi koji sačinjava neki program P:

```
naredba A
F()
naredba B
```

Pokretanjem programa P izvršit će se sljedeće naredbe u navedenom poretku:

```
naredba A
naredba 1
naredba 2
naredba 3
naredba B
```

Dakle, vidimo da pozivom funkcije F, `F()`, izvršavanje programa prelazi na niz naredbi koji je definiran u toj funkciji. Nadalje, nakon izvršenja naredbe `naredba 3` izvršavanje se nastavlja od prve naredbe iza poziva `F()`. Niz naredbi kao što je onaj u funkciji F zove se *tijelo funkcije*, a dio gdje je zadano ime funkcije (i parametri, što ćemo uskoro vidjeti) zove se *zaglavlje funkcije*. Poziv funkcije za početak možemo zamisliti kao da tijelo te funkcije umetnemo na mjesto njenog poziva.[2] U gornjem primjeru, ako `F()` zamijenimo tijelom funkcije F dobit ćemo isti niz naredbi kao i u prethodnom.

Što ako jedna funkcija u svom nizu naredbi sadrži poziv druge funkcije? Neka postoji funkcija G koja sadrži sljedeći niz naredbi:

```
funkcija G:
    naredba X
    F()
    naredba Y
```

Nadalje, neka postoji program R koji se sastoji od sljedeći h naredbi:

```
naredba S
G()
naredba T
```

Pokretanjem programa R izvršio bi se sljedeći niz naredbi:

```
naredba S
naredba X
naredba 1
naredba 2
naredba 3
naredba Y
naredba T
```

Vidimo da je načelo izvršavanja ovakvog programa isto kao i prethodno opisano. Nakon naredbe S počinje se izvršavati niz naredbi funkcije G koji počinje naredbom X. Nakon te naredbe počinje se izvršavati funkcija F sa svoje tri naredbe, 1, 2 i 3. Dakle, do tog trenutka imamo ispis

[2] Za poznavatelje jezika C++ ovo ne implicira bilo kakvu sličnost s takozvanim *inline* funkcijama nego se radi samo o analogiji za lakše razumijevanje poziva funkcija.

```
naredba S
naredba X
naredba 1
naredba 2
naredba 3
```

Nakon povratka iz funkcije F vraćamo se na prvu naredbu iza njenog poziva, a to je naredba Y funkcije G. Nakon toga se vraćamo na prvu naredbu iza poziva funkcije G, a to je naredba T programa R. Općenito, tijek izvršavanja programa R možemo opisati kao

$$R \rightarrow G \rightarrow F \rightarrow G \rightarrow R$$

Vidimo da se povratak iz funkcija odvija u obrnutom redoslijedu od poziva: iz R prvo pozivamo G pa onda iz G pozivamo F, ali se vraćamo iz F u G pa onda iz G u R. Ovo je sasvim logično: da bi program R izvršio naredbu T potrebno je prije toga izvršiti niz naredbi funkcije G, a da bi funkcija G izvršila naredbu Y potrebno je prije toga izvršiti niz naredbi funkcije F.

Općenito, povratak iz funkcije odvija se nakon što je izvršena njena posljednja naredba ili ako je povratak naznačen eksplicitno, naredbom *return* kao što ćemo kasnije vidjeti. Nakon povratka iz funkcije izvršavanje se nastavlja prvom naredbom koja slijedi poziv te funkcije.

2.4.1 Definiranje novih funkcija

Funkcije u Pythonu definiraju se naredbom *def*. U sljedećem primjeru definirana je funkcija *kvadrat* koja vraća kvadrat broja:

```
1 def kvadrat(n):
2     return n * n
```

Sada ovu funkciju možemo pozvati na način sličan onome kod funkcije *print*:

```
1 >>> kvadrat(4)
2 16
```

U zaglavlju ove funkcije nalazi se jedna varijabla, *n*. Takve varijable nazivat ćemo *parametrima*, a vrijednosti zadane kod poziva funkcije koje se pridružuju parametrima, kao što je 4 u gornjem primjeru, nazivat ćemo *argumentima*.

Ovdje treba primjetiti da je naredba `return n * n` uvučena u odnosu na naredbu `def kvadrat(n)`. To je dio sintakse Pythona i na taj se način označava jedan niz naredbi koji može sadržavati jednu ili više naredbi i koji je dio neke druge naredbe. U slučaju funkcije *kvadrat* niz naredbi (koji sadrži samo jednu naredbu) `return n * n` dio je naredbe `def kvadrat(n)`, što znači da je to niz naredbi koji će biti izvršen pozivom ove funkcije. Drugim riječima, taj niz naredbi pripada tijelu funkcije *kvadrat*.

S obzirom da funkcija *kvadrat* vraća kvadrat broja taj povratni rezultat možemo upotrijebiti za neki kompleksniji aritmetički izraz:

```
>>> 2 * kvadrat(4) - 1
31
```

Svaka funkcija u Pythonu vraća neku vrijednost kao rezultat.

Svaka funkcija u Pythonu mora vratiti neku vrijednost, bilo eksplicitno, naredbom *return* ili implicitno. Sljedeća funkcija, na primjer, nema naredbu *return*:

```
def zbroj(a, b):
    print('Zbroj brojeva', a, ' i', b, ' je', a + b)
```

Funkcije kod kojih posljednja izvršena naredba nije *return* vraćaju *None* kao rezultat. To možemo vidjeti ako rezultat funkcije zbroj pridružimo nekoj varijabli:

```
>>> x = zbroj(2, 3)
Zbroj brojeva 2 i 3 je 5

>>> x  # x je None pa se ispod ne ispisuje nista
>>> x is None
True
```

None je vrijednost koja označava odsustvo neke konkretne vrijednosti. Na primjer, ako varijablu x moramo inicijalizirati na neku početnu vrijednost, ali u tom trenutku nemamo neku kontretnu vrijednost koju joj možemo pridružiti, možemo joj pridružiti vrijednost *None*:

```
x = None
```

Po istoj logici funkcija koja nema konkretnu povratnu vrijednost, kao što je *print*, obično vraća *None*.

U sljedećem primjeru prikazana je funkcija *broj_znamenki* koja vraća broj znamenki broja zadanog kao parametar funkcije:

```
def broj_znamenki(n):
    tekstualno = str(n)
    duljina = len(tekstualno)
    return duljina


>>> broj_znamenki(1009)
4
```

Ovu funkciju mogli smo napisati i na ovaj, kraći način:

```
def broj_znamenki(n):
    return len(str(n))


>>> broj_znamenki(217)
3
```

Primjer 2.3: Funkcija *broj_znamenki*.

```
n = None

def broj_znamenki():
    return len(str(n))


>>> n = 217
>>> broj_znamenki()
3
>>> n = 5178
>>> broj_znamenki()
4
```

S obzirom da su *str* i *len* funkcije koje vraćaju neku vrijednost kao rezultat, mogu se kombinirati u složeniji izraz kao što je `len(str(n))`. Kao i kod aritmetičkih izraza ovdje se prvo odredi rezultat unutrašnjeg izraza, što je ovdje `str(n)`. Varijabla *n* ulazni je argument za funkciju *str* koja vraća zadani broj kao tekstualnu vrijednost (niz znakova). Rezultat tog izraza ulazni je argument za funkciju *len* koja vraća broj znakova zadane tekstualne vrijednosti. Ovo možemo prikazati koracima:

$$\texttt{len(str(n))} \rightarrow \texttt{len(str(217))} \rightarrow \texttt{len("217")} \rightarrow \texttt{3}$$

U trećem koraku vidimo da je funkcija *str* vratila znakovni niz ”217”, a ne broj 217 jer funkcija *len* radi s tekstualnim (i nekim drugim) vrijednostima.

2.4.2 Doseg varijabli

U funkciji *broj_znamenki* parametar *n* je takozvana *lokalna* varijabla [19] zbog toga što je ona dostupna samo naredbama i izrazima unutar te funkcije. Kaže se da je *doseg* varijable (parametra) *n* funkcija *kvadrat*. Ovu funkciju mogli smo napisati i kako je pokazano u Primjeru 2.3.

Varijabla *n* sada više nije lokalna nego *globalna*, to jest dostupna je iz bilo kojeg dijela koda. Ovakav način, međutim, nije prihvatljiv iz nekoliko razloga od kojih ćemo ovdje navesti sljedeće:

- U zaglavlju funkcije *broj_znamenki* nije vidljivo koji su joj ulazni podaci potrebni - to moramo zaključiti analizom tijela te funkcije.
- Iz gornjeg koda nije odmah jasno koja je uloga varijable *n* - je li ona tu samo zbog funkcije *broj_znamenki* ili se ta varijabla upotrebljava i na drugim mjestima u programu. Ovo nije značajan problem kod ovako malih programa, ali ako zamislimo program od nekoliko desetaka funkcija jasno je zašto upotreba globalnih varijabli nije preporučljiva.

Općenito, varijable je bolje što više lokalizirati na segment koda unutar kojeg se upotrebljavaju jer to olakšava razumijevanje programa, a time i umanjuje mogućnost grešaka.

2.5 Osnovni tipovi podataka: *str* i *list*

Oni koji su koristili programske jezike kao što su Java, C++ ili C# mogli su primijetiti da varijable iz Primjera 2.3, 2.1 i 2.2 nisu imale oznaku tipa podatka za koji su bile predviđene, kao *int*, *float* i sl. Python pripada skupini takozvanih *dinamičkih* programskih jezika [20] kojima je to jedna od osobina. U Pythonu nekoj varijabli možemo pridružiti bilo koju vrijednost, bez obzira na to kojeg je tipa ta vrijednost:

```
>>> x = 5  # 'x' je cijeli broj 5
>>> x = 'abc'  # 'x' je sada znakovni niz 'abc'
```

Iako to nije preporučljivo, u gornjem primjeru vidimo da istoj varijabli možemo pridružiti vrijednosti različitog tipa. Općenito, varijabla u Pythonu samo je ime za vrijednost i ne podrazumijeva nikakve specifičnosti vezane za tu vrijednost. Iako se varijablama u Pythonu ne mora označiti tip vrijednosti koja im može biti pridružena to ne znači da tip vrijednosti ne igra ključnu ulogu u ovom jeziku. Svaka operacija u Pythonu, kao što je '+', '-', '*' i druge, definirana je samo za određene tipove podataka isto kao i u matematici. Na primjer, oduzimanje nije definirano za znakovne nizove:

```
>>> 'abc' - 'def'
TypeError: unsupported operand type(s) for -: 'str' and 'str'
```

Isto tako ne možemo zbrojiti znakovni niz i broj:

```
>>> 'abc' + 5
TypeError: can only concatenate str (not "int") to str
```

Kao što vidimo, u oba slučaja poruka počinje s *TypeError*, što implicira to da za dotične tipove vrijednosti određena operacija nije definirana. Kao i u drugim programskim jezicima tip podatka jedan je od temeljnih pojmova u Pythonu i u ovom dijelu ćemo opisati sve tipove podataka koji su ugrađeni u ovaj jezik. Python ima nekoliko ugrađenih tipova podataka od kojih su nam ovdje najvažniji sljedeći:

- logički tip: *bool* (s vrijednostima *True* i *False*)
- cijelobrojni tip: *int*
- realni tip: *float*
- kompleksni tip: *complex*
- znakovni niz: *str*
- dva uređena niza vrijednosti: *list* i *tuple*
- dvije neuređene kolekcije vrijednosti: *dict* i *set*

S logičkim, cijelobrojnim i realnim vrijednostima radi se na sličan način kao i u drugim programskim jezicima pa ovdje nećemo ulaziti u detalje. Sve specifičnosti ovih triju tipova podataka na koje naiđemo bit će istaknute po potrebi. Ovdje ćemo samo istaknuti jednu razliku u pisanju logičkih operatora u odnosu na jezike C/C++, Java, C# i neke druge: U Pythonu logički operatori *i*, *ili* i negacija pišu se *and*, *or* i *not* (umjesto "&&", "||" i "!").

Kolekcija je vrsta vrijednosti koja se sastoji od jedne ili više drugih vrijednosti koje mogu biti istog i/ili različitog tipa.

U ovom dijelu proučit ćemo dva tipa podatka: znakovni niz (*str*) i lista ili niz (*list*). Oba ova tipa podatka su kolekcije: znakovni niz sastoji se od jednog ili više znakova, a lista od jedne ili više vrijednosti koje se nazivaju *elementima* liste. [16, 21, 22, 23, 24, 25]

2.5.1 Znakovni niz

Znakovni niz tekstualni je tip podatka koji se sastoji od uređenog niza znakova.

```
>>> s = 'abc'  # ili s = "abc"
```

Neke česte operacije sa znakovnim nizovima su sljedeće: indeksiranje, dohvaćanje duljine (broj znakova), spajanje dvaju znakovnih nizova, segmentiranje, dobivanje položaja zadanog znaka, provjera pripadnosti nekog znaka znakovnom nizu i zamjena dijela znakovnog niza drugačijim sadržajem.

Znakovima znakovnog niza možemo pristupiti preko indeksa, počevši od 0.

S obzirom da je znakovni niz uređeni niz znakova, njegovim znakovima možemo pristupiti preko indeksa, počevši od 0.

```
>>> s = 'abc'
>>> s[1]
'b'
>>> s[5]
...greska...
```

Indeks može biti i negativan, u kojem slučaju označava poziciju s desna. Na primjer, da bismo dohvatili posljednji znak možemo zadati indeks -1:

```
>>> s[-1]
'c'
```

Ovdje je očito da indeks 0 ne može istovremeno označavati i prvi i posljednji element.

Znakovni nizovi su nepromijenjivi.

Važna osobina znakovnih nizova je ta da se oni ne mogu modificirati:

```
>>> s[1] = 'x'
...
TypeError: 'str' object does not support item assignment
```

Ako neki znakovni niz želimo promijeniti, umjesto njegove modifikacije možemo napraviti novi znakovni niz koji sadrži odgovarajuće promjene.

Funkcija *len* vraća broj znakova u znakovnom nizu.

Ako želimo dobiti podatak o broju znakova u zadanom znakovnom nizu možemo koristiti funkciju *len*:

```
>>> len(s)
3
```

Ovdje treba paziti na to da funkcija *len* ne vraća posljednji indeks znakovnog niza – ona vraća vrijednost za 1 veću od tog indeksa! U sljedećem primjeru pokušavamo doći do posljednjeg znaka na pogrešan način:

```
>>> s = 'abc'
>>> s[len(s)]
...greska...
```

Ispravno bi, dakako, bilo

```
>>> s[len(s) - 1]
```

Međutim, nema potrebe koristiti ovakav složeniji izraz kad znakovni niz možemo jednostavno indeksirati i negativnim brojevima kako je prethodno pokazano.

Operator ”+” vraća novi znakovni niz koji je spoj druga dva znakovna niza.

Česta operacija sa znakovnim nizovima spajanje je dvaju znakovnih nizova u jedan novi. Za to se u Pythonu koristi operator ”+”:

```
>>> s + s
'abcabc'
>>> s
'abc'
```

Zapravo, znakovni nizovi se operatorom ”+” ne spajaju nego taj operator stvara novi znakovni niz koji je spoj dvaju zadanih znakovnih nizova.

Segmentiranjem možemo izdvojiti dio znakovnog niza.

Često nam treba samo dio znakovnog niza. Primjerice, ako takav niz sadrži broj mobitela i treba nam samo pozivni broj možemo ga izdvojiti segmentiranjem:

```
>>> broj_tel = '099-1234-567'
>>> broj_tel[0:3]
```

```
'099'
>>> broj_tel[:3]
'099'
>>> broj_tel[4:8]
'1234'
>>> broj_tel[-3:]  # treci s desna pa do kraja
'567'
```

U uglatim zagradama postavljen je interval kojim je obuhvaćen dio znakovnog niza. Za interval *[m:n]*, gdje je $m < n$, obuhvaćeni su elementi na indeksima *m*, *m+1*, *m+2*, ..., *n-1* (znak na indeksu *n* ne ulazi u ovaj segment). Nadalje, u definiciju intervala može biti uključen i korak, to jest broj elemenata koje periodički preskačemo. Ako nije zadana, podrazumijevana vrijednost je 1, što znači da je uključen svaki element intervala. U sljedećem primjeru izdvajamo segment znakovnog niza, ali tako da u zadanom intervalu uzmemo svaki drugi znak:

```
>>> s = 'abcdefghijk'
>>> s[2:10:2]
'cegi'
```

Ovo možemo zamisliti kao da počnemo od znaka na indeksu 2, što je 'c', nakon čega odredimo koji je idući znak tako da tekućem indeksu dodamo korak. U gornjem primjeru indeksi koji su obuhvaćeni bi tada bili 2, 2+2, 4+2, 6+2, ... Ako želimo uzeti u obzir cijeli znakovni niz pisali bismo

```
>>> s[::2]
'acegik'
```

Jedna korisna posljedica prethodno opisanog postupka izračunavanja indeksa je ta da iz jednog znakovnog niza na vrlo jednostavan način možemo dobiti njegovu preokrenutu inačicu tako da korak postavimo na -1:

```
>>> s[::-1]
'kjihgfedcba'
```

Ako primijenimo gornji postupak dobit ćemo indekse 0-1=-1 ('k'), -1-1=-2 ('j'), -2-1=-3 ('i'), ... Kada znakove na ovim indeksima dodajemo u novi znakovni niz tim redoslijedom, s lijeva na desno, dobijemo preokrenuti početni znakovni niz.

Metodom *index* možemo dobiti prvi indeks na kojem se nalazi traženi znak.

Ako želimo vidjeti gdje se u znakovnom nizu nalazi neki znak ili provjeriti postoji li uopće taj znak možemo koristiti metodu *index*. Ona vraća prvi indeks na kojem se nalazi traženi znak (jer isti znak se može nalaziti na više mjesta u znakovnom nizu).

```
>>> broj_tel.index('-')
3
```

Ako zadani znak ne postoji u znakovnom nizu ova funkcija signalizira grešku, tj. *iznimku* (o njima ćemo govoriti kasnije).

```
>>> broj_tel.index('a')
...greska...
```

Ovdje treba obratiti pažnju na izraz *broj_tel.index(...)*, to jest na oblik *vrijednost.funkcija*. U ovom primjeru funkcija *index* isključivo je dio tipa *str*. Takve se funkcije nazivaju *metodama* i one će biti puno detaljnije opisane u poglavlju 7. U nastavku ovog dijela vidjet ćemo još nekoliko metoda znakovnih nizova i drugih tipova podataka.

Metodom *find* također možemo dohvatiti prvi indeks na kojem se nalazi traženi znak.

Metoda slična metodi *index* je *find*. Ona radi gotovo isto kao i *index*, ali u slučaju da traženi znak ne postoji u znakovnom nizu vraća -1.

```
>>> s = 'abc'
>>> s.find('x')
-1
```

Metodom *replace* možemo zamijeniti jedan dio znakovnog niza drugačijim sadržajem.

Još jedna korisna operacija sa znakovnim nizovima zamjena je dijela niza drugim sadržajem što možemo napraviti metodom *replace*. U sljedećem primjeru zamijenit ćemo '-' sa '/'.

```
>>> broj_tel.replace('-', '/')
'099/1234/567'
>>> broj_tel
'099-1234-567'
```

Ovdje treba primijetiti da metoda *replace* ne modificira početni znakovni niz nego vraća novi koji sadrži navedene promjene.

Operatorom *in* provjeravamo postoji li zadani znak u znakovnom nizu.

Važna i česta operacija nad znakovnim nizovima provjera je pripadnosti znaka znakovnom nizu. Na primjer, ako želimo provjeriti postoji li znak '1' u gornjem broju telefona možemo koristiti Pythonov operator *in*.

```
>>> '1' in broj_tel
True
>>> '8' in broj_tel
False
```

Ovo će nam biti korisno kasnije gdje će tijek izvršavanja programa ovisiti o ovakvim provjerama.

Funkcija *str* vraća tekstualni prikaz vrijednosti.

Još jedna korisna funkcija je *str* koja vraća tekstualni prikaz zadane vrijednosti. Ako, primjerice, želimo neki broj pretvoriti u znakovni niz da bismo dobili pristup njegovim znamenkama, to možemo postići ovom funkcijom:

```
>>> str(194)
'194'
```

2.5.2 Lista

Jedna od najčešće korištenih struktura podataka u Pythonu je lista. Lista je niz kod kojeg se elementima pristupa pomoću indeksa.

```
>>> gradovi = ['Zagreb', 'Rijeka', 'Dubrovnik']
>>> gradovi[2]
'Dubrovnik'
```

Liste mogu sadržavati elemente različitog tipa.

Liste u Pythonu su heterogene, što znači da mogu sadržavati elemente različitog tipa.

```
>>> p = ['Jedan', 2, 3.0, [4]]
```

Vidimo da liste mogu sadržavati i druge liste (podliste) kao elemente:

```
>>> p[3]
[4]
```

Kao i kod znakovnih nizova, elementima liste pristupa se preko indeksa (počevši od 0).

Kako doći do broja 4? Vrijednost izraza p[3] je lista [4]; prema tome, tu listu možemo indeksirati kao i svaku drugu listu.

```
>>> p[3][0]
4
```

S obzirom da lista [4] ima samo jedan element i to na indeksu 0, preko tog indeksa pristupamo tom elementu. Ovo možemo napisati u dva koraka:

```
>>> a = p[3]
>>> a[0]
4
```

Kapacitet liste u Pythonu postavlja se automatski.

Kako lista „predviđa“ koliko elemenata će nam trebati? Liste u Pythonu su dinamičke, što znači da se kapacitet automatski postavlja, po potrebi. Drugim riječima, programer o tome ne treba voditi računa.

Liste su promjenjive.

Liste su promjenjive u smislu da element na nekom indeksu liste možemo zamijeniti drugim.

```
>>> p[0] = 1
>>> p
[1, 2, 3.0, [4]]
```

Metodama *append* i *insert* dodajemo nove elemente u listu.

Novi element možemo dodati u listu na dva načina:

- dodavanjem na kraj liste ili
- umetanjem na mjesto specificirano indeksom.

Metode *append* i *insert* možemo koristiti na sljedeći način:

```
>>> p.append('novi')  # dodavanje na kraj
>>> p
[1, 2, 3.0, [4], 'novi']
>>> p.insert(2, 'X')  # umetanje elementa 'X' na indeks 2
>>> p
[1, 2, 'X', 3.0, [4], 'novi']
```

Umetanje elemenata negdje prije posljednjeg zahtijeva pomicanje svih elemenata nakon umetnutog za jedno mjesto, što kod većih lista može utjecati na performanse programa.

Operator ”+” vraća novu listu koja je spoj drugih dviju lista.

Novu listu možemo dobiti tako da operatorom ”+” spojimo dvije liste kao u sljedećm primjeru:

```
>>> a = [1, 2]
>>> b = [3, 4]
>>> a + b
[1, 2, 3, 4]
>>> a
[1, 2]
>>> b
[3, 4]
```

Kao i kod znakovnih nizova, izvorne liste ostaju iste.

Umjesto metode *append* možemo koristiti operator "+=".

Kao i kod znakovnih nizova, spajanjem lista a i b one nisu modificirane nego je konstruirana nova lista. Nadalje, umjesto metode *append* možemo koristiti operator "+=":

```
>>> a += [3, 4]  # dodaj elemente 3 i 4 na kraj liste 'a'
>>> a
[1, 2, 3, 4]
```

Metodom *extend* možemo u listu dodati više elemenata odjednom.

Prednost operatora '+=' je u tome što možemo u listu s lijeve strane dodati više od jednog elementa odjednom. Umjesto ovog operatora možemo koristiti metodu *extend*:

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

Metoda *clear* uklanja sve elemente liste.

Iz neke liste možemo ukloniti sve elemente koristeći metodu *clear*. Ova metoda ne stvara novu listu nego modificira postojeću:

```
>>> p.clear()
>>> p
[]
```

Metoda *remove* uklanja prvi element koji je jednak onom zadanom.

Elemente liste možemo ukloniti i na selektivniji način tako da specificiramo koji element želimo ukloniti. Za to možemo koristiti metodu *remove*. Ako postoji više takvih elemenata ova metoda uklanja prvi.

```
>>> p = ['a', 'b', 'c', 'd', 'b']
>>> p.remove('b')  # uklanja prvi 'b'
>>> p
['a', 'c', 'd', 'b']
```

Funkcijom *del* elemente možemo ukloniti element na zadanom indeksu.

Još jedan način brisanja elemenata iz liste je funkcijom *del* koja briše element koji se nalazi na zadanom indeksu:

```
>>> del p[2]
>>> p
['a', 'c', 'b']
```

Ova funkcija ne pripada samo tipu liste nego je univerzalna.

Funkcija *len* vraća broj elemenata u listi.

Kao i kod znakovnih nizova, funkcija *len* vraća broj elemenata u listi:

```
>>> len(p)
3
>>> len([1, [2, [3, 4]]])
2
```

Ovaj zadnji primjer pokazuje da se broje samo oni elementi liste koji se nalaze na najvišoj razini, što su u ovom primjeru samo dva elementa: 1 i [2, [3, 4]].

Kao i znakovne nizove, i liste možemo segmentirati.

```
>>> p[1:]  # sve od elementa na indeksu 1 do kraja
['c', 'b']
```

Operatorom *in* provjeravamo pripadnost elementa listi.

Isto tako, operator *in* možemo koristiti za provjeru pripadnosti elementa zadanoj listi.

```
>>> 'c' in p
True
>>> 'abc' in p
False
```

Metoda *index* vraća indeks na kojem se u listi nalazi zadani element.

Kao i za znakovne nizove, i za liste postoji metoda *index* koja vraća indeks zadanog elementa ako on postoji, u suprotnom signalizira grešku.

```
>>> p.index('c')
1
```

Funkcija *list* konvertira niz elemenata u listu.

Kao što za znakovne nizove postoji funkcija *str* koja zadanu vrijednost pretvara u znakovni niz, tako i za liste postoji funkcija *list* koja zadanu vrijednost pretvara u listu.

```
>>> list('abc')
['a', 'b', 'c']
```

Ova funkcija, naravno, zahtijeva da je zadana vrijednost nekakva kolekcija iz koje se može izraditi lista.

```
>>> list(194)
...greska...
```

Metoda *join* korisna je za spajanje znakovnih nizova neke liste.

Još jedna korisna metoda je *join*, koja je zapravo dio tipa *str*. Ovom metodom možemo iz liste znakovnih nizova dobiti jedan znakovni niz koji je rezultat spajanja svih znakovnih nizova u listi.

```
>>> p = ['a', 'b', 'c']
>>> ''.join(p)
'abc'
```

Ova metoda očekuje da su svi elementi liste znakovni nizovi. Vidimo da je metoda *join* dio tipa *str*. Dakle, izraz "" daje vrijednost tipa *str* za koju se poziva metoda *join*. Ako umjesto praznog znakovnog niza postavimo neki drugi tekstualni sadržaj u konačnom rezultatu između znakovnih nizova iz liste nalazit će se taj sadržaj:

```
>>> ';'.join(p)
'a;b;c'
```

2.5.2.1 N-torke

N-torke su nepromijenjive liste.

Jedna inačica liste je takozvana *n-torka* (engl. *tuple*). N-torka je u načelu isto što i lista, ali je nepromijenjiva. Sintaksno, n-torke se pišu u okruglim zagradama.

```
>>> k = ('a', 'b', 3)
>>> k[2] = 'c'
TypeError: 'tuple' object does not support item assignment
```

Ova poruka kaže da tip vrijednosti *tuple* ne podržava pridruživanje novih vrijednosti individualnim pozicijama n-torke. Općenito, n-torka može biti korisna kada želimo spriječiti njenu modifikaciju, kako je prethodno pokazano s listama. To, međutim, ne znači da od jedne n-torke ne možemo napraviti drugu:

```
>>> k = ('a', 'b', 'c', 'd', 'e')
>>> k[:2] + k[3:]
('a', 'b', 'd', 'e')
```

Ovdje smo dobili novu n-torku bez elementa "c". N-torku možemo konvertirati u listu funkcijom *list*:

Primjer 2.4: Naredba *if*.

```
if n < 0:
    x = -n
else:
    x = n
```

```
>>> k
('a', 'b', 'c', 'd', 'e')
>>> list(k)
['a', 'b', 'c', 'd', 'e']
```

Ovdje, naravno, nismo promijenili polaznu n-torku nego smo od nje napravili novu listu.

2.6 Kontrola toka programa

U ovom dijelu proći ćemo sljedeće naredbe:

- *if*
- *match*
- *for*
- *while*

2.6.1 Naredbe *if* i *match*

Naredbe *if* i *match* su takozvane *uvjetne naredbe* ili *naredbe grananja* jer one usmjeravaju izvršavanje koda na osnovu zadanog izraza koji predstavlja uvjet. Osnovni oblik naredbe *if* je sljedeće:

if *izraz*:
    naredbe
else:
    naredbe

Semantika ove naredbe ista je kao i kod drugih programskih jezika: ako logički izraz vraća *True* izvršit će se prvi niz naredbi, a ako vraća *False* drugi. U Primjeru 2.4 varijabli x pridružuje se apsolutna vrijednost broja n.

Ovdje treba voditi računa o dvije stvari:

- Iza logičkog izraza *if*-dijela i iza *else* mora se nalaziti dvotočka.
- Niz naredbi koji sačinjavaju jedan blok mora biti uvučen (Python nema blok {...} kao, na primjer, Java i C#).

Primjer 2.5: Naredba *if* s više naredbi unutar *if* i *else* klauzula.

```
if n < 0:
    x = -n
    print(x)
else:
    x = n
    print(x)

print('nastavak')
```

U Primjeru 2.5 prikazan je još jedan primjer naredbe *if* gdje se ispisuje vrijednost varijable *x*. Ako je, na primjer, *n* jednak -5 gornji program bi ispisao 5. Ponovo treba obratiti pažnju na to da će se niz naredbi

```
x = -n
print(x)
```

izvršiti ako je *n* negativan jer taj blok naredbi pripada *if*-dijelu (jer je uvučen pod *if*), dok će se niz naredbi

```
x = n
print(x)
```

izvršiti ako je *n* pozitivan jer taj blok naredbi pripada else-dijelu (jer je uvučen pod *else*). Nakon naredbe *if* izvršit će se naredba *print('nastavak')* jer ta naredba ne pripada niti *if*- niti *else*-dijelu (ona nije uvučena).

Još jedna uvjetna naredba je *match*. Ona je praktična kada imamo više uvjeta gdje za svakog od njih imamo dio koda koji će se izvršiti ako je rezultat uvjeta *True*. Tipičan oblik ove naredbe je sljedeći:

match *varijabla*:
    case *varijabla* [**if** *izraz*]:
        naredbe
    case *varijabla* [**if** *izraz*]:
        naredbe
    ...
    case _:
        naredbe

Ova naredba prikazana je u Primjeru 2.6. U tom primjeru provjeravamo je li vrijednost varijable *n* 1, 2 ili 3. Ako je jedna od tih vrijednosti izvršit će se blok odgovarajuće *case* klauzule. Ako vrijednost nije niti jedno od toga izvršit će se blok klauzule *case* _. Na primjer,

```
>>> f(2)
```

Primjer 2.6: Naredba *match*.

```
def f(n):
    match n:
        case 1:
            print('jedan')

        case 2:
            print('dva')

        case 3:
            print('tri')

        case _:
            print('nista')
```

Primjer 2.7: Naredba *match* s dodatnim uvjetom u klauzuli *case*.

```
def g(n):
    match n:
        case n if 1 <= n <= 5:
            print('izmedju jedan i pet')

        case _:
            print('n nije 1, 2 ili 3!')

>>> g(3)
'izmedju jedan i pet'

>>> g(9)
'nista'
```

```
'dva'

>>> f(12)
'nista'
```

Klauzula *case* može primiti i dodatni uvjet kao što je pokazano na Slici 2.7. Kao i kod naredbe *if*, blok klauzule *case* može se sastojati od više redova.

2.6.2 Naredbe *for* i *while*

Naredbe *for* i *while* naredbe su iteracije ili ponavljanja. Naredba *for* u Pythonu ima dva osnovna oblika:

- iteracija zadani broj puta
- iteracija kroz niz elemenata neke kolekcije.

Prvi oblik naredbe *for* je

for *varijabla* **in range(a, b):**
    naredbe

Jedan primjer prvog oblika može izgledati ovako:

```
for i in range(0, n):  # n nije ukljucen - ide se do n-1
    print(i)
```

Ako je *n* jednak 3 gornja petlja će ispisati brojeve 0, 1 i 2, ali ne i 3. Drugim riječima, u gornjoj petlji *i* se kreće u rasponu od 0 do *n-1*. Dakle, iz gornjeg primjera treba uočiti dvije važne stvari:

- Iteracija kroz `range(0, n)` uključuje 0, 1, 2, ..., *n-1*, ali ne i *n*.
- Varijabla *i* ne mora se uvećavati eksplicitno – ona će se automatski uvećati za 1 nakon svake iteracije.

Drugi oblik naredbe *for* je

for *varijabla* **in** *kolekcija***:**
    naredbe

Jedan primjer drugog oblika može izgledati ovako:

```
for e in niz:
    print(e)
```

Ako, primjerice, niz sadrži listu [100, 'test', 3.14] onda će gornja petlja ispisati vrijednosti 100, 'test' i 3.14. Ovaj oblik petlje možemo riječima opisati kao *za svaki element e niza **niz** ispiši e*. Iz gornjeg primjera treba uočiti sljedeće:

- Ne postoji varijabla koja se uvećava u svakoj iteraciji nego se varijabli *e* u svakoj iteraciji implicitno pridružuje sljedeći element niza *niz*. Dakle, u gornjem primjeru *e* će inicijalno sadržavati 100, u sljedećoj iteraciji sadržavat će znakovni niz "test", a u sljedećoj i posljednjoj iteraciji sadržavat će broj 3.14.
- Ovaj oblik naredbe *for* može se koristiti samo kada imamo neku kolekciju podataka (koja će se nalaziti iza operatora *in*).
- Ovaj oblik petlje prolazi kroz sve elemente zadanog skupa podataka (u ovom primjeru taj skup podataka nalazi se u varijabli *niz*), osim ako izvršavanje petlje ne prekinemo naredbom *break*.

Primjer ispisivanja elemenata liste Neka je zadana lista koja je pridružena varijabli *imena* čije elemente želimo ispisati. Treba napisati program koji ispisuje sve elemente ove liste.

Postoji više načina na koji možemo ispisati sve elemente neke kolekcije. S obzirom na to da se elementima liste može pristupati preko indeksa, to je jedan način:

```
for i in range(0, len(imena)):
    print(imena[i])
```

Međutim, postoji i jednostavniji način upotrebom drugog oblika naredbe *for*:

```
for ime in imena:
    print(ime)
```

Za prolaženje kroz sve elemente neke kolekcije preferira se ovaj oblik naredbe *for* jer je jednostavniji i manja je mogućnost greške u programiranju. Na primjer, u prvom slučaju mogli smo greškom postaviti *range(1, len(imena))* čime bismo preskočili prvi element. Isto tako, mogli smo greškom napisati *range(0, len(imena) – 1))* čime bismo izostavili posljednji element. Da smo napisali *range(0, len(imena) + 1)* izazvali bismo grešku u izvršavanju programa čime bi on bio prekinut. Sve ove greške nisu moguće u drugom obliku.

Još jedna naredba za iteraciju u Pythonu je naredba *while* koja radi na isti način kao i u drugim programskim jezicima. Osnovni oblik ove naredbe je

while *izraz*:
    naredbe

Sljedeći program riješava isti problem kao i primjer u dijelu 2.6.2, ali upotrebom naredbe *while*:

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

Petlja naredbom *for* praktičnija je ako želimo iterirati određeni broj puta ili dohvatiti svaki element kolekcije, dok je petlja naredbom *while* nešto fleksibilnija jer imamo malo više kontrole nad njenim izvršavanjem. Obje petlje, međutim, imaju iste mogućnosti pa je odabir jedne ili druge ponekad stvar osobnih preferenci.

Naredbe *break* i *continue* Petlja se može prekinuti naredbom *break* prije nego što to njen uvjet dozvoli. U sljedećem primjeru, petlja *for* prolazi kroz listu brojeva. Ako naiđe na vrijednost *None* naredbom *break* prekida se izvršavanje petlje nakon čega se izvršavanje programa nastavlja od prve naredbe koja slijedi tu petlju.

```
for e in [0, 1, None, 2]:
    print(e)
```

```
3     if e is None:
4         break
5
6 print('nastavak')
```

Ovaj program ispisuje

```
1 0
2 1
3 None
4 nastavak
```

U retku 3 provjeravamo sadrži li *e None* i, ako sadrži, prekidamo izvršavanje petlje naredbom *break* čime se prelazi na prvu naredbu koja slijedi petlju, što je u ovom primjeru `print('nastavak')`. Naredba *break* radi na isti način i s petljom *while*.

Nekada želimo preskočiti izvršavanje ostatka bloka petlje i nastaviti s idućom iteracijom. To možemo postići naredbom *continue*. U sljedećem primjeru petljom *for* ispisujemo sve elemente niza koji nisu *None*:

```
1 for e in [0, 1, None, 2]:
2     if e is None:
3         continue
4     print(e)
```

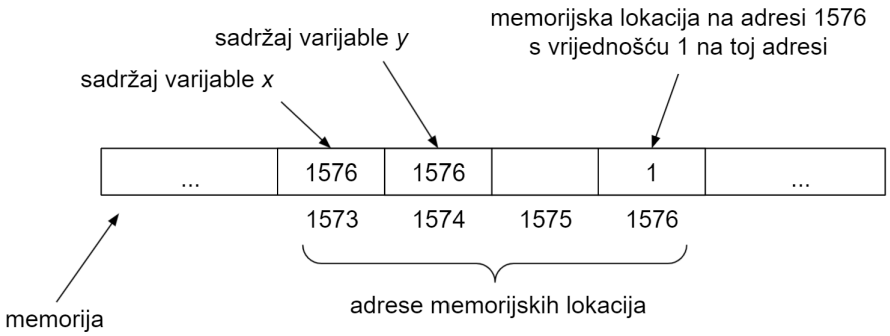
Ovaj program ispisuje

```
1 0
2 1
3 2
```

U retku 2 provjeravamo sadrži li *e None* i, ako sadrži, ne želimo da se ta vrijednost ispiše naredbom `print` u redu 4, odnosno tu vrijednost želimo preskočiti. To smo postigli naredbom *continue*: kada izvršavanje petlje naiđe na naredbu *continue* onda ono odmah prelazi na iduću iteraciju petlje - sve što slijedi naredbu *continue* neće biti izvršeno. Zato u ovom primjeru vidimo da samo *None* nije bio ispisan, ali sve druge vrijednosti prije i poslije *None* jesu. Kao i kod naredbe *break*, naredba *continue* radi na isti način i s petljom *while*.

2.7 Pristup objektima

Na početku smo vidjeli načelo po kojem Python radi s varijablama na razini memorije. Na Slici 2.1 vidjeli smo kako izgleda sadržaj memorije za pridruživanje kao što je `x = 5`. U ovom dijelu proučit ćemo još neke implikacije činjenice da varijabla u Pythonu sadrži adresu vrijednosti koja joj je pridružena. Međutim, prije toga uvest ćemo pojam *objekta* kao nečega što ima identitet odnosno adresu u memoriji. Drugim riječima, objekt je dio memorije koji predstavlja jednu cjelinu. Kada kažemo da je lista `['a', 'b']` objekt onda mislimo na jednu cjelinu na nekom segmentu memorije koja predstavlja niz od dva znakovna niza i koja ima neku memorijsku adresu. Kada kažemo da je lista `['a', 'b']`



Slika 2.3: Varijable x i y sadrže istu vrijednost.

vrijednost onda govorimo samo o nizu od ovih dvaju znakovnih nizova, neovisno o tome kako je on prikazan na razini memorije.

Dvjema ili više varijablama može biti pridružen jedan te isti objekt.

Uzmimo u obzir sljedeći niz naredbi:

```
x = 1
y = 1
```

Broj 1 je objekt kao i svaka druga vrijednost. Funkcijom *id* možemo vidjeti koja je adresa smještena u varijablama x i y:

```
>>> id(x)
2736108497200

>>> id(y)
2736108497200
```

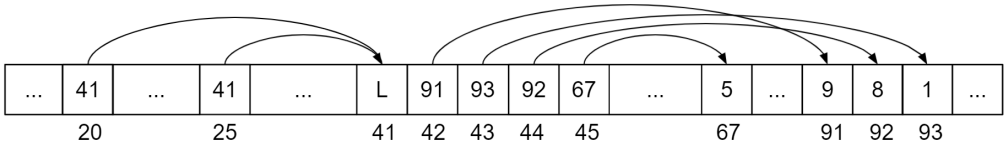
Isto tako možemo vidjeti adresu objekta 1:

```
>>> id(1)
2736108497200
```

Vidimo da i x i y sadrže istu adresu vrijednosti koja im je pridružena, kao što je ilustrirano na Slici 2.3.

Ovo samo po sebi nije problem; s obzirom da je broj 1 konstanta ona se kao takva ne može promijeniti pa se na ovaj način dobije ušteda na memoriji. Međutim, pogledajmo sličan primjer s listama. Neka je varijabli p pridružena lista kao u gornjem primjeru. Ako sada varijabli t pridružimo varijablu p onda će i t i p sadržavati adresu iste liste:

```
>>> p = ['a', 'b', 'c']
>>> t = p
>>> t
['a', 'b', 'c']
>>> p
['a', 'b', 'c']
```



Slika 2.4: Dvije varijable na adresama 20 i 25 sadrže adresu iste liste na adresi 41.

Primjer 2.8: Rješenje Zadatka 1.

```
for s in lista:
    if s.find('1') > -1:
        rezultat += [s]

# Ispisuje: ['jos 1 dan', '330-1234', '1.rujan']
print(rezultat)
```

Ovdje p i t ne sadrže samo identične liste nego i isti objekt. To znači da ako listu promijenimo preko jedne varijable ta će promijena biti vidljiva i u drugoj:

```
>>> t[1] = 'promijenio t'
>>> t
['a', 'promijenio t', 'c']
>>> p
['a', 'promijenio t', 'c']
```

Ovakav slučaj prikazan je na Slici 2.4. Dvije varijable na adresama 20 i 25 sadrže adresu iste liste koja se nalazi na adresi 41. Zbog toga bilo kakva promjena u toj listi, kao dodavanje, promjena ili brisanje elementa, bit će vidljiva kroz obje ove varijable.

Činjenicu da dvije ili više varijabli može sadržavati isti objekt važno je imati u vidu jer to može biti izvor grešaka u programiranju. Ovo nije problem za objekte koji su nepromjenjivi, kao što su brojevi, znakovni nizovi i n-torke, ali jest kod lista, rječnika i bilo kojih drugih promjenjivih tipova objekata.

2.8 Rješeni zadaci

U ovom dijelu pokazano je nekoliko primjera koje obuhvaćaju gradivo ovog poglavlja.

Zadatak 1: Za zadanu listu znakovnih nizova potrebno je napraviti novu listu koja se sastoji samo od znakovnih nizova koji sadrže znamenku 1.
Rješenje: Neka je zadana lista *lista = ['jos 1 dan', '330-1234', '2-3', 'petak je 2. dan', '1.rujna']*. U varijabli *rezultat* spremat ćemo samo one znakovne nizove koji zadovoljavaju uvjet, to jest koji sadrže znamenku 1, pa ćemo je inicijalizirati na praznu listu: *rezultat = []*. Sada slijedi glavna petlja koja prolazi kroz sve znakovne nizove, provjerava sadrži li trenutni znakovni niz znamenku '1' i, ako sadrži, dodaje taj znakovni niz u listu *rezultat* (Primjer 2.8)

Primjer 2.9: Rješenje Zadatka 2.

```
lista = ['091-333-4444', '099-235-6790',
         '095-123-3322', '099-777-6666']
rezultat = []
for s in lista:
    if s[:3] == '099':
        rezultat += [s[4:]]

# Ispisuje: ['235-6790', '777-6666']
print(rezultat)
```

Ovdje smo umjesto *rezultat += [s]* mogli napisati *rezultat.append(s)*. Prisjetimo se dijela 2.5.1, metoda *find* vraća -1 ako traženi znak ne postoji u znakovnom nizu ili indeks (poziciju) tog znaka ako postoji.

Vježba: Proširite ovaj program tako da u znakovnom nizu koji ulazi u rezultat znamenku 1 zamijeni tekstom ”jedan”.

Zadatak 2: Neka je zadana lista telefonskih brojeva u obliku znakovnog niza u formatu *xxx-xxx-xxxx*. Napišite program koji će izdvojiti samo one telefonske brojeve kojima je pozivni broj 099 (prvi dio xxx) i to bez pozivnog broja.
Rješenje: Ovaj zadatak sličan je prethodnom, samo što ovdje moramo izdvajati dijelove znakovnog niza, to jest prva tri znaka i provjeriti jesu li jednaki `'099'`. Dakle, jedno riješenje prikazano je u Primjeru 2.9.

Izraz *s[:3]* daje dio znakovnog niza od indeksa 0 do 3 (bez znaka na indeksu 3), dok izraz *s[4:]* daje dio znakovnog niza od znaka na indeksu 4 do kraja.

