

Vladislav Mišković



OSNOVE PROGRAMIRANJA PYTHON



UNIVERZITET SINGIDUNUM
FAKULTET ZA INFORMATIKU I RAČUNARSTVO
TEHNIČKI FAKULTET

Vladislav Miškovic

OSNOVE PROGRAMIRANJA - PYTHON

Treće izmenjeno i dopunjeno izdanje

Beograd, 2020

OSNOVE PROGRAMIRANJA - PYTHON

Autor:

dr Vladislav Mišković

Recenzenti:

dr Milan Milosavljević

dr Mladen Veinović

dr Boško Nikolić

Izdavač:

UNIVERZITET SINGIDUNUM

Beograd, Danijelova 32

www.singidunum.ac.rs

Za izdavača:

dr Milovan Stanišić

Tehnička obrada:

Vladislav Mišković

Jelena Petrović

Dizajn korica:

Aleksandar Mihajlović

Godina izdanja:

2020.

Tiraž:

1600 primeraka

Štampa:

Caligraph, Beograd

ISBN: 978-86-7912-728-0

Copyright:

© 2020. Univerzitet Singidunum

Izdavač zadržava sva prava.

Reprodukcija pojedinih delova ili celine ove publikacije nije dozvoljena.

Sadržaj

1. Uvod u programiranje i jezik Python	1
2 Osnovni elementi programa	23
3 Izrazi u jeziku Python	39
4 Upravljanje tokom izvršavanja programa: grananje i ponavljanje	54
5 Funkcije u jeziku Python.....	66
6 Rekurzija u jeziku Python	76
7 Osnovne strukture podataka u jeziku Python.....	96
8 Polja i neuređene liste u jeziku Python	114
9 Organizacija programskog koda u jeziku Python	139
10 Rad s fajlovima u jeziku Python	150
11 Analiza algoritama, pretraživanje i sortiranje u jeziku Python	164
12 Osnove objektno orijentisanog programiranja u jeziku Python	181
Literatura	212
Prilozi	214

Sadržaj (detaljniji)

1. Uvod u programiranje i jezik Python	1
1.1 Uvod	1
1.1.1 Računanje i računari.....	2
1.1.2 Algoritam.....	5
1.2 Programski jezici	6
1.2.1 Izvršavanje programa	7
1.2.2 Vrste programskih jezika.....	8
1.2.3 Sintaksa i semantika programskih jezika	9
1.2.4 Bakusova normalna forma.....	10
1.3 Proces razvoja programa	11
1.3.1 Razvoj i implementacija algoritama.....	11
1.3.2 Testiranje programa.....	11
1.4 Programski jezik Python.....	12
1.4.1 Upotreba jezika (zašto Python).....	14
1.4.2 Instalacija.....	14
1.4.3 Osnovni elementi jezika Python.....	16
1.5 Primeri malih programa.....	20
1.5.1 Pozdrav.....	20
1.5.2 Putovanje.....	20
1.6 Poznate aplikacije	21
2 Osnovni elementi programa	23
2.1 Uvod	23
2.2 Promenljive i dodela vrednosti	26
2.2.1 Naredba dodele vrednosti	26
2.2.2 Oblast definisanosti promenljivih	28
2.2.3 Kompleksni brojevi	28
2.3 Izrazi	29
2.3.1 Evaluacija izraza	29
2.3.2 Operatori.....	29
2.4 Grananje (selekcija)	30
2.5 Ponavljanje (iteracija).....	32
2.6 Ugrađene funkcije.....	33
2.7 Korisničke funkcije	34
2.8 Ulaz-izlaz	34
2.9 Primeri programa	36
2.9.1 Indeks telesne mase.....	36

2.9.2	Indeks telesne mase s ocenom stanja.....	37
3	Izrazi u jeziku Python	39
3.1	Uvod	39
3.2	Izrazi	39
3.3	Operatori.....	41
3.3.1	Aritmetički operatori	41
3.3.2	Operatori poređenja.....	43
3.3.3	Specijalni operatori uz dodelu vrednosti.....	43
3.3.4	Logički operatori.....	44
3.3.5	Operatori za rad s bitovima.....	46
3.3.6	Operatori pripadnosti.....	47
3.3.7	Operatori identifikacije.....	48
3.4	Prioritet operatora	48
3.5	Konverzija tipova.....	49
3.6	Zaokruživanje	50
3.7	Primer programa	50
4	Upravljanje tokom izvršavanja programa: grananje i ponavljanje	54
4.1	Uvod	54
4.1.1	Grananje u proceduralnim programskim jezicima.....	55
4.1.2	Ponavljanje u proceduralnim jezicima.....	55
4.2	Grananje (selekcija)	56
4.3	Ponavljanje (iteracija).....	57
4.3.1	Naredba for	57
4.3.2	Naredba while.....	58
4.3.3	Naredbe za prekid ponavljanja.....	58
4.4	Obrada izuzetaka.....	60
4.5	Primeri programa	62
4.5.1	Euklidov algoritam.....	62
4.5.2	Igra pogađanja.....	63
5	Funkcije u jeziku Python.....	66
5.1	Uvod	66
5.2	Definicija korisničke funkcije.....	67
5.3	Upotreba funkcije.....	69
5.4	Argumenti funkcije	69
5.4.1	Prenos argumenata po vrednosti	70
5.4.2	Varijabilna lista argumenata	70
5.5	Prostori imena i oblast važenja promenljivih.....	71
5.6	Anonimne funkcije.....	72

5.7	Primer programa	73
6	Rekurzija u jeziku Python	76
6.1	Uvod	76
6.2	Definicija rekurzivne funkcije	76
6.3	Izvršavanje rekurzivne funkcije	78
6.3.1	Upotreba sistemskog steka.....	79
6.4	Primeri rekurzivnih algoritama	81
6.4.1	Fibonačijev niz.....	81
6.4.2	Vektorska grafika u jeziku Python (turtle graphics).....	83
6.4.3	Fraktali i fraktalne funkcije.....	85
6.5	Rekurzivne i iterativne verzije algoritama	86
6.6	Primeri programa	88
6.6.1	Fraktalno stablo.....	88
6.6.2	Trougao Serpinskog	90
6.6.3	Pahuljica	92
7	Osnovne strukture podataka u jeziku Python.....	96
7.1	Uvod: nizovi u jeziku Python	96
7.2	Stringovi	97
7.3	Liste.....	100
7.3.1	Operacije nad listama	101
7.3.2	Iteracija nad listama	101
7.3.3	Linearno pretraživanje liste.....	102
7.3.4	Dodela vrednosti i kopiranje lista.....	103
7.3.5	Lista kao struktura stek.....	103
7.3.6	Skraćeno generisanje lista.....	104
7.4	N-torke	105
7.5	Osnovne operacije nad nizovima (sekvencama)	105
7.5.1	Sekvence kao argumenti funkcije.....	106
7.5.2	Sekvence i podrazumevane (default) vrednosti argumenata	107
7.5.3	Sekvence kao rezultati funkcija.....	107
7.6	Primeri programa	107
7.6.1	Kineski astrološki kalendar.....	108
7.6.2	Eratostenovo sito.....	110
8	Polja i neuređene liste u jeziku Python	114
8.1	Uvod	114
8.2	Polja	114
8.2.1	Ugnježdene liste.....	115
8.2.2	Predstavljanje matrica	115

8.2.3	Suma svih elemenata matrice.....	116
8.2.4	Množenje matrica.....	116
8.2.5	Predstavljanje hijerarhija (stabala)	118
8.2.6	Obilazak stabla.....	119
8.2.7	Binarna stabla	120
8.3	Neuređene liste: rečnici i skupovi	120
8.3.1	Struktura rečnik (dictionary).....	121
8.3.2	Primena operatora 'in' na strukturu rečnika.....	121
8.3.3	Kombinovanje struktura rečnika i uređene liste.....	122
8.3.4	Spisak promenljivih u jeziku Python.....	122
8.3.5	Struktura skup (set)	123
8.4	Primeri programa	123
8.4.1	Geografija	124
8.4.2	Najbliži gradovi (Dijkstrin algoritam).....	125
8.4.3	Igra pogađanja reči (hangman).....	129
9	Organizacija programskog koda u jeziku Python	139
9.1	Uvod	139
9.2	Upotreba modula u jeziku Python.....	141
9.3	Specifikacija modula.....	141
9.4	Projektovanje softvera s vrha (top-down).....	142
9.5	Moduli u jeziku Python	143
9.5.1	Prostor imena modula.....	144
9.5.2	Moduli i paketi.....	145
9.5.3	Upotreba modula.....	145
9.6	Primer programa	146
10	Rad s fajlovima u jeziku Python	150
10.1	Uvod	150
10.2	Pristup fajlu	151
10.3	Učitavanje teksta.....	151
10.4	Učitavanje numeričkih podataka.....	152
10.5	Upis teksta na fajl.....	153
10.5.1	Operator formatiranja	154
10.5.2	Funkcija i metod format	155
10.5.3	Formatirani stringovi	155
10.6	Strukturirani podaci u formatu JSON.....	156
10.7	Čitanje teksta s Interneta.....	157
10.8	Primeri programa	159
10.8.1	Brojanje različitih slova u tekstu fajla	159

10.8.2	Program za konverziju valuta	160
11	Analiza algoritama, pretraživanje i sortiranje u jeziku Python	164
11.1	Uvod	164
11.2	Složenost algoritama	165
11.2.1	Ocena složenosti algoritama	166
11.2.2	Poređenje vremenske složenosti algoritama	167
11.3	Algoritmi pretraživanja	168
11.3.1	Linearno pretraživanje liste.....	168
11.3.2	Binarno pretraživanje sortirane liste.....	169
11.4	Algoritmi sortiranja.....	170
11.4.1	Sortiranje selekcijom (Selection Sort).....	170
11.4.2	Sortiranje umetanjem (Insertion Sort)	171
11.4.3	Sortiranje mehurom (Bubble Sort)	172
11.5	Primeri programa	174
11.5.1	Program Indeks pojmova.....	174
11.5.2	Poređenje performansi struktura liste i skupa	178
12	Osnove objektno orijentisanog programiranja u jeziku Python	181
12.1	Uvod	181
12.1.1	Objektno orijentisani razvoj softvera.....	181
12.1.2	Objektno orijentisano programiranje.....	182
12.2	Objekti i klase	183
12.2.1	Kreiranje objekata	184
12.2.2	Definisanje korisničke klase	185
12.2.3	Kreiranje objekata pomoću konstruktora.....	186
12.2.4	Pristup članovima objekta.....	186
12.2.5	Parametar self	187
12.3	Skrivanje podataka	188
12.4	Grafička notacija za opis klasa u jeziku UML	189
12.5	Nasleđivanje	191
12.6	Nadjačavanje metoda (override)	196
12.7	Polimorfizam	196
12.7.1	Dinamičko povezivanje	197
12.8	Preklapanje operatora	199
12.9	Kreiranje klasa na osnovu veza.....	201
12.10	Primeri objektno orijentisanih programa	203
12.10.1	Klasa IndeksTelesneMase.....	203
12.10.2	Program IndeksTelesne Mase (objektno orijentisana verzija).....	206
12.10.3	Osnovni elementi biblioteke Tkinter	207

12.10.4 Program za unos podatka preko grafičkog interfejsa.....	208
Literatura	212
Prilozi.....	214

Popis slika

Sl. 1. Prvi programabilni računar Analytical Engine, XIX vek.....	2
Sl. 2. Profesor Čarls Bebidž i njegov asistent Ada Bajron, prvi programer	3
Sl. 3. Matematički model univerzalnog računara (Tjuringova mašina)	3
Sl. 4. Fon Nojmanova arhitektura računara (implementacija Tjuringove mašine)	4
Sl. 5. Prva automatizovana obrada podataka.....	4
Sl. 6. Proces izvršavanja programa pomoću virtuelne mašine	5
Sl. 7. Dijagram toka Euklidovog algoritma	6
Sl. 8. Primer komplilacije i interpretacije izvornog programa	7
Sl. 9. Sintaksa i semantika prirodnih jezika	10
Sl. 10. Dva načina definicije sintakse elementa programskog jezika	10
Sl. 11. Načini upravljanja redosledom izvršavanja naredbi programa.....	54
Sl. 12. Uslovno grananje ispitivanjem vrednosti opšteg tipa i samo binarnih vrednosti.....	55
Sl. 13. Višestruka dekompozicija programskog koda.....	67
Sl. 14. Sistemski stek funkcijskih poziva.....	69
Sl. 15. Sistemski stek poziva prilikom računanja rekurzivne funkcije	80
Sl. 16. Fibonačijev broj: prikaz rasta populacije zečeva za n=6 meseci	82
Sl. 17. Grafički prozor programa za crtanje pravougaonika i šestougla	84
Sl. 18. Primer crtanja u boji oblika cveta (zvezdice)	85
Sl. 19. Kohova kriva, zmajeva kriva i uvećan detalj Mandelbrotove krive.....	85
Sl. 20. Tipične strukture rekurzivnih funkcija.....	87
Sl. 21. Koraci isertavanja fraktalnog stabla	89
Sl. 22. Koraci isertavanja trougla Serpinskog	91
Sl. 23. Kohova kriva rekurzivne dubine 3	93
Sl. 24. Pahuljica na osnovu Kohove krive rekurzivne dubine 3	94
Sl. 25. Strukture podataka u operativnoj memoriji i na disku	96
Sl. 26. Tabela kodova znakova po ASCII standardu	99
Sl. 27. Prikaz više referenci na jednu strukturu liste.....	103
Sl. 28. Prikaz operacija nad strukturom stek.....	104
Sl. 29. Kineski zodijak.....	108
Sl. 30. Ilustracija primene metoda Eratostenovog sita.....	111
Sl. 31. Prikaz podataka o ispitu pomoću ugnježdne liste u dva nivoa	115
Sl. 32. Prikaz opšte hijerarhija zadane ugnježđenom listom	118
Sl. 33. Prikaz strukture stabla simbola zadane ugnježđenom listom	119
Sl. 34. Obilazak opšteg stabla	120
Sl. 35. Obilazak binarnog stabla.....	120
Sl. 36. Obim nekih poznatih programa.....	140
Sl. 37. Primer projektovanja s vrha (top-down)	142
Sl. 38. Primeri funkcija rasta.....	167
Sl. 39. Primer binarnog pretraživanja sortirane liste.....	170
Sl. 40. Princip sortiranja liste selekcijom.....	171
Sl. 41. Princip sortiranja liste umetanjem	172
Sl. 42. Primer binarnog stabla sortiranja za deo ulaznog teksta	174
Sl. 43. Klasična i objektno orijentisana organizacija softverskog sistema.....	182

Sl. 44. Odnos klase i objekta.....	184
Sl. 45. Kreiranje objekata klase Krug pomoću konstruktora.....	186
Sl. 46. Prikaz spiska članova klase prilikom unosa programskog koda.....	187
Sl. 47. Elementi UML dijagrama klase i objekata.....	190
Sl. 48. Primer osnovne i izvedenih klase	192
Sl. 49. Izbor konkretnog metoda u hijerarhiji nasleđivanja klase.....	198
Sl. 50. Primer dinamičkog povezivanja nasleđenog metoda.....	199
Sl. 51. Primeri različitih veza između klase.....	201
Sl. 52. Primer veza asocijacije između klase	202
Sl. 53. Primer veza kompozicije i agregacije između klase	203
Sl. 54. UML dijagram klase indeksTM	204
Sl. 55. Primer grafičkog korisničkog interfejsa	208
Sl. 56. Struktura klase FormaZaUnos.....	208
Sl. 57. Primer grafičkog interfejsa za unos podataka.....	210

Predgovor

Udžbenik je nastao kao osnovna literatura za uvodni kurs programiranja na određenim studijskim programima Fakulteta za informatiku računarstvo i Tehničkog fakulteta Univerziteta Singidunum u Beogradu.

Jezik Python izabran je kao prvi jezik za učenje osnovnih principa programiranja jer manje opterećuje programera obaveznim elementima programskog koda. Minimalistički zamišljen programski jezik omogućava preusmeravanje pažnje programera s problema kodiranja programa u konkretnom programskom jeziku na rešavanje problema, koje uključuje analizu, razvoj algoritama i primenu opštih principa programiranja.

Osnovni cilj materijala je razvoj sposobnosti rešavanja problema na računaru i upoznavanje konkretnog programskog jezika do nivoa koji omogućava izradu jednostavnijih konzolnih aplikacija. Istovremeno se postepeno uvodi objektno orijentisani pristup, upoznaje rad s tekstualnim fajlovima, kao i elementarna upotreba jednostavne vektorske grafike.

U materijalu se izlažu osnove sintakse i semantike aktuelne verzije programskog jezika Python. Upoznaju se osnovni tipovi podataka, upravljačke naredbe i naredbe za strukturiranje koda, linearne i nelinearne strukture podataka u memoriji i njihove kombinacije, kao i programski elementi koji omogućavaju razvoj objektno orijentisanih programa.

Prikazani su tipični primeri upotrebe različitih elemenata jezika, kojima se ilustruje pristup rešavanju određenih kategorija problema uz pomoć računara. U implementaciji rešenja uglavnom se koriste ugrađene funkcije i standardna biblioteka programa, izuzev za prikaz grafike.

Svako poglavlje udžbenika ima kratki teorijski uvod s ilustrativnim primerima, nakon čega se detaljnije izlažu ostali važniji elementi svake teme. Na kraju svakog poglavlja daju se primeri kompletnih programa, s opisom problema koji se rešava, strukture rešenja i prikazom rezultata njihovog izvršavanja. Uz svako poglavlje daju se i pitanja za proveru znanja, koja treba da pomognu u usmeravanju studenata na najvažnije aspekte određene teme.

Beograd, 01.09.2017. godine

Beograd, 31.10.2018. godine

Beograd, 20.08.2020. godine

1. Uvod u programiranje i jezik Python

1. Uvod
2. Programski jezici
3. Proces razvoja programa
4. Programski jezik Python
5. Primeri malih programa
6. Poznate aplikacije

U ovom poglavlju se izlažu osnovni pojmovi programiranja računara, programskih jezika i upotrebe jezika Python.

1.1 Uvod

Osnovna namena udžbenika je učenje načina rešavanja problema pomoću računara, kroz izradu i korišćenje odgovarajućeg računarskog programa.

Program ili softver (*software*) predstavlja niz instrukcija računaru ili nekom računarski zasnovanom uređaju kako da izvrši određeni zadatak, kojim se rešava neki problem [1], [2].

Softver se na današnjem nivou razvoja tehnologije nalazi u osnovi velikog broja uređaja koji se sreću u svakodnevnom životu, od personalnih računara, aviona, automobila i mobilnih telefona do već mašina, pegli i tosteri.

Razvoj softvera uprošćeno se naziva *programiranje*. Razvoj softvera podrazumeva upotrebu posebnih alata kao što su programski jezici. U ovom materijalu koristi se programski jezik Python, pre svega zbog jednostavnosti i brzine učenja, ali i velike popularnosti u različitim primenama [1], [2].

U višedecejskom razvoju računarstva kreiran je veliki broj programskih jezika opšte i posebne namene. Svaki novi programski jezik razvijen je s namerom da se unapredi razvoj softvera za određene namene, ali se nijedan od njih ne može izabrati kao najbolji i dovoljan za sve primene. Zbog toga programeri, koji rešavaju najrazličitije probleme, moraju da upoznaju veći broj programskih jezika i drugih alata za razvoj softvera. Povoljna okolnost je da se učenjem rešavanja problema pomoću jednog programskog jezika nauče i principi rešavanja problema koji su zajednički za veliki broj programskih jezika, pa je učenje novih jezika znatno brže i lakše [1].

Učenje osnovnih principa programiranja povezano je s poznavanjem ideje računanja i osnova funkcionisanja računara. Osnovne ideje i principi računarstva izučavaju se u okviru računarskih nauka (*Computer Science*).

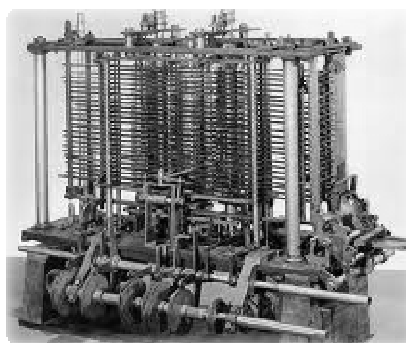
1.1.1 Računanje i računari

Računanje u smislu aritmetičkih operacija automatizovano je različitim uređajima, od antičkog abakusa do današnjih kalkulatora. Računar ili računska mašina (*Computing Machinery*) je programabilni uređaj koji automatizovano izvršava niz aritmetičkih ili logičkih operacija.

Algoritam je opis konačnog broja koraka koje treba izvršiti za zadani ulaz da se proizvede traženi rezultat za konačno vreme [2], [3]. Algoritmi su nezavisni od računara i programskih jezika, ali su računarski programi pogodni za njihovo efikasno izvršavanje.

Program je niz instrukcija za računar zapisan nizom simbola. Matematičar Alan Turing¹ izneo je ideju da se računarski program može upotrebiti za modeliranje ljudskog mišljenja, koja je vremenom prerasla u posebnu disciplinu računarskih nauka pod nazivom *veštačka inteligencija* (*Artificial Intelligence*) [3].

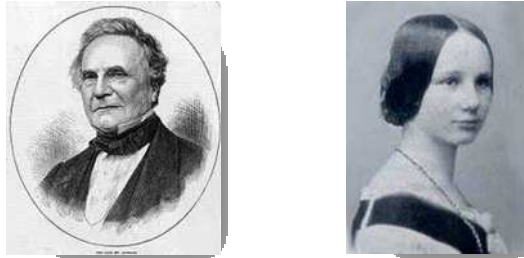
Univerzalni računar (*Computer*), koji može da izvršava bilo kakva računanja, osmišljen je u XIX, a realizovan u XX veku [1]. Prvi *programabilni* (*mehanički*) računar, tzv. *Analytical Engine*, osmislio je početkom XIX veka Čarls Bebidž, profesor matematike Univerziteta Kembridž [19]. Pethodno je konstruisao diferencijalnu mašinu (*Difference Engine*), mehanički kalkulator zasnovan na četiri osnovne aritmetičke operacije za izradu tabela vrednosti polinomskih funkcija, a pomoću njih i aproksimacija logaritamskih i trigonometrijskih funkcija. Analitička mašina, Sl. 1 [19], bio je predlog modela računara opšte namene, koji je imao memoriju, upravljanje tokom izvršavanja pomoću uslovnih naredbi i petlji, te ugrađenu memoriju, tako da je mogao da izračunava bilo koju funkciju, ako je bio poznat algoritam za njeno izračunavanje. Nijedan model nije realizovan u toku njegovog života.



Sl. 1. Prvi programabilni računar Analytical Engine, XIX vek

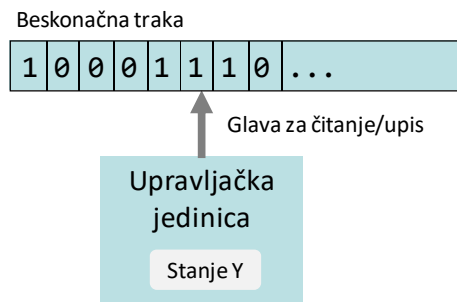
¹ Alan Tjuring (Alan Matheson Turing), engleski matematičar i kriptanalitičar (1912-1954)

Prvi *program* za ovaj računar napisala je Ada Avgusta Bajron, njegov asistent, Sl. 2, tako da se danas smatra i prvim programerom [19]. Po njoj je nazvan poznati programski jezik Ada (1980).



Sl. 2. Profesor Čarls Bebidž i njegov asistent Ada Bajron, prvi programer

Univerzalna *Tjuringova mašina* je teorijski uređaj koji manipuliše simbolima na beskonačnoj traci u skladu s tabelom pravila. Naziva se univerzalnom jer može da simulira svaki računarski algoritam [3], Sl. 3.



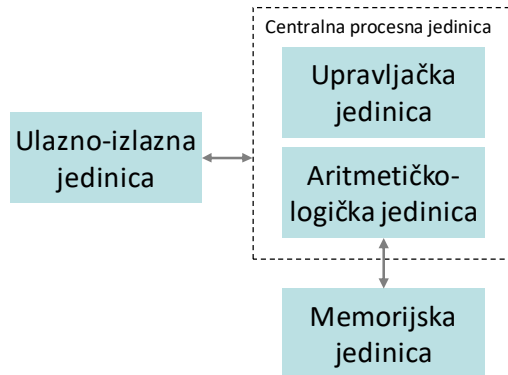
Sl. 3. Matematički model univerzalnog računara (Tjuringova mašina)

Većina savremenih računara opšte namene, koji predstavljaju implementaciju ove ideje, zasnivaju se na modelu realizacije računara koji je izložio Džon fon Nojman (John von Neumann).

Njegova osnovna ideja je da računar izvršava promenljivi program, koji je smešten u memoriju zajedno s potrebnim podacima. Računar ima sledeće komponente, Sl. 4:

- *Memorija* se sastoji od registara kojima se može *direktno* pristupiti (*Random Access Memory*, RAM) radi čitanja ili upisa instrukcija i podataka.

- *Procesor* računara sastoji se od upravljačke i aritmetičko-logičke jedinice (ALU). Čita i izvršava instrukcije programa.
- *Ulazno-izlazna* jedinica ostvaruje vezu računara s okolinom u koju spadaju različiti ulazni i izlazni uređaji, kao što su ekran i tastatura.



Sl. 4. Fon Nojmanova arhitektura računara (implementacija Tjuringove mašine)

Automatizovano računanje i obrada podataka nastali su pre pojave ovakvih univerzalnih elektronskih računara. Prva automatizovana obrada velikog obima podataka je obrada rezultata popisa stanovništva u SAD 1890. godine [19], [20]. Postupak pripreme i obrade podataka popisa, kao i odgovarajuće uređaje za automatizaciju patentirao je američki pronalazač Herman Holerit, osnivač kompanije koja je 1924. godine prerasla u korporaciju IBM (*International Business Machines*). Podaci o stanovnicima kodirani su na bušene papirne kartice, a obrada podataka vršena je korišćenjem čitača bušenih kartica, Sl. 5 [19]. Programiranje se sastojalo od izbora vrste prebrojavanja pomoću preklopnika. Vreme trajanja obrade podataka popisa stanovništva smanjeno je sa 10 godina na svega 6 nedelja.



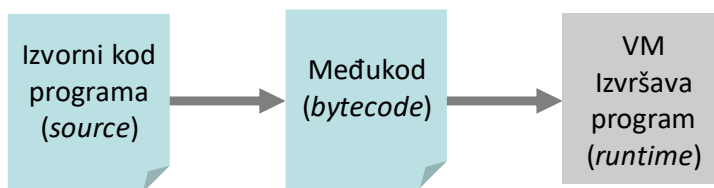
Sl. 5. Prva automatizovana obrada podataka

Prva poslovna obrada podataka na digitalnom elektronskom računar *Univac 1* takođe se odnosi na podatke popisa stanovništva u SAD, 1950. godine. Za ovaj računar razvijen je prvi prevodilac za programski jezik nazvan A1 [19].

Računarski program

Računarski program je skup naredbi u nekom programskom jeziku za izvršenje određenog zadatka na računar. Programski jezici su tako koncipirani da budu razumljivi čoveku koji programira računar. Programi koje kreiraju programeri nazivaju se *izvorni* programi ili izvorni kod (*source code*). No, procesor računara izvršava samo binarno kodirane instrukcije, prilagođene efikasnoj implementaciji hardvera procesora, koje su čoveku teško razumljive i sasvim nepraktične za izradu bilo kakvih složenijih programa. Programi u binarnom jeziku procesora nazivaju se *mašinski* programi ili mašinski kod (*machine code*). Da bi se izvorni program mogao izvršiti, potrebno ga je *prevesti* iz izvornog u mašinski kod.

Programi se mogu prevoditi pre ili u toku njihovog izvršavanja. Prevođenje pre izvršavanja naziva se kompilacija, a prevođenje i istovremeno izvršavanje se naziva interpretacija. Prevođenje se može izvršiti direktno u mašinski jezik ili u neki međukod (*bytecode*). Međukod se izvršava pomoću posebnog programa, tzv. virtuelne mašine (VM), koja se instalira na određenom računar samo jednom za sve programe prevedene u odgovarajući međukod, Sl. 6.



Sl. 6. Proces izvršavanja programa pomoću virtuelne mašine

1.1.2 Algoritam

Algoritam je opis postupka za rešavanje nekog problema. Reč *algoritmi* nastala je od latinskog prevoda prezimena persijskog naučnika, astronoma i matematičara iz IX veka Al Horezmija [2], [4].

Računarski algoritam je opis konačnog broja koraka za rešavanje nekog problema na računar. Algoritam je nezavisan od programskog jezika u kome će se realizovati. Algoritam se može opisati na različite načine, npr. kao tekst u prirodnom jeziku (narativni opis), strukturirani tekst (pseudokod) ili kao grafički prikaz, npr. pomoću dijagrama toka (*flowchart*) ili UML dijagrama [1].

Primer: Opis Euklidovog algoritma

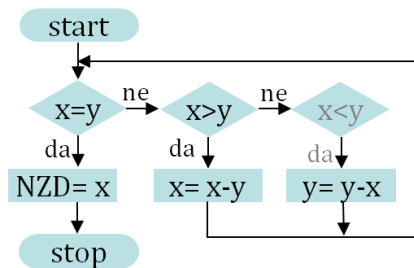
Euklidov algoritam za računanje najmanjeg zajedničkog delioca dva pozitivna cela broja $NZD(x,y)$ jedan je od najstarijih algoritama, opisan još u čuvenom Euklidovom delu *Elementi* [4]. Njegov narativni opis veoma je kratak:

Sve dok su dva broja različita, oduzimaj manji broj od većeg

Pseudokod ili strukturirani tekst opisa algoritma može biti, npr.

*Sve dok je $x \neq y$,
 ako je $x > y$ onda $x = x - y$,
 ako je $x < y$ onda $y = y - x$*

Grafički prikaz istog algoritma pomoću dijagrama toka prikazan je na Sl. 7.



Sl. 7. Dijagram toka Euklidovog algoritma

1.2 Programski jezici

Programski jezik je formalno definisan jezik za pisanje programskog koda u obliku niza naredbi koje računar prevodi i izvršava.

Prvi programski jezik s prevodiocem za računar *Univac 1* je jezik A1, a prva poslovna primena računara je elektronska obrada podataka popisa stanovništva SAD. Autor programskog jezika A1 bila je Grejs Meri Huper, kasnije autor programskog jezika COBOL, više decenija vodećeg jezika za razvoj poslovnih aplikacija [19].

Prvi široko prihvaćeni pogramski jezici pedesetih i šezdesetih godina za velike centralne računare bili su npr. FORTRAN, COBOL, Algol i Pascal.

Programski jezici za mini i mikroračunare sedamdesetih i osamdesetih godina bili su npr. C, BASIC, Visual Basic i Turbo Pascal.

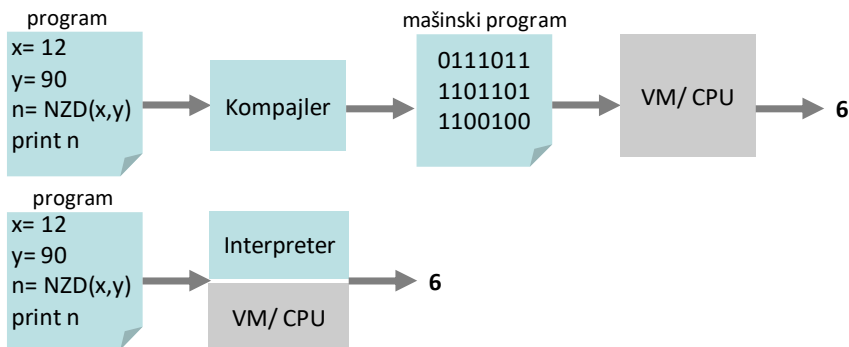
Programski jezici za moderne računare u svetskoj mreži su npr. C++, Java, C#, PHP, Javascript, R i Python.

1.2.1 Izvršavanje programa

Program se, radi izvršavanja, prevodi iz izvornog oblika u mašinski kod. Prevođenje se može obaviti pre izvršavanja programa (kompilacija) ili u toku izvršavanja, jednu po jednu naredbu (interpretacija). Na Sl. 8 prikazana su dva načina izvršavanja istog izvornog programa.

Program u izvornom obliku može se prevesti u izvršni program u mašinskom jeziku procesora, nakon čega se može izvršavati po volji veliki broj puta. Problem je samo što je program potrebno prevesti za svaki tip procesora posebno, kao i posedovati posebne verzije prevodica (*kompajlera*) za svaki od njih. Nakon bilo kakve izmene u programu, neophodno je ponovo izvršiti prevođenje svih verzija programa.

Programi u nekim programskim jezicima mogu se interpretirati, odnosno istovremeno prevoditi i izvršavati jednu po jednu naredbu programa. Ovakav inkrementalni prevodilac naziva se *interpreter*. Prevođenje programa tako ne zavisi od tipa procesora, ali je za njegovo izvršavanje na nekom procesoru potrebno razviti posebnu verziju interpretera.



Sl. 8. Primer kompilacije i interpretacije izvornog programa

Prevođenje se u oba slučaja može izvršiti u mašinski jezik ili neki jezik blizak mašinskom, tzv. *međukod* (*bytecode*, *intermediate code*, *portable code*, *p-code*). Za izvršavanje programa u međukodu koristi se poseban program, koji se obično naziva virtuelna mašina (*Virtual Machine*, VM). Na taj način se isti prevodilac može koristiti za prevođenje programa za druge procesore, odnosno

program preveden na jednom računararu može se prenositi i izvršavati na drugim računarima.

Primeri savremenih programskih jezika, koji se isključivo prevode u mašinski jezik su jezici za razvoj sistemskog softvera i alata, kao što su C i C++. Jezici koji se interpretiraju često se nazivaju *skript* jezicima. Takvi su savremeni programski jezici namenjeni razvoju Veb aplikacija, kao što su Java, C#, Javascript, PHP i Python. Zbog raznovrsne i široke upotrebe, veoma su rasprostranjene virtuelne mašine za jezike Java (*Java Virtual Machine*, JVM), C# (.NET *Common Language Runtime*, CLR) i Python.

Osnovna verzija prevodioca za jezik Python prevodi programe u međukod, koji se izvršava pomoću virtuelne mašine. Naziva se *CPython*, jer je napisana u jeziku C, a verzije ovog interpretera postoje za sve poznatije platforme. Nešto brža višenitna verzija interpretera, koja ne koristi sistemski stek, naziva se *PyPy*. Razvijene su verzije interpretera za mobilne platforme, mikrokontrolere i druge virtuelne mašine (JVM i .NET CLR), kao i prevodioci Python programa u druge programske jezike, kao npr. u jezike C, C++, Javascript i Go [19].

1.2.2 Vrste programskih jezika

Programski jezici mogu se podeliti prema različitim kriterijumima, npr. nivou apstrakcije, osnovnom pristupu razvoju aplikacija, stilu programiranja, načinu izvršavanja programa i sl. Prema nivou apstrakcije računara, programski jezici niskog nivoa su:

- *mašinski* jezici, koji se sastoje od binarno kodiranih instrukcija procesora,
- *asembleri*, koji koriste simbolički zapis mašinskih instrukcija i adresa.

Programski jezici visokog nivoa apstrakcije su:

- *proceduralni* programski jezici, koji opisuju realizaciju postupaka ili algoritama, gde spadaju C, C++, Java, C# i Python;
- *neproceduralni* jezici, koji ne opisuju postupak, već traženi rezultat, kao što su npr. jezici Prolog i SQL.

Proceduralni programski jezici mogu biti:

- *klasični*, kao što su Pascal i C,
- *funkcionalni*, kao što su LISP i F#,
- *objektno orijentisani*, kao što su C++, Java, C# i Python.

Ilustracija: Implementacija programa u različitim programskim jezicima

Na slici je prikazana implementacija najjednostavnijeg programa, koji samo ispisuje poruku "Pozdrav svima!" na ekran računara u tri različita savremena programska jezika.

C++

```
#include <iostream>
using namespace std;
int main() {
    cout << "Pozdrav svima!" << endl;
}
```

Java

```
public class PozdravSvima {
    public static void main(String[] args) {
        System.out.println("Pozdrav svima!");
    }
}
```

Python

```
print("Pozdrav svima !")
```

Program u jeziku Python znatno je kraći i jednostavniji.

1.2.3 Sintaksa i semantika programskih jezika

Programski jezici su "veštački" jezici, za razliku od "prirodnih" jezika kao što su srpski, engleski i kineski. Važni aspekti svakog jezika su njegova sintaksa i semantika [2].

Sintaksa jezika je skup znakova i prihvatljivih nizova tih znakova (sekvenci). Zapis prirodnih jezika, kao što je srpski, obuhvata sva slova azbuke, oznake interpunkcije i ispravno napisane reči jezika u okviru propisno označenih rečenica, npr. "Dobar dan, kako ste?".

Npr. rečenica kao "Dobar dan, keko ste?" nije *sintaktički* ispravna, jer reč "keko" nije reč srpskog jezika.

Semantika nekog jezika bavi se smislom sintaktički ispravnih nizova znakova. Npr. rečenica kao "Bezbojne zelene ideje besno spavaju." je sintaktički ispravna, ali je neispravna *semantički*, jer je besmislena.

Smisao reči razmatra se u okviru istog jezika. Npr. u kineskom jeziku reč "Hao" u latiničnoj transkripciji znači "dobro", dok u srpskom jeziku ta reč nema smisla, Sl. 9 [2].

	Srpski	Kineski (<i>pinyin</i>)	Kineski (<i>kinesko pismo</i>)
Sintaksa	Hao	Hao	好
Semantika	<i>besmisleno,</i> sintaktički neispravno	<i>Dobro</i>	<i>Dobro</i>

Sl. 9. Sintaksa i semantika prirodnih jezika

1.2.4 Bakusova normalna forma

Sintaksa nekog jezika može se izraziti na različite načine, npr. neformalnim narativnim opisom ili formalno, pomoću posebnog meta-jezika. Sintaksa programskih jezika najčešće se preciznije definiše skupom pravila za generisanje ili prepoznavanje ispravnih nizova znakova, npr. formalnim gramatikama ili automatima [5].

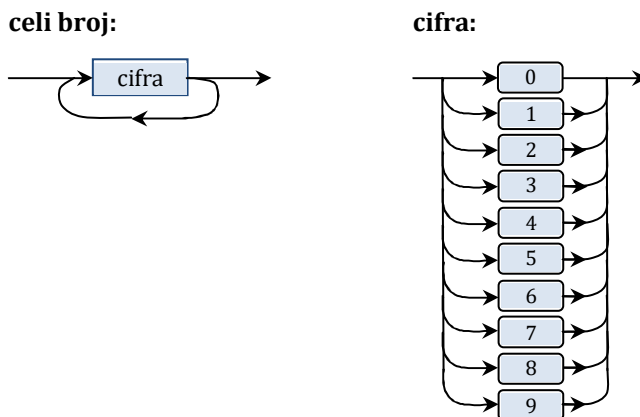
Formalna gramatika programskog jezika može se precizno opisati npr. notacijom koja se naziva Bakusova normalna forma (BNF) ili grafički, pomoću sintaksnih dijagrama [5], [19].

Na Sl. 10 opisana je gramatika formiranja takvih nizova znakova koji predstavljaju cele brojeve u naredbama fiktivnog programskog jezika (element <celi broj>) na dva načina, pomoću (a) BNF notacije i (b) sintaksnih dijagrama.

(a) BNF notacija

```
<celi broj> ::= <cifra> | <celi broj> <cifra>
<cifra> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

(b) Sintakсни dijagrami



Sl. 10. Dva načina definicije sintakse elementa programskog jezika

1.3 Proces razvoja programa

Moderno značenje termina *programiranje* je prevođenje nekog algoritma u računarski program u nekom konkretnom programskom jeziku.

Objektno-orijentisano programiranje je pristup programiranju koji rešenje nekog problema predstavlja pomoću skupa objekata i opisa njihovih svojstava i ponašanja. Objekti međusobno komuniciraju slanjem i primanjem poruka. Ovakav pristup programiranju odgovara ljudskom načinu razmišljanja i predstavlja jedan od načina smanjenja složenosti obimnih softverskih programa.

Proces rešavanja problema uz pomoć računara sastoji se iz više faza, a programiranje je samo jedna od njih. Prva faza je *analiza* problema, koja omogućava da se *dizajnira* računarsko rešenje, koje se zatim *implementira* u nekom programskom jeziku i na kraju *testira*, da se pre praktične upotrebe programa otklone eventualne greške u funkcionisanju [2].

Dizajniranje ili projektovanje računarskog rešenja sastoji se od izbora i opisa struktura podataka i algoritama kojima se rešava problem, na proceduralni ili objektno orijentisan način.

1.3.1 Razvoj i implementacija algoritama

Kako nastaje algoritam koji rešava neki praktični problem? Dizajn algoritma je složen kreativni proces, koji se tipično sastoji od [4], [6]:

1. razumevanja i definisanja problema koji treba rešiti,
2. izgradnje neke vrste modela problema,
3. izbora metoda rešavanja i
4. provere ispravnosti i preciznog opisa metoda rešavanja.

Algoritam se može implementirati u različitim programskim jezicima. Pri tome je algoritam rešenja nekog problema potrebno dokumentovati, npr. objasniti komentarima u kodu programa.

1.3.2 Testiranje programa

Univerzalni računar može da izvrši svaki algoritam.

Proces prevođenja algoritma u kod određenog programskog jezika nije jednoznačan i podložan je greškama. Greške mogu biti

- *sintaksne* greške ili greške u načinu pisanja naredbi programskog jezika, koje otkriva prevodilac i

- *semantičke* greške (logičke, greške u smislu), koje treba da otkrije programer u toku testiranja programa.

Ilustracija: Prva računarska greška

Elektromehanički računar Harvard Mark II je 1947. godine otkazao. Pregledom je otkriven uzrok: između kontakata jednog od relea ušla je sitna mušica (moljac), eng. *bug*. Na taj način je termin *bug*, koji se kao naziv za sitne tehničke greške i probleme koristi još od devetnaestog veka, povezan s istoimenim stvarnim uzrokom [19].

Mušica se danas čuva u muzeju, zajedno s dnevnичkim zapisom o događaju.



Otud popularni naziv *debugging* za proces pronalaženja i uklanjanja grešaka u računarskim programima [19].

1.4 Programski jezik Python

Jezik Python je objektno orijentisan interpretativni programski jezik opšte namene. Autor programskog jezika i prevodioca je holandski programer Gvido van Rosum² ranih devedesetih godina. Jezik je nastao u slobodno vreme, kao jednonedeljni projekt razvoja modernog jezika koji bi bio privlačan Unix/C programerima. Osnova je bio postojeći jezik ABC, zamišljen kao jezik za učenje programiranja i zamena za jezike kao što su BASIC i Pascal, koji je takođe razvijen u nacionalnom institutu CWI (Centrum Wiskunde & Informatica), gde je bio zaposlen [19].

Programski jezik je dobio naziv po poznatoj britanskoj humorističkoj TV seriji "Leteći cirkus Montija Pajtona" (Monty Python's Flying Circus). Razvojno okruženje IDLE takođe nosi ime člana grupe (Eric Idle).

Prva verzija jezika Python pojavila se 1994. godine. Trenutno se istovremeno koriste dve međusobno različite verzije jezika. Verzija 2, koja se prvi put pojavila 2000. godine, koristi se još uvek, zbog velikog obima nasleđenog softvera. Nova, nekompatibilna verzija 3 pojavila se 2008. godine i uticala na

² Guido van Rossum, sada zaposlen u poznatoj softverskoj kompaniji Dropbox

dalji paralelni razvoj prethodne verzije. U praksi se paralelno koriste najnovija izdanja verzija 2 i 3. Programi razvijeni za određenu verziju nisu kompatibilni, mada postoje alati za automatsku konverziju programa iz verzije 2 u verziju 3.

Važnija izdanja pojedinih verzija jezika Python su:

- Python 1 1994.
- Python 2 2000.
- Python 3 2008.
- Python 3.4 2014.
- Python 3.5 2015.
- Python 3.6 2016.
- Python 3.7 2018.
- Python 3.8. 2019. (aktuelna verzija)

Jezik Python je relativno jednostavan, ali se oslanja na obimne i kvalitetne programske biblioteke. Standardna programska biblioteka jezika Python (*Python Standard Library*) obuhvata tipove i strukture podataka, ugrađene funkcije i obradu izuzetaka, koji su uvek raspoloživi, kao i veliki broj međusobno zavisnih modula, koji proširuju mogućnosti jezika nakon što se uključe u program naredbom `import`.

Pošto je i sam naziv jezika šaljiv, softverski principi koji su izvorno uticali na koncipiranje i realizaciju jezika izloženi su kroz 20 aforizama [21], [22]:

Napomena: Pomalo šaljivi principi jezika Python (The Zen of Python)

- | | |
|---|---|
| • Lepo je bolje nego ružno | • Trebalo bi da postoji jedan - poželjno i samo jedan - očigledan način da se nešto uradi |
| • Eksplicitno je bolje nego implicitno | • ... mada taj način ne mora da bude vidljiv na prvi pogled, osim ako niste Holandanin |
| • Jednostavno je bolje nego složeno | • Sada je bolje nego nikada |
| • Složeno je bolje nego komplikovano | • ... mada je nikada često bolje nego upravo sada |
| • Ravno je bolje nego ugnježđeno | • Ako je implementaciju teško objasniti, ideja je loša |
| • Retko je bolje nego gusto | • Ako je implementaciju lako objasniti, mora da je ideja dobra |
| • Čitljivost je najvažnija | • Prostori imena su sjajna ideja - hajde da ih napravimo još! |
| • Posebni slučajevi nisu dovoljno posebni da krše pravila | |
| • Ipak je praktičnost važnija od čistote | |
| • Greške nikad ne treba prihvatiti ćutke | |
| • ... osim ako se eksplicitno ne učiti | |
| • Kod se pojavi dvosmislenost, ne treba pogađati | |

1.4.1 Upotreba jezika (zašto Python)

Prema godišnjoj analizi svetskog udruženja inženjera elektrotehnike i elektronike (*Institute of Electrical and Electronics Engineers, IEEE*), jezik Python je trenutno najpopularniji programski jezik [23]. Najvažniji razlozi za njegovu popularnost su:

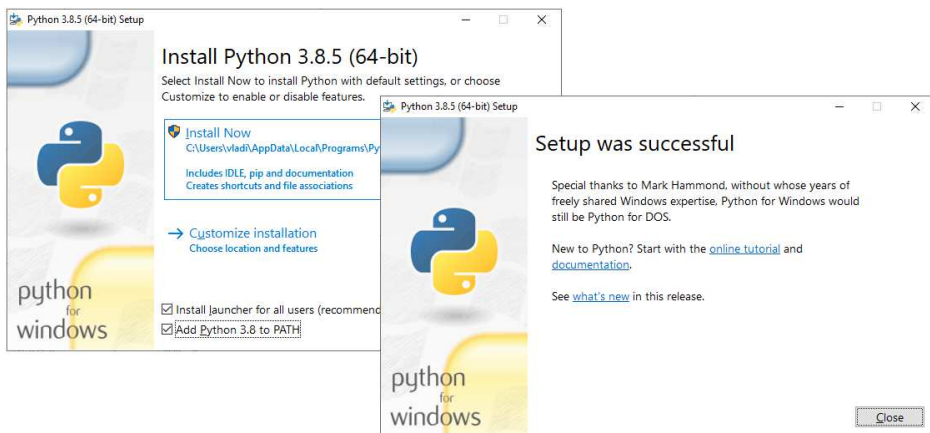
- jednostavna sintaksa i veoma čitljiv kod;
- istovremeno velike programske mogućnosti i široka upotreba;
- koriste ga velike svetske kompanije i organizacije, kao što su npr. YouTube, Google, Yahoo i NASA;
- jezik je otvorenog koda, besplatan i ima dobru korisničku podršku .

Dalji razvoj programskog jezika i održavanje odgovarajućeg softvera je pod nadzorom neprofitne organizacije Python Software Foundation, koja je nosilac svih prava intelektualne svojine. Među sponzorima ove fondacije su npr. Google, Intel, Microsoft, O'Reilly Media, Inc. i Red Hat.

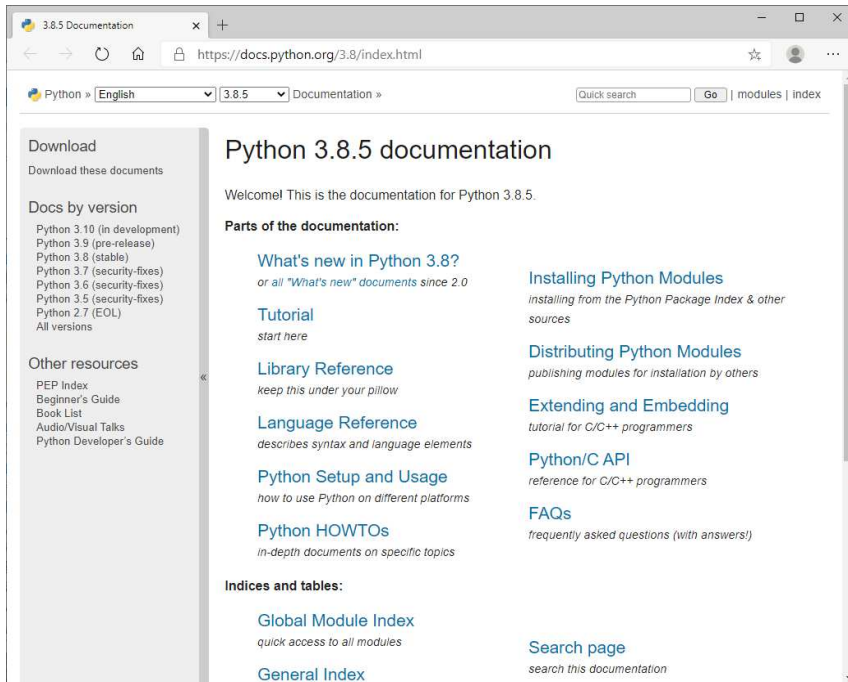
Softver otvorenog koda je besplatan, a može se preuzeti sa zvaničnog sajta www.python.org.

1.4.2 Instalacija

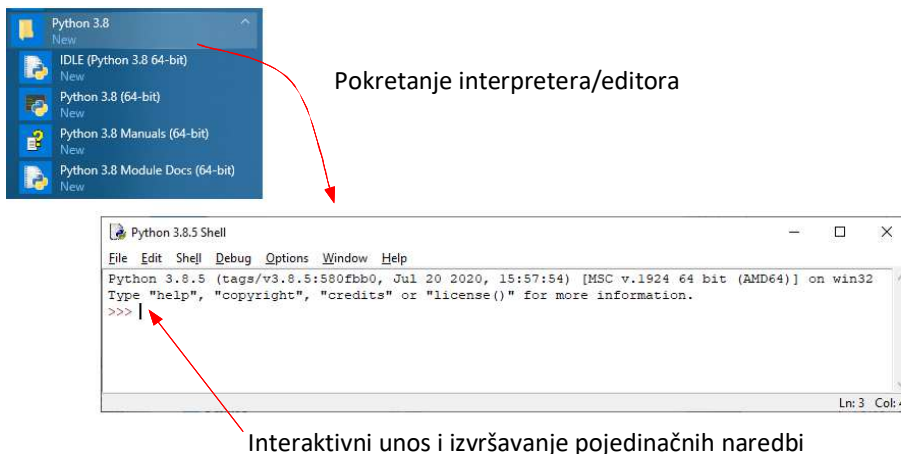
U ovom materijalu koristiće se verzije jezika Python koje nose oznaku 3.5-3.7 i 3.8.5, najnovija u vreme pisanja udžbenika. Zavisno od vrste računara i operativnog sistema, na sajtu se preuzima odgovarajuća verzija instalacionog programa i pokreće instalacija. Na slici je tipičan izgled početnog ekrana i završetka instalacije s podrazumevajućim instalacionim parametrima.



Online dokumentacija može se pronaći na zvaničnom sajtu www.python.org, za aktuelnu i prethodne verzije jezika.



U instalaciju jezika uključen je i interaktivni editor/interpreter IDLE, koji se u ovom materijalu koristi kao osnovni alat za razvoj programa u jeziku Python:



Interaktivni način rada omogućava istovremeni unos i izvršavanje naredbi. Editor omogućava unos i ispravke programa, koji će se izvršavati kao celina.

Napomena: Postoji besplatna podrška za razvoj programa u jeziku Python u razvojnom okruženju Microsoft Visual Studio putem dodatka *Python Tools for Visual Studio* <https://www.visualstudio.com/vs/python/>.

1.4.3 Osnovni elementi jezika Python

Program u jeziku Python je niz naredbi, čiji delimiter je kraj linije teksta. Posebnu ulogu u pisanju programa ima uvlačenje (indentacija), koje takođe služi kao delimiter u složenim višelinijskim naredbama.

U naredbama se razlikuju mala i velika slova (*case sensitive*), tako da se npr. naredbe `print` i `Print` međusobno razlikuju.

Sintaksa jezika Python u ovom poglavlju će se uvoditi postepeno i neformalno, kroz kratki opis osnovnih verzija naredbi i tipične primere njihove upotrebe. U ovom uvodnom pregledu osnovnih osobina jezika Python prikazaće se:

- Komentar,
- Promenljive i izrazi,
- Osnovne strukture podataka,
- Upravljanje tokom programa (grananje i ponavljanje),
- Interaktivni ulaz-izlaz.

Komentar

Komentari su proizvoljan tekst koji prevodilac zanemaruje, ali predstavljaju važan deo programskog koda. Sadrže informacije namenjene osobama koje čitaju program, tako da su istovremeno i neophodna minimalna dokumentacija softvera. U jeziku Python postoje dve vrste komentara: *komentari u jednoj liniji* i *višelinijski komentari*.

Komentari dužine jedne linije počinju znakom "#", a završavaju oznakom kraja linije programa, npr.