Vježba: Modificirajte ovaj program tako da brojeve kao ”123-4567” sprema, umjesto u rezultirajuću listu, u listu gdje bi prvi član bio ”123”, a drugi ”4567”. Rezultat bi tada izgledao ovako: `[['235', '6790'], ['777', '6666']]`.

Zadatak 3: U zadanom tekstu treba izdvojiti parne znamenke, znamenku 0 i velika slova, zajedno s položajima (indeksima) u tekstu na kojima se nalaze. Rezultat treba biti lista parova oblika *(indeks, znamenka)*.
Rješenje: Kao i u prethodnim primjerima i ovdje nam je dovoljna jedna petlja koja iterira kroz sve znakove teksta, provjerava jesu li parna znamenka, 0 ili veliko slovo i, ako jesu, dodaje ih u listu (Primjer 2.10).

U gornjem kodu treba primijetiti tri stvari:

- Tekst koji se nalazi između trostrukih navodnika (”””” ili ”’) znakovni je niz koji se sastoji od više redova.

Primjer 2.10: Rješenje Zadatka 3.

```
tekst = '''
Temperatura ce dosegnuti vise od 30 stupnjeva i
trajat ce barem 4 dana. Osjetno zahladnjenje
nastupit ce 7. rujna.
'''
znamenke = []
for indeks, znak in enumerate(tekst):
    if znak in '02468' or znak.isupper():
        znamenke += [(indeks, znak)]

# Ispisuje: [(1, 'T'), (35, '0'), (65, '4'),
#            (73, '0')]
print(znamenke)
```

- Funkcija *enumerate* vraća istovremeno i indeks elementa i sâm element.

  Funkcija *enumerate* za zadani niz elemenata (ovdje znakovni niz pridružen varijabli *tekst*) vraća par *(indeks, vrijednost)*. Dakle, u prvoj će iteraciji ova funkcija dati (0, 'T'), u drugoj (1, 'e'), u trećoj (2, 'm') i tako dalje. Na taj način imamo podatak o tome na kojem se indeksu nalazi određeni znak, ali i podatak o samom znaku. Nadalje, pridruživanje *indeks, znak in enumerate(tekst)* u jednoj naredbi pridružuje prvi element para varijabli *indeks* i drugi varijabli *znak*.

- Uvjet `if znak in '02468'` na vrlo kratak način provjerava je li varijabla *znak* jednaka jednoj od ovih znamenki. Funkcija *isupper* vraća *True* ako je dotični znakovni niz (u ovom slučaju znak) jedno veliko slovo.

Zadatak 4 Napišite funkciju *poravnaj* koja će za zadanu listu (koja može sadržavati i druge liste kao elemente) vratiti novu listu gdje su svi elementi na istoj, najvišoj razini. Na primjer, za listu [1, 2, [3, [4]], 5] treba dobiti listu [1, 2, 3, 4, 5].

Rješenje (Primjer 2.11): S obzirom da ulazna lista može sadržavati druge liste ovakav problem moramo riješiti rekurzivno. Prvo, vidimo da je funkcija *_poravnanje* dio funkcije *poravnaj* (znak _ ispred imena funkcije ovdje nema nikakvo posebno značenje nego služi samo lakšem uočavanju unutrašnje funkcije). U Pythonu tijelo funkcije može sadržavati i definicije drugih funkcija. Takva unutrašnja funkcija može se pozvati samo iz vanjske funkcije unutar koje je definirana. Drugim riječima, unutrašnja funkcija nije dostupna kodu izvana, isto kao što parametri i lokalne varijable neke funkcije nisu dostupni kodu izvan te funkcije.

Unutrašnje funkcije mogu pojednostaviti strukturu programa.

Primjer 2.11: Funkcija *poravnaj* iz primjera 1.

```
def poravnaj(lista):
    # definicija unutrasnje funkcije
    def _poravnanje(p, rezultat):
        for e in p:
            if isinstance(e, list):
                _poravnanje(e, rezultat)
            else:
                rezultat += [e]
        return rezultat

    # Naredba 'return' je dio funkcije 'poravnaj'.
    # Ovdje pozivamo funkciju '_poravnanje'.
    return _poravnanje(lista, [])

# Ispisuje: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print(poravnaj([1, [2, 3, [4, [5, 6]], [7, 8]], 9, 10]))
```

U ovom primjeru funkcija *_poravnanje* definirana je unutar funkcije *poravnaj* iz dva razloga:

1. Ova prva funkcija poziva se isključivo iz (vanjske) funkcije *poravnanje*, to jest unutrašnja je funkcija samo dio implementacije vanjske funkcije. Na taj način može se pojednostaviti struktura programa time što postoji manji broj globalnih funkcija (funkcija na razini modula).

2. Parametar *rezultat* unutrašnje funkcije je lista koja će biti vraćena kao rezultat. Ovim pojednostavljujemo popis parametara vanjske funkcije, odnosno vanjska funkcija ima samo jedan parametar: lista koju treba poravnati.

Nadalje, funkcija *_poravnanje* je rekurzivna (poziva samu sebe) [17]. S obzirom da ulazna lista može sadržavati druge liste ili podliste kao elemente, kada dođemo do elementa koji je lista moramo pozvati istu funkciju da poravna tu podlistu.

> Funkcijom *isinstance* možemo dobiti tip objekta.

Standardna funkcija *isinstance(x, T)* vraća True ako je objekt x tipa T. Kao što je prethodno objašnjeno, moramo provjeriti je li trenutni element e lista ili „običan“ element, pa nam je stoga funkcija *isinstance* neophodna.

Vježba: Napišite sličan program koji broji koliko elemenata ima zadana lista, ali tako da broji i elemente koji se nalaze u podlistama. Na primjer, za zadanu listu `[[1], [[2, 3], 4]]` taj program treba ispisati 4 (ne 2, koliko bi ispisala funkcija *len*).

Primjer 2.12: Funkcija *palindromi*.

```
def palindromi(*stringovi):
    for s in stringovi:
        if s != s[::-1]:
            return False
    return True

# Ispisuje: False
print(palindromi('abba', 'abcba', '123211'))
```

Zadatak 5 Napišite funkciju *palindromi* koja za zadane znakovne nizove vraća *True* ako su svi nizovi palindromi (isti i s lijeva i s desna). Ta funkcija treba prihvaćati neograničeni broj argumenata.

Rješenje (Primjer 2.12): Ova funkcija samo treba u petlji proći kroz sve znakovne nizove i usporediti svaki od njih sa svojom obrnutom inačicom.

> Parametar funkcije označen sa "*" može primiti neograničeni broj argumenata.

Jedina novost u gornjoj funkciji je parametar označen sa '*'. To znači da takvu funkciju možemo pozvati s nula ili više argumenata, kao što se vidi u pozivu, gdje možemo zadati neograničeni broj argumenata. Parametar *stringovi* tada možemo tretirati kao niz elemenata koji za dotični poziv sadrži elemente 'abba', 'abcba' i '123211', to jest sadrži sve argumente iz poziva. Ovo smo, naravno, mogli riješiti i tako da funkcija *palindromi* ima jedan običan parametar koji prima listu (ili n-torku). Kod nekih funkcija, međutim, kao što je *print*, ovakav način je intuitivniji jer zahtijeva samo osnovni oblik poziva funkcije (za razliku od, primjerice, parametra koji mora biti lista ili n-torka).

Vježba: Napišite funkciju za palindrome tako da je zadan samo jedan znakovni niz u kojem su riječi odvojene jednim razmakom, kao na primjer, „abba abcba 123211". Možete koristiti funkciju *split*.

Zadatak 6 Napišite funkciju *nadji* koja za zadani znakovni niz vraća sve položaje na kojima se nalaze zadani podnizovi. Rezultat treba biti oblika *[(podniz, pozicija)]*.

Rješenje (Primjer 2.13): Ovdje nam je neophodna funkcija *find* iako ćemo kasnije vidjeti kraći način rješavanja ovakvih problema. Kao i u prethodnom primjeru, ovdje nam je praktičan parametar koji može primiti neograničeni broj argumenata. Međutim, za razliku od prethodnog primjera, ovdje je prvi argument obavezan, a svi ostali su proizvoljni. Vidimo da funkcija *find* može primiti i argument koji označava indeks od kojeg počinje pretraživanje. To nam je važno jer kada dobijemo indeks na kojem se nalazi traženi podniz onda nakon toga taj isti podniz moramo tražiti počevši od sljedećeg indeksa, kako je napisano u petlji *while*. Ovdje se trebamo podsjetiti i na činjenicu da funkcija *find* vraća -1 kada traženi podniz ne može naći.

Primjer 2.13: Funkcija *nadji*.

```
def nadji(tekst, *uzorci):
    rezultat = []
    for uzorak in uzorci:
        indeks = tekst.find(uzorak)
        while indeks > -1:
            rezultat += [(uzorak, indeks)]
            indeks = tekst.find(uzorak, indeks + 1)

    return rezultat

s = '''U subotu jos toplije, ponegdje i na kopnu uz
toplu noc, koja ce u jos vise mjesta biti u nedjelju
i ponedjeljak, kada ce dani biti nestabilniji i
oblacniji, te sve manje topli, uz pljuskove i
grmljavinu, ponegdje vjerojatno i olujno nevrijeme i
tucu. U nastavku novoga tjedna promjenljivo, ali i
dalje relativno toplo.'''

# Ispisuje: [('topl', 13), ('topl', 45), ('topl', 172),
# ('topl', 316), ('nedje', 90), ('nedje', 103)]
print(nadji(s, 'topl', 'nedje'))
```

Zadatak 7 Napišite funkciju *kombinacije* koja vraća sve kombinacije elemenata liste. Na primjer, za listu [a, b] kombinacije su [], [a], [b] i [a, b].

Rješenje: Ovaj zadatak možemo riješiti upotrebom funkcije *bin(n)* koja vraća broj *n* u binarnom obliku (kao znakovni niz):

```
>>> bin(3)
"0b11"
```

Svaki element na poziciji niza na kojoj se nalazi znamenka 1 u binarnom zapisu broja (ne uzimajući u obzir prefiks *0b*) dodamo u dotičnu kombinaciju elemenata (Primjer 2.14).

Zadatak 8 Napišite funkciju *podstringovi(s)* koja vraća sve podnizove znakovnog niza *s*.

Rješenje: Počnemo sa segmentom duljine 1 i u rezultat dodajemo segmente (0, 1), (1, 2), (2, 3), ... Nakon toga duljinu segmenta povećamo na 2 pa dodajemo segmente (0, 2), (1, 3), (2, 4), ... Tako nastavimo sve dotle dok duljina segmenta ne dođe do *len(s)* (Primjer 2.15).

Primjer 2.14: Funkcija *kombinacije*.

```
def kombinacije(niz):
    rezultat = []
    for n in range(2 ** len(niz)):
        bin_br = bin(n)[2:].rjust(len(niz), '0')
        rezultat += [{e[0]
                      for e in zip(niz, bin_br)
                      if e[1] == '1'}]

    return rezultat
```

Primjer 2.15: Funkcija *podstringovi*.

```
def podstringovi(s):
    r = []
    for sirina in range(1, len(s) + 1):
        for i in range(len(s) + 1 - sirina):
            r += [s[i:i + sirina]]

    return r
```

2.9 Pregled osnovnih operatora

U ovom dijelu ukratko su u Tablici 2.1 prikazani osnovni operatori Pythona, tipovi podataka za koje su definirani i primjeri upotrebe. Neki od ovdje spomenutih tipova podataka, kao što su *dict* i *set*, bit će detaljnije opisani u narednim poglavljima. Isto tako, neke operatore u tablici ne upotrebljavamo u ovom udžbeniku ili u svim kontekstima (kao što su operatori ”&” i ”|”) - oni su tu navedeni zbog potpunijeg prikaza operatora.

Tablica 2.1: Osnovni operatori.

Op.	**Tip podatka**	**Svrha**	**Primjer upotrebe**
+	int, float, bool, str, list, tuple	zbrajanje, spoj dvaju znakovnih nizova, spoj dviju lista ili n-torki	`>>> [1, 2] + [3]` `[1, 2, 3]`
-	int, float, bool, set	oduzimanje, razlika skupova	`>>> 3.5 - 1` `2.5`
*	int, float, bool, str, list, tuple	množenje, umnožavanje znakovnih nizova, lista ili n-torki	`>>> 'ab' * 3` `'ababab'`
/	dijeljenje	int, float, bool	`>>> 7 / 2` `3.5`
//	int, float, bool	cjelobrojno dijeljenje	`>>> 7 // 2` `3`
**	int, float, bool	potenciranje	`>>> 2 ** 3` `8`
<	int, float, bool, str, list, tuple, set	manje-od, podskup	`>>> 'a' < 'b'` `True`
<=	int, float, bool, str, list, tuple, set	manje-ili-jednako-od, jednaki skupovi ili podskup	`>>> 'a' <= 'b'` `True`
>	int, float, bool, str, list, tuple, set	veće-od, nadskup	`>>> 'a' > 'b'` `False`
>=	int, float, bool, str, list, tuple, set	veće-ili-jednako-od, jednaki skupovi ili nadskup	`>>> 'a' >= 'b'` `False`
<>	int, float, bool, str, list, tuple, set	različito-od	`>>> [1, 2] <> [2, 1]` `True`
==	int, float, bool, str, list, tuple, dict, set	jednako	`>>> [1, 2] == [2, 1]` `False`
&	int, bool, set	bitni-i, presjek skupova	`>>> {'a', 'b'} & {'a', 'c'}` `{'a'}`
|	int, bool, set	bitni-ili, unija skupova	`>>> {'a', 'b'} | {'c'}` `{'a', 'c', 'b'}`

3 Rječnici i skupovi

Od tipova podataka koje smo do sada koristili, liste i n-torke su dva tipa podataka koji se zajednički nazivaju *kolekcijama* zato jer oni sadrže jednu skupinu vrijednosti. Još jedna vrsta kolekcije je *mapa* ili *rječnik* čiji je tip *dict*. Rječnik se još naziva i *asocijativnim nizom* [26]. Druga kolekcija koju ćemo ovdje vidjeti je *skup*. Skupovi podržavaju tipične operacije sa skupovima kao što su unija, presjek, provjera pripadnosti objekta skupu i druge. Tip skupa je *set*.

Tip rječnika je *dict*, a skupa *set*.

3.1 Rječnici

Pretpostavimo da želimo napisati program kojim ćemo voditi evidenciju o telefonskim brojevima i osobama kojima pripadaju. Jedna logična struktura podataka za ovaj problem je sljedeća tablica:

Korisnik	Telefon
Ivan Ivić	011 234-5678
Mara Marić	022 222-3333
Zoran Zorić	033 333-4444
...	...

Međutim, nije dovoljno samo smjestiti podatke u tablicu jer nam trebaju i određene operacije nad tom strukturom podataka. Jedna operacija koja nam treba je dodavanje novog retka u tablicu. Također nam je potrebna i mogućnost pretraživanja tablice. Ako ovakvu tablicu zamislimo kao dva niza onda bi ovakvo pretraživanje bilo jednostavno: prvo nađemo korisnika kojeg tražimo i time dobijemo indeks u listi na kojem se on nalazi; tada na istom indeksu druge liste dobijemo njegov broj telefona. Na primjer, korisnik *Mara Marić* nalazi se na indeksu 1 prvog stupca (prve liste), pa njen broj telefona dobijemo na istom indeksu drugog stupca, odnosno druge liste. Rječnici imaju ovakvo pretraživanje ugrađeno, iako je ono tehnički drugačije riješeno, ali osnovni koncept je isti. U sljedećem primjeru definiran je rječnik u koji su dodani podaci iz gornje tablice:

```
1 m = {}  # prazan rjecnik
2 m['Ivan Ivic'] = '011 234-5678'
```

```
m['Mara Maric'] = '022 222-3333'
m['Zoran Zoric'] = '033 333-4444'
```

Što znači pridruživanje kao što je `m['Ivan Ivic'] = '011 234-5678'`? Vrijednost unutar uglatih zagrada je takozvani *ključ*, a vrijednost desno od znaka jednakosti je *vrijednost* koju pridružujemo tom ključu. Ako opet pogledamo prethodnu tablicu, stupac "Korisnik" sadrži ključeve preko kojih možemo naći njima pridruženu vrijednost. Ako želimo naći vrijednost pridruženu ključu `'Ivan Ivic'` dobili bismo ju na sljedeći način:

```
v = m['Ivan Ivic']
print(v)  # ispisuje '011 234-5678'
```

Općenito, izraz *m[k]*, gdje je *m* rječnik, a *k* ključ koji pripada tom rječniku, daje vrijednost pridruženu ključu *k*.

Operatorom *in* provjeravamo pripadnost ključa rječniku.

Operator *in* koji smo koristili za provjeru pripadnosi elementa nekoj kolekciji (kao što je lista, n-torka ili znakovni niz) možemo koristiti i za provjeru pripadnosi ključa nekom rječniku. Na primjer,

```
>>> 'Ivo Ivic' in m
True
>>> 'Ante Antic' in m
False
```

Rječnik ispisujemo funkcijom *print*.

```
>>> print(m)
{'Ivan Ivic': '011 234-5678', 'Mara Maric': '022 222-3333', 'Zoran
    Zoric': '033 333-4444'}
```

Metodom *keys* možemo dobiti popis ključeva rječnika, a metodom *values* popis vrijednosti.

Ako želimo dobiti popis ključeva u rječniku možemo koristiti metodu *keys*:

```
>>> m.keys()
dict_keys(['Ivan Ivic', 'Mara Maric', 'Zoran Zoric'])
```

Na sličan način, metodom *values* možemo dobiti popis vrijednosti:

```
>>> m.values()
dict_values(['011 234-5678', '022 222-3333',
'033 333-4444'])
```

Povratne vrijednosti ovih dviju metoda je kolekcija koja sadrži sve ključeve odnosno vrijednosti rječnika. To znači da petljom *for* možemo pristupati pojedinačnim elementima te kolekcije:

```
>>> for x in m.keys(): print(x)
Ivan Ivic
Mara Maric
Zoran Zoric
```

S obzirom da je u prethodnoj petlji varijabli x pridružen ključ rječnika, na sljedeći način možemo dobiti sve njegove elemente:

```
>>> for x in m.keys():
        print("m['" + x + "'] =", m[x])
m['Ivan Ivic'] = 011 234-5678
m['Mara Maric'] = 022 222-3333
m['Zoran Zoric'] = 033 333-4444
```

Međutim, operator *in* automatski uzima ključ rječnika tako da gornju petlju možemo konciznije napisati bez upotrebe metode *keys*:

```
>>> for x in m:
        print("m['" + x + "'] =", m[x])
m['Ivan Ivic'] = 011 234-5678
m['Mara Maric'] = 022 222-3333
m['Zoran Zoric'] = 033 333-4444
```

Ključ rječnika ne mora biti samo tekstualnog već može biti bilo kojeg tipa, čak i korisničkog[1]. U gornji rječnik m možemo, primjerice, dodati i sljedeće:

```
>>> m[5] = 'test'  # kljuc je int, vrijednost je str
>>> m[(1, 'pi')] = 3.14  # kljuc je n-torka, vrijednost je float
```

Ključ rječnika može biti vrijednost bilo kojeg nepromjenjivog tipa.

Općenito, ključ rječnika može biti vrijednost bilo kojeg nepromjenjivog tipa kao što je broj, znakovni niz ili n-torka. Lista, međutim, ne može biti ključ rječnika jer su liste promjenjive.[2]

Rječnik je neuređena kolekcija podataka.

Još jedna važna činjenica vezana za rječnike je ta da su one *neuređene* kolekcije [26], što znači da poredak elemenata u rječniku nije definiran. Drugim riječima, za neki rječnik ne možemo tražiti, recimo, element koji je "drugi po redu", kao što možemo kod lista (nizova), n-torki i znakovnih nizova.

Primjer 1 Jedan tipičan primjer koji dobro ilustrira upotrebu rječnika je sljedeći: Neka je zadan niz vrijednosti A. Napišite program koji će ispisati koliko se puta svaka od tih vrijednosti pojavljuje u nizu A.

[1] Upotreba korisničkog tipa kao ključ rječnika je tehnika zasnovana na *klasama* koje su opisane kasnije.
[2] Zašto ključ rječnika mora biti nepromjenjiv?

Primjer 3.1: Funkcija *frekvencija_elemenata*.

```
def frekvencija_elemenata(a):
    mapa = {}
    for e in a:
        if e in mapa:
            mapa[e] += 1  # povecaj vrijednost za 1
        else:
            # dodaj novi kljuc i postavi vrijednost na 1
            mapa[e] = 1

    return mapa
```

Za ovakav zadatak rječnik je prikladna struktura podataka jer za svaki element niza A moramo voditi evidenciju koliko smo puta naišli na taj element prolazeći kroz niz. Na primjer, neka je niz A = [4, 1, 4, 5, 2, 4, 4, 1, 4]. Naš bi se rječnik tada trebao sastojati od sljedećih elemenata:

Ključ	**Vrijednost**
4	5
1	2
5	1
2	1

To znači da se vrijednost 4 pojavljuje pet puta, vrijednost 1 dva puta, itd. Naš program, dakle, može raditi na sljedeći način: prođemo kroz cijeli niz A i za svaki element E tog niza napravimo sljedeće:

1. Ako u rječniku postoji ključ E onda povećamo vrijednost tog ključa za 1;
2. Ako u rječniku ne postoji ključ E onda ga dodamo u rječnik i njegovu vrijednost postavimo na 1.[3]

Rješenje: U Primjeru 3.1 prikazana je funkcija *frekvencija_elemenata* koja radi ono što je gore opisano. Ova funkcija vraća frekvenciju pojavljivanja svakog elementa u nizu:

```
>>> elementi = [4, 1, 4, 5, 2, 4, 4, 1, 4]
>>> frekvencija_elemenata(elementi)
{4: 5, 1: 2, 5: 1, 2: 1}
```

Primjer 2 Napišite program koji će u znakovnom nizu naći dva ista znaka koja su na najmanjoj udaljenosti jedan od drugoga u odnosu na druge takve znakove. To riješite u jednom prolazu kroz niz. Na primjer, za *abcdcbedf* dva najbliža ista znaka su *c* jer je između njih razmak 2 (4 - 2), dok je između dva znaka *b* razmak 4.

[3] Zašto ne na 0?

Primjer 3.2: Funkcija *najblizi*.

```
def najblizi(niz):
    mapa = {}
    for i, e in enumerate(niz):
        if e not in mapa:
            mapa[e] = (i, len(niz))
        else:
            # je li ovaj razmak manji od prethodnog?
            if i - mapa[e][0] < mapa[e][1]:
                mapa[e] = (i, i - mapa[e][0])

    return min(mapa, key=lambda e: mapa[e][1])
```

Rješenje: Za ovaj problem možemo koristiti rječnik u koji ćemo spremati razmake između istih znakova kako je pokazano u funkciji *najblizi* u Primjeru 3.2.

3.2 Skupovi

Skup je neuređena kolekcija koja podržava operacije sa skupovima.

Još jedna neuređena kolekcija je *skup*. Skup je korisna struktura podataka u praksi zbog sljedećeg:

1. Skup podržava matematičke operacije sa skupovima, kao što su unija, presjek, razlika i provjera pripadnosti elementa skupu.
2. Skup nema duplikata pa je stoga konverzija nekog niza elemenata u skup najlakši način eliminacije duplikata.

Skup se u Pythonu označava isto kao i u matematici - vitičastim zagradama. Na primjer, sljedeći skup pridružen varijabli s sadrži jedan cijeli broj, jedan realni broj i jedan znakovni niz:

```
>>> s = {44, 2.8, 'abc'}
>>> print(s)
{2.8, 44, 'abc'}
```

Skupovi, kao i sve ostale kolekcije, mogu sadržavati elemente različitog tipa. U Primjeru 3.3 demonstrirane su tipične operacije sa skupovima. Možemo primjetiti da se kod unije broj 3 ne pojavljuje dva puta u rezultatu jer se duplikati automatski eliminiraju.

Za prazan skup ne postoji specijalna sintaksa nego jednostavno pozovemo konstruktor za skup:

```
>>> p = set()
>>> p
set()
```

Primjer 3.3: Tipične operacije nad skupovima.

```
>>> s1 = {1, 2, 3}
>>> s2 = {3, 4, 5}
>>> s1 | s2  # unija
{1, 2, 3, 4, 5}
>>> s1 & s2  # presjek
{3}
>>> s1 - s2  # razlika
{1, 2}
>>> 2 in s1  # provjera pripadnosti
True
>>> {3, 5} < s2  # je li {3, 5} podskup od s2?
True
```

Primjer 3.4: Primjer upotrebe skupova.

```
>>> kolokvirali = {'Ivo', 'Marija', 'Ana', 'Josip', 'Petar'}
>>> imaju_dolaske = {'Ivo', 'Marija', 'Ana', 'Petar'}
>>> varali = {'Ivo', 'Ana'}
>>> rezultat = kolokvirali & imaju_dolaske - varali
>>> rezultat
{'Petar', 'Marija'}
```

Pomoću ovog konstruktora svaku kolekciju možemo konvertirati u skup:

```
>>> set([1, 2, 3])
{1, 2, 3}
>>> set('abc')
{'a', 'c', 'b'}  # poredak je nedefiniran
```

Iz skupa možemo dobiti listu:

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

U sljedećem primjeru u funkciji *broj_duplikata* pomoću skupova utvrđujemo koliko duplikata postoji u zadanoj listi:

```
>>> def broj_duplikata(p):
        return len(p) - len(set(p))

>>> broj_duplikata([1, 2, 1, 1, 2, 3, 2])
4
```

U funkciji *broj_duplikata* jednostavno izračunamo razliku između broja elemenata zadane liste p i broja elemenata kada tu listu konvertiramo u skup, što nam daje ukupan broj duplikata u listi.

Primjer 3.5: Funkcija *zajednicki*.

```
def zajednicki(a, b):
    return set(a) & set(b)
```

Primjer 1 Pretpostavimo da imamo tri skupine studenata: one koji su kolokvirali, one koji imaju dovoljan broj dolazaka na vježbe i one koji su varali na domaćim zadaćama. Treba naći studente koji su kolokvirali, imaju dovoljan broj dolazaka na vježbe i nisu varali na domaćim zadaćama.

Rješenje: U Primjeru 3.4 pokazan je kôd koji daje odgovarajuće rješenje primjenom skupovnih operacija. U rezultat nisu uključeni Ivo i Ana jer su oni u skupu *varali*, kao ni Josip koji nije u skupu *imaju_dolaske*.

Primjer 2 Upotrebom skupova napišite funkciju *zajednicki(a, b)* koja vraća zajedničke elemente lista *a* i *b*:

```
>>> zajednicki([3, 1, 7, 5], [4, 1, 5, 1, 5])
[1, 5]
```

Rješenje: Za ovaj problem možemo jednostavno obje liste konvertirati u skup i upotrebom skupovne operacije *presjek* (&) dobiti zajedničke elemente (Primjer 3.5).

3.3 Obuhvaćanje lista, rječnika i skupova

3.3.1 Obuhvaćanje lista

Vrijednosti možemo dodati u listu na nekoliko načina:

- metodom *append* ako dodajemo samo jedan element
- operatorom "+=" ili metodom *extend* ako dodajemo više elemenata
- obuhvaćanjem liste.

Ovaj posljednji način, *obuhvaćanje liste*, kraći je način dodavanja elemenata u listu.

Pretpostavimo da u listu želimo dodati nekoliko slučajnih cijelih brojeva. To možemo lako napraviti tako da u petlji dodajemo jedan po jedan broj:

```
import random  # modul za rad sa slucajnim brojevima

lista = []
for i in range(10):
    lista.append(random.randint(0, 100))

>>> lista
[79, 86, 66, 33, 46, 20, 90, 82, 77, 34]
```

Ovaj program ispisuje listu koja sadrži deset slučajnih brojeva čija je vrijednost između 0 i 100. Upotrebom sintakse obuhvaćanja liste ovo možemo napisati i ovako:

```
import random

>>> [random.randint(0, 100) for _ in range(10)]
[22, 68, 93, 70, 15, 73, 42, 50, 14, 63]
```

Kao prvo, vidimo da nam ne trebaju metode *append*, *extend* ili operator "+=". Izrazom `[random.randint(0, 100) for _ in range(10)]` rekli smo od čega se ova lista sastoji: od deset slučajnih brojeva. Da smo htjeli da se lista sastoji od jednog slučajnog broja napisali bismo

```
>>> [random.randint(0, 99)]
[60]
```

Međutim, ovdje smo htjeli da lista sadrži deset vrijednosti pa smo pored izraza `random.randint(0, 99)` naznačili da želimo da se on izvrši deset puta time što smo napisali `for _ in range(10)`. Znak "_" zamjenjuje varijablu, kao što je *i*, jer nam ovdje ta varijabla ne treba, ali mogli smo napisati i nešto kao `for x in range(10)`. Primjerice, ako nam treba lista koja se sastoji od brojeva 0, 1, 2, ..., 9 možemo napisati ovakav izraz:

```
>>> [x for x in range(0, 10)]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Nadalje, desno od *for*-dijela može se nalaziti uvjet koji određuje hoće li neka vrijednost biti dodana u listu. Pretpostavimo da želimo da lista sadrži samo parne brojeve. To možemo postići na sljedeći način:

```
>>> [x for x in range(0, 10) if x % 2 == 0]
[0, 2, 4, 6, 8]
```

Na ovaj način možemo filtrirati neku postojeću listu. Primjerice, neka je zadana lista koja se sastoji od slučajnih brojeva između 0 i 99. Ako iz takve liste želimo izdvojiti samo one brojeve koji su manji od 10 to možemo postići na ovaj način:

```
>>> import random
>>> p = [random.randint(0, 99) for _ in range(10)]
>>> p
[99, 57, 63, 55, 94, 66, 73, 59, 38, 9]

>>> [x for x in p if x < 10]
[9]
```

Izrazom `[x for x in p if x < 10]` rekli smo da želimo definirati listu koja se sastoji od svakog elementa `x` liste `p` koji je manji od 10.

Primjer Napišite funkciju *duplikati* koja za svaki duplikat liste vraća broj njegovog pojavljivanja.

Primjer 3.6: Funkcija *duplikati*.

```
def duplikati(p):
    return set([(x, p.count(x)) for x in p if p.count(x) > 1])
```

```
>>> duplikati([3, 8, 1, 8, 4, 3, 8, 8])
{(3, 2), (8, 4)}
```

Rješenje: Jedan način da riješimo ovaj zadatak je upotrebom rječnika gdje svaki element dodamo kao ključ, ako već nije u rječniku, s tim da mu postavimo početnu vrijednost na 1. Ako je već u rječniku onda mu samo povećamo vrijednost za 1. Međutim, jedno jednostavnije rješenje je upotrebom konstrukta obuhvaćanja liste prikazano je u Primjeru 3.6. Prvo napravimo listu koja se sastoji od parova gdje je prvi član element zadane liste koji se pojavljuje više puta u listi, a drugi broj koliko se puta taj element pojavljuje u listi. Funkcijom *set* tada eliminiramo duplikate iz rješenja jer elementi koji se pojavljuju više puta u listi bit će dodani u rezultat više puta.

3.3.2 Obuhvaćanje rječnika

Kao što postoji obuhvaćanje liste, tako postoji i obuhvaćanje rječnika. Na primjer, ako želimo dobiti rječnik `{'a': 1, 'b': 2, 'c': 3}` možemo napisati sljedeće:

```
>>> {k:v for k, v in [('a', 1), ('b', 2), ('c', 3)]}
{'a': 1, 'b': 2, 'c': 3}
```

S obzirom da se rječnik sastoji od parova *(ključ, vrijednost)* moramo definirati oba elementa što smo gore napravili s `k:v`. Isto tako, u *for*-djelu naznačili smo da vrijednost za `k` dolazi od prvog člana svakog para, a za `v` od drugog.

I ovdje možemo zadati uvjet:

```
>>> {k:v for k, v in [('a', 1), ('b', 2), ('c', 3)]
          if k != 'c'}
{'a': 1, 'b': 2}
```

3.3.3 Obuhvaćanje skupova

Na sličan način možemo definirati skupove. Na primjer, skup od najviše 10 različitih slučajnih pozitivnih cijelih brojeva između 0 i 100 možemo definirati ovako:

```
import random

>>> {random.randint(0, 100) for _ in range(10)}
{3, 9, 22, 28, 33, 51, 71, 75, 88, 99}

>>> {random.randint(0, 100) for _ in range(10)}
{0, 16, 22, 24, 31, 69, 75, 83, 89}
```

U drugom primjeru vidimo da smo dobili skup od devet, a ne deset brojeva. To je zato što je funkcija `randint` generirala jedan broj dva puta, a s obzirom da skupovi ne mogu sadržavati duplikate taj drugi isti broj nije bio dodan u skup.

Obuhvaćanje lista, skupova i rječnika omogućava nam da na koncizan način (najčešće u jednom redu) definiramo takve kolekcije podataka bez eksplicitnog dodavanja elemenata u njih [27].

4 Lambda izrazi i funkcijske vrijednosti

4.1 Lambda izrazi

U Pythonu postoji kraći način pisanja jednostavnih funkcija, to jest funkcija čije se tijelo sastoji od samo jednog izraza čiji rezultat je rezultat te funkcije. Takvi se izrazi nazivaju *lambda izrazima*.

> Naredbom *lambda* možemo definirati funkciju.

Ispod je primjer lambda izraza koji vraća zbroj dvaju brojeva zadanih parametrima tog izraza odnosno funkcije:

```
>>> zbroj = lambda m, n: m + n      # nema 'return'
```

Rezultat lambda izraza je funkcija. Gornju funkciju možemo pozvati na isti način kao i funkciju definiranu naredbom *def*:

```
>>> zbroj(2, 5)
7
```

> Lambda izrazi praktični su za definiranje kratkih funkcija koje će biti zadane kao argument poziva druge funkcije poput uvjeta za selekciju elemenata ili sortiranja.

Opći oblik lambda izraza je

lambda [*parametri*] **:** *izraz*

(dio u uglatim zagradama je opcionalan).

Tijelo funkcije definirane naredbom *lambda* nema naredbu *return*, nego je vraćanje rezultata implicitno. Zašto nam trebaju dva načina definiranja funkcije? Gornji smo primjer mogli napisati i na klasičan način, to jest

```
def zbroj(m, n): return m + n
```

Lambda izrazi korisni su za kratke funkcije koje možemo lako zadati kao argument poziva drugih funkcija. Primjerice, u Pythonu postoji funkcija *filter* koja za zadani niz vrijednosti vraća samo one koje zadovoljavaju zadani uvjet. Taj uvjet često je jednostavan, kao što je „svi brojevi veći od 0“. Takvi se uvjeti na kratak način mogu specificirati upotrebom lambda izraza:

```
>>> elementi = [5, -8, -2, 9, 7]
>>> list(filter(lambda e: e > 0, elementi))
[5, 9, 7]
```

Funkcija *filter* prolazi kroz sve elemente zadane liste (u ovom slučaju elementi) i za svaki element poziva funkciju zadanu u svom drugom parametru, što je u gornjem slučaju funkcija

```
lambda e: e > 0
```

Ako ta funkcija vrati *True* za dotični element onda funkcija *filter* taj element dodaje u rezultat, u suprotnom ga izostavlja. Isto smo mogli napisati i na duži način:

```
>>> def veci_od_0(e): return e > 0
>>> list(filter(veci_od_0, elementi))
[5, 9, 7]
```

Ako nam uvjet za selekciju elemenata treba samo za funkciju *filter* onda nema potrebe definirati funkciju naredbom *def* već je jednostavnije taj uvjet zadati izravno pomoću lambda izraza kao argument poziva funkcije *filter*. Još jedan primjer upotrebe lambda izraza je standardna funkcija *sorted*. Pretpostavimo da imamo sljedeću listu imena:

```
>>> imena = ['Petar', 'Marko', 'Ivo', 'Ivan', 'Anabela']
```

Funkciju *sorted* možemo upotrijebiti da bismo dobili ovaj niz imena sortiran po abecedi:

```
>>> sorted(imena)
['Anabela', 'Ivan', 'Ivo', 'Marko', 'Petar']
```

Međutim, što ako želimo ovakav popis sortirati po nekom drugom kriteriju kao što je duljina znakovnog niza? Bilo bi nepraktično pisati posebnu funkciju koja sortira po tom kriteriju jer nam tako može zatrebati i sortiranje po nekom trećem kriteriju kao što je broj samoglasnika i sl. Funkcija *sorted* omogućava zadavanje kriterija sortiranja kao argument koji mora biti funkcija koja vraća vrijednost za koju je definirana operacija $<$. Na primjer, sortiranje po duljini znakovnog niza možemo specificirati ovako:

```
>>> sorted(imena, key=lambda e: len(e))
['Ivo', 'Ivan', 'Petar', 'Marko', 'Anabela']
```

Kada funkcija *sorted* uspoređuje dva elementa za svaki od njih prvo pozove funkciju zadanu u parametru *key* i onda usporedi rezultate tih poziva funkcija. Na primjer, ako uspoređuje elemente ”Ivo” i ”Anabela” tada pozivom funkcije u parametru *key* dobije vrijednosti 3 i 7 za ova dva znakovna niza te usporedi ta dva broja na osnovu čega odredi koji je element ispred.

Primjer 4.1: Primjer funkcija višeg reda.

```
def kvadrirani_niz(niz):
    rezultat = []
    for e in niz:
        rezultat += [e * e]

    return rezultat

def apsolutni_niz(niz):
    rezultat = []
    for e in niz:
        rezultat += [abs(e)]

    return rezultat

def invertirani_niz(niz):
    rezultat = []
    for e in niz:
        rezultat += [-e]

    return rezultat
```

Općenito, jedina razlika između lambda izraza i operacija definiranih naredbom *def* je u tome da lambda izrazi nemaju naredbu *return* jer se oni sastoje od samo jednog izraza čiji rezultat je rezultat operacije definirane takvim izrazom. Kada se govori o lambda izrazima važno je jasno razlikovati rezultat lambda izraza od rezultata operacije definirane takvim izrazom:

- Rezultat lambda izraza je funkcija definirana tim izrazom, a ne rezultat izvršavanja te same funkcije.
- Rezultat funkcije definirane lambda izrazom je rezultat izvršavanja te funkcije, to jest rezultat izraza koji se nalazi iza znaka ":" u lambda izrazu.

Tretiranje funkcija kao vrijednosti korisna je tehnika programiranja i ovdje ćemo proučiti još nekoliko primjera. Općenito, funkcije koje primaju druge funkcije kroz jedan ili više parametara i/ili koje vraćaju druge funkcije kao rezultat nazivaju se *funkcijama višeg reda* [28, 29]. Funkcija *filter* je, prema tome, jedna od takvih funkcija, to jest funkcija višeg reda [29].

Kao još jedan primjer pogledajmo funkcije u Primjeru 4.1. Te tri funkcije za zadani niz brojeva daju novi niz brojeva gdje je svaki broj tog novog niza dobiven na osnovu broja na istom položaju u zadanom nizu, i to na ovaj način:

1. Broj novog niza je kvadrat broja zadanog niza (funkcija *kvadrirani_niz*).

2. Broj novog niza je apsolutna vrijednost broja zadanog niza (funkcija *apsolutni_niz*).

3. Broj novog niza je invertirana vrijednost broja zadanog niza (funkcija *invertirani_niz*).

Rezultati ovih operacija izgledaju ovako:

```
>>> kvadrirani_niz([4, -2, -7, 3])
[16, 4, 49, 9]

>>> apsolutni_niz([4, -2, -7, 3])
[4, 2, 7, 3]

>>> invertirani_niz([4, -2, -7, 3])
[-4, 2, 7, -3]
```

Kod ove tri funkcije možemo lako primijetiti da su međusobno gotovo identične. Ako izlučimo dio koda koji je isti kod svih triju funkcija možemo vidjeti što se ponavlja:

```
rezultat = []
for e in niz:
    rezultat += [...]

return rezultat
```

Iz ovoga vidimo da je jedini dio koji se razlikuje kod ovih funkcija onaj označen s "...", gdje na tom mjestu

1. *kvadrirani_niz* koristi `e * e`

2. *apsolutni_niz* koristi `abs(e)`

3. *invertirani_niz* koristi `-e`

U sva tri slučaja radi se o jednom izrazu čiji rezultat dodajemo u niz *rezultat*. Može se reći da je način rada, ili općeniti postupak, ovih triju funkcija isti: za svaki element ulaznog niza dodaj rezultat neke funkcije s tim elementom u izlazni niz. Ono što se razlikuje kod ovih funkcija je to što radimo sa svakim elementom ulaznog niza. Praktičnost funkcija višeg reda sada ćemo proučiti kroz nekoliko primjera. Funkcije kao što su gornje tri obično se nazivaju transformacijama jer je njihov rezultat niz koji je transformacija ulaznog niza.

Da bismo izbjegli pisanje više ovakvih funkcija koje sve rade na isti način trebamo mjesto koje smo gore označili sa "..." zamijeniti pozivom neke funkcije koja će dati odgovarajući rezultat za svaki element ulaznog niza:

```
rezultat = []
for e in niz:
    rezultat += [operacija(e)]

return rezultat
```

Primjer 4.2: Funkcija *transformacija*.

```
def transformacija(niz, operacija):
    rezultat = []
    for e in niz:
        rezultat += [operacija(e)]

    return rezultat
```

S obzirom da nam je cilj da se varijabla *operacija* odnosi na različite operacije, zavisno od toga što želimo da budu elementi izlaznog niza, *operacija* može biti jedan parametar kojem ćemo u toku poziva pridružiti neku specifičnu funkciju, slično kao što smo prethodno varijabli *op* pridružili funkciju kvadrat. Sada možemo definirati jednu funkciju višeg reda koju ćemo zvati *transformacija* i koja je prikazana u Primjeru 4.2. Kod poziva ove funkcije parametru *operacija* jednostavno ćemo pridružiti neku funkciju:

```
>>> transformacija([4, -2, -7, 3], kvadrat)
[16, 4, 49, 9]
```

S obzirom da funkcijske vrijednosti možemo pridruživati varijablama kao i sve druge vrijednosti, argument za parametar *operacija* može biti i lambda izraz:

```
>>> transformacija([4, -2, -7, 3], lambda x: x * x)
[16, 4, 49, 9]
```

Prvi argument funkcije transformacija je niz brojeva, a drugi je funkcija čiji će rezultati biti dodavani u izlazni niz *rezultat* u svakoj iteraciji. Ta se funkcija pridružuje parametru *operacija* koji tada (unutar funkcije *transformacija*) postaje ime te funkcije, na isti način kao i kod definiranja funkcija korištenjem naredbe *def*. Funkcija *transformacija* svaki element ulaznog niza vraća kao argument poziva te funkcije čiji rezultat se onda dodaje u izlazni niz. Na primjer, u prvom slučaju parametru *operacija* pridružuje se funkcija kvadrat, dok u drugom slučaju radimo to isto koristeći lambda izraz *lambda x: x * x*. To znači da unutar funkcije *transformacija* možemo pisati izraz *operacija(n)*, isto kao da smo negdje definirali funkciju s imenom *operacija*. Važno je imati u vidu da funkcija pridružena ovom parametru može biti bilo koja funkcija pod uvjetom da ima jedan parametar zato jer joj dajemo jedan argument, *e*, to jest pozivamo je pomoću `operacija(e)`. Funkcija *kvadrat* ima jedan parametar, *x*, a isto tako ga ima i funkcija definirana lambda izrazom tako da ove dvije funkcije zadovoljavaju taj uvjet. Funkciju *transformacija* sada možemo koristiti i za mnoge druge svrhe, gdje god nam treba ovakva transformacija niza:

```
>>> transformacija([4, -2, -7, 3], lambda x: abs(x))
[4, 2, 7, 3]

>>> transformacija([4, -2, -7, 3], lambda x: -x)
[-4, 2, 7, -3]
```

Primjer 4.3: Funkcija *sortiraj_niz*.

```
def sortiraj_niz(niz):
    for i in range(0, len(niz) - 1):
        for j in range(i + 1, len(niz)):
            if niz[j] < niz[i]:
                niz[i], niz[j] = niz[j], niz[i]
```

Primjer 4.4: Funkcija *sortiraj*.