```
>>> # Komentar u jednoj liniji
>>>
```

Višelinijski komentari omogućavaju unos proizvoljnog teksta između dve programske linije koje počinju sa tri uzastopna znaka navoda, npr.

```
>>> """
    Komentar
    u više linija
    """
>>>
```

Promenljive i izrazi

Promenljive u jeziku Python se ne deklariraju i nemaju unapred definisan tip, već je tip svake promenljive određen vrednošću koja joj se dodeli, npr. (novi red je nova naredba), npr.

>>> x = 2	x	2
>>> ime = "Marko"	ime	'Marko'

Tip vrednosti promenljive može se ispitati pomoću ugrađene funkcije `type()`.

Vrednost promenljive se može izračunati pomoću nekog izraza, npr.

>>> x = 2*x	x	4
>>> pozdrav = "Zdravo " + ime	pozdrav	'Zdravo Marko'

Osnovni tipovi i strukture podataka

Tip podatka čini skup vrednosti i skup operatora koji se mogu primeniti na te vrednosti. Osnovni numerički tipovi su celi i decimalni brojevi, kao i logičke vrednosti, npr.

int	2
float	2.0
bool	1

Tip `bool` predstavlja podtip tipa `int`, tako da logičke vrednosti istinitosti imaju celobrojni interpretaciju: `True == 1` i `False == 0`.

Osnovne strukture podataka u jeziku Python su *stringovi*, *liste*, *n-torke* i *rečnici*. Stringovi su nizovi znakova, čija se pozicija u nizu računa od nule, npr.

>>> s = "Marko"	s	'Marko'
0 1 2 3 4 -5 -4 -3 -2 -1		
>>> s = str[0:3]	s	'Mar'

Brojanje elemenata stringa vrši se tako da prvi element ima poziciju ili indeks 0, a svaki naredni za jedan veći. Kada se zadaje više elemenata u rasponu *od:do*, poslednji element nije uključen. Elementi se mogu brojati i unazad, počev od poslednjeg elementa stringa, koji ima indeks -1.

Liste su sekvence ili nizovi podataka, koji mogu biti različitog tipa i kojima se može pristupiti pomoću indeksa, tj. podatka o njihovom redosledu u nizu, npr.

```
>>> x = [1, 2, 3]
>>> y = ['jedan', 'dva', 'tri']
>>> print(x[1], y[1])    # elementi se broje od 0
2 dva
```

Ugnježdene liste predstavljaju liste čiji su elementi takođe strukture tipa liste, pa se u memoriji računara pomoću njih mogu predstavljati složeniji odnosi između podataka, kao npr. hijerarhije. Prethodno definisane liste `x` i `l` mogu se uključiti u strukturu ugnježdene liste, koja tako može da predstavlja hijerarhiju sa dve grane na prvom i po tri grane (elementa) na drugom nivou, npr.

```
>>> ul = [x, y]    ul [[1, 2, 3], ['jedan', 'dva', 'tri']]
```

Struktura *n-torke* takođe je namenjena za predstavljanje nizova ili sekvenci objekata različitog tipa, ali se, za razliku od liste, ne može menjati. Zbog toga se za predstavljanje *n-torki* koriste druge oznake. Npr. prethodna hijerarhijska struktura može se predstaviti i pomoću nepromenljive strukture *n-torke*:

```
>>> sn = ((1, 2, 3), ('jedan', 'dva', 'tri'))
>>> sn
((1, 2, 3), ('jedan', 'dva', 'tri'))
```

Rečnik je struktura kod koje se elementima, koji takođe mogu biti bilo kog tipa, ne pristupa prema numeričkom indeksu, već pomoću nenumeričke informacije, koja se naziva *ključ*. Tako se spisak studenata može zadati u formi rečnika kao niz parova vrednosti `<broj indeksa>:<podaci>`, npr.

```
>>> stud = {"210": "Jovan", "220": "Marija", "245": "Ivan"}
```

Pristup podacima u rečniku vrši se pomoću ključa, npr.

```
>>> stud["220"]
'Marija'
```

Upravljanje tokom programa (grananje i ponavljanje)

Osnovne naredbe za upravljanje tokom izvršavanja programa u jeziku Python su naredbe grananja (selekcije) i naredbe ponavljanja (iteracije).

Naredba selekcije služi za izbor delova programa koji se izvršavaju samo pod određeni uslovima, npr.

```
if lepo_vreme:
    print('Nije potreban kišobran.')
```

Naredbe koje se uslovno izvršavaju uvučene su za isti broj mesta u odnosu na na početak naredbe `if`. Standardno se koristi uvlačenje (indentacija) za 4 mesta.

Najjednostavnija naredba ponavljanja je naredba `for`, koja izvršava skup naredbi zadani broj puta, po jednom za svaku vrednost iz zadanog skupa vrednosti, npr.

```
for element in <skup_vrednosti>:
    print(element)
```

Naredbe koje se ponavljanju za svaki element iz skupa vrednosti takođe se označavaju indentacijom, odnosno uvlače za isti broj mesta.

Skup vrednosti može se zadati nabrojanjem, u obliku liste, npr. `[1, 3, 7]` ili generisati pomoću neke funkcije kao što je `range(1, 101)`, koja proizvodi niz celih brojeva u rasponu od 1 do 100.

Interaktivni ulaz-izlaz

Osnovne naredbe za interaktivni ulaz i izlaz podataka su funkcija `input()`, koja služi za čitanje podataka unesenih putem tastature i funkcija `print()`, koja vrši ispis podataka na ekran računara.

Program pomoću funkcije `input()` postavlja pitanje korisniku, koje se prikazuje na ekranu računara, a kursor se postavlja na sledeće slobodno polje radi unosa i čitanja odgovora. Kada se putem tastature unese odgovor i potvrdi tasterom *Enter*, uneseni tekst se vraća kao rezultat izvršavanja funkcije i može se zapamtiti u nekoj promenljivoj, npr.

```
>>> ime = input("Kako se zovete ? ")
Kako se zovete ? Jovana
```

Uneseni podaci su zapamćeni u promenljivoj `ime` kao tekst, odnosno vrednost tipa string:

```
ime 'Jovana'
```

Standardna biblioteka

Standardna biblioteka jezika Python sastoji se od velikog broja ugrađenih modula realizovanih u jezicima C i Python, koji sadrže npr. matematičke funkcije, funkcije za rad s fajlovima, pristup operativnom sistemu, itd. <https://docs.python.org/3/library/>

Postoji veoma veliki broj dodatnih komponenti, programa, modula, paketa i aplikacija za jezik Python, koji se stalno dopunjava <https://pypi.python.org/pypi>

Verzije jezika Python

U široj upotrebi su dve verzije jezika:

- Verzija 2, aktuelna do 2008. godine. Zbog nasleđenih aplikacija i biblioteka programa razvija se i dalje. Najnovija verzija je 2.7.18 od 2020.
- Verzija 3 je aktuelna od 2008. godine. Najnovija verzija je 3.8.5 od 2020.

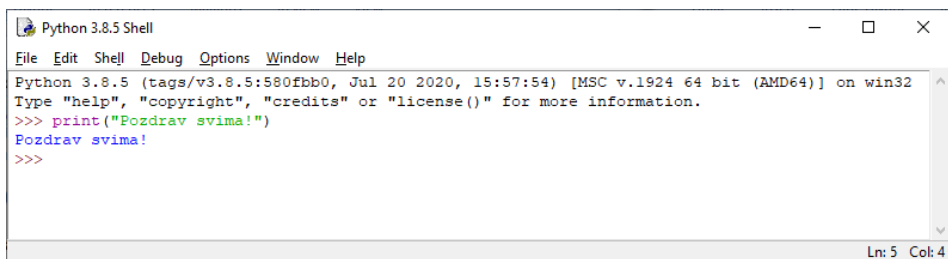
Postoji programski alat koji automatizovano prevodi programe napisane u sintaksi verzije 2 u sintaksu koju koristi verzija 3 jezika Python.

1.5 Primeri malih programa

Na kraju poglavlja daju se dva primera veoma jednostavnih programa koji ilustruju primenu jezika Python: program Pozdrav, koji samo ispisuje poruku i program Putovanje, koji računa vrednost jednostavnog izraza.

1.5.1 Pozdrav

U jeziku Python poruka 'Pozdrav svima' ispisuje se na konzolni ekran naredbom `print()`. U interaktivnom načinu rada, naredba unesena putem programskog editora IDLE (iza simbola `>>>`) interpretira se i izvrši čim se njen unos potvrdi tasterom *Enter*. Rezultat se ispisuje u sledećem redu, od prve kolone:



```
Python 3.8.5 Shell
File Edit Shell Debug Options Window Help
Python 3.8.5 (tags/v3.8.5:580fbb0, Jul 20 2020, 15:57:54) [MSC v.1924 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> print("Pozdrav svima!")
Pozdrav svima!
>>>
```

1.5.2 Putovanje

Program se koristi za računanje vremena trajanja putovanja t na osnovu prosečne brzine kretanja vozila v i udaljenosti d između polazne lokacije i odredišta $t = d/v$. Vrednost promenljivih d i v korisnik unosi interaktivno, na zahtev programa.

Algoritam na kome se program zasniva može se izložiti narativno:

1. učitaj brzinu v (u km/h) i udaljenost d (u km),
2. izračunaj vreme trajanja putovanja $t = d/v$,
3. prikaži rezultat t .

Realizacija algoritma u jeziku Python može biti sedeći program:

```
# Računanje vremena putovanja za zadanu brzinu i udaljenost
# Petar Petrović, 02.10.2017

brzina = input("Unesite brzinu (km/h): ")
brzina = int(brzina)

udaljenost = input("Unesite udaljenost (km): ")
udaljenost = float(udaljenost)

vreme = udaljenost/brzina

print("Uz brzinu", brzina, "km/h, potrebno je")
print(vreme,"sati putovanja da se pređe", udaljenost, "km.")
input("\nPritisni taster Enter za kraj")
```

Vidi se da je u realizaciji programa u jeziku Python bila neophodna konverzija tipova unesenih podataka, jer se uneseni odgovori učitavaju kao nizovi znakova, koje je pre računanja vremena potrebno pretvoriti u binarne brojeve.

Ispis rezultata na ekran vrši se pozivom funkcije `print()`, koja ispisuje niz vrednosti navedene unutar zagrada, odvojene zarezom. Funkcija automatski pretvara binarni zapis brojeva u nizove znakova koje ispisuje na ekran. U tekst je umetnut specijalni znak za prelazak u novi red, označen kao `'\n'`.

Programi u jeziku Python čuvaju se na fajlovima koji u svom nazivu imaju sufiks `".py"`, npr. `Putovanje.py`. Izvršavaju se kao jedna celina. Primer rezultata izvršavanja prethodnog programa je:

```
Unesite brzinu (km/h): 120
Unesite udaljenost (km): 210
Uz brzinu 120 km/h, potrebno je
1.75 sati putovanja da se pređe 210.0 km.

Pritisni taster Enter za kraj
>>>
```

Program završava s radom (terminira) kad se pritisne taster *Enter*.

1.6 Poznate aplikacije

Brojne poznate aplikacije koriste ili su u celini napisane u jeziku Python. Primeri nekih poznatijih aplikacija su, npr. [19], [24], [25]:

- *Blender3D* - softver za 3D animacije, igre i video montažu, koji koristi Python kao programski jezik;
- *Dropbox* - poznati Veb sajt za hosting fajlova;

- *Firefox* - jedan od najpoznatijih Veb čitača (*browser*) implementiran je u više programskih jezika, uključujući Python;
- *Google App Engine* - mnoge komponente najpoznatijeg pretraživača i platforme za razvoj Veb aplikacija napisani su u jeziku Python;
- *Instagram* - poznata društvena mreža za deljenje fotografija;
- *Pinterest* - društvena mreža za deljenje fotografija, videa i drugih sadržaja;
- *Reddit* - sajt za razmenu informacija i zabavu, originalno razvijen u jeziku CommonLisp, a zatim ponovo napisan u jeziku Python;
- *Quora* - poznati sajt i mreža za postavljanje pitanja i dobijanje kvalifikovanih odgovora;
- *Yahoo Maps* - određeni servisi softvera za rad s mapama implementirani su u jeziku Python;
- *Youtube* - najpoznatiji sajt za deljenje video materijala prikazuje video posredstvom programa realizovanih u jeziku Python.

Pitanja za ponavljanje

1. Kada je nastao prvi programabilni računar i ko je njegov autor? Ko se smatra prvim programerom?
2. Objasnite pojam univerzalne Turingove mašine.
3. Kako se naziva arhitektura većine savremenih računara opšte namene?
4. Šta je algoritam i kako je nastao taj naziv?
5. Kako se nazivaju programi koji istovremeno prevode i izvršavaju računarske programe?
6. Navedite podelu programskih jezika visokog nivoa apstrakcije.
7. Objasnite razliku između sintakse i semantike programskog jezika.
8. Koje se vrste programskih grešaka otkrivaju testiranjem?
9. Objasnite nastanak i poreklo naziva jezika Python.
10. Čemu služe i kako se pišu komentari u jeziku Python?
11. Kako se određuje tip promenljive u jeziku Python?
12. Navedite osnovne tipove i strukture podataka u jeziku Python.

2 Osnovni elementi programa

1. Uvod
2. Promenljive i dodela vrednosti
3. Izrazi
4. Grananje (selekcija)
5. Ponavljanje (iteracija)
6. Ugrađene funkcije
7. Korisničke funkcije
8. Ulaz-izlaz
9. Primeri programa

U ovom poglavlju daje se pregled i ukratko izlažu osnovni elementi programskog jezika Python neophodni za klasično proceduralno programiranje.

2.1 Uvod

Formalna definicija sintakse naredbi jezika Python u obliku gramatike u uprošćenoj BNF notaciji, za aktuelnu verziju programskog jezika, može se naći na zvaničnom sajtu [24]. U ovom poglavlju daje se neformalni pregled najvažnijih naredbi jezika u obliku u kome se najčešće koriste.

Proceduralni ili imperativni programi sastoje se od nizova naredbi. Programske naredbe koje se koriste za izradu ovakvih programa mogu biti:

- različite *deklaracije*, npr. definicije konstanti ili tipova i strukture promenljivih,
- naredbe za *dodelu vrednosti* promenljivoj, pri čemu vrednost može biti izračunata na osnovu nekog izraza (formule),
- naredbe *grananja* (selekcije), kojima se menja redosled izvršavanja niza naredbi,
- naredbe *ponavljanja* (iteracije), kojima se izabrani niz naredbi ponavlja više puta,
- *definicije* funkcija ili procedura (potprograma),
- naredbe za *ulaz-izlaz* podataka, za interakciju s korisnikom računara (tastatura, ekran) ili rad s fajlovima.

Program u jeziku Python je niz naredbi, koje mogu da budu u jednoj ili više logičkih linija programa. Logička linija se može sastojati od jedne ili više fizičkih linija, koje završavaju oznakom kraja linije teksta. Oznaka kraja linije u u sistemu *Unix* je samo znak LF³, dok je u sistemu *Windows* kombinacija dva znaka, CR (*Carriage Return*) i LF (*Line Feed*), npr.

³ Nazivi i oznake specijalnih znakova definisane su u ASCII tabeli kodova

```
x = 1
if x > 0:
    print("Tri fizičke i logičke linije")
```

Za podjelu jedne logičke linije programa na više fizičkih linija, često radi skraćivanja dužine jednog reda programskog koda na ekranu, koristi se znak "\", koji označava nastavak logičke linije, npr.

```
godina = 2017
prestup = (godina % 4 == 0 and godina % 100 != 0) \
or (godina % 400 == 0)
```

Sintaksa jezika dozvoljava i pisanje više kratkih naredbi u jednoj fizičkoj liniji, odvojenih znakom ";". Radi bolje čitljivosti, dužina naredbi programa ne treba da bude prevelika. Preporučuje se da ne bude duža od 79 znakova, što je pogodno za korisnike koji imaju ekrane manjih dimenzija.

Posebnu ulogu ima uvlačenje teksta programa ili indentacija, koja služi za definisanje blokova naredbi u funkcijama, klasama i naredbama grananja i ponavljanja. Dubina uvlačenja nije definisana, ali mora biti ista za sve naredbe jednog bloka. Uvlačenje se postiže unosom niza praznih mesta ili pomoću tabulatora. Uobičajeno je da se uvlačenje vrši s 4 prazna mesta, bez upotrebe tabulatora.

U naredbama se razlikuju mala i velika slova (*case sensitive*), tako da se npr. naredbe `print` i `Print` međusobno razlikuju.

Primer: Program za računanje površine kruga u jeziku Python

Algoritam praktičnog računanja površine kruga je:

1. učitaj poluprečnik kruga r
2. izračunaj površinu po formuli

$$P = r^2 \cdot \pi$$
3. prikaži rezultat

Pitanja koja treba rešiti za implementaciju prikazanog algoritma su, npr.

- Kako se učitava podatak? (naredba `input()`)
- Kako se pamti učitana vrednost? (dodela vrednosti `=`)
- Kako se računa površina kruga prema formuli? (*aritmetički izraz*)
- Kako se prikazuje rezultat? (naredba `print()`)

Elementi programa za računanje površine kruga u jeziku Python su:

```
# Učitavanje vrednosti poluprečnika
poluprecnik = float(input("Unesi poluprečnik: "))
# Računanje površine kruga
povrsina = poluprecnik * poluprecnik * 3.14159
# Prikaz rezultata
print("Površina kruga R=", poluprecnik, "je", povrsina)
```

izraz (dve ugnježdene funkcije)

aritmetički izraz

naredba za ulaz-izlaz (poziv ugrađene funkcije)

Pokretanjem izvršavanja programa dobija se rezultat:

```
Unesi poluprečnik: 25
Površina kruga R= 25.0 je 1963.4937499999999
>>>
```

Napomena: naredba (poziv ugrađene funkcije) *input* čita uneseni tekst i vraća vrednost tipa *string*. Ugrađena funkcija *float* vrši konverziju ove vrednosti u numerički tip podatka, neophodan za izračunavanje površine.

U sledećoj tabeli je kratki pregled najpoznatijih savremenih proceduralnih ili imperativnih programskih jezika. Neki od njih su klasični programski jezici, kao npr. C i Pascal, ali većina ima elemente neophodne za izgradnju objektno orijentisanih programa.

Programski jezik	Kratki opis
Ada	programski jezik za izradu pouzdanih programa za posebne namene, nazvan po Adi Bajron, prvom programeru
C	jezik visokog nivoa koji omogućava izradu sistemskih programa
C++	objektno-orijentisan programski jezik zasnovan na jeziku C
C#	programski jezik kompanije Microsoft, hibrid jezika Java i C++
Java	programski jezik namenjen izradi prenosivih aplikacija, nezavisnih od računara i operativnog sistema (Interent)
Pascal	jednostavni strukturirani programski jezik opšte namene, koji se danas uglavnom koristi za učenje programiranja
Python	jednostavni programski jezik opšte namene, pogodan za izradu svih tipova aplikacija
Visual Basic	programski jezik kompanije Microsoft, za brzu izradu Windows aplikacija

2.2 Promenljive i dodela vrednosti

Promenljiva (variable) je naziv za vrednost zapamćenu u memoriji računara, koja se može menjati.

Nazivi promenljivih u jeziku Python mogu se formirati od malih i velikih slova, znaka "_" i brojeva, ali prvi znak naziva mora biti slovo ili znak "_". Neformalna stilska konvencija je da se za imena promenljivih koriste mala slova. Iako dužina naziva nije ograničena, radi dobre čitljivosti treba da bude u skladu s preporukom o najvećoj dužini linije programa.

Imena se takođe moraju razlikovati od rezervisanih reči jezika Python, koje su za verziju 3:

and, as, assert, break, class, continue, def, del, elif, else, except, False, finally, for, from, global, if, import, in, is, lambda, None, nonlocal, not, or, pass, raise, return, True, try, while, with, yield

Imenovana konstanta je naziv za vrednost zapamćenu u memoriji računara, koja se načelno ne može menjati. Jezik Python nema posebne oznake za konstante, već se one kreiraju kao promenljive čiju vrednost program dobrovoljno neće menjati, što se samo stilski dokumentuje velikim slovima naziva promenljive, npr.

```
PI = 3.14159
E = 2.71828
BRZINA_SVETLOSTI = 299792458
GOOGOL = 10**100
```

2.2.1 Naredba dodele vrednosti

Naredba *dodele vrednosti* u jeziku Python ima složenu sintaksu, ali je najčešći oblik u kome se koristi:

```
<promenljiva> = <izraz>
```

Tip promenljive definisan je tipom dodeljene vrednosti, npr.

```
x = 1
x = 1.0
x = 2*x
ime = "Jelena"
pozdrav = "Zdravo " + ime
```

Jezik Python dozvoljava višestruke dodele vrednosti u jednoj naredbi, npr.

```
a, b, c = 1, 2, 3
a = b = c = 0
```

Nakon prve naredbe, vrednosti promenljivih su $a=1$, $b=2$ i $c=3$. Nakon druge naredbe dodele vrednosti, sve promenljive imaju vrednost 0.

Tip vrednosti promenljive može se ispitati funkcijom `type()`, npr. nakon prethodnih dodela vrednosti, `type(x)` daje rezultat `<class 'float'>`, a `type(ime)` daje `<class 'str'>`.

Celi broj tipa `int` u jeziku Python implementiran je tako da može imati proizvoljan broj cifara, npr.

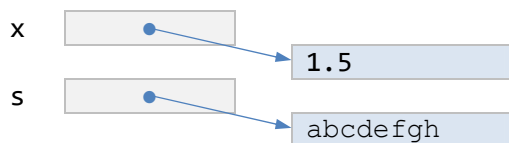
```
>>> a = 2345678901234567890123456789012345678901234567890
>>> a
12345678901234567890123456789012345678901234567890
>>> type(a)
<class 'int'>
>>>
```

Promenljive u jeziku Python, za razliku od strogo tipizovanih programskih jezika, ne predstavljaju naziv za deo memorije u kome se čuvaju podaci, već adresu ili pokazivač na takav memorijski prostor s podacima. Dok su u strogo tipizovanim jezicima nazivi promenljivih adrese na kojima se nalaze *vrednosti* promenljivih, u jeziku Python su to adrese na kojima se nalaze *adrese vrednosti* promenljivih, a same vrednosti su na različitim memorijskim lokacijama.

Npr. dodela vrednosti promenljivima $x=1.5$ i $s="abcdefgh"$ u strogo tipizovanim jezicima proizvodi sledeće strukture u memoriji

x	1.5
s	abcdefgh

dok je njihov rezultat u jeziku Python drugačiji:



Ova osobina jezika Python omogućava da se tip vrednosti promenljive ne mora u programu unapred definisati, kao i da se bilo kad može promeniti pomoću naredbe za dodelu vrednosti.

2.2.2 Oblast definisanosti promenljivih

Oblast definisanosti neke promenljive (*scope*) predstavlja deo programa u kome je vrednost promenljive definisana i može se koristiti.

Npr. ako je na početku programa naredba

```
brojac = broj + 1
```

a vrednost promenljive `brojac` do tada još nije definisana, sistem dojavljuje grešku izvršavanja

```
NameError: name 'brojac' is not defined
```

2.2.3 Kompleksni brojevi

Jezik Python omogućava upotrebu i drugih tipova vrednosti, kao što su npr. kompleksni brojevi [1]. Kompleksni broj ima oblik polinoma $x+iy$, gde je x realni, y imaginarni deo, a $i = \sqrt{-1}$ je imaginarna jedinica.

Kompleksna vrednost u jeziku Python može se kreirati pomoću izraza ili funkcijom `complex(x,y)` iz biblioteke `cmath`. Vrednost realnog i imaginarnog dela kompleksnog broja su svojstva `.real` i `.imag`. Osim osnovnih računskih operacija sabiranja, oduzimanja, množenja, deljenja i apsolutne vrednosti, za rad s kompleksnim brojevima mogu se koristiti i druge funkcije, npr. `conjugate()`, `polar()`, `rect()`, `exp()`, `log()`, itd.

Primer kreiranja i upotrebe kompleksnih vrednosti:

```
import cmath

c = 3 + 5j          # kreiranje kompleksnog broja 3+5j
x = 3
y = 5
z = complex(x,y)   # kreiranje kompleksnog broja x+iy

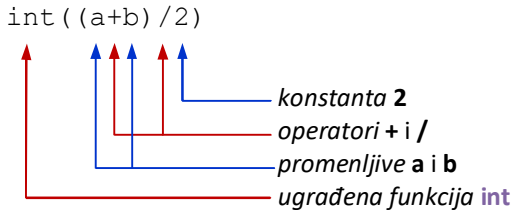
# Prikaz kompleksnog broja i njegovih komponenti
print("Kompleksni broj c = ", c)
print("Kompleksni broj z = ", z)
print(" Realni deo z =", z.real)
print(" Imaginarni deo z = ", z.imag)
```

Izvršavanje prethodnog segmenta koda daje:

```
Kompleksni broj c = (3+5j)
Kompleksni broj z = (3+5j)
Realni deo z = 3.0
Imaginarni deo z = 5.0
```

2.3 Izrazi

Izrazi su pravilna kombinacija vrednosti (konstanti), promenljivih, funkcija i operatora, npr.



Nakon izračunavanja ili evaluacije izraza dobija se rezultat određenog tipa, koji se može dodeliti nekoj promenljivoj. Rezultat može biti numerički, nenumerički ili logički (koji je u stvari podtip tipa `int`). U gornjem primeru, za `a=1` i `b=2`, rezultat evaluacije je `1`, vrednost tipa `int`.

2.3.1 Evaluacija izraza

Redosled operacija prilikom izračunavanja nekog izraza zavisi od tipova upotrebljenih vrednosti i operatora.

Za aritmetičke izraze, redosled se određuje prema redosledu operacija u matematici, pre svega prioritetu operatora i zagradama, npr. za izraz

`3 + 4 * 5 ** 2` rezultat evaluacije je `103`

što se razlikuje od izraza

`(3 + 4) * 5 ** 2` rezultat evaluacije je `175`

Kod logičkih izraza, operacija `and` ima prednost nad `or`, npr.

`True or False and False` daje vrednost `True`
`False or True and False` daje vrednost `False`

Svi operatori istog prioriteta evaluiraju se s leva u desno. Zgrade menjaju redosled evaluacije svih tipova operatora.

2.3.2 Operatori

Operatori u jeziku Python definisani su za određeni tip vrednosti na koje se mogu primeniti. Razlikuju se aritmetički, operatori poređenja, specijalni operatori (koji se koriste uz operator dodele vrednosti), logički operatori, operatori za rad s bitovima, operatori pripadnosti i operatori identifikacije. U odnosu na broj argumenata operatori mogu biti *unarni* (jedan argument) i *binarni* (dva argumenta).

U ovom poglavlju će se kratko nabrojati aritmetički i relacioni operatori. *Aritmetički* operatori su

- +** - sabiranje numeričkih vrednosti i unarni operator predznaka,
- - oduzimanje numeričkih vrednosti i unarni operator predznaka,
- *** - množenje numeričkih vrednosti,
- /** - deljenje numeričkih vrednosti koje proizvodi decimalni rezultat,
- //** - celobrojno deljenje kojim se decimalni deo rezultata odbacuje,
- **** - stepenovanje ili eksponenciranje,
- %** - ostatak od deljenja ili modul.

Binarni operator modul kao rezultat daje ostatak od deljenja vrednosti navedene ispred operatora vrednošću navedenom iza njega. Treba napomenuti da je ostatak od deljenja parnog broja brojem 2 jednak je nuli. Ovaj operator je primer jednostavne jednosmerne funkcije, za koju ne postoji inverzna funkcija. Takve funkcije su pogodne za šifrovanje podataka, jer se bez podatka o deliocu, ne može znati koji broj je bio deljenik.

Relacioni ili operatori poređenja u izrazima su:

- <** - manje od
- <=** - manje ili jednako
- >** - veće od
- >=** - veće ili jednako
- ==** - jednako
- !=** - različito (nije jednako)

Više o operatorima i njihovoj upotrebi u narednom poglavlju, posvećenom izrazima.

2.4 Grananje (selekcija)

Grananje ili selekcija je naredba za izbor (selekciju) grupe naredbi koje će se izvršiti ako je zadovoljen određeni uslov. Sintaksa naredbe selekcije u BNF je:

```
<if_naredba> ::= if <uslov> : <blok naredbi>
                (elif <uslov> : <blok naredbi>) *
                [else : <blok naredbi>]
```

Naredba selekcije može da ima jednostavnu ili složenu formu, za izbor jednog, dva ili više blokova naredbi, npr.

Jednostavna selekcija (1 blok naredbi)	Složena selekcija (2 bloka naredbi)	Složena selekcija (3 ili više blokova)
<pre>if uslov: blok naredbi</pre>	<pre>if uslov: blok naredbi 1 else: blok naredbi 2</pre>	<pre>if uslov1: blok naredbi 1 elif uslov2: blok naredbi 2 ... else: blok naredbi n</pre>

Uslov je izraz koji se evaluira u vrednost bilo kog tipa. Rezultat evaluacije se tretira kao istina (`True`) za sve vrednosti osim za numeričku vrednost nula, prazan string, praznu listu i sl.

U jednostavnoj formi naredbe selekcije, ako je logički uslov istinit, blok naredbi se izvršava, a ako *nije* istinit (`False`), naredba selekcije okončava i izvršavanje programa nastavlja od sledeće naredbe programa.

Kod složene selekcije sa dva bloka naredbi, ako je logički uslov istinit, izvršava se prvi blok naredbi, a ako logički uslov *nije* istinit, izvršava se drugi blok naredbi. Kod složene selekcije sa tri ili više blokova naredbi, na isti način se redom ispituje više uslova i izvršava prvi blok naredbi za koji je neki uslov istinit. Ako nijedan uslov nije istinit, izvršava se blok naredbi iza poslednjeg uslova (`else`).

Primer jednostavnog grananja, s jednostavnim uslovom je:

```
lepo_vreme = False
if lep_vreme:
    print('Nije vam potreban kišobran.')
```

Početak bloka naredbi označen je znakom ":" iza uslova, a sve naredbe bloka su označene uvlačenjem (indentacijom), u ovom slučaju za 4 mesta.

Složenija forma grananja, koja omogućava izbor jedne od tri različite mogućnosti, oslanja se na uslove koji predstavljaju relacione izraze:

```
if starost < 18:
    print('popust za decu i učenike')
elif starost > 65:
    print('nije potrebna karta')
else:
    print('puna cena karte')
```

U navedenom primeru vrši se izbor jedne od tri varijante određivanja cene karte u javnom prevozu na osnovu starosti putnika.

2.5 Ponavljanje (iteracija)

U jeziku Python postoje dve osnovne forme naredbe za ponavljanje izvršavanja bloka naredbi:

- **for** je naredba za ponavljanje bloka naredbi unapred poznati broj puta,
- **while** je naredba za ponavljanje bloka naredbi sve dok važi neki logički uslov, gde broj ponavljanja nije unapred poznat.

Osnovna sintaksa naredbe ponavljanja poznati broj puta je:

```
for <promenljiva> in <sekvenca>:
    <blok naredbi>
```

Sintaksa najčešće korišćenog oblika naredbe ponavljanja prema logičkom uslovu je:

```
while <uslov>:
    <blok naredbi>
```

Uslov ove naredbe ponavljanja je izraz koji mora biti istinit da bi se ponavljanje izvršilo bar jednom. Okončanje ponavljanja obezbeđuje se odgovarajućom promenom uslova u bloku naredbi.

Primer ponavljanja zadani broj puta je sledeća petlja:

```
for element in [1,3,7]:
    print("Indeks=", element )
```

Ponavljjanje se vrši tri puta, za svaki element eksplicitno zadanog skupa vrednosti u obliku liste konstanti od 3 elementa [1, 3, 7].

Primer ponavljanja prema zadanom logičkom uslovu je:

```
broj = 16
while broj > 0:
    print("Broj=", i)
    broj = broj - 4
```

Uslov broj>0 je istinit pre početka izvršavanja petlje, pa se blok naredbi istaknut indentacijom izvršava u prvom ponavljanju. U samom bloku se logički uslov modifikuje promenom vrednosti promenljive broj. Sistematskim smanjivanjem vrednosti promenljive do vrednosti 0 obezbeđuje se da uslov petlje prestane da važi, odnosno okončanje ponavljanja.

2.6 Ugrađene funkcije

Jezik Python ima veliki broj ugrađenih funkcija za različite primene, koje su uvek na raspolaganju:

<code>abs()</code>	<code>filter()</code>	<code>locals()</code>	<code>setattr()</code>
<code>all()</code>	<code>float()</code>	<code>map()</code>	<code>slice()</code>
<code>bin()</code>	<code>format()</code>	<code>max()</code>	<code>any()</code>
<code>bool()</code>	<code>frozenset()</code>	<code>memoryview()</code>	<code>sorted()</code>
<code>bytearray()</code>	<code>getattr()</code>	<code>min()</code>	<code>ascii()</code>
<code>bytes()</code>	<code>globals()</code>	<code>next()</code>	<code>staticmethod</code>
<code>callable()</code>	<code>hasattr()</code>	<code>object()</code>	<code>()</code>
<code>chr()</code>	<code>hash()</code>	<code>oct()</code>	<code>str()</code>
<code>classmethod()</code>	<code>help()</code>	<code>open()</code>	<code>sum()</code>
<code>compile()</code>	<code>hex()</code>	<code>ord()</code>	<code>super()</code>
<code>complex()</code>	<code>id()</code>	<code>pow()</code>	<code>tuple()</code>
<code>delattr()</code>	<code>input()</code>	<code>print()</code>	<code>type()</code>
<code>dict()</code>	<code>int()</code>	<code>property()</code>	<code>vars()</code>
<code>dir()</code>	<code>isinstance()</code>	<code>range()</code>	<code>zip()</code>
<code>divmod()</code>	<code>issubclass()</code>	<code>repr()</code>	<code>__import__()</code>
<code>enumerate()</code>	<code>iter()</code>	<code>reversed()</code>	
<code>eval()</code>	<code>len()</code>	<code>round()</code>	
<code>exec()</code>	<code>list()</code>	<code>set()</code>	

Ugrađene funkcije mogu da se koriste kao programske naredbe ili kao elementi različitih izraza, npr.

- apsolutna vrednost rezultata evaluacije izraza

```
>>> abs(3+2-7)
2
```

- zaokruživanje zadane decimalne vrednosti

```
>>> round(2.71)
3
```

- dužina zadanog stringa

```
>>> len("Dobar dan")
9
```

- prikaz vrednosti promenljivih i rezultata evaluacije izraza

```
>>> a = "2 + 2 = 4"
>>> print(a)
2 + 2 = 4
>>> a = print("2 + 2 =", 2+2)
2 + 2 = 4
>>> a
```

2.7 Korisničke funkcije

Korisničke funkcije su *imenovane grupe naredbi* koje izvršavaju određeni zadatak i mogu se upotrebiti više puta i tako skratiti kod programa.

Nova funkcija se definiše naredbom **def**, npr.

```
def sum(i1, i2):
    result = i1 + i2
    return result
```

Funkcije mogu da imaju parametre, čije vrednosti se zadaju prilikom svakog pokretanja. Prethodno definisana funkcija `sum` ima dva *paramatera* pod nazivom `i1` i `i2`. Prilikom poziva funkcije zadaju se neke konkretne vrednosti kao aktuelni *argumenti* funkcije, npr.

```
>>> a = 10
>>> b = sum(a, 5)
>>> b
15
```

Korisničke funkcije ne moraju da vraćaju neku vrednost kao rezultat. Takve funkcije, nakon izvršenja osnovnog zadatka, za povratak koriste naredbu `return` bez povratne vrednosti.

2.8 Ulaz-izlaz

Naredbe za ulaz-izlaz omogućavaju unos podataka s nekog uređaja u memoriju računara i prenos podataka iz memorije na uređaje za prikaz i štampanje, kao i njihovo stalno čuvanje u obliku fajlova.

U jeziku Python naredbe za interaktivni ulaz i izlaz realizovane su kao ugrađene funkcije `input()` i `print()`.

Funkcije uvek vraćaju neku vrednost kao rezultat izvršavanja:

- naredba `input()` kao rezultat izvršavanja vraća string koji je interaktivno uneo korisnik;
- naredba `print()` kao rezultat vraća vrednost **None**.

Program naredbom `input()` postavlja pitanje korisniku i čita odgovor, npr.

```
>>> ime = input("Kako se zovete ? ")
Kako se zovete ? Jovana
>>> broj = input("Unesite broj ")
Unesite broj 12
>>> broj
```

'12'

Unesene vrednosti su tekstovi 'Jovana' za ime i cifre '12' za broj. U oba slučaja uneseni tekst je tipa *string*, a konverzija u neki drugi tip vrednosti vrši se pomoću ugrađenih funkcija, kao što su npr. `int()` i `float()`.

Program može da prikaže rezultate na ekranu naredbom `print()`, npr.

```
>>> ime = 'Jovana'
>>> print("Korisnik je", ime)
Korisnik je Jovana
>>>
```

U prethodnim primerima ispis vrednosti promenljivih vrši se u formatu koji određuje sam sistem. Format ispisa podataka se preciznije može zadati korišćenjem funkcije `format`:

```
format(<podatak>, <specifikacija_formata>)
```

gde je *podatak* string ili broj, a *specifikacija formata* je string koji detaljnije opisuje način ispisa podatka, npr. dužinu, tip vrednosti i poravnanje. Tako se npr. ispis stringa, celog ili decimalnog broja u dužini od 10 znakova, dok se od toga za decimalne brojeve odvajaju dva decimalna mesta s desne strane, može opisati posebnim formatima:

```
>>> print(format("tekst", "10s"))
tekst
>>> print(format(3, "10d"))
3
>>> print(format(3.14, "10.2f"))
3.14
```

Tip podatka zadaje se odgovarajućim znakom u specifikaciji formata, odmah iza oznake dužine ispisa. Npr. za string se navodi znak `s`, za celi broj `d` i, a za decimalni broj se koriste `f` ili `e`.

Podrazumeva se levo poravnanje stringova i desno poravnanje numeričkih vrednosti u predviđenoj dužini ispisa. Poravnanje se može eksplicitno definisati korišćenjem znaka `<` za levo poravnanje i znaka `>` za desno poravnanje unutar zadane dužine polja za ispis, npr.

```
>>> print(format("tekst", ">10s"))
tekst
>>> print(format(3, "<10d"))
3
>>> print(format(3.14, "<10.2f"))
3.14
```

Detaljniji opis specifikacije formata daje se u poglavlju 10 posvećenom ulazu i izlazu podataka, odnosno radu s fajlovima.

2.9 Primeri programa

U ovom poglavlju će se kao ilustracija upotrebe osnovnih naredbi i ugrađenih funkcija jezika Python prikazati dva kratka programa za brzu analizu stanja telesne mase (težine) odrasle osobe.

2.9.1 Indeks telesne mase

Za analizu uhranjenosti koristi se indeks telesne mase (*Body Mass Index*, BMI) [19], koji predstavlja indikator pothranjenosti ili preterane uhranjenosti, koja predstavlja povećan rizik od oboljevanja od određenih bolesti. Indeks se računa tako što se *masa* u kilogramima podeli s kvadratom *visine* u metrima

$$BMI = \frac{masa}{visina^2}$$

Idealna telesna masa odrasle osobe je kad je vrednost indeksa telesne mase u rasponu od 18,5 do 24,9. Sledeći kratki program računa i prikazuje indeks telesne mase na osnovu podataka o težini i visini osobe:

```
# Program računa indeks telesne mase osobe

# Unos telesne težine u kg
težina = float(input("Unesite telesnu težinu (kg): "))

# Unos visine u cm
visina = float(input("Unesite visinu (cm): "))

# Računanje indeksa telesne mase
indeks_tm = težina / ((visina/100) ** 2)

# Prikaz rezultata
print("Indeks telesne mase je", indeks_tm)
```

Rezultat izvršavanja programa, za prikazane podatke je:

```
Unesite telesnu težinu (kg): 87
Unesite visinu (cm): 184
Indeks telesne mase je 25.697069943289225
```

2.9.2 Indeks telesne mase s ocenom stanja

Program iz prethodne tačke proširen je tako da na osnovu indeksa telesne mase daje opisnu ocenu stanja telesne mase (težine), koje može biti: *nedovoljna*, *normalna*, *povećana težina* ili *gojaznost*.