```
def sortiraj(niz, predikat):
    for i in range(0, len(niz) - 1):
        for j in range(i + 1, len(niz)):
            if predikat(niz[j], niz[i]):
                niz[i], niz[j] = niz[j], niz[i]
```

Umjesto da definiramo nekoliko funkcija koje sve rade na isti način i u načelu izvršavaju isti set naredbi, to jest transformiraju niz, definirali smo jednu funkciju koju možemo "konfigurirati" za razne transformacije koristeći funkcijske vrijednosti.

Ovdje treba uočiti praktičnost lambda izraza koja je u tome da možemo definirati kratke funkcije na mjestu gdje nam one trebaju, kao što je u gornjem primjeru mjesto argumenta za poziv funkcije *transformacija*. Lambda izrazi, prema tome, postoje samo iz praktičnih razloga, da ne moramo definirati puno kratkih funkcija korištenjem naredbe *def* koje imaju samo naredbu *return* u tijelu funkcije.

Funkcije kao što je *transformacija* korisne su zbog svoje višestruke upotrebe. Takve funkcije možemo definirati kada god uočimo da više funkcija radi na isti način. Uzmimo u obzir funkciju *sortiraj_niz* iz Primjera 4.3 koja sortira niz vrijednosti. Ova funkcija sortira vrijednosti od najmanje prema najvećoj. Ako, međutim, želimo te vrijednosti sortirati od najveće prema najmanjoj, ovu funkciju ne možemo koristiti zato jer u njoj ne možemo lako mijenjati način usporedbe dviju vrijednosti kako i kada nam to odgovara. To ne bi bio problem kada bi ova funkcija bila napisana tako da je uspoređivanje vrijednosti odvojeno od same operacije sortiranja, kao što je pokazano u novoj funkciji *sortiraj* u Primjeru 4.4.

Vidimo da su funkcije *sortiraj_niz* i *sortiraj* jako slične - jedina je razlika u tome da u funkciji *sortiraj* umjesto usporedbe "<" koristimo neku funkciju pridruženu parametru *predikat*, a ta funkcija mora imati dva parametra i njen rezultat mora biti logička vrijednost. Sada ovu funkciju možemo koristiti za razne vrste sortiranja:

```
>>> a = [7, 2, 1, 6]
>>> sortiraj(a, lambda x, y: x > y)
>>> a
[7, 6, 2, 1]

>>> sortiraj(a, lambda x, y: x < y)
>>> a
```

```
[1, 2, 6, 7]
```

Da bismo vidjeli kako radi izraz kao što je `sortiraj(a, lambda x, y: x < y)`, prvo možemo lambda izraz pridružiti nekoj varijabli:

```
predikat = lambda x, y: x < y
```

Ovdje imamo istu situaciju kao i kod gornjeg poziva funkcije *sortiraj*, gdje je ovaj lambda izraz bio pridružen parametru *predikat*. Varijablu *predikat* sada možemo koristiti isto kao i bilo koju drugu funkciju:

```
>>> predikat(1, 2)
True
```

U ovom pozivu, vrijednost 1 bila je pridružena parametru x lambda izraza, a vrijednost 2 parametru y. Na potpuno isti način koristimo parametar *predikat* u funkciji *sortiraj*, s tim da tamo koristimo vrijednosti iz ulaznoga niza. Rezultat je logička vrijednost izraza `x < y` koju koristimo za naredbu *if*, isto kao i da smo napisali `if x < y`. Prednost korištenja lambda izraza za funkciju *sortiraj* je u tome da redosljed sortiranih elemenata nije unaprijed utvrđen, nego ga se može specificirati po potrebi, što nam omogućava puno veću fleksibilnost i širu primjenjivost ove funkcije.

Funkciju *sortiraj* možemo koristiti i s kompleksnijim podacima. Pretpostavimo da imamo niz parova gdje je prvi član svakog para neki broj:

```
b = [[2150, 'A'], [450, 'B'], [1200, 'C'], [25, 'D']]
```

Funkcijom *sortiraj* možemo sortirati ovaj niz na osnovu numeričke vrijednosti u svakom unutrašnjem nizu:

```
>>> sortiraj(b, lambda x, y: x[0] < y[0])
>>> b
[[25, 'D'], [450, 'B'], [1200, 'C'], [2150, 'A']]
```

Parametri x i y funkcijske vrijednosti sadržavat će po jedan par, kao što je [450, 'B'], jer ti su parovi elementi ulaznog niza za funkciju *sortiraj* i argumenti su za funkciju pridruženu parametru *predikat*. Prema tome, izrazom `x[0] < y[0]` uspoređujemo brojeve tih parova. Usporedba elemenata niza može biti i kompleksnija od nečega što možemo napisati kao lambda izraz. Pretpostavimo da želimo sortirati niz znamenki napisanih riječima:

```
c = ['tri', 'dva', 'četiri', 'jedan']
```

Riječi ovog niza želimo sortirati kao brojeve koje te riječi označavaju, a ne kao tekst (abecednim redom). Za tu svrhu definirat ćemo funkciju `ispred(x, y)` koja vraća *True* ako je x ispred y (Primjer 4.5).Ova funkcija radi tako da prolazi kroz niz znamenki napisanih riječima (za ovaj primjer koristimo znamenke od 1 do 5), s tim da su te znamenke poslagane po redu, i ako nađe riječ koju sadrži x prije nego što nađe riječ koju sadrži y onda je x ispred y. U suprotnom, x nije ispred y. Ovu operaciju sada možemo koristiti kao argument za operaciju sortiranja:

```
>>> sortiraj(c, ispred)
>>> c
['jedan', 'dva', 'tri', 'četiri']
```

Primjer 4.5: Funkcija *ispred*.

```
def ispred(x, y):
    for broj in ['jedan', 'dva', 'tri', 'četiri', 'pet']:
        if broj == x:
            # x je pronadjen prvi - x je ispred y
            return True
        elif broj == y:
            # y je pronadjen prvi - x nije ispred y
            return False
```

Primjer 4.6: Funkcija *transformacija_n*.

```
def transformacija_n(niz, operacija):
    rezultat = []
    for indeks in range(0, len(niz)):
        rezultat += [operacija(niz[indeks], indeks)]

    return rezultat
```

Funkcija *ispred* vraća *None* ako zadana riječ ne postoji u njenom nizu, ali to nije problem jer naredba *if* tretira *None* kao *False*. Funkciju *ispred* mogli smo napisati kraće i jednostavnije koristeći postojeću funkciju *index* koja vraća indeks na kojem se u nizu nalazi traženi element:

```
>>> c.index('tri')
2
```

Funkcija *ispred* tada bi mogla izgledati ovako:

```
def ispred(x, y):
    znamenke = ['jedan', 'dva', 'tri', 'četiri', 'pet']
    return znamenke.index(x) < znamenke.index(y)
```

Ako pogledamo funkcije *sortiraj* i *ispred* vidimo da su one definirane svaka za sebe i u niti jednoj od njih nema ništa što bi zahtjevalo ili upućivalo na postojanje druge. Međutim, te dvije funkcije možemo kombinirati tako da rade zajedno kao jedna funkcija! Kombiniranje funkcija na ovakav način izuzetno je korisna tehnika programiranja i mnogi je moderni programski jezici omogučavaju. Kao što smo vidjeli iz ovih nekoliko primjera, osnovno je načelo ove tehnike da jednu funkciju tretiramo kao vrijednost koju možemo pridružiti nekoj varijabli, gdje zatim tu varijablu koristimo kao ime te funkcije. U nastavku ćemo definirati još nekoliko funkcija koje rade na sličan način kao i *transformacija*.

Funkcija *transformacija_n* u Primjeru 4.6 radi isto što i *tranformacija*, ali kod poziva funkcije zadane u parametru *operacija* vraća i indeks elementa ulaznog niza. To znači da funkcija pridružena tom parametru mora imati dva parametra - prvi je element koji se transformira, a drugi indeks tog elementa.

U programiranju nam je često potrebna mogućnost izdvajanja samo onih elemenata

Primjer 4.7: Funkcija *selekcija*.

```
def selekcija(niz, predikat):
    rezultat = []
    for e in niz:
        if predikat(e):
            rezultat += [e]

    return rezultat
```

Primjer 4.8: Funkcija *broj_pojavljivanja*.

```
def broj_pojavljivanja(niz):
    rezultat = []
    while len(niz) > 0:
        e = niz[0]
        duljina = len(niz)    # duljina prije selekcije
        niz = selekcija(niz, lambda x: x != e)
        rezultat += [[e, duljina - len(niz)]]

    return rezultat

>>> broj_pojavljivanja(['a', 'f', 'c', 'f', 'f', 'c'])
[['a', 1], ['f', 3], ['c', 2]]
```

jednog niza koji zadovoljavaju neki uvjet. Na primjer, iz niza brojeva možemo izdvojiti samo parne ili neparne brojeve. S obzirom da ne želimo pisati na desetke funkcija od kojih svaka izdvaja drugačije elemente, definirat ćemo funkciju *selekcija* koja za zadani ulazni niz kao rezultat vraća novi niz koji se sastoji samo od onih elemenata ulaznog niza koji zadovoljavaju neki zadani uvjet (Primjer 4.7). Ova funkcija radi slično kao *transformacija*, samo što u izlazni niz ne dodaje svaki rezultat, nego samo one za koje funkcija pridružena parametru *predikat* vraća *True* (očito je da ta funkcija mora biti predikat, odnosno njen rezultat mora biti logička vrijednost). Ta funkcija služi kao uvjet za element koji joj je zadan kao argument. Za razliku od funkcije *transformacija*, ovdje rezultirajući niz može imati manje elemenata od ulaznog niza. Sada, na primjer, funkciju *selekcija* možemo koristiti za izlučivanje parnih ili neparnih brojeva iz zadanog niza brojeva:

```
>>> a = [73, 216, 41, 100, 99, 7, 18, 21]
>>> selekcija(a, lambda x: x % 2 == 0)
[216, 100, 18]

>>> selekcija(a, lambda x: x % 2 != 0)
[73, 41, 99, 7, 21]
```

U Primjeru 4.8 koristimo funkciju *selekcija* u definiciji funkcije *broj_pojavljivanja* koja kaže koliko se puta svaki element pojavljuje u zadanom nizu. Ova funkcija radi

Primjer 4.9: Funkcija *prvi*.

```
def prvi(niz, operacija):
    for e in niz:
        if operacija(e):
            return e
```

tako da uzme element na poziciji 0 u nizu, napravi selekciju tog niza tako da ostanu samo elementi koji nisu jednaki tom elementu, i onda izračuna razliku između broja elemenata početnog niza i reduciranog niza (to jest niza nakon selekcije). Ovdje ne možemo provesti selekciju elemenata koji su jednaki zadanom elementu jer nam trebaju svi ostali elementi za nastavak selekcije. U suprotnom bismo nakon prve iteracije ostali bez ostalih elemenata i dogodila bi se beskonačna petlja. U svakoj iteraciji varijabli *rezultat* dodajemo niz od dva elementa: element koji je bio eliminiran iz početnog niza i broj koji predstavlja ovu razliku, odnosno koliko se puta taj element pojavljuje u nizu. Ovaj se proces nakon toga ponavlja za sljedeći element koristeći taj reducirani niz. S obzirom da ova funkcija radi s bilo kakvim nizom elemenata možemo je, na primjer, koristiti da saznamo koliko puta se neko slovo pojavljuje u zadanoj rečenici:

```
>>> broj_pojavljivanja('danas je lijep dan')
[['d', 2], ['a', 3], ['n', 2], ['s', 1], [' ', 3], ['j', 2], ['e',
    2], ['l', 1], ['i', 1], ['p', 1]]
```

Jednostavna inačica funkcije *selekcija*, koju ćemo zvati *prvi*, kao rezultat vraća samo onaj prvi element niza koji zadovoljava uvjet, bez prolaska kroz ostatak niza. Zbog toga funkcija *prvi* taj element ne dodaje u niz nego ga samo vraća kao rezultat (Primjer 4.9). S obzirom da ova funkcija vraća samo prvi element u nizu koji zadovoljava zadani uvjet, korisna je kada nas zanima samo postoji li takav element, a ne koji je to element. Ako, na primjer, želimo saznati postoje li u nekom nizu negativni brojevi možemo koristiti funkciju *prvi*:

```
>>> if prvi(a, lambda x: x < 0) == None:
        print('Niz nema negativnih brojeva')
    else:
        print('Niz ima negativnih brojeva')

Niz nema negativnih brojeva
```

Prednost funkcije *prvi* u odnosu na funkciju *selekcija* samo je u broju iteracija - *selekcija* uvijek prolazi cijeli niz, dok funkcija *prvi* stane čim nađe prvi element koji zadovoljava zadani uvjet (što može biti i posljednji element u nizu, gdje bi tada i ova funkcija prošla cijeli niz, kao i slučaju kada takav element ne bi postojao).

Još jedna korisna funkcija, koju ćemo zvati *redukcija*, za zadani niz vrijednosti i funkciju kao rezultat vraća jednu vrijednost. Preciznije rečeno, za ulazni niz *[a_1, a_2, …, a_n]* i funkciju *f* rezultat ove operacije odgovara rezultatu izraza *… f(f(f(f(a1, a2), a3), a4), a5) …*. Na primjer, za ulazni niz [14, 5, 20, 1, 9] i funkciju *max* (koja je već

Primjer 4.10: Funkcija *redukcija*.

```
def redukcija(niz, operacija):
    if len(niz) > 0:
        rezultat = niz[0]
        for e in niz[1:]:
            rezultat = operacija(rezultat, e)

        return rezultat
```

definirana u Pythonu), koja za dvije zadane vrijednosti vraća onu veću[1], rezultat bi odgovarao rezultatu izraza *max(max(max(max(14, 5), 20), 1), 9)*. Funkciju *redukcija* možemo definirati kako je pokazano u Primjeru 4.10. Ovu funkciju možemo, na primjer, koristiti da dobijemo zbroj svih brojeva u nizu:

```
>>> redukcija([1, 2, 3, 4], lambda a, b: a + b)
10
```

Funkcija *redukcija* radi tako da počne s prvim elementom kao rezultatom. Nakon toga taj element koristi kao argument za funkciju pridruženu parametru *operacija* zajedno s idućim elementom, nakon čega taj novi rezultat opet pridruži varijabli *rezultat*. Ovaj postupak ponavlja se sve dok ima elemenata u nizu. Vidimo da petlja koristi segment *niz[1:]* - ovaj izraz daje segment niza bez 0-tog elementa, to jest od drugog do posljednjeg, što nam ovdje treba jer smo prvi element već pridružili varijabli *rezultat*. Isto tako, na početku funkcije moramo provjeriti je li niz prazan. Ako je onda funkcija vraća *None*, što u ovoj funkciji nije napisano eksplicitno. Kao što je već rečeno, ako jedan tok izvršavanja naredbi neke funkcije dođe do kraja, a posljednja naredba nije *return*, onda funkcija vraća *None* kao rezultat.

U ovom smo primjeru koristili funkcijsku vrijednost `lambda a, b: a + b` da bismo dobili zbroj svih brojeva u nizu. Rezultat odgovara izrazu `+(+(+(1, 2), 3), 4)` gdje je operator "+" prikazan kao funkcija. Kao i druge funkcije koje smo na ovaj način definirali, funkciju *redukcija* možemo koristiti gdje god nam treba rezultat izraza formuliranog na gore opisan način. Na primjer, najveći element niza sada možemo dobiti ovim izrazom:

```
>>> redukcija([14, 5, 20, 1, 9], max)
20
```

Ovim primjerom možemo pratiti kako *redukcija* radi. Prvo se broj 14 pridruži varijabli *rezultat*, odnosno početni rezultat je 14. Ako sada zamijenimo izraz *operacija(rezultat, e)* sa *max(rezultat, e)* imamo sljedeći niz koraka:

```
1. rezultat = niz[0] = 14
2. e = 5, rezultat = max(14, 5), rezultat je 14
```

[1]Funkcija *max* može primiti i više od dvije vrijednosti kao argumente ili listu elemenata. Na isti način funkcija *min* vraća najmanji od zadanih elemenata.

```
3. e = 20, rezultat = max(14, 20), rezultat je 20
4. e = 1, rezultat = max(20, 1), rezultat je 20
5. e = 9, rezultat = max(20, 9), rezultat je 20
```

Korištenje funkcija kao što su *transformacija*, *selekcija*, *redukcija* i *prvi* ima nekoliko prednosti:

1. Omogućavaju nam izdvojiti osnovni postupak kojim rade mnoge funkcije i koji se kao takav često ponavlja u programiranju na jedno mjesto.

2. Kod definicije takvih funkcija možemo ignorirati određene detalje i usredotočiti se na jedno opće načelo po kojem funkcija radi. Detalji se nalaze u funkciji koju dajemo kao argument.

3. Ovakve nam funkcije omogućavaju izradu jednog postupka od dviju ili više drugih, međusobno nepovezanih funkcija. Na primjer, funkcija *redukcija* i lambda izraz za zabrajanje dvaju brojeva međusobno su nepovezani, ali smo ih izrazom `redukcija([1, 2, 3, 4], lambda a, b: a + b)` povezali u jedan cjeloviti postupak koji kao rezultat daje zbroj svih brojeva u nizu. Taj rezultat ne možemo dobiti samo funkcijom koja zbraja dva broja jer ona radi sa samo dvije vrijednosti. Isto ga tako ne možemo dobiti funkcijom *redukcija* jer ona ne radi ništa specifično, kao što je zbrajanje svih brojeva u nizu. Međutim, kombinacijom ovih dviju nepovezanih funkcija dobili smo željeni rezultat. Funkcija *redukcija* omogućila je osnovni postupak rada, a funkcija *lambda a, b: a + b* davala je specifične rezultate kojima smo tim postupkom dobili konačni rezultat.

Za kraj, ovako bismo mogli opisati načelo rada funkcije *transformacija*: *za svaki element E na indeksu I ulaznog niza i za neku funkciju F postavi rezultat izraza F(E) na indeks I izlaznog niza.*

Ovom definicijom ne uvodimo nikakve pretpostavke o tome što su elementi ulaznog i izlaznog niza, nego samo kako su oni strukturirani. Na taj način, umjesto da definiramo više funkcija koje sve rade na isti način, možemo definirati jednu funkciju koja služi samo kao jedan općeniti postupak na koji pomoću parametra pridodamo drugu funkciju koja vraća specifičnu funkcionalnost tom postupku i od koje zavisi konačni rezultat. Taj postupak određuje kako nešto radi, a funkcija u kombinaciji s njim određuje točno što on radi. Na primjer, kod funkcije *redukcija* čiji je argument bila funkcija *lambda a, b: a + b*, *redukcija* određuje kako taj postupak radi, a *lambda a, b: a + b* određuje što radi, to jest što će biti rezultat.

Općenito, funkcije višeg reda mogu se kombinirati s drugim funkcijama na puno načina, od kojih ne mora svaki biti jednostavna kombinacija kao što su funkcije koje smo prethodno vidjeli. Upotreba funkcija višeg reda vrlo je korisna tehnika programiranja kojom možemo izbjeći dupliciranje koda, ali isto tako i poboljšati njegovu strukturu, fleksibilnost i razumljivost.

4.2 Funkcijske vrijednosti

Naredba *def* pridružuje funkciju varijabli koja služi kao ime te funkcije. Na primjer,

```
def kvadrat(n):
    return n * n
```

pridružuje varijabli *kvadrat* funkciju koja ima jedan parametar i vraća rezultat izraza *n * n*. Što znači pridružiti funkciju nekoj varijabli? Do sada smo uvijek govorili o pozivu neke funkcije, gdje rezultat tog poziva možemo pridružiti nekoj varijabli. Samu funkciju, međutim, možemo tretirati kao vrijednost, na isti način na koji tretiramo neki broj, znakovni niz, listu i ostale vrijednosti. Funkciju, jednako kao brojeve, znakovne nizove i liste, možemo promatrati kao vrijednost koju možemo pridružiti nekoj varijabli, vratiti kao rezultat poziva neke funkcije ili dati kao argument pri pozivu neke funkcije. Na primjer, sljedeće pridruživanje varijabli f pridružuje funkciju koja je pridružena varijabli *kvadrat*:

```
>>> f = kvadrat

>>> f
<function kvadrat at 0x000001AA7AABC1E0>

>>> kvadrat
<function kvadrat at 0x000001AA7AABC1E0>
```

Sada tu funkciju možemo pozvati preko varijable f:

```
>>> f(9)
81
```

Ovdje nema nikakve razlike između gornjeg poziva i izravnog poziva (to jest, poziva ove funkcije preko varijable *kvadrat*):

```
>>> kvadrat(9)
81
```

> Funkcijska vrijednost predstavlja funkciju.

Prema tome, funkcijska vrijednost predstavlja funkciju. Važno je razlikovati vrijednost koju neka funkcija vraća pri pozivu i vrijednost koja predstavlja samu funkciju. U gornjim primjerima, rezultat izraza *kvadrat* je funkcija (funkcijska vrijednost), a rezultat izraza *kvadrat(9)* je broj koji ta funkcija pri pozivu vraća.

Primjer Napišite program koji u nizu slova pronalazi najdulji podniz koji se sastoji samo od samoglasnika (*a*, *e*, *i*, *o* i/ili *u*).

Rješenje: Zadatak možemo riješiti u tri koraka:

1. Transformiraj ulazni znakovni niz tako da znakove koji nisu samoglasnici zamijenimo jednim znakom, recimo '*'.

Primjer 4.11: Funkcija *samoglasnici*.

```
def samoglasnici(s):
    p = map(lambda x: x if x in 'aeiou' else '*', s)
    r = ''.join(p)
    t = r.split('*')
    return max(t, key=lambda s: len(s))
```

2. Rastavi znakovni niz dobiven u koraku 1 na mjestima gdje se nalazi znak '*' čime dobivamo listu podnizova koji sadrže samo samoglasnike ili prazan znakovni niz ("").

3. Nađi najdulji znakovni niz u listi iz koraka 3, što je ujedno rezultat.

Primjena ovog postupka pokazana je u Primjeru 4.11. Funkcija *map* radi isto što i funkcija *transformacija* iz prethodnog dijela.

5 Karakteristike lista i rječnika

5.1 Karakteristike lista

Indeksiranje liste učinkovita je operacija. To znači da vrijeme koje je potrebno računalu da dođe do elementa liste na zadanom indeksu ne ovisi o

- broju elemenata u listi
- veličini zadanog indeksa.

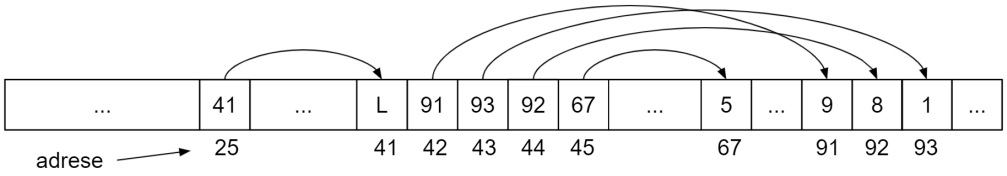
U ovom dijelu opisane su karakteristike liste kao strukture podataka i način na koji organizacija elemenata u njoj utječe na performanse operacija nad listama.

Neka je varijabli p pridružena sljedeća lista:

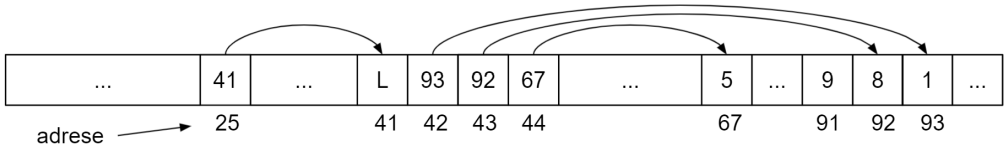
```
1 p = [9, 1, 8, 5]
```

Kada želimo doći do elementa na nekom indeksu program ne mora "pronaći" gdje se taj element nalazi nego samo izračunati njegovu lokaciju. Na Slici 5.1 ilustriran je sadržaj memorije s fiktivnim adresama nakon izvršenja gornje naredbe pridruživanja. Na adresi 25 nalazi se varijabla p koja sadrži adresu objekta koji predstavlja listu pridruženu toj varijabli. Taj se objekt nalazi na adresi 41 (s obzirom da objekt liste ima i neke druge podatke osim samih elemenata, na adresi 41 naznačen je početak tog objekta sa slovom L). Kao što je slučaj s varijablama, elementi liste adrese su na kojima se nalaze objekti koji su u nju smješteni. Na slici, dakle, element na indeksu 0 nalazi se na adresi 42, onaj na indeksu 1 je u memoriji smješten odmah iza njega, na adresi 43, idući, na indeksu 2, odmah je iza njega na adresi 44 i posljednji, na indeksu 3, na adresi 45. Indeksi liste su, dakle, kao varijable. Ovdje treba primjetiti da same vrijednosti liste, to jest brojevi 9, 1, 8 i 5, ne moraju u tom redoslijedu biti smješteni u memoriji. Međutim, ovdje treba opet istaknuti, elementi liste su *adrese* vrijednosti, ne same vrijednosti. Prema tome, te adrese moraju biti smještene u poretku u kojem se vrijednosti na tim adresama pojavljuju u listi. Ako pratimo strelice na Slici 5.1 prvi element je na adresi 91 koja sadrži vrijednost 9, drugi je na adresi 93 koja sadrži 1, treći na adresi 92 koja sadrži vrijednost 8 i posljednji je na adresi 67 koja sadrži vrijednost 5. Vidimo, dakle, da je poredak elemenata liste, to jest adresa, takav da odgovara poretku vrijednosti u njoj.

Neka je sada zadan sljedeći izraz:



Slika 5.1: Lista s četiri elementa pridružena varijabli na adresi 25.



Slika 5.2: Lista sa slike 5.1 nakon uklanjanja elementa na indeksu 0.

```
p[2]
```

Prvi element liste (onaj na indeksu 0) nalazi se na adresi 42. Python će izračunati adresu na kojoj se nalazi element na indeksu i formulom *<adresa prvog elementa> + <i>*. Element na indeksu 2 se, dakle, nalazi na adresi $42 + 2 = 44$. Na slici ta adresa sadrži vrijednost 92, što je adresa na kojoj se nalazi vrijednost 8 pa je rezultat gornjeg izraza 8.

Iz ovoga vidimo da broj elemenata u listi ne utječe na brzinu pristupa tim elementima putem indeksa. Isto tako jasno je da ne ovisi niti o veličini zadanog indeksa. Razlog ovoj učinkovitosti indeksiranja liste je u organizaciji elemenata: oni su u memoriji "poredani" jedan iza drugoga, odnosno na susjednim memorijskim adresama u poretku u kojem su dodani u listu. Bez obzira na to želimo li pristupiti elementu na indeksu 1 ili 1000000 način je uvijek isti: *<adresa elementa> = <adresa početnog elementa> + <indeks>*. To je isto kao da želimo doći do stranice 100 u knjizi: s obzirom da su one poslagane u rastućem redoslijedu moramo samo znati gdje je stranica 1, odnosno početak knjige.

> Pythonova lista učinkovita je struktura podataka za brzi pristup elementima putem indeksa.

Činjenica da su elementi liste u memoriji računala smješteni jedan iza drugoga u poretku u kojem su dodani u listu, kao što smo vidjeli, ima prednosti kada se elementima pristupa putem indeksa. Međutim, kao i sve ostalo, ovakva organizacija elemenata ima i nedostatke. Pretpostavimo da želimo iz liste ukloniti element na indeksu 0 naredbom

```
del p[0]
```

Na primjeru na Slici 5.1 to je element na adresi 42. Jasno je da taj element ne možemo jednostavno ignorirati jer on se i dalje fizički nalazi na indeksu 0. Da bismo ga uklonili, prema tome, moramo pomaknuti sve elemente koji se nalaze desno od njega za jedno mjesto ulijevo. Tada bismo dobili sadržaj memorije pokazan na Slici 5.2.

Vidimo da je sada element na indeksu 0 broj 1 odnosno adresa 93 na kojoj se nalazi vrijednost 1. Ovisi li učinkovitost brisanja elementa liste o broju elemenata u listi i indeksu kojeg brišemo? Za razliku od indeksiranja liste, brisanje elemenata iz nje složeniji je postupak. Broj elemenata liste i indeks s kojeg uklanjamo element su povezani. Kao što smo vidjeli, ako uklanjamo element na početku liste onda svi elementi iza njega (na većim indeksima) moraju biti pomaknuti za jedan indeks niže. Iako moderne arhitekture procesora mogu učinkovito kopirati sadržaj memorije s jednog mjesta na drugo, vrijeme potrebno za tu operaciju i dalje ovisi o broju elemenata [18]. Međutim, što ako uklanjamo element koji se nalazi blizu ili na kraju liste? U tom slučaju bit će manje elemenata koje treba pomaknuti (ili niti jedan ako uklanjamo posljednji element liste). Isto važi i za dodavanje novih elemenata samo što se tada elementi na višim indeksima moraju pomaknuti da bi se dobio prostor za novi element. Isto tako, ako se novi element dodaje na kraj dodavanje je učinkovito. Općenito, važi sljedeće pravilo:

> Za dodavanje ili brisanje elemenata koji nisu na kraju liste Pythonova lista **nije** učinkovita struktura podataka.

5.1.1 Stog kao lista

Pythonova lista može poslužiti i kao struktura podataka koja se zove *stog* (engl. *stack*) [26]. Stog je struktura podataka koja se sastoji od niza elemenata i za koju su definirane dvije operacije:

- *push e*: dodaj element *e* na vrh stoga
- *pop*: ukloni i vrati element s vrha stoga.

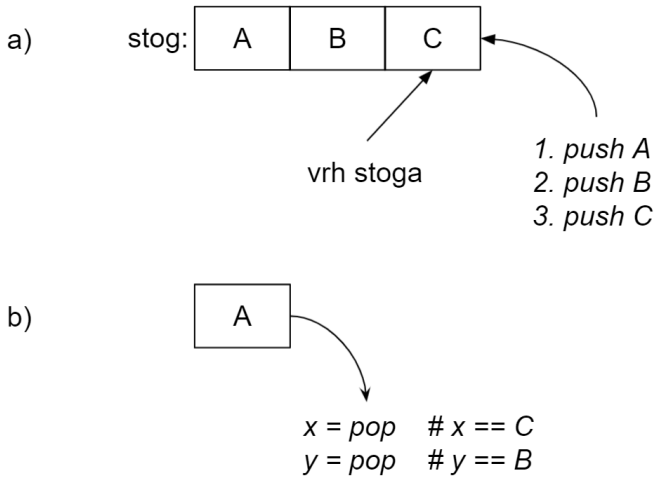
Obje ove operacije rade samo s elementima na vrhu stoga - nema dodavanja ili uklanjanja elemenata unutar stoga. Na Slici 5.3 ilustrirane su operacije sa stogom. Zbog načina rada ovih operacija stog se zove LIFO (engl. *Last In First Out*, posljednji unutra, prvi van) struktura podataka.

U Pythonu ne postoji poseban tip podataka za stog jer se stog može lako simulirati listom. U kodu u Primjeru 5.1 ilustrirane su operacije sa Slike 5.3.

Operacije nad stogom primjenom lista u Pythonu su učinkovite jer se elementi dodaju i uklanjaju samo s kraja liste pa zbog toga nema potrebe pomicati postojeće elemente.

> Za dodavanje ili brisanje elemenata koji se nalaze na kraju liste, Pythonova lista učinkovita je struktura podataka.

Stog kao struktura podataka ima veliku primjenu u računarstvu kao što je implementacija programskih jezika, sintaksna analiza, algoritmi i strukture podataka.



Slika 5.3: Struktura podataka *stog*. Na slici a) prikazan je stog nakon tri *push* operacije u prikazanom redoslijedu. Elementi se na stog dodaju samo na vrhu. Na slici b) prikazan je isti stog nakon dvije *pop* operacije u prikazanom redoslijedu. Elementi sa stoga također uzimaju se samo s vrha.

Primjer 5.1: Primjer operacija nad stogom.

```
>>> stog = []  # prazan stog
>>> stog.append('A')
>>> stog.append('B')
>>> stog.append('C')
>>> x = stog.pop()
>>> y = stog.pop()
>>> x
'C'
>>> y
'B'
```

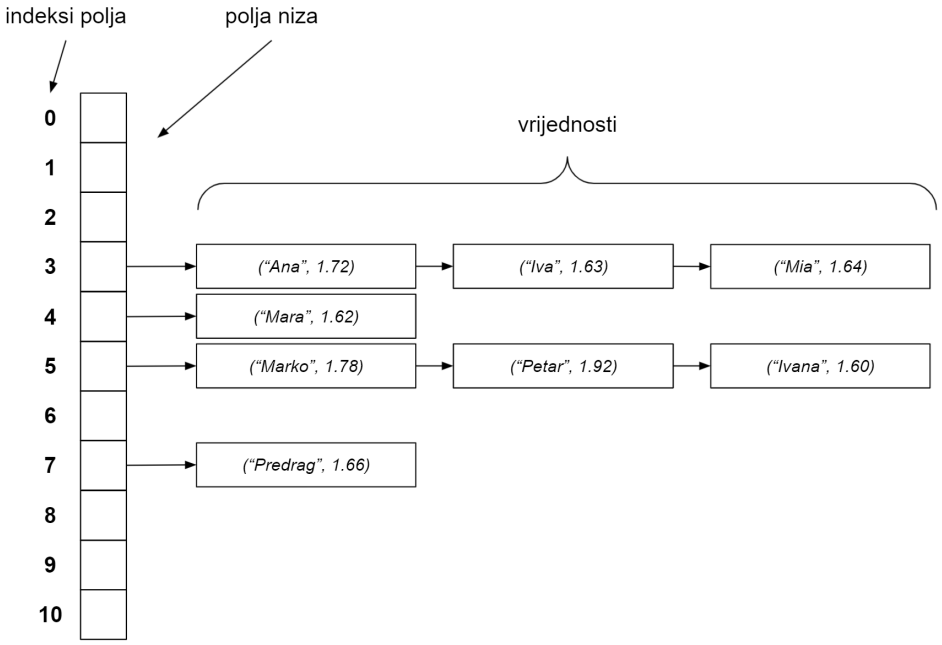
5.2 Karakteristike rječnika

Kao i Pythonove liste, rječnici su učinkoviti za neke operacije dok za druge nisu ili ih uopće ne podržavaju.

Rječnici su implementirani kao podatkovna struktura zvana *hash* tablica (engl. *hash table*) [26]. To je struktura podataka koju možemo zamisliti kao niz parova kod kojih je prvi element para ključ, a drugi jedna ili više vrijednosti. Na Slici 5.4 skicirana je ova struktura podataka.

Pretpostavimo da radimo s podacima o osobama i njihovom visinom. Na primjer, Ana je visoka 1.72m, Predrag 1.66m, Petar 1.92m itd. Takve podatke možemo lako spremiti u rječnik:

```
>>> r = {'Ana': 1.72, 'Predrag': 1.66, 'Petar': 1.92, 'Iva': 1.63,
```



Slika 5.4: Primjer strukture hash tablice.

```
          'Mara': 1.62, 'Ivana': 1.6, 'Marko': 1.78, 'Mia': 1.64}
```

Dakle, ime osobe je ključ, a visina je vrijednost pridružena tom ključu:

```
>>> r['Petar']
1.92
```

Iako ovaj izraz izgleda kao indeksiranje niza, postupak kojim Python dolazi do vrijednosti potpuno je drugačiji od onog s nizovima. Kao i kod indeksiranja niza, niti ovdje Python ne traži zadani ključ u smislu da prolazi ključeve jedan po jedan dok ne dođe do onog koji traži. Na Slici 5.4 vidimo da je pojedinim poljima niza s lijeve strane pridružena jedna ili više vrijednosti prikazane kao par koji se sastoji od ključa kao prvog elementa i vrijednosti kao drugog. Jedan način na koji bismo mogli doći do vrijednosti s ključem *Petar* (ili bilo kojim drugim ključem) je taj da pretražujemo sve ove parove, počevši od nekog "prvog", dok ne dođemo do onog s odgovarajućim ključem. Međutim, takav bi pristup bio krajnje neučinkovit. Rječnici postoje upravo zato da bi pristup vrijednostima bio brz, gotovo jednako brz kao pristup elementima niza preko indeksa. Osnovna ideja je ta da se, umjesto traženja, na osnovu zadanog ključa izračuna na kojem se indeksu niza nalazi njegova vrijednost. Na primjer, na osnovu ključa "Predrag" dobije se indeks 7 čime bismo odmah dobili njegov par s vrijednošću 1.66 umjesto ispitivanja svih osam parova. To znači da ključ moramo svesti na broj između 0 i N, gdje je N veličina hash tablice. Iako ključ rječnika ne mora biti isključivo znakovni niz, ovdje ćemo u svrhu ilustracije pretpostaviti da je ključ tipa *str*.

Postoji puno načina da se neki znakovni niz svede na broj na osnovu nekih njegovih karakteristika. Izračunavanje vrijednosti ključa izvodi se korištenjem *funkcije sažimanja*

Primjer 5.2: Implementacija jedne funkcije sažimanja za znakovne nizove.

```
def str_hash(k, velicina_tablice):
    h = 7
    for c in k:
        h = h * 31 + ord(c)

    return h % velicina_tablice
```

(engl. *hash function*). Za znakovne nizove jedna od najjednostavnijih funkcija sažimanja je ona koja vraća duljinu znakovnog niza. Ovakvu funkciju sažimanja u Pythonu jednostavno je definirati:

```
def hash(k):
    return len(k)
```

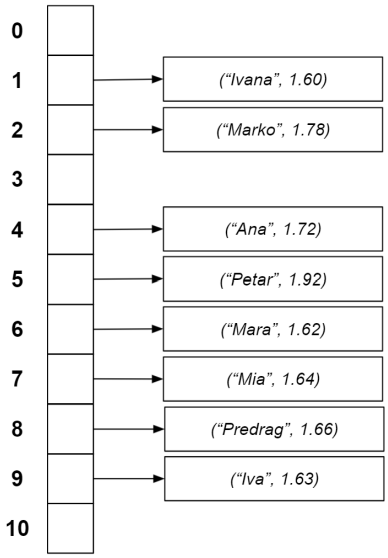
Na Slici 5.4 možemo primjetiti da indeks zauzetih polja niza odgovara duljini ključa; ”Ana”, ”Iva” i ”Mia” su na indeksu 3, ”Mara” na indeksu 4, ”Petar”, ”Marko” i ”Ivana” na indeksu 5, a ”Predrag” na indeksu 7. Ovaj pristup, međutim, nije idealan jer neka polja, specifično 3 i 5, sadrže više od jedne vrijednosti. U kontekstu hash tablica ovo se zove *kolizija* [26]. U našem primjeru, ako želimo dobiti vrijednost pridruženu ključu ”Ivana” ova hash funkcija za taj ključ vraća 5. Međutim, na indeksu 5 ne nalazi se samo jedna vrijednost nego njih tri pa bi se stoga morala ispitati svaki od ta tri para i vratiti vrijednost onog koji sadrži zadani ključ. To bi pretraživanje znatno usporilo pristup vrijednostima hash tablice pa zbog toga ovakva hash-funkcija nije prihvatljiva, barem ne za veće količine podataka.

Jedna dobra funkcija sažimanja za znakovne nizove prikazana je u Primjeru 5.2. Funkcija *ord* vraća ASCII kôd zadanog znaka. U posljednjem redu uzimamo ostatak dijeljenja (modulo) vrijednosti h i veličine tablice tako da nam rezultat bude između 0 i najvećeg indeksa tablice. Sada za ključeve sa Slike 5.4 možemo testirati funkciju *str_hash*:

```
n = 11
print('Iva:', str_hash('Iva', n))
print('Ana:', str_hash('Ana', n))
print('Mia:', str_hash('Mia', n))
print('Mara', str_hash('Mara', n))
print('Petar', str_hash('Petar', n))
print('Marko', str_hash('Marko', n))
print('Ivana:', str_hash('Ivana', n))
print('Predrag:', str_hash('Predrag', n))
```

Ispis izgleda ovako:

```
Iva: 9
Ana: 4
Mia: 7
Mara 6
```



Slika 5.5: Hash tablica sa Slike 5.4, ali bez kolizija.

```
Petar 5
Marko 2
Ivana: 1
Predrag: 8
```

Vidimo da sada nema kolizija nego hash tablica izgleda kao na Slici 5.5. Općenito, idealna funkcija sažimanja je ona kod koje nema kolizija. Kada bismo u ovu hash tablicu unijeli više od 11 vrijednosti jasno je da bi se stvorile kolizije. To bismo riješili povećanjem same hash tablice.

U ovom dijelu vidjeli smo načelo po kojem u Pythonu i drugim programskim jezicima funkcioniraju rječnici. Rječnici imaju nekoliko važnih karakteristika.

Rječnici omogućavaju brz pristup vrijednostima preko ključa pod uvjetom da je za tip podatka ključa definirana dobra hash funkcija.

Ovo je možda jedna od najvažnijih činjenica vezanih za rječnike. Ako je funkcija sažimanja loša onda pristup podacima može biti neučinkovit. U tom slučaju je bolje koristiti neku drugu strukturu podataka. Koristeći dobru funkciju sažimanja pristup podacima rječnika uglavnom je učinkovitiji nego kod bilo koje druge strukture podataka.

S obzirom da je rječnik neuređena struktura podataka, ne podržava operaciju indeksiranja, odnosno elementi nemaju unaprijed poznat redoslijed.

Ključevi rječnika smješteni su u hash tablicu na osnovu funkcije sažimanja pa je zbog toga njihov redoslijed nedefiniran. Prema tome, operacija indeksiranja za rječnike

nije definirana.

Dodavanje i brisanje elemenata iz rječnika učinkovita je operacija.

Za razliku od nizova, dodavanje ili brisanje elemenata kod rječnika ne zahtijeva nikakvo pomicanje elemenata pa su stoga te dvije operacije učinkovite.

6 Struktura programa

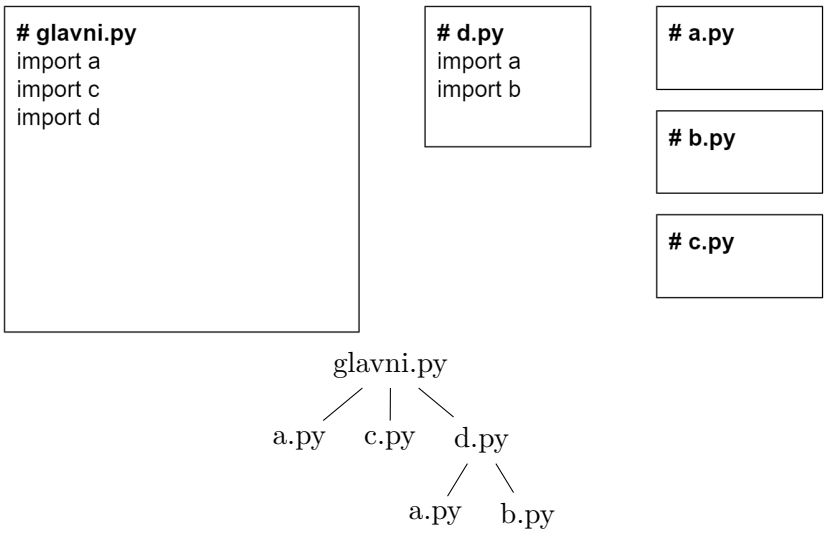
Većina malih programa kakve smo vidjeli do sada i kakve ćemo prikazati u ovom tekstu mogu se napisati u jednoj *.py* datoteci. Kod većih programa, međutim, takav pristup ne bi bio praktičan iz nekoliko razloga:

- Veći programi (nekoliko tisuća redova koda) sastoje se od dijelova koda koji se upotrebljavaju na više mjesta u programu. Primjerice, modul *random* i njegovu funkciju `randint` možemo pozivati iz više mjesta u nekoj aplikaciji gdje je takva funkcija potrebna. Modulima se, dakle, omogućava ponovna upotreba koda (engl. *code reuse*).
- Veći programi obično se sastoje od dijelova koji sačinjavaju jednu logičku cjelinu. Na primjer, aplikacija za grafički prikaz podataka imat će dijelove koda koji su predviđeni za crtanje grafičkih dijagrama i one za unos podataka. Dio za unos podataka može se dalje podijeliti na onaj koji podatke učitava iz datoteke i onaj koji ih učitava s web servisa. Svaki od tih dijelova sačinjava jednu zasebnu logičku cjelinu i takvu aplikaciju lakše je razumjeti ako je ona fizički strukturirana tako da su te cjeline zasebno napisane.
- Veće programe lakše je testirati ako su podijeljeni u više logičkih cjelina zato jer svaku takvu cjelinu možemo zasebno testirati. To se zove *unit-testing* i važno je načelo testiranja.
- Na većim programima obično radi više ljudi pa bi bilo nepraktično da svi rade na kodu koji je u jednoj *.py* datoteci. Ako bi više programera radilo istovremeno na jednoj *.py* datoteci kasnije bi bilo teško ukomponirati svaku promjenu u kodu svakog pojedinačnog programera.

Iz ovih razloga svi popularni programski jezici imaju konstrukte kojima se programski kôd može rastaviti na više datoteka koje se onda pridodaju u glavni program. Za to u Pythonu postoje *moduli*.

6.1 Moduli

Modul u Pythonu jedna je *.py* datoteka. Jedan modul može se naredbom *import* povezati s drugim tako da cijeli program ima hijerarhijsku strukturu kako je pokazano na Slici



Slika 6.1: Tipična struktura programa u Pythonu koja se sastoji od jednog glavnog modula i jednog ili više modula s kojima se on povezuje naredbom *import* ili *from..import...*

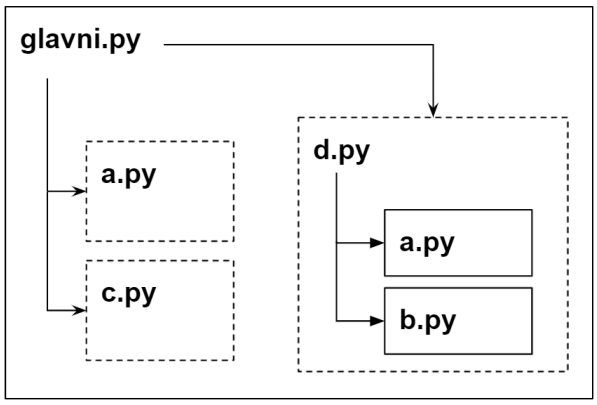
6.1. Svaki program u Pythonu sastoji se od jednog glavnog modula - to je modul koji sadrži kôd kojim se pokreće program. Na Slici 6.1 to je modul *glavni.py*. Vidimo da on koristi tri druga modula: *a*, *c* i *d*. Isto tako vidimo da modul *d* koristi dva druga modula: *a* i *b*. Prema tome, strukturu ovog programa u memoriji mogli bismo prikazati kao na Slici 6.2.

Naredba *import*

Unutar glavnog programa *glavni.py* modulima *a*, *c* i *d* možemo pristupati tako da navedemo ime modula. Na primjer, ako modul *d* ima funkciju *f* bez parametara tu funkciju možemo iz modula *glavni.py* pozvati tako da napišemo `d.f()`. Iako je modul *d* učitao modul *b* taj modul (*b.py*) nije dostupan u modulu *glavni.py* jer u glavnom modulu modul *b* nije eksplicitno učitan naredbom *import*. Na Slici 6.2 modul *glavni.py* ima pristup samo modulima koji su prikazani iscrtkanom crtom. Nadalje, vidimo da modul *a* koriste i modul *glavni* i modul *d* što je čest slučaj kod upotrebe modula standardnih biblioteka. Primjerice, modul s funkcijama za rad sa znakovnim nizovima, *string*, može biti učitan od strane više drugih modula jedne aplikacije.

Naredba *import d* u modulu *glavni* omogućava kodu modula *glavni* pristup svim *atributima* modula *d* koji su definirani na najvišoj razini tog modula. Pod atributom podrazumijevamo sve ono što ima ime (definicije funkcija, klasa i ostalih objekata). U naredbi *import* naziv modula ima dvostruku ulogu:

- označava ime *.py* datoteke
- uvodi novu varijablu tog naziva kojoj je pridružen učitani modul.



Slika 6.2: Unutrašnja struktura programa.

Na primjer, neka je modul *glavni* definiran ovako:

```
# modul "glavni.py"

import d

x = d.zbroj(a, b)
print(x)
```

Neka je modul *d* definiran ovako:

```
# modul "d.py"

import a
import b

print('modul "d"')

PI = 3.14

def zbroj(x, y):
    return x + y
```

Atributi modula *d* su *PI* i *zbroj*. Naredba *import* u modulu *glavni* učitava sve definicije modula *d* (*PI* i *zbroj*) u modul *glavni*, ali isto tako uvodi i novu varijablu *d* u modul *glavni*. Toj je varijabli pridružen modul *d* i pomoću nje možemo pristupati definicijama tog modula iz modula *glavni* upotrebom sintakse *<ime modula>.<atribut>*.

Kako radi *import* naredba

Naredba *import* izvodi se u tri koraka:

1. Pronađi datoteku navedenog modula.
2. Prevedi je na bajt-kôd ako je potrebno.

3. Izvrši kôd modula da bi se dobili njegovi atributi.

Ovi se koraci odvijaju samo za prvo učitavanje modula. Ako je modul već učitan (nije bitno koji ga je modul učitao), samo ga se dohvaća iz memorije i ovi se koraci preskaču. Moduli koji su već učitani mogu se naći u rječniku *sys.modules*:

```
>>> import sys
>>> import random
>>> sys.modules
{..., 'random': <module 'random' from 'C:\\...\\Python39\\lib\\random
    .py'>}
```

U ispisu možemo vidjeti (pored mnogih drugih) modul *random* koji smo uvezli i putanju njegove datoteke.

6.1.1 Pronalaženje modula

Da bi učitao modul, Python prvo mora locirati njegovu datoteku. Naredba *import* ne navodi mapa u kojem se ona nalazi. Zbog toga se Python oslanja na *standardnu putanju modula*. Ta se putanja sastoji od liste mapa dobivene iz sljedećeg:

1. polazišna mapa programa
2. mape sistemske varijable PYTHONPATH (ako je postavljena)
3. mape standardnih biblioteka
4. sadržaj *.pth* datoteka (ako postoje)
5. polazišna mapa od *site-packages* (mapa za nestandardne biblioteke).

Polazišna mapa programa zavisi od toga kako pokrećemo program. Ako ga pokrećemo iz glavnog modula onda se u ovoj mapi nalazi taj glavni modul. Ako radimo interaktivno ova mapa odgovara trenutnoj *radnoj mapi*, to jest onoj u kojoj se u tom trenutku nalazimo. Primjerice, ako smo Python pokrenuli iz mape *c:\dev\python* (na Windows sustavu) onda će to biti radna pa stoga i polazišna mapa.

PYTHONPATH mapa koja sadrži popis mapa (ako je definiran) Python pretražuje po redu, s lijeva na desno. Na primjer, ako je sadržaj ove sistemske varijable *c:\dev\app; c:\python\lib*, Python će prvo tražiti zadani modul u *c:\dev\app*, a onda u *c:\python\lib*. Ova se varijabla po potrebi može kreirati i modificirati ako želimo da Python traži module u mapama koji su u njoj navedeni. Nadalje, ova varijabla važna je samo u slučajevima kada želimo importirati module koji se nalaze u mapama drugačijima od onog u kojem se nalazi modul koji ih učitava. Ova je mogućnost korisna kada radimo veće programe.

Mape standardnih biblioteka Python pretražuje automatski pa se oni, prema tome, ne moraju dodavati u PYTHONPATH varijablu ili *.pth* datoteke.

Mape *.pth* datoteka manje je korištena mogućnost gdje mape za pretraživanje možemo dodati u tekstualnu datoteku, po jedan u svakom redu, čije ime završava sa *.pth*.

Ove datoteke imaju sličnu ulogu kao i varijabla PYTHONPATH. Na primjer, možemo kreirati datoteku *config.pth* i smjestiti je u najvišu mapu u kojoj se nalazi Python instalacija (primjerice, *c:\Python39*) i Python će dodati njene mape blizu kraja liste mapa za pretraživanje (iza sadržaja varijable PYTHONPATH). Detalji ove mogućnosti mogu se naći u standardnoj dokumentaciji Pythona.

***Site-packages* mapa** je mapa koju Python automatski doda u putanju za pretraživanje. To je obično mapa u koju se instaliraju nestandardne biblioteke i druga proširenja Pythona.

Prevođenje modula Nakon što je pronašao *.py* datoteku modula Python je, ako je potrebno, prevodi u *byte-code* instrukcije.[1] Ovaj korak radi na sljedeći način: Ako byte-code ne postoji, starija je od izvorne (*.py*) datoteke ili je kreirana drugačijom verzijom Pythona, Python će je regenerirati (ili kreirati) prilikom učitavanja. U suprotnom Python ne izvršava ovaj korak. Nadalje, ako postoji samo *.pyc* datoteka (bez *.py* datoteke) Python će je izravno učitati što znači da programe možemo isporučivati samo kao byte-code datoteke, bez izvornog koda. Preskakanjem ovog koraka ubrzava se izvršavanje programa.

Izvršavanje modula Posljednji korak učitavanja modula izvršenje je njegovog byte-koda. Kao i kod pokretanja glavnog modula, sve naredbe učitanog modula u ovom koraku bit će izvršene. Ako se vratimo na prethodni primjer modula *d* koji se učitava iz modula *glavni*, prvo će biti izvršena naredba *print* (pa će se zbog toga vidjeti ispis 'modul "d"' na ekranu), nakon toga će biti izvršeno pridruživanje varijabli *PI* te na kraju naredba *def* kojom će varijabli *zbroj* biti pridružen funkcijski objekt. Nakon toga atributi *PI* i *zbroj* bit će dostupni modulu *glavni*. Razumljivo je da modul po učitavanju mora biti izvršen (pokrenut), u suprotnom ovi atributi ne bi bili definirani. Naredba *import M*, dakle, učitava i izvršava kôd modula *M*.

6.2 Naredba *from .. import ..*

> Imenski prostor dio je koda koji sačinjava jednu cjelinu i kojem se pristupa preko naziva kojim je označen taj imenski prostor.

Na primjer, modul u Pythonu sačinjava jedan imenski prostor čije ime je ime tog modula. U Pythonu imenskim prostorom možemo smatrati i neke druge konstrukte kao što su klase (što je tema za kasnije).

U Pythonu postoji još jedan način učitavanja modula: naredba *from .. import ...* U sljedećem primjeru umjesto da učitamo cijeli modul *random* učitat ćemo samo njegovu funkciju *randint*:

[1] To su instrukcije na koje se prevodi izvorni kôd i koje izvršava Pythonov virtuelni stroj. Razlog iz kojeg se to radi je poboljšanje performansi.

```
# modul test.py
from random import randint

print(randint(0, 10))
```

Vidimo da u ovom slučaju ne pišemo `random.randint` nego samo `randint`. To je zbog toga što ime (varijabla) `random` nije definirana u modulu *test* jer nismo koristili naredbu `import random`. Naredbom `from random import randint` samo smo funkciju *randint* uvezli u modul *test*, ali ne i cijeli modul. Drugim riječima, funkciju *randint* dodali smo u imenski prostor modula *test* tako da je ona sada dostupna kodu modula *test* kao da je u njemu definirana.

Naredba *from M import ...* korisna je kada želimo neko ime iz modula M koristiti izravno, bez navođenja naziva modula. Međutim, ponekad je jasnije ako se navede i ime modula (to jest, ako se koristi oblik *import M*) jer se tada lakše vidi gdje je neka varijabla definirana. Na primjer, izraz `kalendar.danasnji_dan()` jasno pokazuje da je funkcija *danasnji_dan* definirana u modulu *kalendar* nego kada je poziv samo `danasnji_dan()`.

6.2.1 Naredba *from .. import **

Još jedan oblik naredbe *from .. import ..* je uvoženje svih atributa zadanog modula. Za neki modul *M* ta bi se naredba pisala `from M import *`. Neka je modul *Test* definiran na sljedeći način:

```
# modul test.py
from random import *

print(choice(['a', 'b', 'c', 'd']))
```

Naredbom `from random import *` unosimo sve atribute modula *random* u imenski prostor modula *test*. Primjerice, modul *random* ima funkciju *choice* koja za zadani niz elemenata vraća jedan od njih slučajnim odabirom. Iako tu funkciju nismo eksplicitno naveli u naredbi *import* ona je dio modula *random* pa je zbog toga bila postavljena u modul *test*. Općenito, naredbom *from M import ** koja se nalazi u nekom modulu *T* svi atributi modula *M* postaju atributi modula *T*. To znači da atributu *A* modula *M* tada nije potrebno pristupati s *M.A* unutar modula *T* nego samo s *A*.

7 Klase i objekti

Do sada smo govorili o vrijednostima kao što su brojevi, znakovni nizovi, liste i drugo. Svaka od tih vrijednosti pripada određenom *tipu podataka*. Primjerice, znakovni niz `"abc"` je tipa *str*, broj `25.8` je tipa *float*, broj `5` je tipa *int*, a lista `[2, 3, 1.1]` je tipa *list*. Preciznije, ovakve su vrijednosti *instance* dotičnog tipa. Instanca nekog tipa još se zove i *objekt* tog tipa.

> Vrijednosti koje su instance nekog tipa podataka nazivaju se objektima. Nadalje, objekt je nešto što zauzima prostor u memoriji računala.

Važno je razlikovati objekt od njegovog tipa. Objekt (znakovni niz „abc“ nije isto što i tip *str*. Tip je obrazac ili predložak na osnovu kojeg se može izraditi konkretan objekt tog tipa. Nadalje, za neki tip definirane su operacije (metode) koje su na raspolaganju za sve objekte tog tipa.[1] Na primjer, jedna od metoda za znakovne nizove je *index* ili operacija ”+” za spajanje znakovnih nizova. Međutim, operacija ”-” nije dio tipa *str* pa tako ta operacija nije primjenjiva nad objektima tipa *str*. Isto tako, za cijelobrojni tip *int* definirane su operacije ”+”, ”-”, ”*”, ”/” i druge dok, primjerice, taj tip nema metodu *index*.

> Objekt je cjelina koja objedinjuje podatke i operacije nad njima.

Jedna korisna strana pojma objekta je ta da o njima možemo razmišljati kao o jednoj cjelini. Primjerice, lista `[1, 2, 3]` sastoji se od triju elemenata, ali tu listu možemo pridružiti jednoj varijabli bez obzira na to od koliko se elemenata ona sastoji. Prema tome, o ovakvoj listi ne razmišljamo kao o tri broja nego kao o jednoj vrijednosti, to jest o jednom objektu.

> Svaka vrijednost u Pythonu je objekt.

U Pythonu svaka je vrijednost instanca nekog tipa, a time i objekt. To isto tako

[1] Pojmovi *metoda* i *operacija* označavaju jedan te isti koncept. Često se pod pojmom operacija smatraju operatori koji se pišu u takozvanoj *školskoj notaciji*, to jest *a+b* umjesto *+(a, b)*, dok bi za ovaj drugi oblik rekli da je ”+” funkcija. Kada kažemo operacija podrazumijevat ćemo aritmetičke i druge slične operacije, a za funkciju sve ostalo što se poziva.

znači da su za svaku vrijednost definirane i odgovarajuće operacije. Objekti nisu samo vrijednosti kao što su znakovni nizovi, brojevi, liste i drugi ugrađeni tipovi podataka. Kasnije ćemo vidjeti da su u Pythonu i datoteke objekti, kao i mnogi drugi koncepti s kojima radimo u programiranju.

U ovom dijelu vidjet ćemo kako programer može definirati nove tipove podataka definiranjem *klase*. Ovdje, međutim, moramo postaviti pitanje zašto bi nam trebao novi tip podatka. Ako pogledamo koje smo sve tipove podataka do sada koristili možemo odmah zaključiti i razlog iz kojeg smo ih koristili: svi ti tipovi podataka nudili su funkcionalnost (operacije) koja nam je za nešto trebala. Primjerice, objekti za datoteke imali su funkcije za čitanje i upisivanje u datoteku. Znakovni nizovi imali su funkcije za pristup pojedinačnim znakovima indeksiranjem, pronalaženje pojedinačnog znaka u znakovnom nizu, za spajanje znakovnih nizova i određivanje njihove duljine. Liste su imale funkcije za dodavanje novog elementa u listu, pronalaženje elemenata, spajanje dviju lista, brisanje elemenata i mnoge druge. Brojevi su imali aritmetičke operacije.

Python se isporučuje s velikim brojem tipova podataka kao dio njegove standardne biblioteke čija funkcionalnost obuhvaća mnoge potrebe u programiranju. Međutim, te potrebe programiranja su "generičke" u smislu da se dovoljno često javljaju u raznim domenama programiranja da ih je vrijedilo uključiti u Python. Primjerice, rad s JSON podacima danas je dovoljno zastupljen da većina programskih jezika ima podršku za JSON kao dio standardne biblioteke. Isto tako, spajanje na web servise dovoljno je česta potreba u programiranju da većina jezika ima biblioteke za to. Ovdje bismo mogli navesti i mnoga druga područja primjene kao što je šifriranje podataka, obrada teksta, numeričke funkcije, pristup operacijskom sustavu, serijalizacija podataka, komprimiranje i arhiviranje podataka, mrežno programiranje, pristup internetu i sl. Sve su ove funkcionalnosti implementirane kroz odgovarajuće tipove podataka. Zašto bi nam onda trebao novi tip podatka? Osnovni razlog leži u domeni za koju pišemo program. Pretpostavimo da radimo aplikaciju za hrvatsko Ministarstvo znanosti i obrazovanja koja se treba spojiti na ISVU (Informacijski sustav visokih učilišta) sustav. Bi li imalo smisla u Python uključiti nekakvu podršku za taj sustav kao dio njegove standardne biblioteke? Teško, jer ISVU sustav ima ograničenu primjenu na vrlo mali broj korisnika pa bi i takvo proširenje Pythonove standardne biblioteke najvećim dijelom bilo neiskorišteno i nepotrebno. Međutim, nekome tko želi razviti aplikaciju koja će se spajati na taj sustav itekako može trebati, recimo, tip *Predmet* koji u sebi sadrži operaciju *ekvivalenti* koja vraća listu šifri predmeta koji su ekvivalenti nekom zadanom predmetu. Ovakvih primjera, gdje nam treba neki tip podatka specifičan za određenu domenu, mogli bismo navesti bezbroj. Upravo zbog toga programski jezici kao što je Python omogućavaju programerima definiranje novih tipova podataka.

7.1 Klase

Klasa je konstrukt kojim se definira novi tip podatka. U Primjeru 7.1 definiran je tip podatka *Kalendar*. Taj tip podatka, međutim, nije osobito zanimljiv jer nema nikakve pripadajuće operacije (metode). Jedna korisna metoda za ovaj tip podatka bila

Primjer 7.1: Minimalna klasa *Kalendar*.

```
class Kalendar:
    pass  # ne radi nista
```

Primjer 7.2: Klasa *Kalendar* s jednom metodom, *danas*.

```
import datetime

class Kalendar:
    def danas(self):
        return datetime.date.today()
```

bi *danas* koja bi vraćala današnji datum. Naša klasa tada bi izgledala kako je prikazano u Primjeru 7.2.

Parametar *self* kod poziva ove metode sadržavat će objekt za koji je pozvana, ali detalje oko tog parametra proučit ćemo kasnije.

Funkcije koje su definirane unutar neke klase nazivaju se *metodama*. Rekli bismo, prema tome, da je *danas* metoda klase *Kalendar*.

Skup metoda neke klase sačinjava njeno *sučelje*.

Kao što za uređaje postoje razna sučelja, primjerice USB, tako i skup metoda neke klase možemo promatrati kao sučelje te klase koji sačinjava funkcionalnost njenih objekata koja je kroz njih na raspolaganju drugim dijelovima programa. Kada kažemo da dvije klase imaju drugačije sučelje to znači da se one razlikuju u jednoj ili više metoda i to u tome da te metode imaju dragačiji naziv i/ili drugačije parametre.

Još jedna logična funkcionalnost našeg kalendara bila bi ta da jednom datumu pridružimo vrijeme i tekst koji opisuje nešto što je korisniku važno za taj dan i vrijeme. Na primjer, za dan 17.4. i vrijeme 10:30 možemo postaviti tekst "sastanak". To možemo implementirati tako da napišemo novu metodu klase *Kalendar* koja će se zvati *dodaj_zapis* s parametrima za dan, vrijeme i tekst kao što je pokazano u Primjeru 7.3.

Kao prvo, metoda *dodaj_zapis* mora negdje spremiti podatke o danu, vremenu i tekstu koji su pridruženi parametrima ove metode tako da po izlasku iz nje ovi podaci ostanu sačuvani i dostupni. Kao drugo, moramo uzeti u obzir i sljedeće:

- način na koji će korisnik pretraživati podatke
- strukturu podataka koja će omogućiti pristup tim podacima.

Za pretraživanje podataka možemo pretpostaviti ono što je očito: podatke možemo pretraživati po danu, vremenu i tekstu. Usredotočit ćemo se na prva dva gledišta. Da bi korisnik pronašao aktivnosti za sutrašnji dan, treba unijeti taj dan kao *ključ* za

Primjer 7.3: Klasa *Kalendar* s metodom *dodaj_zapis*.

```
import datetime

class Kalendar:
    def danas(self):
        return datetime.date.today()

    def dodaj_zapis(self, dan, vrijeme, tekst):
        ...
```

pretraživanje, onda unutar tog dana možemo unijeti i vrijeme, također kao ključ za pretraživanje. Python već ima strukturu podataka pogodnu za ovakve slučajeve, a to je rječnik koji smo već vidjeli. Primjerice, ako u rječnik želimo spremiti dan 17.9.2020. i za taj dan dodati tekst "sastanak" za vrijeme 10:30 onda bismo to mogli izravno napisati ovako:

```
>>> m = {'2020-9-17': {'10:30': 'sastanak'}}
```

Ovo je rječnik koji kao ključ sadrži znakovni niz koji predstavlja datum, a kao vrijednost sadrži drugi rječnik. Taj drugi rječnik kao ključ sadrži znakovni niz koji predstavlja vrijeme, a kao vrijednost tekst. Ako tom rječniku pošaljemo upit što je rezervirano za dan '2020-9-17' dobit ćemo vrijednost pridruženu ključu '2020-9-17':

```
>>> m['2020-9-17']
{'10:30': 'sastanak'}
```

Sada za dan '2020-9-17' možemo dodati i druga vremena:

```
>>> m['2020-9-17']['13:00'] = 'pregled inventara'
>>> m
{'2020-9-17': {'10:30': 'sastanak',
               '13:00': 'pregled inventara'}}
```

Sljedeća naredba ne bi imala isti učinak:

```
m['2020-9-17'] = {'13:00': 'pregled inventara'}
```

Razlog je u tome što bismo tada izgubili prethodno pridruženu vrijednost ključu '2020-9-17', odnosno vrijednost *{'10:30': 'sastanak'}* zamijenili bismo vrijednošću *{'13:00': 'pregled inventara'}*. To možemo objasniti na sljedeći način: izraz `m['2020-9-17']['13:00']` možemo rastaviti na dva dijela:

```
x = m['2020-9-17']
x['13:00'] = 'pregled inventara'
```

Izraz *m['2020-9-17']* kao rezultat daje vrijednost pridruženu ključu '2020-9-17', a to je rječnik *{'10:30': 'sastanak'}*. Drugi dio ovog izraza tom rječniku dodaje novi ključ, "13:00", kojem pridružuje vrijednost "pregled inventara".

Primjer 7.4: Klasa *Kalendar* s implementacijom metode *dodaj_zapis*.

```
import datetime

class Kalendar:
    def __init__(self):
        self.zapis = {}

    def danas(self):
        return datetime.date.today()

    def dodaj_zapis(self, dan, vrijeme, tekst):
        if dan in self.zapis:
            self.zapis[dan][vrijeme] = tekst
        else:
            self.zapis[dan] = {vrijeme: tekst}
```

Sada ovo načelo možemo implementirati u metodi *dodaj_zapis* kako je prikazano u Primjeru 7.4.

Metoda *dodaj_zapis* sastoji se od jedne naredbe *if*. U toj naredbi prvo provjerimo postoji li ključ u rječniku za zadani dan. Ako postoji tek onda za taj ključ možemo postaviti tekst za rječnik koja je vrijednost tog ključa, odnosno za koju je ključ *vrijeme*. Ako ne postoji, za ključ *dan* moramo postaviti novi rječnik s jednim ključem, *vrijeme* i vrijednošću *tekst*.

Metoda `__init__` služi za inicijalizaciju objekta i zove se *konstruktor*.

Ovdje vidimo još jednu novu metodu, *___init___*. Unutar te metode varijabli *zapis* pridružuje se prazan rječnik. Ta će varijabla sadržavati rječnik u koji spremamo podatke o kalendaru na prethodno opisani način. Metoda *___init___* služi za *inicijalizaciju objekta* i naziva se *konstruktorom*. Suprotno onome što taj naziv implicira, konstruktor ništa ne konstruira ili kreira - on isključivo služi za inicijalizaciju novostvorenog objekta. Varijable objekta kao što je *zapis* nazivaju se *poljima* ili *atributima*. Konstruktorom najčešće inicijaliziramo polja objekta iako on može sadržavati bilo kakav kôd. Primjerice, konstruktori često sadrže kôd za validaciju parametera predanih pri instanciranju klase. Iako validacija parametara tehnički nije inicijalizacija, može se smatrati jednim aspektom inicijalizacije i često je njen dio.

Rezultat instanciranja klase je *objekt* te klase. Objekti klase nazivaju se i *instancama* te klase. Svaka klasa može imati više instanci, a svaka instanca ima svoju kopiju podataka.

Osnovna svrha klase omogućavanje je kreiranja objekata koji imaju zajedničke karakteristike određene poljima i sučeljem te klase. To sučelje specificira koje su operacije

na raspolaganju za objekte te klase. Na primjer, sa svakim znakovnim nizom možemo raditi isto: dohvatiti njegovu duljinu, dohvatiti indeks na kojem se nalazi traženi znak, izdvojiti jedan njegov dio i slično. Primjerice, za znakovni niz *abcde* pridružen varijabli *s* možemo izrazom `s.find('c')` dohvatiti indeks znaka *c*. Iako u datom trenutku izvršavanja programa može postojati velik broj znakovnih nizova, metoda *find* radi samo s podacima objekta *s*. Taj objekt sadrži niz znakova *abcde*, neovisno o tome što u tom trenutku izvršavanja programa sadrže drugi objekti ovog tipa, to jest tipa *str*. To je svrha instanciranja klase - da dobijemo objekt koji sadrži vlastite podatke i koji ima metode koje su dio sučelja te klase, kao što je metoda *find* dio sučelja klase *str*.

Vratimo se sada na klasu *Kalendar* i njenu metodu *dodaj_zapis*. Namjena te metode je da sa zadanim danom i vremenom poveže zadani tekst. Ali za koji kalendar? Rekli smo da svaka klasa može imati više instanci pa prema tome možemo napraviti više objekata klase *Kalendar* odnosno više kalendara. Primjerice, možemo imati poslovni kalendar i osobni kalendar ili u nekoj aplikaciji možemo voditi evidenciju o više osoba gdje svaka od njih ima svoj kalendar. Pogledajmo pobliže metodu *dodaj_zapis*:

```
class Kalendar:
    ...

    def dodaj_zapis(self, dan, vrijeme, tekst):
        if dan in self.zapis:
            self.zapis[dan][vrijeme] = tekst
        else:
            self.zapis[dan] = {vrijeme: tekst}
```

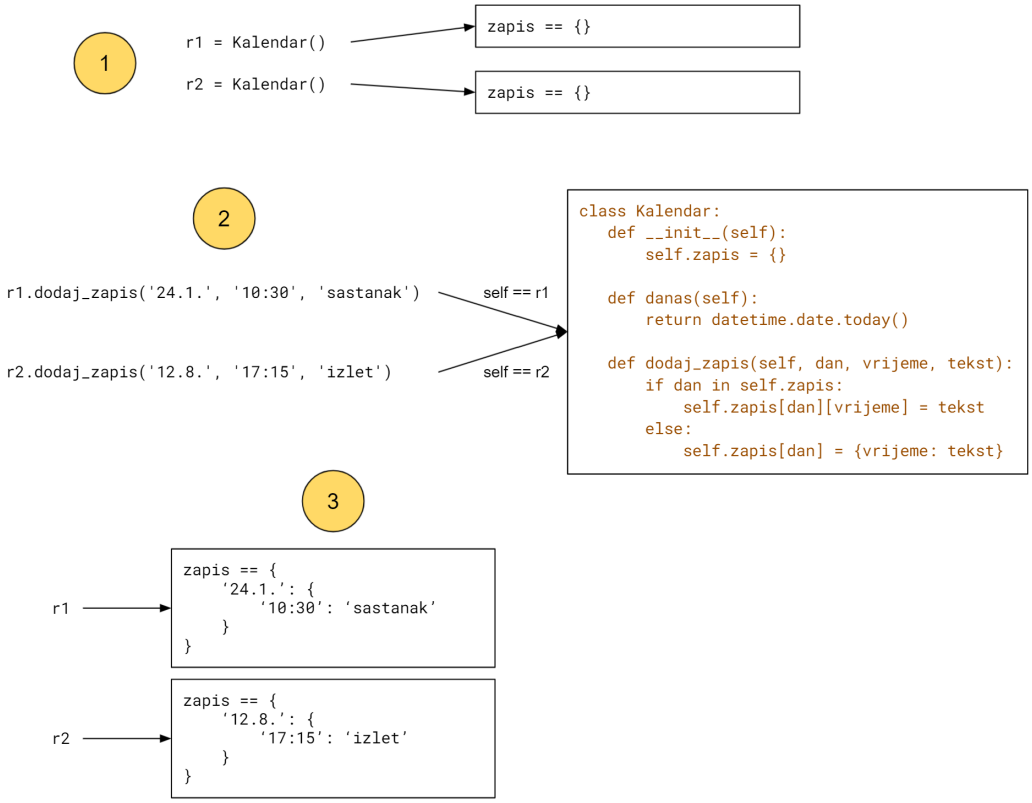
Možemo vidjeti da polju *zapis* pristupamo preko parametra *self*. To je neophodno jer želimo podatke unijeti u kalendar koji je dio instance ove klase, a ta je instanca pridružena parametru *self*. Da bi ovo bilo jasnije instancirat ćemo klasu *Kalendar*:

```
r1 = Kalendar()
r2 = Kalendar()
```

Varijable *r1* i *r2* će nakon izvršenja ovih dviju naredbi svaka sadržavati jednu instancu klase *Kalendar*, to jest jedan objekt te klase. Sada metodu *dodaj_zapis* možemo pozvati na ovaj način:

```
r1.dodaj_zapis('24.1.', '10:30', 'sastanak')
r2.dodaj_zapis('12.8.', '17:15', 'izlet')
```

Ovaj postupak prikazan je na Slici 7.1. U prvom koraku instancirana je klasa *Kalendar* dva puta. Nakon toga svakoj od varijabli *r1* i *r2* pridružen je inicijalizirani objekt. To znači da je bila izvršena metoda *___init___*, a ona se poziva automatski kao dio postupka instanciranja klase. U drugom koraku pozivaju se metode *dodaj_zapis*, jednom za objekt *r1* i jednom za *r2*. Kod svakog tog poziva parametru *self* metode *dodaj_zapis* bit će pridružen odgovarajući objekt; *r1* kod prvog poziva, a *r2* kod drugog. Time se dodavanje podataka u rječnik *zapis* odnosi samo na polje *zapis* koje pripada objektu pridruženom parametru *self*. Rezultat ovih dvaju poziva vidljiv je u koraku 3: svaki od objekata *r1* i *r2* ima isto polje, *zapis*, jer ono je definirano u klasi *Kalendar* čiji su ti



Slika 7.1: Instanciranje klase *Kalendar*. Nakon prvog koraka dobiju se inicijalizirani objekti. U drugom koraku poziva se metoda *postavi_tekst* čiji je rezultat prikazan u koraku 3.

objekti instance. Međutim, sadržaj tog polja kod svakog je objekta drugačiji jer su za svaki poziv metode *dodaj_zapis* bili zadani drugačiji podaci.

Izraz kao što je `r1.dodaj_zapis('24.1.', '10:30', 'sastanak')` može se na prvi pogled činiti pogrešnim: metodu *dodaj_zapis* definirali smo s četiri parametra, *self*, *dan*, *vrijeme* i *tekst*, ali kod njenog poziva zadali smo ih samo tri. Radi se o tome da se parametar *self* tretira kao posebni parametar koji se postavlja automatski kod poziva metoda. U gornjem primjeru parametru *self* pridružen je onaj isti objekt koji je pridružen varijabli *r1*. Općenito, parametar *self* sadrži objekt za koji se poziva neka metoda. U Pythonu se ovaj poziv može napisati i na sljedeći način:

```
1 Kalendar.dodaj_zapis(r, '24.1.', '10:30', 'sastanak')
```

Sada je prvi parametar, *self*, zadan eksplicitno kroz varijablu *r*. Ova dva oblika poziva metoda po funkcionalnosti su ekvivalentni, ali je oblik *objekt.metoda()* puno češće zastupljen.

U Pythonu metode koje u svom nazivu počinju i završavaju sa `__` nazivaju se *specijalnim* metodama i Python ima poveći broj takvih metoda. Zašto je za inicijalizaciju

Primjer 7.5: Program za brojanje samoglasnika u znakovnom nizu.

```
tekst = 'grad'
broj = 0
for c in tekst:
    if c in 'ieaou':
        broj += 1

print(broj)
```

potrebna posebna metoda? Uzmimo za primjer jednostavan program koji broji samoglasnike u znakovnom nizu (i, e, a, o i u) prikazan u Primjeru 7.5.

Ovdje smo varijable *tekst* i *broj* morali postaviti na neku početnu vrijednost, to jest morali smo ih inicijalizirati. To je u ovakvim slučajevima lako jer znamo gdje program počinje i završava pa smo varijable inicijalizirali na početku. Međutim, ako pogledamo klasu *Kalendar*, gdje bi bio njen "početak"? Jasno je da metode možemo pozivati u bilo kojem redoslijedu, recimo prvo *dodaj_zapis*, pa onda *danas* ili obratno. Gdje tada možemo inicijalizirati varijable tog objekta? Upravo iz tog razloga postoji metoda *___init___* za koju se zna da će biti izvršena nakon instanciranja klase. U primjeru klase *Kalendar* izraz *Kalendar()* prvo rezervira memoriju za objekt te klase i u nju smjesti taj objekt, a nakon toga poziva metodu *___init___*. Ovo se sve odvija automatski - programer ne mora eksplicitno pozivati metodu *___init___*. Nadalje, vidimo da je unutar metode *___init___* ovo pridruživanje napisano kao

```
self.zapis = {}
```

umjesto

```
zapis = {}
```

(bez *self.*). To je zato što želimo da svaki objekt ove klase ima svoje polje *zapis*. Da smo ovo pridruživanje napisali na drugi način, `zapis = {}`, tada bi varijabla *zapis* bila uobičajena, lokalna varijabla metode *___init___* i nakon izlaska iz te metode njena bi se vrijednost izgubila. Općenito, kada napišemo `self.x`, x se odnosi na polje tog objekta, to jest na polje koje je dostupno svim metodama koje su definirane u toj klasi. Doseg ili vidljivost polja cijeli je objekt.

Sada naš kalendar možemo isprobati na jednom primjeru:

```
r = Kalendar()
r.dodaj_zapis('2020-9-13', '10:30', 'sastanak')
r.dodaj_zapis('2020-9-13', '13:45', 'rucak')
r.dodaj_zapis('2020-9-27', '8:00', 'razgovor s kandidatima')
print(r.zapis)
```

Ovaj program ispisuje

Primjer 7.6: Proširenje klase *Kalendar* s metodom *ispis*.

```
class Kalendar:
    ...

    def ispis(self):
        for dan in self.zapis:
            print(dan)
            for vrijeme in self.zapis[dan]:
                print('\t', vrijeme, '\t', self.zapis[dan][vrijeme])
```

```
{'2020-9-13': {'10:30': 'sastanak',
               '13:45': 'rucak'},
'2020-9-27': {'8:00': 'razgovor s kandidatima'}}
```

S obzirom da je ovakav ispis teško čitljiv dodat ćemo još jednu metodu, *ispis*, koja će na čitljiviji način ispisati sadržaj kalendara (Primjer 7.6).

Sada ispis izgleda ovako:

```
2020-9-13
     10:30    sastanak
     13:45    rucak
2020-9-27
     8:00     razgovor s kandidatima
```

Ovdje smo dobili ispis koji je sortiran, ali na to se ne smijemo oslanjati - rječnici ne sortiraju ni ključeve ni vrijednosti iako postoje alternative koje zadržavaju poredak ključeva prema poretku dodavanja u rječnik. Metoda *ispis* ima petlju u petlji: vanjska petlja prolazi kroz dane, a unutrašnja kroz vrijeme za svaki taj dan. Znakovni niz `'\t'` u naredbi `print('\t', ...)` ispisuje *tabulator*, slično funkciji tipke *Tab* na tipkovnicama. Time smo dobili ispis koji je okomito poravnat. Python ima i puno boljih mogućnosti za formatiranje ispisa koje ovdje nećemo koristiti, kao što su formatirani znakovni nizovi [25].

Treba li *Kalendar* biti klasa?

Jesmo li gornji program mogli napisati kao tri funkcije s jednom globalnom varijablom, *kalendar*? Drugim riječima, što smo dobili time da je *Kalendar* klasa koju eventualno moramo instancirati? Kao odgovor na prvo pitanje, *Kalendar* smo mogli napisati kao jedan modul s par funkcija i varijablom *kalendar* kako je prikazano u Primjeru 7.7.

Za početak možemo identificirati sljedeće razloge zašto je praktičnije da je *Kalendar* klasa:

1. Možemo imati više kalendara, odnosno više instanci klase *Kalendar*. Svaka od tih instanci ima svoju kopiju podataka, to jest svaka od njih je jedan zaseban kalendar. Instanca klase *Kalendar* predstavlja jednu cjelinu koja se sastoji od podataka (polje *zapis*) i operacija (metoda) čime smo definirali novi tip podatka.

Primjer 7.7: Modul *kalendar.py* s funkcijama kalendara.

```
# datoteka "kalendar.py"
import datetime

zapis = {}

def danas():
    return datetime.date.today()

def postavi_tekst(dan, vrijeme, tekst):
    if dan in zapis:
        zapis[dan][vrijeme] = tekst
    else:
        zapis[dan] = {vrijeme: tekst}

def ispis():
    for dan in zapis:
        print(dan)
        for vrijeme in zapis[dan]:
            print('\t', vrijeme, '\t', zapis[dan][vrijeme])

postavi_tekst('2020-9-13', '10:30', 'sastanak')
postavi_tekst('2020-9-13', '13:45', 'rucak')
postavi_tekst('2020-9-27', '8:00', 'razgovor s kandidatima')
ispis()
```

2. Kôd za inicijalizaciju nalazi se na jednom mjestu i poziva se u jasno definiranom trenutku.

3. Objekte možemo kombinirati u veće cjeline. To se zove *kompozicija* objekata [19] i izuzetno je korisna tehnika programiranja kojom se od zasebnih dijelova može sastaviti jedna veća cjelina.

4. Još jedna prednost upotrebe klasa je u tome što njih možemo proširivati *nasljeđivanjem*. Nasljeđivanje je jedna od karakteristika onog što se popularno naziva *objektno orijentiranim programiranjem* [30, 31] o čemu će više riječi biti kasnije.

Pored ovih prednosti, klase sačinjavaju temeljne elemente strukture programa što dolazi do izražaja u izradi većih programa. Iz tog razloga standardne biblioteke modernih programskih jezika sastoje se od velikog broja klasa, a isto je slučaj i s većim programima. Općenito, računalni program lakše je organizirati kao skupinu klasa nego kao skupinu funkcija jer svaka nam klasa može predstavljati jedan koncept s kojim radimo (kao što je kalendar), a to je teže postići sa skupinom funkcija koje nisu sintaksno povezane u jednu semantičku cjelinu kao što je klasa.

7.2 Iznimke

Model rada s iznimkama u Pythonu isti je kao i onaj kod drugih programskih jezika kao što su C#, Java i C++. Iznimke se u Pythonu definiraju naredbom *try/except/finally*. Opći oblik ove naredbe je

try:
    naredbe
except [*tip-iznimke* **as** *varijabla*]**:**
    naredbe
[**else:**
    naredbe]
[**finally:**
    naredbe]

Iznimku signaliziramo ("bacamo" ili "dižemo") naredbom *raise*. Dijelove *try* naredbe možemo nazivati *klauzulama*, tako da možemo reći da se ona sastoji od tri klauzule: *try* i *except* kao obavezne i *finally* kao opcionalne klauzule. Blok klauzule *else* bit će izvršen samo ako unutar klauzule *try* nije došlo do iznimke. Kada unutar naredbe *try* dođe do iznimke, izvršavanje bloka unutar klauzule *try* na tom se mjestu (naredbi) prekida i prelazi se na klauzulu *except* (kojih može biti više). Ako ta klauzula nema specificiran tip iznimke, izvrši se njen blok nakon čega se izvrši blok klauzule *finally* ako je definiran. Ako klauzula *except* ima specificiran tip iznimke, njen se blok izvrši samo ako signalizirana iznimka odgovara tom tipu. Blok klauzule *finally*, ako je prisutan, bit će izvršen uvijek – ili nakon bloka klauzule *try* ako nije došlo do iznimke, nakon bloka klauzule *except* ako je došlo do iznimke i njen tip odgovara specificiranom tipu *except* klauzule, ili nakon mjesta u klauzuli *try* gdje je došlo do iznimke, a ne postoji odgovarajuća klauzula *except* (to jest, nema klauzule *except* čiji tip odgovara tipu iznimke).

Sada ćemo proučiti osnovne mehanizme iznimki u Primjeru 7.8.

Ovaj program od korisnika traži unos imena datoteke, nakon čega tu datoteku otvara i poziva funkciju *ispisi* koja ispisuje prvi red te datoteke. Ovdje postoje dva scenarija koji signaliziraju iznimku:

1. datoteka ne postoji

2. prvi red datoteke je prazan znakovni niz.

Ako datoteka ne postoji, iznimka će biti signalizirana u funkciji *open*. Ta iznimka će biti tipa *IOError*. Nakon toga Python pokušava naći klauzulu *except* koja prihvaća taj tip iznimke, što je u ovom slučaju prva klauzula *except*. Izvršavanje bloka te klauzule ispisuje „Greska – [Errno 2] No such file or directory: '...'". S obzirom da je došlo do iznimke, blok klauzule *else* neće biti izvršen. Isto tako, s obzirom da smo definirali klauzulu *finally*, njen blok će biti izvršen pa će na kraju biti ispisan tekst „Sa ili bez greske ...".

Primjer 7.8: Primjer bacanja i hvatanja iznimki.

```
def ispisi(s):
    if s == '':
        raise EOFError()
    print(s)


datoteka = input('Ime datoteke: ')
try:
    with open(datoteka) as f:
        ispisi(f.readline())
except IOError as e:
    print('Greska --', e)
except Exception as e:
    print('Neka druga greska --', e)
else:
    print('Bez greske!')
finally:
    print('Sa ili bez greske ...')
```

U drugom scenariju dolazi do iznimke, ali ovaj put u funkciji *ispisi*. Ovdje program funkcionira kao i u prvom scenariju, s razlikom u tome da će se ovdje izvršiti blok druge klauzule *except* jer je sada tip iznimke *EOFError*, a ne *IOError*, što se vidi u naredbi *raise EOFError()*. Iako ova druga klauzula *except* prima iznimke tipa *Exception*, sve iznimke u stvari spadaju pod taj tip, pa tako i *EOFError* (više o ovome u dijelu o programiranju s klasama i objektima). Na ovaj način smo u prvoj klauzuli *except* naveli specifičnu iznimku za koju želimo izvršiti neki blok, dok smo drugom klauzulom obuhvatili sve ostale moguće iznimke. Ovo je čest način rada s iznimkama.

U slučaju da izvršavanjem ovog programa nema iznimki prvo će se izvršiti blok klauzule *try*, nakon toga blok klauzule *else* te na kraju blok klauzule *finally*.

Iznimke omogućavaju prekidanje izvršavanja programa i ispisivanje poruke o nastaloj greški.

Općenito, iznimke najčešće služe da se u slučaju greške u izvršavanju programa korisniku ispiše neka smislena poruka. Rijetko se u klauzuli *except* greška može ispraviti, ali nekada je i to moguće (na primjer, ako datoteka ne postoji možemo je u bloku klauzule *except* kreirati). Ako znamo da neka funkcija može signalizirati iznimku, kao funkcija *open*, onda je najbolje poziv te funkcije obuhvatiti naredbom *try/except* i u klauzuli *except* ispisati odgovarajuću poruku.

7.3 Programiranje s klasama i objektima

Malo toga u Pythonu možemo izraditi bez korištenja objekata neke klase. Ne samo zbog toga što je u Pythonu svaka vrijednost objekt nego zato što je većina korisnih biblioteka organizirana kao skup klasa kroz čije je instance njihova funkcionalnost dostupna. Općenito, gotovo svaki složeniji program radit će s instancama više klasa jer takav je način programiranja danas općeprihvaćen i podržan od strane mnogih popularnih programskih jezika [32].[2] U prethodnom smo dijelu prošli osnove klasa i objekata. U ovom dijelu proučit ćemo neke korisne tehnike programiranja s njima i vidjeti neke dodatne mogućnosti za koje postoji izravna podrška u Pythonu.

7.3.1 Klasa *Datum*

U ovom dijelu definirat ćemo klasu *Datum* koja će predstavljati datume koji se sastoje od dana, mjeseca i godine (bez vremena) i koja će zadovoljavati sljedeće uvjete:

- Da bismo instancirali klasu *Datum* moramo zadati dan, mjesec i godinu.
- Ako zadani datum nije logički ispravan (primjerice, mjesec nije u rasponu između 1 i 12) onda klasa *Datum* mora baciti iznimku tipa *NeispravanDatum*.
- Za objekte klase *Datum* trebaju biti definirani relacijski operatori (<, >, ==, ...).

Svrha ovog dijela je na jednostavnom primjeru demonstrirati osnovna načela programiranja s klasama.

7.3.1.1 Inicijalizacija i validacija

Inicijalizacija je općenito važna u programiranju. Mnoge se greške u kodu mogu pripisati pogrešnoj ili odsutnoj inicijalizaciji. U takvim su slučajevima klase vrlo koristan konstrukt jer podrazumijevaju mehanizam automatske inicijalizacije pomoću konstruktora, što smo vidjeli u prethodnom dijelu gdje smo koristili metodu ___*init*___. U sljedećem primjeru vidjet ćemo da inicijalizacija može sadržavati i validaciju.

Prvi uvjet kaže da se *Datum* ne smije moći instancirati kao

```
x = Datum()  # greska!
```

nego je potrebno zadati dan, mjesec i godinu:

```
x = Datum(17, 4, 2020)
```

Ovo postižemo definiranjem konstruktora s parametrima kako je pokazano u Primjeru 7.9.

Ako sada pokušamo instancirati klasu *Datum* bez ovih parametara Python će dojaviti grešku:

[2] Neki programski jezici, primjerice Rust i Go, nemaju klase iako podržavaju pojam objekta sličan onome koji je ovdje opisan. Bez obzira na to, takvi jezici obično imaju neki drugi koncept sličan klasi koji uključuje metode (iako se ne moraju tako zvati) i sučelje.

Primjer 7.9: Klasa *Datum* s konstruktorom.