```
# Program računa indeks telesne mase i daje opisnu ocenu

# Unos telesne težine u kg
težina = float(input("Unesite telesnu težinu (kg): "))

# Unos visine u cm
visina = float(input("Unesite visinu (cm): "))

# Računanje indeksa telesne mase
indeks_tm = težina / ((visina/100) ** 2)

# Prikaz rezultata
print("Indeks telesne mase je", indeks_tm)

if indeks_tm < 18.5:
    print("Nedovoljna težina")
elif indeks_tm < 25:
    print("Normalna težina")
elif indeks_tm < 30:
    print("Povećana težina")
else:
    print("Gojaznost")
```

Primer rezultata izvršavanja programa za iste ulazne podatke je:

```
Unesite telesnu težinu (kg): 87
Unesite visinu (cm): 184
Indeks telesne mase je 25.697069943289225
Povećana težina
```

Pitanja za ponavljanje

1. Navedite tipične programske elemente proceduralnih programa.
2. Da li se u jeziku Python uvek razlikuju velika i mala slova u naredbama?
3. Koja je razlika između fizičke i logičke linije programa?
4. Šta je oblast definisanosti promenljivih u jeziku Python?
5. Šta su izrazi u jeziku Python i koje vrste izraza postoje?
6. Navedite najvažnije vrste operatora u jeziku Python.
7. Koliko vrsta naredbi za grananje i ponavljanje postoji u jeziku Python?
8. Koja su osnovne razlike između ugrađenih i korisničkih funkcija?
9. Koji je tip podatka koji se učitava pomoću funkcije input?
10. Kako se može precizno definisati format ispisa podataka u jeziku Python?

3 Izrazi u jeziku Python

1. Uvod
2. Izrazi
3. Operatori
4. Prioritet operatora
5. Konverzija tipova
6. Zaokruživanje
7. Primeri programa

U ovom poglavlju opisuje se upotreba izraza u jeziku Python. Izrazi omogućavaju računanje različitih numeričkih i nenumeričkih vrednosti, koje se koriste u različitim naredbama jezika Python.

3.1 Uvod

Izrazi u proceduralnim programskim jezicima opisuju računanja koja proizvode jedinstvenu vrednost, koja se može dodeliti promenljivoj ili koristiti kao vrednost argumenta neke funkcije, kao npr.

<code>a = 2 + 3.14</code>	izraz kao deo naredbe za dodelu vrednosti
<code>print(2 + 3.14)</code>	izraz kao argument funkcije

3.2 Izrazi

Izrazi u jeziku Python predstavljaju ispravne kombinacije *vrednosti* (konstanti), *promenljivih*, *funkcija* i *operatora*. Sintaksa izraza u jeziku Python dosta je složena (videti u [26]), pa se neće eksplicitno navoditi u ovom materijalu.

Nakon izračunavanja (evaluacije) izrazi daju rezultat određenog tipa, koji se može koristiti u drugim naredbama, npr. u naredbi dodele vrednosti, u uslovima naredbi grananja i ponavljanja ili kao argument u pozivu funkcije. Rezultat može biti numerički, nenumerički ili logički (logički tip je podtip tipa `int`).

Dozvoljena je upotreba operacija nad različitim tipovima vrednosti u jednom izrazu, tako što se eksplicitno ili implicitno izvrši njihovo pretvaranje u neki zajednički (kompatibilni) tip vrednosti. Npr. u izrazu

```
a = 2 + 3.14
```

zbir celog i decimalnog broja vrši se nakon implicitnog pretvaranja celog broja u decimalni broj 2.0, a zatim se kao rezultat dobija decimalni broj 5.14.

Redosled operacija prilikom izračunavanja izraza zavisi od tipa vrednosti i upotrebljenih operatora. Svi operatori istog prioriteta evaluiraju se s leva u desno. Zgrade menjaju redosled evaluacije svih tipova operatora.

Za aritmetičke izraze, redosled evaluacije izraza određuje se prema redosledu operacija u matematici, po prioritetu operatora i zagradama, npr.

`(3 + 4) * 5 ** 2` rezultat evaluacije je 175

Kod logičkih operatora, operator `and` ima prednost nad operatorom `or`, npr.

`True or False and False` rezultat evaluacija je `True`

`False or True and False` rezultat evaluacija je `False`

U gornjim primerima prvo se evaluira operator `and`, a zatim operator `or`, tako da je redosled kao da se koriste zgrade:

`True or (False and False)` rezultat evaluacija je `True`

`False or (True and False)` rezultat evaluacija je `False`

U odnosu na broj argumenata operatori mogu biti *unarni* (jedan argument) i *binarni* (dva argumenta). Složeni izrazi se sastoje od više operatora i operanada različitih tipova.

Operatori u jeziku Python definisani su za određeni tip vrednosti na koje se mogu primeniti. Ako se pokuša upotreba operatora nad vrednostima za koje nije definisan, interpreter će dojaviti grešku izvršavanja, npr.

```
>>> 2+"2"
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    2+"2"
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>>
```

Aritmetički operator `+` definisan je za operande koji predstavljaju numeričke vrednosti. Ako se operand tipa string `"2"` konvertuje u (celi) broj pomoću funkcije za konverziju `int()`, izraz će se ispravno evaluirati:

```
>>> 2+int("2")
4
>>>
```

3.3 Operatori

Operatori su simboli koji označavaju *operacije* koje se mogu izvršiti nad jednim ili više *operanada*. Najčešći se koriste *unarni* operatori, koju imaju jedan operand i *binarni* operatori, koji vrše operacije nad dva operanda. U ovom poglavlju opisać će se operatori koji se koriste u izrazima jezika Python razvrstani u grupe prema vrsti računanja i tipovima operanada:

- aritmetički operatori,
- operatori poređenja,
- specijalni operatori uz dodelu vrednosti,
- logički operatori,
- operatori za rad s bitovima,
- operatori pripadnosti,
- operatori identifikacije.

3.3.1 Aritmetički operatori

Aritmetički operatori postoje za četiri osnovne aritmetičke operacije: stepenovanje, celobrojno deljenje i računanje ostatka od deljenja:

- +** - sabiranje numeričkih vrednosti i unarni operator predznaka,
- - oduzimanje numeričkih vrednosti i unarni operator predznaka,
- *** - množenje numeričkih vrednosti,
- /** - deljenje numeričkih vrednosti, decimalni rezultat,
- //** - celobrojno deljenje, decimalni deo se odbacuje,
- **** - stepenovanje i
- %** - ostatak od deljenja (modul).

Aritmetički izrazi se evaluiraju prema prioritetima operatora u matematici, s tim da se operatori istog prioriteta evaluiraju s leva u desno. Npr. evaluacija izraza

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

vrši se kao da je unesen izraz

```
>>> 2+(3*2)-(2**3)+1
1
```

Operator deljenja vraća decimalni rezultat, a operator celobrojnog deljenja kao rezultat daje celobrojni deo decimalnog rezultata, npr.

```
>>> 5/2
2.5
>>> 5//2
2
```

Ostatak od deljenja ili modul daje ostatak deljenja prvog operanda drugim, npr.

```
>>> 5 % 2
1
>>> 9 % 3
0
```

Operandi ne moraju da budu samo celi brojevi, npr.

```
>>> 9.5 % 3
0.5
>>> 9 % 3.5
2.0
```

U jeziku Python oznake aritmetičkih operatora sabiranja i množenja (+ i *) koriste se i kao oznake za operacije s nenumeričkim vrednostima. Za argumente tipa *string*,

- operator + označava konkatenciju (nastavljanje, spajanje) stringova,
- operator * u kombinaciji s brojem označava višestruku konkatenciju, odnosno ponavljanje (repeticiju) stringa.

Primeri primene ovih operatora na nenumeričke vrednosti su:

- za kreiranje novog stringa spajanjem više stringova:

```
>>> "termo" + "metar"
'termometar'
>>>
```

- za kreiranje novog stringa višestrukim ponavljanjem i spajanjem:

```
>>> "Snežana i" + " patuljak"*7
'Snežana i patuljak patuljak patuljak patuljak
patuljak patuljak patuljak'
>>>
```

3.3.2 Operatori poređenja

Operatori poređenja ili *relacioni* operatori su:

- <** - manje od,
- <=** - manje od ili jednako,
- >** - veće od,
- >=** - veće od ili jednako,
- !=** - različito (nije jednako) i
- ==** - jednako (dva znaka jednakosti, jer jedan označava dodelu vrednosti).

Rezultat poređenja dve vrednosti pomoću relacionih operatora je logička vrednost, `True` ili `False`. Nekoliko primera računanja relacionih izraza:

```
>>> 2 > 3
False
>>> 3 != 2
True
>>> 3 == 2
False
>>> 3 >= 3
True
>>>
```

Jednakost dveju vrednosti ispituje se operatorom `"=="`.

```
>>> 3 == 3
True
>>>
```

Pokušaj upotrebe samo jednog znaka jednakosti za poređenje vrednosti prouzrokuje grešku izvršavanja:

```
>>> 3 = 3
SyntaxError: can't assign to literal
>>>
```

3.3.3 Specijalni operatori uz dodelu vrednosti

Neki aritmetički operatori mogu se kombinovati s operatorom dodele vrednosti radi skraćivanja teksta naredbi programa i to:

+, -, *, /, //, % i **

Skraćivanje zapisa često korišćenih naredbi s jednostavnim aritmetičkim izrazima, kojima se vrši inkrement i dekrement promenljive, preuzeto je iz jezika C i C++. Osim skraćivanja teksta programa, ovakva tehnika je pomagala prevodiocu da optimizuje prevođenje izraza u efikasniji mašinski kod. Npr. umesto naredbe

```
brojac = brojac + 1
```

dozvoljeno je pisati kraće

```
brojac += 1
```

Promenljiva se može menjati na oba načina, npr.

```
>>> brojac = 0
>>> brojac = brojac + 1
>>> brojac
1
>>> brojac += 1
>>> brojac
2
>>>
```

U prethodnom primeru inicijalna vrednost brojača povećana je za jedan dva puta, jednom na klasičan način, a drugi put pomoću skraćenog izraza dodele vrednosti.

U skraćenom izrazu dodele vrednosti ne sme postojati praznina između ovako kombinovanih operatora, npr.

```
>>> brojac + 1
SyntaxError: invalid syntax
>>>
```

Veća efikasnost skraćenog zapisa zasniva se na posebnim instrukcijama procesora za inkrement i dekrement, koje se izvršavaju kao unarne. Pošto je fiksni iznos povećanja vrednosti poznat, pristup drugom operandu nije potreban, čime se štedi vreme procesora. Savremeni prevodioci prepoznaju ovakve naredbe i sami vrše neophodne optimizacije.

3.3.4 Logički operatori

Logički operatori nad celobrojnim vrednostima su

- not** - unarni operator negacije,
- and** - binarni operator logičko I,
- or** - binarni operator logičko ILI.

Logički operatori vrše operacije na vrednostima logičkog tipa. Unarne operacija negacije (`not`) vraća negiranu operanda, npr. `True` u `False` i obrnuto.

Rezultati binarnih operacija I (`and`) i ILI (`or`) nad različitim logičkim vrednostima prikazani su u tabeli:

operand 1	operand 2	and	or
True	True	True	True
True	False	False	True
False	True	False	True
False	False	False	False

Za ilustraciju, navešće se nekoliko primera evaluacije jednostavnijih logičkih izraza:

```
>>> a = True
>>> not a
False
>>> a and True
True
>>> a and False
False
>>> a or True
True
>>> a or False
True
>>> not (a and True or False)
False
```

Ako bilo koji operand operatora I (`and`) ima vrednost `False`, rezultat evaluacije je `False`. Takođe, ako je bilo koji od operanada operatora ILI (`or`) ima vrednost `True`, rezultat je `True`.

Prevodilac jezika Python koristi ova svojstva da ubrza evaluaciju ovih operatora, tako da, kad ustanovi da prvi operand operatora I ima vrednost `False`, odnosno prvi operand operatora ILI ima vrednost `True`, ne vrši dalje računanje, već odmah za operator I vraća rezultat `False`, odnosno za operator ILI rezultat `True`. Zbog toga se ovi binarni operatori nazivaju uslovnim ili kratko spojenim (*short-circuit*) operatorima.

Primer praktične upotrebe izraza je kratki program koji učitava godinu i određuje da li je prestupna i ispisuje rezultat:

```
# Program za određivanje prestupne godine

#Unos godine
godina = int(input("Unesi godinu: "))

# Određivanje da li je godina prestupna
prestupna = (godina % 4 == 0 and godina % 100 != 0) \
            or (godina % 400 == 0)

# Prikaz rezultata
if prestupna:
    print("Godina", godina, "je prestupna!")
else:
    print("Godina", godina, "nije prestupna.")
```

Primer rezultata izvršavanja ovog programa je:

```
Unesi godinu: 2017
Godina 2017 nije prestupna.
```

3.3.5 Operatori za rad s bitovima

Operatori za rad s bitovima u jeziku Python su:

- ~ - negacija na nivou bita (*bitwise NOT*), unarna operacija,
- & - operacija I nad bitovima (*bitwise AND*),
- ^ - operacija ekskluzivno ILI nad bitovima (*bitwise XOR*),
- | - operacija ILI nad bitovima (*bitwise OR*),
- << - pomeranje bitova ulevo određen broj mesta (*shift left*) i
- >> - pomeranje bitova udesno određen broj mesta (*shift right*).

Operacije nad nizovima bitova koriste se npr. u kriptologiji, za kodiranje i dekodiranje, kao i za rad s uređajima i komunikacionim protokolima.

Binarni prikaz sadržaja može se dobiti ugrađenom funkcijom `bin()`, koja može da prikaže efekte izvršavanja binarnih operatora. Navešće se nekoliko primera izvršavanja operatora negacije na nivou bita:

```

>>> a = 8                                a 00001000
>>> bin(a)
'0b1000'
>>> b = ~a                                b 11110111
>>> bin(b)
'0b1001'
>>> b
-9

```

Primeri rezultata izvršavanja operacija I i ILI na nivou bita nad prethodno definisanim promenljivima su:

```

>>> c = a & b                            c 00000000
>>> bin(c)
'0b0'
>>> d = a | b                            d 11111111
>>> bin(d)
'-0b1'
>>> d
-1

```

Efekti operacije pomeranja sadržaja promenljive a za dva bita ulevo su:

```

>>> e = a << 2                          e 00100000
>>> bin(e)
'0b100000'
>>> e
32

```

Operator ekskluzivno ILI (*bitwise XOR*) daje rezultat istoimene logičke operacije nad bitovima operanada. Treba napomenuti da oznaka ovog operatora u nekim programskim jezicima predstavlja oznaku operatora stepenovanja. U jeziku Python, stepenovanje se označava sa dve zvezdice. Primena ova dva operatora na iste argumente daje sasvim različite rezultate, npr.

```

>>> print(2^3, 2**3)
1 8

```

3.3.6 Operatori pripadnosti

Operatori pripadnosti (*membership operators*) u izrazima jezika Python su [2]:

- in** - proverava pripadnost neke vrednosti zadanom nizu vrednosti, npr. nizu znakova (string), listi ili n-torci;
- not in** - negacija pripadnosti nekom nizu vrednosti.

Npr. primena operatora pripadnosti na listu vrednosti u strukturi n-torke daje sledeć rezultate:

```
>>> 10 in (10, 20, 30)
True
>>> "plavo" in ("cveno", "zeleno", "plavo")
True
>>> 20 not in (10, 20, 30)
False
```

Pošto su stringovi liste znakova, operator pripadnosti može da proveri da li je neki string deo drugog stringa, npr.

```
>>> "španska" in "Dolazi španska inkvizicija"
True
```

Operatori vraćaju logičku vrednost, koja se može koristiti u logičkim izrazima.

3.3.7 Operatori identifikacije

Operatori identifikacije (*identity operators*) su [26]:

- is** - proverava da li dve promenljive pokazuju na isti objekt u memoriji,
- is not** - daje suprotnu vrednost provere identičnosti objekata.

Operatori identifikacije proveravaju identite objekata: `x is y` daje `True` ako je u pitanju isti objekt, odnosno ista memorijska adresa. Rezultat je ekvivalentan računanju izraza `id(x) == id(y)`, npr.

```
>>> a = "a"
>>> "aa" is a+a
False
>>> "aa" == a+a
True
```

U prvom izrazu se proverava da li su identične adrese konstante "aa" i rezultata evaluacije izraza `a+a`, a u drugom da li su njihove vrednosti iste.

3.4 Prioritet operatora

Pregled prioriteta izvršavanja razmatranih operatora u izrazima jezika Python prikazan je u sledećoj tabeli :



Operator	Naziv
**	eksponent (<i>exponentiation</i>)
+, -, ~x	unarni operatori predznaka i negacija na nivou bita (<i>bitwise NOT</i>)
*, @, /, //, %	množenje, množenje matrica, deljenje i ostatak deljenja
+, -	dodavanje i oduzimanje (<i>addition, subtraction</i>)
<<, >>	pomeranje bitova (<i>shifts</i>)
&	operacija AND nad bitovima (<i>bitwise AND</i>)
^	operacija XOR nad bitovima (<i>bitwise XOR</i>)
 	operacija OR nad bitovima (<i>bitwise OR</i>)
in, not in, is, is not, <, <=, >, >=, !=, ==	poređenja i testovi pripadnosti (<i>in</i>) i identiteta (<i>is</i>)
not x	logička operacija NOT (<i>boolean NOT</i>)
and	logička operacija AND (<i>boolean AND</i>)
or	logička operacija OR (<i>boolean OR</i>)

Vidi se da najviši prioritet u evaluaciji izraza imaju operator eksponenta i unarni operatori, a najniži logički operatori.

3.5 Konverzija tipova

Jezik Python prilikom evaluacije operacija u izrazima vrši prilagođavanje tipova vrednosti, tako što se vrši implicitna konverzija operanada u kompatibilni tip. Npr. aritmetička operacija nad celim i decimalnim vrednostima vraća kao rezultat decimalnu vrednost. Eksplicitna konverzija tipa vrednosti vrši se ugrađenim funkcijama `int()`, `float()`, `bool()` i `str()`, npr.

```
>>> float("1234.567")
1234.567
>>> bool("1234")
True
>>> int("1234")
1234
>>> str("1234.567")
'1234.567'
```

3.6 Zaokruživanje

Aritmetičke operacije proizvode decimalne rezultate određene preciznosti. Zaokruživanje decimalnih vrednosti vrši se ugrađenom funkcijom `round()`, koja zaokružuje decimalni broj na zadani broj decimala, npr.

```
round(2.71, 1)    daje rezultat 2.7
round(2.71, 2)    daje rezultat 2.71
```

Ukoliko se broj decimala ne navede, zaokruživanje se vrši na celi broj, npr.

```
round(3.14)       daje rezultat 3
round(2.71)       daje rezultat 3
round(3.5)        daje rezultat 4
round(-3.5)       daje rezultat -4
round(-2.71)      daje rezultat -3
```

Funkcija `round()` zaokružuje na najbliži broj zadane preciznosti. Zbog problema decimalno-binarnе konverzije [1], ponekad se zaokruživanjem na određeni broj decimala dobiju neočekivani rezultati, npr.

```
round(2.675, 2)   daje rezultat 2.67, a ne 2.68
```

Poseban vid zaokruživanja na celi broj je prosto odbacivanje decimalnog dela. U jeziku Python, to se postiže ugrađenom funkcijom `int()`, npr.

```
int(3.5)          daje rezultat 3
int(2.71)         daje rezultat 2
int(-3.5)         daje rezultat -3
```

3.7 Primer programa

Za ilustraciju praktične upotrebe izraza prikazaće se program koji izračunava broj dana između dva datuma.

Kalendarsko merenje vremena zasniva se na trajanju solarne godine, odnosno jednoj rotaciji zemlje oko sunca, koje se deli na dane, koji odgovaraju jednoj rotaciji zemlje oko svoje ose. Ovaj količnik nije celi broj, nešto je veći od 365, tako da se kalendar u danima mora povremeno usklađivati s trajanjem stvarnih astronomskih pojava.

Datum se u svim kalendarima računa brojanjem dana u odnosu na neki fiksni datum u prošlosti, koji zavisi od izabranog kalendara (jevrejski, julijanski, gregorijanski, islamski, itd.). Računarski početak računanja često je datum u neposrednoj prošlosti, npr. za Gregorijanski kalendar to je dan njegovog uvođenja 1582. godine, kada je nastao diskontinuitet u kalendaru.

Broj dana n između zadanog datuma $dan.mesec.godina$ i nekog fiksnog početnog datuma, npr. početka nove ere, može se dobiti pomoću aritmetičkog izraza [27]:

$$n = 365 \cdot g + g // 4 - g // 100 + g // 400 + (m \cdot 306 + 5) // 10 + dan - 1$$

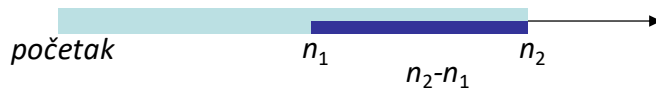
gde su:

$$m = (mesec + 9) \% 12$$

$$g = godina - m // 10$$

Broj dana između dva datuma računa se jednostavno kao razlika broja dana od početka računanja za prvi (n_1) i drugi datum (n_2):

$$n = n_2 - n_1$$



Osnovnu algoritam računarskog rešenja je:

1. učitati dan_1 , $mesec_1$, $godina_1$
2. učitati dan_2 , $mesec_2$, $godina_2$
3. prilagoditi brojanje meseci $m_1 = (mesec_1 + 9) \% 12$ i $m_2 = (mesec_2 + 9) \% 12$
4. prilagoditi brojanje godina $g_1 = godina_1 - m_1 // 10$ i $g_2 = godina_2 - m_2 // 10$
5. izračunati redne brojeve dana:
 $n_1 = 365 \cdot g_1 + g_1 // 4 - g_1 // 100 + g_1 // 400 + (m_1 \cdot 306 + 5) // 10 + dan_1 - 1$
 $n_2 = 365 \cdot g_2 + g_2 // 4 - g_2 // 100 + g_2 // 400 + (m_2 \cdot 306 + 5) // 10 + dan_2 - 1$
6. Broj dana između dva zadana datuma je $n = n_2 - n_1$.

Program za računanje broja dana između dva datuma u jeziku Python je:

```
""" Računanje broja dana između dva datuma
po Gregorijanskom kalendaru (od oktobra 1582):
1. učitati datum1 i datum 2 (dan, mesec, godina)
2. prilagoditi brojanje meseci
   m1= (mesec1+9) % 12 i m2= (mesec2+9) % 12
3. prilagoditi brojanje godina
   g1 = godina1 - m1//10 i g2 = godina2 - m2//10
4. izračunati redne brojeve dana n1 i n2:
   n = 365*g+g//4-g1//100+g1//400+(m*306+5)//10+dan-1
5. Broj dana je n = n2 - n1
"""
```

```

# Unos prvog datuma
print("Unesite prvi datum")
dan1 = int(input(" dan: "))
mesec1 = int(input(" mesec: "))
godinal = int(input(" godina: "))

# Unos drugog datuma
print("Unesite drugi datum")
dan2 = int(input(" dan: "))
mesec2 = int(input(" mesec: "))
godina2 = int(input(" godina: "))

# Prilagođavanje brojanja meseci (mart je 01) i godina
# (u skladu s brojanjem meseci)
m1= (mesec1+9) % 12
m2= (mesec2+9) % 12
g1 = godinal - m1 // 10
g2 = godina2 - m2 // 10

# Računanje rednih brojeva dana
n1 = 365*g1 + g1 // 4 - g1//100 + g1 // 400 + \
    (m1*306+5) // 10 + dan1 - 1
n2 = 365*g2 + g2 // 4 - g2//100 + g2 // 400 + \
    (m2*306+5) // 10 + dan2 - 1

# Računanje broja dana između dva datuma i ispis
n = n2 - n1
print("Broj dana između", dan1, mesec1, godinal, " i ", \
    dan2, mesec2, godina2, " je ", n)

```

Primer izvršavanja programa kada se traži broj dana između Dana primirja u prvom svetskom ratu i istog datuma 2017. godine je:

```

Unesite prvi datum
dan: 11
mesec: 11
godina: 1918
Unesite drugi datum
dan: 11
mesec: 11
godina: 2017
Broj dana između 11 11 1918 i 11 11 2017 je 36160

```

Pitanja za ponavljanje

1. Šta predstavljaju izrazi i šta proizvode u proceduralnim programskim jezicima?
2. Koje vrste operatora prema vrsti računanja i tipovima operandata postoje u jeziku Python?
3. Navedite najvažnije aritmetičke operatore u jeziku Python.
4. Koji aritmetički operatori se koriste i za operacije s nenumeričkim vrednostima (stringovima)?
5. Šta predstavljaju specijalni operatori dodele vrednosti u jeziku Python?
6. Objasnite koji se operatori nazivaju kratko spojeni (*short-circuit*) operatori?
7. Čemu služe operatori za rad s bitovima? Koja je namena operatora `>>` i `<<`?
8. Koji su i kakve vrednosti proizvode operatori pripadnosti?
9. Šta ispituju operatori identifikacije?
10. Koje ugrađene funkcije vrše konverziju između numeričkih vrednosti?
11. Koje ugrađene funkcije vrše konverziju između numeričkih i nenumeričkih vrednosti?
12. Koje ugrađene funkcije vrše zaokruživanje decimalnih vrednosti? Navedite tipične primere.

4 Upravljanje tokom izvršavanja programa: grananje i ponavljanje

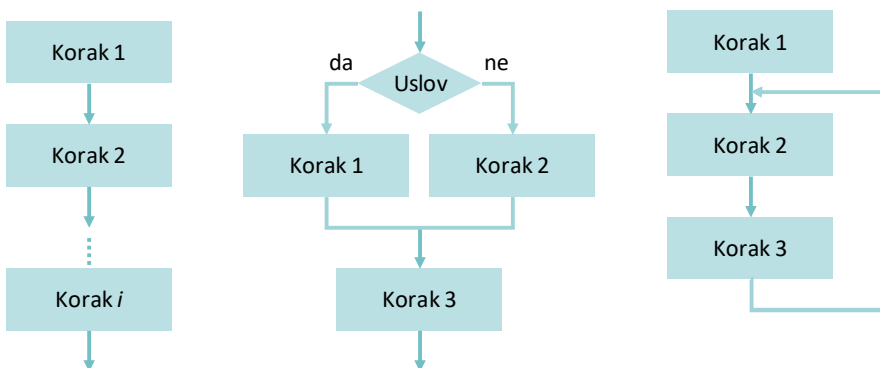
1. Uvod
2. Grananje (selekcija)
3. Ponavljanje (iteracija)
4. Obrada izuzetaka
5. Primeri programa

U ovom poglavlju izlažu se osnovni koncepti upravljanja tokom izvršavanja programa u proceduralnim programskim jezicima i ukratko opisuju naredbe grananja (selekcije) i ponavljanja (iteracije) u jeziku Python.

4.1 Uvod

Za realizaciju bilo kog algoritma dovoljna su tri načina upravljanja redosledom izvršavanja pojedinih koraka algoritma, Sl. 11:

- *sekvencijalno* izvršavanje, jedan korak za drugim (podrazumeva se),
- *uslovno* izvršavanje, gde naredni korak zavisi od određenih uslova i
- *ponavljanje*, gde se određeni niz koraka izvršava više puta.

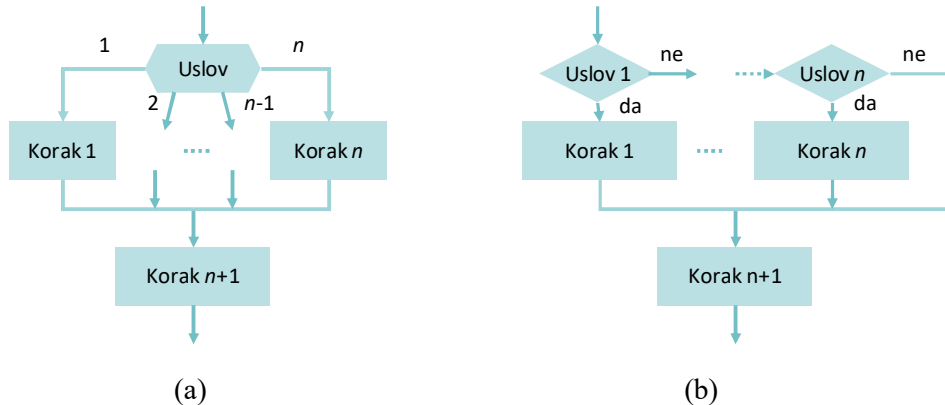


Sl. 11. Načini upravljanja redosledom izvršavanja naredbi programa

4.1.1 Grananje u proceduralnim programskim jezicima

U proceduralnim programskim jezicima grananje se u programu može realizovati prema logičkoj vrednosti (dve alternative) ili na osnovu vrednosti nekog drugog tipa (više od dve alternative).

Jezik Python podržava grananja prema logičkom uslovu, odnosno nizu povezanih logičkih uslova (`if ... elif ... else`).



Sl. 12. Uslovno grananje ispitivanjem vrednosti opšteg tipa i samo binarnih vrednosti

Na Sl. 12 prikazane su ekvivalentne šeme uslovnog grananja, koje se realizuje (a) ispitivanjem vrednosti opšteg tipa, odnosno (b) ispitivanjem samo binarnih vrednosti.

4.1.2 Ponavljanje u proceduralnim jezicima

Ponavljjanje se u proceduralnim programskim jezicima realizuje u odnosu na uslov okončanja ponavljanja (tzv. petlje) na dva načina:

- ponavljanje *unapred određen broj puta*,
- ponavljanje *prema logičkom uslovu*, koji se proverava pre početka ponavljanja ili nakon svakog ponavljanja.

U ponavljanju prema uslovu, prvo izvršavanje niza naredbi podrazumeva početnu istinitost uslova ponavljanja. Niz naredbi koji se ponavlja mora da obezbedi *promenu uslova ponavljanja*, tako da se, pod određenim uslovima, ponavljanje okonča.

Jezik Python podržava ponavljanje unapred određen broj puta (`for`) i ponavljanje prema logičkom uslovu koji se proverava pre početka ponavljanja (`while`).

4.2 Grananje (selekcija)

Naredba grananja omogućava izbor na osnovu *logičkog uslova* sledećeg bloka naredbi koji će se izvršiti, npr.

```
if True:
    print("da")
else:
    print("ne")
```

Blokovi naredbi u jeziku Python označeni su uvlačenjem (indentacijom). Grananje nije ograničeno na ispitivanje jednog logičkog uslova, već se selekcija bloka naredbi može izvršiti na osnovu niza logičkih uslova. Opšti oblik naredbe grananja u jeziku Python je:

```
if <uslov 1>:
    <blok naredbi 1>
elif uslov 2:
    <blok naredbi 2>
...
elif uslov n:
    <blok naredbi n>
else:
    <blok naredbi n+1>
```

Logički uslov je izraz koji nakon evaluacija proizvodi logičku vrednost (`True` ili `False`). To može biti rezultat primene različitih operatora, kao što su operatori poređenja, logički operatori i operatori pripadnosti.

Izraz se izračunava u skladu s prioritetima operatora, npr.

```
a = 2
if a > 10 or a < 0 or a == 6 :
    print("Neispravno")
else:
    print("Ispravno")
```

Pošto relacioni operatori imaju prioritet u odnosu na logičke, izraz u uslovu naredbe grananja izračunava se kao da je unesen u obliku

```
(a > 10) or (a < 0) or (a == 6)
```

Za vrednost `a=2`, nakon evaluacije izraza u zagradama, evaluira se `False or False or False` u `False`, a rezultat izvršavanja je poruka `Ispravno`.

4.3 Ponavljanje (iteracija)

Naredba ponavljanja u jeziku Python omogućava višestruko izvršavanje određenog bloka naredbi. Broj ponavljanja može biti unapred određen (naredba `for`) ili se može postaviti logički uslov za izvršavanje ponavljanja (naredba `while`).

Važno je napomenuti da se logički uslov naredbe `while` proverava *pre* prvog izvršavanja bloka naredbi i mora biti istinit pre početka njenog izvršavanja da se blok naredbi izvrši najmanje jednom.

4.3.1 Naredba for

Naredba za ponavljanje (iteraciju) zadani broj puta ima oblik:

```
for <promenljiva> in <skup_vrednosti>:
    <blok naredbi>
```

Blok naredbi se izvršava za svaku pojedinačnu vrednost iz zadanog skupa vrednosti. Skup vrednosti može biti eksplicitno zadan, nabranjem pojedinačnih vrednosti, ili proizveden, npr. pomoću funkcije `range(<početak>, <kraj-nije-uključen>, <korak>)`, koja omogućava generisanje skupa vrednosti za koje se vrše ponavljanja, npr.

<code>range(1, 10, 1)</code>	skup vrednosti 1, 2, 3, 4, 5, 6, 7, 8, 9
<code>range(20, 1, -2)</code>	skup vrednosti 20, 18, 16, 14, 12, 10, 8, 6, 4, 2
<code>range(10)</code>	skup vrednosti 0, 1, ..., 9
<code>range(1, 11)</code>	skup vrednosti 1, 2, ..., 10
<code>range(0, 10, 3)</code>	skup vrednosti 0, 3, 6, 9
<code>range(0, -10, -1)</code>	skup vrednosti 0, 1, 2, ..., -9
<code>range(0)</code>	prazan skup vrednosti
<code>range(1, 0)</code>	prazan skup vrednosti

`range(0.0)` za decimalne argumente, funkcija nije definisana:

```
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    range(0.0)
TypeError: 'float' object cannot be interpreted as
an integer
```

4.3.2 Naredba while

Osnovna sintaksa naredbe za ponavljanje prema uslovu je:

```
while <uslov>:
    <blok naredbi s modifikacijom uslova>
```

Uslov treba da bude istinit pre izvršavanja petlje da bi se blok naredbi izvršio.

Primer je petlje s uslovom je petlja koja menja brojač unazad s korakom 3, sve dok je vrednost brojača veća od nule:

```
broj = 10
while broj > 0:
    print("Broj=", i)
    broj = broj - 3
```

Modifikacija uslova petlje omogućava okončavanje ponavljanja, jer se vrednost brojača monotono umanjuje ka vrednostima koje nisu veće od nule. Kada bi se naredba za promenu uslova zamenila naredbom `broj= broj+1`, petlja ne bi normalno okončala.

4.3.3 Naredbe za prekid ponavljanja

U jeziku Python definisane su posebne naredbe za prekid ponavljanja. Naredbe `break` i `continue` omogućavaju prekid ponavljanja na dva načina:

1) Naredba `break` terminira petlju, tako da se izvršavanje programa nastavlja od prve sledeće naredbe. Npr. sledeći programski kod dodaje promenljivoj zbir cele brojeve od 1 do 20 i prekida rad kad je vrednost sume veća ili jednaka 100:

```

zbir = 0
broj = 0
while broj < 20:
    broj = broj + 1
    zbir = zbir + broj
    if zbir >= 100:
        break // skok izvan petlje

print("Vrednost promenljive broj je", broj)
print("Zbir je", zbir)

```

Rezultat izvršavanja programskog segmenta je:

```

Vrednost promenljive broj je 14
Zbir je 105

```

Kad ne bi bilo selekcije i naredbe `break`, program bi izračunao zbir brojeva od 1 do 20, koji bi bio 210.

2) Naredba `continue` terminira samo tekuću iteraciju petlje u kojoj se izvršava, tako da program i dalje nastavlja rad u petlji, ali od sledeće iteracije.

Npr. sledeći program sabira cele brojeve od 1 do 20, izuzev brojeva 10 i 11:

```

zbir = 0
broj = 0
while broj < 20:
    broj = broj + 1
    if broj == 10 or broj == 11:
        continue // skok na kraj tekuće iteracije
    zbir = zbir + broj

print("Vrednost promenljive broj je", broj)
print("Zbir je", zbir)

```

Rezultat izvršavanja programa je:

```

Vrednost promenljive broj je 20
Zbir je 189

```

Naredbom `continue` preskače se sabiranje brojeva 10 i 11, tako da je zbir preostalih brojeva $210 - (10 + 11) = 189$.

Obe naredbe predstavljaju naredbe skoka ili bezuslovnog grananja, jer se njihovim izvršavanjem program nastavlja od neke unapred određene naredbe

programa. Ponekad upotreba ovih naredbi može učiniti kod kraćim ili razumljivijim. Npr. program koji pronalazi najmanji delilac različit od 1 za zadani celi broj $n \geq 2$ može se nešto kraće napisati korišćenjem naredbe `break`:

```
n = int(input("Unesi celi broj >= 2: "))
delilac = 2
while delilac <= n:
    if n % delilac == 0:
        break
    delilac = delilac + 1
print("Najmanji delilac broja", n, "je", delilac)
```

Isti algoritam može se realizovati bez korišćenja naredbe skoka, dodavanjem jedne logičke promenljive i proverom njene vrednosti:

```
n = int(input("Unesi celi broj >= 2: "))
nadjen = False
delilac = 2
while delilac <= n and not nadjen:
    if n % delilac == 0:
        nadjen = True
    else:
        delilac = delilac + 1
print("Najmanji delilac broja", n, "je", delilac)
```

Treba napomenuti da instrukcija bezuslovnog skoka postoji u mašinskom jeziku većine procesora, kao što u mnogim starijim višim programskim jezicima postoji ekvivalentna naredba **goto**, čija upotreba programe čini osjetljivijim na programske greške [5]. Pošto se naredbe `break` i `continue` koriste samo u okviru petlji, manje su problematične od ove opšte naredbe bezuslovnog skoka, koja se u modernom programiranju izbegava [1], [5].

U opštem slučaju naredbe skoka je bolje izbegavati i u realizaciji algoritama koristiti samo standardne naredbe selekcije i ponavljanja.

4.4 Obrada izuzetaka

U svakom programu mogu se pojaviti nepredviđene greške, kao što su npr. greške izvršavanja koje nastaju zbog pogrešnog unosa podataka. U takvom slučaju, sistem dojavljuje poruku o grešci i terminira program:

```
>>> celiBroj = input('Unesite celi broj: ')
```

```
Unesite celi broj: x
>>> celiBroj = int(celiBroj) # konverzija u celi broj
Traceback (most recent call last):
File "<pyshell#4>", line 1, in <module>
celiBroj = int(celiBroj)
ValueError: invalid literal for int() with base 10: 'x'
```

Sistemska poruka je tehničke prirode i najčešće je potpuno nerazumljiva krajnjem korisniku. Zbog toga je razvijen poseban sistem obrade ovakvih greški, koje se pojavljuju u toku izvršavanja programa i nazivaju se *izuzeci* (*exceptions*).

Posebna naredba za upravljanje tokom izvršavanja programa omogućava definisanje grupe naredbi koje će se izvršiti u slučaju pojave nekog izuzetka. U takvom bloku naredbi korisniku se može dati detaljniji opis greške i načina na koji se ona može otkloniti. Program ne mora da terminira, već može korisniku da pruži novu priliku da unese ispravan podatak. Naredba `try` ima opšti oblik:

```
try:
    <naredbe koje mogu da izazovu pojavu izuzetka>
except:
    <naredbe za obradu izuzetka>
[else:
    <naredbe za slučaj da se izuzetak ne dogodi>]
```

Ključna `else` nije obavezna i obično se ne navodi, a izvršavanje programa se nastavlja od prve sledeće naredbe iza naredbe `try`. Jedan od načina upravljanje tokom izvršavanja programa prilikom obrade izuzetaka je upotreba petlje za unos određenog podatka, koja se izvršava sve dok unos ne bude ispravan, npr.

```
while True:
    celiBroj = input('Unesite celi broj: ')
    try:
        celiBroj = int(celiBroj)
        break
    except:
        print('Niste uneli celi broj, ponovite.')
        continue
```

Primer ispravnog unosa podatka, nakon ispravke greški je npr.

```
Unesite celi broj: y
Niste uneli ispravan celi broj, pokušajte ponovo.
Unesite celi broj: 2.2
Niste uneli ispravan celi broj, pokušajte ponovo.
Unesite celi broj: 2
>>>
```

4.5 Primeri programa

Kao prvi primer jednostavnog, ali veoma poznatog algoritma, koji u realizaciji koristi uvedene programske elemente prikazaće se Euklidov algoritam za izračinavanje najmanjeg zajedničkog delioca dva cela broja i jednostavna igra pogađanja slučajnog broja, koja ilustruje primenu naredbi za prekid ponavljanja.

4.5.1 Euklidov algoritam

Pronalaženje najmanjeg zajedničkog delioca dva broja vrši se algoritmom poznatim od antičkih vremena (tzv. Euklidov algoritam), koji se narativno može opisati:

Sve dok su brojevi različiti, oduzmi manji od većeg broja.

Algoritam se može preciznije opisati u obliku pseudokoda:

```
Sve dok je  $x \neq y$ ,
    ako je  $x > y$  onda  $x = x - y$ ,
    ako je  $x < y$ , onda  $y = y - x$ .
```

Koja vrsta ponavljanja je odgovarajuća za realizaciju Euklidovog algoritma u jeziku Python?

Ponavljanje oduzimanja definisano je prema uslovu, pa je prirodno da se u jeziku Python realizuje pomoću naredbe `while`. Program koji implementira Euklidov algoritam je:

```
# Euklidov algoritam računanja NZD
# (iterativni)

x = int(input("Unesi X "))
y = int(input("Unesi Y "))
while x != y:
    if x > y:
        x = x - y
    else:
        y = y - x
print("NZD =", x)
```

Primer upotrebe programa za računanje NZD(1.116, 90) je:

```
Unesi X 1116
Unesi Y 90
NZD = 18
>>>
```

4.5.2 Igra pogađanja

Interaktivna igra pogađanja sastoji se u pogađanju "zamišljenog" (slučajno generisanog) celog broja u relativno uskom intervalu.

Program generiše slučajni celi broj u uskom zadanom intervalu i traži od korisnika da pogodi koji je broj u pitanju. Ako se broj koji unese korisnik razlikuje od zamišljenog, korisniku se daje informacija da li je njegov broj veći ili manji od slučajnog broja. Korisnik mora da pogodi broj u konačnom broju pokušaja, inače program prekida rad uz odgovarajuću poruku.

Pseudoslučajni broj generiše se funkcijom `randrange(od, do)` iz modula `random`.

Jednostavna igra pogađanja brojeva može se realizovati sledećim programom u jeziku Python:

```
# Program za pogađanje slučajnog broja
import random

MAX_POKUSAJA = 5 # najveći broj pogađanja
MAX_BROJ      = 20 # najveći slučajni broj

# Uvodno obaveštenje
print("Program za pogađanje slučajnog broja"\
      "između 1 i", MAX_BROJ, ".")
print("Dozvoljeno je", MAX_POKUSAJA, "pokušaja.")

# Generisanje slučajnog broja
broj = random.randrange(1, MAX_BROJ+1)

brojacPokusaja = 0 # inicijalizacija brojača
while True:      # beskonačna petlja

    # Unos pogađanja
    pokusaj= int(input('\nPogodite slučajni broj: '))
    brojPokusaja = brojPokusaja + 1 # brojač pokušaja
```

```

# Ako je broj pogođen, čestitka i prekid petlje
if pokusaj == broj:
    print("Čestitamo, pogodili ste broj!")
    print("Broj pokušaja:", brojacPokusaja)
    break
elif pokusaj < broj:
    print("Suviše mali broj.") # broj premali
else:
    print("Suviše veliki broj.") # broj prevelik

# Ako je dostignut najveći broj pokušaja, prekid
if brojacPokusaja == MAX_POKUSAJA:
    print("Nažalost, niste pogodili u", \
          MAX_POKUSAJA, "pokušaja.")
    print("Slučajni broj je bio:", broj)
    break

print("Hvala na igri.")

```

Primer izvršavanja programa, u kome je korisnik pogodio broj je:

**Program za pogađanje slučajnog broja između 1 i 20 .
Dozvoljeno je 5 pokušaja.**

**Pogodite slučajni broj: 10
Suviše mali broj.**

**Pogodite slučajni broj: 15
Suviše veliki broj.**

**Pogodite slučajni broj: 13
Čestitamo, pogodili ste broj!
Broj pokušaja: 3
Hvala na igri.
>>>**

Pitanja za ponavljanje

1. Koji se načini upravljanja tokom izvršavanja programa dovoljni za realizaciju bilo kog algoritma?
2. Da li jezik Python ima naredbu za grananje programa prema vrednosti koja može biti bilo kog tipa?
3. Da je naredba grananja u jeziku Python ograničena na ispitivanje jednog logičkog uslova?
4. Koje vrste ponavljanja postoje u jeziku Python i koja je njihova osnovna namena?

5. Na koji način se određuje broj ponavljanja u petlji *for*?
6. Da li ugrađena funkcija *range()* omogućava generisanje opadajućeg niza vrednosti?
7. Šta se događa ako logički uslov naredbe ponavljanja prema uslovu (*while*) nije istinit na početku izvršavanja naredbe?
8. Da li naredba ponavljanja prema uslovu obavezno okončava?
9. Koje naredbe u jeziku Python omogućavaju безусловni prekid ponavljanja?
10. Da li su naredbe prekida ponavljanja ekvivalentne naredbi безусловnog skoka u nekim drugim programskim jezicima?

5 Funkcije u jeziku Python

1. Uvod
2. Definicija korisničke funkcije
3. Upotreba funkcije
4. Argumenti funkcije
5. Prostori imena i oblast definisanosti promenljivih
6. Privremene funkcije
7. Primeri programa

U ovom poglavlju se objašnjava koncept upotrebe funkcija i način njihove realizacije u jeziku Python.

5.1 Uvod

Korisničke funkcije su *imenovane grupe naredbi* koje izvršavaju određeni zadatak. Funkcije omogućavaju:

- višestruku upotrebu dela programskog koda i time skraćenje dužine programa i
- bolju organizaciju koda programa njegovom podelom na manje celine, koje je lakše razumeti.

Funkcije imaju listu parametara koje koriste za izvršavanje određenog zadatka, kao npr. funkcija `abs(n)`.

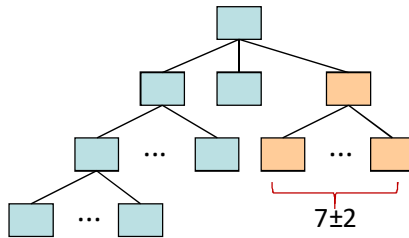
Funkcije mogu (ali ne moraju) da kao rezultat vrate određenu vrednost.

Dekompozicija programskog koda u manje celine neophodna je prilikom razvoja kvalitetnog softvera, da se obiman programski kod podeli na manje i razumljivije celine.

Višestruka dekompozicija proizvodi hijerarhiju manjih programskih celina. Poželjno je na svakom nivou hijerarhije izvršiti podelu na *ograničen* broj manjih celina, jer celina koja se sastoji od velikog broja manjih delova i sama postaje nepregledna i nerazumljiva.

Kao preporuka za *maksimalni* broj grananja u hijerarhiji dekompozicije programa može se uzeti tzv. "Milerov magični broj 7 ± 2 " [19], koji se odnosi na sposobnost čoveka da nakratko zapamti različite sadržaje, npr. niz slova, brojeva ili reči. Ova sposobnost nije povezana s njihovim obimom, već brojem, eksperimentalno ustanovljenim kao 7 ± 2 . Broj predstavlja korisnu heuristiku za izbegavanje dugih lista objekata, koje je teško zapamtiti, odnosno istovremeno obraditi.

Na Sl. 13 prikazana je tipična hijerarhijska dekompozicija, gde se, prema opisanom principu, na svakom nivou hijerarhije pojavljuje najviše 5-9 elemenata, dok dubina hijerarhije ne treba da bude više od nekoliko nivoa.



Sl. 13. Višestruka dekompozicija programskog koda

Treba naglasiti da broj elemenata raste eksponencijalno s dubinom hijerarhije, tako da se veoma složeni programi mogu realizovati u svega nekoliko hijerarhijskih nivoa.

5.2 Definicija korisničke funkcije

Korisnička funkcija se definiše posebnom naredbom oblika:

```
def <nazivFunkcije>(<lista_parametara>):
    <blok naredbi>
```

Definicija funkcije sastoji se od *zaglavlja*, koje čine naziv funkcije i lista parametara i *tela* funkcije, koje čine naredbe grupisane radi izvršavanja nekog zajedničkog zadatka.

Promenljive u listi parametara koje preuzimaju aktuelne vrednosti iz poziva funkcije su *parametri*, a same vrednosti koje se prenose funkciji prilikom njene upotrebe su *argumenti* funkcije.

Izvršavanje funkcije završava poslednjom naredbom ili naredbom `return`, a rad programa se nastavlja od prve sledeće naredbe iza poziva funkcije.

Korisnički definisana funkcija može da vrati neku vrednost kao rezultat pomoću posebne naredbe u telu funkcije, koja ima oblik `return <vrednost>`. Osim toga, dozvoljeno je da funkcija istovremeno vrati i više vrednosti naredbom oblika `return <vrednost1>, <vrednost2>, ...`. Funkcije ne moraju da vrate nikakvu vrednost, odnosno mogu da završe samo naredbom `return` ili čak ibez nje i tada se vraća posebna vrednost `None`.

Funkcija se u programu koristi u izrazima na isti način kao i ugrađene funkcije, pomoću naziva i liste aktuelnih argumenata funkcije.

Npr. funkcija koja vraća vrednost većeg od dva zadana broja može se definisati naredbama:

```
def maksimum(n1, n2):
    if n1 > n2:
        rezultat = n1
    else:
        rezultat = n2
    return rezultat
```

Upotreba (poziv) ove funkcije za aktuelne argumente x i y je:

```
veci = maksimum(x, y)
```

Definisanjem korisničke funkcije izbegava se ponavljanje sličnih blokova naredbi i program postaje kraći i jasniji. Npr. umesto ponavljanja tri veoma slična bloka naredbi, u kojima se traži zbir brojeva u različitim granicama:

```
zbir = 0
for i in range(1,11):
    zbir = zbir + i
print("Zbir brojeva od 1 do 10 je", zbir)
...
zbir = 0
for i in range(21,61):
    zbir = zbir + i
print("Zbir brojeva od 21 do 60 je", zbir)
...
zbir = 0
for i in range(91,151):
    zbir = zbir + i
print("Zbir brojeva od 91 do 150 je", zbir)
...
```

Program je kraći i razumljiviji ako se prvo definiše funkcija `zbir()`, koja se zatim pozove tri puta:

```
def zbir(n1, n2):
    n = 0
    for i in range(n1, n2+1):
        n = n + i
    return n
...
print("Zbir brojeva od 1 do 11 je", zbir(1,10))
...
```

```
print("Zbir brojeva od 21 do 60 je", zbir(21,60))
...
print("Zbir brojeva od 90 do 150 je", zbir(180,220))
...
```

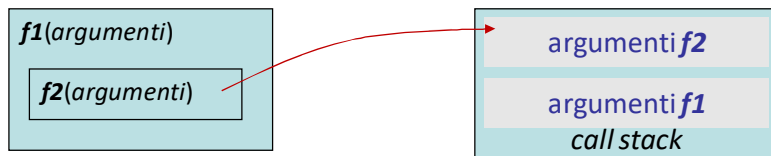
Glavni deo programa često se odvaja u posebnu funkciju, koja se u praksi obično naziva `main()`, po ugledu na programske jezike kao što je C++. U ovom materijalu se ovaj način realizacije glavnog programa neće koristiti.

5.3 Upotreba funkcije

Funkcija se poziva prema svom *nazivu*, uz navođenje liste aktuelnih argumenata. U jeziku Python se iz funkcija mogu pozivati druge funkcije, tako da su dozvoljeni ungnježdeni pozivi. Takođe je dozvoljeno da funkcija sadrži poziv same sebe, odnosno dozvoljeni su i rekurzivni pozivi.

Prilikom svakog aktiviranja funkcije, koja kao programski kod postoji u *jednom primerku*, različiti aktuelni parametri funkcije iz svakog poziva čuvaju se u posebnoj sistemskoj strukturi, tzv. steku izvršavanja ili steku poziva (*call stack*), Sl. 14. Argumenti se koriste obrnutim redom od redosleda uključivanja u strukturu stek (*Last In-First Out*, LIFO).

Funkcija se izvršava za vrednosti argumenata koje se nalaze na vrhu steka (*top*). Na Sl. 14 prvo se izvrši za argumente funkcije *f2*, a nakon toga funkcije *f1*.



Sl. 14. Sistemski stek funkcijskih poziva

5.4 Argumenti funkcije

Aktuelni argumenti se korisničkoj funkciji mogu prenositi po poziciji (redosledu) ili prema njihovom nazivu, u obliku `<naziv>=<vrednost>`. Npr. funkcija *f* definisana kao:

```
def f(p1,p2,p3)
```

može se pozvati s vrednostima aktuelnih argumenata *po poziciji* kao:

```
f(1, 12, 400)
```

ili po nazivima argumenata, koji se tada mogu navoditi proizvoljnim redom, npr.

```
f(p3=400, p1=1, p2=12)
```

Moguće je istovremeno koristiti oba načina prenosa argumenata, ali se obavezno prvo navode vrednosti argumenata po poziciji:

```
f(1, p2=12, p3=400)
```

To znači da se argumenti više ne mogu navoditi po poziciji *nakon* što se prvi put vrednost nekog argumenta prenese po nazivu, tako da je sledeći poziv neispravan:

```
f(1, p2=12, 400)
```

5.4.1 Prenos argumenata po vrednosti

Prilikom izvršavanja proceduralnih programa, prenos argumenata funkciji može se vršiti *po vrednosti* ili *po referenci*. Prenos po vrednosti znači da se funkciji prenosi trenutna vrednost promenljive ili rezultata računanja nekog izraza, dok prenos po referenci znači da se funkciji prenosi samo referenca na vrednost, odnosno informacija o njenom položaju. Kad je aktuelni argument promenljiva, prenos po referenci omogućava funkciji da pristupi promenljivoj i promeni njenu vrednost, dok prenos po vrednosti funkciji prenosi samo kopiju trenutne vrednosti promenljive, koja je smaštena u sistemski stek poziva.

Prenos argumenata funkciji u jeziku Python vrši se po vrednosti (*pass-by-value*). Navođenje naziva promenljive kao aktuelnog parametra samo kopira vrednost promenljive, koja se smešta u stek i zatim koristi u telu funkcije. Npr. funkcija `povecaj` neće promeniti vrednost promenljive `brojac`, čija vrednost se prenosi kao argument, već samo lokalnu vrednost parametra `x`:

<code>def povecaj(x):</code>		
<code>x = x + 1</code>	<code>x</code>	00000002
<code>brojac = 1</code>	<code>brojac</code>	00000001
<code>povecaj(brojac)</code>	<code>brojac</code>	00000001

5.4.2 Varijabilna lista argumenata

Jezik Python pomoću oznake `*` u listi parametara omogućava prenos dodatnih argumenata u pozivu funkcije:

```
def <naziv funkcije>([<lista_par>], *<dodatni_par>):
    <telo funkcije>
    return [<izraz>]
```

Dodatni argumenti koriste se samo po poziciji u promenljivoj < dodatni par > koja ima strukturu tipa fiksne liste, odnosno n-torke. Ako nema dodatnih argumenata, vrednost ove promenljive je prazna n-torka.

Sledeći primer ilustruje upotrebu dodatnih parametara u parametar listi korisnički definisane funkcije:

```
def stampaj(par1, *dodatni_par):
    print("Vrednosti: ")
    print(par1)
    for arg in dodatni_par:
        print(arg)
    return
```

Rezultat izvršavanja poziva funkcije bez dodatnih argumenata je:

```
>>> stampaj(10)
Vrednosti:
10
```

Poziv funkcije s dva dodatna argumenta daje sledeći rezultat:

```
>>> stampaj(20, 30, 40)
Vrednosti:
10
20
30
40
```

5.5 Prostori imena i oblast važenja promenljivih

Prostor imena (namespace) je preslikavanje između naziva objekata i samih objekata, npr. promenljivih i funkcija. U jeziku Python postoje ugrađeni (*built-in*), globalni i lokalni objekti. Prostori imena se mogu gnezditi, odnosno neki prostor imena se može uključiti u drugi prostor imena.

Dostupnost ili oblast definisanosti (*scope*) promenljive predstavlja deo programa u kome je promenljiva definisana i može se koristiti.

Lokalne promenljive nastaju i dostupne su samo u okviru funkcije u kojoj se koriste. Nakon završetka rada funkcije, lokalne promenljive više nisu dostupne. Promenljive se ponovo kreiraju prilikom svakog narednog izvršavanja funkcije.

Globalne promenljive, definisane u glavnom programu, imaju globalnu dostupnost i mogu se koristiti iz svake funkcije. Globalnu dostupnost imaju i ugrađeni (*built-in*) objekti.

Zavisnost od globalnih objekata ograničava višestruku upotrebljivost programskog koda i može biti uzrok programskih greški, pa se njihova upotreba ne smatra dobrom programerskom praksom [2]. Upotreba promenljivih istog naziva s različitim oblastima važenja prikazana je u sledećem primeru programa [2], [3]:

```
def f1():
    a = 2          # Lokalna promenljiva za f1
    def f2():
        a = 3      # Lokalna promenljiva za f2
        print(a)
    f2()
    print(a)
a = 1             # Globalna promenljiva
f1()
print(a)
```

Rezultat izvršavanja programa daje različite vrednosti promenljive *a* u sva tri konteksta:

```
3
2
1
```

Dodavanjem naredbe '*global a*' na početku funkcija *f1* i *f2* menja se njihov način rada, pošto se prilikom izvršavanja više ne kreiraju nove lokalne promenljive, već se koristi postojeća globalna promenljiva definisana u glavnom programu. Rezultat izvršavanja tako izmenjenog programa je:

```
3
3
3
```

5.6 Anonimne funkcije

Anonimna ili *lambda* funkcija je funkcija koja nema naziv, jer se definiše i koristi samo u određenom kontekstu. Osnovna forma definicije *lambda* funkcije je [7], [19]:

```
lambda <lista_parametara>: <izraz>
```

Funkcija kao rezultat vraća rezultat evaluacije izraza.

Anonimne funkcije imaju ograničenu, funkcionalnu formu, tako da ne mogu da sadrže naredbe grananja ni ponavljanja, osim uslovnih izraza.

Npr. umesto definicije funkcije $f(x) = x^2 - 1$ kao posebne funkcije u jeziku Python:

```
def f(x): return x**2-1
```

može se koristiti privremena ili anonimna funkcija (*lambda*), koja računa isti izraz. Ako se definicija zapamti u nekoj promenljivoj, npr.

```
f = lambda x: x**2-1
```

tada se privremena funkcija može koristiti u izrazima na uobičajen način i prenositi kao argument drugim funkcijama:

```
print(f(100))
```

Anonimne funkcije se koriste u funkcionalnom programiranju. Uz *lambda* funkcije, jezik Python je dopunjen s nekoliko naredbi koje omogućavaju korišćenje i ovog načina programiranja [7].

5.7 Primer programa

Praktični primer programa u ovom poglavlju je program za računanje broja dana između dva datuma, samo je u ovom poglavlju realizovan korišćenjem funkcija.

Osnovni algoritam već je opisan u primeru programa iz poglavlja 3: broj dana između dva datuma računa se jednostavno kao razlika broja dana od početka računanja za prvi (n_1) i drugi datum (n_2):

$$n = n_2 - n_1$$

Broj dana n između zadanog datuma *dan.mesec.godina* i nekog fiksnog početnog datuma, npr. početka naše ere, može se dobiti izrazom [27]:

$$n = 365 \cdot g + g // 4 - g // 100 + g // 400 + (m \cdot 306 + 5) // 10 + dan - 1$$

gde su:

$$\begin{aligned} m &= (\text{mesec} + 9) \% 12 \\ g &= \text{godina} - m // 10 \end{aligned}$$

Računanje broja dana od početka računanja može se realizovati kao posebna funkcija `brojDana()`, čiji argument je zadani datum podeljen na komponente *dan*, *mesec* i *godina*.

Program u jeziku Python je, zbog upotrebe funkcije, kraći i jasniji nego implementacija istog algoritma u poglavlju 3, jer nema ponavljanja koda:

```
""" Računanje broja dana između dva datuma
    po Gregorijanskom kalendaru (od oktobra 1582):
    1. učitati datum1 i datum 2 (dan, mesec, godina)
    2. prilagoditi brojanje meseci
        m1= (mesec1+9) % 12 i m2= (mesec2+9) % 12
    3. prilagoditi brojanje godina
        g1 = godina1 - m1//10 i g2 = godina2 - m2//10
    4. izračunati redne brojeve dana n1 i n2:
        n = 365*g+g//4-g1//100+g1//400+(m*306+5)//10+dan-1
    5. Broj dana je n = n2 - n1
    """
```

```
def brojDana (dan, mesec, godina):
    # Prilagođavanje brojanja meseci i godina
    m = (mesec+9) % 12
    g = godina - m//10
    # Računanje rednih brojeva dana
    n = 365*g +g//4 -g//100 +g//400 +(m*306+5)//10+dan-1
    return n
```

```
# Unos prvog datuma
print("Unesite prvi datum")
dan1 = int(input(" dan: "))
mesec1 = int(input(" mesec: "))
godina1 = int(input(" godina: "))

# Unos drugog datuma
print("Unesite drugi datum")
dan2 = int(input(" dan: "))
mesec2 = int(input(" mesec: "))
godina2 = int(input(" godina: "))

# Računanje broja dana između dva datuma i ispis rezult.
print("Broj dana između", dan1, mesec1, godina1, " i ", \
      dan2, mesec2, godina2, " je ", \
      brojDana(dan2, mesec2, godina2) - \
      brojDana(dan1, mesec1, godina1))
```

Primer računanja broja dana od dana završetka prvog svetskog rata, koji se u Srbiji slavi kao Dan primirja u prvom svetskom ratu, do istog datuma 2017. godine je:

```
Unesite prvi datum
dan: 11
mesec: 11
godina: 1918
Unesite drugi datum
dan: 1
mesec: 10
godina: 2020
Broj dana između 11 11 1918 i 1 10 2020 je 37215
>>>
```

Pitanja za ponavljanje

1. Šta su korisničke funkcije i koja je njihova osnovna namena?
2. Objasnite značenje tzv. "Milerovog magičnog broja 7 ± 2 ".
3. Da li korisnička funkcija u jeziku Python uvek mora da vrati jedinstven rezultat?
4. Koja se sistemska struktura koristi prilikom prenosa argumenata korisničkim funkcijama u jeziku Python?
5. Na koji sve načine se mogu zadati vrednosti parametara koje se prenose korisničkoj funkciji?
6. Da li funkcija može da promeni vrednosti aktuelnih argumenata?
7. Objasnite pojmove prostora imena (*namespace*) i dostupnosti promenljivih (*scope*).
8. Navedite vrste promenljivih prema njihovoj dostupnosti.
9. Objasnite zašto se upotreba globalnih promenljivih ne smatra dobrom programerskom praksom.
10. Koja je namena anonimnih (*lambda*) funkcija?

6 Rekurzija u jeziku Python

1. Uvod
2. Definicija rekurzivne funkcije
3. Izvršavanje rekurzivne funkcije
4. Primeri rekurzivnih algoritama
5. Rekurzivne i iterativne verzije algoritama
6. Primeri programa

U ovom poglavlju ukratko se objašnjava pojam rekurzije i upotreba rekurzivanih funkcija u jeziku Python, kao i načini implementacije rekurzivnih algoritama.

6.1 Uvod

Rekurzija omogućava definisanje nekog pojma pomoću njega samog.

Rekurzija u programiranju je poseban način ponavljanja grupa naredbi pomoću funkcija koje pozivaju same sebe.

Primeri rekurzivnih struktura, gde se u delovima ponavlja celina su:

- *u svakodnevnom životu*, slika ogledala u ogledalu ili snimak ekrana koji prikazuje sliku same kamere;
- *u matematici*, mnoge definicije su rekurzivne, npr. definicija prirodnog broja:
 - (a) broj 1 je prirodni broj;
 - (b) ako je n prirodni broj, onda je $n+1$ takođe prirodni broj.
- *u prirodi*, biološke strukture stabla i cveta, oblaci, topografske strukture, galaksije, itd.

6.2 Definicija rekurzivne funkcije

Osnovni oblik definicije rekurzivne funkcije je:

```
def <nazivFunkcije>(<lista_parametara>):
    <blok naredbi, s pozivima nazivFunkcije(...)>
```

Primer rekurzivne funkcije koja ne vraća rezultat je funkcija odbrojavanja unazad:

```
def odbrojavanje(n):
    print(n)
    if n > 1:
        odbrojavanje(n-1)
```

Funkcija štampa redom brojeve $n, n-1, \dots, 2, 1$. Rekurzivno poziva samu sebe s izmenjenom vrednošću argumenta. Za vrednost argumenta $n=1$ funkcija okončava bez novog rekurzivnog poziva. Nakon toga okončavaju sve rekurzivno pozvane funkcije, obrnutim redom od redosleda pozivanja, sve dok ne okonča i originalna funkcija.

Funkcija se može modifikovati tako da odbrojavanje bude između dva zadana broja, unapred ili unazad.

Algoritam rekurzivne funkcije za brojanje *unapred* je:

- prvo se rekurzivno pozove funkcija za prikaz sledećeg broja, a zatim prikaže broj $n_2 > n_1$;
- za $n_2 \leq n_1$ samo se prikaže broj i završava proces pozivanja.

Implementacije funkcije u jeziku Python je:

```
def broj_unapred(n1, n2):
    if n2 > n1:
        broj_unapred(n1, n2-1)
        print(n2)
    else:
        print(n2)
```

Primer brojanja unapred od 2 do 5:

```
>>> broj_unapred(2, 5)
2
3
4
5
>>>
```

Algoritam rekurzivne funkcije za brojanje *unazad* je:

- prvo se prikaže broj $n_2 > n_1$, a zatim rekurzivno pozove funkcija za prikaz sledećeg broja;
- za $n_2 \leq n_1$ samo se prikaže broj i završava proces pozivanja.

Implementacije ove funkcije u jeziku Python je:

```
def broj_unazad(n1, n2):
    if n2 > n1:
        print(n2)
        broj_unazad(n1, n2-1)
    else:
        print(n2)
```

Primer brojanja unazad od 5 do 2 je:

```
>>> broj_unazad(5,2)
5
4
3
2
>>>
```

Funkcija faktorijel u matematici se obično definiše rekurzivno, kao:

$$0! = 1$$

$$n! = n \cdot (n-1)!$$

U jeziku Python rekurzivna funkcija $n!$ može se realizovati direktno na osnovu ove rekurzivne definicije kao:

```
def faktorijel(n):
    if n == 0:
        return 1
    else:
        return n*faktorijel(n-1)
```

Primer računanja broja 9! je:

```
>>> print("9!=", faktorijel(9))
9!= 362880
>>>
```

6.3 Izvršavanje rekurzivne funkcije

U jeziku Python su dozvoljeni rekurzivni pozivi funkcija. Rekurzivno definisana funkcija definiše ponavljanje i u samom kodu funkcije mora da obezbedi uslov terminiranja.

Radi sprečavanja namerne ili slučajne beskonačne rekurzije, u interpreteru jezika Python definisana je maksimalna dubina rekurzivnih poziva funkcija, koja je inicijalno 1.000 poziva. Trenutna vrednost ograničenja dubine rekurzivnih poziva uvek se može ustanoviti pomoću ugrađene funkcije `sys.getrecursionlimit()`.

Pošto je u jeziku Python dubina rekurzivnih poziva ograničena, funkcija koja predstavlja beskonačnu rekurziju ipak okončava, npr.

```
def rekurzija():
    rekurzija()
```

Poziv ovako definisane funkcije daje rezultat:

```
>>> rekurzija()
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    rekurzija()
  File "<pyshell#1>", line 2, in rekurzija
    rekurzija()
    ...
RecursionError: maximum recursion depth exceeded
>>>
```

6.3.1 Upotreba sistemskog steka

Prilikom svakog aktiviranja rekurzivne funkcije, kao i kod drugih funkcija, aktuelni parametri funkcije se čuvaju u posebnoj sistemskoj strukturi, tzv. steku izvršavanja ili steku poziva (*Call Stack*), Sl. 15.

Funkcija se izvršava za aktuelne vrednosti parametara, koje se nalaze na *vrhu steka* (*top*). Stek je struktura slična listi u koju se elementi dodaju i brišu samo s jedne strane, tako da je uvek dostupan samo element koji je poslednji dodat u stek i nalazi se na "vrhu" steka. Zbog ovog ograničenja, stek predstavlja tzv. strukturu LIFO (*Last In, First Out*). Nakon okončanja jednog poziva funkcije, vrednosti aktuelnih parametara s vrha steka poziva se brišu.

Rekurzivno računanje faktoriijela pomoću funkcije iz prethodnog primera može se prikazati sledećom sekvencom:

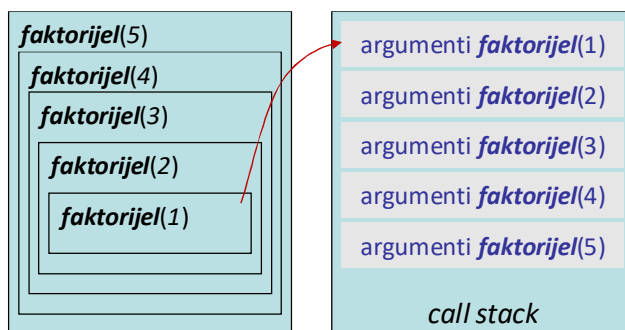
faktorijel(5) =

```
= 5 · faktorijel(4)
= 5 · (4 · faktorijel(3))
= 5 · (4 · (3 · faktorijel(2)))
= 5 · (4 · (3 · (2 · faktorijel(1))))
```

```
= 5 · (4 · (3 · (2 · 1)))
= 5 · (4 · (3 · 2))
= 5 · (4 · 6)
= 5 · 24
= 120
```

U prvoj fazi računanja vrednost aktuelnog argumenta n se pamti u steku, a zatim se rekurzivnim pozivom istoj funkciji prenosi vrednost $n-1$. Proces rekurzivnog pozivanja okončava kad vrednost argumenta n postane nula, kada se prvoj funkciji pozivniku vraća vrednost 1. Vraćena vrednost se množi s argumentom iz steka i proizvod vraća kao rezultat. Postupak se nastavlja, sve dok se stek poziva ne isprazni, odnosno dok se ne vrati konačni rezultat funkcije.

Na Sl. 15 prikazan je izgled steka u fazi rekurzivnog pozivanja, pre početka vraćanja kontrole i množenja argumenata s vrednostima iz steka. Svaki rekurzivni poziv dodaje na vrh steka vrednosti argumenata funkcije iz poziva.



Sl. 15. Sistemski stek poziva prilikom računanja rekurzivne funkcije

U narednoj fazi računanja rezultata, argumenti iz steka se koriste unazad, za množenje s prethodnim rezultatima rekurzivnih poziva, po okončavanju rekurzivno pozvanih primeraka funkcije.

6.4 Primeri rekurzivnih algoritama

Radi ilustracije praktične upotrebe rekurzivnih algoritama, daju se tipični primeri upotrebe rekurzivnih funkcija:

1. Fibonačijev niz
2. Vektorska grafika u jeziku Python (*turtle graphics*)
3. Fraktali i fraktalne funkcije

6.4.1 Fibonačijev niz

Italijanski matematičar Leonardo Bonači (Fibonači) razmatrao je rast populacije zečeva u polju uz (veoma pojednostavljene) pretpostavke:

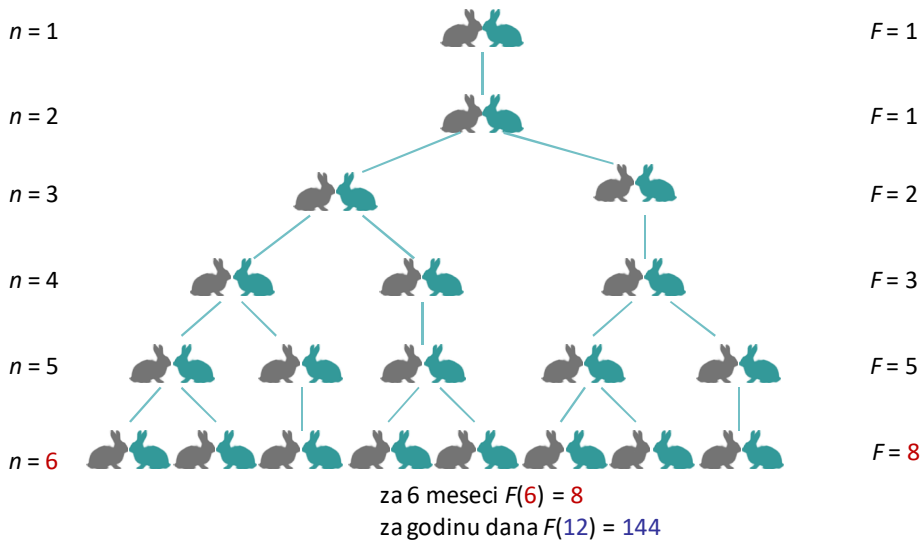
- zečevi su u stanju da se pare u dobu od mesec dana, tako da na kraju svog drugog meseca ženka može da okoti još jedan par zečeva;
- zečevi ne umiru i jedan par zečeva nakon parenja uvek daje novi par, od drugog meseca pa nadalje, i to uvek samo po jednog muškog i jednog ženskog potomka.

Fibonačija je zanimalo: koliko parova zečeva će biti u polju nakon godinu dana? Na kraju prvog meseca zečevi se pare, ali još uvek nema potomaka i postoji samo jedan par. Na kraju drugog meseca ženka okoti novi par, tako da postoje dva para zečeva u polju. Na kraju trećeg meseca, prva ženka okoti novi par, tako da je svega tri para u polju. Na kraju četvrtog meseca, prva ženka okoti još jedan novi par, a ženka rođena pre dva meseca okoti svoj prvi par, tako da je ukupno pet parova zečeva u polju. Na kraju n -tog meseca, broj parova jednak je broju novih (od meseca $n-2$) uvećan za broj postojećih parova (od meseca $n-1$).

Fibonačijev n -ti broj definiše se rekurzivno:

$$\begin{aligned} F(1) &= 1, \quad F(2) = 1 \\ F(n) &= F(n-1) + F(n-2) \end{aligned}$$

Na Sl. 16 prikazan je rast broja zečeva u toku $n=1$ do $n=6$ meseci.



Sl. 16. Fibonačijev broj: prikaz rasta populacije zečeva za $n=6$ meseci

Jednostavni program u jeziku Python za računanje populacije nakon određenog vremenskog perioda je [1]:

```
# Funkcija računa Fibonačijev broj za zadani indeks
def fib(n):
    if n == 1: # prvi nivo
        return 1
    elif n == 2: # drugi nivo
        return 1
    else: # Rekurzivni pozivi
        return fib(n-1) + fib(n-2)

n = int(input("Unesite indeks za Fibonačijev broj: "))

# Računanje i prikaz Fibonačijevog broja
print("Fibonačijev broj za indeks", n, "je", fib(n))
```

Primer računanja Fibonačijevog broja, odnosno populacije nakon 24 meseca je:

```
Unesite indeks za Fibonačijev broj: 24
Fibonačijev broj za indeks 24 je 46368
```

6.4.2 Vektorska grafika u jeziku Python (turtle graphics)

Vektorska grafika je način kreiranja grafičkih prikaza pomoću osnovnih geometrijskih objekata. Poseban pristup kreiranju vektorske grafike je tzv. *turtle* grafika, gde se objekti crtaju pomoću relativnih koordinata i komandi za crtanje geometrijskih objekata koje se izdaju zamišljenoj olovci, popularno nazvanoj "kornjača" (*turtle*). Olovka ili "kornjača" se na ekranu prikazuje u obliku malog trougla, koji pokazuje smer crtanja ►.

Za korišćenje ovakvih grafičkih naredbi u jeziku Python potrebno je prethodno uključiti odgovarajuću programsku biblioteku naredbom `import turtle`.

Neke od osnovnih funkcija iz ove biblioteke su, npr.

- `pendown()` - spuštanje olovke, nakon čega olovka prilikom pomeranja crta liniju odgovarajuće dužine, boje i debljine, koji se mogu podesiti;
- `left(ugao)`, `right(ugao)` - promena smera strelice u izabranom smeru, za zadani ugao;
- `forward(rastojanje)`, `back(rastojanje)` - pomeranje olovke za zadani broj piksela u smeru strelice ili u suprotnom smeru.

Crtanje jednostavnih geometrijskih figura vrši se postavljanjem olovke na izabranu početnu tačku ekrana s koordinatama (x, y), nakon čega se linije crtaju promenom smera i pomeranjem olovke i izabranom smeru za određeni broj tački ekrana (piksela).

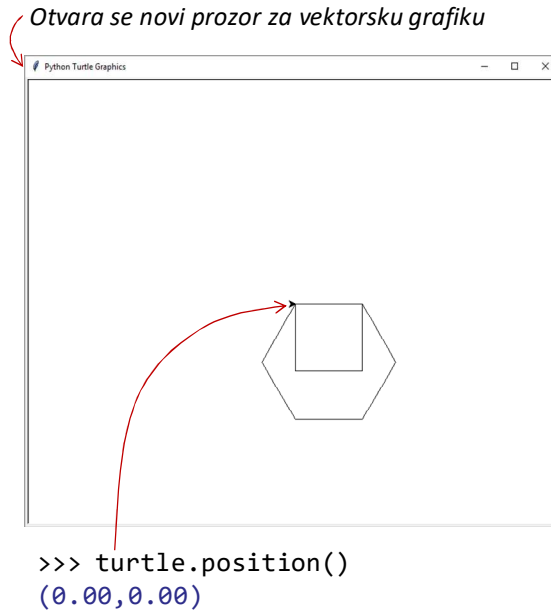
Program za crtanje pravougaonika i šestougla, čije su dužine stranica 100 piksela od podrazumevajuće početne pozicije olovke (0,0), može se izvršiti sledećim kratkim programom u jeziku Python:

```
import turtle
turtle.pendown()

# Pravougaonik
for i in range(4):
    turtle.forward(100)
    turtle.right(90)

# Šestougao
for i in range(6):
    turtle.forward(100)
    turtle.right(60)
```

Pokretanjem programa otvara se novi grafički prozor za prikaz vektorske grafike. Pokazivač (olovka, "kornjača") na početku je u težištu grafičkog prozora, koje ima koordinate $x=0$ i $y=0$, Sl. 17.



Sl. 17. Grafički prozor programa za crtanje pravougaonika i šestougla

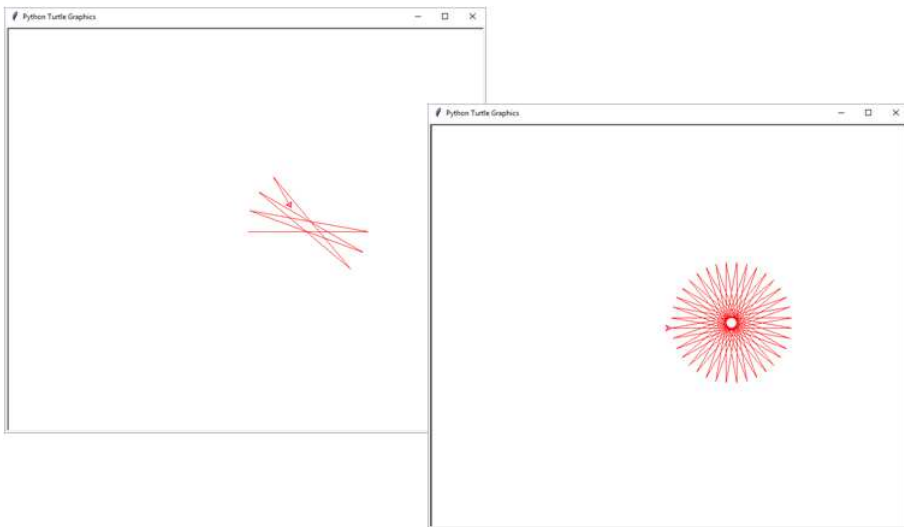
Podrazumevajuća boja pozadine ekrana je bela, a boja pera crna. Pomoću metoda `turtle.color()` može se izabrati druga boja, koja se zadaje kao tekst (engleski naziv ili Veb kod boje) ili numerički, pomoću RGB komponenti. Ako se postavi `turtle.colormode(255)`, sledeće naredbe su ekvivalentne:

```
turtle.color('red')      # naziv boje
turtle.color('#FF0000') # Veb kod crvene boje
turtle.color(255, 0, 0)  # numerički kod crvene boje
```

Iterativni program za crtanje oblika koji podseća na cvet ili zvezdicu određene boje, zadane funkcijom `turtle.color()`, prikazan je na slici:

```
import turtle
turtle.color('red')
crtaj = True
while crtaj:
    turtle.forward(200)
    turtle.left(170)
    crtaj = abs(turtle.position()) >= 1
```

Pokretanjem programa, otvara se grafički prozor i iscrtava niz obojenih duži, koje zajedno daju geometrijsku figuru koja podseća na cvet, Sl. 18.



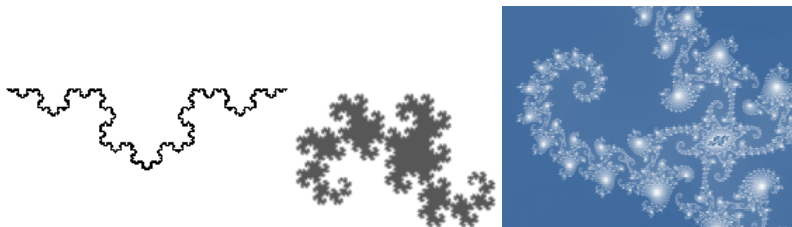
Sl. 18. Primer crtanja u boji oblika cveta (zvezdice)

6.4.3 Fraktali i fraktalne funkcije

Fraktali su geometrijski objekti koji pružaju isti nivo detalja bez obzira na stepen uvećanja pod kojim se posmatraju [19]. Imaju rekurzivnu strukturu, jer se sastoje od umanjenih verzija samih sebe (poseduju semisličnost), ali su suviše nepravilni da bi se opisali nekom jednostavnom geometrijom.

Neke od poznatih fraktalnih funkcija prikazane se na Sl. 19 [19]:

- Kohova kriva,
- Zmajeva kriva (*dragon curve*) <https://youtu.be/m4-ILvsOFG0>,
- Mandelbrotove krive.



Sl. 19. Kohova kriva, zmajeva kriva i uvećan detalj Mandelbrotove krive

Program za crtanje Kohove krive prikazan je u praktičnom primeru na kraju ovog poglavlja.

6.5 Rekurzivne i iterativne verzije algoritama

Rekurzivna definicija funkcije faktorijel $n! = n \cdot (n-1)!$ jednostavno se realizuje kao rekurzivna funkcija u jeziku Python. Funkcija faktorijel se može definisati i iterativno, kao $n! = n \cdot (n-1) \cdot (n-2) \cdot \dots \cdot 2 \cdot 1$, odnosno, zbog komutacije operacije množenja, i kao $n! = 1 \cdot 2 \cdot \dots \cdot (n-2) \cdot (n-1) \cdot n$. Realizacija iterativnog algoritma za računanje funkcije faktorijel u jeziku Python je, npr.

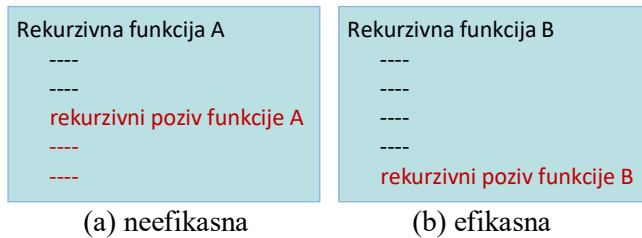
```
def faktorijel(n):
    f = 1
    for i in range(1, n+1):
        f = f * i
    return f
```

Izvršavanje funkcije je efikasnije, jer nije potrebno ažuriranje sistemskog steka na svakom koraku i ne zavisi više od ograničenja rekurzivne dubine.

```
>>> faktorijel(10)
3628800
>>> faktorijel(30)
265252859812191058636308480000000
>>>>
```

Iterativne verzije algoritama su efikasnije, jer nemaju potrebu za višestrukim čuvanjem vrednosti argumenata u sistemskom steku poziva, pošto se računanje obavlja u okviru jednog poziva funkcije. Rekurzivni algoritmi načelno se mogu pretvoriti u iterativne korišćenjem interne promenljive, koja predstavlja strukturu podataka tipa stek. Interna struktura tipa stek koristi se za pamćenje redosleda izvršavanja naredbi, odnosno odgovarajućih vrednosti lokalnih promenljivih. Algoritam se tada može realizovati pogodnom upotrebom petlji i strukture stek, umesto rekurzijom.

Funkcije su rekurzivne na kraju (*tail recursive*) kada se rekurzivno pozivanje vrši na kraju izvršavanja funkcije, nakon završetka svih ostalih operacija [1]. Funkcije koje nisu rekurzivne na kraju zahtevaju čuvanje obimnijeg rekurzivnog konteksta u steku i nisu efikasne. Na Sl. 20 prikazane su tipične strukture koda neefikasnih i efikasnih rekurzivnih funkcija [1].



Sl. 20. Tipične strukture rekurzivnih funkcija

Vidi se da funkcija A nakon povratka iz rekurzivnog poziva mora da izvrši još neke operacije pre završetka rada, pa je zbog toga neophodno čuvanje svih argumenata, međurezultata i povratnih informacija u sistemskom steku.

Primer *neefikasnih* rekurzivnih funkcija je standardna realizacija funkcije faktorijel:

```
# Funkcija koja nije rekurzivna na kraju
def faktorijel(n):
    if n == 0:
        return 1
    else:
        return n * faktorijel(n-1)
```

Primer *efikasnih* realizacija iste rekurzivne funkcije je funkcija faktorijel koja ima dodatni argument, u kome se rezultat formira u toku pozivanja [1]:

```
# Funkcija rekurzivna na kraju
def faktorijel(n, rezultat):
    if n == 0:
        return rezultat
    else:
        return faktorijel(n - 1, n*rezultat)
```

Funkcija se može koristiti kao `faktorijel(n,1)`, ali se može definisati pomoćna funkcija, kako bi se pozivala samo s jednim argumentom.

6.6 Primeri programa

Primeri programa koji koriste rekurzivne funkcije, u ovom poglavlju su programi za crtanje fraktalnih oblika: fraktalnog stabla, trougla Serpinskog i fraktalne pahuljice (*snowflake*).

6.6.1 Fraktalno stablo

Fraktalno stablo je rekurzivni geometrijski oblik, koji podseća na stablo drveta čije se grane dalje dele tako da detaljnije reprodukuju istu strukturu grananja. Osnovna struktura je linija, koja se na svom kraju deli na dve grane, svaka pod uglom od 45° u odnosu na stablo.



Svaka od grana osnovne strukture dalje se rekurzivno deli na isti način, samo se dužina osnovne strukture svaki put smanjuje.

Funkcija za crtanje fraktalnog stabla ima jedan parametar, fraktalnu dužinu stabla, koji predstavlja aktuelnu dužinu stabla u pikselima. Prilikom prvog poziva nacrtat se vertikalno stablo, a zatim se rekurzivno poziva ista procedura radi crtanja listova kao kraćih stabala, pod različitim uglovima u odnosu na prethodno nacrtano stablo.

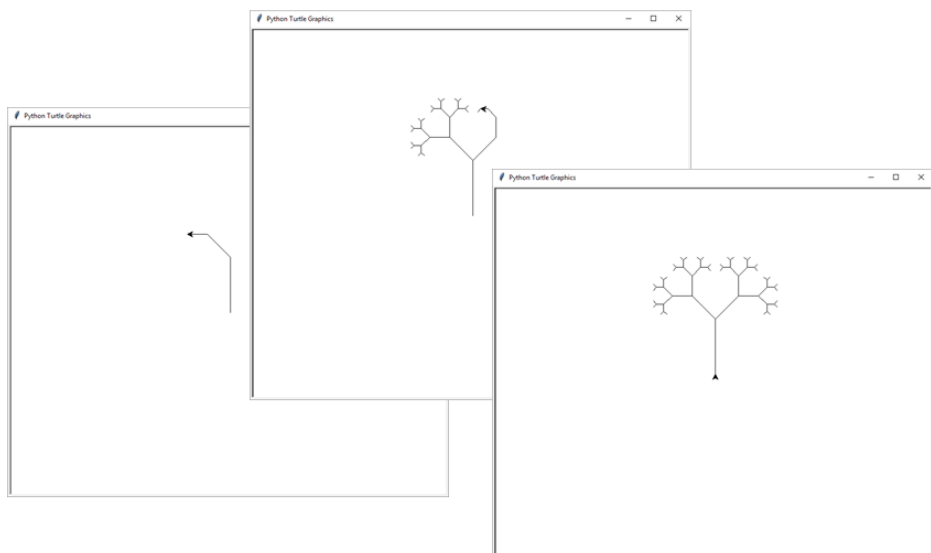
Program za crtanje fraktalnog stabla zadane veličine je:

```
import turtle

def fraktalnoStablo(f_duzina, min_duzina=10):
    # Funkcija rekurzivno crta stablo s dva lista
    turtle.forward(f_duzina)
    if f_duzina > min_duzina:
        turtle.left(45)
        fraktalnoStablo(0.6*f_duzina, min_duzina)
        turtle.right(90)
        fraktalnoStablo(0.6*f_duzina, min_duzina)
        turtle.left(45)
    turtle.back(f_duzina)

turtle.left(90)
fraktalnoStablo(100)
```

Izvršavanje programa za crtanje fraktalnog stabla dimenzija 100 piksela prikazano je na Sl. 21

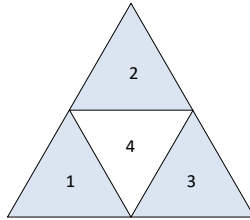


Sl. 21. Koraci iscrtavanja fraktalnog stabla

6.6.2 Trougao Serpinskog

Trougao Serpinskog⁴, prikazan na Sl. 22, primer je fraktalnog oblika, koji se dobija rekurzivnom deobom stranica trougla sledećim postupkom [1], [2]:

1. Nacrta se osnovni trougao.
2. Svaki veliki trougao podeli se na četiri manja trougla povezivanjem sredina njegovih stranica.



3. Ista procedura rekurzivno se primenjuje na tri nova manja trougla, koji se nalaze u uglovima većeg, dok se srednji novi trougao dalje ne deli.

Dubina rekurzije određuje ukupni broj trouglova koji se crta. Svaki rekurzivni poziv umanjuje promenljivu dubina za jedan, a kad promenljiva poprimi vrednost nula, postupak rekurzivnog pozivanja se prekida.

Program za crtanje trougla Serpinskog, koji je zadan koordinatama uglova osnovnog trougla i brojem nivoa rekurzivnih podela je [2]:

```
import turtle

# Trougao Serpinskog zadan koordinatama uglova
def trougao(uglovi):
    turtle.color('violet')
    turtle.up()
    turtle.goto(uglovi[0][0],uglovi[0][1])
    turtle.down()
    turtle.goto(uglovi[1][0],uglovi[1][1])
    turtle.goto(uglovi[2][0],uglovi[2][1])
    turtle.goto(uglovi[0][0],uglovi[0][1])

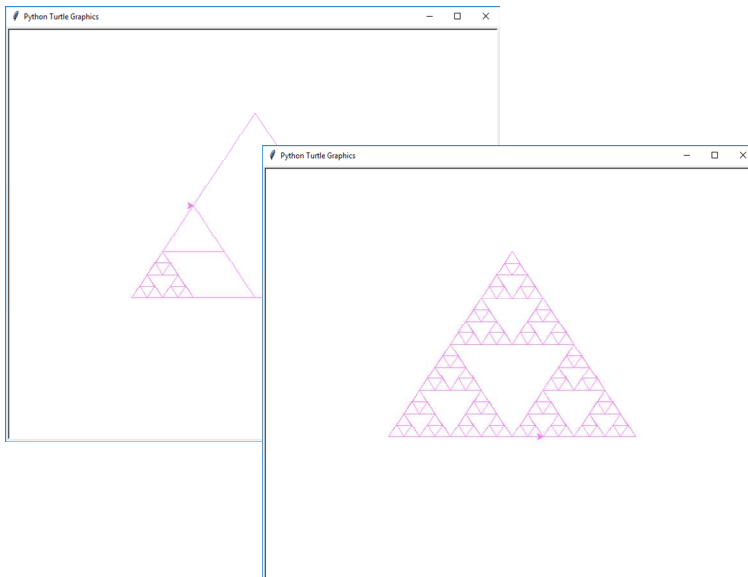
# Vraća koordinate sredine stranice između dva ugla
def sredina(u1,u2):
    return [(u1[0]+u2[0])/2, (u1[1]+u2[1])/2]
```

⁴ Waclaw Franciszek Sierpiński (1882-1969), poljski matematičar, dao izuzetan doprinos teoriji skupova, teoriji brojeva, teoriji funkcija i topologiji. Po njemu su nazvana tri poznata fraktala (trougao, tepih i kriva Serpinskog) i brojevi Serpinskog.