```
class Datum:
    def __init__(self, dan, mjesec, godina):
        ...
```

```
>>> x = Datum()
...
TypeError: __init__() missing 3 required positional arguments: 'dan',
    'mjesec', and 'godina'
```

Ova greška kaže da nedostaju vrijednosti za parametre ”dan”, ”mjesec” i ”godina”. Na ovaj smo način spriječili korisnike klase *Datum* da počine grešku instanciranjem klase *Datum* bez zadanih konkretnih vrijednosti.

Drugi uvjet kaže da u slučaju neispravnog datuma mora biti bačena iznimka tipa *NeispravanDatum*. Ovdje prilikom validacije treba uzeti u obzir to da je broj za mjesec unutar intervala od 1 do 12 i za dan unutar intervala 1 do broja dana zadanog mjeseca. Isto tako, moramo uzeti u obzir prijelazne godine kod kojih je broj dana u veljači 29. Algoritam za provjeru je li godina prijelazna jednostavan je, ali Pythonova standardna biblioteka ima modul *calendar* koji već ima funkciju za to.

Sada možemo napisati konstruktor klase *Datum* kako je pokazano u Primjeru 7.10. Kôd metode *datum_ispravan* mogli smo staviti u sam konstruktor, to jest metodu *__init__*, ali uvijek je bolje razdvojiti dijelove koda koji izvode različite operacije, u ovom slučaju inicijalizaciju i validaciju. Pored toga, validacija u zasebnoj metodi omogućava naknadno validiranje datuma neovisno od inicijalizacije objekta ako se to pokaže potrebnim.

U redovima 17 do 19 vidimo inicijalizaciju polja koja sadrže dan, mjesec i godinu. Ta su polja označena s `__`. Takva se polja nazivaju *pseudoprivatnim* poljima i njihova je svrha onemogućavanje izravnog pristupa tim poljima. Na primjer,

```
x = Datum(2, 9, 2020)
print(x.__dan)  # greska - __dan je nedostupan!
```

Označavanjem polja pseudoprivatnim onemogućili smo im pristup ”izvana”, odnosno iz koda koji nije dio klase *Datum*.[3] Ova smo polja mogli definirati kao `dan`, `mjesec` i `godina` čime bismo tim poljima imali direktan pristup:

```
x = Datum(2, 9, 2020)
print(x.dan)  # ispravno
```

Problem bi, međutim, bio u tome što bi tada korisnicima klase *Datum* bilo omogućeno mijenjati datume izravno bez validacije čime bi se znatno povećala mogućnost pogreške. Na primjer,

[3] Tehnički, ovakvim poljima i dalje je moguće pristupiti putem `x._Datum__dan`, ali tada barem možemo lakše uočiti kôd koji ”radi nešto što ne bi trebao”.

Primjer 7.10: Klasa *Datum* s inicijalizacijom i validacijom.

```
import calendar
from datetime import datetime

class NeispravanDatum(Exception):
    pass

class Datum:
    def __init__(self, dan, mjesec, godina):
        # validacija
        if not self.datum_ispravan(dan, mjesec, godina):
            raise NeispravanDatum()

        # inicijalizacija

        self.__dan = dan
        self.__mjesec = mjesec
        self.__godina = godina
        self.__tekst = (str(godina)
                        + str.rjust(str(mjesec), 2, '0')
                        + str.rjust(str(dan), 2, '0'))

    def datum_ispravan(self, dan, mjesec, godina):
        dana_u_mjesecu = {
            1: 31,
            # prelazna godina?
            2: 29 if calendar.isleap(godina) else 28,
            3: 31,
            4: 30,
            5: 31,
            6: 30,
            7: 31,
            8: 31,
            9: 30,
            10: 31,
            11: 30,
            12: 31
        }

        if (1 <= dan <= dana_u_mjesecu[mjesec]
                and 1 <= mjesec <= 12
                and 1600 <= godina <= 9999):
            return True
        else:
            return False
```

Primjer 7.11: Modul *datum.py*.

```
# modul datum.py

from datetime import datetime


class Datum:
    ...


def danasnji_datum():
    danas = datetime.today()
    return Datum(danas.day, danas.month, danas.year)
```

```
x = Datum(2, 9, 2020)

# jesmo li sigurni da funkcija "odredi_dan" vraca ispravan rezultat?
x.dan = odredi_dan()
```

U slučaju da funkcija *odredi_dan* ima grešku i ne vraća uvijek ispravan rezultat, primjerice, da ponekad vrati dan koji nije u rasponu dana dotičnog mjeseca, ovim smo napravili još jedan propust u kodu jer sada ne bi bila bačena iznimka za neispravan datum.

U konstruktoru definirano je još jedno pseudoprivatno polje, ___*tekst*. To polje će sadržavati tekstualni zapis datuma oblika *ggggmmdd*. Primjerice, za datum 8.12.2020. polje ___*tekst* sadržavat će znakovni niz "20201208". To će nam polje trebati kasnije za uspoređivanje datuma.[4]

Jesmo li mogli konstruktor napisati tako da, ako nisu zadani parametri za datum, to podrazumijeva današnji datum? Mogli smo, ali za nekoga tko ne poznaje klasu *Datum* naredba kao što je `x = Datum()` može značiti najmanje dvije stvari:

1. Varijabla x sadržavat će današnji datum, ili
2. Varijabla x sadržavat će neki *podrazumijevani* datum, što može biti nešto kao 1.1.1800. ili 1.1. tekuće godine ili neki drugi datum.

Iz tog razloga najbolje je spriječiti mogućnost ovakvih pretpostavki i to riješiti na eksplicitan način pomoću funkcije koja je definirana u istom modulu kao i *Datum*, ali izvan te klase kako je pokazano u Primjeru 7.11.

[4]Funkcija *rjust* (engl. *right justify*) poravnava zadani znakovni niz udesno, s tim da umetne znakovni niz zadan u trećem parametru na početak tako da je ukupna duljina rezultirajućeg znakovnog niza jednaka onoj zadanoj u drugom parametru. Na primjer, `str.rjust('5', 2, '0')` vraća `"05"`, a `str.rjust('15', 2, '0')` vraća `"15"`.

Ako pretpostavimo da se ova funkcija nalazi u modulu *datum.py* u kojem se nalazi i klasa *Datum* onda instancu klase *Datum* inicijaliziranu na današnji datum možemo dobiti jednostavnim pozivom funkcije:

```
1 import datum
2
3 x = datum.danasnji_datum()  # "datum" je modul, ne klasa "Datum"
```

Sada više nema nedoumica oko toga što će sadržavati x.

7.3.1.2 Nepromjenjivost objekata

Jedan dodatak klasi *Datum* koji se čini korisnim mogućnost je promjene datuma, primjerice kao `datum.postavi_mjesec(1)`. Međutim, to bi utjecalo na razumljivost koda u kojem bi se upotrebljavala ova klasa i povećala mogućnost pogrešaka. To možemo vidjeti u sljedećoj ilustraciji:

```
1 # u nekom modulu
2 x = Datum(15, 7, 2022)
3
4 # puno koda
5 ...
6
7 # u istom ili nekom drugom modulu
8 prikazi x na web stranici
```

Analizirajući modul ili dio koda u kojem se nalazi redak 8 i ako pretpostavimo da se datum može mijenjati metodama kao *postavi_dan*, *postavi_mjesec* i *postavi_godinu*, možemo li zaključiti da će datum prikazan na web stranici biti 15.7.2022.? Sigurno ne jer

1. datum pridružen varijabli x mogao se promijeniti ili preko varijable x ili preko neke druge varijable koja sadrži isti objekt kao i x, i

2. varijabli x mogao je biti pridružen neki drugi objekt klase *Datum*.

Ne omogućavajući promjene u datumu pridruženom varijabli x eliminirali smo prvi problem i time umanjili mogućnost pogreške. Problem pod 1 nije samo u tome što smo negdje mogli napisati `x.postavi_dan(...)` nego i u tome što datum (objekt) koji je pridružen varijabli x može biti pridružen i nekoj drugoj varijabli. Tada smo promjenom datuma preko x utjecali i na sadržaj te druge varijable i dijela koda u kojem se ona upotrebljava i kojeg možda nismo svjesni da postoji ili kako radi.

Razlog pod 2 moguće je eliminirati upotrebom *konstantnih* varijabli iako Python izravno ne podržava definiranje takvih varijabli.

> Nepromjenjivost je poželjna karakteristika objekta (kada je to moguće).

Primjer 7.12: Klasa *Datum* sa svojstvima *dan*, *mjesec* i *godina*.

```
class Datum:
    ...

    @property
    def dan(self):
        return self.__dan

    @property
    def mjesec(self):
        return self.__mjesec

    @property
    def godina(self):
        return self.__godina
```

Objekti koji se ne mogu modificirati kao što su ovi klase *Datum* nazivaju se *nepromjenjivim objektima*. Općenito, nepromjenjivost objekata poželjna je karakteristika jer se onemogućavanjem modifikacije objekta mogu spriječiti određene vrste grešaka u programiranju. Prema tome, odluka o promjenjivosti ili nepromjenjivosti važan je dio dizajna klase. Što se tiče klase *Datum*, nema potrebe omogućiti modifikacije njenih objekata - ako je potrebno dobiti drugačiji datum od nekog postojećeg treba napraviti novu instancu datuma.

7.3.1.3 Pristup elementima datuma i enkapsulacija

Prethodno smo vidjeli da je problematično omogućiti izravan pristup poljima za dan, mjesec i godinu. Problem, međutim, nije u samom pristupu tim poljima nego u tome što nam takav pristup omogućava da ih modificiramo. S druge strane, korisno je, čak i neophodno, imati mogućnost pristupa ovim podacima. To možemo riješiti tako da za ova polja definiramo takozvana *svojstva* odnosno posebne metode kojima samo čitamo polja objekta, ali bez mogućnosti modifikacije istih. Za klasu *Datum* definirat ćemo tri takve metode, po jednu za svako od polja za dan, mjesec i godinu (Primjer 7.12).

Anotacija `@property` transformira metodu ispod u metodu koja je upotrebljiva kao polje:

```
>>> x = Datum(19, 7, 2019)
>>> x.dan  # poziv funkcije "dan", ali bez ( i )
19
```

Vidimo da metode *dan*, *mjesec* i *godina* upotrebljavamo kao obična polja jer te metode pozivamo bez zagrada za parametre. Mogli smo umjesto ovakvih metoda dodati i obične metode koje bi pozivali kao `x.dan()`, ali ovaj način je praktičniji.

Enkapsulacijom objedinjavamo podatke i metode objekta te ograničavamo izravan pristup podacima.

Na ovaj smo način zaštitili polja za dan, mjesec i godinu od (potencijalno) pogrešnih ili nevalidiranih vrijednosti, a istovremeno smo omogućili njihovo čitanje. Polja smo zaštitili tako da smo definirali *pristupne* metode koje samo vraćaju njihove vrijednosti. U terminologiji objektno orijentiranog programiranja to se zove *enkapsulacija* ili *učahurivanje* i podrazumijeva objedinjavanje podataka i metoda u jednu cjelinu te ograničavanje izravnog pristupa nekim ili svim poljima objekta.

Detalji implementacije klase odnose se na njenu unutrašnju implementaciju koja nije vidljiva ni dostupna korisnicima te klase. Promjene u detaljima implementacije klase ne smiju utjecati na njene korisnike.

Pseudoprivatna polja za dan, mjesec i godinu služe unutrašnjem prikazu datuma, ne korisnicima te klase. Ta su polja, prema tome, dio implementacije klase *Datum* i njima mogu pristupiti samo metode te klase. Pretpostavimo da umjesto tih polja odlučimo datum prikazati kao rječnik s ključevima "dan", "mjesec" i "godina". U tom slučaju moramo modificirati funkcije *dan*, *mjesec*, *godina*, *___repr___* i sve ostale funkcije u kojima se upotrebljavaju jer one bi sada podatke o datumu morale čitati iz rječnika. Međutim, sučelje klase *Datum* i dalje se neće mijenjati - sve metode i dalje će biti tu i njihova funkcionalnost, naziv i parametri i dalje će biti isti. Na korisnike klase, dakle, ta promjena neće utjecati čime je ostvaren cilj enkapsulacije.

7.3.1.4 Tekstualni prikaz objekta

Ako sada pokušamo ispisati neki objekt tipa *Datum*, dobili bismo ispis sličan ovome:

```
>>> Datum(18, 6, 2020)
<datum.Datum object at 0x000001FC8C446CD0>
```

U Pythonu možemo definirati metodu koja vraća znakovni niz kada se mora ispisati objekt klase u kojoj je ta metoda definirana. To je jedna od posebnih metoda koja se zove *___repr___*. Za klasu *Datum* tu metodu ćemo definirati tako da vraća znakovni niz oblika *dan.mjesec.godina.* kako je pokazano u Primjeru 7.13.

Sada ispis datuma izgleda bolje:

```
>>> Datum(18, 6, 2020)
18.6.2020.
```

Ova metoda samo formira i vraća znakovni niz na osnovu vrijednosti u poljima za dan, mjesec i godinu.

7.3.1.5 Uspoređivanje datuma

Datumi su vrijednosti koje je prirodno uspoređivati po "veličini" odnosno nekakvom uređenju kao što je "jednako", "manje", "veće" i ostalo. Kao što znakovne nizove us-

Primjer 7.13: Klasa *Datum* s metodom *__repr__*.

```
class Datum:
    ...

    def __repr__(self):
        return (str(self.__dan) + '.'
                + str(self.__mjesec) + '.'
                + str(self.__godina) + '.')
```

poređujemo s "==", "<", ">" i ostalim relacijskim operatorima, tako i datume možemo uspoređivati na isti način.

> Operatori trebaju oponašati konvencionalnu upotrebu i njihova semantika treba biti jasna u kontekstu tipa podataka za koji se upotrebljavaju.

Primjerice, upotreba operatora "+" za zbrajanje matrica uobičajena je u matematici pa bi i za takav tip podatka bilo korisno definirati taj i druge aritmetičke operatore. S druge strane, za tip *Datum* mogli bismo definirati operaciju dodavanja dana koja bi vraćala novi objekt klase *Datum*. Bi li imalo smisla dodavanje dana implementirati operatorom "+"? Ovakvu upotrebu operatora "+" bilo bi teško opravdati jer taj se operator konvencionalno ne upotrebljava za datume i nekome tko nije upoznat s klasom *Datum* bilo bi teško zaključiti što znači izraz kao `x + 5` gdje je x objekt tipa *Datum*. U ovom bi slučaju puno jasnije bilo definirati funkciju ili metodu koja se zove "dodaj_dane" (ili nešto slično) s kojom bismo jednostavno pisali `dodaj_dane(x, 5)`. Ime ove funkcije jasno govori što ona radi.

Uzevši sve ovo u obzir, ima li smisla datume *d1* i *d2* uspoređivati kao `d1 < d2` ili je bolje pisati `manje_od(d1, d2)` ili `d1.manje_od(d2)`? S tehničke strane je svejedno - sva tri oblika podjednako je jednostavno implementirati i upotrebljavati. Međutim, uzmimo sljedeći slučaj u obzir:

```
>>> sorted([4, 2, 8, 1])
[1, 2, 4, 8]

>>> sorted(['jedan', 'dva', 'tri'])
['dva', 'jedan', 'tri']

>>> sorted([[9, 2], [3, 1, 7], [101], [1, 2, 3, 4]])
[[1, 2, 3, 4], [3, 1, 7], [9, 2], [101]]

>>> sorted([Datum(19, 1, 2020), Datum(2, 7, 2020),
            Datum(23, 2, 2019)])
TypeError: '<' not supported between instances of 'Datum' and 'Datum'
```

Funkcija *sorted* sortira vrijednosti neke kolekcije podataka (u ovom slučaju liste). Vidimo da istom funkcijom možemo sortirati brojeve, znakovne nizove, liste, ali i druge

Primjer 7.14: Klasa *Datum* s metodom *__eq__*.

```
class Datum:
    ...

    def __eq__(self, desni):
        return (self.__dan == desni.__dan
                and self.__mjesec == desni.__mjesec
                and self.__godina == desni.__godina)
```

tipove podataka. Kako funkcija *sort* sortira sve ove različite tipove podataka? Budući da su to ugrađeni tipovi podataka kao i sama funkcija, mogli bismo pomisliti da je i u nju unaprijed ugrađena funkcionalnost sortiranja ugrađenih tipova podataka. Odgovor se, međutim, može naslutiti iz posljednjeg primjera sa sortiranjem datuma, to jest vrijednosti tipa *Datum*. Rezultirajuća greška kaže da za instance tipa *Datum* nije definiran operator "<". Upravo zbog toga funkcija *sort* radi s brojevima, znakovnim nizovima, listama i drugim tipovima podataka. U sljedećem primjeru vidimo da instance ovih tipova podataka možemo uspoređivati operatorom "<":

```
>>> 4 < 2
False

>>> 'jedan' < 'dva'  # 'j' je po abecedi iza 'd'
False

>>> [3, 1, 7] < [101]  # 3 je ispred 101
True
```

Operator "<" je operator na koji se oslanja funkcija *sort* da bi za dvije vrijednosti odredila njihov poredak. Ako bi svaki od ovih tipova podataka imao neku svoju funkciju ili metodu za usporedbu (sličnu gore spomenutoj metodi `manje_od`) bilo bi teže napisati generičku funkciju kao što je *sort* koja bi radila s mnogim tipovima podataka, a i bilo bi ju teže koristiti.

S obzirom da za datume postoji opće prihvaćen poredak, klasa *Datum* dobar je kandidat za implementaciju relacijskih operatora. Iako to naizgled nije očito, operatori su tehnički funkcije. Primjerice, funkciju koja vraća najveći zajednički djelitelj dvaju brojeva mogli bismo specificirati kao *nzd(a, b)*. Slično, funkciju koja zbraja dva broja mogli bismo specificirati kao *+(a, b)*, a isto i s operatorom "<", to jest *<(a, b)* kao i sa svim ostalim relacijskim, aritmetičkim i logičkim operatorima. S obzirom da u Pythonu ove operatore definiramo za specifične tipove, njihove funkcije moraju biti dio objekata koje uspoređujemo, odnosno moraju biti definirani unutar klase. To znači da bi s tehničke strane poziv funkcije `<(a, b)` trebao biti poziv metode "<", to jest `a.<(b)`, gdje je *a* objekt tipa za koji je definiran operator "<". Izraz `a < b`, dakle, isto je što i `a.<(b)`. Isto vrijedi i za druge operatore: `a + b` je isto što i `a.+(b)`, a `a == b` isto što i `a.==(b)`.

U Pythonu postoji skup posebnih funkcija (onih koje započinju i završavaju s `__`)

Primjer 7.15: Klasa *Datum* s metodom ___*ne*___ koja je u ovom slučaju nepotrebna.

```
class Datum:
    ...
    def __ne__(self, desni):
        return not (self == desni)
```

Primjer 7.16: Klasa *Datum* s metodom ___*lt*___.

```
class Datum:
    ...

    def __lt__(self, desni):
        return self.__tekst < desni.__tekst
```

koje predstavljaju ovakve operatore. Jedna takva funkcija je ___*eq*___ (engl. *equality*) za relacijsku operaciju jednakosti. Time bi izraz `a == b` zapravo bio poziv metode ___*eq*___, to jest `a.__eq__(b)`. Ovo je načelo po kojem u Pythonu definiramo operatore za nove (neugrađene) tipove podataka. Na primjer, operator jednakosti za klasu *Datum* definira se kako je pokazano u Primjeru 7.14. Za izraz `a == b` parametar *self* sadržavao bi objekt *a*, a parametar *desni* objekt *b* (ono što je desno od "=="). S obzirom da relacijski operatori trebaju vratiti logičku vrijednost kao rezultat (*True* ili *False*) tako i metoda ___*eq*___ vraća rezultat logičkog izraza kojim uspoređujemo vrijednosti za dan, mjesec i godinu.

Ovdje se dobro vidi da su operatori obične metode s jednim parametrom (i *self*) koje vraćaju tip vrijednosti koji odgovara njihovoj semantici. Jedino što ih razlikuje od većine ostalih funkcija i metoda je to što se pišu u *infiksnom* ili takozvanom *školskom* obliku (dva parametra sa strane i operator u sredini), a ne kao obična funkcija.

Operator za nejednakost (engl. *not-equal*) mogli bismo definirati tako da negira operator za jednakost (Primjer 7.15). Međutim, ako je definirana metoda ___*eq*___ Python za "!=" negira njen rezultat ako ___*ne*___ nije definirana za objekt s lijeve strane "!=" pa zbog toga u većini slučajeva metodu ___*ne*___ nema potrebe definirati.

Na ovaj način možemo definirati i ostale relacijske operatore. Za njih će nam trebati polje ___*tekst* koje inicijaliziramo u konstruktoru. Da se podsjetimo, to polje sadrži tekstualni prikaz datuma u obliku *gggg.mm.dd*, recimo "20210904" za datum 4.9.2021. Upotrebom tog polja možemo datume usporediti leksički (kao tekst) čime izbjegavamo pisanje logičkih izraza za usporedbu pojedinačnih polja za dan, mjesec i godinu. Operator *manje-od* (engl. *less-than*) bi, dakle, izgledao kao u Primjeru 7.16.

Sada je operator za *manje-ili-jednako* (engl. less-or-equal) trivijalan jer već imamo operatore za "<" i "==" (Primjer 7.17).

Na sličan bismo način mogli definirati i metode za operatore ">" i ">=", međutim ako smo definirali operatore "<" i "<=" onda Python koristi njih da bi došao do rezultata za ">" i ">=". Na primjer, ako operator ">" nije implementiran u klasi objekta

Primjer 7.17: Klasa *Datum* s metodom *__le__* koja poziva metode *__lt__* (za "<") i *__eq__* za "

```
class Datum:
    ...

    def __le__(self, desni):
        return self < desni or self == desni
```

a za izraz `a > b`, Python taj izraz interpretira kao `b < a` te pokuša pozvati metodu za "<" objekta *b*. Slično radi i s operatorom ">=".

Sada sortiranje datuma funkcijom *sorted* radi bez greške:

```
>>> sorted([Datum(19, 1, 2020), Datum(2, 7, 2020),
            Datum(23, 2, 2019)])
[23.2.2019., 19.1.2020., 2.7.2020.]
```

Vidimo i da je ispis datuma u rezultirajućoj listi onaj koji smo definirali metodom *__repr__*.

7.3.1.6 Pomoćne funkcije

Da bi bila korisna, klasa kao što je *Datum* treba nuditi puno više funkcionalnosti od onoga što ona sada ima. Na primjer, funkcije za "aritmetiku" datuma, kao što je dodavanje ili oduzimanje dana, mjeseci i godina, funkcije za određivanje idućeg radnog dana ili idućeg vikenda, razne mogućnosti formatiranja datuma neke su od tih mogućnosti.

Pretpostavimo da želimo omogućiti aritmetiku datuma dodavanjem nekoliko funkcija ili metoda. Na koji bi način bilo najbolje proširiti postojeći kôd? Imamo nekoliko mogućnosti:

- dodati nove metode u klasu *Datum*
- dodati nove statičke metode u klasu *Datum* (opisano u dijelu o statičkim metodama)
- dodati nove funkcije izvan klase *Datum* kao dio modula u kojem se nalazi *Datum*
- dodati nove funkcije u zasebni modul namijenjen pomoćnim funkcijama za rad s datumima ili neki modul u kojem bi se nalazile sve pomoćne funkcije.

Koji je od gore navedenog najbolji pristup? Odgovoriti možemo postavljajući nekoliko pitanja:

1. Trebaju li ove nove funkcije imati pristup privatnim poljima klase *Datum* (`__dan`, `__mjesec`, `__godina`)?
2. Može li promjena u detaljima implementacije klase *Datum* utjecati na njih?

Odgovor na prvo pitanje zavisi od toga želimo li da te funkcije modificiraju postojeći objekt ili da vrate novi objekt (datum) kao rezultat. Ako se odlučimo za ovo drugo onda je odgovor na prvo pitanje negativan: S obzirom da instance klase *Datum* imaju samo podatke za dan, mjesec i godinu, i dostupni su pomoću funkcija *dan*, *mjesec* i *godina*, pristup privatnim poljima ovim funkcijama nije potreban. Ovdje se treba prisjetiti činjenice da su funkcije *dan*, *mjesec* i *godina* dio sučelja klase *Datum*, a sučelje klase ne mijenja se često.

Drugo pitanje povezano je s prvim: Detalji implementacije više su podložni promjenama nego sučelje klase jer nisu vidljivi korisnicima klase pa takve promjene na njih ne utječu. Prema tome, ako nove funkcije (koje bismo tada implementirali kao metode) upotrebljavaju polja za dan, mjesec i godinu onda će svaka promjena kao ova zahtijevati i promjenu u tim metodama.

Iz ovoga možemo zaključiti da ako funkcija ne treba pristup detaljima implementacije klase i ako može koristiti postojeće sučelje te klase onda ju je bolje definirati kao samostalnu funkciju jer je tada manje podložna promjenama u detaljima implementacije te klase.[5]

Ove bismo funkcije, dakle, implementirali izvan same klase *Datum*.[6] S obzirom da su usko povezane s tom klasom, treba ih definirati u istom modulu. Na taj će način korisnici moći učitati klasu *Datum* i pripadajuće funkcije naredbom `import datum`. Isto vrijedi i za ostale pomoćne funkcije koje ne trebaju pristup implementaciji klase, kao što su funkcije za dodavanje i oduzimanje dana, mjeseci ili godina, određivanje sljedećeg radnog dana, sljedećeg vikenda, broj dana između dvaju datuma i druge.

Pretpostavimo da želimo u modul *Datum* dodati sljedeće funkcije:

- `dodaj_dane(d, n)` - funkcija koja vraća novi objekt klase *Datum* kojem je dodano n dana datumu d
- *oduzmi_dane(d, n)* - isto kao *dodaj_dane*, ali se dani oduzimaju
- `dodaj_mjesece(d, n)` - funkcija koja vraća novi objekt klase *Datum* kojem je dodano n mjeseci datumu d
- `oduzmi_mjesece(d, n)` - isto kao *dodaj_mjesece*, ali se mjeseci oduzimaju
- `dodaj_godine(d, n)` - funkcija koja vraća novi objekt klase *Datum* kojem je dodano n godina datumu d
- `oduzmi_godine(d, n)` - isto kao *dodaj_godina*, ali se godine oduzimaju.

[5] Pod "samostalnom" misli se na funkciju koja nije definirana unutar neke klase. U nekim se jezicima kao što je C++ takve funkcije nazivaju *nečlanskim* funkcijama.

[6] Neki pobornici objektno orijentiranog programiranja vjerojatno bi rekli da to nije objektno orijentirani pristup. Studentima se preporučuje realno sagledavanje problema iz perspektive objektivnih prednosti i nedostataka, a ne ideologije, općeprihvaćenih stavova ili trenutno popularne paradigme programiranja.

Primjer 7.18: Struktura modula *datum.py*.

```
import calendar
from datetime import datetime

# klasa Datum

class Datum:
    ...

# ostale klase

...

# konstante

PRAZNICI = ...
NERADNI_DANI = ...

# Pomocne funkcije koje ne trebaju pristup detaljima implementacije
# klase Datum.

def dodaj_dane(d, n):
    ...

def oduzmi_dane(d, n):
    ...

def dodaj_mjesece(d, n):
    ...

def oduzmi_mjesece(d, n):
    ...

def dodaj_godine(d, n):
    ...

def oduzmi_godine(d, n):
    ...
```

Primjer 7.19: Enumeracija za dane.

```
from enum import Enum

class Dani(Enum):  # enumeracija Dani
    PONEDJELJAK = 1,
    UTORAK = 2,
    SRIJEDA = 3,
    CETVRTAK = 4,
    PETAK = 5,
    SUBOTA = 6,
    NEDJELJA = 7
```

Struktura modula *datum* time bi izgledala kao u Primjeru 7.18. Taj je modul imenski prostor koji sadrži logički povezane dijelove koda, odnosno klase, funkcije i sve ostalo što ima veze s datumima. Poredak ovih elemenata u modulu *datum* nije važan. U dijelu pod komentarom *konstante* možemo postaviti nepromjenjive podatke kao što su praznici, neradni dani i ostale konstante vezane za datume, zavisno od toga gdje i kako se klasa *Datum* upotrebljava. Možemo primijetiti da ovaj modul može sadržavati više klasa, ne samo klasu *Datum*. Na primjer, može sadržavati klasu namijenjenu isključivo formatiranju datuma.

7.3.1.7 Enumeracije

Kako bismo, primjerice, u kodu prikazali dane u tjednu? Jedan način je brojevima od 1 do 7. Tako bismo, na primjer, varijabli *dan* pridružili broj za utorak:

```
dan = 2  # utorak
```

Jedno poboljšanje bilo bi to da definiramo konstante za dane:

```
PONEDJELJAK = 1
UTORAK = 2
SRIJEDA = 3
...
```

Ovdje je još uvijek problem u tome što nemamo jasan raspon vrijednosti koji bi označavao dane u tjednu. Idealan bi bio tip podatka koji bi predstavljao dane u tjednu. Upravo to možemo postići upotrebom *enumeracija*.

> Enumeracija je tip podatka. To je klasa koja se najčešće sastoji od više imenovanih konstanti.

Na primjer, enumeracija za dane u tjednu može izgledati kao u Primjeru 7.19. Sada takvu enumeraciju možemo koristiti na sljedeći način:

```
dan = Dani.UTORAK
```

Brojevi pridruženi poljima enumeracije *Dani* i njihov poredak su arbitrarni. Iako se na prvi pogled ne čini da smo dobili puno više nego upotrebom zasebnih konstanti, postoje barem tri prednosti korištenja enumeracija:

- Polja enumeracije su nepromjenjiva; primjerice, polju UTORAK ne možemo naknadno pridružiti neku vrijednost.
- Sva polja enumeracije obuhvaćena su jednim imenskim prostorom, onim klase enumeracije kao što je *Dani*.
- Ovaj način posebno je koristan kada varijablama unaprijed postavimo tip podatka koji im može biti pridružen. Tada pomoćni alati za pisanje koda[7] mogu upozoriti na neispravno pridruživanje.

Treći argument prikazan je u sljedećem primjeru:

```
def f(dan: Dani):
    if dan in {Dani.SUBOTA, Dani.NEDJELJA}:
        print('vikend')
    else:
        print('radni dan')

>>> f(Dani.SRIJEDA)  # ispravan poziv
'radni dan'

>>> f(3)  # neispravan poziv
'radni dan'
```

Iako smo za poziv `f(3)` dobili ispravan rezultat, taj bi poziv na neki način bio označen kao problematičan jer parametru tipa *Dani* pridružujemo vrijednost tipa *int*.

7.3.1.8 Statička polja i metode

Klasa, pored ostalog, predstavlja imenski prostor. Za razliku od objekata koji su instance neke klase, može postojati samo jedna definicija neke klase unutar istog imenskog prostora. Primjerice, ne možemo imati dvije ili više klasa koje se zovu *Datum*. Sve metode neke klase sastavni su dio te klase, a komunikacija s njenim instancama odvija se kroz parametar *self*. To se u Pythonu lako vidi po činjenici da su sljedeća dva poziva metode *dan* ekvivalentna:

```
d = Datum(1, 1, 2021)

dan = d.dan()
dan = Datum.dan(d)  # isto kao i poziv iznad
```

U pozivu `Datum.dan(d)` metodu *dan* pozivamo putem klase u kojoj je definirana, gdje je parametru *self* eksplicitno pridružena instanca te klase (*d*), dok u pozivu `d.dan()`

[7] Kao što je Pylance za VS Code editor.

Primjer 7.20: Klasa *datum* s enumeracijom i poljem za formatiranje.

```
class Datum:
    class Format(Enum):
        EU = 1,
        SAD = 2

    format = Format.EU

    ...
```

tu metodu pozivamo putem instance klase *Datum* (kao i kod većine drugih programskih jezika). Metode, dakle, rade s instancama klase koje su zadane u parametru *self*, ali sama metoda dio je klase, a ne objekta. Drugim riječima, objekti nemaju svoje kopije metoda.

Ne mora, međutim, svaka metoda u svom prvom parametru *self* prihvaćati instancu te klase. Takve se metode tada nazivaju *statičkim metodama*. Statičkim, zato jer ne rade s instancama klase (koje su kreirane dinamički, odnosno u toku izvršavanja programa) nego sa samom klasom. Zbog toga statičke metode nemaju pristup instancama svoje klase pa nemaju ni pristup podacima tih instanci. Takve se metode obično upotrebljavaju za sljedeće:

- Kao funkcije koje trebaju pristup statičkim poljima i statičkim metodama klase u kojoj su definirane, ali koje ne trebaju pristup instancama te klase.
- Kao pomoćne funkcije kojima klasa u kojoj su definirane služi samo kao imenski prostor.

Statička polja često se upotrebljavaju za podatke koji trebaju biti "zajednički" svim instancama klase, ali gdje nije potrebno ili poželjno da svaka od njih ima svoju kopiju tih podataka.

Pretpostavimo da za klasu *Datum* želimo dodati dva formata ispisivanja datuma: europski format gdje se dan stavlja prije mjeseca, kao 27.3.2020., i američki format gdje se mjesec stavlja prije dana, kao 3/27/2020. Isto tako, format ne želimo zadavati kod svakog instanciranja klase *Datum*. Prema tome, ovakav problem možemo riješiti tako da dodamo jedno statičko polje koje će sadržavati format za ispis. Nadalje, to polje može primiti vrijednost enumeracije umjesto znakovnog niza ili broja. U Primjeru 7.20 prikazan je dio klase *Datum* s ovim dodacima. Enumeracija *Format* stavljena je unutar klase *Datum* jer je semantika te enumeracije usko povezana s datumima. Polje *format* je polje koje će sadržavati format ispisa i ovdje je inicijalno postavljeni na europski format. Kao što vidimo, polja kao što su *format* nisu inicijalizirana u konstruktoru klase jer konstruktor služi za inicijalizaciju *instanci* klase. Statička polja inicijaliziraju se zasebno, kao dio definicije same klase. Format datuma sada možemo postaviti ovako:

Primjer 7.21: Klasa *datum* s enumeracijom i pseudoprivatnim poljem za formatiranje.

```
class Datum:
    class Format(Enum):
        EU = 1,
        SAD = 2

    __format = Format.EU

    ...
```

Primjer 7.22: Klasa *Datum* sa statičkom metodom *postavi_format*.

```
    class Datum:
        class Format(Enum):
            EU = 1,
            SAD = 2

        __format = Format.EU

        @staticmethod
        def postavi_format(fmt):  # staticka metoda
            Datum.__format = fmt
```

```
>>> Datum.format = Datum.Format.SAD
>>> x = Datum(27, 3, 2020)
>>> x
3/27/2020
>>> Datum.format = Datum.Format.EU
>>> x = Datum(27, 3, 2020)
>>> x
27.3.2020.
```

S obzirom da je polje *format* dio klase *Datum*, a ne njene instance, pristupamo mu navođenjem te klase. Isto vrijedi i za enumeraciju *Format*. Međutim, i instance klase imaju pristup statičkim poljima pa formatu možemo pristupiti i instancom klase *Datum*:

```
>>> x = Datum(27, 3, 2020)
>>> x.format
Format.EU
```

Polje *format* može biti i pseudoprivatno kako je pokazano u Primjeru 7.21. Sada ne možemo izravno pristupiti tom polju pomoću `Datum.__format`. Tu imamo dvije mogućnosti: pristup ovom polju putem metode objekta ili metode klase, tj. statičke metode. Statičku metodu možemo definirati kako je pokazano u Primjeru 7.22. Vidimo da statičke metode nemaju *self* kao prvi parametar što je logično jer one nemaju pristup instancama klase. Sada bismo format postavili na ovaj način:

```
Datum.postavi_format(Datum.Format.SAD)
```

Isto tako, mogli smo napisati i običnu metodu koja bi funkcionirala na isti način, ali s obzirom da *postavi_format* radi isključivo sa statičkim poljem nije potrebno zahtijevati instanciranje klase. Ovdje treba imati u vidu to da u funkciji *postavi_format* ne možemo pisati `__format = fmt` (bez `Datum.`) jer bi to Python interpretirao kao pridruživanje lokalnoj varijabli *___format* čime ne bismo dobili željeni učinak.

7.3.2 Kompozicija objekata

Pretpostavimo sada da s datumom želimo po potrebi zadati i vrijeme. Ovdje imamo dvije mogućnosti na raspolaganju:

- proširiti klasu *Datum* tako da se može zadati i vrijeme, ili
- napisati novu klasu, primjerice, *DatumVrijeme*, koja upotrebljava postojeću klasu *Datum* i novu klasu *Vrijeme* koja predstavlja samo vrijeme.

Obje mogućnosti su prihvatljive. Nedostatak prve je što bi klasa *Datum* tada bila malo složenija. Primjerice, usporedba datuma morala bi uzeti u obzir i vrijeme, a isto tako i sama inicijalizacija bila bi složenija jer ne bi bilo obavezno zadati i vrijeme pored datuma te bi trebalo podatke o vremenu smjestiti u zasebna polja. Drugi nedostatak nešto je dublji: vrijeme često možemo koristiti i u druge svrhe osim kao dio datuma. Na primjer, vrijeme možemo koristiti i za potrebe mjerenja (štoperica) ili kao mjerač vremena (engl. *timer*) kojem možemo zadati neko vrijeme, recimo 20 minuta, gdje on zvučno ili na neki drugi način signalizira kada to vrijeme istekne. Iz ovih razloga odabrat ćemo drugu mogućnost jer ona nam daje više slobode ako želimo mogućnosti s vremenom zasebno razvijati i proširivati.

Prvo možemo definirati klasu *Vrijeme* koja će predstavljati vrijeme koje se sastoji od sati, minuta i sekundi. Ta je klasa gotovo identična klasi *Datum* s tim da je validacija jednostavnija jer nema prijelaznih godina i mjeseci s različitim brojem dana (Primjer 7.23).

Klasu *DatumVrijeme* sada možemo implementirati kombinacijom klasa *Datum* i *Vrijeme* kako je pokazano u Primjeru 7.24. Tu klasu možemo koristiti na sljedeći način:

```
x = DatumVrijeme(Datum(15, 2, 2021), Vrijeme(20, 25, 0))
print(x)  # ispisuje 15.2.2021. 20:25:0
```

> *Kompozicija objekata* tehnika je programiranja kod koje se jedan objekt sastoji od jednog ili više drugih [19].

Objekt klase *DatumVrijeme* sastoji se, dakle, od dvaju drugih objekata: jednog klase *Datum* i drugog klase *Vrijeme*. Ovakva kombinacija objekata zove se *kompozicija objekata* i korisna je tehnika dizajna programa u slučajevima gdje želimo iskoristiti postojeće objekte za izradu novih u svrhu proširenja funkcionalnosti aplikacije.

Primjer 7.23: Modul *vrijeme.py*.

```
# modul "vrijeme.py"

class NeispravnoVrijeme(Exception):
    pass


class Vrijeme:
    def __init__(self, sati, minute, sekunde):
        if not self.vrijeme_ispravno(sati, minute, sekunde):
            raise NeispravnoVrijeme

        self.__sati = sati
        self.__minute = minute
        self.__sekunde = sekunde

        self.__tekst = (str(sati)
                        + str.rjust(str(minute), 2, '0')
                        + str.rjust(str(sekunde), 2, '0'))

    def vrijeme_ispravno(self, sati, minute, sekunde):
        if (0 <= sati <= 23
                and 0 <= minute <= 59
                and 0 <= sekunde <= 59):
            return True

        return False

    # metode za pristup poljima (ostavljeno za vjezbu)
    # ...

    def __repr__(self):
        return (str(self.__sati) + ':'
                + str(self.__minute) + ':'
                + str(self.__sekunde))

    def __eq__(self, desni):
        return (self.__sati == desni.__sati
                and self.__minute == desni.__minute
                and self.__sekunde == desni.__sekunde)

    def __lt__(self, desni):
        return self.__tekst < desni.__tekst

    def __le__(self, desni):
        return self < desni or self == desni
```

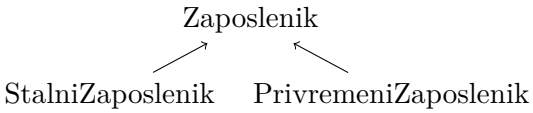
Primjer 7.24: Klasa *DatumVrijeme*.

```
# modul "datum_vrijeme.py"

from datum import Datum
from vrijeme import Vrijeme


class DatumVrijeme:
    def __init__(self, datum, vrijeme):
        self.__datum = datum
        self.__vrijeme = vrijeme

    def __repr__(self):
        return str(self.__datum) + ' ' + str(self.__vrijeme)

    def __eq__(self, desni):
        return (self.__datum == desni.__datum
            and self.__vrijeme == desni.__vrijeme)

    def __lt__(self, desni):
        if self.__datum == desni.__datum:
            return self.__vrijeme < desni.__vrijeme
        elif self.__datum > desni.__datum:
            return False
        else:
            return True

    def __le__(self, desni):
        return self < desni or self == desni
```

7.3.3 Nasljeđivanje

Klasom definiramo novi tip podatka. Svaki tip podatka podržava određene operacije nad tim tipom. Primjerice, klase *int* i *float* dva su brojčana tipa podatka koji podržavaju tipične aritmetičke operacije (+, -, *, /, %, ...). U mnogim programskim jezicima tipove podataka možemo grupirati u sveobuhvatnije koncepte. Primjerice, tipove *int* i *float* možemo obuhvatiti jednim širim konceptom, recimo tipom *broj*. Tada bismo rekli da su tipovi *int* i *float* brojevi, to jest oba tipa su *podtipovi* tipa *broj*. Tip *broj* bi, prema tome, bio tip koji obuhvaća ono što je zajedničko tipovima *int* i *float*, kao što su aritmetičke operacije. Međutim, za tip *int* bilo bi specifično to da su za njega definirane operacije pomicanja bitova (”«” i ”»”) koje nisu definirane za tip *float*. To znači da tip *broj* ne bi trebao podržavati te dvije operacije jer one nisu zajedničke tipovima *int* i *float*.

Sličnom logikom možemo promatrati i korisničke tipove podataka. Neka nam je zadan sljedeći zadatak: treba napisati aplikaciju za vođenje evidencije o zaposlenicima. Zaposlenici se dijele na stalne i privremene. Privremeni zaposlenici zapošljavaju se na



Slika 7.2: Hijerarhija zaposlenika.

određeni vremenski period. Primanja zaposlenika izračunavaju se tako da se izračuna koeficijent koji uzima u obzir staž i kategoriju zaposlenika. Svi zaposlenici imaju osnovni dohodak od 1000 eura. Kategorija može biti radnik, voditelj tima, voditelj projekta i voditelj odjela. Ako je staž od dvije do pet godina koeficijent je 1.1, od pet do osam godina 1.2, a za više od osam je 1.5. Za staž manji od dvije godine koeficijent je 1. Isto tako, osnovni dohodak je 1000 eura. Nadalje, stalni zaposlenici dobivaju 25 dana godišnjeg odmora s dodatkom jednog dana za svakih dodatnih pet godina radnog staža. Za privremene zaposlenike godišnji odmor računa se na isti način samo što obračunavanje počinje s 20 dana.

Tipičan pristup rješavanju ovakvog problema bio bi postavljanjem ključnih koncepata kao što su zaposlenik, stalni i privremeni u hijerarhiju tipova prikazanu na Slici 7.2. Ovakva hijerarhija u kontekstu nasljeđivanja podrazumijeva da su *StalniZaposlenik* i *PrivremeniZaposlenik* podtipovi tipa *Zaposlenik* odnosno klase *StalniZaposlenik* i *PrivremeniZaposlenik* nasljeđuju od klase *Zaposlenik*. Nadalje, ovakav odnos među klasama često se interpretira kao *je* (engl. *IS-A*), primjerice *StalniZaposlenik* ***je*** *Zaposlenik*. Međutim, ovakvu interpretaciju nasljeđivanja ne treba previše doslovno shvatiti jer nije uvijek idealna - nasljeđivanje se često može i treba interpretirati i kao odnos *ponaša-se-kao* ili *funkcionira-kao*. Takvim se odnosom želi istaknuti činjenica da je tip *Zaposlenik* općenitiji koncept, a tipovi *StalniZaposlenik* i *PrivremeniZaposlenik* više specijalizirani ili specifični koncepti. Nadalje, ovakvim odnosom implicira se da klasa *Zaposlenik* sadrži metode i/ili polja koja su zajednička izvedenim klasama. Primjerice, ako zaposlenici općenito imaju broj onda ga imaju i *StalniZaposlenik* i *PrivremeniZaposlenik*. Neka klasa *Zaposlenik* izgleda kao u Primjeru 7.25.

Prema definiciji problema možemo zaključiti da ono što je definirano u klasi *Zaposlenik* važi za stalne kao i za privremene zaposlenike. Te dvije klase možemo definirati kao u Primjeru 7.26. U zagradama nakon naziva klase nalazi se naziv nadređene klase. Dakle, klasa *Zaposlenik* nadređena je objema ovim klasama koje time nasljeđuju od nje. Rekli bismo da su *StalniZaposlenik* i *PrivremeniZaposlenik* *izvedene* iz klase *Zaposlenik*. Sada možemo instancirati klasu *StalniZaposlenik* ovako:

```
>>> x = StalniZaposlenik('Mara', 'Maric', 8587,
                         Kategorije.VODITELJ_ODJELA)
>>> x
Mara Maric (8587) Kategorije.VODITELJ_ODJELA
>>> x.izracunaj_dohodak(26)
2200.0
>>> x.osnovni_dohodak
1000
>>> x.godisnji_odmor(15)
```

Primjer 7.25: Klasa *Zaposlenik*.