```
# Rekurzivni prikaz osnovnog i tri manja trougla
def serpinski(uglovi, dubina):
    trougao(uglovi)
    if dubina > 0:
        serpinski([uglovi[0],
                    sredina(uglovi[0], uglovi[1]),
                    sredina(uglovi[0], uglovi[2])],
                    dubina-1)
        serpinski([uglovi[1],
                    sredina(uglovi[0], uglovi[1]),
                    sredina(uglovi[1], uglovi[2])],
                    dubina-1)
        serpinski([uglovi[2],
                    sredina(uglovi[2], uglovi[1]),
                    sredina(uglovi[0], uglovi[2])],
                    dubina-1)

uglovi = [[-150, -50], [0, 200], [150, -50]]
serpinski(uglovi, 4)
```

Rezultat izvršavanja programa za zadane koordinate osnovnog trougla i tri nivoa rekurzivne podele prikazan je na Sl. 22.

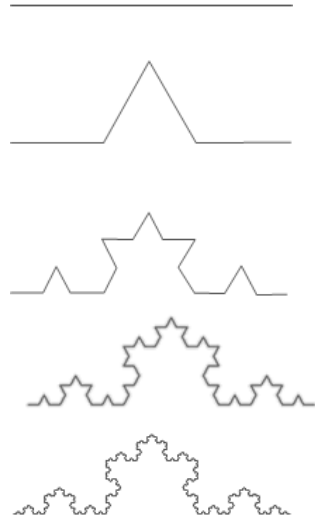


Sl. 22. Koraci iscrtavanja trougla Serpinskog

6.6.3 Pahuljica

Osnova fraktalnog oblika pahuljice (*snowflake*) prikazanoj na Sl. 24 je Kohova kriva sa Sl. 23, koja se dobija rekurzivno, sledećim postupkom [1], [2]:

1. Osnova je duž iscrtana u određenom položaju, npr. horizontalno.
2. Duž se zameni oblikom na slici, jednakostraničnim trouglom, čije su stranice pod uglom od 60° .
3. Svaka od četiri duži prve iteracije zameni se istim oblikom.
4. Svaka od 16 duži prethodne iteracije zameni se istim oblikom.
5. Postupak se nastavlja do zadane rekurzivne dubine.



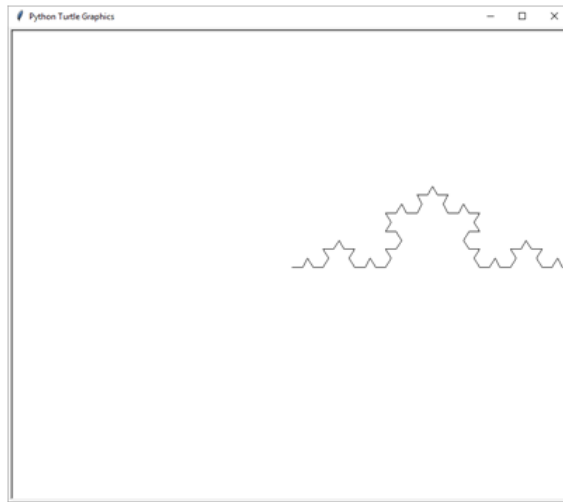
Program za crtanje Kohove krive u jeziku Python, relizovan pomoću funkcija iz modula *turtle* grafike, veoma je kratak:

```
import turtle

# Kohova kriva
def koch(f_duzina, dubina):
    if dubina > 0:
        for ugao in [60, -120, 60, 0]:
            koch(f_duzina/3, dubina-1)
            turtle.left(ugao)
        else:
            turtle.forward(f_duzina)
# Test
koch(400, 3)
```

Rekurzivna funkcija `Koch()` poziva samu sebe za iscrtavanje tri duži sve manje dužine pod tri različita ugla, sve dok vrednost promenljive `dubina` ne poprimi vrednost nula. Duži iste dužine iscrtavaju se naredbom `forward(f_duzina)` samo na konačnoj rekurzivnoj dubini.

Kohova kriva rekurzivne dubine 3, koja se dobije nakon tri rekurzivna koraka, prikazana je na Sl. 23.



Sl. 23. Kohova kriva rekurzivne dubine 3

Pahuljica se dobija tako što se Kohova kriva crta više puta, svaki put pod različitim uglom. Kompletan program za crtanje pahuljice u jeziku Python je:

```
import turtle

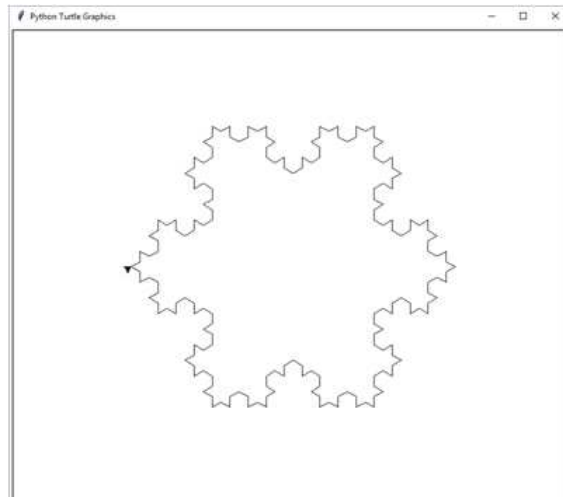
# Kohova kriva
def koch(f_duzina, dubina):
    if dubina > 0:
        for ugao in [60, -120, 60, 0]:
            koch(f_duzina/3, dubina-1)
            turtle.left(ugao)
    else:
        turtle.forward(f_duzina)

# Test
velicina = 400
dubina = 3
```

```
# Centriranje pahuljice
turtle.penup()
turtle.backward(velicina/1.73)
turtle.left(30)
turtle.pendown()

# Tri Kohove krive
for i in range(3):
    koch(velicina, dubina)
    turtle.right(120)
```

Kompletna pahuljica, prikazana je na Sl. 24, dobijena je crtanjem tri Kohove krive rekursivne dubine 3, s tim da je prva pod uglom od 30° u odnosu na prethodni primer, a naredne su međusobno rotirane za 120° .



Sl. 24. Pahuljica na osnovu Kohove krive rekursivne dubine 3

Pitanja za ponavljanje

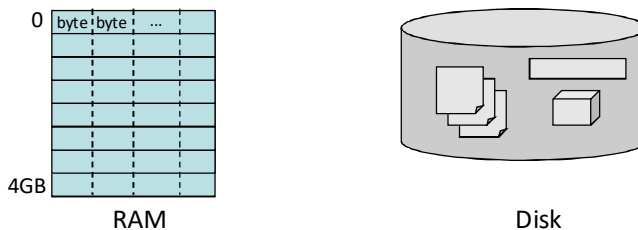
1. Objasnite pojam rekurzije na primerima iz prirode i svakodnevnog života.
2. Po čemu se rekurzivne funkcije razlikuju od ostalih korisničkih funkcija?
3. Na koji način se sprečava beskonačna rekurzija u jeziku Python?
4. Objasnite ulogu sistemskog steka poziva (*call stack*) prilikom izvršavanja rekurzivne funkcije.
5. Šta je Fibonačijev broj?
6. Objasnite osnovne funkcije za vektorsku grafiku iz Python modula *turtle*.
7. Kako se definiše boja linije pomoću funkcija iz Python modula *turtle*.
8. Koja je osnovna razlika između rekurzivne i iterativne verzije algoritma?
9. Koje rekurzivne funkcije su efikasne?
10. Šta su fraktali? Navedite primere fraktalnih funkcija.

7 Osnovne strukture podataka u jeziku Python

1. Uvod: nizovi u jeziku Python
2. Stringovi
3. Liste
4. N-torke
5. Osnovne operacije nad nizovima (sekvencama)
6. Primeri programa

Struktura podataka je kolekcija više delova podataka strukturiranih na takav način da im se može efikasno pristupati.

Zbog različitog načina pristupa podacima, razlikuju se strukture podataka u memoriji s direktnim pristupom (operativna memorija, RAM) i strukture podataka na drugim medijima, npr. na disku, Sl. 25.



Sl. 25. Strukture podataka u operativnoj memoriji i na disku

U ovoj temi razmatraju se samo strukture podataka u memoriji i to najjednostavnija organizacija podataka u obliku *linearne sekvence* ili *niza*.

7.1 Uvod: nizovi u jeziku Python

U jeziku Python, nizovi su linearno uređene sekvence podataka, čijim elementima se pristupa pomoću numeričke vrednosti ili *indeksa*.

Nozovi (sekvence) u jeziku Python su:

- *stringovi* - nizovi karaktera
- *liste* - nizovi elemenata bilo kog tipa
- *n-torke* - nizovi elemenata bilo kog tipa koji se ne mogu menjati

Indeksi, odnosno položaj elemenata sekvence, računaju se *unapred* od početne pozicije 0 ili *unazad*, od poslednje pozicije niza koja je -1.

7.2 Stringovi

String je niz znakova u navodnicima ('...' ili "...") , npr.

```
'rekao je "naravno"'
```

```
"rekao je 'naravno'"
```

Znak '\' se može koristiti za definisanje specijalnih znakova, između ostalog i simbola navodnika kao dela stringa:

```
>>> 'rekao je \' naravno\''
```

```
"rekao je 'naravno'"
```

Elementi stringa se indeksiraju od nule, kao na slici:

'	P	y	t	h	o	n	'
	0	1	2	3	4	5	
	-6	-5	-4	-3	-2	-1	

U jeziku Python se pristup elementima stringa u petlji i ispis svih znakova, po jedan u redu, može izvršiti jednom naredbom ponavljanja, koja blok naredbi izvršava za svaki element niza, u ovom slučaju stringa:

```
>>> for ch in 'Monty Python':
        print(ch)
```

```
M
```

```
o
```

```
n
```

```
t
```

```
y
```

```
P
```

```
y
```

```
t
```

```
h
```

```
o
```

```
n
```

```
>>>
```

Ponekad je pogodnije umesto samih elemenata niza u petlji koristiti njihove pozicije ili indekse, npr.

```
>>> s = 'Monty Python'
>>> for i in range(len(s)):
        print(s[i])

M
o
n
t
y

P
y
t
h
o
n
>>>
```

Treba naglasiti da se elementi strukture string ne mogu menjati, npr. pokušaj promene nekog znaka u stringu proizvodi poruku o grešci:

```
>>> s = "Pithon"
>>> s[1] = "y"

Traceback (most recent call last):
  File "<pyshell#35>", line 1, in <module>
    s[1] = "y"
TypeError: 'str' object does not support item
assignment
```

Za promenu sadržaja stringa potrebno je kreirati novi string s izmenjenim sadržajem, koji se zatim može dodeliti nekoj promenljivoj, npr.

```
>>> s = s[0] + "y" + s[2:]
>>> s
'Python'
```

Novi string 'Python' dobijen je konkatencijom prvog znaka stringa s (pozicija 0), znaka 'y' i dela stringa s koji počinje od trećeg znaka (pozicija 2) do kraja stringa, pošto pozicija poslednjeg znaka nije navedena.

Osnovne funkcije i operatori povezani sa strukturom *string* su:

- `len(s)` - dužina stringa *s*,
- `str(x)` - konverzija drugih tipova podataka *x* u tip string,
- `repr(x)` - vraća string koji je čitljiva reprezentacija vrednosti *x*,
- `s[i]` - pristup elementu *i* stringa *s*,
- `s[i:j:k]` - izdvajanje dela stringa *s* od pozicije *i* do *j*-1, s korakom *k*,
- `+` - operator spajanja (konkatenacije) stringova,
- `*` - operator multiplikacije (umnožavanja) stringova.

Funkcije za konverziju zapisa pojedinačnih znakova su:

- `ord(c)` - vraća numerički kod znaka *c* u ASCII tabeli kodova,
- `chr(i)` - vraća znak s numeričkim kodom *i* iz ASCII tabele kodova.

Skraćenica **ASCII** (*American Standard Code for Information Interchange*) označava standard binarnog zapisa različitih znakova koji se koriste u računarstvu i telekomunikacijama. Ovaj sistem kodiranja razvijen je još 1963. godine za potrebe tadašnjih telekomunikacija, koje su koristile analogne linije i teleprintere [19]. Standardni skup od 128 znakova kodira se sa 7 bita, dok je osmi bit u komunikacijama služio za kontrolu pariteta, odnosno ispravnog prenosa pojedinačnih znakova, Sl. 26.

Dec	Oct	Hex	Znak	Dec	Oct	Hex	Znak	Dec	Oct	Hex	Znak	Dec	Oct	Hex	Znak
0	0	00	NUL (null)	32	40	20	(space)	64	100	40	@	96	140	60	`
1	1	01	SOH (start of header)	33	41	21	!	65	101	41	A	97	141	61	a
2	2	02	STX (start of text)	34	42	22	"	66	102	42	B	98	142	62	b
3	3	03	ETX (end of text)	35	43	23	#	67	103	43	C	99	143	63	c
4	4	04	EOT (end of transmission)	36	44	24	\$	68	104	44	D	100	144	64	d
5	5	05	ENQ (enquiry)	37	45	25	%	69	105	45	E	101	145	65	e
6	6	06	ACK (acknowledge)	38	46	26	&	70	106	46	F	102	146	66	f
7	7	07	BEL (bell)	39	47	27	'	71	107	47	G	103	147	67	g
8	10	08	BS (backspace)	40	50	28	(	72	110	48	H	104	150	68	h
9	11	09	HT (horizontal tab)	41	51	29	)	73	111	49	I	105	151	69	i
10	12	0a	LF (line feed - new line)	42	52	2a	*	74	112	4a	J	106	152	6a	j
11	13	0b	VT (vertical tab)	43	53	2b	+	75	113	4b	K	107	153	6b	k
12	14	0c	FF (form feed - new page)	44	54	2c	,	76	114	4c	L	108	154	6c	l
13	15	0d	CR (carriage return)	45	55	2d	-	77	115	4d	M	109	155	6d	m
14	16	0e	SO (shift out)	46	56	2e	.	78	116	4e	N	110	156	6e	n
15	17	0f	SI (shift in)	47	57	2f	/	79	117	4f	O	111	157	6f	o
16	20	10	DLE (data link escape)	48	60	30	0	80	120	50	P	112	160	70	p
17	21	11	DC1 (device control 1)	49	61	31	1	81	121	51	Q	113	161	71	q
18	22	12	DC2 (device control 2)	50	62	32	2	82	122	52	R	114	162	72	r
19	23	13	DC3 (device control 3)	51	63	33	3	83	123	53	S	115	163	73	s
20	24	14	DC4 (device control 4)	52	64	34	4	84	124	54	T	116	164	74	t
21	25	15	NAK (negative acknowledge)	53	65	35	5	85	125	55	U	117	165	75	u
22	26	16	SYN (synchronous idle)	54	66	36	6	86	126	56	V	118	166	76	v
23	27	17	ETB (end of transmission block)	55	67	37	7	87	127	57	W	119	167	77	w
24	30	18	CAN (cancel)	56	70	38	8	88	130	58	X	120	170	78	x
25	31	19	EM (end of medium)	57	71	39	9	89	131	59	Y	121	171	79	y
26	32	1a	SUB (substitute)	58	72	3a	:	90	132	5a	Z	122	172	7a	z
27	33	1b	ESC (escape)	59	73	3b	;	91	133	5b	[	123	173	7b	{
28	34	1c	FS (file separator)	60	74	3c	<	92	134	5c	\	124	174	7c	
29	35	1d	GS (group separator)	61	75	3d	=	93	135	5d	]	125	175	7d	}
30	36	1e	RS (record separator)	62	76	3e	>	94	136	5e	^	126	176	7e	~
31	37	1f	US (unit separator)	63	77	3f	?	95	137	5f	_	127	177	7f	DEL (delete)

Sl. 26. Tabela kodova znakova po ASCII standardu

Proizvođači računara vremenom su ASCII standard usvojili za prikaz teksta u memoriji računara, koristeći pri tome svih 8 bita za prikaz ukupno 256 znakova. Iako međusobno različiti, ovi sistemi su omogućili predstavljanje šireg skupa znakova, koji je uključio i slova određenih nacionalnih pisama. Način kodiranja proširenog ASCII sistema definisan je standardom ISO/IEC 8859, dok je npr. kompanija Microsoft koristila svoj raspored Windows-1252.

Osmobitni kod za predstavljanje znakova vremenom je zamenjen sistemom sa 8, 16 ili 32 bita, tzv. sistemom Unicode, koji može da prikaže znakove nacionalnih pisama koja imaju desetine hiljada znakova. Pojedinačni znakovi u sistemu Unicode predstavljaju se različitim brojem bita (UTF-8, UTF-16 ili UTF-32), zavisno od skupa znakova koji je potrebno predstaviti. Stari ASCII standard čini prvih 128 znakova novog sistema, tako da nije potrebna promena starijih programa koji ne koriste celi Unicode. Sledećih 128 znakova sistema Unicode raspoređeno je prema postojećem standardu ISO-8859-1 (Latin 1).

7.3 Liste

Liste su sekvence bilo kakvih vrednosti, uključujući rezultate evaluacije izraza i druge liste.

Lista u jeziku Python najjednostavnije se definiše kao niz vrednosti u uglastim zagrada. Može da sadrži vrednosti različitog tipa, npr. lista koja sadrži samo numeričke vrednosti je:

```
>>> a = [10, 20, 30, 40]
>>> a
[10, 20, 30, 40]
>>>
```

Primer liste koja sadrži samo nenumeričke vrednosti je:

```
>>> b = ['kruška', 'jabuka', 'banana', 'limun']
>>> b
['kruška', 'jabuka', 'banana', 'limun']
>>>
```

Primer liste koja istovremeno sadrži vrednosti različitog tipa je:

```
>>> c = ['kruška', 1, 'jabuka', 2.0]
>>> c
['kruška', 1, 'jabuka', 2.0]
>>>
```

7.3.1 Operacije nad listama

U jeziku Python predviđene su sledeće operacije nad strukturom liste, koje se uprogramima koriste kao *metodi* objekata tipa liste:

<code><lista>.append(x)</code>	Dodaje novi element na kraj liste - ekvivalentno naredbi <code>a[len(a):] = [x]</code>
<code><lista>.extend(L)</code>	Proširuje listu dodavanjem na kraj svih elemenata zadane liste <i>L</i> , ekvivalent <code>a[len(a):] = L</code>
<code><lista>.insert(i, x)</code>	Umeće novi element <i>x</i> u listu od zadane pozicije <i>i</i> . Argument <i>i</i> je indeks elementa <i>ispred</i> kog treba umetnuti element <i>x</i> , npr. <code>a.insert(0, x)</code> dodaje element na čelo liste, a <code>a.insert(len(a), x)</code> ekvivalentno je <code>a.append(x)</code>
<code><lista>.remove(x)</code>	Uklanja prvi element liste čija je vrednost jednaka <i>x</i> . Ukoliko u listi ne postoji takav element, dobija se poruka o grešci
<code><lista>.pop([i])</code>	Uklanja element na zadanoj poziciji <i>i</i> liste i vraća njegovu vrednost. Ako indeks <i>i</i> nije naveden, <code>a.pop()</code> uklanja <i>poslednji</i> element u listi i vraća njegovu vrednost
<code><lista>.clear()</code>	Uklanja sve elemente liste, ekvivalent <code>del a[:]</code>
<code><lista>.index(x)</code>	Vraća indeks liste prvog elementa čija je vrednost <i>x</i> . Ukoliko ne postoji, dobija se poruka o grešci
<code><lista>.count(x)</code>	Vraća broj pojavljivanja vrednosti <i>x</i> u listi
<code><lista>.sort(key=None, reverse=False)</code>	Sortira elemente liste u zadanom redosledu, definisanom argumentima <i>key</i> i <i>reverse</i>
<code><lista>.reverse()</code>	Postavlja elemente liste u obrnutom redosledu
<code><lista>.copy()</code>	Vraća kopiju cele liste - ekvivalentno <code>a[:]</code>

7.3.2 Iteracija nad listama

Pristup svim elementima liste pojedinačno, može se realizovati korišćenjem naredbi ponavljanja, npr. za listu:

```
lista = [10, 20, 30, 40, 60, 90]
```

pristup svakom elementu liste, npr. radi njegovog prikaza, može se realizovati:

a) pomoću petlje `for`, u okviru jedne naredbe:

```
for n in lista:
    print(n)
```

b) pomoću petlje `while`, nešto složenijim programskim segmentom:

```
n = 0
while n < len(lista):
    print(lista[n])
    n = n + 1
```

7.3.3 Linearno pretraživanje liste

Ponekad je potrebno pronaći neku određenu vrednost u nizu vrednosti, što se može realizovati *linearnim pretraživanjem*. Počev od prvog elementa, redom se upoređuju svi elementi liste s traženom vrednošću, sve dok se vrednost ne pronađe ili se dođe do kraja liste.

Primer realizacije linearnog pretraživanja niza u jeziku Python pomoću petlje *while* je:

```
lista = [10, 30, 90, 40, 50, 80]

podatak = 40
pronadjen = False

i = 0
while i < len(lista) and not pronadjen:
    if lista[i] == podatak:
        pronadjen = True
    else:
        i = i + 1
if pronadjen:
    print("Podatak je pronađen")
else:
    print("Podatak nije pronađen")
```

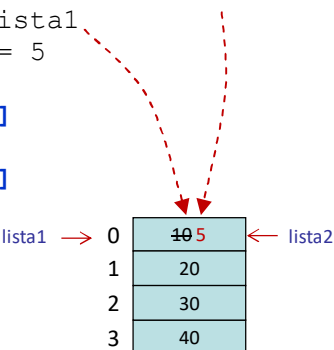
7.3.4 Dodela vrednosti i kopiranje lista

Dodela vrednosti jedne liste drugoj ne stvara kopiju samih vrednosti, tako da se promena elemenata jedne liste odnosi i na drugu, Sl. 27.

Nakon dodele vrednosti promenljivoj `lista2` obe promenljive pokazuju na istu strukturu podataka.

Zasebna kopija strukture podataka koja bi se nezavisno koristila pod nazivom `lista2` dobija se naredbom:

```
>>> lista2 = lista1[:]
>>> lista1 = [10, 20, 30, 40]
>>> lista2 = lista1
>>> lista1[0] = 5
>>> lista1
[5, 20, 30, 40]
>>> lista2
[5, 20, 30, 40]
```



lista1 →	0	5	← lista2
	1	20	
	2	30	
	3	40	

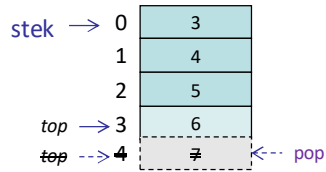
Sl. 27. Prikaz više referenci na jednu strukturu liste

7.3.5 Lista kao struktura stek

Lista se jednostavno koristi kao struktura stek (*stack*), u kojoj se pristupa samo jednom elementu liste koji se nalazi na kraju (vrh steka, *top*). Izmene strukture steka vrše se samo dodavanjem i brisanjem elementa na vrhu steka, za šta se mogu koristiti metodi `append()` i `pop()`.

Sledeće naredbe kreiraju strukturu stek, dosaju dva elementa, a zatim jedna briše i daju rezultat prikazan na Sl. 28:

```
>>> stek = [3, 4, 5]
>>> stek.append(6)
>>> stek.append(7)
>>> stek
[3, 4, 5, 6, 7]
>>> stek.pop()
7
>>> stek
[3, 4, 5, 6]
```



Sl. 28. Prikaz operacija nad strukturom stek

7.3.6 Skraćeno generisanje lista

Funkcija `range()` omogućava generisanje sekvenci celih brojeva s fiksnim korakom. Skraćeno generisanje lista (*list comprehension*) u jeziku Python omogućava generisanje raznovrsnijih sekvenci.

Skraćeni zapis izraza koji generiše elemente liste sastoji se od uglastih zagrada u kojima je izraz za kojim sledi `for` klauzula (iza koje može biti još `for` ili `if` klauzula):

```
[<izraz> for ... in ...]
```

Rezultat evaluacije izraza je lista koja nastaje evaluacijom izraza na svakom koraku izvršavanja petlje `for`, npr.

```
>>> [x**3 for x in [1, 2, 3]]
[1, 8, 27]
```

Izrazi se mogu koristiti za skraćeni zapis lista u naredbama jezika Python.

Primeri različitih izraza za skraćeno generisanje lista su:

```
>>> [x**3 for x in range(5)]
[0, 1, 8, 27, 64]
>>> lista = [-1, 1, -2, 2, -3, 3, -4, 4]
>>> [x for x in lista if x >= 0]
[1, 2, 3, 4]
>>> #Pronalaženje svih pojava samoglasnika u tekstu:
>>> samoglasnici = ('a', 'e', 'i', 'o', 'u')
>>> tekst = 'Programiranje'
>>> [ch for ch in tekst if ch in samoglasnici]
['o', 'a', 'i', 'a', 'e']
>>> #Unija elemenata dve liste:
>>> [(x, y) for x in [1, 2, 3] for y in [3, 1, 4] if x != y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

Naredna tema obrađuje rekurzivne strukture: listu kao element liste, koja se može upotrebiti za predstavljanje matrica, npr.

```
>>> matrica = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
               [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

7.4 N-torke

N-torke su nepromenljive liste. Zadaju se u običnim zagradama, a elementi se ne mogu menjati, kao ni sama struktura (nema odgovarajućih metoda), npr.

```
>>> a = (1, 2, 3, 4)
>>> a
(1, 2, 3, 4)
>>> a[1]=1
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    a[1]=1
TypeError: 'tuple' object does not support item
assignment
```

7.5 Osnovne operacije nad nizovima (sekvencama)

Osnovne operacije nad strukturama tipa sekvence, stringovima, listama i n-torkama u jeziku Python su:

<code>len(s)</code>	Dužina sekvence <i>s</i> (<i>length</i>)
<code>s[i]</code>	Pristup elementu <i>i</i> sekvence (<i>select</i>)
<code>s[i:j:k]</code>	Izdvajanje dela sekvence <i>i..j-1</i> , s korakom <i>k</i> (<i>slice</i>)
<code>s.count()</code>	Brojanje elemenata sekvence (<i>count</i>)
<code>s.index(x)</code>	Pronalaženje pozicije elementa <i>x</i> (<i>index</i>)
<code>v in s</code>	Provera pripadnosti elementa <i>v</i> (<i>membership</i>)
<code>s + w</code>	Spajanje sekvenci <i>s</i> i <i>w</i> (<i>concatenation</i>)
<code>min(s)</code>	Najmanja vrednost u sekvenci (<i>minimum value</i>)
<code>max(s)</code>	Najveća vrednost u sekvenci (<i>maximum value</i>)
<code>sum(s)</code>	Zbir svih elemenata sekvence (<i>sum</i>)

7.5.1 Sekvence kao argumenti funkcije

Sekvence se mogu prenositi kao argumenti funkcije. Pri tome se u stek poziva ne kopira cela struktura, već samo pokazivač na originalnu strukturu podataka. Na taj način se sadržaj liste može *menjati* u samoj funkciji, jer funkcija ima pristup elementima strukture.

Stringovi i n-torke su *nepromenljivi* objekti, pa im se iz funkcije može pristupati, ali se elementi ne mogu menjati. U sledećoj sekvenci definiše se i pokreće funkcija koja menja vrednost svog aktuelnog argumenta:

```
>>> def fun(s):
        s[0]= "*" #nova vrednost prvog elementa sekvence
        return

>>> a = ["a", "b", "c", "d"]
>>> print(fun(a),a)
None ['*', 'b', 'c', 'd']
```

Funkcija može da promeni vrednosti elementa liste, ali ne i vrednosti elemenata n-torke i stringa, kao što se vidi u sledećim primerima:

```
>>> a = ("a", "b", "c", "d")
>>> print(fun(a),a)
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    print(fun(a),a)
  File "<pyshell#1>", line 2, in fun
    s[0] = "*"
TypeError: 'tuple' object does not support item
assignment
>>> a = "abcd"
>>> print(fun(a),a)
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    print(fun(a),a)
  File "<pyshell#0>", line 2, in fun
    s[0]= "*" #nova vrednost prvog elementa sekvence
TypeError: 'str' object does not support item
assignment
```

Aktuelni argument funkcije može biti struktura zadana kao eksplicitni niz vrednosti, npr.

```
fun([10,20,30], ... )
```

Takva sekvenca je anonimna, kreira se prilikom poziva funkcije i nestaje nakon njenog završetka, jer ne postoji promenljiva koja bi ovu vrednost pamtila.

7.5.2 Sekvence i podrazumevane (default) vrednosti argumenata

Sekvence koje se prenose kao argumenti funkcija mogu imati podrazumevajuće vrednosti. Za razliku od vrednosti osnovnih tipova, dodela definisane podrazumevane vrednosti izostavljenom argumentu vrši se *samo kod prvog poziva*, npr.

```
>>> def dodaj(x, lista=[]):
        if x not in lista:
            lista.append(x)

>>>> l1 = dodaj(1) # broj se dodaje u default listu []
>>>> print(l1)
[1]
>>>> l2 = dodaj(2) # broj se dodaje u listu [1]
>>>> print(l2)
[1, 2]
```

7.5.3 Sekvence kao rezultati funkcija

Funkcija može da kao rezultat vrati više vrednosti

```
return <r1>, <r2>, ...
```

Vrednosti mogu biti osnovnog tipa ili neka od struktura podataka. Npr. funkcija koja vraća listu u kojoj su elementi u obrnutom redosledu od elemenata zadane liste:

```
>>> def obrnuto(lista):
        rezultat = [] # kreiranje nove liste
        for element in lista:
            rezultat.insert(0, element)
        return rezultat # rezultat je nova lista

>>> print(obrnuto([1,2,3,4,5])) # prikaz el. nove liste
[5, 4, 3, 2, 1]
```

7.6 Primeri programa

Primena osnovnih struktura podataka, stringova, lista i n-torki, prikazaće se na primerima realizacije programa za upotrebu uprošćenog kineskog astrološkog kalendara i pronalaženje liste prostih brojeva manjih od nekog zadanog broja (tzv. Eratostenovo sito).

7.6.1 Kineski astrološki kalendar

Kineski astrološki kalendar ima ciklus od 12 godina, koje su nazvane po životinjama, akterima jedne legende, Sl. 29 [28].



Sl. 29. Kineski zodijak

Algoritam programa je jednostavan: na osnovu godine rođenja osobe, određuje se njen astrološki znak prema indeksu, koji se računa na osnovu godine rođenja kao ostatak od deljenja $(godina - 1900) | 12$. Zatim se, prema prema indeksu iz tabele, biraju znak i lične karakteristike osobe, prikazane u tabeli:

1	pacov	<i>iskren, vredan, osetljiv, intelektualac, druželjubiv</i>
2	vo	<i>pouzdan, metodičan, skroman, rođeni vođa, strpljiv</i>
3	tigar	<i>nepredvidljiv, buntovan, strastven, smeo, impulsivan</i>
4	zec	<i>dobar drug, dobar, tih, oprezan, umetnički</i>
5	zmaj	<i>jak, samouveren, ponosan, odlučan, odan</i>
6	zmija	<i>dubokouman, kreativan, odgovoran, miran, usmeren</i>
7	konj	<i>veseo, dovitljiv, perceptivan, pričljiv, otvoren</i>
8	koza	<i>iskren, simpatičan, stidljiv, velikodušan, pažljiv</i>
9	majmun	<i>motivator, radoznao, fleksibilan, inovativan, rešava probleme</i>
10	petao	<i>organizovan, samouveren, odlučan, perfekcionista, revnosta</i>
11	pas	<i>pošten, nepretenciozan, idealista, moralista, ležeran</i>
12	svinja	<i>miroljubiv, vredan, poverljiv, pun razumevanja, zamišljen</i>

Za prikaz podataka iz tabele u programu koriste se dve n-torke i dvanaest imenovanih konstanti s tekstom ličnih osobina pojedinih znakova. Jedna n-torka stringova je spisak naziva životinja, a druga spisak imenovanih konstanti s opisom osobina znakova zodijskih.

```
import datetime
```

```
# Inicijalizacija struktura podataka
```

n-torka stringova koji predstavljaju nazive životinja

```
astroloske_zivotinje = ('pacov', 'vo', 'tigar', 'zec', 'zmaj', 'zmija', 'konj', 'koza', 'majmun', 'petao', 'pas', 'svinja')
```

```
pacov = 'iskren, vredan, osetljiv, intelektualac, druželjubiv'
vo = 'pouzdan, metodičan, skroman, rođeni vođa, strpljiv'
tigar = 'nepredvidljiv, buntovan, strastven, smeo, impulsivan'
zec = 'dobar drug, dobar, tih, oprezan, umetnički'
zmaj = 'jak, samouveren, ponosan, odlučan, odan'
zmija = 'dubokouman, kreativan, odgovoran, miran, usmeren'
konj = 'veseo, dovrtljiv, perceptivan, pričljiv, otvoren'
koza = 'iskren, simpatičan, stidljiv, velikodušan, pažljiv'
majmun = 'motivator, radoznao, fleksibilan, inovativan, '+\
'rešava probleme'
petao = 'organizovan, samouveren, odlučan, perfekcionista, '+\
'revnostan'
pas = 'pošten, nepretenciozan, idealista, moralista, '+\
'ležeran'
svinja = 'miroljubiv, vredan, poverljiv, pun razumevanja, '+\
'zamišljen'
```

n-torka sadrži sekvencu prethodno definisanih stringova

```
osobine = (pacov, vo, tigar, zec, zmaj, zmija, konj, koza, majmun, petao, pas, svinja)
```

```
# Određivanje znaka i ispis rezultata
```

```
# Pozdravna poruka
```

```
print('Program na osnovu godine rođenja pronalazi znak osobe',
      'iz kineskog horoskopa i prikazuje njene lične sobine\n')
```

```
# Pronalaženje tekuće godine
```

```
tekuca_godina = datetime.date.today().year
```

upotreba metoda *today()* za dobijanje tekućeg datuma i izdvajanje tekuće godine

```
kraj = False
```

```
while not kraj: ← petlja while se izvršava sve dok se ne promeni vrednost promenljive kraj
```

```
# Unos datuma rođenja
```

```
godina_rodjenja = 0
```

```
while godina_rodjenja < 1900 or \ ← petlja proverava ispravnost godine i ponavlja unos sve dok se ne unese ispravna godina
      godina_rodjenja > tekuca_godina:
    godina_rodjenja = \
        int(input('Unesite datum rođenja osobe (gggg): '))
```

```
# Prikaz rezultata
```

broj ciklusa je indeks u strukturama podataka

```
broj_ciklusa = (godina_rodjenja - 1900) % 12
```

```
print('Vaš znak u kineskom horoskopu je', \
      astroloske_zivotinje[broj_ciklusa], '\n')
```

```
print('Vaše lične osobine su ...')
```

```
print(osobine [broj_ciklusa])
```

ispis naziva životinje na osnovu indeksa *broj_ciklusa*

```
# Uslovni kraj programa
```

ispis ličnih osobina na osnovu indeksa *broj_ciklusa*

```

odgovor = \
    input('\nDa li želite znak još neke osobe? (d/n): ')
if odgovor == 'n':
    kraj = True

```

Izvršavanje programa prikazano je na dva primera:

Program na osnovu godine rođenja pronalazi znak osobe iz kineskog horoskopa i prikazuje njene lične osobine

Unesite datum rođenja osobe (gggg): 1994

Vaš znak u kineskom horoskopu je pas

Vaše lične osobine su ...

pošten, nepretenciozan, idealista, moralista, ležeran

Da li želite znak još neke osobe? (d/n): d

Unesite datum rođenja osobe (gggg): 1996

Vaš znak u kineskom horoskopu je pacov

Vaše lične osobine su ...

iskren, vredan, osetljiv, intelektualac, druželjubiv

Da li želite znak još neke osobe? (d/n): n

>>>

Program proverava unesenu godinu rođenja, koja mora biti između 1900. godine i tekuće godine, koju program pronalazi pomoću funkcije `today()` iz modula `datetime`.

7.6.2 Eratostenovo sito

Složeni brojevi se mogu predstaviti kao proizvod drugih brojeva, kao što je npr. broj $12=2\cdot2\cdot3$, dok su prosti brojevi deljivi samo s njima samima i brojem 1. Proste brojeve su poznavali i proučavali još antički matematičari, npr. Euklid je dokazao da je broj prostih brojeva beskonačan.

Značaj prostih brojeva naglo je porastao u savremeno doba, jer se na njima zasnivaju važne računarske aplikacije, kao npr. kriptografski algoritmi, koji omogućavaju zaštitu informacija u javnoj računarskoj mreži.

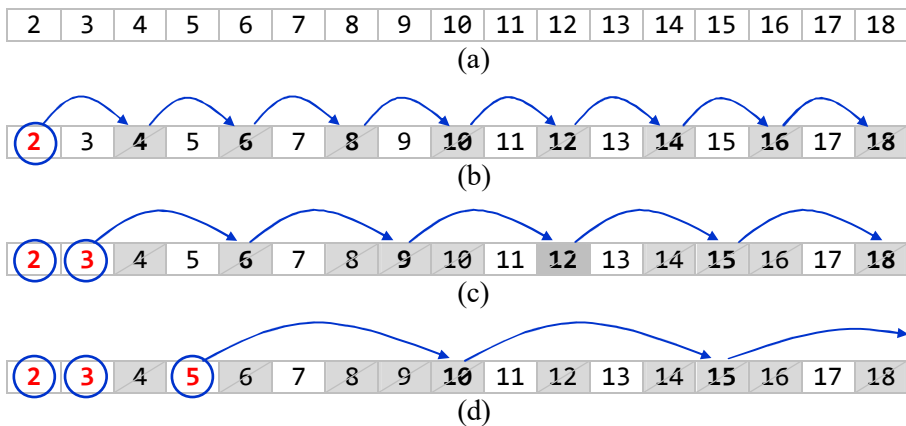
Problem je što ne postoji jednostavan obrazac ili pravilo otkrivanja sledećeg prostog broja, već se koristi neki metod sistematskog pronalaženja prostih brojeva u intervalu od interesa. Metod poznat još od antičkih vremena naziva se Eratostenovo sito (*Sieve of Eratosthenes*) [8].

Pronalaženje svih prostih brojeva manjih od nekog broja može se izvršiti proverom deljivosti svakog od brojeva iz intervala, ali bi to bilo veoma neefikasno. Eratostenovo sito jedan je od efikasnijih algoritama [19], a njegov osnovni oblik je [8], [19]:

- 1) Napravi se lista *svih* brojeva od 2 do n ;
- 2) Počev od prvog broja u listi (broj dva) označe se svi brojevi *deljivi* sa dva, dok dva ostaje kao prost broj. Postupak se ponavlja sa sledećim neoznačenim brojem, npr. m , označe se svi brojevi deljivi sa m , a on ostaje kao prost broj;
- 3) Lista prostih brojeva je skup svih neoznačenih brojeva.

Ilustracija primene Eratostenovog metoda u pronalaženja prva tri prosta broja prikazana je na Sl. 30.

Vidi se da algoritam počinje od spiska prirodnih brojeva (a) i prvog prostog broja 2 (b), čije sve množenike u spisku pronalazi i uklanja. Nakon toga prelazi na broj 3, sledeći koji nije uklonjen i ponavlja isti postupak (c), zatim prelazi na broj 5 (d), itd. sve dok ne ostanu neoznačeni samo prosti brojevi.



Sl. 30. Ilustracija primene metoda Eratostenovog sita

Metod je realizovan programom u jeziku Python, koji koristi strukturu liste za predstavljanje niza brojeva u zadanom intervalu:

```

from math import sqrt, ceil

def sito(n):
    # Vraća listu svih prostih brojeva manjih od n
    brojevi = [False, False] + [i for i in range(2, n)]
    for k in range(2, ceil(sqrt(n))):
        if brojevi[k] is not False:
            prosej(k, brojevi)
    return samo_prosti(brojevi)

def prosej(k, lista):
    # Uklanja iz liste brojeve deljive s k
    for i in range(2*k, len(lista), k):
        lista[i] = False

def samo_prosti(lista):
    # Izdvaja listu prostih brojeva
    prosti = []
    for i in lista:
        if i is not False:
            prosti.append(i)
    return prosti

print("Spisak svih prostih brojeva manjih od n")
n = int(input("Unesite n: "))
print("Prosti brojevi manji od", n, "su:\n", sito(n))

```

Inicijalno su u listi brojeva prikazani svi brojevi od 2 do zadanog broja kao potencijalni prosti brojevi. Označavanje ostalih brojeva koji nisu prosti brojevi vrši se na isti način, tako što se broj u listi zameni logičkom vrednošću False. Da bi indeksi liste bili jednaki samim brojevima, prikazani su i brojevi 0 i 1, ali su odmah označeni vrednošću False, tako da se ne smatraju prostim. Označavanje složenih brojeva, proizvoda tekućeg prostog broja, vrši se funkcijom `prosej()`, a izdvajanje konačne liste neoznačenih prostih brojeva vrši pomoću funkcije `samo_prosti()`.

Primer izvršavanja programa za pronalaženje prostih brojeva manjih od 50 je:

```

Spisak svih prostih brojeva manjih od n
Unesite broj n: 50
Prosti brojevi manji od 50 su:
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
>>>

```

Pitanja za ponavljanje

1. Šta su strukture podataka?
2. Objasnite razlike između struktura stringa, liste i n-torke.
3. Da li se elementi stringa mogu indeksirati računajući od kraja stringa?
4. Da li se elementi stringa mogu menjati?
5. Objasnite vezu između standarda binarnog zapisa znakova ASCII i Unicode.
6. Navedite funkcije za konverziju zapisa znakova.
7. Da li lista u jeziku Python može da sadrži elemente različitih tipova?
8. Da li naredba dodele vrednosti promenljive tipa lista drugoj promenljivoj proizvodi kopiju liste?
9. Navedite metode pomoću kojih se lista može koristiti kao struktra stek.
10. Navedite opšti oblik izraza za skraćeno generisanje elemenata liste. Dajte nekoliko primera.

8 Polja i neuređene liste u jeziku Python

1. Uvod
2. Polja
3. Neuređene liste
4. Primeri programa

U ovom poglavlju se ukratko opisuju nelinearne strukture podataka u jeziku Python, polja ili ugnježdene liste i rečnici ili neuređene liste.

8.1 Uvod

Struktura podataka je kolekcija više delova podataka strukturiranih na takav način da im se može efikasno pristupati.

Linearne strukture podataka u jeziku Python su nizovi: stringovi, jednostavne liste i n-torke.

Nelinearne strukture podataka u jeziku Python su *ugnježdene liste* i neuređene liste, odnosno *rečnici* (*dictionaries*).

Ugnježdene liste i rečnici su rekurzivne strukture, jer se kao element strukture javljaju opet one same. Rekurzivne strukture podataka omogućavaju predstavljanje i rad s podacima povezanim rekurzivnim relacijama, npr. grafovima, stablima i matricama.

Liste su strukture kod kojih elementi imaju numeričku poziciju, odnosno elementu liste se pristupa pomoću *indeksa* koji predstavlja *broj*.

Rečnici su strukture slične listama, kod kojih je indeks opštijeg tipa, a sama struktura nema linearno uređenje.

8.2 Polja

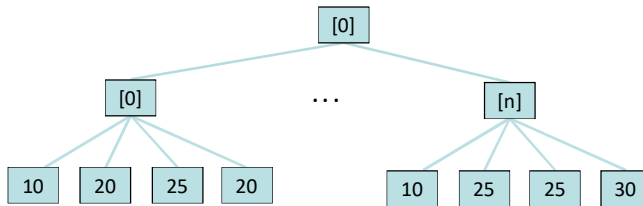
Polja ili matrice su višedimenzionalne strukture podataka, istog tipa, čijim elementima se pristupa preko skupa indeksa, po jedan indeks za svaku dimenziju. Jezik Python nema posebnu strukturu polja ili matrice, već se za prikaz višedimenzionalnih polja ili matrica koriste ugnježdene liste [1].

8.2.1 Ugnježdene liste

Ugnježdene liste imaju elemente koji su sami liste. Mogu se upotrebiti npr. za smeštanje sledećih podataka o poenima ostvarenim na ispitu iz nekog predmeta: aktivnost, kolokvijum 1, kolokvijum 2 i završni ispit:

```
uspeh_na_ispitu= [[10,20,25,20], ..., [10,25,25,30]]
```

Ovako definisana struktura podataka može se posmatrati kao plitka hijerarhija, u dva nivoa, Sl. 31.



Sl. 31. Prikaz podataka o ispitu pomoću ugnježdene liste u dva nivoa

8.2.2 Predstavljanje matrica

Dvodimenzionalne matrice dimenzija $m \times n$ mogu se predstaviti kao ugnježdene liste s dva nivoa indeksa, npr.

```
>>> matrica = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
               [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Pristup elementima matrice vrši se navođenjem indeksa, vodeći računa da indeksi počinju od nule, npr.

```
>>> matrica [1][2]
6
>>> matrica [0][1] = 44
>>> matrica
[[1, 44, 3], [4, 5, 6], [7, 8, 9]]
```

Kvadratna matrica 5x5 može se zadati sledećom listom:

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

Pristup elementu matrice a_{ij} vrši se navođenjem dva indeksa, gde prvi indeks *red*, odnosno indeks liste koja sadrži elemente jednog reda matrice, a drugi

indeks *kolona*, odnosno indeks elementa u okviru reda. Npr. za prethodnu matricu element a_{04} je:

```
>>> a = matrica[0][4]
5
```

Red matrice a_i , npr. s indeksom 1, dobija se kao:

```
>>> a = matrica[1]
[6, 7, 0, 0, 0]
```

8.2.3 Suma svih elemenata matrice

Matrica se predstavlja ugnježđenom listom s fiksnim brojem elemenata na jednom nivou. Zbir elemenata matrice dobija se sumiranjem svih elemenata po redovima i kolonama:

```
>>> matrica = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> suma = 0
>>> for red in matrica:
        for element in red:
            suma = suma + element
>>> print("Zbir elemenata matrice=", suma)
Zbir elemenata matrice= 45
```

Na sličan način vrši se obilazak matrice radi pronalaženja maksimalnog ili minimalnog elementa, štampanja svih elemenata matrice i sl.

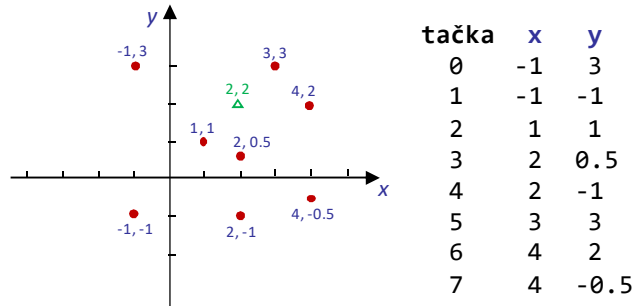
8.2.4 Množenje matrica

Množenje dve matrice $\mathbf{M} = \mathbf{AB}$ je operacija koja je definisana samo kad je broj redova jedne matrice jednak broju kolona druge, odnosno množenje matrica dimenzija $m \times n$ i $n \times p$ daje matricu dimenzija $m \times p$: $m_{ij} = \sum_{k=1..m} a_{ik} \cdot b_{kj}$. Npr. množenje matrica dimenzija 4×2 i 2×3 daje matricu dimenzija 4×3 :

```
>>> matA = [[1, 2], [4, 5], [7, 8], [4, 5]]
>>> matB = [[1, 2, 3], [4, 5, 6]]
>>> matC = [[0,0,0], [0,0,0], [0,0,0], [0,0,0]]
>>> for i in range(len(matA)): # petlja po redovima A
        for j in range(len(matB[0])): # petlja po kolonama B
            for k in range(len(matB)): # petlja po redovima B
                matC[i][j]=matC[i][j] + matA[i][k]*matB[k][j]
>>> matC
[[27, 12, 15], [24, 33, 42], [39, 54, 69], [24, 33, 42]]
```

Primer: Najbliža tačka u ravni

Matrica $n \times 2$ može se upotrebiti za prikaz rasporeda n tačaka u ravni. Elementi u kolonama matrice su koordinate tačaka x i y .



Pronalaženje tačke koja je prostorno najbliža zadanoj tački može se izvršiti linearnim pretraživanjem tabele (matrice) koordinata svih tačaka:

1. Postavi se da je prva tačka najbliža
2. Za sve preostale tačke:
 - na svakom koraku se računa udaljenost

$$d_{ab} = \sqrt{(x_a - x_b)^2 + (y_a - y_b)^2}$$
 od tačke b do zadane tačke a .
 - ako je udaljenost manja od do tada najbliže tačke, zapamte se koordinate x, y i udaljenost d nove najbliže tačke.

Tačke u ravni mogu se predstaviti u memoriji matricom 2×7 , koja se u jeziku Python realizuje ugnježđenom listom.

Pretraživanje je pogodno realizovati `for` petljom u kojoj se, počev od druge tačke, računa euklidska udaljenost tačaka do trenutno izabrane najbliže tačke, a zatim poredi s do tada ustanovljenom najmanjom udaljenošću i zapamti ako je udaljenost manja.

Pošto se često koristi, računanje euklidskog rastojanja dobro je realizovati pomoću posebne funkcije. Za računanje kvadratnog korena u ovoj funkciji po-godno je upotrebiti namensku funkciju `sqrt` (*square root*) iz modula `math`, kome se pristupa naredbom `import`.

Program u jeziku Python koji implementira opisani postupak:

```
# Pronalaženje najbliže tačke u ravni
# linearnim pretraživanjem
import math

def euklid(x,y):
    return math.sqrt( (x[0]-y[0])**2+(x[1]-y[1])**2 )

tacke    = [[-1, 3], [-1, -1], [1, 1], [2, 0.5],
             [2, -1], [3, 3], [4, 2], [4, -0.5]]
zadana   = [2, 2]
najbliza = tacke[0]
mind     = euklid(zadana, najbliza)
for i in range(1,len(tacke)):
    d = euklid(zadana, tacke[i])
    if d < mind:
        najbliza = tacke[i]
        mind     = d
print("Najbliza tačka tački", zadana, \
      "je", najbliza,"udaljena", mind)
```

Izvršavanje programa za zadanu tačku s koordinatama (2, 2) daje:

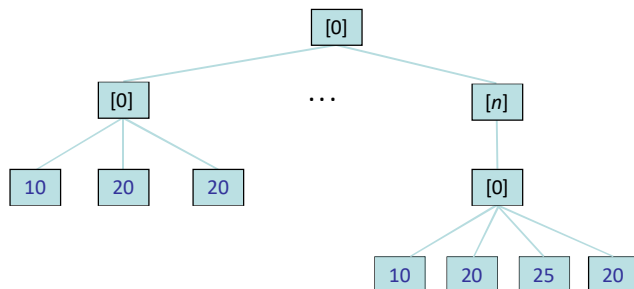
**Najbliza tačka tački [2, 2] je [1, 1] udaljena
1.4142135623730951**

8.2.5 Predstavljanje hijerarhija (stabala)

Hijerarhije su rekurzivne strukture, koje se mogu predstaviti ugnježdenim listama s različitim brojem elemenata na svakom nivou, npr. sledeća lista

```
[ [10,20,20] ... [ [10,20,25,20] ] ]
```

može se grafički predstaviti hijerarhijom prikazanom na Sl. 32.



Sl. 32. Prikaz opšte hijerarhija zadane ugnježdenom listom

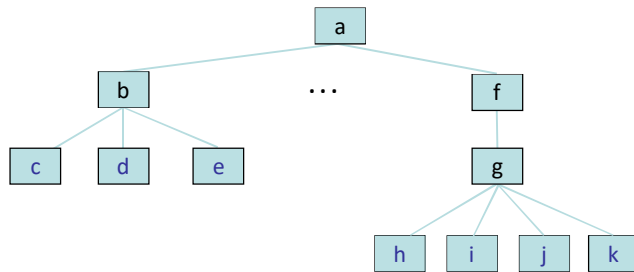
Struktura stabla se može predstaviti na različite načine, npr. tako da svaki čvor bude predstavljen listom, koja sadrži sam *čvor* i *listu* njegovih sledbenika:

```
[<čvor>, [<sledbenik1>, ... ,<sledbenikn>] ]
```

Sledbenik može biti podatak (terminalni simbol) ili ugnježdjena lista (struktura, neterminalni simbol). Na ovaj način se npr. sledećom strukturom podataka:

```
[ 'a',  
  [ 'b', [ 'c', 'd', 'e' ] ],  
  [ 'f' [ 'g' [ 'h', 'i', 'j', 'k' ] ] ] ]
```

može da predstavi opšte stablo, s proizvoljnim brojem grana u čvorovima, prikazano na Sl. 33.



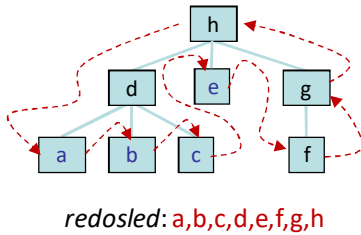
Sl. 33. Prikaz strukture stabla simbola zadan ugnježđenom listom

8.2.6 Obilazak stabla

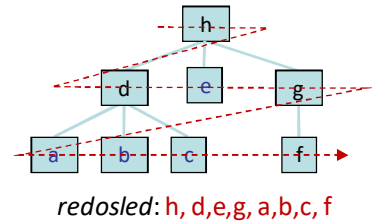
Radi izvršenja određenih operacija, struktura opšteg stabla može se obilaziti na dva načina, prikazana na Sl. 34:

- (a) u dubinu (*depth-first*), tako da se prvo se obiđu grane, a zatim koren stabla,
- (b) u širinu (*breadth-first*), tako što se prvo se obiđu svi čvorovi istog nivoa/dubine, a zatim se obilaze čvorovi na sledećem nivou.

Za obilazak stabla mogu se koristiti (1) rekurzivni algoritmi ili (2) iterativni algoritmi, koji koriste strukturu stek za pamćenje koraka koje treba izvršiti nakon povratka s nižih nivoa stabla.



(a)



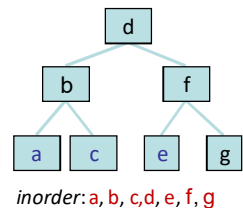
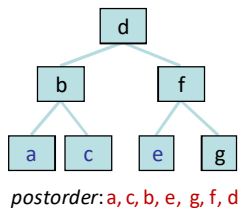
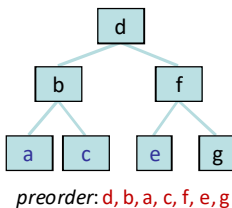
(b)

Sl. 34. Obilazak opšteg stabla

8.2.7 Binarna stabla

Binarna stabla imaju *najviše dve grane* svakog čvora stabla. Za binarna stabla, obilazak *u dubinu* može se realizovati na više načina, u odnosu na redosled operacija koje se izvršavaju na svakom čvoru može biti, Sl. 35:

- *preorder*: prvo se obiđe *čvor*, pa zatim oba njegova *podstabla*,
- *postorder*: prvo obuđu oba *podstabla* nekog čvora, a zatim sam *čvor*,
- *inorder*, po unutrašnjem uređenju: prvo se obiđe *levo podstablo*, zatim *čvor*, pa *desno podstablo*.



Sl. 35. Obilazak binarnog stabla

Kao i kod opštih stabala, obilazak binarnog stabla može se izvršiti pomoću rekursivnih algoritama ili iterativnim algoritmima koji koriste strukturu stek.

8.3 Neuređene liste: rečnici i skupovi

Neuređene liste ili strukture *rečnik* i *skup* su takve strukture podataka kod kojih redosled elemenata nije od značaja, jer im se pristupa na osnovu nenumeričkog indeksa (*ključa*) ili samo pomoću njihove vrednosti, koja je tada ključ za pristup.

8.3.1 Struktura rečnik (dictionary)

Sintaksa za definisanje strukture rečnika je:

```
{<ključ>:<vrednost>, ..., <ključ>:<vrednost>}
```

Ključ je podatak na osnovu kojeg se pristupa vrednosti elementa u rečniku. *Vrednost* elementa može biti bilo kog tipa, uključujući sam rečnik, tako da se pomoću rečnika može predstaviti hijerarhijska struktura podataka. Pristup podacima pomoću zadanog ključa može se prikazati na primeru podataka o nekretninama, gde je pojedinačna nekretnina prikazana rečnikom, npr.

```
kuca = {'boja' : 'bež', 'stil' : 'pedesetih',
        'brojSoba' : 4, 'imaGarazu' : True, 'imaAlarm': False,
        'kucniBroj' : 123, 'ulica' : 'Trešnjina', 'grad' :
        'Beograd', 'cena' : 125000}
```

Pristup određenom podatku o nekretnini iz prethodne strukture vrši se zadavanjem vrednosti njegovog ključa, npr.

```
>>> kuca['cena']
125000
```

8.3.2 Primena operatora 'in' na strukturu rečnika

Neka su podaci o broju stanovnika nekoliko gradova u Srbiji prikazani u sledećoj strukturi rečnika:

```
gradovi = {'Beograd':1659440, 'Novi Sad':341625,
           'Niš':373404, 'Kragujevac':179417}
```

Pošto rečnik definiše skup elemenata, za obilazak rečnika može se koristiti petlja for:

```
for grad in gradovi:
    print(grad)
```

Prethodna petlja prikazuje samo spisak gradova (ključeva). Prikaz ostalih podataka može se dobiti upotrebom ključa kao indeksa, npr.

```
>>> for grad in gradovi:
        print(grad, gradovi[grad])
Novi Sad 341625
Niš 373404
Beograd 1659440
Kragujevac 179417
```

8.3.3 Kombinovanje struktura rečnika i uređene liste

Rečnici su generalizovane liste, a obe strukture mogu da sadrže kao elemente druge strukture, tako da omogućavaju hijerarhijsku organizaciju podataka.

Strukture podataka rečnika i liste mogu se međusobno kombinovati, zavisno od vrste podataka i operacija koje su potrebne na različitim nivoima hijerarhije, npr.

- struktura rečnika koja ima vrednosti elemenata koji su liste podataka
`{<ključ>:[<vrednost>,...], ... }`
- ugnježdene liste koje predstavljaju strukturu stabla koje ima čvorove u obliku rečnika
`[{<ključ>:<vrednost>}, [{<ključ>:<vrednost>},...]]`

8.3.4 Spisak promenljivih u jeziku Python

Promenljive programa u jeziku Python čuvaju se u strukturi tipa rečnika. Prilikom pokretanja interpretera, postoji unapred definisan skup sistemskih promenljivih, koje se mogu prikazati pomoću ugrađene funkcije `vars()`:

```
>>> vars()
{'__name__': '__main__', '__doc__': None,
 '__package__': None, '__loader__': <class
 'frozen_importlib.BuiltinImporter'>, '__spec__': None,
 '__annotations__': {}, '__builtins__': <module
 'builtins' (built-in)>}
```

Kreiranjem novih promenljivih, u ovaj rečnik se dodaju novi elementi, kojima se može pristupiti na standardni način. Npr. vrednost neke promenljive može se dobiti kao:

```
>>> a = b = 123
>>> vars()
{'__name__': '__main__', '__doc__': None,
 '__package__': None, '__loader__': <class
 'frozen_importlib.BuiltinImporter'>, '__spec__': None,
 '__annotations__': {}, '__builtins__': <module
 'builtins' (built-in)>, 'a': 123, 'b': 123}
>>> imena = vars()
>>> imena["a"]
123
```

8.3.5 Struktura skup (set)

Struktura skup u jeziku Python služi za predstavljanje neuređene liste *jedinstvenih* elemenata, tako da ne sadrži duplikate elemenata. Skup se zadaje nabiranjem njegovih elemenata u velikim zagradama {} ili pomoću funkcije `set(<lista_elementa>)`, npr.

```
>>> set()           # funkcija kreira prazan skup
set()
>>> {2, 4, 6}       # zadavanje elemenata u zagradama
{2, 4, 6}
>>> set([2, 4, 6])  # zadavanje skupa funkcijom set
{2, 4, 6}
>>> set("abcd")     # zadavanje skupa znakova
{'d', 'c', 'a', 'b'}
```

Za razliku od rečnika, elementi strukture skup nemaju posebne ključeve za pristup, već se pristup elementima strukture vrši pomoću njih samih i operatora pripadnosti `in` i `not in`.

Osnovne operacije nad strukturom skup su *dodavanje* elemenata skupa metodom `add()` i *uklanjanje* metodom `remove()`.

Osnovne operacije nad skupovima u jeziku Python su: unija (*union*, operator `|`), presek (*intersection*, operator `&`), razlika (*difference*, operator `-`) i simetrična razlika (*symmetric difference*, *XOR*, operator `^`), npr.

```
>>> s1 = {1, 2, 3}
>>> s2 = {3, 4, 5}
>>> s1.union(s2)    # isto kao: s1 | s2
{1, 2, 3, 4, 5}
>>> s1 - s2         # isto kao: s1.difference(s2)
{1, 2}
```

8.4 Primeri programa

Primeri programa koji ilustruju upotrebu više različitih struktura podataka u jeziku Python su:

1. Geografija - upotreba rečnika,
2. Najbliži gradovi (Dijkstrin algoritam) - upotreba liste kao matrice,
3. Igra pogađanja reči (*hangman*) - upotreba n-torki.

8.4.1 Geografija

Struktura rečnika treba da sadrži podatke o površini dvadesetak najvećih svetskih država (u km²) [19]:

• Rusija	16.377.742
• Kanada	9.093.507
• Kina	9.569.901
• Sjedinjene Američke Države	9.158.960
...	
• Južnoafrička Republika	1.214.470
• Kolumbija	1.038.700

Program daje informacije o broju stanovnika izabrane države prema sledećem algoritmu:

1. Korisnik zadaje *ime* države za koju se prikažu podaci o njenoj površini.
2. Program pronalazi u *rečniku* broj stanovnika prema *imenu* države i prikazuje na ekranu računara. Ako država nije u rečniku, program ispisuje poruku "Nažalost nemamo informacija o toj državi".
3. Program završava rad kad korisnik unese prazan string (samo taster *Enter*).

Kompletan program u jezikuPython je:

```
# Površina najvećih zemalja sveta (km2)

drzave = {
    'Rusija' : 16377742, 'Kanada' : 9093507,
    'Kina' : 9569901, 'Sjedinjene Američke Države' :
9158960,
    'Brazil' : 8460415, 'Australija' : 7682300,
    'Indija' : 2973193, 'Argentina' : 2736690,
    'Kazahstan' : 2699700, 'Alžir' : 238741,
    'Demokratska Republika Kongo' : 2267048,
    'Saudijska Arabija' : 2149690, 'Meksiko' : 1943945,
    'Indonezija' : 1811569, 'Libija' : 1759540,
    'Iran' : 1531595, 'Mongolija' : 1553556,
    'Peru' : 1279996, 'Čad' : 1259200, 'Niger' : 1266700,
    'Angola' : 1246700, 'Mali' : 1220190,
    'Junoafrička Republika' : 1214470,
    'Kolumbija' : 1038700, 'Srbija' : 881361 }
```

```

kraj = False
while not kraj:
    nazivDrzave = input('Unesi naziv države: ')
    if nazivDrzave == '':
        kraj = True
    else:
        if nazivDrzave in drzave:
            površina = drzave[nazivDrzave]
            print('Površina države', nazivDrzave, 'je',
                  površina, ' (km2) ')
        else:
            print('Nažalost nemamo informacija o: ',
                  nazivDrzave)
print()

```

Primer izvršavanja programa za dve države iz spiska je:

```

Unesi naziv države: Alžir
Površina države Alžir je 238741  (km2)

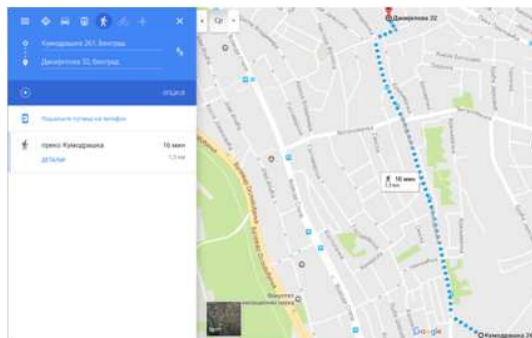
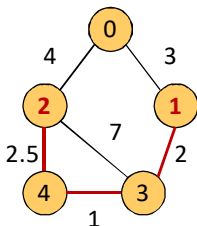
Unesi naziv države: Srbija
Površina države Srbija je 881361  (km2)

Unesi naziv države:
>>>

```

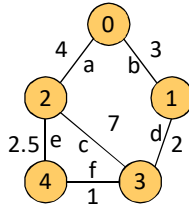
8.4.2 Najbliži gradovi (Dijkstrin algoritam)

U ovom primeru se demonstrira upotreba matrica u rešavanju grafovskih problema. Graf $G(v,e)$ je skup čvorova v (*vertex*, *node*) povezanih lukovima e (*edge*). Osim oznaka, lukovi grafa mogu imati i različita kvantitativna obeležja, koja predstavljaju npr. cenu, dužinu puta i sl. Pretraživanjem je moguće pronaći najkraći put između dva čvora grafa, npr. između čvorova 1 i 2 grafa na sledećoj slici.



Ako čvorovi grafa predstavljaju naselja, a lukovi direktne puteve između njih, *najkraći put u grafu* odgovara najkraćem putu između dva naselja (lokacije), koji se npr. može dobiti pomoću aplikacije www.google.com/maps.

Grafovi se mogu predstaviti matricama susedstva i matricama incidencije, u koje se smeštaju neophodne informacije iz grafa, kao što je npr. sledeći graf:



- *Matrica susedstva* (adjacency) $A = [a_{ij}]$ pokazuje da li su dva čvora povezana lukom. Indeksi i i j su indeksi čvorova, a elementi matrice a_{ij} mogu da označavaju npr. da li su dva čvora povezana (0 ili 1), broj lukova ili dužinu luka ij

$$\begin{matrix} & 0 & 1 & 2 & 3 & 4 \\ \begin{matrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} 0 & 3 & 4 & - & - \\ 3 & 0 & - & 2 & - \\ 4 & - & 0 & 7 & 2.5 \\ - & 2 & 7 & 0 & 1 \\ - & 2.5 & 1 & 0 & 0 \end{bmatrix} \end{matrix}$$

- *Matrica incidencije* $B = [b_{ij}]$ pokazuje da li je neki čvor povezan (incidentan) s nekim lukom. Indeks i je čvor, j je oznaka luka, a element matrice $b_{ij} = 0$ ili 1 označava da li je čvor i povezan s lukom j

$$\begin{matrix} & a & b & c & d & e & f \\ \begin{matrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix} \end{matrix}$$

Jedan od najpoznatijih algoritama za pronalaženje najkraćeg puta između dva čvora u grafu je Dijkstrin algoritam [11].

Osnovna verzija Dijkstrinog algoritma predstavlja varijantu pretraživanja u širinu (*breadth-first-search*), jer posećuje susedne čvorove po redosledu njihovog pronalaženja. Za pamćenje njihovog redosleda koristi se struktura reda čekanja (*queue*), u koju se novi elementi dodaju na kraj reda, a uzimaju s početka reda.

Prilikom pretraživanja, poseta svakom čvoru se označava, kako bi se kasnije posećenost mogla proveriti i pretraživanje nastavilo obilaskom ostalih povezanih, još neposećenih čvorova.

Pojednostavljena verzija Dijkstrinog algoritma pretpostavlja jednake težine svih lukova [11]:

1. Početnom čvoru s dodeli se rastojanje 0 i on se dodaje u red čekanja.
Ostalim čvorovima dodeli se beskonačno rastojanje od početnog čvora.
2. Sve dok red čekanja nije prazan:
 - 2.1 Uklanja se iz reda čekanja prvi čvor i dodeli mu se rastojanje d .
 - 2.2 Pronađu se svi lukovi povezani s ovim čvorom.
 - 2.3 Za svaki povezani čvor s beskonačnim rastojanjem, rastojanje se zameni s $d+1$, a čvor se dodaje na kraj reda čekanja.

Osnovna verzija Dijkstrinog algoritma ne pretpostavlja jednake težine/dužine lukova i čvorove označava dužinom najkraćeg puta (od početnog čvora) i za pronalazi najkraće puteve od početnog čvora do svih ostalih čvorova grafa.

Unapređena verzija algoritma koristi strukturu prioritetnog reda čekanja umesto običnog radi značajnog unapređenja performansi.

Program za osnovnu verziju Dijkstrinog algoritma u jeziku Python je [29]:

```
"""Program pronalazi najkraći put između dva čvora
u zadanom grafu.

U ovom školskom primeru graf je definisan unapred
unesenom matricom dužina lukova (adjacency matrix).
Za pronalaženje najkraćeg puta koristi se
Dijkstrin algoritam.

"""

INF = float('inf')

def dijkstra(graf, od):
    #
    # Dijkstrin algoritam koji pronalazi najkraće puteve
    # u grafu od zadanog čvora do svih ostalih
    #
    cvorovi = [i for i in range(len(graf))]
```

```

poseceni = [od]
put = {od:{od:[]}}
cvorovi.remove(od)
rastojanja_cvorova = {od:0}
preth = sled = od

while cvorovi:
    rastojanja = INF
    for v in poseceni:
        for d in cvorovi:
            novo_rastojanje = graf[od][v]+graf[v][d]
            if novo_rastojanje < rastojanja:
                rastojanja = novo_rastojanje
                sled = d
                preth = v
                graf[od][d] = novo_rastojanje

    put[od][sled] = [i for i in put[od][preth]]
    put[od][sled].append(sled)

    rastojanja_cvorova[sled] = rastojanja

    poseceni.append(sled)
    cvorovi.remove(sled)

return rastojanja_cvorova, put

# Testni primer grafa
graf = [[0,3,4,INF,INF],
        [3,0,INF,2,INF],
        [4,INF,0,7,2.5],
        [INF,2,7,0,1],
        [INF,INF,2.5,1,0]]

print("Program pronalazi najkraći put između dva čvora")
print("u grafu koji je definisan matricom dužina lukova")
print("(adjacency matrix):")
print("
      (0)
      4 / \ 3
      /   \
(2)      (1)
 | \ 7 /
2.5 | \ / 2
(4) -- (3)
      1
")

$$\begin{bmatrix}
0 & 3 & 4 & - & - \\
3 & 0 & - & 2 & - \\
4 & - & 0 & 7 & 2.5 \\
- & 2 & 7 & 0 & 1 \\
- & - & 2.5 & 1 & 0
\end{bmatrix}$$


```

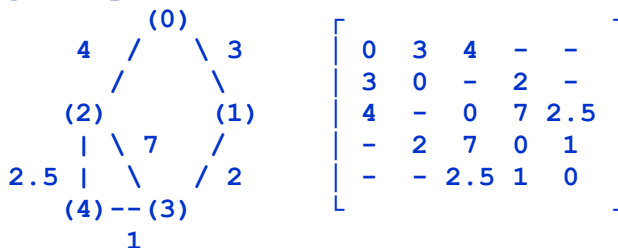
```
# Najkraći put između dva zadana čvora
od = int(input("Unesi početni čvor putanje (0..4): "))
do = int(input("Unesi krajnji čvor putanje (0..4): "))

# Pronalaženje najkraćih puteva do svih ostalih čvorova
rastojanja, put = dijkstra(graf, od)

print("\nNajkraći put između čvorova", od, "i", do, \
      "dužine", rastojanja[do], "je:\n", put[od][do])
```

Izvršavanje programa za pronalaženje najkraćeg puta između čvorova 2 i 1 testnog primera grafa daje sledeći rezultat:

Program pronalazi najkraći put između dva čvora u grafu koji je definisan matricom dužina lukova (adjacency matrix):



Unesi početni čvor putanje (0..4): 2

Unesi krajnji čvor putanje (0..4): 1

**Najkraći put između čvorova 2 i 1 dužine 5.5 je:
[4, 3, 1]**

8.4.3 Igra pogađanja reči (hangman)

Programska realizacija poznate društvene igre pogađanja sastoji se u pogađanju (slovo po slovo) nepoznate reči prema sledećem algoritmu:

1. Program slučajno bira reč iz zadane liste reči;
2. Korisnik pokušava da pogodi nepoznatu reč, slovo po slovo;
3. Sistem prikazuje svako slovo ispravno pogođenih ili pogrešnih slova;
4. Za ispravno pogođena slova prikazuje se i položaj u nepoznatoj reči;
5. Za pogrešna slova, prikazuje se jedan od delimičnih prikaza različitih faza "obešenog" igrača, svaki put s većim stepenom detalja;

Pogađanje reči (slovo po slovo):

- igrač unosi jedno slovo, nakon čega se proverava se da li je uneseno tačno jedno slovo i ništa drugo,
- igrač se upozorava ako unese više slova, ponovi neko od prethodnih ili unese veliko slovo i traži se da ponovi unos.

Ispitivanja ispravnosti unesenog slova:

- proverava se da li slovo pripada nepoznatoj reči
- ako slovo pripada nepoznatoj reči, proverava se da li su pogođena sva slova, kada se igrač obaveštava da je pobedio, a tekuća igra završava. Ako igrač ima pravo još da pogađa, igra se nastavlja,
- ako slovo ne pripada nepoznatoj reči, dodaje se u pogrešna slova,
- ako je igrač pogađao prevelik broj puta i izgubio, prikazuje se poslednja slika "vešala" (7) i tekuća igra završava.

Igra pogađanja realizovan je sledećim programskim kodom:

```
import random

SLIKE_VESALA = ( '''
+---+
|   |
|   |
|   |
===== ' ', ' '

+---+
|   |
O   |
|   |
===== ' ', ' '

+---+
|   |
O   |
|   |
===== ' ', ' '
''' )
```

```

+---+
|   |
O   |
/|  |
    |
===== ' ', ' '

```

```

+---+
|   |
O   |
/|\ |
    |
===== ' ', ' '

```

```

+---+
|   |
O   |
/|\ |
/   |
===== ' ', ' '

```

```

+---+
|   |
O   |
/|\ |
/ \ |
===== ' ' )

```

```

RECI = ('ajkula', 'babun', 'ćurka', 'dabar', 'foka', \
        'gavran', 'golub', 'guska', 'gušter', \
        'jaguar', 'jarac', 'jastreb', 'jelen', \
        'kamila', 'kit', 'kobra', 'kojot', 'labud', \
        'lama', 'lasica', 'lav', 'lisica', 'los', \
        'losos', 'mačka', 'magarac', 'majmun', \
        'mazga', 'medved', 'miš', 'mrav', 'mula', \
        'nosorog', 'orao', 'ovan', 'ovca', 'pacov', \
        'panda', 'papagaj', 'pas', 'patka', 'pauk', \
        'piton', 'roda', 'šišmiš', 'školjka', 'sova', \
        'tigar', 'tvor', 'vidra', 'vrana', 'vuk', \
        'žaba', 'zebra', 'zec', 'zmija')

```

```

def Inicijalizacija():

    # Inicijalizacija globalnih promenljivih
    global pogresnaSlova, ispravnaSlova, nepoznataRec
    pogresnaSlova = ''
    ispravnaSlova = ''

    # Slučajni broj [0..broj reči]
    indeks = random.randint(0, len(RECI)-1)
    nepoznataRec = RECI[indeks]
    return

def prikaziVesala():

    # Prikaz odgovarajuće slike na osnovu
    # broja pogrešnih pogađanja

    print(SLIKE_VESALA[len(pogresnaSlova)])
    print()
    print('Pogrešna slova:', end=' ')
    for slovo in pogresnaSlova:
        print(slovo, end=' ')
    print()
    praznine = '_' * len(nepoznataRec)

    # Zamena praznina ispravno pogođenim slovima
    for i in range(len(nepoznataRec)):
        if nepoznataRec[i] in ispravnaSlova:
            praznine = praznine[:i] + nepoznataRec[i] + \
                praznine[i+1:]

    # Prikaz tajne reči s praznim mestom između slova
    for slovo in praznine:
        print(slovo, end=' ')
    print()
    return

```

```

def unosSlova(dosadasnjaPogadjanja):

    # Funkcija vraća slova koja je korisnik uneo
    # i proverava da je uneseno samo jedno slovo
    # i ništa drugo

    while True:
        print('Pogodite slovo.')
        pogadjanje = input()
        pogadjanje = pogadjanje.lower()
        if len(pogadjanje) != 1:
            print('Molim unesite jedno slovo.')
        elif pogadjanje in dosadasnjaPogadjanja:
            print('Već ste naveli ovo slovo. ',
                  'Izaberite neko drugo.')
        elif pogadjanje not in \
              'abcdefghijklmnopqrstuvwxyzčćžšđ':
            print('Molim unesite malo SLOVO.')
        else:
            return pogadjanje

print("-----")
print("IGRA POGAĐANJA REČI 'V E Š A L A'")

Inicijalizacija()

```

```

krajIgre = False
while not krajIgre:

    prikaziVesala()

    # Igrač treba da unese (pogodi)
    # jedno slovo nepoznate reči
    pogadjanje = unosSlova(pogresnaSlova + ispravnaSlova)

    if pogadjanje in nepoznataRec:
        ispravnaSlova = ispravnaSlova + pogadjanje
        # Provera da li je igrač već pobedio
        pronadjenaSvaSlova = True
        for i in range(len(nepoznataRec)):
            if nepoznataRec[i] not in ispravnaSlova:
                pronadjenaSvaSlova = False
                break
        if pronadjenaSvaSlova:
            print('\nDa! Tajna reč je bila "'+
                  nepoznataRec + '"! Pobedili ste!')
            krajIgre = True
    else:
        pogresnaSlova = pogresnaSlova + pogadjanje
        # Provera da li je igrač pogađao
        # prevelik broj puta i izgubio
        if len(pogresnaSlova) == len(SLIKE_VESALA) - 1:
            prikaziVesala()
            print('\nPrekoračili ste dozvoljeni broj',
                  'pogađanja posle ' +
                  str(len(pogresnaSlova)) +
                  ' pogrešnih i ' +
                  str(len(ispravnaSlova)) +
                  ' ispravnih pogađanja.\nReč je bila "'+
                  nepoznataRec + '"')
            krajIgre = True

    # Ako je igra završena, pitanje igraču
    # da li želi da igra ponovo
    if krajIgre == True:
        odg = input('\nŽelite li da igrate ponovo?(d/n)')
        if odg != 'n':
            Inicijalizacija()
            krajIgre = False

```

Primer izvršavanja programa, prilikom uspešnog pogađanja jedne reči je:

IGRA POGAĐANJA REČI 'V E Š A L A'

Pogrešna slova:

Pogodite slovo.

a

Pogrešna slova:

— — — — a
Pogodite slovo.

e

Pogrešna slova: e

— — — — ^a
Pogodite slovo.

i

```

+---+
|   |
O    |
     |
     |
=====

```

Pogrešna slova: e

_ i _ _ a

Pogodite slovo.

v

```

+---+
|   |
O    |
     |
     |
=====

```

Pogrešna slova: e

v i _ _ a

Pogodite slovo.

d

```

+---+
|   |
O    |
     |
     |
=====

```

Pogrešna slova: e

v i d _ a

Pogodite slovo.

r

Da! Tajna reč je bila "vidra"! Pobili ste!

Želite li da igrate ponovo?(d/n)n

>>>

Pitanja za ponavljanje

1. Koje se strukture podataka u jeziku Python mogu označiti kao nelinearne?
2. Na koji način se u jeziku Python predstavljaju matrice?
3. Kako se pristupa elementu matrice predstavljene ugnježđenim listama?
4. Objasnite način realizacije operacije sumiranja elemenata i množenja matrica predstavljenih ugnježđenim listama.
5. Kako se pomoću liste može predstaviti struktura opšteg stabla?
6. Navedite metode obilaska opštih stabala.
7. Navedite metode obilaska u dubinu binarnih stabala.
8. Kako se petlja for može upotrebiti za obilazak elemenata rečnika?
9. Na koji način se mogu kombinovati strukture liste i rečnika?
10. Koja je namena Dijkstrinog algoritma?

9 Organizacija programskog koda u jeziku Python

1. Uvod
2. Upotreba modula u jeziku Python
3. Specifikacija modula
4. Projektovanje softvera s vrha (top-down)
5. Moduli u jeziku Python
6. Primeri programa

U ovom poglavlju objašnjava se potreba za podelom programa na manje celine, kao i načini modularizacije programa u jeziku Python.

9.1 Uvod

Softverski sistemi mogu da predstavljaju izuzetno složene entitete. Neki od softverskih sistema spadaju u najsloženije sisteme koje su ljudi stvorili, npr.

- operativni sistemi i Veb čitači sastoje se od 50–100 miliona linija koda (*lines of code*, LOC),
- poslovni informacioni sistem SAP sastoji se od oko 240 miliona linija koda u programskom jeziku visokog nivoa,
- pretraživač Google je softverski sistem čiji obim se procenjuje na 2 milijarde linija koda.

Na Sl. 36 prikazan je obim nekih poznatijih softverskih sistema iz kategorije ugrađenih, operativnih i informacionih sistema [19], [30]. Obim izvornog koda ovih programa izražen je brojem programskih linija (*Lines of Code*, LOC). Smatra se da je za prikaz jednog miliona linija koda potrebno oko 18.000 štampanih stranica [30].

Programski kod složenijih softverskih sistema nije dovoljno pregledan ako se predstavi kao niz funkcija istog nivoa. Osim toga, složeni programi se razvijaju timski, tako da je veoma važno da se program podeli na takve celine koje mogu da razvijaju manje grupe ili pojedinci.

Zbog toga se programski kod u jeziku Python na najvišem nivou organizuje kao skup *modula* (*modules*), manjih programskih celina sastavljenih od funkcija i drugih objekata, koji se mogu hijerarhijski organizovati u *pakete* (*packages*).

Modularno projektovanje (*modular design*) omogućava:

- podelu izuzetno velikih programa u manje delove, koji imaju jasne funkcije i kojima je lakše upravljati;
- raspodelu programskih zadataka na veći broj programera ili razvojnih timova;
- nezavisni razvoj i testiranje pojedinih celina (modula), koji se mogu kasnije uključiti u različite složenije sisteme;
- lakše izmene programskog koda, koje se mogu vršiti samo u određenim modulima, te ih nije potrebno sprovesti i u ostalim delovima složenog softverskog sistema.

Softver	Obim (Lines of Code, LOC)	
Ugrađeni softver (embeeded)		
Space Shuttle		400.000
Rover Curiosity		2.500.000
Mercedes klase S 2014		65.000.000
Moderni luksuzni automobili		100.000.000
Operativni sistemi		
Red Hat Linux 7		30.000.000
Windows XP		45.000.000
Windows 7, 8, 10		50 .. 80.000.000
MAC OS X 10		86.000.000
ERP poslovni informacioni sistemi		
SAP NetWeaver (ABAP)		238.000.000
Google pretraživač (2015)		2.000.000.000

Sl. 36. Obim nekih poznatih programa

Prednosti upotrebe modula su:

- U *projektovanju* softvera (*software design*) upotreba modula je način razvoja dobro projektovanog softvera.

- U *razvoju* softvera (*software development*) predstavlja način podele zadataka programiranja i višestruke upotrebe razvijenog programskog koda.
- U *testiranju* softvera omogućava zasebno testiranje delova programa i njihovu integraciju u toku testiranja.
- U *održavanju* softvera, koji čini najveći deo životnog veka softvera, olakšava unošenje izmena u pojedine funkcije programa.

9.2 Upotreba modula u jeziku Python

Dobro projektovan softver u jeziku Python sastoji se od skupa *modula*. Načelno, modul označava projektovane i/ili implementirane funkcionalnosti koje će se uključiti u neki program.

Modul se načelno sastoji od skupa funkcija i drugih programskih objekata. Primeri programskih modula su ranije korišćeni moduli `turtle` i `random`.

Svaki *fajl* na kojem je program u jeziku Python predstavlja modul i može se uključiti u druge programe pomoću naredbe `import`.

9.3 Specifikacija modula

Svaki modul treba da ima specifikaciju, koja omogućava njegovu pravilnu upotrebu (*intefejs* modula).

Svaki program koji koristi modul je njegov *klijent*, a specifikacija treba da omogući klijentu da ga efiksano koristi

Npr. specifikacija modula za računanje broja prostih brojeva u zadanom rasponu može biti veoma jednostavna:

```
def brojProstihBrojeva(od, do):
    """Vraća broj prostih brojeva
       između dva zadana broja."""
```

U praksi je ipak potrebna detaljnija specifikacija, koja treba da razjasni važne detalje funkcionisanja, npr. da li su u rezultat uključene vrednosti `od` i `do`, šta se vraća kad nema nijednog prostog broja i slično.

Neformalna konvencija u jeziku Python podrazumeva da ovakva dokumentacija (tzv. *docstring*) na početku ima opšti opis funkcije u jednoj liniji, iza koga sledi prazna linija, nakon koje može da sledi duža detaljnija specifikacija funkcije.

Npr. u prethodnu funkciju se dodaje prazna linija i nastavak opisa:

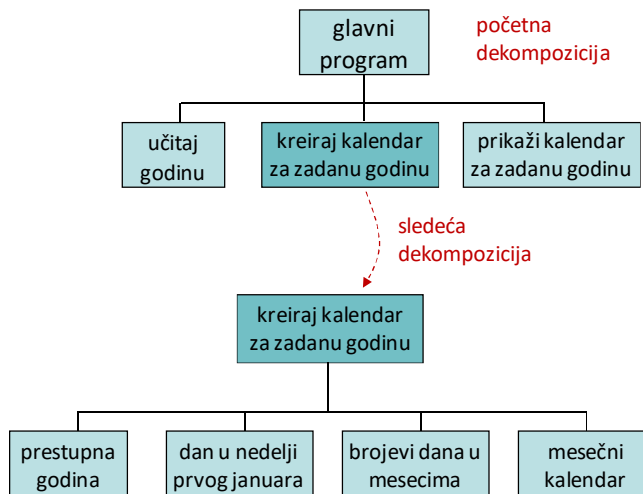
```
def brojProstihBrojeva (od, do):
    """Vraća broj prostih brojeva između dva
       zadana broja, uključno.

       Vraća kod -1 ako je vrednost 'od' veća
       od vrednosti 'do'
    """
```

Indentacija dodatnog teksta opisa funkcije treba da bude u istom nivou.

9.4 Projektovanje softvera s vrha (top-down)

Projektovanje softvera "s vrha" ili "odozgo" (*top-down*) podrazumeva postepenu dekompoziciju problema u manje module, dok se ne dobiju dovoljno male i jasne celine, koje je lako programski realizovati. Primer projektovanja s vrha programa za štampanje kalendara za zadanu godinu, Sl. 37.



Sl. 37. Primer projektovanja s vrha (top-down)

Projektovanje s vrha može započeti unosom specifikacija budućih modula, npr.

```
def učitaj_godinu():
    """Vraća celi broj između 1800 i 2099,
       uključno ili -1 ."""
```

```
def prestupna_godina(godina):
    """Vraća True ako je unesena godina prestupna
       inače vraća vrednost False."""
```

```
def dan_u_nedelji_prvog_januara(godina, prestupna):
    """Vraća dan u nedelji za 1. janur zadane godine.
       Godina mora biti između 1800 i 2099,
       a funkcija prestupna_godina() mora biti True
       ako je godina prestupna, inače je False.
    """
```

itd ...

9.5 Moduli u jeziku Python

Moduli u jeziku Python sadrže definicije, ali mogu da sadrže i naredbe, koje se izvršavaju samo jednom, obično radi inicijalizacije.

Standardna biblioteka jezika Python sadrži ugrađene (*built-in*) module, koji se koriste pomoću naredbe `import`. Prilikom uvoza, modul se pronalazi u fajl sistemu i smešta u memoriju. Redosled pretraživanja je:

1. Prvo se traži u tekućem folderu;
2. Ako se fajl ne pronade, traži se u folderu navedenom u promenljivoj `PYTHONPATH` (može se postaviti komandom `set PYTHONPATH`);
3. Ako se ne pronade ili ova promenljiva nije definisana, traži se u folderu koji je definisan prilikom instalacije sistema, npr. `C:\Python\Lib`;
4. Ako se modul ne pronade, dojavljuje se greška (*ImportError*).

Primer koda jednostavnog modula, koji može da se sačuva na na radnom folderu u fajlu `simple.py`:

```
# Modul simple
print('modul "simple" napunjen')

def funkcija1():
    print('"funkcija1" je pozvana')

def funkcija2():
    print('"funkcija2" je pozvana')
```

Ukoliko je potrebno promeniti radni folder, može se koristiti funkcija `chdir` iz modula `os`, koji sadrži različite funkcije operativnog sistema:

```
>>> import os
>>> os.chdir('C:\\Users\\korisnik\\Documents\\')
```

9.5.1 Prostor imena modula

Moduli se u programima tipično koriste tako da njihov prostor imena ostaje zaseban. Zbog toga je uz naziv objekta u nekom modulu neophodno dodati i naziv tog modula, npr.

```
>>> import simple
modul "simple" napunjen
>>> simple.funkcija2()
"funkcija2" je pozvana
```

Drugi način je da se oba prostora imena objedine naredbom `import`, pa nije potrebno navoditi naziv modula uz nazive objekata u modulu, npr.

```
>>> from simple import *
modul "simple" napunjen
>>> funkcija2()
"funkcija2" je pozvana
```

Da se smanji verovatnoća preklapanja imena, u naredbi `import` se mogu nabrojati samo oni objekti koje je potrebno uvesti, npr.

```
>>> from simple import funkcija1
modul "simple" napunjen
>>> funkcija1()
"funkcija1" je pozvana
>>> funkcija2()
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    funkcija2()
NameError: name 'funkcija2' is not defined
>>> simple.funkcija2()
```

```
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    simple.funkcija2()
NameError: name 'simple' is not defined
```

U primeru dolazi do greške jer `funkcija2` nije uvezena naredbom `import`.

9.5.2 Moduli i paketi

Modul u jeziku Python je programski fajl, koji sadrži kod koji će se koristiti u drugim programima. Veliki broj modula na istom nivou otežava njihovu upotrebu, pa je uveden mehanizam paketa, koji omogućava hijerarhijsku organizaciju modula.

Paket je folder koji sarži skup modula i jedan fajl s rezervisanim nazivom `__init__.py` [9]. Na folderu paketa čuva se skup međusobno povezanih modula.

Npr. sadržaj foldera paketa s modulima za rad sa slikama može biti [9]:

```
Slike/
  __init__.py
  Bmp.py
  Jpeg.py
  Png.py
  Tiff.py
  Xpm.py
```

Prilikom uvoza ovih modula, nazivu fajla se kao prefiks dodaje naziv foldera na kome se nalaze, npr. za uvoz modula `Bmp.py` treba navesti:

```
import Slike.Bmp
```

Ponekad je pogodno da se kompletan paket modula učita jednom naredbom. Za to je potrebno u fajlu `__init__.py` pripremiti *spisak modula* u posebnoj promenljivoj rezervisanog naziva `__all__`, npr.

```
__all__ = ["Bmp", "Jpeg", "Png", "Tiff", "Xpm"]
```

Za uvoz svih modula jednog paketa tada je dovoljna jedna naredba, npr.

```
from Slike import *
```

9.5.3 Upotreba modula

U kodu modula se na nivou glavnog programa može se izvršiti provera načina njegove upotrebe: da li se koristi kao zaseban program ili kao modul. Prilikom

učitavanja modula, interpreter postavlja vrednost atributa `__name__` na vrednost koja odgovara nazivu modula i izvršava sve naredbe modula, uključujući i naredbe na najvišem nivou, koje mogu biti namenjene inicijalizaciji modula, ali i naredbe koje *ne treba izvršiti* kada se određeni program koristi kao modul od strane drugih programa.

Za glavni program koji koristi modul vrednost atributa `__main__` je `"__main__"`, tako da je pogodno na najvišem nivou odvojiti kod modula i aplikativnog programa jednostavim ispitivanjem:

```
if __name__ == '__main__':
    <glavni program>
```

Na taj način se programski kod modula na osnovnom nivou neće izvršiti, jer nije namenjen inicijalizaciji modula.

9.6 Primer programa

Primer upotrebe modula u razvoju softvera je mala biblioteka grafičkih funkcija razvijenih korišćenjem postojećih funkcija modula `turtle`, koje se mogu višestruko upotrebljavati i pojednostaviti programiranje.

Modul `korisneTurtleFunkcije` sadrži nekoliko jednostavnih funkcija potrebnih u vektorskoj grafici za [1]:

- crtanje linije između dve zadane tačke,
- crtanje tačke ili ispis teksta na zadanim koordinatama,
- crtanje kružnice zadanog poluprečnika s centrom u zadanoj tački,
- crtanje pravougaonika zadane širine i visine s centrom u zadanoj tački.

Programska realizacija biblioteke funkcija je program koji treba sačuvati na fajlu s nazivom `korisneTurtleFunkcije.py`:

```

""" Modul korisneTurtleFunkcije.

    Funkcije za ctanje linije, tačke, kružnice,
    pravougaonika i ispis teksta pomoću funkcija
    iz modula Turtle.

"""
import turtle

# Crtanje linije od (x1, y1) do (x2, y2)
def drawLine(x1, y1, x2, y2):
    turtle.penup()      # podizanje pera
    turtle.goto(x1, y1)
    turtle.pendown()    # spuštanje pera
    turtle.goto(x2, y2)

# Ispis teksta od koordinata (x, y)
def writeText(s, x, y):
    turtle.penup()      # podizanje pera
    turtle.goto(x, y)
    turtle.pendown()    # spuštanje pera
    turtle.write(s)     # ispis teksta

# Crtanje tačke na koordinatama (x, y)
def drawPoint(x, y):
    turtle.penup()      # podizanje pera
    turtle.goto(x, y)
    turtle.pendown()    # spuštanje pera
    turtle.begin_fill() # početak popunjavanja
    turtle.circle(3)
    turtle.end_fill()   # popunjavanje oblika

# Crtanje kruga zadanog poluprečnika s centrom u(x,y)
def drawCircle(x=0, y=0, radius=10):
    turtle.penup()      # podizanje pera
    turtle.goto(x, y-radius)
    turtle.pendown()    # spuštanje peran
    turtle.circle(radius)

```

```
# Crtanje pravougaonika zadanih dimenzija od (x, y)
def drawRectangle(x=0, y=0, width=10, height=10):
    turtle.penup()          # podizanje pera
    turtle.goto(x + width/2, y + height/2)
    turtle.pendown()        # spuštanje pera
    turtle.right(90)
    turtle.forward(height)
    turtle.right(90)
    turtle.forward(width)
    turtle.right(90)
    turtle.forward(height)
    turtle.right(90)
    turtle.forward(width)
```

Na osnovu modula, s ovako definisanim funkcijama za crtanje, mogu se pisati kraći programi, kao što je npr. kratki testni program koji po jednom poziva svaku od funkcija modula korisneTurtleFunkcije:

```
""" Testni program za modul korisneTurtleFunkcije.

    Crta liniju, tačku, kružnicu, pravougaonik i
    ispisuje tekst pomoću funkcija iz modula
    korisneTurtleFunkcije.

"""

import turtle
from korisneTurtleFunkcije import *

# Crtanje linije između(-80,-80) i (80,80)
drawLine(-80, -80, 80, 80)

# Ispis teksta od (-40,-60)
writeText("Test korisnih Turtle funkcija", -40, -60)

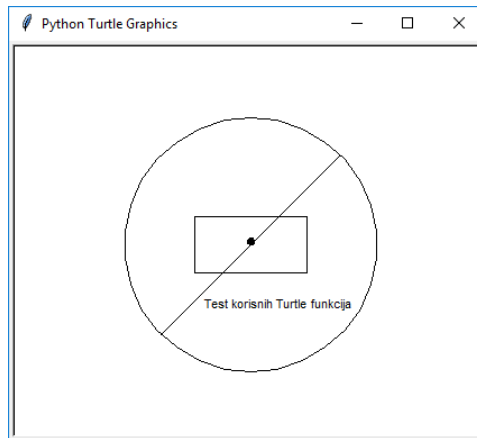
# Crtanje tačke u (0, 0)
drawPoint(0, 0)

# Crtanje kružnice poluprečnika 113 od (0,0)
drawCircle(0, 0, 113)
```

```
# Crtanje pravougaonika 100x50 od (0,0)
drawRectangle(0, 0, 100, 50)

turtle.hideturtle()
turtle.done()
```

Rezultat izvršavanja programa prikazuje se u grafičkom prozoru:



Pitanja za ponavljanje

1. Zbog čega se vrši podela obimnih programa na manje celine?
2. Navedite prednosti upotrebe modula u različitim fazama razvoja softvera.
3. Šta je modul u jeziku Python?
4. Šta znači izraz *docstring* u razvoju softvera u jeziku Python?
5. Objasnite način projektovanja softvera s vrha (*top-down*).
6. Kako se koristi modul u jeziku Python?
7. Na koji način sistem pronalazi modul koji treba uvesti?
8. Objasnite varijante uvoza modula u program u jeziku Python.
9. Šta su i čemu služe paketi (*packages*) u jeziku Python?
10. Kako se jednom naredbom može uvesti ceo paket modula?

10 Rad s fajlovima u jeziku Python

1. Uvod
2. Pristup fajlu
3. Učitavanje teksta
4. Učitavanje numeričkih podataka
5. Upis teksta na fajl
6. Strukturirani podaci u formatu JSON
7. Čitanje teksta s Interneta
8. Primeri programa

U ovom poglavlju se ukratko izlaže upotreba sekvencijalnih fajlova u jeziku Python za smeštanje i čitanje standardnih tipova podataka.

10.1 Uvod

Svetska računarska mreža omogućava pristup i deljenje informacija smeštenih na različitim serverima, najčešće u obliku *fajlova* na diskovima računara. Osim deljenja informacija, dele se i programi za upotrebu tih informacija [2]. Prema formatu zapisa i načinu pristupa sadržaju, razlikuju se tekstualni i binarni fajlovi.

Tekstualni fajlovi sadrže znakove strukturirane kao linije teksta. Osim znakova koji se mogu štampati, tekstualni fajlovi sadrže znakove koji nemaju vizuelnu interpretaciju, tako da se ne mogu štampati. Takav je npr. znak za kraj linije i prelazak u novu liniju teksta (*newline*, `\n`). Prikaz ovog znaka na ekranu računara premešta kursor na početak sledećeg reda ekrana. Dobra osobina tekstualnih fajlova je da se mogu kreirati pomoću običnog tekst editora.

Binarni fajlovi sadrže različite tipove podataka, npr. binarne vrednosti koje predstavljaju brojeve ili još složenije objekte, kao što su slike, audio i video zapisi. Binarni fajlovi imaju složeniju organizaciju i mogu se koristiti samo pomoću namenskih programa.

Osnovne operacije za sve tipove fajlova su pristup ili *otvaranje* fajla, *čitanje* fajla, *upis* na fajl i *zatvaranje* fajla [1], [2].

10.2 Pristup fajlu

Prvo je neophodno naredbom `open` kreirati objekt u memoriji koji je povezan s određenim fajlom:

```
<file> = open(<parametri>)
```

Navode se parametri u obliku teksta, kao što su npr.

- *putanja fajla*, koja može biti jednostavna ('primer.txt'), relativna ili apsolutna, npr. 'C:\\Users\\korisnik\\Documents\\Desktop\\primer.txt'. Umesto oznake za specijalni znak "\\" u putanji se može koristiti znak "/", npr. 'C:/Users/korisnik/Documents/Desktop/primer.txt';
- *način pristupa*, npr. čitanje ('r'), pisanje ('w') i dodavanje ('a') ili čitanje i pisanje binarnog zapisa ('rb', 'wb');
- *vrsta kodiranja*, npr. `encoding='utf-8'` je zapis po standardu Unicode, koji je kompatibilan s ASCII standardom i predstavlja W3C⁵ preporuku. Svaki znak predstavljen je pomoću 1 do 4 osmobarbajta (više od milion kombinacija).