```
from enum import Enum

class Kategorije(Enum):
    RADNIK = 1,
    VODITELJ_TIMA = 2,
    VODITELJ_PROJEKTA = 3,
    VODITELJ_ODJELA = 4

class Zaposlenik:
    def __init__(self, ime, prezime, broj, kategorija):
        self.__ime = ime
        self.__prezime = prezime
        self.__broj = broj
        self.__kategorija = kategorija
        self.__OSNOVNI_DOHODAK = 1000  # eura

    @property
    def ime(self): return self.__ime

    @property
    def prezime(self): return self.__prezime

    @property
    def broj(self): return self.__broj

    @property
    def kategorija(self): return self.__kategorija

    @property
    def osnovni_dohodak(self):
        return self.__OSNOVNI_DOHODAK

    def izracunaj_dohodak(self, staz):
        match staz:
            case staz if 2 <= staz <= 5:
                koeficijent = 1.1
            case staz if 5 < staz <= 8:
                koeficijent = 1.2
            case staz if staz > 8:
                koeficijent = 1.5
            case _:
                koeficijent = 1

        match self.__kategorija:
            case Kategorije.VODITELJ_TIMA:
                koeficijent += 0.2
            case Kategorije.VODITELJ_PROJEKTA:
                koeficijent += 0.4
            case Kategorije.VODITELJ_ODJELA:
                koeficijent += 0.7

        return self.__OSNOVNI_DOHODAK * koeficijent

    def __repr__(self):
        return (f'{self.__ime} {self.__prezime} ({self.__broj}) '
                f'{self.__kategorija}')
```

Primjer 7.26: Klase *StalniZaposlenik* i *PrivremeniZaposlenik*.

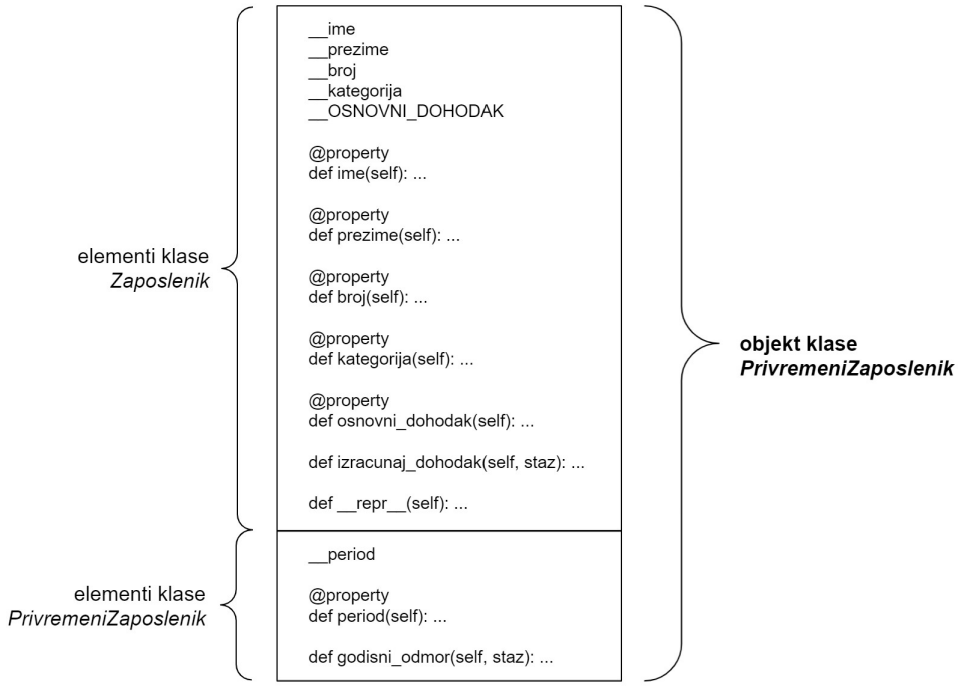
```
class StalniZaposlenik(Zaposlenik):
    def godisnji_odmor(self, staz):
        return 25 + staz // 5


class PrivremeniZaposlenik(Zaposlenik):
    def __init__(self, ime, prezime, broj, kategorija, period):
        Zaposlenik.__init__(self, ime, prezime, broj, kategorija)
        self.__period = period

    @property
    def period(self):
        return self.__period

    def godisnji_odmor(self, staz):
        return 20 + staz // 5
```

```
28
```

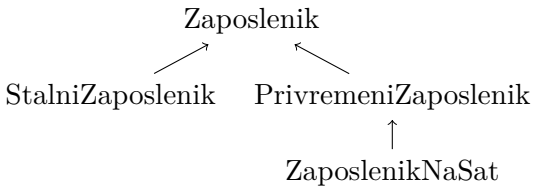
Vremensko razdoblje za privremene zaposlenike zadaje se kao par od dvaju objekata tipa *Datum*:

```
>>> p = PrivremeniZaposlenik('Ivo', 'Ivic', 2101,
                  Kategorije.VODITELJ_TIMA,
                  (datum.Datum(1, 1, 2021), datum.Datum(1, 7, 2021)))
>>> p
Ivo Ivic (2101) Kategorije.VODITELJ_TIMA
>>> p.izracunaj_dohodak(9)
1700.0
>>> p.period
(1.1.2021., 1.7.2021.)
```

Vidimo da smo za objekte klase *StalniZaposlenik* i *PrivremeniZaposlenik* mogli pozvati metodu *izracunaj_dohodak* iako ona nije eksplicitno definirana u toj klasi. Ta metoda u njoj nasljeđena je od nadređene klase *Zaposlenik*. Isto tako, klasa *StalniZaposlenik* nema konstruktor (metodu *___init___*) jer je u toj klasi i on nasljeđen od klase *Zaposlenik*. Konstruktor klase *StalniZaposlenik* nije potrebno definirati jer bi imao iste parametre kao i konstruktor klase *Zaposlenik*. Međutim, konstruktor klase *PrivremeniZaposlenik* treba parametar za vremensko razdoblje pa stoga za tu klasu moramo definirati konstruktor. Prvi red tog konstruktora je naredba `Zaposlenik.__init__(self, ime, prezime, broj, kategorija)`. To je eksplicitan poziv konstruktora nadređene klase. S obzirom da ovaj konstruktor nije nasljeđen od klase *Zaposlenik* bez ovog poziva polja te klase ne bi bila inicijalizirana. Ovo je lakše razumijeti ako objekt klase *PrivremeniZaposlenik* prikažemo kao na Slici 7.3 gdje vidimo da se sastoji od dva dijela: elementi klase *Zaposlenik* i elementi klase *PrivremeniZaposlenik*. Ta dva dijela možemo smatrati podobjektima ovog objekta.



Slika 7.3: Struktura objekta klase *PrivremeniZaposlenik*.



Slika 7.4: Hijerarhija zaposlenika sa zaposlenicima po satu.

Primjer 7.27: Klasa *ZaposlenikNaSat*.

```
1  class ZaposlenikNaSat(PrivremeniZaposlenik):
2      def __init__(self, ime, prezime, broj, satnica, period):
3          PrivremeniZaposlenik.__init__(self, ime, prezime, broj,
4                                        Kategorije.RADNIK, period)
5          self.__satnica = satnica
6
7      @property
8      def satnica(self):
9          return self.__satnica
10
11     def izracunaj_dohodak(self, br_sati, staz):
12         return br_sati * self.__satnica + staz * 100
```

Pretpostavimo da trebamo dodati i zaposlenike po satu. To su zaposlenici koji se tretiraju kao privremeni radnici, a njihov se dohodak računa po formuli *broj_sati * satnica + staž * 100* (staž je cijeli broj koji označava godine) i oni nemaju osnovni dohodak. S obzirom da i takvi zaposlenici imaju ime, prezime, broj, kategoriju i period zaposlenja čini se logično da definiramo klasu *ZaposlenikNaSat* tako da nasljeđuje od klase *PrivremeniZaposlenik* čime bismo dobili hijerarhiju klasa prikazanu na Slici 7.4.

U Primjeru 7.27 definirana je klasa *ZaposlenikNaSat*. Slično kao i s konstruktorom klase *PrivremeniZaposlenik*, u konstruktoru ove klase vidimo poziv `PrivremeniZaposlenik.__init__(ime, prezime, broj, Kategorije.RADNIK)`. S obzirom da zaposlenici po satu pripadaju kategoriji radnika, ovom naredbom želimo izbjeći da korisnici ove klase moraju specificirati kategoriju. Isto tako, ovdje nam treba i jedno novo polje, *satnica*, koje će sadržavati satnicu po kojoj je plaćena ova vrsta radnika. Sada informaciju o zaposleniku i njegovom dohotku možemo dobiti ovako:

```
>>> zs = ZaposlenikNaSat('Pero', 'Peric', 8587, 15,
                  (datum.Datum(1, 1, 2021), datum.Datum(1, 7, 2021)))
>>> zs
Pero Peric (8587) Kategorije.RADNIK
>>> zs.izracunaj_dohodak(4, 8)
860
>>> zs.period
(1.1.2021., 1.7.2021.)
```

Isto tako, možemo za ovakvog zaposlenika dobiti informaciju o osnovnom dohotku:

```
>>> zs.osnovni_dohodak
1000
```

Problem je, međutim, u tome što osnovni dohodak nije uopće definiran za zaposlenike po satu (barem ne u ovom primjeru)! Prema zahtijevima, njihov dohodak ovisi isključivo o broju odrađenih sati i radnom stažu. Ovo može biti problem ako je nekoj varijabli pridružen objekt ove klase, a programer nije svjestan da ona ne bi trebala imati metodu za osnovni dohodak već očekuje da je imaju sve vrste zaposlenika. Ovo je tipičan problem s nasljeđivanjem, kada se nova, nepredviđena klasa naizgled uklapa u postojeću hijerarhiju, ali joj neki postojeći elementi (polja i/ili metode) te hijerarhije ne odgovaraju. Ovdje postoji nekoliko mogućnosti:

1. u klasu *ZaposlenikNaSat* dodati metodu *osnovni_dohodak*, ali iz nje samo baciti iznimku

2. u klasu *ZaposlenikNaSat* dodati metodu *osnovni_dohodak* koja vraća 0

3. klasu *ZaposlenikNaSat* definirati izvan hijerarhije zaposlenika (da ne nasljeđuje od klase *PrivremeniZaposlenik* ili neke druge od ovih klasa), to jest zaposlenike po satu ne tretirati kao zaposlenike

4. iz klase *Zaposlenik* izdvojiti podatke o zaposleniku (ime, prezime, ...) i one vezane za zaposlenje u zasebne klase

5. ukloniti polje i metodu za osnovni dohodak iz klase *Zaposlenik*

6. implementirati klasu *ZaposlenikNaSat* kao klasu-omotač.

Mogućnost 1 više je improvizacija nego suvislo rješenje - ako neka metoda nema svrhe za određeni tip podatka, bolje je da ju on uopće nema. Mogućnost 2 prilično je logična s obzirom da nepostojanje osnovnog dohotka za zaposlenike na sat možemo tumačiti kao da je on 0. Za mogućnost 3 moramo pronaći način da iskoristimo postojeću funkcionalnost kao što je pristup osobnim podacima zaposlenika i podatka o periodu ili tu funkcionalnost duplicirati. Za mogućnost 4 trebali bismo metode za osobne podatke zaposlenika odvojiti od onih koji se odnose na zaposlenje kao što je *osnovni_dohodak* ili *izracunaj_dohodak*. Mogućnost 5 čini se najjednostavnijom gdje bismo podatak o osnovnom dohotku postavili negdje izvan klasa ove hijerarhije, primjerice kao konstantu ili podatak učitan iz baze podataka. Mogućnost 6 opisana je u sljedećem dijelu.

Općenito, postojanje neke metode u baznoj klasi (onoj u korijenu hijerarhije) znači da ona važi za sve podklase te bazne klase. Prema tome, smještanje metode *osnovni_dohodak* u klasu *Zaposlenik* podrazumijeva da **svi** zaposlenici imaju osnovni dohodak. Međutim, to je važilo u početku, ali proširenjem zahtjeva uvođenjem zaposlenika po satu nastao je problem. Po ovoj logici zaposlenici po satu ne bi trebali biti tretirani kao zaposlenici. Osnovni problem s nasljeđivanjem je u tome što ono trajno povezuje klase koje se nalaze u takvom odnosu. Primjerice, izdvajanje klase *ZaposlenikNaSat* iz hijerarhije u kojoj se nalazi zahtijevalo bi puno promjena u toj klasi. Iz tog razloga dizajn programa koji je baziran na nasljeđivanju nije fleksibilan.

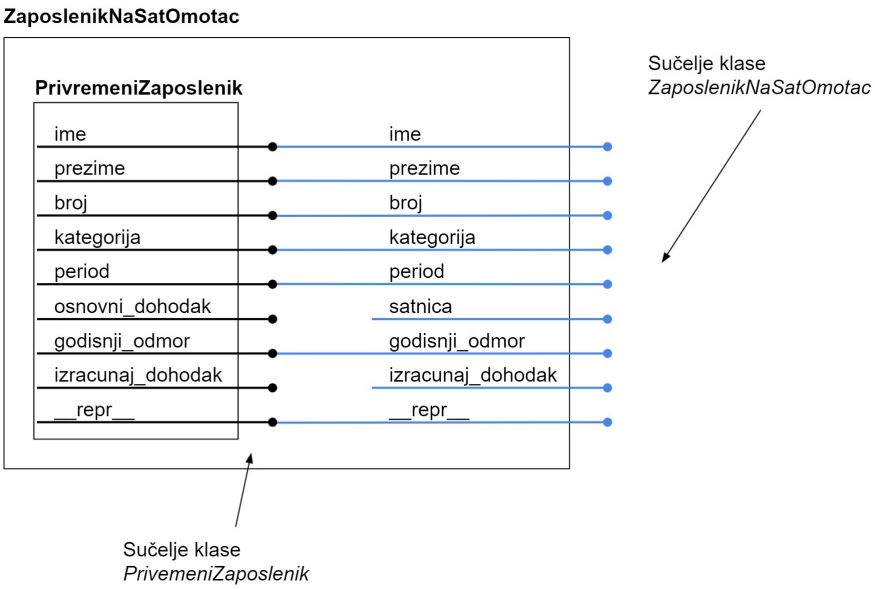
7.3.4 Klase omotači

Klase-omotači tehnika je programiranja zasnovana na kompoziciji objekata. U prethodnom dijelu implementirali smo klasu *ZaposlenikNaSat* upotrebom nasljeđivanja gdje joj je nadređena klasa bila *PrivremeniZaposlenik*. Isto tako, istaknuli smo osnovni nedostatak tog pristupa, a to je da neke metode nadređenih klasa, u ovom slučaju *osnovni_dohodak*, nisu definirane za zaposlenike po satu. Kod klasa-omotača ne koristimo nasljeđivanje nego klasu koja služi kao baza za funkcionalnosti koje nam trebaju. Nju instanciramo unutar klase omotača i po potrebi koristimo njene metode preko te instance.

U Primjeru 7.28 definirana je klasa-omotač za zaposlenike po satu. Vidimo da klasa *ZaposlenikNaSatOmotac* ne nasljeđuje od klase *PrivremeniZaposlenik* nego tu klasu instancira u svom konstruktoru. Ovo je ilustrirano na Slici 7.5. Skoro sve metode koje nam trebaju u klasi *ZaposlenikNaSatOmotac* pozivaju istu metodu u klasi *PrivremeniZaposlenik* dok metoda *izracunaj_dohodak* ima svoju implementaciju koja odgovara pravilima za zaposlenike na sat. Isto tako, tu je i metoda *satnica* koja postoji samo za ovu vrstu zaposlenika, a metoda *osnovni_dohodak* ovdje nije definirana. Na ovaj smo način definirali sučelje klase *ZaposlenikNaSatOmotac* neovisno o drugim klasama, što je jedna prednost klasa-omotača. Upotrebom omotača prilagodili smo pristup sučelju neke klase *PrivremeniZaposlenik* tako što smo tu klasu koristili neizravno kroz klasu *ZaposlenikNaSatOmotac*.

Primjer 7.28: Klasa-omotač *ZaposlenikNaSatOmotac*.

```
class ZaposlenikNaSatOmotac:
    def __init__(self, ime, prezime, broj, satnica, period):
        self.__priv = PrivremeniZaposlenik(ime, prezime, broj,
                                           Kategorije.RADNIK, period)
        self.__satnica = satnica

    @property
    def ime(self):
        return self.__priv.ime

    @property
    def prezime(self):
        return self.__priv.prezime

    @property
    def broj(self):
        return self.__priv.broj

    @property
    def kategorija(self):
        return self.__priv.kategorija

    @property
    def period(self):
        return self.__priv.period

    @property
    def satnica(self):
        return self.__satnica

    def godisnji_odmor(self, staz):
        return self.__priv.godisnji_odmor(staz)

    def izracunaj_dohodak(self, br_sati, staz):
        return br_sati * self.__satnica + staz * 100

    def __repr__(self):
        return self.__priv.__repr__()
```



Slika 7.5: Struktura omotača *ZaposlenikNaSatOmotac*.

Delegiranjem samo prenosimo poziv jedne metode na drugu, a da u prvoj metodi ne radimo bilo što drugo.

Nedostatak omotača je u tome što moramo definirati sve metode koje nam trebaju zajedno s onima koje samo prenose poziv na objekt druge klase. Na primjer, metoda *godisnji_odmor* klase *ZaposlenikNaSatOmotac* prenosi svoj poziv na istoimenu metodu klase *PrivremeniZaposlenik*. To se zove *delegiranje*. Obje metode imaju isti naziv, što je nebitno.

8 Sistemski alati

8.1 Datoteke

Python razlikuje dvije vrste datoteka:

- tekstualne i
- binarne. [33]

Tekstualne datoteke općenito mogu sadržavati Unicode ili ASCII znakove. Sadržaj takvih datoteka redovi su znakovnih nizova, a oznake za kraj reda automatski se ispravno interpretiraju i prevode.

Binarne datoteke sadrže 8-bitne vrijednosti i u Pythonu se mogu smjestiti u takozvane *byte-string*ove, to jest objekt tipa *bytes*. Za ovu vrstu datoteka ne provodi se automatsko prevođenje podataka kao što je kraj redka. Primjeri binarnih datoteka su zvučne datoteke, datoteke koje predstavljaju program (na primjer, .exe datoteke na sustavu Windows) i slike. Opće načelo rada s datotekama je sljedeće:

1. otvori datoteku
2. čitaj i/ili upisuj u datoteku
3. zatvori datoteku.

Zašto datoteku moramo otvoriti i zatvoriti? Otvaranje datoteke u stvari je upućivanje zahtjeva operacijskom sustavu (kao što su Windows, Linux i drugi) za resursom koji je u ovom slučaju datoteka. Taj sustav, pored ostalog, mora provjeriti ima li naš program pravo pristupa dotičnoj datoteci. Nadalje, operacijski sustav mora provjeriti koja prava naš program ima za tu datoteku, kao što je čitanje i pisanje u datoteku. Isto tako, operacijski sustav može pripremiti svojevrsna memorijska spremišta (engl. *buffer*) za datoteku da bi se rad s njom ubrzao. Na kraju, nakon što je datoteka spremna za korištenje, operacijski sustav daje korisničkom programu poveznicu (engl. *handle*) koja služi kao identifikator za tu datoteku. Nakon korištenja, datoteku treba zatvoriti. Zatvaranje datoteke omogućuje operacijskom sustavu prijenos svih preostalih podataka iz memorije u datoteku, oslobađanje memorije rezervirane za vođenje evidencije o datoteci

i sve druge radnje koje su neophodne u tom slučaju, zavisno od specifičnog operacijskog sustava. Nadalje, ako je neka datoteka otvorena operacijski sustav može zabraniti pristup toj datoteci drugom programu.

Kod otvaranja datoteke moramo navesti dvije stvari:

- ime datoteke
- svrhu za koju je otvaramo (čitanje i/ili pisanje).

Funkcija *open* otvara datoteku.

Datoteku otvaramo funkcijom *open* koja prima dva parametra: ime datoteke i svrhu otvaranja. Svrha otvaranja zadaje se kao znakovni niz. Rezultat ove funkcije objekt je koji predstavlja datoteku koja je navedena kao parametar. Da bismo otvorili datoteku potrebno je specificirati za koju svrhu je otvaramo, za što je zadužen drugi parametar. Taj parametar može biti sljedeće:

- "r" - otvaranje datoteke za čitanje od početka. Ovo je podrazumijevana postavka.
- "r+" - otvaranje datoteke za čitanje i pisanje od početka.
- "w" - otvaranje datoteke za pisanje. Prethodni se sadržaj datoteke (ako ta datoteka postoji) briše. U protivnom, kreira se nova datoteka.
- "w+" - otvaranje datoteke za čitanje i pisanje od početka. Prethodni sadržaj datoteke (ako ta datoteka postoji) se briše. U protivnom, kreira se nova datoteka.
- "a" - datoteka se otvara za pisanje od kraja. Ako već ne postoji, kreira se nova datoteka.
- "a+" - otvaranje datoteke za čitanje i pisanje od kraja. Ako već ne postoji, kreira se nova datoteka.

Funkcije *read*, *readline* i *readlines* čitaju tekstualnu datoteku.

Funkcija *read* učitava sadržaj cijele tekstualne datoteke u znakovni niz, zajedno s oznakama za kraj reda (koje ne moraju biti vidljive).

U sljedećem primjeru ispisuje se sadržaj datoteke *primjer.txt* koja sadrži tri reda: Prvi red Drugi red Treći red

```
1 with open('primjer.txt') as dat:
2     sadrzaj = dat.read()
3     print(sadrzaj)
```

Ovdje smo pomoću naredbe with otvorili datoteku *primjer.txt*. Rezultat funkcije *open* je objekt koji predstavlja datoteku, a naredba *with* u gornjem primjeru taj objekt pridružuje varijabli *dat*. Ovo smo mogli napisati i ovako:

```
dat = open('primjer.txt')
sadrzaj = dat.read()
print(sadrzaj)
dat.close()
```

Prednost naredbe *with* je u tome što će ona datoteku zatvoriti automatski nakon izvršenja posljednje naredbe u bloku. Uostalom, upotreba naredbe *with* u radu s datotekama idiomatski[1] je način rada s datotekama u Pythonu pa ćemo se toga pridržavati i u ovom udžbeniku [13].

Ako želimo pristupati sadržaju tekstualne datoteke red po red, od prvog prema posljednjem (s tim da možemo prekinuti kad je to potrebno), idiomatski način da se to postigne u Pythonu je upotrebom petlje *for*:

```
for redak in open('primjer.txt'):
    print(redak)
```

Iako ovakva petlja *for* na prvi pogled izgleda neobično (kao da pristupamo elementima nekakvog niza), radi se o tome da u Pythonu tekstualnu datoteku možemo promatrati kao niz redova, pa je u tom slučaju ovaj oblik naredbe *for* sasvim logičan. Još jedna prednost ovakvog čitanja tekstualne datoteke je da u ovom slučaju Python ne učitava cijelu datoteku u memoriju nego pristupa pojedinačnim redovima po potrebi, pa se time štedi i na memoriji. U ovom slučaju datoteka će biti zatvorena kada sakupljač smeća ukloni njen objekt (što najčešće nije odmah po završetku ovakve petlje).

Funkcija *readline* za svaki poziv vraća sljedeći redak datoteke:

```
>>> dat = open('primjer.txt')
>>> dat.readline()
'jedan\n'
>>> dat.readline()
'dva\n'
>>> dat.readline()
'tri\n'
>>> dat.readline()
''
>>> dat.readline()
''
```

Kada dođe do kraja datoteke ova funkcija počne vraćati prazan znakovni niz za svaki sljedeći poziv.

Kada se govori o datotekama jedno pitanje koje se postavlja je što ako datoteka koju želimo čitati ne postoji. Uzevši u obzir prethodni primjer, ako datoteka *primjer.txt* ne postoji Pythonov bi interpreter na pozivu funkcije *open* prekinuo izvršavanje programa i dojavio grešku:

[1] Tipičan ili općeprihvaćen stil.

```
Traceback (most recent call last):
  File "main.py", line 1, in <module>
    with open('primjer.txt') as dat:
FileNotFoundError: [Errno 2] No such file or directory:
'primjer.txt'
```

U drugom redu gornjeg ispisa vidimo da je do greške došlo u redu 1 (u kojem se nalazi poziv funkcije *open*) u kodu koji se nalazi u modulu *main.py*. Isto tako, u trećem redu vidimo i jedan red koda u kojem je došlo do greške.

Funkcije *write* i *writelines* upisuju novi sadržaj u datoteku.

Funkcijom *write* u datoteku upisujemo jedan znakovni niz, dok funkcijom *writelines* u datoteku upisujemo sve znakovne nizove zadane u listi. Na primjer, nakon izvršenja sljedećeg programa datoteka *primjer.txt* sadržavat će 01234:

```
with open('primjer.txt', 'w') as dat:
    for br in range(0, 5):
        sadrzaj = dat.write(str(br))
```

S obzirom da funkcija *write* prima samo znakovne nizove, numeričke vrijednosti varijable *br* moramo konvertirati u znakovni niz funkcijom *str*. U sljedećem primjeru upisujemo znakovne nizove jedan, dva i tri u istu datoteku, ali ovaj put znakovni nizovi smješteni su u listi:

```
lista = ['jedan', 'dva', 'tri']
with open('primjer.txt', 'w') as dat:
    sadrzaj = dat.writelines(lista)
```

Ovdje nam ne treba petlja jer funkcija *writelines* sama dodaje sve znakovne nizove iz liste. Nakon ovoga datoteka će sadržavati *jedandvatri*. Vidimo da su svi znakovni nizovi u istom redu. Ako ih želimo upisati u datoteku tako da je svaki u svom redu onda možemo na kraj svakog znakovnog niza dodati znak za novi red:

```
lista = ['jedan\n', 'dva\n', 'tri\n']
```

8.1.1 Primjeri

U ovom dijelu pokazano je nekoliko primjera rada s tekstualnim datotekama.

Primjer 1

Zadatak: Napišite program koji uklanja prazne redove iz tekstualne datoteke.

Riješenje: Ovaj zadatak možemo riješiti tako da prolazimo kroz datoteku red po red i za svaki provjeravamo je li prazan. Ako nije uključimo ga u rezultat. Ovaj postupak prikazan je funkcijom *ukloni_prazne_redove* u Primjeru 8.1. S obzirom da svaki učitani red ima znak za kraj reda na kraju ('\n'), taj znak možemo maknuti da bi nam ostali

Primjer 8.1: Funkcija *ukloni_prazne_redove*.

```
def ukloni_prazne_redove(ime_dat):
    with open(ime_dat, encoding='utf-8') as dat:
        for redak in dat:
            # makni znakove za kraj reda
            s = redak.replace('\n', '')
            if s != '':
                print(s)  # print ispisuje red po red
```

samo prepoznatljivi znakovi. Ako u nekom redu nije ostao niti jedan takav znak, taj red preskačemo.

Modul *argparse* olakšava rad s ulaznim parametrima.

Za pokretanje ovog programa na raspolaganju su nam dva načina unosa ulazne datoteke:

1. funkcijom *input* ili
2. parametrom u komandnoj liniji.

Ova druga mogućnost je praktičnija i za nju već postoji standardna biblioteka *argparse*. Ona nam omogućava da na jednostavan način dođemo do ulaznih parametara zadanih u komandnoj liniji i da u slučaju da neki od njih nedostaje dobijemo smislenu poruku. Naš program sada izgleda ovako:

```
import argparse

def ukloni_prazne_redove(ime_dat):
    ...

parser = argparse.ArgumentParser()
parser.add_argument(dest='ime_datoteke')
args = parser.parse_args()
ukloni_prazne_redove(args.ime_datoteke)
```

Naredbom *import* učitan je modul u kojem se nalazi kôd koji želimo uključiti u naš program. Metodom *add_argument* dodajemo ime argumenta koji će sadržavati vrijednost ulaznog parametra, nakon čega do njega možemo doći kako je pokazano u zadnjem redu. Detalji ove biblioteke mogu se naći na mrežnoj stranici Pythona *https://www.python.org/*. Ako ovaj program spremimo u datoteku *prazni_redovi.py*, a ulazna datoteka se zove *tekst.py*, možemo ga pokrenuti na sljedeći način:

```
...> python prazni_redovi.py tekst.txt
```

U ovom slučaju izraz *args.ime_datoteke* daje znakovni niz „tekst.txt“.

Operatorom ”>” komandne linije ispis na ekranu možemo preusmjeriti u datoteku.

U ovom primjeru treba primijetiti da ulaznu datoteku u stvari ne modificiramo nego samo ispisujemo učitane redove na konzolu. Rezultat ovog programa možemo lako spremiti u neku datoteku upotrebom redirekcije komandne linije kao što su powershell [34], bash [35] i drugi. Ako smo gornji program spremili kao *prazni_redovi.py* onda taj program možemo pokrenuti kao

```
$ python prazni_redovi.py tekst.txt > rezultat.txt
```

Znak „>“ označava da ispis programa umjesto na konzolu želimo preusmjeriti u datoteku pod imenom *rezultat.txt*. Ova je mogućnost implementirana na većini operacijskih sustava, uključujući Windows, Linux i MacOS. Zadavanje parametara i preusmjeravanje ispisa fleksibilan je način korištenja programa jer omogućuje automatizaciju ovakvih radnji pisanjem skriptova komandne linije.

Primjer 2

Zadatak: Napišite program koji će u zadanoj datoteci koja sadrži rezervirana mjesta zamijeniti svako to mjesto sadržajem koji je zadan kao parametar. Na primjer, datoteka tekst.txt može sadržavati tekst „Dana <dan> održat će se izlaganje na temu <tema>.“. Ako ovaj program pokrenemo sa

```
...> python predlosci.py tekst.txt dan='28.8.2020.' tema='Klimatske
    promjene'
```

Ovaj program sada bi trebao ispisati „Dana 28.8.2020. održat će se izlaganje na temu Klimatske promjene.“

Rješenje: Ovdje samo trebamo u tekstu pronaći dijelove koji sadrže ključ, gdje je ključ dio parametra *ključ=vrijednost* kao u gornjem primjeru. Ovaj se program, prema tome, može napisati upotrebom funkcije *replace* kako je pokazano u Primjeru 8.2. U naredbi `parser.add_argument(dest='parametri', nargs='+')`, `nargs='+'` znači da nakon naziva datoteke mora uslijediti jedan ili više drugih parametara, što nam je ovdje nužno jer datoteka može imati više predložaka koje treba zamijeniti nekom vrijednošću, kao što je pokazano u opisu zadatka.

Primjer 3

Zadatak: Napišite program koji spaja dvije CSV datoteke tako da mu se zada indeks polja po kojem spojene datoteke trebaju biti sortirane, te datoteke čiji sadržaj treba spojiti. Na primjer, neka datoteka *dat1.txt* sadrži

```
Ivo,Ivic,0991112222
Pero,Peric,0962223333
Mara,Maric,0958887777
```

Neka datoteka *dat2.txt* sadrži

Primjer 8.2: Program za zamjenu sadržaja u datoteci.

```
import argparse

def zamijeni(args):
    parametri = {}
    for kljuc in args.parametri:
        ime, vr = tuple(kljuc.split('='))
        parametri[ime] = vr

    for redak in open(args.datoteka,
                      encoding='utf8'):
        for kljuc in parametri:
            polje = '{{' + kljuc + '}}'
            if polje in redak:
                redak = redak.replace(
                            polje,
                            parametri[kljuc])
        print(redak, end='')

parser = argparse.ArgumentParser()
parser.add_argument(dest='datoteka')
parser.add_argument(dest='parametri', nargs='+')
args = parser.parse_args()
zamijeni(args)
```

```
Stjepan,Ivanic,0937637465
Toni,Alic,0912345678
Ante,Antic,0998887777
Miro,Maric,0945556666
Ana,Anic,0972227777
```

Pokretanjem ovog programa sa

```
...> python spajanje_csv_datoteka.py 1 dat1.txt dat2.txt
```

treba ispisati

```
['Toni', 'Alic', '0912345678\n']
['Ana', 'Anic', '0972227777\n']
['Ante', 'Antic', '0998887777\n']
['Stjepan', 'Ivanic', '0937637465\n']
['Ivo', 'Ivic', '0991112222\n']
['Mara', 'Maric', '0958887777\n']
['Miro', 'Maric', '0945556666\n']
['Pero', 'Peric', '0962223333\n']
```

Vidimo da je popis sortiran po zadanom indeksu 1, odnosno elementu na tom indeksu.

Primjer 8.3: Program za spajanje CSV datoteka.

```
import argparse


def umetni(redovi, redak, polje):
    for i, v in enumerate(redovi):
        if redak[polje] < v[polje]:
            redovi.insert(i, redak)
            return
    redovi += [redak]


def spoji(polje, datoteke):
    sadrzaj = []
    for ime in datoteke:
        with open(ime, encoding='utf-8') as dat:
            sadrzaj += dat.readlines()

    redovi = []
    for redak in sadrzaj:
        umetni(redovi, redak.split(','), polje)

    for s in redovi:
        print(s)


parser = argparse.ArgumentParser()
parser.add_argument(dest='polje', type=int)
parser.add_argument(dest='datoteke', nargs='+')
args = parser.parse_args()
spoji(args.polje, args.datoteke)
```

Rješenje: Prvo možemo učitati cijeli sadržaj svih zadanih datoteka, a zatim svaki red umetnuti na pravo mjesto kako je pokazano u Primjeru 8.3. Funkcija umetni prolazi kroz sve redove prethodno umetnute u listu redovi i umeće novi red na mjesto gdje je prethodna vrijednost zadanog polja manja od njegove, a sljedeća veća ili jednaka.

8.2 Povezivanje programa u komandnoj liniji

Jedna izuzetno korisna mogućnost operacijskog sustava Unix bila je uvođenje operatora „|“ (engl. *pipe*) [36] kojim se rezultat jednog programa može zadati kao ulazni podatak za drugi. Na sreću, danas je taj operator na raspolaganju u sva tri popularna operacijska sustava – Windows, Linux i MacOS. Linux i MacOS inačice su sustava Unix pa je njihova komandna linija velikim dijelom kompatibilna s onom u Unixu. Iako Windows ima taj operator kao dio programa PowerShell, za ovaj se sustav može instalirati i program

Cygwin ili Git Bash koji simuliraju komandnu liniju sustava Unix, tako da je znanje koje se tu stekne prenosivo na sva tri operacijska sustava. Nadalje, od verzije 6 PowerShell (trenutno PowerShell Core) na raspolaganju je za sva tri operacijska sustava pa je i to znanje prenosivo.

Operatorom "|" komandne linije rezultat jednog programa može se zadati kao ulazni podatak za drugi program.

U ovom dijelu prikazan je primjer povezivanja programa pisanih u Pythonu ovim operatorom i to u kombinaciji s nekim drugim programima koji su dostupni kroz komandnu liniju. Pretpostavimo da nam treba program koji će raditi sljedeće:

- Pregled telefonskih brojeva i duljine razgovora s tih brojeva kao broj minuta. Podaci su spremljeni u CSV datoteku u sljedećem formatu: *<broj telefona>,<broj minuta>*. Na primjer:

  ```
  0951234567,44
  0995556666,71
  0912223333,105
  ...
  ```

  Nadalje, program treba omogućiti filtriranje po broju minuta, tako da se mogu zadati sljedeći kriteriji: maksimalan broj minuta, minimalan broj minuta i raspon broja minuta.

- Konverzija podataka gornjeg formata u JSON. Na primjer, prvi red gornjeg primjera bio bi konvertiran u "telefon": "0951234567", „minuta": 44. Ova konverzija treba obuhvatiti samo one podatke koji su izdvojeni filterom ili sve podatke ako filter nije zadan.

Sada imamo dvije mogućnosti:

1. Napisati jedan program s parametrima pomoću kojih možemo zadati filter i po potrebi neke druge podatke. Taj bi program onda rezultat spremio u neku zadanu datoteku.
2. Napisati dva programa koje možemo povezati operatorom „|" komandne linije.

Iako prva mogućnost nije loša, druga nam mogućnost daje veću fleksibilnost. Na primjer, ako kasnije želimo filtrirati podatke iz datoteke CSV po nekom drugom kriteriju kao što je pozivni broj mobilne mreže, što je već riješeno programom *grep* u komandnom jeziku Bash ili regularnim izrazima u komandnom jeziku PowerShell. Isto tako, ako rezultat selekcije telefonskih brojeva ili konverzije u JSON ispisujemo na standardni

izlaz (naredbom *print*) taj rezultat možemo dalje proslijediti nekom drugom programu komandne linije, kao što je *sort*. Na taj način programe možemo kombinirati s drugim programima komandne linije čime bismo iskoristili ono što već postoji i dobili fleksibilnost koju nude takve razne kombinacije. Isto tako, ovakav način nam olakšava testiranje programa jer se ulazni podaci mogu zadati na više načina:

- preko tipkovnice
- pomoću naredbe echo
- iz datoteke naredbom cat
- iz nekog drugog programa.

U nastavku ćemo proučiti ovakav način rada na primjeru dva programa kojima ćemo implementirati gore opisanu funkcionalnost.

8.2.1 Primjer s dva programa

Ovdje su prikazana dva programa kojima će biti demonstrirana upotreba operatora "|":

1. program za pregled telefonskih brojeva kojim pregledavamo popis telefonskih brojeva po nekom zadanom kriteriju
2. program za konverziju u JSON format koji popis telefonskih brojeva i minuta konvertira u JSON.

Program za pregled telefonskih brojeva

Ovaj program pokretat ćemo na sljedeći način:

```
1 > python tel_brojevi.py n+|n-|a..b
```

Parametar *n+* postavlja filter za trajanje *n* ili više minuta, dok parametar *n-* postavlja filter za trajanje *n* ili manje minuta. Parametar *a..b* postavlja filter za trajanje unutar intervala *[a, b]* minuta. Ovaj program možemo implementirati na način prikazan u Primjeru 8.4. Ovdje za svaku vrstu filtera napravimo funkciju koja uspoređuje vrijednost minuta iz pojedinog ulaznog redka sa zadanim ograničenjem. Nakon toga, unutar petlje *while* čitamo red po red i pozivamo funkciju pridruženu varijabli *uvjet* (koja predstavlja filter). Ako ta funkcija vrati *True* taj se redak šalje na standardni izlaz, što je ekran, naredbom *print*. U petlji *while* vidimo da se čitanje podataka obavlja unutar naredbe *try/except*. Razlog tome je što funkcija *input* signalizira grešku tipa *EOFError* kada dođe do kraja ulaznih podataka, gdje u tom slučaju prekidamo ovu petlju i izvršavanje programa. Ovaj program sada možemo koristiti na ovaj način:

```
> cat .\tel.txt | py .\tel_brojevi.py 100+
0988765432,117
```

Primjer 8.4: Program za filtriranje telefonskih brojeva.

```
import argparse

def filtriraj(args):
    uvjet = None
    if '..' in args.filter:
        interval = args.filter.split('..')
        a = int(interval[0])
        b = int(interval[1])
        uvjet = lambda m: a <= m <= b
    elif args.filter[-1] == '+':
        n = int(args.filter[:-1])
        uvjet = lambda m: m > n
    elif args.filter[-1] == '-':
        n = int(args.filter[:-1])
        uvjet = lambda m: m < n
    while True:
        try:
            unos = input()
        except EOFError:  # detektiran kraj
            break
        else:
            minute = int(unos.split(',')[1])
            if uvjet(minute):
                print(unos)

parser = argparse.ArgumentParser()
parser.add_argument(dest='filter')
args = parser.parse_args()

try:
    filtriraj(args)
except Exception as e:
    print('??? Greska --', __file__)
```

Primjer 8.5: Program za konverziju u JSON format.

```
import json

def napravi_json():
    while True:
        try:
            unos = input()
        except EOFError:  # detektiran kraj
            break
        else:
            tel, minute = unos.split(',')
            s = json.dumps({'telefon': tel,
                            'minuta': int(minute)})
            print(s)

try:
    napravi_json()
except Exception:
    print('??? Greska --', __file__)
```

```
0995556666,109
0932223333,174
0953092876,192
```

S obzirom da *tel_brojevi.py* očekuje ulazne podatke na standardnom ulazu u gornjem primjeru te smo mu podatke poslali programom (komandom) *cat*.

Funkcija *filtriraj* također se poziva unutar naredbe *try/except*. Možemo primijetiti da u ovoj funkciji ne provjeravamo jesu li parametri zadani ispravno. Na primjer, ako ovaj program pokrenemo s

```
> py tel_brojevi.py 100..
```

bit će signalizirana iznimka negdje u funkciji *filter* jer će nedostajati znak „+“ ili „-„ nakon broja, pa će negdje unutar naredbe *if* doći do iznimke prilikom indeksiranja niza ili će varijabla *uvjet* ostati *None* jer niti jedan uvjet naredbe *if* neće biti zadovoljen. Na ovaj način možemo barem ispisati poruku da je došlo do greške. Varijabla *__file__* sadrži ime datoteke (*.py*) u kojem se nalazi kôd koji se trenutno izvršava.

Program za konverziju u JSON format

Ovaj program radi po istom načelu kao i prethodni, ali je još jednostavniji jer samo podatak oblika <telefon>,<minuta> ispisuje u JSON formatu upotrebom modula json (Primjer 8.5). Za ulazni znakovni niz kao što je `„0951112222,45"` ovaj program ispisat će `„telefon": „0951112222", „minuta": 45`. Ovaj program bit će spremljen u datoteku *json_format.py*.

Povezivanje programa operatorom "|"

Gdje unosimo podatke u prethodno prikazanim programima? Ako pokrenemo program *json_format.py* sa

```
> py json_format.py
```

program će čekati na naš unos podataka preko tipkovnice. Ako sada unesemo neke podatke, kao `123,45` program će ispisati

```
{"telefon": "123", "minuta": 45}
```

i čekati na sljedeći unos. Na ovaj način možemo lako testirati program s nekoliko podataka, ali za unos veće količine podataka taj način nije praktičan. Na primjer, što ako želimo da ovaj program radi na oba načina: unos preko tipkovnice (za brzo testiranje) i unos iz datoteke (za veću količinu podataka)? To možemo lako dodati u sam program, ali način na koji je on napisan već omogućava takav način rada - upotrebom operatora "|".

Pretpostavimo da smo sljedeće podatke spremili u datoteku *brojevi.txt*:

```
0951234567,21
0988765432,117
0995556666,109
0932223333,174
0979991111,12
0983450987,44
0952999922,8
0953092876,192
```

U komandnoj liniji možemo dobiti sadržaj te datoteke naredbom *cat*:

```
> cat brojevi.txt
0951234567,21
0988765432,117
0995556666,109
0932223333,174
0979991111,12
0983450987,44
0952999922,8
0953092876,192
```

Naredba *cat* ispisala je sadržaj ove datoteke na ekran ili konzolu zato jer je to standardni izlazni uređaj. Ovaj smo ispis isto tako mogli preusmjeriti na neku postojeću ili nepostojeću datoteku upotrebom operatora ">":

```
> cat brojevi.txt > ispis.txt
```

Sada će datoteka *ispis.txt* imati isti sadržaj kao i datoteka *brojevi.txt* jer smo ispis datoteke *brojevi.txt* (što je u biti cijeli sadržaj te datoteke) preusmjerili na datoteku *ispis.txt*.

Sadržaj tekstualne datoteke možemo proslijediti i nekom programu. To možemo zamisliti kao da smo pokrenuli program *json_format.py*, ali umjesto da svaki red unosimo pojedinačno preko tipkovnice ti redovi dolaze jedan po jedan iz neke tekstualne datoteke kao što je *brojevi.txt*. Upravo to možemo dobiti upotrebom operatora "|":

```
> cat .\brojevi.txt | py .\json_format.py
{"telefon": "0951234567", "minuta": 21}
{"telefon": "0988765432", "minuta": 117}
{"telefon": "0995556666", "minuta": 109}
{"telefon": "0932223333", "minuta": 174}
{"telefon": "0979991111", "minuta": 12}
{"telefon": "0983450987", "minuta": 44}
{"telefon": "0952999922", "minuta": 8}
{"telefon": "0953092876", "minuta": 192}
```

Ovdje umjesto da naredba *cat* ispiše sadržaj datoteke *brojevi.txt* na konzolu, operacijski sustav taj sadržaj šalje dalje programu koji je zadan desno od operatora "|", što je u ovom slučaju Pythonov interpreter. Kada se pokrene program *json_format.py* funkcija *input* učitava ulazni redak teksta koji je došao preko operatora "|" i s tim tekstom radi na isti način kao i da je unesen preko tipkovnice. S obzirom da se ulazni tekst učitava u petlji, ovaj će program učitavati sve redke teksta koji dolaze iz datoteke *brojevi.txt*, sve dok ne naiđe na oznaku za kraj datoteke, nakon čega se petlja *while* prekida, čime završava izvršavanje programa *json_format.py*.

Sada naša dva programa, *tel_brojevi.py* i *json_format.py*, možemo povezati na isti način:

```
> cat .\tel.txt | py .\tel_brojevi.py 100+ | py .\json_format.py
{"telefon": "0988765432", "minuta": 117}
{"telefon": "0995556666", "minuta": 109}
{"telefon": "0932223333", "minuta": 174}
{"telefon": "0953092876", "minuta": 192}
```

Vidimo da smo, kao i prethodno, dobili brojeve telefona s razgovorima duljim od 100 minuta. Međutim, ovdje smo ih dobili u formatu JSON jer smo rezultat programa *tel_brojevi.py* proslijedili programu *json_format.py*.

9 Rekurzija

Rekurzija je jedna od temeljnih tehnika programiranja koja je korisna za rješavanje mnogih vrsta računalnih problema. Kod osnovnog oblika rekurzije funkcija poziva samu sebe. Rekurziju možemo promatrati kao jedan oblik iteracije odnosno ponavljanja, ali pri tome moramo uzeti u obzir neke karakteristike takvog pristupa [37, 38]. U ovom dijelu proučit ćemo osnovna načela rekurzije i vidjeti neke primjere u kojima je ona primjenjiva.

9.1 Rekurzivne funkcije

S obzirom da su funkcije osnovni elementi gotovo svakog programa, one kao takve određuju zakonitosti o tome kako se neki računalni proces odvija na razini tih funkcija. One određuju kako se jedna faza tog procesa nadograđuje na prethodnu. U programiranju postoje dva osnovna načina kojima se definira takva evolucija nekog procesa: iteracija (petlje) i rekurzija. Do sada smo petlje koristili kao osnovni mehanizam za iteraciju, odnosno ponavljanje jednog niza naredbi. Rekurzija se, kao i iteracija, također odnosi na ponavljanje, ali se za to ne koristi petlja nego rekurzivni poziv funkcije. Funkcija koja u svojoj definiciji sadrži jedan ili više takvih poziva zove se *rekurzivna funkcija*:

```
def rek(x):
    ...
    ... rek(n) ...
    ...
```

> Rekurzivna funkcija je ona koja poziva samu sebe.

Općenito, rekurzivna funkcija u osnovnom obliku je ona koja poziva samu sebe s jednog ili više mjesta unutar svoje definicije. Rekurzija, međutim, ne mora biti izravna kao u gornjem primjeru:

```
def f(x):
    ...
    ... g(n) ...
    ...
```

Primjer 9.1: Iterativno izračunavanje faktorijela.