Primer naredbe za otvaranje tekstualnog fajla `podaci.txt` samo radi čitanja je

```
ulaz = open("podaci.txt", "r")
```

Ukoliko se fajl ne nalazi na tekućem folderu, putanja se može zadati na jedan od uobičajenih načina, npr.

```
ulaz = open("c:/neki_folder/podaci.txt", "r")
```

10.3 Učitavanje teksta

Nakon otvaranja fajla radi čitanja, koriste se odgovarajući metodi za čitanje teksta i njegovo smeštanje u memoriju:

- metod `read(n)` koristi se za čitanje `n` znakova kao jednog stringa. Ako se broj znakova izostavi, čita se ceo sadržaj fajla kao jedan string;
- metod `readline()` koristi se a čitanje sledeće *linije* teksta kao stringa. Čitanje počinje od prve linije;
- metod `readlines()` koristi se za čitanje *svih linija* iz fajla, koji se u memoriji predstavljaju u obliku *liste* stringova.

⁵ World Wide Web Consortium (W3C) je međunarodna organizacija čiji je zadatak razvoj i izrada standarda Interneta.

Sledeći programski segment učitava ceo sadržaj tekstualnog fajla `podaci.txt` kao jedan string, uključujući i specijalne znakove `\n`, koji označavaju kraj svake od linija teksta:

```
f = open("podaci.txt", "r")
# Čitanje celog sadržaja fajla kao stringa, s \n
sadržaj = f.read()
f.close()
```

Ukoliko je fajl prevelik da se ceo učitava u memoriju i obradi, njegovi delovi (obično pojedinačne linije) čitaju se u petlji i redom obrađuju:

```
linija = f.readline()      # prva linija teksta
while line != "":
    # Obrada jedne linije teksta ...
    linija = f.readline()  # čitanje sledeće linije
```

Kraj fajla ustanovljava se proverom učitanoog sadržaja, jer `readline()` vraća prazan string kad više nema teksta.

Jezik Python omogućava čitanje i obradu podataka s fajla pomoću petlje `for`, tako što se koristi forma naredbe:

```
for linija in f:
    # Obrada jedne linije teksta ...
```

Ovakav način je pogodan za čitanje celog fajla, bez potrebe za ispitivanjem uslova za završetak petlje.

10.4 Učitavanje numeričkih podataka

Numerički podaci su na fajlu zapisani u obliku teksta, uglavnom niza cifara. Prilikom čitanja, tekstualni zapis broja potrebno je pretvoriti u binarni zapis brojeva u memoriji. Podaci na fajlu međusobno su razdvojeni nekim nenumeričkim znacima (delimiterima), kao što su npr. prazno mesto ili kraj linije `'\n'`. Upotreba standardnih delimitera omogućava automatizaciju podele učitanoog stringa na delove pomoću string funkcije `split()`.

Npr. izraz:

```
"100 200 300 400".split()
```

daje listu stringova:

```
['100', '200', '300', '400']
```

Svaki od stringova se zatim može konvertovati u odgovarajući tip podatka u memoriji pomoću ugrađene funkcije `eval`, npr.

```
# Otvaranje fajla i čitanje svih podataka
f = open("brojevi.txt", "r")      # lista brojeva
s = f.read()

# Podela stringa s na delove i konverzija s eval()
brojevi = [eval(x) for x in s.split()]
broj in brojevi:                  # obilazak liste brojeva
    print(broj, end = " ")       # prikaz u istom redu

f.close() # Zatvaranje fajla
```

Funkcija vrši konverziju zadanog stringa u numeričku vrednost odgovarajućeg tipa, npr. `eval('100')` vraća celobrojnu vrednost 100, a `eval('100.')` vraća decimalnu vrednost 100.0. Argument ove funkcije može biti i izraz, koji sadrži promenljive i zagrade. Npr. ako je vrednost promenljive `x=2`, poziv funkcije `eval('100+(x-1)')` kao rezultat evaluacije izraza daje celobrojnu vrednost 101.

10.5 Upis teksta na fajl

Upis brojeva na fajl naredbom `write()` vrši se nakon njihove konverzije u string, koji se sastoji od niza cifara i eventualnih dodatnih oznaka, kao što su predznak, decimalni znak i eksponent. Prilikom upisa, između ovih stringova cifara umeću se delimiteri, za odvajanje brojeva prilikom čitanja.

U sledećem primeru otvara se fajl radi upisa i na njega se upisuje niz pseudoslučajnih brojeva, koji se pre upisa konvertuju u stringove pomoću funkcije `str()`:

```
# Otvaranje fajla radi upisa podataka
f = open("brojevi.txt", "w")

# Zapis deset pseudoslučajnih brojeva na fajl
for i in range(10):
    broj = randint(0, 9)      # slučajni broj 0..9
    f.write(str(broj)+" ")    # zapis stringa broj+" "

f.close() # Zatvaranje fajla
```

Prilikom ovakve konverzije ispis vrednosti promenljivih vrši se u formatu koji određuje sam sistem. Format ispisa podataka u jeziku Python može se preciznije zadati korišćenjem više različitih metoda formatiranja. U ovom poglavlju prikazaće se način formatiranja pomoću *operatora formatiranja*, *funkcije* i *metoda format* i *formatiranih stringova* (f-stringova) [1], [2], [10].

10.5.1 Operator formatiranja

Formatiranje pomoću opratora formatiranja najstariji je način formatiranja (*old-string-formatting*), uveden još u u verziji 2 jezika, po ugledu na jezik C [11]. Za opis izlaza koristi se operator formatiranja % (modul), koji u jeziku Python ima dvostruku ulogu. Kad su oba operanda brojevi, operator vraća ostatak od deljenja prvog operanda drugim. Kada je prvi argument string, predstavlja operator formatiranja oblika:

```
<format> % <lista_vrednosti>
```

Prvi operand je string u kome su označena mesta umetanja vrednosti iz liste, a istovremeno i opisan njihov format, npr.

```
>>> "Heksadecimalni zapis broja %d je %X" % (77, 77)
'Heksadecimalni zapis broja 77 je 4D'
```

Decimalni prikaz celobrojnih vrednosti označen je specifikacijom '%d', a heksadecimalnih sa '%X'. Prikaz decimalnih vrednosti definiše pomoću specifikacije '%f', a način prikaza stringova sa '%s'. Osim tipa prikaza, mogu se navesti i dužine pojedinih elemenata prikaza ispred oznaka, npr.

```
>>> "Vrednost %d/%d je %5.3f" % (1, 3, 1/3)
'Vrednost 1/3 je 0.333'
```

Dužina prikaza decimalnih brojeva daje se pomoću dva podatka odvojena tačkom: ukupne dužine prikaza broja i dužine decimalnog dela.

10.5.2 Funkcija i metod format

Noviji pristup formatiranju koristi opis formata teksta pomoću ranije pomenute *funkcije* `format`, ili *metoda* `format` objekata tipa `string`.

Funkcija `format` omogućava formatiranje pojedinačnih objekata:

```
format(<podatak>, <specifikacija_formata>)
```

Funkcija formatira zadanu vrednost prema specifikaciji koja se navodi kao drugi argument, npr.

```
>>> format(1/3, "10.3f")
'      0.333'
```

U verziji 3 jezika Python uvedeno je formatiranje pomoći metoda `format` objekata tipa *string*, kojim se mogu opisati složeniji formati. Definiše se osnovni string u kome su velikim zagradama obeležena polja, mesta na koja se umeću vrednosti navedene kao argumenti funkcije `format()`, npr.

```
>>> "Vrednost {0}/{1} je {2}".format(1, 3, 1/3)
'Vrednost 1/3 je 0.3333333333333333'
```

Pojedina polja mogu se formatirati dodatnom specifikacijom formata, npr. broj decimalnih mesta iz prethodnog primera može se ograničiti na tri:

```
>>> "Vrednost {0}/{1} je {2:5.3f}".format(1, 3, 1/3)
'Vrednost 1/3 je 0.333'
```

Oznake koje se mogu koristiti u specifikacijama formatiranja za oznake tipa vrednosti, širine polja, znaka za popunu praznih mesta, poravnanja, predznaka, broja decimalnih mesta itd. navedni su *Prilogu C* udžbenika.

10.5.3 Formatirani stringovi

Najnovija verzija formatiranja, koja je uvedena u verziji 3.6 jezika [10] koristi *formatirane stringove*, kraće *f-stringove*. Formatirani stringovi imaju prefiks `'f'` ili `'F'` i osim teksta sadrže *izraze* zatvorene u velike zagrade, koji se *evaluiraju* u toku izvršavanja programa, npr.

```
>>> ime = "Milan"
>>> f"Moje ime je {ime!r}."
'Moje ime je 'Milan'.'
```

Oznaka `'!'` u izrazu označava da je potrebno izvršiti neku konverziju vrednosti, a `'r'` označava konverziju u string koji se može prikazati (štampati).

Formatiranje numeričkih podataka na ovaj način vrši se takođe pomoću umetnutih izraza, koji se mogu gnezditi, kao npr. širina i broj cifara u opisu formata decimalnog broja:

```
>>> sirina = 10
>>> cifara = 5
>>> broj = float("12.34567")
>>> f"rezultat: {broj:{sirina}.{cifara}}"
'rezultat:      12.346'
```

Pošto formatirani stringovi mogu da sadrže izraze, njihova potpuna sintaksa je dosta složena (videti [10]).

10.6 Strukturirani podaci u formatu JSON

Format zapisa podataka pomoću čitljivog teksta u formi rečnika razvijen je za jezik *JavaScript* kao otvoreni format JSON (*JavaScript Object Notation*), nezavisan od računara. Podrška za ovaj format zapisa i razmene podataka postoji u više programskih jezika, uključujući Python. Modul `json` deo je standardne biblioteke jezika Python.

Format JSON veoma je sličan strukturama rečnika i liste u jeziku Python, tako da je konverzija trivijalna. Npr. sledeći rečnik s ugnježdenim strukturama liste i rečnika veoma je sličan JSON strukturi podataka [12]:

```
>>> ime={'ime':'Petar', 'prezime':'Petrović'}
>>> rec = {'ime':ime, 'polozaj':['razvoj', 'menadžer'],
'starost':40.5}
>>> rec
{'ime': {'ime': 'Petar', 'prezime': 'Petrović'},
'polozaj': ['razvoj', 'menadžer'], 'starost': 40.5}
```

Proces prevođenja objekata jezika Python u podatke u fajlovima koji su u formatu JSON, kao i njihovo čitanje vrši se odgovarajućim metodima iz modula `json`. Npr. zapis prethodno kreirane strukture rečnika `rec` u tekstualni fajl u formatu JSON može se izvršiti naredbom:

```
>>> import json
>>> json.dump(rec, fp=open('podaci_json.txt', 'w'),
indent=4)
```

Sadržaj fajla se može učitati i prikazati naredbom:

```
>>> print(open('podaci_json.txt').read())
{
    "ime": {
        "ime": "Petar",
        "prezime": "Petrovi\u0107"
    },
    "polozaj": [
        "razvoj",
        "menad\u017eer"
    ],
    "starost": 40.5
}
```

Potpuno tekstualni format nezavisan je i od sistema kodiranja znakova, tako da su znakovi koji ne pripadaju ASCII skupu označeni posebnim kodom.

Učitana struktura se prevodi u strukturu rečnika jezika Python naredbom:

```
>>> P = json.load(open('podaci_json.txt'))
>>> P
{'ime': {'ime': 'Petar', 'prezime': 'Petrović'},
 'polozaj': ['razvoj', 'menadžer'], 'starost': 40.5}
```

Vidi se da je kreirana odgovarajuća struktura u memoriji, u kojoj je prikaz znakova po standardu Unicode.

Slična podrška u jeziku Python postoji i za rad s podacima koji su zapisani po standardu XML [12].

10.7 Čitanje teksta s Interneta

Jezik Python omogućava čitanje teksta fajlova koji se nalaze na serverima mreže Internet. Za pristup ovim fajlovima potrebno je koristiti metod `urlopen` iz modula `urllib.request`.

Prilikom njihovog otvaranja, putanja fajla je njegova URL adresa, npr.

```
>>> import urllib.request
>>> f= urllib.request.urlopen
("https://docs.python.org/3/index.html")
>>> htmltekst = f.read().decode()
>>> print(htmltekst[0:123])
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0
Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">
```

Funkcija `read()` koristi metod `decode()` za konverziju teksta učitano iz fajla u string.

Podaci se između Veb servera često razmenjuju u formatu JSON. Npr. podaci o aktuelnim koeficijentima konverzije valuta sa sajta <http://fixer.io/> dobijaju se upitom, pomoću funkcije `read()` iz modula `urllib`, u odnosu na zadanu osnovnu valutu:

```
URL = 'http://api.fixer.io/latest?base=' + <o_valuta>
odgovor = urllib.request.urlopen(URL).read()
```

Odgovor se konvertuje u rečnik jednim pozivom funkcije iz modula `json`:

```
recnik = json.loads(odgovor)
```

Odgovor ima formu hijerarhijskog rečnika, gde su na najvišem nivou tri elementa s ključevima: "base", "date" i "rates". Elementi "base" i "date" su stringovi: naziv osnovne valute i datum važenja. Treći element "rates" je ugnježdjeni rečnik, u kome je svakoj valuti (ključu) pridružen faktor konverzije iz osnovne valute (base). Npr. ako se kao osnovna valuta navede "USD", dobija se rečnik:

```
{ "base": "USD", "date": "2016-05-
13", "rates": { "AUD": 1.3728, "BGN": 1.7235, "BRL": 3.486, "CAD":
1.2882, "CHF": 0.97145, "CNY": 6.5211, "CZK": 23.811, "DKK":
6.556, "GBP": 0.69403, "HKD": 7.7632, "HRK": 6.6109, "HUF": 277
.73, "IDR": 13318.0, "ILS": 3.7737, "INR": 66.78, "JPY": 108.88
, "KRW": 1172.8, "MXN": 18.05, "MYR": 4.0299, "NOK": 8.1669, "NZ
D": 1.4709, "PHP": 46.645, "PLN": 3.8711, "RON": 3.9633, "RUB":
65.271, "SEK": 8.2204, "SGD": 1.3705, "THB": 35.43, "TRY": 2.96
43, "ZAR": 15.168, "EUR": 0.88121 } }
```

Odgovaarjućem koeficijentu konverzije valuta u rečniku pristupa se na osnovu *ključa*, koji predstavlja skraćenicu valute u koju se vrši konverzija:

```
recnik1 = recnik['rates']
koeficijent = recnik1['EUR']
```

10.8 Primeri programa

U ovom poglavlju rad s tekstualnim fajlovima ilustruje se kratkim programom koji ustanovljava ukupni broj pojavljivanja različitih slova abecede u zadanom fajlu i programom za konverzije valuta, koji koristi JSON format podataka.

10.8.1 Brojanje različitih slova u tekstu fajla

Program na početku traži od korisnika naziv tekstualnog fajla u kome treba ustanoviti broj pojavljivanja svakog od slova engleskog alfabeta, bez obzira da li su slova velika ili mala.

Osnovni algoritam je:

1. Učitati ime fajla na kome je tekst, koje unosi korisnik;
2. Tekst se učitava u memoriju računara, a svako učitano slovo pretvara se u malo slovo pomoću string funkcije `lower()`;
3. Kreira se lista brojača za 26 znakova engleskog alfabeta `[0]..[25]`;
4. Za svaki znak u stringu, čiji kod predstavlja malo slovo u ASCII tabeli, odgovarajući brojač u listi povećava se za 1;
5. Na kraju se prikazuje spisak onih slova čiji je broj pojavljivanja u tekstu fajla veći od nule.

Program u jeziku Python koji realizuje ovaj algoritam je [1]:

```
""" Program broji različita slova u tekstualnom fajlu
    i prikazuje rezultat.

    Broje se samo slova iz ASCII podskupa,
    bez obzira da li su velika ili mala.

    """
# Brojanje različitih slova u stringu (ASCII podskup)
def brojacSlova(linija, brojac):
    for ch in linija:
        if ch.isalpha():
            brojac[ord(ch) - ord('a')] =
                brojac[ord(ch) - ord('a')] + 1

filename = input("Unesite naziv fajla: ").strip()
infile = open(filename, "r") # Otv. fajla za čitanje
```

```

brojac = 26 * [0] # Inicijalizacija brojača slova
for linija in infile:
    # Poziv funkcije brojacSlova za za svaku liniju
    brojacSlova(linija.lower(), brojac)

# Prikaz rezultata
for i in range(len(brojac)):
    if brojac[i] != 0:
        print(chr(ord('a') + i) + " se pojavljuje "
              + str(brojac[i])
              + (" put" if brojac[i]==1 else " puta"))

infile.close() # Zatvaranje fajla

```

Sadržaj fajla `test.txt` je:

```

ABCDE
abcde
0123456789

```

Rezultat izvršavanja programa nakon čitanja fajla `test.txt` je:

```

Unesite naziv tekstualnog fajla (ASCII): test.txt
a se pojavljuje 2 puta
b se pojavljuje 2 puta
c se pojavljuje 2 puta
d se pojavljuje 2 puta
e se pojavljuje 2 puta
>>>

```

10.8.2 Program za konverziju valuta

Program za konverziju valuta [13] računa faktor konverzije između dve izabrane valute na osnovu njihovog aktuelnog kursa, koji se preuzima na Veb sajtu <http://fixer.io/> u JSON formatu. Program ilustruje upotrebu modula `urllib` i rad s podacima u JSON formatu.

Pošto je besplatna samo aktuelna lista koeficijenata za konverziju iz EUR u ostale svetske valute, u ovom primeru je osnovna valuta EUR.

Osnovni algoritam programa za konverziju valuta je:

1. Korisnik zadaje zvaničnu skraćenicu *osnovne* valute (EUR) i skraćenicu ciljne valute za koju se traži koeficijent konverzije.

2. Program pronalazi koeficijente konverzije za osnovnu valutu na Veb sajtu <http://fixer.io> u JSON formatu i konvertuje ih u rečnik.
3. Program po skraćenici u rečniku pronalazi koeficijent konverzije i prikazuje ga, a zatim učitava iznos za konverziju iz osnovne valute u ciljnu i prikazuje iznos u ciljnoj valuti [13]

Program u jeziku Python, koji realizuje ovaj algoritam je:

```

""" Računanje faktora konverzije iz jedne u drugu valutu

    Koeficijenti konverzije sa sajta: http://data.fixer.io
    - za korišćenje podataka potrebno je obezbediti API KEY
    - besplatni su samo koeficijenti za osnovnu valutu EUR
"""
import urllib.request
import json
import sys

# Funkcija preuzima spisak valuta i vraća listu skraćenica
def listaSkracenicaValuta():
    URL =
    http://data.fixer.io/api/symbols?access_key=cb11f1ed214add8
    ad136c2af30b315dd'
    # Odgovor servera konvertuje se u rečnik
    odgovor = urllib.request.urlopen(URL).read()
    rečnik = json.loads(odgovor)
    lista = [sk for sk in rečnik['symbols'] if sk!="EUR"]
    return lista

# Funkcija preuzima i koristi podatke kao string
def getInfo(izValute, uValutu):
    URL =
    'http://data.fixer.io/api/latest?access_key=cb11f1ed214add8
    ad136c2af30b315dd'

    # Odgovor servera konvertuje se u rečnik
    odgovor = urllib.request.urlopen(URL).read()
    rečnik = json.loads(odgovor)

    # Izdvajanje dela stringa koji sadrži vrednost
    rečnik = rečnik['rates']
    faktor = rečnik[uValutu]

    return float(faktor)

```

```

def ucitajValutu(pitanje):
    while True:
        valuta = input(pitanje)
        valuta = valuta.upper() # oznaka u velika slova
        if valuta == '':
            sys.exit()
        if valuta in listaValuta:
            break
        else:
            print('Nažalost', valuta, 'nije u listi.')
    return valuta

print("Program računa koeficijent konverzije")
print("između dve od navedenih valuta:")
listaValuta = listaSkracenicaValuta() # lista skr. bez EUR
skracenice = ' '.join(listaValuta) # string skr. valuta
print(skracenice)
izValute = 'EUR'

# Izbor ciljne valute
uValutu = ucitajValutu('\nIzaberite ciljnu valutu: ')

# Faktor konverzije
faktorKonverzije = getInfo(izValute, uValutu)
print('\nKoeficijet konverzije', izValute, 'u', uValutu, \
      'je:', faktorKonverzije)

# Konverzija traženog iznosa
iznos = float( \
    input('\nUnesite iznos za konverziju (EUR): '))
print("Iznos u " + uValutu + " je", iznos*faktorKonverzije)
print()

```

Primer izvršavanja programa prilikom konverzije EUR u dinare (RSD) je:

```

Program računa faktor konverzije između EUR
i jedne od navedenih valuta:
AED AFN ALL AMD ANG AOA ARS AUD AWG AZN BAM BBD BDT BGN
BHD BIF   BMD BND BOB BRL BSD BTC BTN BWP BYN BYR BZD
CAD CDF CHF CLF CLP CNY COP CRC CUC CUP CVE CZK DJF DKK
DOP DZD EGP ERN ETB FJD FKP GBP GEL GGP GHS GIP GMD GNF
GTQ GYD HKD HNL HRK HTG HUF IDR ILS IMP INR IQD IRR ISK
JEP JMD JOD JPY KES KGS KHR KMF KPW KRW KWD KYD KZT LAK
LBP LKR LRD LSL LTL LVL LYD MAD MDL MGA MKD MMK MNT MOP
MRO MUR MVR MWK MXN MYR MZN NAD NGN NIO NOK NPR NZD OMR
PAB PEN PGK PHP PKR PLN PYG QAR RON RSD RUB RWF SAR SBD

```

```
SCR SDG SEK SGD SHP SLL SOS SRD STD SVC SYP SZL THB TJS
TMT TND TOP TRY TTD TWD TZS UAH UGX USD UYU UZS VEF VND
VUV WST XAF XAG XAU XCD XDR XOF XPF YER ZAR ZMK ZMW ZWL
```

Izaberite ciljnu valutu: **rsd**

Koeficijet konverzije EUR u RSD je: 117.558599

Unesite iznos az konverziju (EUR): 200

Iznos u RSD je 23511.7198

Pitanja za ponavljanje

1. Koja je osnovna razlika između binarnih i tekstualnih fajlova?
2. Objasnite način i parametre pristupa tekstualnom fajlu u jeziku Python.
3. Navedite metode čitanja teksta iz tekstualnog fajla. Kako se ustanovljava kraj podataka u tekstualnom fajlu?
4. Da li se podaci iz tekstualnog fajla mogu čitati petljom for?
5. Objasnite način čitanja numeričkih podataka iz tekstualnih fajlova i način upisa u tekstualne fajlove.
6. Navedite načine formatiranja podataka u jeziku Python.
7. Objasnite upotrebu operatora formatiranja.
8. Objasnite upotrebu funkcije i string metoda format.
9. Šta su f-stringovi?
10. Šta označava skraćenica JSON?

11 Analiza algoritama, pretraživanje i sortiranje u jeziku Python

1. Uvod
2. Složenost algoritama
3. Algoritmi pretraživanja
4. Algoritmi sortiranja
5. Primeri programa

U ovom poglavlju ukratko se izlaže problem analize algoritama i daju primeri osnovnih algoritama pretraživanja i sortiranja.

11.1 Uvod

Analiza algoritama je oblast računarskih nauka koja proučava performanse algoritama, posebno vreme izvršavanja i memorijske zahteve [11]. Predstavlja važan deo teorije složenosti (*complexity theory*), koja daje teorijske ocene resursa koji su potrebni bilo kom algoritmu koji rešava posmatrani računarski problem. Takve ocene omogućavaju bolji uvid u svojstva algoritama, usmeravaju razvoj efikasnih algoritama i omogućavaju međusobno poređenje različitih algoritama. Primer jednostavnog algoritma realizovanog u jeziku Python, čije performanse zavise od unesenih podataka je:

```
n = int(input("Unesite celi broj > 0: "))
for i in range(1, n+1):
    for j in range(1, i+1):
        print(i * j)
print("Kraj!")
```

Izvršavaju programa za unesenu vrednost $n=4$ daje rezultat:

```
Unesite celi broj > 0: 4
1
2
4
3
6
9
4
8
12
```

16

Kraj!

Broj koraka prilikom izvršavanja ovog programa zavisi od vrednosti n :

<u>n</u>	<u>broj koraka</u>
10	55
100	5.050
1.000	500.500

Vreme izvršavanja programa proporcionalno je broju podataka koji se obrađuje. Ukupan broj množenja i prikazivanja rezultata je $n \cdot (n+1)/2$, pa se uzima da je vremenska složenost algoritma reda n^2 .

U ovoj temi će se, kao primeri važnih kategorija algoritama, prikazati neki od jednostavnijih algoritama neophodnih za rešavanje opštih i veoma čestih problema programiranja: *pretraživanja* i *sortiranja* podataka u memoriji.

Neformalno se izlažu osnovne ideje algoritama i daju primeri njihove realizacije u obliku funkcija u jeziku Python.

Složeniji algoritmi pretraživanja i sortiranja, kao i algoritmi sortiranja podataka u permanentnoj memoriji izlaze iz okvira uvodnog kursa programiranja i izučavaju se u odgovarajućim stručnim predmetima.

11.2 Složenost algoritama

Složenost algoritma je ocena potrebnih resursa za rešavanje određenog računarskog problema [11], [19].

Najčešće se razmatraju *vreme izvršavanja* i potrebna *veličina memorije* za rešavanje nekog problema. Pošto se algoritmi dizajniraju za obradu bilo kog obima podataka, njihove performanse se obično izražavaju nekom funkcijom složenosti koja povezuje obim podataka s potrebnim brojem koraka algoritma (*vremenska složenost*) i potrebnom veličinom memorije (*prostorna složenost*).

Prilikom analize i poređenja različitih algoritama neophodno je isključiti uticaj hardvera, nekih svojstava samih podataka i veličine (obima) problema koji se rešava:

- *Hardver*: Neki algoritmi imaju bolje performanse od drugih na jednoj vrsti računarskih sistema, a slabije na drugoj vrsti. Zbog toga se u analizi svih razmatranih algoritama koristi *apstraktni model računara*, npr. Turingova mašina;

- *Osobine samih podataka*: Performanse algoritama mogu da zavise i od osobina samih podataka, npr. ako se koriste delom već sortirani podaci, za koje neki algoritmi sortiranja pokazuju bolje, a drugi slabije performanse. Zbog toga se u analizama najčešće razmatra ponašanje algoritama u *najgorem slučaju* (*worst case*);
- *Obim podataka*: Algoritmi često imaju različite performanse za različiti obim ulaznih podataka. Npr. neki algoritmi sortiranja su brži za mali obim podataka, dok su za veći obim podataka relativno sporiji od drugih, dok su neki drugi algoritmi sortiranja brži od ostalih tek za veliki obim podataka. Zbog toga se vreme izvršavanja izražava brojem operacija koji *funkcionalno* zavisi od veličine problema, a prilikom poređenja se razmatra *asimptotsko* ponašanje ovih funkcija.

11.2.1 Ocena složenosti algoritama

Složenost algoritama na osnovu asimptotskog ponašanja funkcije složenosti algoritma izražava se notacijom

 $O(\text{funkcija rasta})$

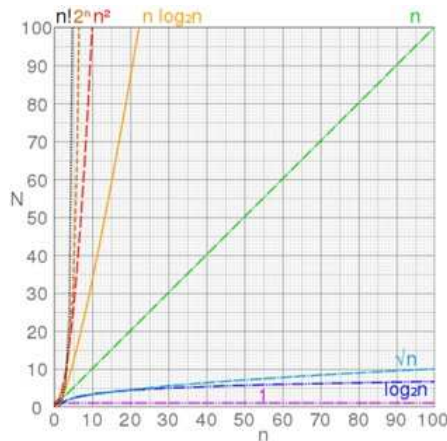
gde je O skraćeno od "*Order of magnitude*". Najčešće se razmatraju sledeće funkcije rasta:

$O(1)$	konstanta
$O(\log_b n)$	logaritamska (bilo koja baza b)
$O(n)$	linearna (polinomska)
$O(n \log_b n)$	tzv. " $n \log n$ " funkcija
$O(n^2)$	kvadratna (polinomska)
$O(n^3)$	kubna (polinomska)
$O(c^n)$	eksponencijalna (za bilo koji c)
$O(n!)$	faktorijelska

Na Sl. 38 [19] je grafički prikaz navedenih funkcija rasta za male vrednosti n . Linearna, kvadratna i druge funkcije rasta reda n^c nazivaju se *polinomskim* funkcijama rasta. U analizi algoritama razmatraju se i druge funkcije rasta, osim funkcija navedenih u ovom kratkom pregledu.

Brzina rasta nekih od prikazanih funkcija vrlo brzo uvećava potreban broj operacija algoritma, koji utiče na brzo povećanje vremena izvršavanja stvarnih programa, npr.

- 10^{12} operacija, koje na nekom računarju traju po jednu milisekundu, izvršavalo bi se više od 30 godina;
- faktorijska složenost algoritma već za $n=64$ zahteva broj operacija reda $64!$, što je broj veći od 10^{80} , procenjeni broj atoma u poznatom, odnosno vidljivom svemiru.



Sl. 38. Primeri funkcija rasta

11.2.2 Poređenje vremenske složenosti algoritama

Poređenje vremenske složenosti algoritama vrši se na osnovu asimptotskih ocena broja operacija u odnosu na obim ulaznih podataka. Za poređenje je najvažniji deo funkcije složenosti s *najvećim rastom*. Npr. algoritmi čija složenost *linearно* zavisi od ulaznih podataka, kao što je funkcija rasta $a \cdot n + b$ (za konstante a i b), uvek su brži od *eksponencijalnih* algoritama za dovoljno veliko n .

U analizi algoritama smatra se da su funkcije rasta koje pokazuju isto asimptotsko ponašanje *ekvivalentne*, pa je za njihovo puno poređenje potrebno razmotriti i ostale elemente. Npr. funkcije složenosti $2n$, $100n$ i $n+1$ spadaju u istu kategoriju (linearna složenost).

Ako jedan algoritam za obradu n podataka zahteva $100 \cdot n + 1$ koraka, a drugi algoritam $n^2 + n + 1$, njihovo vreme izvršavanja je kao u tabeli [11]:

n	Vreme 1: $100*n+1$	Vreme 2: n^2+n+1
10	1.001	111
100	10.001	10.111
1.000	100.001	1.001.001
10.000	1.001.001	$>10^{10}$

Vidi se da je za mali obim podataka ($n < 100$) bolji drugi algoritam, dok je za veći obim podataka ($n > 100$), kao i u opštem slučaju, znatno bolji prvi algoritam. Uočava se još da vreme izvršavanja algoritama čija se složenost izražava eksponencijalnom funkcijom reda n^2 raste veoma brzo s obimom podataka n .

Pošto je vremenska složenost algoritama izražena funkcijama iz različitih kategorija, dovoljno je uporediti najvažnije faktore: n prema n^2 .

11.3 Algoritmi pretraživanja

Pretraživanje liste je proces pronalaženja nekog zadanog elementa u listi. Npr. ugrađena funkcija `index()` vrši pretraživanje liste i kao rezultat vraća indeks prvog pronađenog elementa:

```
i = lista.index(element)
```

Vreme pronalaženja elementa proporcionalno je dužini liste, tako da linearno pretraživanje lista s velikim brojem elemenata može biti sporo.

Pretraživanje je opšti zadatak u programiranju, pa su razvijeni različiti metodi efikasnijeg pretraživanja lista. Npr. traženi element se može brže pronaći ako je lista prethodno sortirana.

11.3.1 Linearno pretraživanje liste

Prilikom lineranog pretraživanja, sekvencijalno se (u petlji) poredi svaki element liste s traženim elementom. Pretraživanje se zaustavlja kad se element pronađe ili se dođe do kraja liste. Ako je element pronađen, rezultat je njegov indeks u listi, inače se vraća vrednost -1 (koja ne može biti indeks). Funkcija linearnog pretraživanja u jeziku Python može da ima sledeći oblik [1]:

```
# Funkcija za pronalaženje elementa u listi
def linearSearch(lista, element):
    for i in range(len(lista)):
        if element == lista[i]:
            return i
    return -1
```

11.3.2 Binarno pretraživanje sortirane liste

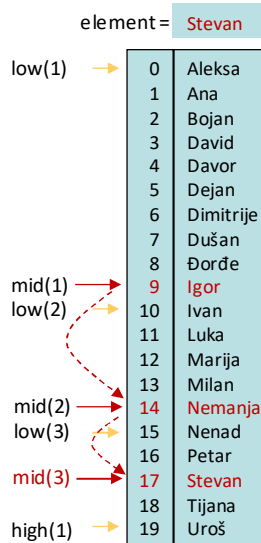
Binarno pretraživanje je često korišćen metod za pronalaženje elementa u listi koja je prethodno sortirana, najčešće u rastućem poretku vrednosti elemenata.

Pretraživanje počinje poređenjem vrednosti traženog elementa s elementom koji je u sredini liste. Ako su vrednosti elemenata jednake, element je pronađen. Inače, ako je vrednost manja od vrednosti srednjeg elementa, pretraživanje se nastavlja u prvoj polovini liste, a ako je veća u drugoj polovini liste na isti način, rekursivno ili inkrementalno.

Funkcija binarnog pretraživanja u jeziku Python može da ima sledeći oblik [1]:

```
# Funkcija za pronalaženje elementa sortirane liste
def binarySearch(lista, element):
    low = 0
    high = len(lista) - 1
    while high >= low:
        mid = (low + high) // 2
        if element < lista[mid]:
            high = mid - 1
        elif element == lista[mid]:
            return mid
        else:
            low = mid + 1
    return -1 # element nije pronađen
```

Na Sl. 39 prikazan je primer pretraživanja liste od 19 elemenata.



Sl. 39. Primer binarnog pretraživanja sortirane liste

Linearnim pretraživanjem element 'Stevan' se pronalazi u 18 koraka (ispitivanja), dok se binarnim pretraživanjem pronalazi u samo 3 koraka.

11.4 Algoritmi sortiranja

Sortiranje elemenata liste po nekom principu opšti je zadatak u programiranju, često je važan deo nekog dugog algoritma. Razvijen je veliki broj metoda sortiranja za različite situacije. Poznati efikasni metodi sortiranja su npr. *Quick Sort*, *Heap Sort* i *Radix Sort* [1], [4], [14], [15], [16], [17].

U najjednostavnije metode spadaju sortiranje selekcijom (*Selection Sort*), sortiranje umetanjem (*Insertion Sort*) i sortiranje mehurom (*Bubble sort*), koji se ukratko razmatraju u ovoj temi. Neformalno se izlažu osnovni algoritmi i daje primer njihove realizacije u obliku funkcija u jeziku Python.

11.4.1 Sortiranje selekcijom (Selection Sort)

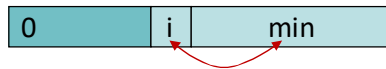
Ako se sortiranje vrši u rastućem redosledu, metod sortiranja selekcijom (*Insertion Sort*) pronalazi *najmanji* element u listi i menja ga s prvim elementom. Sortiranje u opadajućem redosledu traži *najveći* element.

Postupak pronalaženja najmanjeg elementa i zamene s prvim ponavlja se za ostatak liste, sve dok ne preostane samo jedan element.

Opšti oblik metoda sortiranja selekcijom je [1]:

```
for i in range(len(lista)-1):
    # izabere se najmanji element iz [i:len(lista)]
    # ako je potrebno, element se zameni s lista[i]
```

Na kraju svake iteracije element `lista[i]` je na konačnoj poziciji, Sl. 40. Sledeća iteracija se primenjuje na ostatak liste `[i+1:len(lista)]`.



Sl. 40. Princip sortiranja liste selekcijom

Funkcija za sortiranje elemenata liste selekcijom u jeziku Python [1]:

```
# Funkcija sortiranja elemenata u rastućem poretku

def selectionSort(lista):
    for i in range(len(lista) - 1):
        # Pronađe se najmanji element u[i:len(lista)]
        tekuciMin = lista[i]
        tekuciMinInd = i
        for j in range(i + 1, len(lista)): #selekcija
            if tekuciMin > lista[j]:
                tekuciMin = lista[j]
                tekuciMinInd = j
        # Zamene se elementi [i] i [tekuciMinInd],
        # po potrebi
        if tekuciMinInd != i:
            lista[tekuciMinInd] = lista[i]
            lista[i] = tekuciMin
```

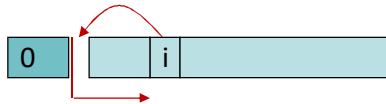
11.4.2 Sortiranje umetanjem (Insertion Sort)

Ako se sortiranje vrši u rastućem redosledu, algoritam sortiranja umetanjem (*Insertion Sort*) sortira listu vrednosti ponavljanjem umetanja novog elementa u sortiranu podlistu, sve dok se ne sortira cela lista.

Ošti opis metoda sortiranja umetanjem je [1]:

```
for i in range(1, len(lista)):
    # Umetne se element [i] u sortiranu
    # podlistu [0:i] tako da je lista[0:i+1] sortirana
```

Metod na svakom koraku pronalazi mesto tekućeg elementa i u do tada sortiranom delu liste i umeće ga tako što pomera prema kraju liste sve elemente koji su iza njega, Sl. 41.



Sl. 41. Princip sortiranja liste umetanjem

Funkcija za sortiranje elemenata liste umetanjem u jeziku Python može da ima sledeći oblik [1]:

```
# Funkcija sortiranja elemenata u rastućem poretku

def insertionSort(lista):
    for i in range(1, len(lista)):
        # Umetne se element [i] u sortiranu podlistu
        # [0:i] tako da lista[0:i+1] bude sortirana
        tekuciElement = lista[i]
        k = i - 1
        while k >= 0 and lista[k] > tekuciElement:
            lista[k+1] = lista[k]
            k = k - 1
        # Umetne se tekući element u lista[k+1]
        lista[k+1] = tekuciElement
```

11.4.3 Sortiranje mehurom (Bubble Sort)

Algoritam sortiranja mehurom (*Bubble Sort*) takođe je jednostavan, iako neefikasan algoritam za sortiranje lista. Metod poredi susedne elemente liste i, ako nisu u traženom poretku, međusobno im zameni mesta. Na taj način male vrednosti, kroz niz zamena mesta, kao mehurići plove ka vrhu liste, dok veće vrednosti padaju ka dnu liste [15].

Ošti opis metoda je, za sortiranje u rastućem poretku:

```
for i in range(n-1):
    # Protok (bubble) najvećeg elementa na kraj liste
    for j in range((n-1)-i):
        # ako elementi j i j+1 nisu u traženom poretku
        # zameniti elemente j i j+1
```

Nakon prvog prolaza, poslednji element je na ispravnoj poziciji. Nakon drugog prolaza, dva poslednja elementa su na ispravnoj poziciji, itd. Nakon svakog prolaza kroz listu nesortirani deo liste kraći je za jedan element.

Za listu L od n elemenata, ovaj algoritam prolazi kroz celu listu $n-1$ puta i poredi i -ti element liste s narednim elementom. Ako elementi i i $i+1$ nisu u traženom redosledu (rastući ili opadajući), međusobno zamenjuju mesta. Složenost algoritma je $O(n^2)$, jer se izvrši u $n \cdot (n-1)/2 + n$ koraka (poređenja) [15].

Funkcija za sortiranje elemenata liste mehurom u jeziku Python može da ima sledeći oblik [15]:

```
# Funkcija sortiranja elemenata u rastućem poretku

def bubbleSort(lista):
    n = len(lista)
    for i in range(n-1):
        # Protok (bubble) najvećeg elementa na kraj
        for i in range(n-1-i):
            if lista[j] > lista[j+1]:
                temp = lista[j]
                lista[j] = lista[j+1]
                lista[j+1] = temp
```

Treba napomenuti da se u jeziku Python zamena vrednosti dveju promenljivih može napisati pomoću samo jedne naredbe:

```
lista[j], lista[j+1] = lista[j+1], lista[j]
```

U realizacijama prethodna tri algoritama sortiranja zamena vrednosti elemenata liste ipak je izvršena pomoću tri naredbe, jer je takvo rešenje uobičajeno u većini proceduralnih programskih jezika, pa je programski kod potencijalno jasniji širem skupu programera.

11.5 Primeri programa

U ovom poglavlju će se kao praktični primeri programa, koji objedinjavaju materijal nekoliko prethodnih tema, izložiti program za kreiranje indeksa pojmova u zadanom tekstu (*glossary*) i program za poređenje performansi struktura liste i skupa.

11.5.1 Program Indeks pojmova

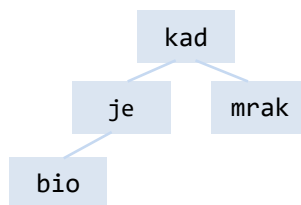
Indeks pojmova gradi sortiranu listu svih reči u zadanom tekstu, koji se učitava iz tekstualnog fajla, zajedno s pozicijama njihovog pojavljivanja:

1. Program prvo učitava *naziv* tekstualnog fajla u kome se nalazi tekst koji treba analizirati;
2. Pomoću rekurzivne procedure, program redom čita linije fajla i gradi binarno stablo, u koje se smeštaju učitane reči.

Nove reči dodaju se zavisno od njihovog *alfabetskog poretka* u odnosu na reč u tekućem čvoru binarnog stabla: u *levo* podstablo, ako je nova reč ispred, ili u *desno* podstablo, ako je nova reč iza reči u tekućem čvoru.

Npr. čitanjem teksta "kad je bio mrak", dobija se binarno stablo kao na Sl. 42. Obilazak stabla prema unutrašnjem uređenju (*inorder*) daje sortirani niz reči "bio je kad mrak".

Kad se prilikom učitavanja teksta kreira čvor za novu reč ili se posmatrana reč pronađe u stablu, u tom čvoru stabla se ažurira lista brojeva linija dodavanjem rednog broja tekuće linije teksta.



Sl. 42. Primer binarnog stabla sortiranja za deo ulaznog teksta

3. Nakon izgradnje stabla, program prikazuje ukupni broj učitanih linija i, procedurom obilaska stabla u unutrašnjem poretku, alfabetski sortiranu listu svih reči, zajedno s listom brojeva linija teksta u kojima se svaka od prikazanih reči pojavljuje.

Program za kreiranje liste pojmova u jeziku Python, gde se pojavljivanja pojmova prikazuju po linijama teksta [31], zasniva se na predstavljanju binarnog stabla pomoću ugnježđenih lista sledeće strukture:

```
[<reč>, <lista_levo_podstablo>, <lista_desno_podstablo>]
```

Program je realizovan tako da se tekst fajla čita funkcijom `binaryTree`, jedna po jedna linija teksta. Učitani string jedne linije teksta deli se na reči ugrađenom funkcijom `split()`.

Nakon uklanjanja specijalnih znakova i znakova interpunkcije ugrađenom funkcijom `strip`, reč `w` se dodaje u binarno stablo na odgovarajuće mesto funkcijom `treeInsert`, uz dodavanje broja linije `n` u kojoj se reč pojavila.

Obilazak stabla u unutrašnjem poretku vrši se rekurzivnom funkcijom `inorderTraversal`, koja vrši ispis sortirane tabele reči analiziranog teksta sa spiskom linija u kojima se svaka od reči pojavljuje.

Program se može izmeniti, tako da kao indikator položaja reči ne koristi redosled *linija* teksta, već neki drugi podatak, npr. redosled grupa linija koje predstavljaju jednu *stranicu* teksta.

Kompletan kod programa za izradu indeksa pojmova, odnosno sortirane liste svih reči u zadanom tekstu je [31]:

```
"""Kreiranje liste pojmova u tekstualnom fajlu

Gradi se i prikazuje lista reči od kojih se sastoji
tekst u zadanom fajlu. Za svaku reč prikazuju se
brojevi svih linija teksta u kojima se reč pominje.

"""
import sys

num = 0          # globalni brojač linija
word_count = 0   # globalni brojač reči
```

```

def treeInsert(s, w, n):
    if s == []: # inicijalizacija stabla (prvi put)
        s.append([]) # za levu granu
        s.append([])
        s.append([])
        s[0] = (w, [n])
    else:
        while True:
            if w == s[0][0]: # reč već postoji u stablu
                if not n in s[0][1]:
                    s[0][1].insert(len(s[0][1]), n)
                return
            elif w < s[0][0]: # spada u levo podstablo
                if s[1] == []: # reč ne postoji, insert
                    s[1] = [(w, [n]), [], []]
                else:
                    s = s[1] # na levi
            elif w > s[0][0]: # spada u desno podstablo
                if s[2] == []: # reč ne postoji, insert
                    s[2] = [(w, [n]), [], []]
                else:
                    s = s[2] # u desno podstablo

def inorderTraversal(s):
    global word_count
    if (s[1] != []):
        inorderTraversal(s[1])
    word_count += 1
    sys.stdout.write("%4i" % word_count) + " " + s[0][0])
    for i in range(0, 17 - len(s[0][0])):
        sys.stdout.write(' ')
    for i in s[0][1]:
        sys.stdout.write(str(i) + " ")
    sys.stdout.write('\n')
    if (s[2] != []):
        inorderTraversal(s[2])

```

```

def binaryTree(filename):
    global num
    f = open(filename, 'r', encoding='utf-8') # pristup fajlu
    r = []
    for line in f:
        num += 1
        wp = line.split()
        for w in wp:
            w = w.strip(".,;?!-\\n").lower()
            if len(w) > 0:
                treeInsert(r, w, num)
    return r

tekst = input("Unesite ime fajla s tekstem: ")
stablo = binaryTree(tekst)

print("Broj linija teksta fajla",
      '{}={:d}'.format(tekst, num))
print("Indeks pojmova:")
print("      Reč          Linije")
inorderTraversal(stablo)

```

U testnom primeru se analizira tekst poznate dečije pesme Dušana Radovića:

```

Kad je bio mrak,
kad je bio mrak,
pojurla mačka miša
čak, čak, čak.
Pojurla mačka miša
čak, čak, čak,
a da l' ga je progutala,
to ni ona nije znala
- jer je bio mrak,
jer je bio mrak...

```

Rezultat izvršavanja programa, nakon analize sadržaja fajla 'Dusko.txt', je:

```

Unesite ime fajla s tekстом: dusko.txt
Broj linija teksta fajla 'dusko.txt'=10
Indeks pojmov:
      Reč          Linije
1 a                7
2 bio              1 2 9 10
3 da               7
4 ga               7
5 je               1 2 7 9 10
6 jer              9 10
7 kad              1 2
8 l'               7
9 mačka            3 5
10 miša            3 5
11 mrak            1 2 9
12 mrak...         10
13 ni              8
14 nije            8
15 ona             8
16 pojurila        3 5
17 progutala       7
18 to              8
19 znala           8
20 čak             4 6
>>>

```

U spisku reči se ne pojavljuje skup znakova interpunkcije koji su, prema zadanom spisku, uklonjeni funkcijom `strip()`. Ovaj skup znakova se po potrebi može ažurirati u kodu programa.

11.5.2 Poređenje performansi struktura liste i skupa

Poređenje performansi struktura liste i skupa može se izvršiti merenjem vremena *pronalaženja* elementa i vremena *uklanjanja* elementa u strukturama liste i skupa [1].

Radi precizniheg poređenja, svaka od osnovnih operacija pronalaženja i uklanjanja elementa u skupu i listi ponavlja se po 10.000 puta. Program meri i prikazuje ukupno vreme trajanja svih operacija. Vreme se meri pomoću metoda `time()` iz modula `time`, koji vraća *sistemska vreme* `time.time()`. Datum i vreme se interno računaju u odnosu na neki početni datum, koji zavisi od operativnog sistema. Proteklo vreme t je *razlika* vremena završetka t_2 i vremena početka t_1 neke operacije $t = t_2 - t_1$.