```
def fakt_iter(n):
    rezultat = 1
    while n > 0:
        rezultat *= n
        n -= 1

    return rezultat

>>> fakt_iter(4)
24
```

```
def g(x):
    ...
    ... f(m) ...
    ...
```

U ovom primjeru funkcija f poziva funkciju g, a funkcija g poziva funkciju f. Ovakve se funkcije nazivaju *uzajamno rekurzivnim*.

> Svaka rekurzivna funkcija može se napisati iterativno (bez rekurzije).

Svaki postupak koji se može definirati iteracijom može se također definirati rekurzijom i obratno. Oba ova načina imaju prednosti i nedostatke, a koji od njih treba koristiti zavisi, kao uvijek, o prirodi problema koji rješavamo.

Za usporedbu iteracije i rekurzije proučit ćemo jednostavnu matematičku funkciju koja se zove *faktorijel* i daje umnožak svih (cijelih) brojeva u intervalu $1..n$ gdje je n zadan kao parametar. Ova se operacija u matematici označava s $n!$ i definira na sljedeći način:

$$n! = n * (n-1) * (n-2) * (n-3) * ... * 1.$$

Na primjer, 5! = 4 * 3 * 2 * 1 = 24. Iterativna inačica funkcije, koju ćemo zvati *fakt_iter*, može se napisati kao u Primjeru 9.1. Ako u svakom koraku zamijenimo varijable *rezultat* i n s njihovom vrijednošću u tom koraku možemo jasno vidjeti kako smo došli do rezultata:

```
1. rezultat = 1 (početna vrijednost)
2. rezultat = 1 * 4, n = 4
3. rezultat = 4 * 3, n = 3
4. rezultat = 12 * 2, n = 2
5. rezultat = 24 * 1, n = 1
```

Primjer 9.2: Rekurzivno izračunavanje faktorijela.

```
1 def fakt_rek(n):
2     if n == 0:  # uvjet zavrsetka rekurzije
3         return 1
4     else:
5         return n * fakt_rek(n - 1)
6
7 >>> fakt_rek(4)
8 24
```

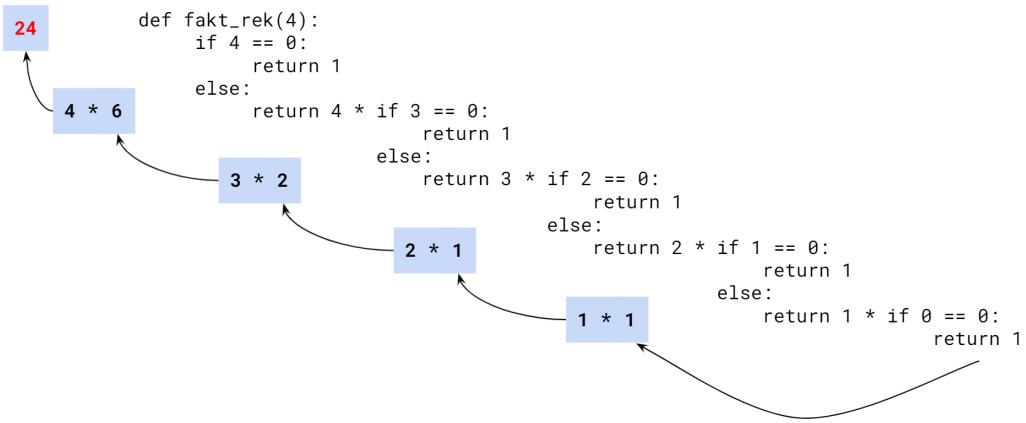
Vidimo da se ovaj postupak odvija tako da je u svakom koraku vrijednost varijabli rezultat i n bliža nekoj konačnoj vrijednosti. Isto tako, uvjet petlje n > 0 osigurava njegovo prekidanje kada n dosegne vrijednost 0. Faktorijel se često definira i na ovaj način:

$$n! = \begin{cases} 1, & \text{za } n = 0 \\ n(n\text{-}1)!, & \text{za } n > 0 \end{cases}$$

U drugom retku vidimo da je u ovoj definiciji faktorijel definiran pomoću faktorijela, ali za manju vrijednost, odnosno za *n - 1*. Ovo se naziva *rekurzivnom definicijom* i za nju možemo napisati rekurzivnu inačicu ove funkcije, koju ćemo zvati *fakt_rek* i koja je prikazana u Primjeru 9.2. Prvo što ovdje treba primijetiti je da u ovoj funkciji ne koristimo petlju nego pozivamo funkciju *fakt_rek* unutar te iste funkcije! Ovdje se odmah možemo pitati čime množimo *n* ako svaki put idemo na početak ove funkcije baš kada nam treba neki broj. Rekurzivni poziv funkcije odvija se na potpuno isti način kao i svaki drugi poziv, odnosno ne postoji nikakav posebni mehanizam koji je ugrađen u Pythonov interpreter i koji bi se koristio za rekurzivne funkcije. Ovaj rekurzivni poziv *fakt_rek(n - 1)* zapravo daje umnožak brojeva u intervalu 1 .. n - 1 koji onda pomnožimo sa *n*. Drugim riječima, konačni rezultat dobijemo tako da svaki broj *k* pomnožimo s umnoškom brojeva na manjem intervalu, 1 .. k - 1. Kao što smo iterativnu inačicu ove funkcije prikazali kao jedan niz naredbi koji se ponavlja više puta, njenu rekurzivnu inačicu prikazujemo kao jedan izraz koji se sastoji od više podizraza. Funkciju *fakt_rek* možemo zamisliti tako da na svako mjesto gdje se nalazi *fakt_rek(n - 1)* stavimo cijelu definiciju ove funkcije s tim da varijablu *n* zamijenimo njenom trenutnom vrijednošću, kako je pokazano na Slici 9.1.

Ako izdvojimo samo izraz iza naredbe *return* dobili bismo sljedeći niz koraka:

```
1. return 4 * fakt_rek(4 - 1)
2. return 4 * 3 * fakt_rek(3 - 1)
3. return 4 * 3 * 2 * fakt_rek(2 - 1)
4. return 4 * 3 * 2 * 1 * fakt_rek(1 - 1)
5. return 4 * 3 * 2 * 1 * 1
6. return 4 * 3 * 2 * 1
```



Slika 9.1: Rekurzivno izračunavanje faktorijela.

```
7. return 4 * 3 * 2
8. return 4 * 6
9. return 24
```

U svakom koraku imamo isti izraz: *n * fakt_rek(n - 1)*. U prvom koraku, prema tome, rješavamo početni izraz *fakt_rek(4)*, čiji je rezultat *4 * fakt_rek(4 - 1)*. U sljedećem koraku rješavamo izraz iz prethodnog koraka, odnosno *fakt_rek(3)*, čiji je rezultat *3 * fakt_rek(3 - 1)*. Ovdje, međutim, ne možemo zaboraviti na 4 iz prethodnog koraka jer s tom vrijednošću moramo pomnožiti rezultat ovog idućeg koraka, to jest rezultat poziva *fakt_rek(4 - 1)*, što je *3 * fakt_rek(3 - 1)*, i tako dalje. Množenje brojem 4 i drugim brojevima koji slijede do koraka 5 je odgođeno. Tek kada se dosegne korak 5 mogu se početi množiti vrijednosti jer su tek tada sve one na raspolaganju. Izraz *if n == 0: return 1* osigurava prekidanje ovog procesa kada *n* dosegne vrijednost 0. To se vidi nakon koraka 4, gdje više ne pozivamo funkciju *fakt_iter* sa sljedećim manjim argumentom jer smo "došli do kraja", to jest došli smo do *fakt_iter(0)* za koji znamo da je 1, pa prema tome nema potrebe dalje pozivati ovu funkciju. Nakon toga, u koraku 5 imamo kompletan izraz množenja koji se postupno izračunava do konačnog rezultata. Koracima 5 do 8 vraćamo se iz svih ovih prethodnih poziva - korakom 5 vraćamo se iz poziva u koraku 4, korakom 6 iz poziva u koraku 3, korakom 7 iz poziva u koraku 2, i na kraju korakom 8 vraćamo se iz poziva u prvom koraku čime dobivamo rezultat ove funkcije.

Da bismo promatrali izvršavanje ove funkcije možemo je napisati odvajanjem rekurzivnog poziva funkcije od množenja te ispisivanjem vrijednosti varijable *n* prije i poslije svakog poziva. Svaku ćemo vrijednost od *n* prije poziva sačuvati u jednoj tekstualnoj varijabli tako da se vidi čime množimo rezultat tog poziva. Funkciju, koju ćemo zvati *fakt_rek_koraci*, možemo napisati kako je pokazano u Primjeru 9.3. Funkcija *fakt_rek_koraci* radi na isti način kao i *fakt_rek* samo što ovdje rezultat rekurzivnog poziva smještamo u varijablu prije nego što ga pomnožimo varijablom *n*.

Primjer 9.3: Rekurzivno izračunavanje faktorijela s ispisom koraka.

```
def fakt_rek_koraci(n, s):
    if n == 0:
        return 1
    else:
        s += str(n) + ' * '
        print(s, 'fakt_rek_koraci(', n - 1, ')')
        t = fakt_rek_koraci(n - 1, s)
        print(s, t)
        r = n * t
        return r

>>> fakt_rek_koraci(4, '')
4 *  fakt_rek_koraci( 3 )
4 * 3 *  fakt_rek_koraci( 2 )
4 * 3 * 2 *  fakt_rek_koraci( 1 )
4 * 3 * 2 * 1 *  fakt_rek_koraci( 0 )
4 * 3 * 2 * 1 *  1
4 * 3 * 2 *  1
4 * 3 *  2
4 *  6
24
```

Sada se vidi razlika između iterativnog i rekurzivnog procesa. Rad iterativne funkcije bio je baziran na postupnim promjenama vrijednosti varijabli *rezultat* i *n* - varijablu *rezultat* povećavali smo u svakom koraku tako da smo je pomnožili varijablom *n*, a varijabla *n* nam je istovremeno služila kao brojač koraka. Kaže se da su *rezultat* i *n* varijable stanja ovog procesa. To znači da je u svakom koraku njegov napredak bio određen tim dvjema varijablama. Općenito, iterativni proces je onaj čije je stanje u svakom koraku određeno varijablama stanja i pravilom kako se mijenjaju vrijednosti tih varijabli iz jedne iteracije u iduću. Kod našeg primjera možemo reći da svaki put nakon što smo varijablu *rezultat* postavili na *rezultat * n* i varijablu *n* na *n - 1*, ovaj proces je prešao na sljedeći korak. Rekurzivna funkcija izvršavala se tako da se svaka vrijednost i operacija (kao što je množenje) morala pamtiti da bismo na kraju izračunali konačni rezultat. Operacije množenja ovdje su bile "odgođene" dok nismo dobili potpuni izraz. Ovo odgađanje operacija nije ništa drugo nego određivanje vrijednosti argumenata koje mora prethoditi pozivu dotične funkcije. S obizrom da je sam argument rezultat te iste funkcije dobivamo ovaj tipični kružni proces koji eventualno završava nekom konkretnom vrijednošću za taj argument (kao što je u našem primjeru broj 1 kada je *n* došao do 0). To se vidi na gornjem nizu koraka kao izraz koji se širi nakon čega slijedi skupljanje (kao posljedica vraćanja iz poziva funkcije) i takav je oblik tipičan za rekurzivne procese. Općenito, proces koji je baziran na takvom nizu odgođenih funkcija zove se *rekurzivan proces*.

Primjer 9.4: Rekurzivno izračunavanje Fibonaccijevog niza.

```
def fib_rek(n):
    if n in {0, 1}:
        return n
    else:
        return fib_rek(n - 1) + fib_rek(n - 2)

>>> fib_rek(4)
3
```

Rekurzivne funkcije mogu biti kompleksnije, gdje se ista funkcija može koristiti kao argument na više mjesta unutar jednog izraza. Za primjer proučit ćemo još jednu matematičku operaciju koja izračunava vrijednosti takozvanog Fibonaccijevog niza [39] koji izgleda ovako:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, ...$$

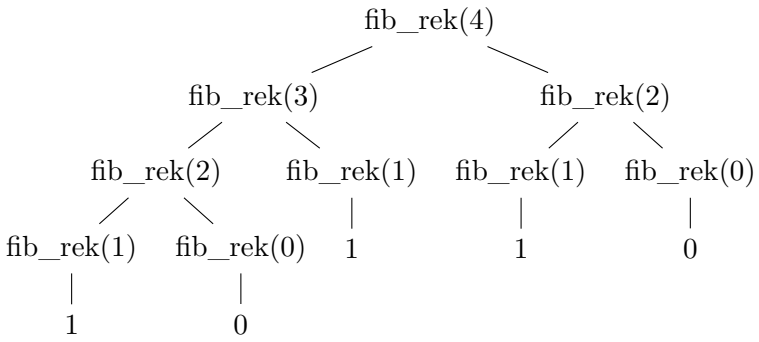
Ovaj niz počinje brojevima 0 i 1, a svaki idući broj zbroj je prethodna dva. Prema tome, funkcija *fib(n)* kojom se izračunava vrijednost ovog niza na položaju n (gdje je 0 početni položaj) može se matematički definirati ovako:

$$fib(n) = \begin{cases} 0, & \text{za } n = 0 \\ 1, & \text{za } n = 1 \\ fib(n\text{-}1) + fib(n\text{-}2), & \text{za } n > 1 \end{cases}$$

U Primjeru 9.4 ova je definicija prevedena u Python. Ovaj proces sada možemo prikazati na isti način kao u prethodnom primjeru za izraz *fib_rek(4)*:

```
1. return fib_rek(4 - 1) + fib_rek(4 - 2)
2. return fib_rek(3 - 1) + fib_rek(3 - 2) + fib_rek(4 - 2)
3. return fib_rek(2 - 1) + fib_rek(2 - 2) + fib_rek(3 - 2)
                                                + fib_rek(4 - 2)
4. return 1 + fib_rek(2 - 2) + fib_rek(3 - 2) + fib_rek(4 - 2)
5. return 1 + 0 + fib_rek(3 - 2) + fib_rek(4 - 2)
6. return 1 + 0 + 1 + fib_rek(4 - 2)
7. return 1 + 0 + 1 + fib_rek(2 - 1) + fib_rek(2 - 2)
8. return 1 + 0 + 1 + 1 + fib_rek(2 - 2)
9. return 1 + 0 + 1 + 1 + 0
10. return 1 + 1 + 1
11. return 2 + 1
12. return 3
```

Grananje izraza ovog procesa dobivamo tako da svaki poziv *fib_rek* zamijenimo dvama novim pozivima jer se rezultat funkcije *fib_rek* sastoji od dva rekurzivna poziva (za $n > 1$). Zbog toga je oblik izraza koje generira ovaj proces takav da se oni šire i



Slika 9.2: Rekurzivno stablo izraza *fib_rek(4)*.

Primjer 9.5: Iterativno izračunavanje Fibonaccijevog niza.

```
def fib_iter(n):
    prethodni = 0
    sljedeci = 1
    while n > 0:
        t = prethodni + sljedeci
        prethodni = sljedeci
        sljedeci = t
        n -= 1

    return prethodni

>>> fib_iter(4)
3
```

skupljaju više puta. Takav oblik izraza generiranih nekom rekurzivnom funkcijom zove se *rekurzivno stablo* [40] jer ima oblik stabla. Takvo stablo za izraz *fib_rek(4)* pokazano je na Slici 9.2 na kojoj vidimo da se svaki poziv ove funkcije grana na dvije strane jer u ovoj funkciji imamo dva rekurzivna poziva povezana operatorom "+". Na krajevima ovog stabla promatranog odozgo prema dolje nalaze se brojevi koji zbrojeni daju rezultat ove funkcije.

U rekurzivnom stablu ovog procesa možemo vidjeti da je dio tog stabla za $n = 2$ ponovljen dva puta. Prvi dio bio je formiran pozivom *fib(n - 2)* za $n = 4$, a drugi pozivom *fib(n - 1)* za $n = 3$, što se lako vidi na slici. Iz ovoga možemo zaključiti da rekurzivni procesi mogu biti neučinkoviti u nekim slučajevima jer se tada, kao u ovom slučaju, vrijednost za jedan te isti izraz izračunava više puta.

Iterativna inačica za Fibonaccijev niz, prikazana u Primjeru 9.5, po strukturi je slična funkcije *fakt_iter*. Kao što je bio slučaj kod *fakt_iter*, proces za *fib_iter(4)* izgleda puno jednostavnije od rekurzivnog procesa:

```
1. prethodni = 0, sljedeći = 1 (početne vrijednosti)
2. t = 1, prethodni = 1, sljedeći = 1, n = 4
```

Primjer 9.6: Funkcija *postoji*.

```
def postoji(elem, niz):
    for e in niz:
        if e == elem:
            return True

    return False

>>> postoji('plavo', ['crveno', 'zeleno', 'zuto', 'plavo',
                      'ljubicasto', 'sivo'])
True
```

```
3. t = 2, prethodni = 1, sljedeći = 2, n = 3
4. t = 3, prethodni = 2, sljedeći = 3, n = 2
5. t = 5, prethodni = 3, sljedeći = 5, n = 1
```

Ovdje vidimo kako se varijabla *prethodni* postupno približavala rezultatu, a varijabla *sljedeći* je sadržavala vrijednost koja dolazi nakon one u varijabli *prethodni*. Ovaj proces, kao i onaj za *fakt_iter*, nema nikakvog odgođenog računanja i svaki korak je isključivo određen vrijednostima varijabli *t*, *prethodni*, *sljedeći* i *n*.

Grananje koje je tipično za rekurzivne procese ima jednu, u nekim situacijama negativnu posljedicu, a to je potreba za većom količinom memorije u odnosu na iterativne procese. U primjeru funkcija *fakt_rek* i *fib_rek* vidjeli smo da izraz, to jest broj vrijednosti koje treba pamtiti, sve više raste, pa su i potrebe za memorijom tijekom izvršavanja tih funkcija sve veće. To nije bio slučaj kod njihovih iterativnih inačica jer smo tamo koristili varijable stanja u kojima smo držali sve privremene vrijednosti dok nismo došli do konačnog rezultata. Općenito, kod rekurzivne inačice što je n bio veći trebalo je više memorije, dok je kod iterativne inačice potreba za memorijom bila konstantna, to jest nije ovisila o n. U primjeru Fibonaccijevog niza, funkcija *fib_rek* imala je, osim memorije, jedan dodatni problem, a to je da je neke izraze izračunavala više puta, što bi kod većih vrijednosti za n rezultiralo značajnim rastom vremena potrebnog da pronađe rezultat. Iz ovoga možemo zaključiti da su iterativna rješenja u nekim slučajevima učinkovitija od rekurzivnih, sa stajališta veličine potrebne memorije i brzine izvršavanja. To, međutim, nije uvijek slučaj, odnosno njihova učinkovitost može biti i podjednaka.

Za kraj je ostalo još jedno pitanje vezano za ove dvije vrste procesa: Kada neki proces, odnosno funkciju treba definirati iterativno, a kada rekurzivno? Odgovor najčešće ovisi o tome koja je najprirodnija definicija problema koji rješavamo; međutim, ni ta odluka nije uvijek jednostavna. U našim primjerima s izračunavanjem faktorijela i Fibonaccijevog niza lako smo mogli definirati oba problema na iterativan ili rekurzivan način. U takvim slučajevima možemo uzeti u obzir učinkovitost izvršavanja, tako da bismo se u oba slučaja opredijelili za iterativnu inačicu. S druge strane, kod nekih problema rekurzija je prirodno i prihvatljivo riješenje.

Pretpostavimo da želimo napisati program koji traži neku zadanu vrijednost u nizu,

Primjer 9.7: Dubinsko pretraživanje niza.

```
def jednako(elem_niza, trazeni_elem):
    if isinstance(elem_niza, list):  # je li elem_niza niz?
        return postoji(trazeni_elem, elem_niza)
    else:
        return elem_niza == trazeni_elem

def postoji(elem, niz):
    for e in niz:
        if jednako(e, elem):
            return True

    return False

>>> postoji('plavo',
            ['crveno', 'zeleno', ['zuto', 'plavo'],
                    'ljubicasto', 'sivo'])
True
```

s tim da sam taj niz može imati druge nizove kao elemente [41]. Za početak možemo definirati funkciju (Primjer 9.6) koja jednostavno pretražuje niz za zadanu vrijednost, ne uzimajući u obzir činjenicu da elementi tog niza mogu biti drugi nizovi. Ova funkcija radi ispravno ako se element koji tražimo nalazi u glavnom nizu, to jest ne nalazi se u nekom unutrašnjem nizu. Na primjer, za sljedeći ulazni niz ova funkcija ne pronalazi traženi element:

```
>>> postoji('plavo', ['crveno', 'zeleno',
                ['zuto', 'plavo'], 'ljubicasto', 'sivo'])
False
```

To je zato što `"plavo"` nije isto što i `["plavo"]` - u prvom slučaju imamo tekstualnu vrijednost, a u drugom niz. Budući da su to dvije različite vrste podataka ne možemo ih uspoređivati, to jest ne mogu biti jednaki.

Da bi ova funkcija radila s unutrašnjim nizovima moramo joj omogućiti da pristupi takvom nizu ako dođe do takvog elementa, te da nastavi traženje unutar tog niza, počevši od prvog njegovog elementa. Takav program možemo napisati kako je prikazano u Primjeru 9.7. Dio unutar petlje izvršava se gotovo identično prethodnoj inačici, ali umjesto operatora "==" koristimo funkciju *jednako*. Ova funkcija prvo provjeri je li element koji uspoređujemo sa zadanom vrijednošću u stvari sam niz. Ako nije, onda ga jednostavno usporedimo s tom vrijednošću kao što smo radili u prethodnoj inačici. Međutim, ako je, onda funkcija *jednako* poziva funkciju *postoji* s tim nizom kao ulaznim nizom! Prema tome, pretraživanje unutrašnjih nizova također obavlja funkcija *postoji* čiji rezultat kaže postoji li traženi element u unutrašnjem nizu. Taj rezultat upotrebljava ta ista funkcija dok pretražuje vanjski niz. Ovdje se također radi o rekurziji, samo što funkcija ne poziva samu sebe izravno nego preko neke druge funkcije. U ovom primjeru

Primjer 9.8: Funkcija *postoji* s direktnom rekurzijom.

```
def postoji(elem, niz):
    for e in niz:
        if isinstance(e, list):
            if postoji(elem, e):
                return True
        else:
            if e == elem:
                return True

    return False
```

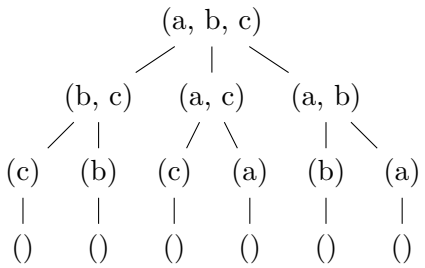
funkcija *postoji* poziva funkciju *jednako* koja poziva funkciju *postoji* pa su stoga te dvije funkcije uzajamno rekurzivne.

Na ovaj način možemo naći bilo koji element bez obzira na to koliko se on duboko nalazi u unutrašnjim nizovima. Na primjer,

```
>>> postoji('zeleno', ['crveno', [['zeleno', 'zuto']]],
            ['plavo'], 'ljubicasto', 'sivo'])
True

>>> postoji('zeleno', ['crveno', [['zeleno']], ['zuto']],
                    ['plavo'], 'ljubicasto', 'sivo'])
True

>>> postoji('bijelo', ['crveno', [['zeleno', 'zuto']]],
                    ['plavo'], 'ljubicasto', 'sivo'])
False
```

Funkciju *postoji* mogli smo napisati i kako je pokazano u Primjeru 9.8. Ova inačica izvršava se na isti način kao i prethodna, ali koristi izravnu rekurziju. Prethodna je inačica eksplicitnije napisana jer su prolaz kroz niz i uspoređivanje (funkcijom *jednako*) odvojeni.

Ako pogledamo neki niz koji sadrži unutarnje nizove, svaki od tih nizova možemo pretraživati na isti način kao i vanjski niz, to jest koristeći istu funkciju. Ulazni niz za tu funkciju upravo je taj unutarnji niz. U ovakvim slučajevima upotreba rekurzije izrazito je korisna tehnika programiranja jer nam omogučava rad s podacima koji su organizirani po nekom određenom načelu u jednu strukturu koja se sastoji od više takvih istih, ali manjih struktura! Na primjer, niz se sastoji od elemenata koji mogu biti drugi nizovi. Za svaki od tih nizova također važi isto pravilio, odnosno i oni se sastoje od elemenata koji mogu biti drugi nizovi, i tako dalje. Ovdje treba uočiti to da smo za definiciju niza koristili pojam niza - jedan niz može sadržavati drugi niz. Ova sama definicija je, prema tome, rekurzivna, pa smo i problem pretraživanja niza riješili koristeći rekurziju. Kao što je već rečeno, međutim, u nekim situacijama rekurzija nije najučinkovitije rješenje. Čak i ako je problem definiran rekurzivno potrebno je imati uvid u svojstva takvih procesa



Slika 9.3: Stablo generiranja kombinacija. Svaki čvor sadrži jednu kombinaciju (iako se neke ponavljaju).

(kao što smo napravili kod funkcije *fib_rek*) prije nego što se odlučimo na iteraciju ili rekurziju, a isto tako može postojati i više rekurzivnih rješenja od kojih neka mogu biti učinkovitija od drugih. Rekurzija općenito nije učinkovitija od iteracije (iako može biti podjednako efikasna), ali funkciju može učiniti jasnijom, pogotovo ako izravno odgovara definiciji problema, kao što je bio slučaj s pretraživanjem nizova. Nadalje, svaki se rekurzivan algoritam može napisati i bez rekurzije, tj. iterativno, što znači da svaku rekurzivnu funkciju možemo konvertirati u iterativan oblik.

Primjer: kombinacije i permutacije

Kako možemo generirati skup svih kombinacija elemenata nekog niza? Primjerice, za n-torku `('a', 'b', 'c')` želimo dobiti `{('a', 'c'), ('c',), ('a', 'b', 'c'), ('b', 'c'), ('a', 'b'), ('b',), (), ('a',)}` (u bilo kojem poretku) [42]. Broj kombinacija za n elemenata je 2^n. Ova nam činjenica može poslužiti za provjeru ispravnosti rezultata, ali ne i za njihovo generiranje. Kod ovakvih problema često nam može pomoći skica stabla po kojoj kasnije možemo definirati algoritam te njegovu implementaciju. Za generiranje kombinacije ćemo, dakle, krenuti od stabla prikazanog na Slici 9.3. U korijenu se nalazi početni niz. Na razini ispod taj se niz grana u podnizove bez prvog, drugog i trećeg elementa. Svaki se od tih nizova opet grana na isti način kao i korijen. Očito je da bi za više elemenata stablo bilo dublje. Vidimo da ovo stablo sadrži sve kombinacije, samo nam treba sustavan način generiranja. Načelo je, dakle, da za svaki niz izdvojimo podnizove bez elementa na položaju 1, 2, ..., n, a isto učinimo i za svaki od tih podnizova. Ako sada prođemo ovo stablo u dubinu dobili bismo kombinacije `(a, b, c), (b, c), (c), (), (b), (), (a, c), (c), ...`.

U implementaciji ovog postupka duplikate možemo lako preskočiti jednostavnom provjerom gdje bi nam na kraju ostale samo jedinstvene kombinacije. Cijeli postupak implementiran je funkcijom *gen_kombinacije* prikazanom u Primjeru 9.9.

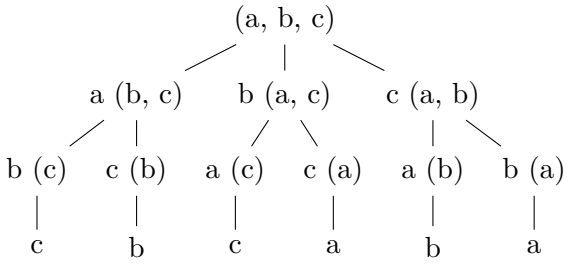
Na sličan način možemo generirati i permutacije. Za niz od n elemenata broj permutacija je $n!$. Za razliku od kombinacija, svaka permutacija sadrži isti broj elemenata kao u početnom nizu. I za ovaj se problem možemo poslužiti stablom na Slici 9.3. Ovdje ćemo, međutim, to stablo drugačije tumačiti. Za razliku od kombinacija koje se nalaze u čvorovima stabla ovdje ćemo uzimati u obzir put od korijena do posljednjeg čvora (koji

Primjer 9.9: Funkcija *gen_kombinacije*.

```
def gen_kombinacije(niz):
    def kombinacije(elem, rezultat):
        for i in range(len(elem)):
            p = elem[:i] + elem[i + 1:]  # bez i-tog elementa
            rezultat.add(p)
            if len(p) > 0:
                kombinacije(p, rezultat)  # iduca razina rek. stabla

    k = {niz}
    kombinacije(niz, k)
    return k

>>> gen_kombinacije(('a', 'b', 'c'))
{('b',), ('a', 'b', 'c'), ('a', 'b'), ('b', 'c'), ('a',), ('c',), (),
 ('a', 'c')}
```

Primjer 9.10: Funkcija *permutacije*.

```
def permutacije(niz, permutacija=(), rezultat=set()):
    if len(niz) == 0:  # jesmo li dosli do kraja stabla?
        rezultat.add(permutacija)
    else:
        for i in range(len(niz)):
            prvi = niz[i]
            ostatak = niz[:i] + niz[i + 1:]
            permutacije(ostatak,
                        permutacija + (prvi,),
                        rezultat)

        return rezultat

>>> permutacije(('a', 'b', 'c'))
{('b', 'c', 'a'), ('c', 'a', 'b'), ('a', 'c', 'b'), ('c', 'b', 'a'),
 ('a', 'b', 'c'), ('b', 'a', 'c')}
```



Slika 9.4: Stablo generiranja permutacija. U zagradi se nalazi ostatak elemenata iz prethodnika (roditelja) nakon što je izdvojen element koji se nalazi izvan zagrada. Izdvojeni elementi ulaze u permutaciju koja se dobiva od izdvojenih elemenata na putu od korijena do lista stabla.

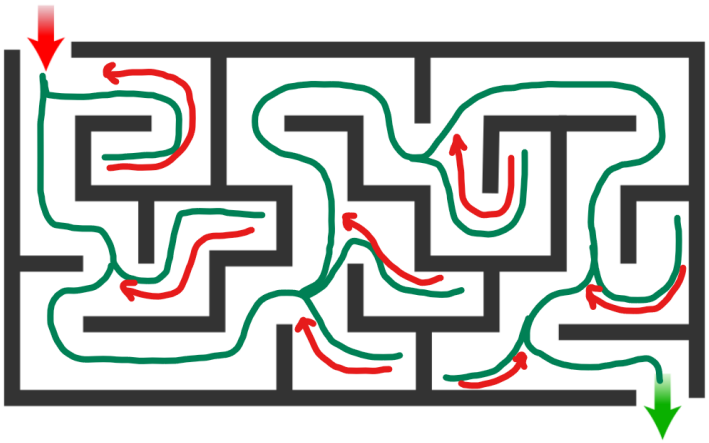
nije list stabla). Primjerice, jedan takav put lijeva je strana stabla s čvorovima `(a, b, c)`, `(b, c)`, `(c)`. Permutacije možemo dobiti tako da za svaki čvor prođemo sve elemente po redu prolazeći stablo u dubinu. Prvi takav prolaz dao bi `(a, b, c)` gdje smo uzimali prvi element svakog čvora. Po povratku iz čvora *c* uzimamo sljedeći element čvora `(b, c)`, to jest *c* i idemo na sljedeću granu `(b)` čime bismo generirali permutaciju `(a, c, b)`. Povratkom u korijen nastavljamo s njegovim drugim elementom, *b*, kao polaznim, čime bismo dobili permutacije `(b, a, c)` i `(b, c, a)`. Ponavljanjem ovog postupka dobili bismo sve permutacije. Ovo je implementirano funkcijom *permutacije* prikazanom u Primjeru 9.10. Funkcija *permutacije* na svakoj razini stabla uzima različiti element kao polazni (iz niza elemenata u kojem nema elementa koji je uzet na prethodnoj višoj razini) kao što je prikazano na Slici 9.4. Ta je slika ista kao i Slika 9.3 samo što su elementi koji ulaze u permutaciju izdvojeni tako da se za svaki put u stablu jasno vidi permutacija koju on generira.

9.2 Rekurzivno traženje i vraćanje istim putem

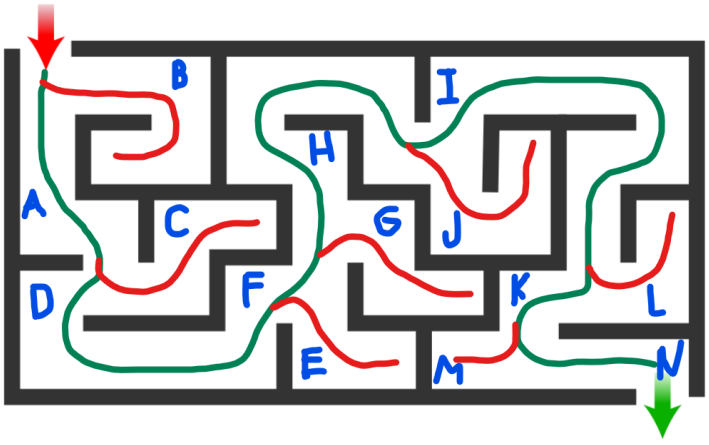
Na Slici 9.5 prikazan je jednostavan labirint. Ako pratimo zelenu crtu doći ćemo do mjesta u labirintu od kojeg ne možemo dalje. Crvena crta označava da se moramo vratiti na mjesto posljednjeg grananja i probati neku drugu alternativu. To se naziva *vraćanjem istim putem* (engl. *backtracking*). Na Slici 9.6 putevi u labirintu označeni su slovima. Svaki dio puta od jednog grananja do idućeg označen je svojim slovom. Na ovoj se slici može uočiti stablo kod kojeg je svaka grana označena jednim slovom (Slika 9.7).

Vidimo da je ovo stablo binarno jer ovaj labirint ima samo dvije alternative za svako grananje. Kod složenijih labirinata i njihovo bi stablo bilo složenije. Ovo smo stablo mogli nacrtati i tako da nije uvijek lijeva grana ta koja vodi k izlazu. Primjer takvog stabla prikazan je na Slici 9.8. Ovo stablo identično je prethodnom jer su čvorovi identični kao i sljedbenici svakog čvora. Isto tako, vidimo da je put do izlaza isti kod oba stabla: *ulaz, A, D, F, H, I, K, N, izlaz.*

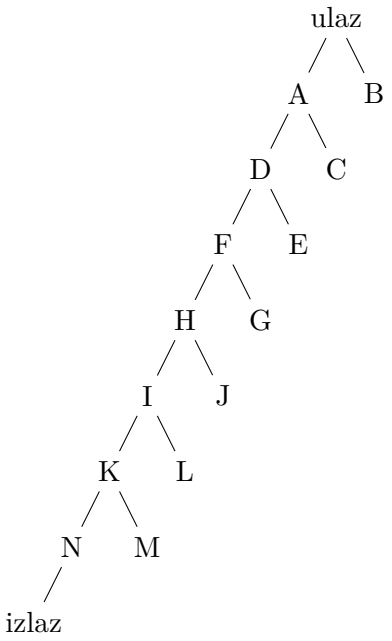
Labirint, dakle, možemo prikazati kao stablo. Labirint na Slici 9.6 možemo u Pythonu prikazati kao listu na Slici 9.9. Na toj slici vidimo da se ta lista sastoji od



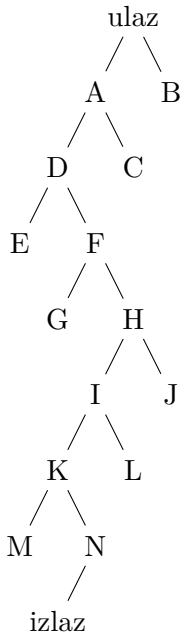
Slika 9.5: Labirint s označenim uzmakom.



Slika 9.6: Labirint s označenim izborima puteva.



Slika 9.7: Labirint kao stablo.



Slika 9.8: Labirint kao stablo.

```
stablo = ['ulaz',
          ['A',
           ['D',
            'E',
            ['F',
             'G',
             ['H',
              ['I',
               ['K',
                'M',
                ['N', 'izlaz']],
               'L'],
              'J']]],
           'C'],
          'B']
```

Slika 9.9: Stablo sa Slike 9.8 prikazano kao Pythonova lista.

podlista od kojih svaka ima dva člana koji predstavljaju lijevu (prvi član) i desnu (drugi član) granu.

Na Slici 9.10 prikazan je još jedan labirint gdje su slovima označena grananja. Tom labirintu odgovara stablo na Slici 9.11. Traženje izlaza u takvom labirintu se, prema tome, svodi na pretraživanje stabla. Funkcija *nadji_izlaz* u Primjeru 9.11 implementira pretraživanje u dubinu vraćanjem istim putem i ispisuje prođene puteve (Primjer 9.12).

Vidimo da smo se nakon čvora L morali vratiti nazad na razinu čvorova A, B i C gdje je pronađen izlaz. To je vraćanje istim putem: kada smo došli do čvora L vidjeli smo da ne možemo dalje pa smo se vraćali unazad sve dok nismo došli do čvora čije sljedbenike još nismo ispitali. Ako govorimo o labirintu, vratili smo se unazad do onog mjesta kod

Primjer 9.11: Funkcija *nadji_izlaz*.

```
1  def nadji_izlaz(stablo, ident):
2      for tocka in stablo:
3          if isinstance(tocka, list):
4              print(ident, tocka[0])  # ispisi korijen
5              if nadji_izlaz(tocka[1:], ident + ' ' * 2):
6                  return True
7          else:
8              print(ident, tocka)  # ispisi list stabla
9              if tocka == 'izlaz':
10                 return True
11
12     return False  # vracanje unatrag (ne moze dalje)
```

Primjer 9.12: Ispis puteva kroz labirint na Slici 9.10.

```
>>> stablo = ['ulaz',
                  'A',
                  ['B',
                      ['F', 'G',
                          ['H', 'I',
                              ['M', 'J', 'K', 'L']]],
                      'E', 'D'],
                  ['C', 'izlaz']]

>>> nadji_izlaz([stablo], ' ')
  ulaz
    A
    B
      F
        G
        H
          I
          M
            J
            K
            L
      E
      D
    C
      izlaz
True
```

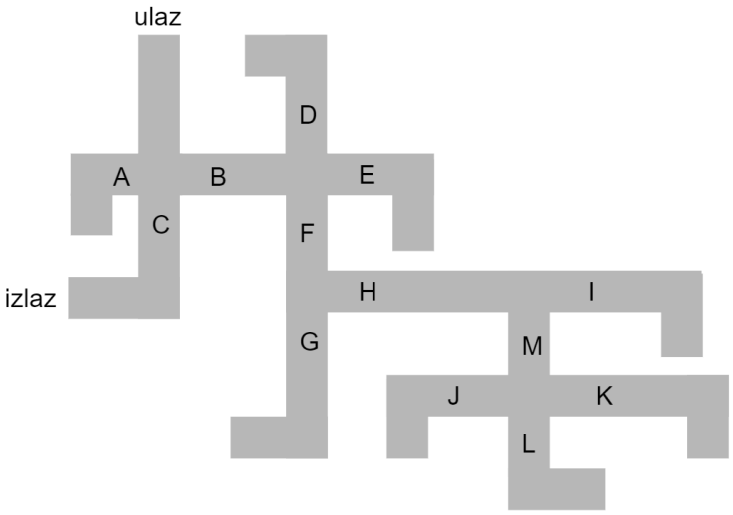
kojeg je postojao put koji još nismo istražili, što je u ovom primjeru C. Prema tome, ako slijedimo čvorove počevši od čvora *ulaz* doći ćemo do izlaza putem $ulaz \rightarrow C \rightarrow izlaz$.

> Vraćanje istim putem korisna je tehnika programiranja za rekurzivno traženje rješenja.

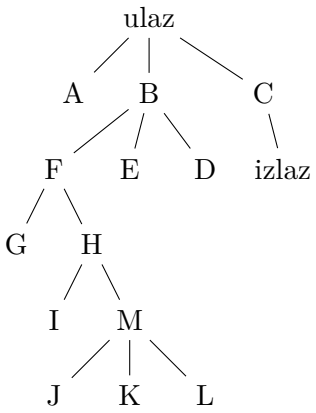
Općenito, vraćanje istim putem karakteristično je za neke rekurzivne postupke kao što je pretraživanje stabla, ali i mnoge druge. Kada se kaže da se u nekom rekurzivnom postupku upotrebljava vraćanje istim putem, znamo da se radi o nečemu što uključuje alternativne pokušaje grananja (govoreći o rekurzivnom stablu) da bi se došlo do rješenja.

9.2.1 Primjer: Elementi u poretku s ograničenjima

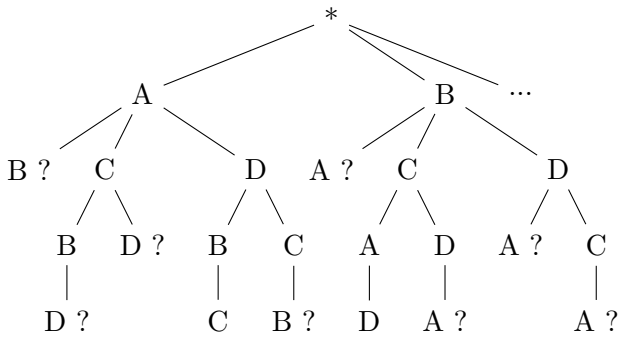
U ovom dijelu proučit ćemo još jedan primjer upotrebe rekurzije vraćanjem istim putem. Neka je zadan ovaj problem: Četiri osobe, A, B, C i D treba rasporediti za okruglim stolom po sljedećim pravilima:



Slika 9.10: Labirint.



Slika 9.11: Stablo labirinta na Slici 9.10.



Slika 9.12: Dio stabla traženja odgovarajućeg poretka sjedenja.

- B ne smije sjediti pored A
- D mora sjediti odmah lijevo od A.

Na koji način možemo dobiti sve kombinacije sjedenja koje zadovoljavaju gornja pravila? Jedan način bio bi generirati sve permutacije s ova četiri elementa, ali za veći broj elemenata to bi bilo nepraktično. Ovdje ćemo primjeniti tehniku programiranja s rekurzijom i vraćanjem istim putem. Prvo od čega možemo krenuti je stablo traženja primjenom sljedeće strategije: krenemo od prvog elementa, recimo A, i onda uzmemo iduće element, recimo B, i ako kombinacija tih dvaju elemenata ne narušava bilo koje pravilo uzmemo treći element, i tako dalje. Ako je barem jedno pravilo narušeno ne uzimamo sljedeći element nego zamijenimo zadnji koji smo uzeli nekim drugim i opet ponavljamo isti postupak. Postupak ponavljamo dok ne nađemo sve kombinacije sjedenja koje zadovoljavaju zadana pravila.

Početak takvog stabla traženja prikazan je na Slici 9.12. Ako počnemo s elementom A i nakon njega odaberemo B već imamo poredak koji narušava prvo pravilo. Zbog toga nema potrebe dalje tražiti i na tom mjestu prekidamo traženje i pokušavamo novu kombinaciju koja počinje s A, onu koja nakon A odabire C. Djelomična kombinacija AC ne krši niti jedno pravilo pa nakon C uzimamo B. ACB i dalje ne krši niti jedno pravilo pa nakon B uzimamo posljednji element, D. Međutim, kombinacija ACBD krši drugo pravilo, a to je da D mora sjediti odmah lijevo od A. U ovom slučaju D se nalazi desno od A. Sada se opet moramo vratiti unazad do čvora gdje možemo odabrati neki drugi element, a to je C. Nakon C možemo umjesto B odabrati samo D pa imamo parcijalnu kombinaciju ACD koja ne zadovoljava drugo pravilo. Ponovo se vraćamo unazad sve do A gdje za idući element probamo jedinu preostalu alternativu, D. Vidimo da polazeći od ovog elementa možemo dobiti jednu ispravnu kombinaciju, ADBC. Na isti način, polazeći od prvog elementa B možemo dobiti ispravnu kombinaciju BCAD. Na isti način možemo dobiti i preostale dvije kombinacije.

Ovaj postupak primijenjen je u Primjeru 9.14 funkcijom *poredak*. Funkcija *uvjeti_zadovoljeni* prikazana je u Primjeru 9.13. Ovaj bismo postupak mogli proširiti i na više elemenata i/ili pravila jer funkcija *poredak* ne pretpostavlja ništa o broju elemenata i

Primjer 9.13: Funkcija *uvjeti_zadovoljeni*.

```
def uvjeti_zadovoljeni(p):
    def b_pored_a():
        zadovoljen = False
        if len(p) < 4:
            # Mozemo odmah provjeriti nalazi li se B pored
            # A, ne moramo imati potpunu permutaciju. Na
            # primjer, nakon DBA mozemo odmah prekinuti
            # pretrazivanje jer B se ne smije nalaziti
            # pored A.
            zadovoljen = 'AB' not in p and 'BA' not in p
        else:
            prvi_zadnji = p[0] + p[-1]
            zadovoljen = ('AB' not in p
                          and 'BA' not in p
                          and prvi_zadnji not in {'AB', 'BA'})

        return zadovoljen

    def d_lijevo_od_a():
        zadovoljen = False
        if len(p) == 4:
            zadovoljen = ('AD' in p
                          or (p[0] == 'D'
                              and p[-1] == 'A'))
        else:
            # Mozemo odmah provjeriti nalazi li se D lijevo
            # od A, ne moramo imati potpunu permutaciju.
            # Ako A ili D nije u nizu onda je ovaj uvjet
            # zadovoljen.
            zadovoljen = ('AD' in p
                          or 'A' not in p
                          or 'D' not in p)

        return zadovoljen

    return b_pored_a() and d_lijevo_od_a()
```

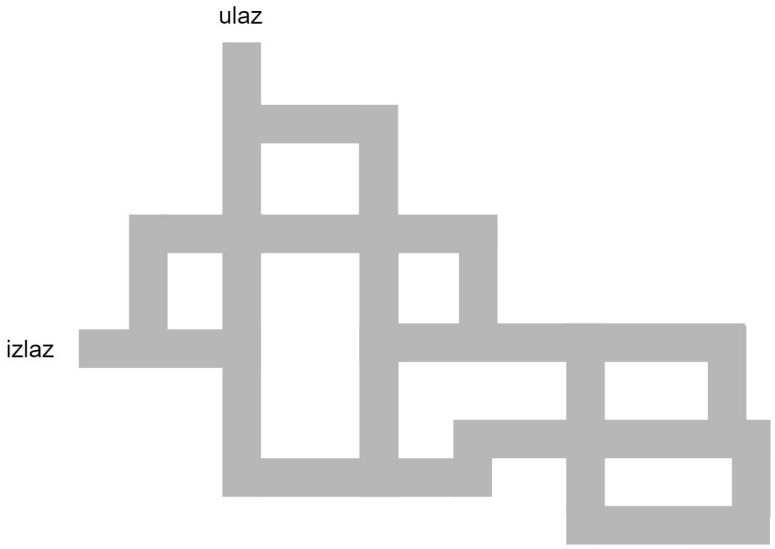
Primjer 9.14: Funkcija *poredak*.

```
def poredak(niz):
    def _fn(elementi, trenutni_poredak, rezultat, razmaci):
        if not uvjeti_zadovoljeni(trenutni_poredak):
            print(' ?', end='')
            return  # backtrack

        if len(elementi) == 0:  # jesmo li dosli do kraja stabla?
            print(' \u2713', end='')
            rezultat.add(trenutni_poredak)
        else:
            for i in range(len(elementi)):
                prvi = elementi[i]
                ostatak = elementi[:i] + elementi[i + 1:]
                print('\n', razmaci, prvi, end='')
                _fn(ostatak,
                    trenutni_poredak + prvi,
                    rezultat,
                    razmaci + '   ')

            return rezultat

    return _fn(niz, '', set(), '')
```

pravila - ona samo treba podatak o tome zadovoljava li neki poredak (djelomičan ili potpuni) uvjete.

Problemi ovog tipa, kod kojih treba naći rješenje koje zadovoljava zadane uvjete nazivaju se *problemi zadovoljavanja ograničenja* (engl. *constraint satisfaction problems*) [39].

Iz ovog primjera vidimo da je vraćanje istim putem zapravo prekidanje traženja dublje u stablo: ako vidimo da iz bilo kojeg razloga nema potrebe ići dalje (dublje) prekidamo postupak, odnosno vraćamo se iz rekurzivnog poziva i probamo neku drugu alternativu. Nadalje, ovdje smo uštedjeli i na vremenu jer nismo analizirali sve permutacije zadanih elemenata nego smo kod mnogih prekinuli traženje dok još nisu bile potpune. Možemo primjetiti da je funkcija *poredak* jako slična funkciji *permutacije*. Razlog je što funkcija *poredak* također generira permutacije, ali je razlika u tome što ona traži samo određene permutacije, to jest one koje zadovoljavaju zadane uvjete. Dakle, funkcija *poredak* traži, a funkcija *permutacije* samo generira. Rekurzija vraćanjem istim putem korisna je upravo za ovakve probleme gdje tražimo nešto što zadovoljava zadana ograničenja.



Slika 9.13: Labirint u kojem se može ići u krug.

10 Zadaci

10.1 Općeniti zadaci

1. Napišite program koji radi prema sljedećem algoritmu za pronalaženje korijena broja:

   (1) Odredi neku početnu vrijednost a kao što je 1.

   (2) Neka je x vrijednost za koju izračunavamo korijen. Ako su a^2 i x dovoljno blizu (vidi ispod), rezultat je a; inače idi na sljedeći korak.

   (3) Postavi a na poboljšani rezultat u odnosu na prethodni, tj. na prosjek od a i x/a, što je *(a + x / a) / 2*.

   (4) Idi na korak 2.

   Neka je aproksimacija rezultata dovoljno blizu stvarnom rezultatu ako je razlika između a^2 i x manja od 0.001. U Tablici 10.1 prikazan je primjer ovog postupka na izračunavanju korijena broja 4.

2. Napišite funkciju *korijen* za program iz zadatka 1.

3. Napišite funkciju *djeljivi* koja će za zadani interval cijelih pozitivnih brojeva ispisati samo one koji su djeljivi zadanim brojem. Ako zadani interval ne sadrži samo cijele brojeve funkcija treba ispisati odgovarajuću poruku. Operator za ostatak dijeljenja je `%`.

Tablica 10.1: Računanje korijena.

a	**Prosjek**
1	(1 + 4 / 1) / 2 = 2.5
2.5	(2.5 + 4 / 2.5) / 2 = 2.05
2.05	(2.05 + 4 / 2.05) / 2 = 2.0006
2.0006	(2.0006 + 4 / 2.0006) / 2 = 2.0000
2.0000 ←rezultat	-

```
# ispisati sve brojeve u intervalu (a, b) djeljive s n
def djeljivi(a, b, n):
    ...

>>> djeljivi(1, 10, 3)  # primjer upotrebe
3
6
9
```

4. Napišite funkciju *eksp* koja za dva parametra, a i b, koji su pozitivni cijeli brojevi vraća vrijednost a^b:

```
>>> eksp(2, 4)  # primjer upotrebe
16
```

Napomena: Za ovu funkciju ne smijete koristiti gotova rješenja kao što je operator ** ili funkcija *pow*.

5. Napišite funkciju *faktorijel* koja za zadani pozitivni cijeli broj vraća faktorijel tog broja:

```
>>> faktorijel(4)  # primjer upotrebe
24
```

Faktorijel broja n označava se sa "n!" i jednak je $1 \cdot 2 \cdot 3 \cdot ... \cdot n$. Na primjer, $4! = 1 \cdot 2 \cdot 3 \cdot 4 = 24$.

6. Napišite funkciju *nzd* koja vraća najveći zajednički djelitelj dvaju brojeva zadanih kao parametre te funkcije:

```
>>> nzd(8, 12)  # primjer upotrebe
4
```

7. Napišite funkciju *prost_broj* koja vraća *True* ako je zadani broj prost ili *False* ako nije.

```
>>> prost_broj(11)  # primjer upotrebe
True

>>> prost_broj(15)  # primjer upotrebe
False
```

Prosti brojevi su oni koji su djeljivi samo brojem 1 i samim sobom.

8. Napišite funkciju *zbroj_razlomaka* koja vraća zbroj dvaju razlomaka. Razlomci trebaju biti zadani u obliku liste. Na primjer, razlomak $\frac{2}{7}$ bio bi zadan kako lista `[2, 7]`. Isto tako, funkcija treba vratiti listu koja predstavlja razlomak kao rezultat.

```
>>> zbroj_razlomaka([1, 3], [2, 5])  # primjer upotrebe
[11, 15]
```

9. Napišite funkciju *prosjek* koja za zadanu listu brojeva vraća njihov prosjek.

```
>>> prosjek([4, 11, 7, 3, 10])  # primjer upotrebe
7
```

Za sljedeća tri zadatka ispod matrica kao što je $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$ **treba u kodu biti prikazana kao lista [[1, 2, 3], [4, 5, 6]].**

10. Napišite funkciju *zbroj_matrica(x, y)* koja vraća zbroj zadanih matrica.

```
# primjer upotrebe
>>> zbroj_matrica([[12, 7, 3], [4, 5, 6], [7, 8, 9]],
                  [[5, 8, 1], [6, 7, 3], [4, 5, 9]])
[[17, 15, 4], [10, 12, 9], [11, 13, 18]]
```

11. Napišite funkciju *trans(x)* koja vraća transponiranu matricu zadane matrice *x*.

```
# primjer upotrebe
>>> trans([[12, 7], [4, 5], [3, 8]])
[[12, 4, 3], [7, 5, 8]]
```

12. Napišite funkciju *umnozak_matrica(x, y)* koja vraća umnožak zadanih matrica.

```
# primjer upotrebe
>>> umnozak_matrica([[12, 7, 3], [4, 5, 6], [7, 8, 9]],
                    [[5, 8, 1, 2], [6, 7, 3, 0], [4, 5, 9, 1]])
[[114, 160, 60, 27], [74, 97, 73, 14], [119, 157, 112, 23]]
```

13. Napišite funkciju *n_najvecih(p, n)* koja za niz *p* vraća (u istom redoslijedu kao i u *p*) samo one elemente koji su jednaki ili veći od *n* najvećih vrijednosti u nizu. Primjer:

```
>>> n_najvecih([8, 1, -6, 4, -14, 14, 9, 2, 14, 8], 3)
[8, 14, 9, 14, 8]
```

U ovom primjeru tri najveće vrijednosti niza su 14, 9 i 8. Ovdje 3 ne znači da trebamo dobiti tri vrijednosti u rezultatu, nego sve vrijednosti niza koje su jednake ili veće od svake od te tri vrijednosti. Isto tako, poredak vrijednosti u rezultatu mora biti sačuvan. U gornjem primjeru, zadnje dvije vrijednosti 14 i 8 moraju se nalaziti iza vrijednosti 9, kao što se nalaze i u zadanom nizu.

14. Napišite funkciju *komp* koja komprimira tekstualne podatke tako da nađe znak *Z* koji se ponavlja više od dva puta zaredom i njega smjesti kao par *(n, Z)*, gdje je *n* broj ponavljanja znaka *Z* za određeni segment teksta. Ova funkcija može vratiti listu ili n-torku kao rezultat. Druga funkcija, *dekomp*, treba iz komprimiranog zapisa vratiti originalni tekst.

```
>>> komp("abbaaaccccaade")
('a', 'bb', (3, 'a'), (5, 'c'), 'aa', 'd', 'e')

>>> dekomp(('a', 'bb', (3, 'a'), (5, 'c'), 'aa', 'd', 'e'))
'abbaaaccccaade'
```

15. Napišite funkciju *isti* koja za zadani znakovni niz vraća listu koja sadrži sve njegove podnizove koji se čitaju isto slijeva nadesno i sdesna nalijevo. Na primjer, u tekstu *istitisak* postoje dva takva podniza: *tit* i *iti*.

```
>>> isti("istitisak")
['tit', 'iti']  # redoslijed može biti drugačiji
```

16. Dva znakovna niza *anagrami* su ako se od jednog može dobiti drugi premještanjem znakova. Na primjer, "abcb" i "bcab" su anagrami jer sadrže ista slova, samo u drugačijem redoslijedu, ali "abcc" i "abbc" nisu. Napišite funkciju *anagram(s1, s2)* koja vraća *True* ako su znakovni nizovi *s1* i *s2* anagrami.

17. Upotrebom rječnika napišite funkciju *zajednicki* koja vraća zajedničke elemente dviju Pythonovih lista. Na primjer,

```
>>> zajednicki([5, 2, 7, 2, 8], [1, 2, 3, 4, 5])
[2, 5]  # redoslijed može biti drugačiji
```

18. Napišite funkciju koja sortira vrijednosti tako da svaki put nađe najmanju vrijednost u nizu. Ovdje možete upotrijebiti operaciju *del* za brisanje elemenata iz liste.

19. Upotrebom rječnika napišite funkciju *frekv* koja vraća broj pojavljivanja svakog elementa liste:

```
>>> frekv([5, 2, 4, 4, 3, 1, 3, 8])
{5: 1, 2: 1, 4: 2, 3: 2, 8: 1, 1: 1}
```

20. Napišite funkciju *samoglasnici*(s) koja vraća rječnik koji sadrži sve samoglasnike (a, e, i, o, u) u zadanom znakovnom nizu i broj njihovog pojavljivanja.

```
>>> samoglasnici("poplava")  # primjer upotrebe
{'o': 1, 'a': 2}  # poredak može biti drugačiji
```

21. Napišite funkciju *duplvr* koja će iz rječnika (*dictionary*) ukloniti sve ključeve čija je pridružena vrijednost ista kao i vrijednost nekog drugog ključa. Tada u rječniku trebaju ostati samo ključevi čija je pridružena vrijednost jedinstvena.

```
>>> duplvr({'a': 84, 'b': 11, 'c': 84})
{'a': 84, 'b': 11}  # redoslijed može biti drugačiji

# ili

>>> duplvr({'a': 84, 'b': 11, 'c': 84})
{'c': 84, 'b': 11}  # redoslijed može biti drugačiji
```

22. Za zadani niz brojeva A napišite funkciju *zbroj(n)* koja vraća ukupan broj podnizova niza A kod kojih je zbroj vrijednosti jednak *n*. Neka ta funkcija vrati i same podnizove.

23. Upotrebom skupova napišite funkciju koja vraća broj duplikata u listi:

```
>>> duplikati([5, 2, 5, 1, 1, 1, 2, 3])
4
```

24. Napišite funkciju *trazi* koja implementira binarno pretraživanje liste i vraća True ako traženi element postoji ili False ako ne postoji. Za sortiranje liste možete upotrijebiti Pythonove funkcije *sort* ili *sorted*.

```
>>> trazi([18, 1, 40, 25, 27, 18, 3], 3)
True
```

25. Napišite funkciju *preuredi* koja preraspoređuje elemente liste na slučajan način (bez upotrebe neke takve postojeće funkcije). Na primjer,

```
>>> preuredi([1, 2, 3, 4])
[3, 1, 4, 2]  # ili neki drugi raspored
```

    Za generiranje slučajnih brojeva koristite funkciju *randrange* modula *random*.

26. Neka je zadan niz A koji se sastoji samo od cijelih brojeva. Upotrebom funkcije *combinations* modula *itertools* napišite program koji će rezultirati listom koja se sastoji od lista brojeva iz niza A čiji je zbroj neka zadana vrijednost K. Primjerice, ako je A = [3, 8, 1, 7, 8, 2, 4, 5] i K = 5 onda je rješenje [[3, 2], [1, 4], [5]].

27. Napišite funkciju *sastavljena(a, b)* koja vraća *True* ako se lista *a* može sastaviti od skupine lista u *b*. U prvom primjeru ispod, lista [5, 1, 7, 4, 4, 9, 2, 1] može se sastaviti od lista [5] + [1] + [7, 4, 4] + [9, 2, 1], tako da funkcija vraća *True*. U drugom primjeru ne možemo dobiti listu *a* jer poredak elemenata u listi [9, 1, 2] ne odgovara onome u listi *a*. U trećem primjeru također ne možemo sastaviti zadanu listu.

```
>>> sastavljena([5, 1, 7, 4, 4, 9, 2, 1], [[9, 2, 1],
                 [1], [7, 4, 4], [5]])
True

>>> sastavljena([5, 1, 7, 4, 4, 9, 2, 1], [[9, 1, 2],
                [1], [7, 4, 4], [5]])
False

>>> sastavljena([5, 9, 2, 1, 1, 7, 4, 4],
                [[9, 2, 1], [1, 1, 7], [4, 4], [5]])
False
```

28. Za dvodimenzionalni niz kao što je onaj ispod napišite program koji će dati par točaka koje su na najmanjoj udaljenosti jedna od druge. U primjeru ispod to su dvije točke na koordinatama (6, 4) i (7, 6), gdje je koordinata par *(<redak>, <stupac>)*.

```
tocke = [
    '..................',
    '..................',
    '.....*............',
    '..................',
    '............*.....',
    '..................',
```

```
    '....*.......*.......',
    '......*.............',
    '....................',
    '..........*.......*.',
]
```

29. Napišite program koji će na osnovu matrice kao što je ona ispod utvrditi koliko je prostora omeđenih (to jest, zatvorenih) znakom "#" te koliko je točaka u svakom od njih. U primjeru ispod tri su takva prostora gdje se u jednom nalazi sedam točaka, u drugom pet, a u trećem dvije. Prostor je zatvoren ako se iz njega ne može doći do barem jedne rubne točke po horizontali ili vertikali (ne uzimaju se u obzir dijagonale).

```
prostor = [
    '.............#......',
    '.............#####..',
    '...#####.....#......',
    '...#...#.....#######',
    '...#....#....#.....#',
    '...########..#######',
    '...#................',
    '...###...######.....',
    '.........###..####..',
    '...........###......',
]
```

30. Napišite operaciju sortiranja tako da se redoslijed elemenata u rezultirajućem nizu može zadati kao parametar:

```
>>> sortiraj([1, 5, 2], lambda x, y: x > y)
[5, 2, 1]

>>> sortiraj([1, 5, 2], lambda x, y: x < y)
[1, 2, 5]

>>> sortiraj([4, "2", 8, "5"], lambda x, y: int(x) < int(y))
['2', 4, '5', 8]
```

31. Koristeći samo funkciju *reduce* (iz modula *functools*) i bez upotrebe petlje ili rekurzije, napišite funkciju *ukupno* koja će za zadani niz rječnika s parovima cijena/vrijednost dati zbroj svih vrijednosti, kako je ispod pokazano (funkcija *reduce* radi isto kao funkcija *redukcija* iz jednog od prethodnih zadataka).

```
>>> podaci2 = [{'cijena': 25}, {'cijena': 10},
               {'cijena': 10}, {'cijena': 15},
               {'cijena': 30}]

>>> ukupno(podaci2)
90
```

32. Upotrebom funkcije *reduce* napišite izraz kojim se binarni broj prevodi u decimalni koristeći funkciju $f(a, b) = 2a + b$, gdje su a i b dvije susjedne binarne znamenke.

33. Napišite program za konverziju broja iz jednog brojevnog sustava u drugi. Program treba imati funkciju *konverzija(v, b_1, b_2)* gdje je v znakovni niz koji sadrži broj koji konvertiramo, b_1 je baza iz koje konvertiramo, a b_2 baza u koju konvertiramo. Ispod su pokazani primjeri s bazama 2, 10 i 16 (vaš program treba raditi s bilo kojom bazom):

```
>>> konverzija('2622', 10, 16)
'a3e'

>>> konverzija('a3e', 16, 10)
'2622'

>>> konverzija('2622', 10, 2)
'101000111110'

>>> konverzija('101000111110', 2, 10)
'2622'

>>> konverzija('101000111110', 2, 16)
'a3e'
```

34. Napišite funkciju *spoji_rječnike(a, b)* koja će spojiti dva rječnika, a i b, na sljedeći način:

    (a) Ako za isti ključ vrijednost niti u jednom od rječnika nije tipa *list*, onda za vrijednost x iz rječnika a i vrijednost y iz rječnika b u novom rječniku vrijednost treba biti lista `[x, y]`.

    (b) Ako za isti ključ jedan rječnik sadrži listu $[e_1, e_2, \ldots, e_n]$, a drugi vrijednost x nekog drugog tipa, onda novi rječnik za taj ključ treba sadržavati $[e_1, e_2, \ldots, e_n, x]$.

    (c) Ako oba rječnika za dotični ključ sadrže liste, onda novi rječnik za taj ključ treba sadržavati listu s vrijednostima iz obje liste.

35. Napišite svoju implementaciju Pythonove funkcije *groupby*.

10.2 Klase

1. Napišite klasu koja predstavlja kompleksan broj i definirajte metodu za računanje modula takvog broja.

2. Napišite jednostavnu biblioteku kao skup klasa za rad s geometrijskim likovima. Neka objekti tih klasa imaju metode za računanje površine tog geometrijskog lika. Hoćete li to riješiti nasljeđivanjem ili na neki drugi način? Komentirajte svoj izbor.

3. Napišite jednostavan program za knjižnicu. Odredite koje vam klase trebaju i kako ćete ih objediniti u jedan program. Program treba omogućavati pretraživanje knjiga po nazivu, autoru, datumu izdanja ili ISBN-u, dodavanje i brisanje knjiga te prikaz članova knjižnice i knjiga koje su posudili, a još nisu vratili.

4. Implementirajte strukturu podataka sličnu skupu, ali koja dozvoljava duplikate i koja će se zvati *Multiset*. Ta struktura podataka treba podržavati sljedeće operacije za skupove: provjera pripadnosti elementa, unija, presjek, razlika i jednakost. Dva multiseta su jednaka ako imaju isti broj svakog elementa. Ovaj zadatak riješite na dva načina:

   (a) kao samostalnu klasu

   (b) kao klasu koja je izvedena (nasljeđuje) od ugrađenog tipa *set*.

10.3 Sistemski alati

1. Neka postoji tekstualna datoteka koja sadrži niz cijelih brojeva odvojenih zarezom kao `17, 199, 57, 30, 26, ...`. Napište program koji će se nalaziti u datoteci `prosjek.py` koji će učitati ove vrijednosti iz datoteke i kao rezultat ispisati prosječnu vrijednost. Nadalje, ovaj program treba pokretati tako da ga se može zadati u komandnoj liniji zajedno s nazivom ulazne datoteke. Na primjer,

```
> py prosjek.py podaci.txt
122
```

2. Riješite prethodni zadatak tako da se podaci iz datoteke mogu prenijeti u program `prosjek.py` pomoću operatora "|":

```
> cat podaci.txt | py prosjek.py
122
```

3. Napište program `filter.py` koji iz tekstualne datoteke u kojoj se nalazi nekakav tekst (primjerice, članak s nekog portala) izdvaja samo one rečenice u kojima se nalaze zadane riječi i te rečenice sprema u drugu tekstualnu datoteku čije ime je zadano kao parametar komandne linije. Taj program treba pokretati na ovaj način:

```
> py filter.py rezultat.txt cesta more telefon
```

Prvi parametar nakon naziva programa datoteka je u koju će biti spremljene rečnice koje sadrže riječi "cesta", "more" i "telefon" (sve te riječi su također zadane kao parametri komandne linije).

10.4 Rekurzija

1. Napišite rekurzivnu funkciju *okreni(p)* koja okreće listu *p*. Primjer:

```
>>> okreni([1, 2, 3])
[3, 2, 1]
```

2. Proširite funkciju *gen_kombinacije* tako da generira kombinacije koje se sastoje samo od zadanog broja elemenata. Na primjer, za zadani broj elemenata 2 i niz `(a, b, c)` ova funkcija trebala bi vratiti samo `{('a', 'c'), ('b', 'c'), ('a', 'b')}`.

3. U sustavu ISVU predmet može imati druge predmete koji su mu ekvivalenti. Napišite rekurzivnu funkciju koja će izdvojiti sve predmete koji spadaju u istu skupinu ekvivalenata za neki zadani predmet. Ulazni podaci će biti oblika <ISVU šifra 1>:<ISVU šifra 2>, što znači da su predmeti s ISVU šiframa 1 i 2 ekvivalenti. Na primjer,

```
190:200
199:400
118:105
190:140
120:100
105:132
144:145
105:199
```

Sada bi prema gornjem primjeru za predmet 118 ovaj program izdvojio sljedeće ekvivalente: [105, 132, 199, 400].

4. Napišite rekurzivnu funkciju koja za niz brojeva ispisuje sve kombinacije tih brojeva čiji je zbroj neka zadana vrijednost. Na primjer, za niz [5, 2, 1, 2, 1, 8, 4] i vrijednost 4 program treba ispisati [[2, 2], [1, 1, 2], [1, 1, 2], [4]].

5. Napišite rekurzivnu funkciju koja generira sve liste zadane duljine tako da se ispišu sve kombinacije dvaju elemenata (slova, znamenke i sl.). Na primjer, za elemente 0 i 1 i duljinu 3 ta funkcija treba vratiti/ispisati sljedeće:

```
[0, 0, 0]
[1, 0, 0]
[0, 1, 0]
[1, 1, 0]
[0, 0, 1]
[1, 0, 1]
[0, 1, 1]
[1, 1, 1]
```

   Na jednostavnom primjeru prikažite rekurzivno stablo.

6. Napišite rekurzivnu funkciju koja će ispisati sve načine na koje se može proći N stepenica ako se možemo penjati po jednu ili dvije stepenice odjednom. Na primjer, tri stepenice mogu se proći na tri načina: 1-1-1, 1-2 i 2-1.

7. Neka je zadan broj N i dvije operacije:

   (a) `dodaj 1` (*x+1*)
   (b) `udvostruči` (*x*2*)

   gdje je x rezultat prethodne operacije. Napišite rekurzivnu funkciju koja će pronaći minimalni broj operacija da bi se došlo do broja N počevši od 0. Neka je, na primjer, N = 7. Tada je minimalni niz operacija 0+1=1, 1+1=2, 1+2=3, 3*2=6, 6+1=7, što je ukupno pet operacija. Međutim, sljedeći niz operacija nije minimalan: 0+1=1, 1+1=2, 2*2=4, 4+1=5, 5+1=6, 6+1=7, što daje ukupno šest operacija. Ovdje je problem u tome što nakon množenja 2*2=4 moramo imati još tri zbrajanja. **Program treba ispisati pronađeni minimalni niz operacija.**

8. Neka je zadan niz od N brojeva gdje svaki broj predstavlja najveći pomak udesno koji možemo napraviti od indeksa tog broja (s tim da pomak može biti i manji od tog broja). Napišite rekurzivnu funkciju koja će dati najmanji broj skokova potrebnih da se dođe do kraja niza počevši od prvog broja, gdje broj 0 označava kraj. Na primjer, za niz [2, 4, 1, 1, 1] najmanji broj skokova bio bi 2 ako idemo kroz brojeve 2, 4, 1, gdje od broja 2 napravimo skok za jedno mjesto udesno, a od broja 4 napravimo skok za tri mjesta udesno.

9. Neka je zadana matrica veličine NxM i niz slova A, B, C, ... Napišite rekurzivnu funkciju koja će posložiti zadani niz slova tako da se dva ista slova ne nalaze jedno

pored drugoga po vertikali, horizontali ili dijagonali. Na primjer, za matricu 4x2 i niz slova A, B, C i D, kombinacija

```
ABCD
DCBA
```

nije dobro rješenje jer su dva slova B jedno pored drugoga (po dijagonali), isto kao i dva slova C. Međutim, sljedeće rješenje je dobro:

```
ABCD
CDAB
```

10. Neka je zadana matrica veličine NxN u kojoj je u svakom polju različit broj. Napišite rekurzivnu funkciju koja će u takvoj matrici pronaći najdulji put od jedne vrijednosti do druge tako da se taj put sastoji samo od vrijednosti koje rastu za 1 (počevši od neke polazne vrijednosti) i da se nalaze na poljima koja su jedno do drugoga samo po vertikali ili horizontali. Drugim riječima, za polazno polje *(i, j)* sljedeće polje može bit jedno od *(i+1, j), (i-1, j), (i, j+1), (i, j-1)*. U sljedećem primjeru zadana je matrica 3x3:

$$\begin{bmatrix} 7 & 8 & 9 \\ 2 & 5 & 6 \\ 3 & 4 & 1 \end{bmatrix}$$

Najdulji put koji možemo dobiti je duljine 5 za vrijednosti 2-3-4-5-6, počevši od polja *(1, 2)* s vrijednošću 2.

11. Napišite program koji traži izlaz iz labirinta u kojem se može ići u krug (vidi sliku 9.13).

12. Za problem u dijelu 9.2.1 napišite ili upotrijebite program koji generira sve permutacije i odabire one koje zadovoljavaju zadane uvjete. Nakon toga usporedite njegovo vrijeme izvršavanja s programom opisanim u dijelu 9.2.1. Koji od navedenih programa je brži?

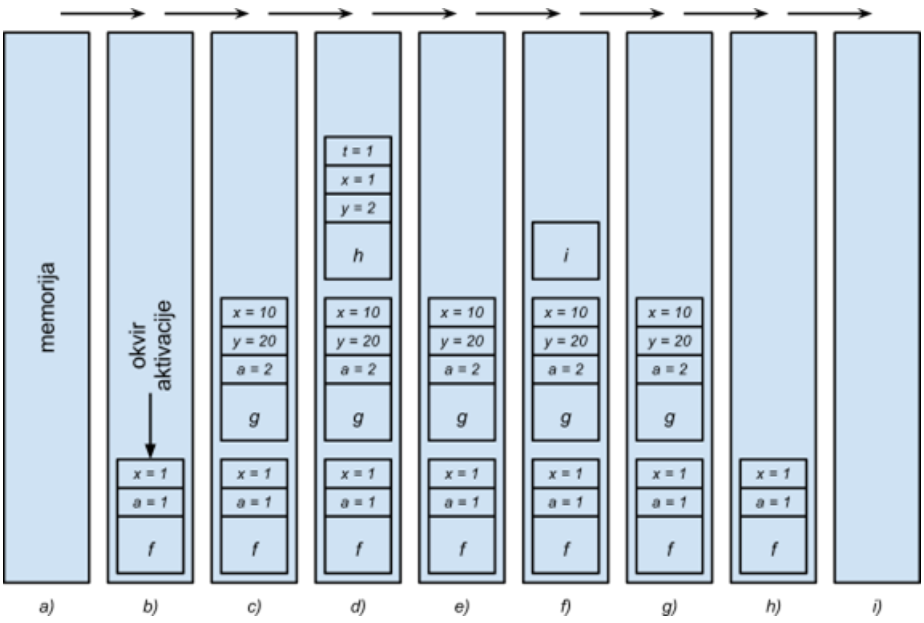
DODATAK

A Tehnički detalji poziva funkcija

S obzirom da su funkcije osnovni elementi svakog programa, korisno je poznavati načelo po kojem se odvija poziv funkcija. Ovdje ćemo ga ukratko proučiti uz nekoliko novih pojmova koje je važno razumjeti. Za potrebe ilustracije koristit ćemo sljedeći program:

```
1  def f(x):
2      a = 1
3      g(10, 20)
4      print(x, a)
5
6  def g(x, y):
7      a = 2
8      h(1)
9      i()
10     print(x, y, a)
11
12 def h(t):
13     x = 1
14     y = 2
15     print(t)
16
17 def i():
18     print('operacija i')
19
20 f(1)
```

Na slici A.1 vanjski okviri predstavljaju memoriju računala. Kod svakog poziva funkcije interpreter formira tzv. *okvir aktivacije* za tu funkciju. Detalji oko toga koje sve informacije sadrži jedan okvir aktivacije zavise od programskog jezika i nekih drugih tehničkih faktora. Ovdje ćemo se, međutim, usredotočiti na ono što je zajedničko mnogim programskim jezicima kao što je Python, bez ulaženja u previše detalja koji nisu važni za razumijevanje načela o kojima je ovdje riječ. Jedan aktivacijski okvir obično sadrži podatke kao što su vrijednosti parametara i lokalnih varijabli. Tijekom izvršavanja neke funkcije vrijednosti njenih lokalnih varijabli (uključujući i parametre) uzimaju se samo iz njenog okvira aktivacije. Ti okviri nisu vidljivi programeru - oni samo služe kao mehanizam koji u pozadini upravlja pozivima funkcija i koji je dio implementacije programskog jezika. Načelo poziva funkcije sada ćemo proučiti na gornjem primjeru. Prije poziva *f(1)* imamo situaciju kao na slici A.1-a, gdje je dio memorije



Slika A.1: Formiranje okvira aktivacije tijekom poziva funkcija.

rezerviran za okvire aktivacije prazan. Nakon poziva *f(1)* u memoriji će biti formiran okvir aktivacije za funkciju *f*, što je pokazano na slici A.1-b. Vidimo da taj okvir sadrži vrijednost parametra x i lokalne varijable a ove funkcije. Nakon poziva funkcije f slijedi poziv funkcije g i to iz funkcije *f*. To znači da okvir aktivacije za funkciju f mora biti sačuvan u memoriji jer će se izvršavanje u njoj nastaviti nakon povratka iz funkcije g. Za poziv funkcije g formira se novi okvir (na sljedećem slobodnom segmentu memorije) gdje on također sadrži vrijednosti parametara i lokalnih varijabli, što je pokazano na slici A.1-c. Izvršavanje se sada odvija u funkciji g, gdje se koriste vrijednosti za varijable x, y i a isključivo iz tog okvira. Sada iz funkcije g slijedi poziv funkcije h tako da se ovdje događa isto što i u prethodnom slučaju - sačuva se okvir za g i formira novi za h, nakon čega imamo situaciju kao na slici A.1-d. Nakon povratka iz funkcije h njen okvir više nije potreban. Izvršavanje se sada nastavlja u funkciji g, ponovno koristeći vrijednosti varijabli u njenom okviru, što je prikazano na slici A.1-e. Nakon toga, opet iz funkcije g, slijedi poziv funkcije i za koju se formira novi okvir. S obzirom da ova funkcija nema parametre i lokalne varijable njen okvir ne sadrži te podatke, kao što se vidi na slici A.1-f. Nakon povratka iz i izvršavanje se opet nastavlja u funkciji g kako je pokazano na slici A.1-g gdje se koristi njen okvir aktivacije, dok okvir za i više nije potreban. Nakon toga slijedi povratak iz g u f, što znači da se opet vraćamo na okvir za f koristeći tamošnje vrijednosti za x i a, što se vidi na slici A.1-h. Nakon povratka iz f okvir te funkcije više nije potreban pa se time vraćamo na prvobitno stanje pokazano na slici A.1-i, koje je identično onome na slici A.1-a. Jedan niz okvira aktivacije, kao što je onaj na slici A.1-d, zove se *stog poziva* i po njemu se vidi kako je tekao poziv funkcija do određenog trenutka. Na primjer, na slici A.1-f na osnovu stoga poziva vidimo da se

u funkciju *i* došlo iz funkcije *g*, a u funkciju *g* iz funkcije *f*. Stog poziva, međutim, ne pokazuje koja se operacija prethodno izvršavala; na primjer, stog poziva na slici A.1-e ne pokazuje da smo se iz funkcije *h* vratili u funkciju *g* - u ovom slučaju stog poziva izgleda isto kao i oni na slikama A.1-c i A.1-g. Stog poziva, dakle, pokazuje samo posljednji niz poziva funkcija, ali ne i povratke iz njih. Svi moderni alati za programiranje mogu pokazati ovakav stog poziva funkcija u nekom trenutku izvršavanja programa (iako u više tekstualnom obliku), što programeru znatno olakšava analizu njegovog izvršavanja.

Dio memorije koji je rezerviran za okvire aktivacije često se naziva stogom jer načelo dodavanja i uklanjanja aktivacijskih okvira odgovara onom kod istoimene strukture podataka.

Popis slika

Popis primjera

Popis tablica

Bibliografija

[1] Ronald Mak. *Writing Compilers and Interpreters: A Software Engineering Approach*. 3rd ed. Wiley, 2009. ISBN: 978-0-47017-707-5.

[2] Alfred Aho et al. *Compilers: Principles, Techniques, and Tools*. 2nd ed. Addison Wesley, 2006. ISBN: 978-0-32148-681-3.

[3] Frank Vahid. *Embedded System Design: A Unified Hardware / Software Introduction*. Wiley, 2001. ISBN: 978-0-47138-678-0.

[4] Doug Hellmann. *The Python Standard Library by Example*. Pearson, 2011. ISBN: 978-0-32176-734-9.

[5] Wes McKinney. *Python for Data Analysis*. 2nd ed. O'Reilly Media, 2018. ISBN: 978-1-49195-766-0.

[6] Julia Elman and Mark Lavin. *Lightweight Django: Using REST, WebSockets, and Backbone*. O'Reilly Media, 2014. ISBN: 978-1-49194-594-0.

[7] Andreas C. Müller and Sarah Guido. *Introduction to Machine Learning with Python*. O'Reilly Media, 2017. ISBN: 978-1-44936-941-5.

[8] Aurélien Géron. *Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow*. 2nd ed. O'Reilly Media, 2019. ISBN: 978-1-49203-264-9.

[9] Davy Cielen, Arno D. B. Meysman, and Mohamed Ali. *Introducing Data Science*. Manning, 2016. ISBN: 978-1-63343-003-7.

[10] Francois Chollet. *Deep Learning with Python*. Manning, 2021. ISBN: 978-1-61729-686-4.

[11] Laurence Moroney. *AI and Machine Learning for Coders: A Programmer's Guide to Artificial Intelligence*. O'Reilly Media, 2020. ISBN: 978-1-49207-819-7.

[12] Jessica McKellar. *Twisted Network Programming Essentials: Event-driven Network Programming with Python*. 2nd ed. O'Reilly Media, 2013. ISBN: 978-1-44932-611-1.

[13] Mark Lutz. *Programming Python*. 4th ed. O'Reilly Media, 2011. ISBN: 978-0-59615-810-1.

[14] Towards data science. 2021. URL: `https://towardsdatascience.com/top-programming-languages-for-ai-engineers-in-2020-33a9f16a80b0`.

[15] Wikipedia. 2022. URL: `https://en.wikipedia.org/wiki/Open_source`.

[16] Mark Lutz. *Learning Python*. 5th ed. O'Reilly Media, 2013. ISBN: 978-1-44935-573-9.

[17] Daniel P. Friedman. *Essentials of Programming Languages*. 3rd ed. The MIT Press, 2008. ISBN: 978-0-26206-279-4.

[18] David A. Patterson and John L. Hennessy. *Computer Organization and Design*. 2nd ed. Morgan Kaufmann, 2020. ISBN: 978-0-12820-331-6.

[19] Robert W. Sebesta. *Concepts of Programming Languages*. 11th ed. Pearson, 2015. ISBN: 978-0-13394-302-3.

[20] Michael L. Scott. *Programming Language Pragmatics*. 4th ed. Morgan Kaufmann, 2015. ISBN: 978-0-12410-409-9.

[21] Luciano Ramalho. *Fluent Python*. O'Reilly Media, 2015. ISBN: 978-1-49194-600-8.

[22] David Beazley and Brian K. Jones. *Python Cookbook*. O'Reilly Media, 2013. ISBN: 978-0-13403-428-7.

[23] Allen B. Downey. *Think Python: How to Think Like a Computer Scientist*. 2nd ed. O'Reilly Media, 2016. ISBN: 978-1-49193-936-9.

[24] Mark Lutz. *Python Pocket Reference*. 5th ed. O'Reilly Media, 2014. ISBN: 978-1-44935-701-6.

[25] Alex Martelli, Anna Ravenscroft, and Steve Holden. *Python in a Nutshell*. 3rd ed. O'Reilly Media, 2017. ISBN: 978-1-44939-292-5.

[26] Thomas H. Cormen et al. *Introduction to Algorithms*. 3rd ed. The MIT Press, 2009. ISBN: 978-0-26203-384-8.

[27] Brett Slatkin. *Effective Python*. Addison-Wesley, 2015. ISBN: 978-0-13403-428-7.

[28] Steven F. Lott. *Functional Python Programming*. 2nd ed. Packt Publishing, 2018. ISBN: 978-1-78862-706-1.

[29] Neal Ford. *Functional Thinking*. O'Reilly Media, 2014. ISBN: 978-1-44936-551-6.

[30] Bernd Bruegge and Allen H. Dutoit. *Object-Oriented Software Engineering*. 3rd ed. Prentice Hall, 2010. ISBN: 978-0-13606-125-0.

[31] Bertrand Meyer. *Object-Oriented Software Construction*. 2nd ed. Pearson College Div, 2000. ISBN: 978-0-13629-155-8.

[32] Bjarne Stroustrup. *Programming: Principles and Practice Using C++*. 2nd ed. Addison-Wesley, 2014. ISBN: 978-0-32199-278-9.

[33] Andrew S. Tanenbaum and Herbert Bos. *Modern Operating Systems*. 4th ed. Pearson, 2014. ISBN: 978-0-13359-162-0.

[34] Lee Holmes. *PowerShell Cookbook*. 4th ed. O'Reilly Media, 2021. ISBN: 978-1-09810-160-2.

[35] Cameron Newham. *Learning the bash Shell*. 3rd ed. O'Reilly Media, 2005. ISBN: 978-0-59600-965-6.

[36] Brian W. Kernighan and Rob Pike. *The Unix Programming Environment*. Prentice-Hall, 1983. ISBN: 978-0-13937-681-8.

[37] John V. Guttag. *Introduction to Computation and Programming Using Python*. 2nd ed. The MIT Press, 2016. ISBN: 978-0-26252-962-4.

[38] Michael T. Goodrich, Roberto Tamassia, and Michael H. Goldwasser. *Data Structures and Algorithms in Python*. Wiley, 2013. ISBN: 978-1-11829-027-9.

[39] David Kopec. *Classic Computer Science Problems in Python*. Manning, 2019. ISBN: 978-1-61729-598-0.

[40] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. 2nd ed. The MIT Press, 1996. ISBN: 978-0-26251-087-5.

[41] Aleksandar Stojanović. *Elementi računalnih programa*. Element, 2012. ISBN: 978-9-53197-616-9.

[42] Kenneth H. Rosen. *Discrete Mathematics and Its Applications*. McGraw-Hill, 2012. ISBN: 978-0-07338-309-5.

Ključne riječi