Kompletnan program za poređenje performansi struktura liste i skupa je [1]:

```

""" Program prikazuje vreme izvršavanja nekih operacija
    nad strukturama liste i skupa.
    Testira se
    (1) vreme pronalaženja elementa i
    (2) vreme uklanjanja elementa
    iz struktura liste i skupa.
"""
import random
import time

BROJ_ELEMENATA = 10000

# Kreiranje liste zadane dužine
lista = list(range(BROJ_ELEMENATA))
random.shuffle(lista)

# Kreiranje strukture skupa na osnovu liste
skup = set(lista)

# Pronalaženje elemenata u skupu
vremePocetka = time.time() # vreme početka
for i in range(BROJ_ELEMENATA):
    i in skup
vremeZavrsetka = time.time() # vreme završetka
vremeTrajanja = int((vremeZavrsetka - vremePocetka)*1000)
print("Pronalaženje", BROJ_ELEMENATA,
      "elemenata u skupu traje", vremeTrajanja, "ms")

# Pronalaženje elemenata u listi
vremePocetka = time.time() # vreme početka
for i in range(BROJ_ELEMENATA):
    i in lista
vremeZavrsetka = time.time() # vreme završetka
vremeTrajanja = int((vremeZavrsetka - vremePocetka)*1000)
print("\nPronalaženje", BROJ_ELEMENATA,
      "elemenata u listi traje", vremeTrajanja, "ms")

# Uklanjanje pojedinačnih elemenata iz skupa
vremePocetka = time.time() # vreme početka
for i in range(BROJ_ELEMENATA):
    skup.remove(i)
vremeZavrsetka = time.time() # vreme završetka
vremeTrajanja = int((vremeZavrsetka - vremePocetka)*1000)
print("\nUklanjanje", BROJ_ELEMENATA,
      "elemenata iz skupa traje", vremeTrajanja, "ms")

```

```
# Uklanjanje pojedinačnih elemenata iz liste
vremePocetka = time.time() # vreme početka
for i in range(BROJ_ELEMENATA):
    lista.remove(i)
vremeZavrsetka = time.time() # vreme završetka
vremeTrajanja = int((vremeZavrsetka - vremePocetka)*1000)
print("\nUklanjanje", BROJ_ELEMENATA,
      "elemenata iz liste traje", vremeTrajanja, "ms")
```

Rezultat izvršavanja programa, za zadanih 10.000 operacija, je:

Pronalaženje 10000 elemenata u skupu traje 0 ms

Pronalaženje 10000 elemenata u listi traje 2436 ms

Uklanjanje 10000 elemenata iz skupa traje 0 ms

Uklanjanje 10000 elemenata iz liste traje 1234 ms

Vidi se da je vreme *pristupa* i vreme *uklanjanja* elemenata iz strukture skupa neuporedivo je kraće. Za pronalaženje ili uklanjanje 10^5 elemenata strukture *skup* potrebno je manje od 1 milisekunde, dok je za isti broj elemenata strukture *liste* neophodno nekoliko sekundi.

Pitanja za ponavljanje

1. Šta je zadatak analize algoritama?
2. Šta je složenost algoritma?
3. Koja je matematička notacija ocene složenosti algoritma?
4. Navedite nekoliko funkcija rasta, koje se često koriste u oceni složenosti algoritama.
5. Navedite primere algoritama pretraživanja i njihovu ocenu složenosti.
6. Objasnite princip binarnog pretraživanja.
7. Navedite primere algoritama sortiranja i njihovu ocenu složenosti.
8. Objasnite metod sortiranja selekcijom (*Selection Sort*).
9. Objasnite metod sortiranja umetanjem (*Insertion Sort*).
10. Objasnite metod sortiranja mehurom (*Bubble Sort*).

12 Osnove objektno orijentisanog programiranja u jeziku Python

1. Uvod
2. Objekti i klase
3. Skrivanje podataka
4. Grafička notacija za opis klasa u jeziku UML
5. Nasleđivanje
6. Nadjačavanje metoda (override)
7. Polimorfizam
8. Primeri programa

U ovom poglavlju ukratko se izlažu osnove objektno orijentisanog programiranja u jeziku Pythoni i uvodi grafička notacija objektno orijentisanih sistema prema standardu UML.

12.1 Uvod

U klasičnom proceduralnom programiranju, programi se sastoje od funkcija, a podaci se predstavljaju zasebno. Realizacija složenih softverskih sistema zahteva drugačiji pristup, jer složene interakcije programskog koda i podataka iz različitih delova programa čine program složenim i teškim za održavanje.

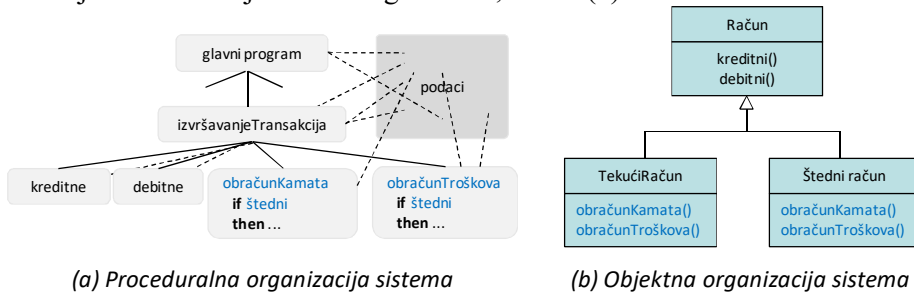
Objektno orijentisani pristup složene softverske sisteme predstavlja kao skup objekata, koji se sastoje od podataka i metoda kojima se vrše operacije nad objektima. Ovaj pristup organizuje programe na isti način kako je organizovan stvarni svet, gde su svi predmeti u *međusobnoj vezi*, kako po atributima, tako i po aktivnostima.

12.1.1 Objektno orijentisani razvoj softvera

Objektno orijentisani razvoj softvera koristi *apstrakciju* kao metod smanjenja kompleksnosti procesa razvoja.

Proceduralna apstrakcija, koja se koristi u klasičnom proceduralnom programiranju, organizuje softver prema funkcijama, a apstrakcija podataka se vrši odvojeno, pomoću struktura podataka, Sl. 43 (a).

Objektno orijentisani pristup posmatra sistem kao skup objekata, koji se sastoje od procedura i podataka, na različitim nivoima apstrakcije i međusobno sarađuju radi izvršenja određenog zadatka, Sl. 43 (b).



Sl. 43. Klasična i objektno orijentisana organizacija softverskog sistema

12.1.2 Objektno orijentisano programiranje

Objektno orijentisani pristup razvoju softvera rešava mnoge probleme koji su svojstveni proceduralnim programiranju, gde su podaci i operacije međusobno *razdvojeni*, tako da je uvek neophodno slanje podataka metodima. Objektno orijentisano programiranje smešta podatke i operacije koje se odnose na njih *zajedno* u objektu.

Objektno orijentisano programiranje u jeziku Pithon podrazumeva način razmišljanja u terminima objekata. Program se može posmatrati kao kolekcija objekata koji međusobno sarađuju.

Korišćenje objekata poboljšava višestruku upotrebljivost softvera i program čini lakšim za razvoj i održavanje [1], [2].

Većina savremenih programskih jezika su objektno orijentisani i mogu biti:

- *objektni jezici (class-based)*, koji imaju mogućnost definisanja klasa i nasleđivanje, npr. C++, Java, C# i Python;
- *objektno zasnovani (prototype-based)*, s ograničenim objektnim svojstvima, kao što je npr. nasleđivanje. Pretežno koriste postojeće ugrađene objekte, kao npr. starije verzije jezika JavaScript.

Primeri jezika koji nisu objektno orijentisani su C, Pascal i ostali stariji proceduralni programski jezici.

12.2 Objekti i klase

U jeziku Python svi podaci, numerički i nenumerički, predstavljaju *objekte*. Objektu se prilikom njegovog kreiranja automatski dodeljuje jednoznačna celobrojna *identifikacija*. Objekti iste vrste su istog *tipa*.

Informacije o pojedinačnim objektima daju ugrađene funkcije `id()` i `type()`:

```
>>> n = 3      # n je definisan kao celi broj
>>> id(n)      # identifikacija dodeljena promenljivoj n
1801060880
>>>> type(n)  # tip promenljive n
<class 'int'>
```

U jeziku Python tip objekta definisan je klasom kojoj pripada. Npr. klasa celobrojnog tipa je `'int'`, decimalnog broja je `'float'`, a tipa string je `'str'`. Termini *tip* (*type*) i *klasa* (*class*) imaju isto značenje, a sama promenljiva je referenca na objekt.

Operacije nad objektima definisane su funkcijama koje se nazivaju *metodi* objekata. Metode mogu pokrenuti samo određeni objekti. Npr. tip string ima metode kao što su `lower()` i `upper()`, koji kao rezultat vraćaju novi string s malim ili velikim slovima:

```
>>> s = "Dobrodošli"
>>> s.lower()
' dobrodošli'
```

Sintaksa za pokretanje metoda nekog objekta je

```
<objekt>.<metod>()
```

Npr. metod `strip()` tipa *string* uklanja znake " ", `\t`, `\f`, `\r` i `\n` (praznine, *whitespace*) s oba kraja stringa:

```
>>> s = "\tDobrodošli!\n"
>>> s.strip()
'Dobrodošli!'
```

Objekt predstavlja entitet u stvarnom svetu, koji se može jasno identifikovati. Npr. objekti su student, radni sto, krug, dugme i kredit.

Objekt ima jedinstven identitet, stanje i ponašanje:

- *identitet* jedinstveni broj koji se dodeljuje svakom objektu;
- *stanje* prikazuje se pomoću promenljivih ili polja podataka - svojstava ili atributa, kao npr. poluprečnik kruga, visina i širina pravougaonika;

- *ponašanje* metodi su funkcije određenog objekta, koje se pokreću radi realizacije neke akcije vezane za objekt.

Apstrakcija klase je razdvajanje implementacije same klase od njene upotrebe. Korisnik klase ili klijent zna samo da klasu koristi, pri čemu opisi metoda i njihovog ponašanja služe kao ugovor (*contract*) između klase i klijenta [1].

Enkapsulacija je skrivanje od korisnika detalja načina objedinjavanja podataka i metoda u jednu celinu. U stvarnosti su detalji implementacije različitih pojava, odnosno sistema, po pravilu skriveni od korisnika. Primeri enkapsulacije su tehnički sistemi, kao što su računari i uređaji u domaćinstvu, za koje korisnici znaju samo funkcije i način njihove upotrebe, te bankarski krediti, za koje korisnici znaju uslove, rokove i konačne efekte, dok ih detalji procesa odobravanja i obrade kredita ne zanimaju.

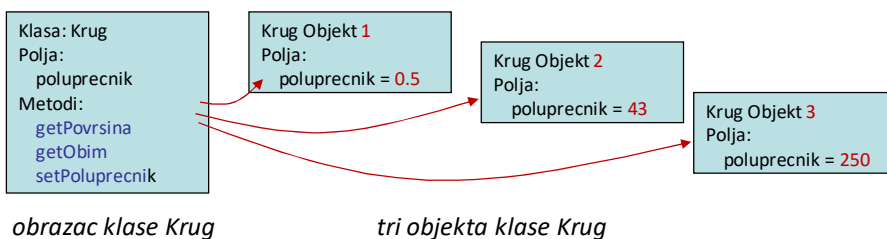
Objektno orijentisano programiranje (OOP) podrazumeva korišćenje objekata za stvaranje programa.

12.2.1 Kreiranje objekata

Klasa predstavlja opis objekata iste vrste, kao što recept za kolač (klasa) predstavlja opis velikog broja konkretnih kolača (objekt).

U jeziku Python korisnički definisana klasa koristi promenljive kao polja za smeštanje podataka i definiše metode za izvršavanje akcija. Tako klasa predstavlja obrazac (*template*), nacrt (*blueprint*) ili ugovor (*contract*).

Objekt je primerak ili instanca (*instance*) klase. Kreiranje instance klase se naziva instancijacija (*instantiation*), Sl. 44 [1].



Sl. 44. Odnos klase i objekta

Za kreiranje i inicijalizaciju objekata korisnički definisanih klasa u objektno orijentisanim jezicima koristi se poseban metod, koji se obično naziva *konstruktor*. U jeziku Python, instancijacija objekata vrše se pomoću dva metoda rezervisanih imena `__new__()` i `__init__()`. Prvi metod kreira

novu instancu, a drugi vrši inicijalizaciju polja podataka i naziva se *inicijalizator* (*initializer*).

12.2.2 Definisane korisničke klase

Korisnička klasa objekata u jeziku Python tipično se definiše naredbom oblika:

```
class <NazivKlase>:
    <inicijalizator>
    <metodi klase>
```

Klasa za smeštanje podataka koristi *polja* (atribute). Prilikom kreiranja novog objekta neke klase, za inicijalizaciju podataka koristi se *inicijalizator*, poseban metod rezervisanog imena `__init__()`. Ovaj metod može da vrši i neke druge akcije.

Inicijalizator ima najmanje jedan parametar (*self*), ali se u parametar listi mogu deklarirati dodatni parametri za prenos različitih vrednosti inicijalizatoru. Ako korisnička klasa nema polja podataka koja treba inicijalizovati, ne mora da ima definiciju sopstvenog inicijalizatora.

Primer definisanja korisničke klase u jeziku Python je nova klasa `Krug` [1].

```
import math
class Krug:
    def __init__(self, poluprecnik = 1):
        self.poluprecnik = poluprecnik
    def getObim(self):
        return 2 * self.poluprecnik * math.pi
    def getPovrsina(self):
        return self.poluprecnik **2 * math.pi
    def setPoluprecnik(self, poluprecnik):
        self.poluprecnik = poluprecnik
```

Inicijalizator rezervisanog naziva `__init__` kreira novu promenljivu, polje podataka `poluprecnik`, u samom *objektu*. To omogućava sistemska promenljiva `self`, koja u nekom objektu predstavlja referencu na sam taj objekt.

Metodi klase su funkcije za uvid u stanje polja podataka objekta klase `getObim()` i `getPovrsina()`, popularno nazvane "geteri" i funkcije za

postavljanje vrednosti polja podataka, popularno "seteri", kao što je funkcija `setPopuprecnik()`.

12.2.3 Kreiranje objekata pomoću konstruktora

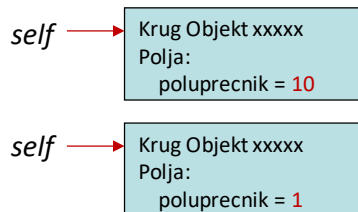
Nakon definisanja klase, objekti se kreiraju pomoću konstruktora, koji kreira u memoriji objekt određene klase i pokreće metod `__init__` za inicijalizaciju objekta. Svi metodi unutar objekta koriste kao prvi argument `self`, referencu na sam objekt, čija vrednost se definiše prilikom kreiranja objekta.

Sintaksa pokretanja konstruktora klase je

```
<NazivKlase>(<argumenti>)
```

Objekti ranije definisane klase `Krug` kreiraju se naredbama:

```
>>> a = Krug(10)
>>> b = Krug()      # default poluprecnik=1
```



Sl. 45. Kreiranje objekata klase `Krug` pomoću konstruktora

Na Sl. 45 prikazana su dva objekta kreirana prethodnim naredbama. Prvi objekt je inicijalizovan na zadanu, a drugi na podrazumevajuću vrednost polja.

12.2.4 Pristup članovima objekta

Polja i metodi klase nazivaju se *članovima* (*members*). Pristup članovima, podacima i metodima, vrši se korišćenjem naziva objekta i operatora `"."`:

```
<promenljivaObj> = <NazivKlase>(<argumenti>)
<promenljivaObj>.<polje>
<promenljivaObj>.<metod>(<argumenti>)
```

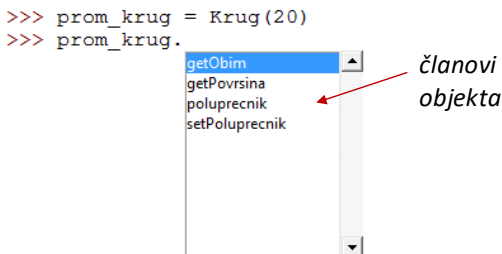
Npr. pristup članovima objekt klase `Krug` poluprečnika 20 i postavljanje nove vrednosti poluprečnika kruga:

```

>>> prom_krug = Krug(20)
>>> prom_krug.poluprecnik
20
>>> prom_krug.getObim()
125.66370614359172
>>> prom_krug.getPovrsina()
1256.6370614359173
>>> prom_krug.setPoluprecnik(5)
>>> prom_krug.poluprecnik
5

```

Prilikom unosa programskog koda, savremeni programski editori tumače unesni tekst i, kada na osnovu tipa promenljive ustanove da se pristupa članu klase, nude izbor njegovog naziva umesto unosa, kao na Sl. 46.



Sl. 46. Prikaz spiska članova klase prilikom unosa programskog koda

Na isti način editori prepoznaju i pomažu kod unosa ključnih reči, parametara funkcija i drugih elemenata programa. Ovakva podrška smanjuje broj greški koje nastaju prilikom unosa teksta, posebno dugih naziva objekata, ali i pomaže programeru da bolje razume i neprekidno proverava kod programa.

12.2.5 Parametar *self*

Parametar *self* omogućava referisanje nekog objekta na samog sebe, odnosno omogućava metodima klase da u konkretnim objektima pristupaju članovima klase koristeći izraze `self.<nazivPolja>` i `self.<nazivMetoda>()`.

Oblast definisanosti promenljivih kreiranih kao `self.<nazivPolja>` je cela klasa. Metodi klase mogu kreirati i lokalne promenljive:

```
class NazivKlase:
    def __init__(self, ...):
        self.x = 1      # promenljiva x na nivou klase
    ...
    def metod1(self, ...):
        self.y = 2      # promenljiva y na nivou klase
    ...
    z = 5 # lokalna promenljiva na nivou metoda1
```

Primer upotrebe postojeće klase je klasa `Krug`, koja je definisana u modulu `krug.py`. Fajl s istim nazivom treba da se nalazi na tekućem ili nekom drugom folderu u kojem ga sistem može pronaći prilikom izvršavanja naredbe `import`:

```
from krug import Krug

# Kreiranje objekta tipa krug s poluprečnikom 1
krug1 = Krug()
print("Površina kruga poluprečnika", \
      krug1.poluprecnik, "je", krug1.getPovrsina())

# Kreiranje objekta tipa krug s poluprečnikom 250
krug2 = Krug(250)
print("Površina kruga poluprečnika", \
      krug2.poluprecnik, "je", krug2.getPovrsina())
# Promena poluprečnika objekta krug2 u 100
krug2.poluprecnik = 100      # ili setPoluprecnik(100)
print("Površina kruga poluprečnika", \
      krug21.poluprecnik, "je", krug2.getPovrsina())
```

Rezultat izvršavanja ovog kratkog programa je:

```
Površina kruga poluprečnika 1 je 3.141592653589793
Površina kruga poluprečnika 250 je 196349.54084936206
Površina kruga poluprečnika 100 je 31415.926535897932
>>>
```

12.3 Skrivanje podataka

Skrivanje polja podataka vrši se radi sprečavanja njihovog oštećivanja i lakšeg održavanja. U jeziku Python skrivena privatna polja klase definišu se tako što se koriste nazivi polja koji počinju s dve donje crtice (*underscore*). U definiciji klase `Krug`, polje `poluprečnik` može se sakriti od direktnog pristupa promenom naziva u `__poluprecnik`.

Pristup skrivenim poljima definiše se posebnim metodima za pristup tipa *get* (uvid u vrednost), *is* (logička vrednost) i *set* (promena vrednosti), koja prema konvenciji imaju zaglavlja:

```
def <getNazivSvojstva>(self):
def <isNazivSvojstva>(self):
def <setNazivSvojstva>(self, <vrednostSvojstva>):
```

Neki tipovi, odnosno strukture podataka u jeziku Python su nepromenljivi (*immutable*), npr. stringovi i n-torke, dok su neki promenljivi (*mutable*), npr. liste i rečnici.

Prenos složenih struktura podataka kao parametara funkcija omogućava funkcijama da menjaju samo *promenljive objekte*.

12.4 Grafička notacija za opis klasa u jeziku UML

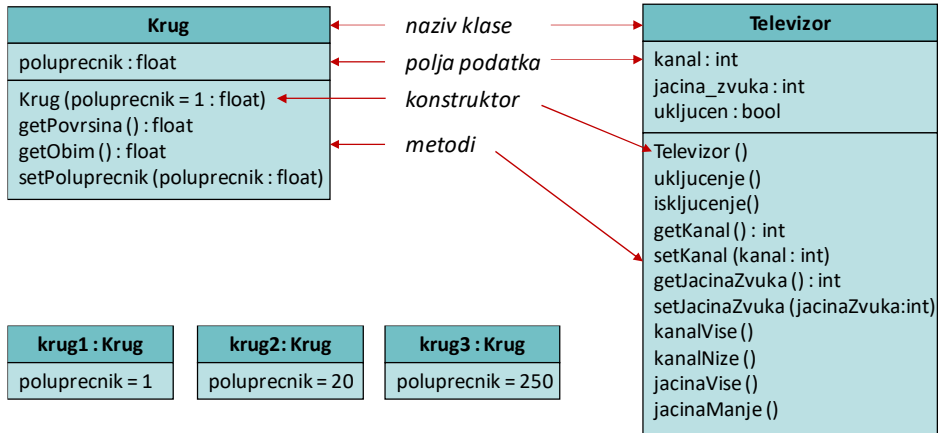
U prethodnim primerima klase i objekti su grafički prikazivani kao pravougaonici s tekstualnim nazivima članova klase ili objekta. Takav prikaz se zasniva na ISO standardu grafičkog opisa strukture softvera.

Standard UML (*Unified Modeling Language*) predstavlja semi-formalni grafički jezik namenjen razvoju sistema zasnovanom na modelima. Autori ovog grafičkog jezika su naučnici Grady Booch, Ivar Jacobson i James Rumbaugh 1994-1995. godine, koji su zatim 1996. godine osnovali softversku kompaniju *Rational*, koja je sada deo korporacije IBM.

Razvoj standarda UML od 1997. godine vodi organizacija *Object Management Group* (OMG), www.omg.org, koja se bavi razvojem tehnoloških standarda za podršku objektno orijentisanog razvoja softvera, posebno metoda razvoja zasnovanih na modelima. Standard UML je 2005. godine potvrdila i svetska organizacija za standardizaciju ISO (*International Organization for Standardization*).

Aktuelna verzija standarda jezika ima oznaku UML 2.5 (2015) i definiše 14 vrsta dijagrama za opis softvera. Softverski sistem se prikazuje s različitih aspekata, posebnim vrstama dijagrama za prikaz *strukture* i *ponašanja* softvera. Jedan od osnovnih dijagrama, koji opisuje statičku strukturu softvera je UML dijagram klasa (*Class diagram*).

Dijagram klasa omogućava opis statičke strukture softvera pomoću grafičkog prikaza specifikacije klasa, objekata i njihovih veza. Na Sl. 47 prikazane su klase *Krug* i *Televizor*, kao i nekoliko objekata, instanci klase *Krug*.

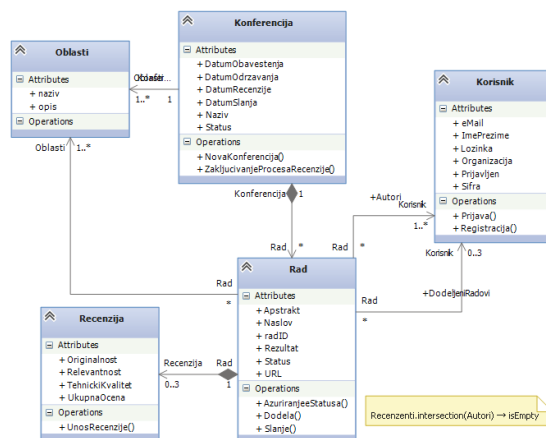


Sl. 47. Elementi UML dijagrama klasa i objekata

Jezik UML se može koristiti na više načina, a najčešće se upotrebljava za (1) izradu skica prilikom analize problema, (2) izradu projektnih dijagrama uz pomoć računara i (3) kao programski jezik, za programiranje pomoću UML.

Izradu skica prilikom analize problema pomaže projektantima i programerima u razumevanju sistema, kao i u komunikaciji s korisnicima.

Izradu detaljnih projektnih dijagrama uz pomoć računara omogućava članovima tima da projektuju i realizuju sistem uz pomoć računara.



Razvojni alati se koriste za vizualizaciju i bolje razumevanje postojećeg koda preko UML dijagrama (*reverse engineering*), kao i za generisanje strukture programskog koda na osnovu dijagrama klasa (*forward engineering*).

Jezik UML se koristi i kao programski jezik, za razvoj kompletne izvršne specifikacije softverskih, posebno ugrađenih sistema. Izvorni programski kod aplikacija automatizovano generiše sam razvojni alat.

12.5 Nasleđivanje

Klasično proceduralno programiranje bavi se razvojem funkcija nad globalnim strukturama podataka. Objektno orijentisano programiranje bavi se razvojem objekata i operacija nad objektima.

Nasleđivanje je postupak razvoja novih klasa na osnovu postojećih klasa, tako što se objedinjavaju njihovi zajednički elementi, čime se izbegava redundansa i olakšava razumevanje i održavanje softvera. Npr. klase koje u programu modeliraju geometrijske objekte krug, pravougaonik i trougao mogu da imaju neke zajedničke elemente, kao što su boja, debljina linije, pozicija i sl.

Klase se koriste za modeliranje objekata istog tipa. Objekti različitih tipova mogu da imaju zajedničke elemente, koji se mogu dalje generalizovati u novu klasu, tako da ih može deliti više klasa.

Nasleđivanje omogućava definisanje opštije klase (*superclass*) i nakon toga njeno proširenje definisanjem specifičnijih klasa objekata (*subclass*). Primer su geometrijski objekti (krug, pravougaonik), čiji zajednički elementi mogu biti boja, debljina linije i datum kreiranja objekta.

Specifične klase geometrijskih objekata mogu biti krug i pravougaonik, svaka sa svojim posebnim atributima i metodima. Klasa `GeometrijskiObjekt` je osnovna klasa (*base class*), a klase pojedinih geometrijskih objekata su iz nje izvedene klase (*derived class*).

Osnovna klasa `GeometrijskiObjekt` definiše zajedničke elemente, Sl. 48. Izvedene specifične klase geometrijskih objekata `Krug` i `Pravougaonik` imaju svoje posebne attribute i metode.

Opšta klasa `GeometrijskiObjekt` ima attribute `boja` i `obojen`, te odgovarajuće metode `get` i `set`. Osim toga može da ima atribut `__str__` i metod `__str__()`, koji vraća tekstualni opis same klase.

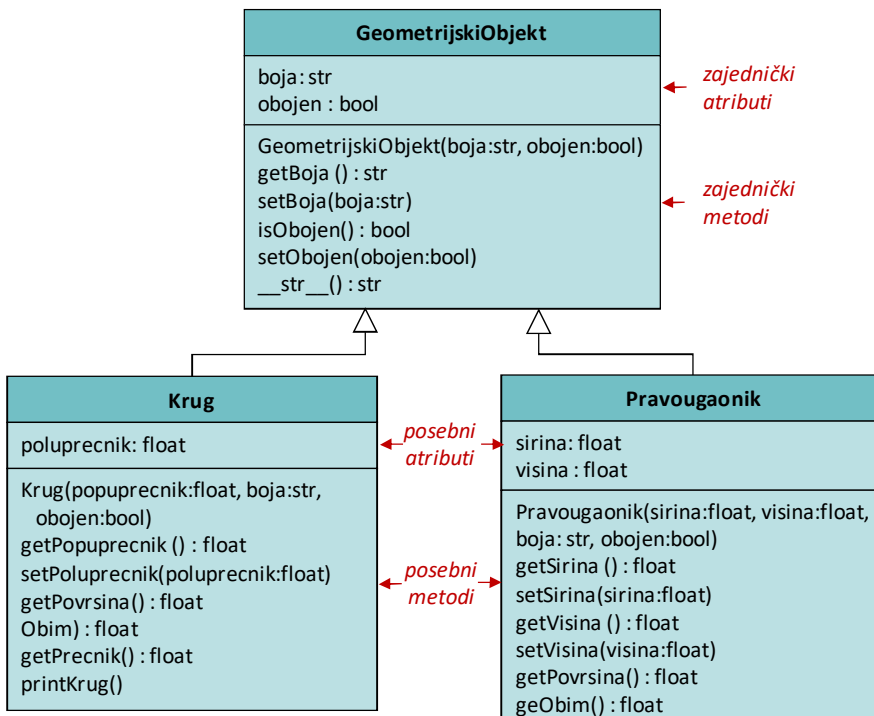
Specifične klase geometrijskih objekata su `Krug` i `Pravougaonik`, svaka sa svojim posebnim atributima i metodima za računanje veličina specifičnih za svaku izvedenu klasu objekata.

U jeziku Python proširenje osnovne klase definiše se naredbom:

```
class <IzvedenaKlasa>(<OsnovnaKlasa>)
```

Specifičnije klase nasleđuju sva dostupna polja i metode osnovne klase GeometrijskiObjekt. Izvedene klase nasleđuju promenljive boja i obojen, ali ih moraju kreirati i inicijalizovati pozivom metoda `__init__` nadređene klase. Prikaz informacija o objektima pomoću metoda `printKlasa()` takođe je specifičan za svaku izvedenu klasu.

Primer definisanje nasleđivanja u jeziku Python je definisanje klase `Krug` kao specijalizacije prethodno definisane osnovne klase `GeometrijskiObjekt`.



Sl. 48. Primer osnovne i izvedenih klasa

Klase prikazane na Sl. 48 mogu se realizovati u obliku programa u jeziku Python.

Prvo se definiše osnovna klasa `GeometrijskiObjekt`, a njena definicija se kao modul sačuva na fajlu pod nazivom `GeometrijskiObjekt.py`:

```
class GeometrijskiObjekt:
    def __init__(self, boja="plava", obojen=True):
        # inicijalizator
        self.__boja = boja
        self.__obojen = obojen

    def getBoja(self):
        return self.__boja

    def setBoja(self, boja):
        self.__boja = boja

    def isObojen(self):
        return self.__obojen

    def setObojen(self, obojen):
        self.__obojen = obojen

    def __str__(self):
        return "boja: " + self.__boja + \
            ", obojen: " + str(self.__obojen)
```

Klasa `Krug` definisana je kao *izvedena klasa*, na osnovu uvezene definicije klase `GeometrijskiObjekt`:

```
import math
from GeometrijskiObjekt import GeometrijskiObjekt

class Krug(GeometrijskiObjekt): # izvedena klasa
    def __init__(self, poluprecnik):
        # Inicijalizator nove klase
        super().__init__() # inicijalizator osn. kl.
        self.__poluprecnik = poluprecnik

    def getPoluprecnik(self):
        return self.__poluprecnik

    def setPoluprecnik(self, poluprecnik):
        self.__radius = poluprecnik
```

```

def getPovrsina(self):
    return self.__poluprecnik ** 2 * math.pi

def getPrecnik(self):
    return 2 * self.__poluprecnik

def getObim(self):
    return 2 * self.__poluprecnik * math.pi

def printKrug(self):
    print(self.__str__() + " Poluprečnik: " + \
          str(self.__poluprecnik))

```

Definicija izvedene klase može se kao modul sačuvati na fajlu pod nazivom `KrugIzvedeniGeometrijskiObjekt.py`.

Na isti način definiše se i *izvedena klasa* `Pravougaonik`, kao modul koji se može sačuvati na fajlu `KrugIzvedeniGeometrijskiObjekt.py` [1]:

```

from GeometrijskiObjekt import GeometrijskiObjekt

```

```

class Pravougaonik(GeometrijskiObjekt): #izvedena kl.
    def __init__(self, sirina=1, visina=1):
        # Inicijalizator nove klase
        super().__init__() # inicijalizator osn. kl.
        self.__sirina = sirina
        self.__visina = visina

    def getSirina(self):
        return self.__sirina

    def setSirina(self, sirina):
        self.__sirina = sirina

    def getVisina(self):
        return self.__visina

    def setVisina(self, visina):
        self.__visina = visina

```

```

def getPovrsina(self):
    return self.__sirina * self.__visina

def getPrecnik(self):
    return 2 * self.__poluprecnik

def getObim(self):
    return 2 * (self.__sirina * self.__visina)

def printPravougaonik(self):
    print(self.__str__() + \
          " Sirina: " + str(self.__sirina) + \
          " Visina: " + str(self.__visina))

```

Primer jednostavnog testnog programa, koji koristi ove dve izvedene iz klase je:

```

from KrugIzvedeniGeometrijskiObjekt import Krug
from PravougaonikIzvedeniGeometrijskiObjekt import
Pravougaonik

krug = Krug(12.5)
print("Krug", krug)
print("Poluprecnik=", krug.getPoluprecnik())
print("Površina   =", krug.getPovrsina())
print("Obim       =", krug.getObim())

pravougaonik = Pravougaonik(10, 20)
print("\nPravougaonik", pravougaonik)
print("Površina =", pravougaonik.getPovrsina())
print("Obim     =", pravougaonik.getObim())

```

Rezultat izvršavanja kratkog testnog programa je:

```

Krug boja: plava, obojen: True
Poluprecnik= 12.5
Površina   = 490.8738521234052
Obim       = 78.53981633974483

Pravougaonik boja: plava, obojen: True
Površina = 200
Obim     = 400

```

Vidi se da oba objekta imaju iste vrednosti svojstava `boja` i `obojen`, koja nasleđuju od osnovne klase, dok su specifična svojstva objekata različita. Za njihov prikaz su upotrebljeni specifični metodi svake od izvedenih klasa.

12.6 Nadjačavanje metoda (override)

Ponekad je korisno da se u izvedenoj klasi za određenu funkciju koristi dugačiji metod od nasleđenog. Izmena nasleđenog metoda naziva se *nadjačavanje metoda* (*method overriding*).

Primer takvog metoda je nasleđeni metod `__str__()`, koji se može promeniti u izvedenoj klasi, tako da umesto opisa osnovne klase `GeometrijskiObjekt` vraća samo opis *izvedene* klase [1]:

```
class Krug(GeometrijskiObjekt):
    ...
    # Nadjačavanje metoda __str__ osnovne klase
    def __str__(self):
        return + " poluprecnik: " + str(poluprecnik)
```

Izvedena klasa `Krug` može koristiti oba metoda, sopstveni metod `__str__()` ili metod osnovne klase pozivom `GeometrijskiObjekt.__str__()`.

12.7 Polimorfizam

Naziv *polimorfizam* izveden je od grčkog izraza za višestrukost oblika. Polimorfizam u programiranju odnosi se na višestrukost definicije neke klase objekata, u kojoj se definišu osnovni elementi, koji su zajednički za više izvedenih klasa, a svaka od tih klasa ne samo da može da definiše svoje specifične elemente, već i *izmeni* postojeće [1].

Često se kaže da osnovna klasa definiše *opšti interfejs*, koje će imati sve klase koje je nasleđuju, a svaka od klase izvedenih klasa ima mogućnost da definiše sopstvenu *implementaciju* tog interfejsa.

Polimorfizam predstavlja mogućnost kreiranja jedinstvenog interfejsa za objekte različitih tipova. Metodi polimorfnog tipa mogu se primeniti na vrednosti jednog ili više drugih tipova objekata.

U jeziku Python polimorfizam je mogućnost da se objekt podređene klase (*subclass*) može preneti kao parametar tipa nadređene klase (*superclass*). Npr. objekti tipa `Krug` i `Pravougaonik` mogu se koristiti u funkcijama koje očekuju objekte tipa `GeometrijskiObjekt`.

Polimorfnost funkcija koje očekuju objekte tipa `GeometrijskiObjekt` može se ilustrovati sledećim primerom [1]:

```
from KrugIzvedeniGeometrijskiObjekt import Krug
from PravougaonikIzvedeniGeometrijskiObjekt \
    import Pravougaonik

# Prikaz svojstava geometrijskog objekta
def prikaziObjekt(g) :
    print(g.__str__())

# Poređenje površine dva geometrijska objekta
def isIstaPovrsina(g1, g2) :
    return g1.getPovrsina() == g2.getPovrsina()

# Prikaz svojstava kruga i pravougaonika
k = Krug(4)
p = Pravougaonik(2, 5)
prikaziObjekt(k)
prikaziObjekt(p)
print("Da li su krug i pravougaonik iste površine?", \
      isIstaPovrsina(k, p))
```

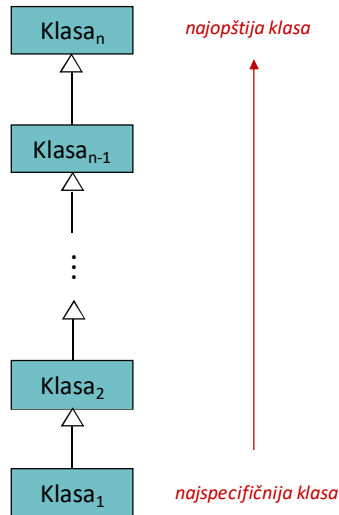
Rezultat izvršavanja programa je:

```
boja: plava, obojen: True poluprecnik: 4
boja: plava, obojen: True širina: 2 visina: 5
Da li su krug i pravougaonik iste površine? False
```

12.7.1 Dinamičko povezivanje

U hijerarhiji nasleđivanja objekti nasleđuju svojstva osnovne klase, kojima dodaju svoja specifična svojstva. Svaka izvedena klasa je specijalizacija nadređene klase, tako da je svaka instanca izvedene klase istovremeno instanca nadređene klase, koja se može prenositi kao parametar funkcije koja očekuje objekt tipa nadređene klase.

Polimorfizam je mogućnost da se objekt izvedene klase može preneti kao parametar tipa osnovne klase. Neki metod se može implementirati u više klasa povezanih u hijerarhiju nasleđivanja, Sl. 49.



Sl. 49. Izbor konkretnog metoda u hijerarhiji nasleđivanja klasa

Izbor metoda koji će se pokrenuti obavlja se u toku izvršavanja programa, prema hijerarhiji nasleđivanja, što se naziva *dinamičko povezivanje* (*dynamic binding*). Interpreter jezika Python traži zadani metod počev od najspecifičnije klase (Klasa1, Klasa2, ...), Sl. 49 i primenjuje prvi metod traženog naziva koji pronađe [1].

Primer dinamičkog povezivanja metoda nadređene klase s objektima podređene klase je veza klase Student i iz nje izvedene klase DiplomiraniStudent:

```
class Student:
    def __str__(self):
        return "Student"
```

```
def printStudent(self):
    print(self.__str__())
```

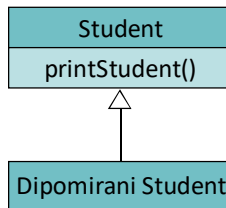
```
class DiplomiraniStudent(Student):
    def __str__(self):
        return "Diplomirani student"
```

```
a = Student()
b = DiplomiraniStudent()
a.printStudent()
b.printStudent()
```

Rezultat izvršavanja programa je:

```
Student
Diplomirani student
```

Klasa `DiplomiraniStudent` izvedena je od klase `Student`. Nasleđuje metod `printStudent()`. Metod `printStudent()` primenjen je na objekte `a` i `b`, različitih klasa, Sl. 50 [1].



Sl. 50. Primer dinamičkog povezivanja nasleđenog metoda

12.8 Preklapanje operatora

Jezik Python omogućava upotrebu određenih operatora za rad s objektima više različitih klasa, npr.

- operatori `+` i `*` mogu se koristiti za numeričke vrednosti i objekte klase *string*. Oznacavaju različite operacije: množenje i deljenje za numeričke vrednosti i konkatenciju različitih ili većeg broja istih stringova;
- relacioni operatori `>`, `>=`, `==`, `<`, `<=`, operator pristupa elementu niza `[]` i operatori pripadnosti takođe se mogu koristiti za objekte različitih klasa.

Višestruka upotreba operatora za različite operacije nad objektima različitih klasa naziva se preklapanje operatora (*operator overloading*). Preklapanje operatora omogućava *upotrebu ugrađenih operatora za izvršavanje korisnički definisanih metoda*.

U jeziku Python ugrađeni operatori realizovani su kao metodi odgovarajućih klasa. Definisanje novih metoda za upotrebu postojećih operatora nad novim korisničkim klasama vrši se na osnovu njihovog rezervisanog naziva. Npr. upotreba operatora `+` za objekte korisničke klase omogućava se definisanjem metoda `__add__`:

```
class KorisnickaKlasa(...):
    ...
    # Preklapanje ugrađenog metoda __add__
    def __add__(self, objektKlase):
        rezultat = ...
        return rezultat
```

U tabeli su prikazani nazivi metoda ugrađenih operatora jezika Python [1]:

Operator	Naziv metoda	Opis
+	<code>__add__(self, other)</code>	Sabiranje (<i>addition</i>)
*	<code>__mul__(self, other)</code>	Množenje (<i>multiplication</i>)
-	<code>__sub__(self, other)</code>	Oduzimanje (<i>subtraction</i>)
%	<code>__mod__(self, other)</code>	Ostatak (<i>remainder</i>)
/	<code>__truediv__(self, other)</code>	Deljenje (<i>division</i>)
<	<code>__lt__(self, other)</code>	Manje od (<i>less than</i>)
<=	<code>__le__(self, other)</code>	Manje ili jednako (<i>less than or equal</i>)
==	<code>__eq__(self, other)</code>	Jednako (<i>equal to</i>)
!=	<code>__ne__(self, other)</code>	Različito (<i>not equal to</i>)
>	<code>__gt__(self, other)</code>	Veće od (<i>greater than</i>)
>=	<code>__ge__(self, other)</code>	Veće ili jednako (<i>greater than or equal</i>)
[index]	<code>__getitem__(self, index)</code>	Indeks elementa (<i>index operator</i>)
in	<code>__contains__(self, value)</code>	Provera pripadnosti (<i>membership</i>)
len	<code>__len__(self)</code>	Broj elemenata (<i>number of elements</i>)
str	<code>__str__(self)</code>	Opis u obliku stringa (<i>string repr.</i>)

Primer: Definisanje operacije + za objekte klase **Krug**

```
import math

class Krug:
    def __init__(self, poluprecnik):
        self.__poluprecnik = poluprecnik

    def setPoluprecnik(self, poluprecnik):
        self.__poluprecnik = poluprecnik

    def getPoluprecnik(self):
        return self.__poluprecnik

    def Povrsina(self):
        return math.pi * self.__poluprecnik ** 2

    def __add__(self, krug2):
        return Krug(self.__poluprecnik +
krug2.__poluprecnik)
```

```

krug1 = Krug(5)
krug2 = Krug(10)
print("Krug1=", krug1.getPoluprecnik(), \
      "\nKrug2=", krug2.getPoluprecnik())

krug3= krug1 + krug2 # preklopljeni operator +
print("Krug 3 =", krug3.getPoluprecnik())

```

Izvršavanje programa daje ispis:

```

Krug1 = 5
Krug2 = 10
Krug3 = 15

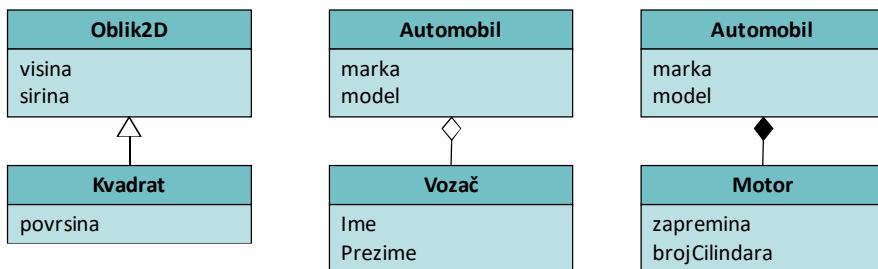
```

12.9 Kreiranje klasa na osnovu veza

Za kreiranje međusobno povezanih klasa potrebno je poznavanje njihovih međusobnih veza u stvarnosti.

Opšta veza između klasa naziva se *asocijacija* i označava se pravom linijom.

Izvedena klasa još može predstavljati specijalni slučaj osnovne klase (veza tipa *is-a*) ili može biti njen neobavezni ili obavezni sastavni deo (veza tipa *has-a*). Odgovarajuće veze između klasa nazivaju *generalizacija* (Δ), *agregacija* ($\diamond$) i *kompozicija* ($\blacklozenge$), Sl. 51.



Sl. 51. Primeri različitih veza između klasa

Veze *asocijacije* modeliraju opšte međusobne odnose između stvarnih entiteta. Primer asocijacije je veza između klasa *Student*, *Predmet* i *Profesor* prikazan je na Sl. 52 [1]. Svaka veza klase ima *multiplikativnost*, odnosno ograničenja u broju objekata klase koji se mogu povezati. Npr. oznaka *m..n* predstavlja minimalni i maksimalni broj objekata jedne klase koji se mogu povezati, a znak *** označava da nema ograničenja. Na Sl. 52 se vidi da student

može da upiše bilo koji broj predmeta, dok na jednom predmetu može biti 5..20 studenata. Profesor može da predaje najviše tri predmeta, a jedan predmet može da predaje samo jedan profesor.



Sl. 52. Primer veza asocijacije između klasa

Implementacija relacije asocijacije u jeziku Python vrši se pomoću polja podataka i odgovarajućih metoda. Npr. veze klasa sa Sl. 52 mogu se realizovati pomoću koda čija je struktura [1]:

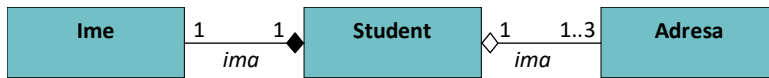
```

class Student:
    # Dodavanje novog predmeta listu
    def dodajPredmet(self, predmet):
    ...
class Predmet:
    # Dodavanje novog studenta u listu
    def dodajStudent(self, student):
    ...
    def unesiProfesor(self, profesor):
    ...
class Profesor:
    # Dodavanje novog predmeta u listu
    def dodajPredmet(self, predmet):
    ...
  
```

Klasa Student ima listu upisanih predmeta, klasa Profesor listu predmeta koje predaje, a klasa Predmet listu upisanih studenata i polje za profesora koji predaje predmet.

Relacija "student *upisuje* predmet" realizovana je metodom `dodajPredmet()` u klasi Student i metodom `dodajStudent` u klasi Predmet. Relacija "profesor *predaje* premet" realizovana je metodom `dodajPredmet()` u klasi Profesor i `unesiProfesor()` u klasi Predmet.

Na sličan način se modeliraju se i relacije agregacije i kompozicije. Na Sl. 53 prikazane su veze agregacije i kompozicije između klasa Ime, Student i Adresa [1], koje modeliraju kompoziciju "student *ima* ime" (obavezno, samo jedno) i agregaciju "student *ima* adresu" (jedan do tri studenta mogu da imaju istu adresu).



Sl. 53. Primer veza kompozicije i agregacije između klasa

Realizacija prikazanih veza u jeziku Python ima opšti oblik [1]:

```

class Ime:
    ...
class Student:
    def __init__(self, ime, adresa):
        self.ime = ime
        self.adresa = adresa
    ...
class Adresa:
    ...

```

Relacija kompozicije i agregacije realizovane su kao polja podataka ime i adresa u klasi Student.

12.10 Primeri objektno orijentisanih programa

U ovom poglavlju daju se dva primera objektno orijentisanih programa u jeziku Python. Prvi primer je objektno orijentisana verzija ranije prikazanog programa za računanje indeksa telesne mase. Drugi primer je program za unos podatka pomoću grafičkog korisničkog interfejsa, koji je razvijem korišćenjem klasa iz biblioteke `tkinter`.

12.10.1 Klasa IndeksTelesneMase

Prvo se u posebnom modulu definiše odgovarajuća klasa Indeks Telesne Mase, koja se zatim koristi za izradu objektno orijentisanog programa.

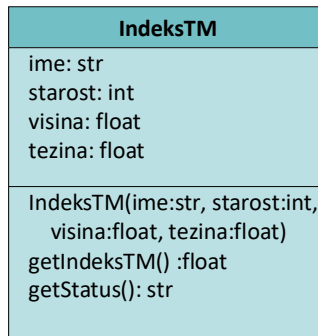
Objektno orijentisani razvoj bavi se povezivanjem podataka i operacija nad njima u jedinstvene strukture: objekte.

Npr. jednostavni program `indeks_telesne_mase.py` iz poglavlja 2 nije dobro softversko rešenje, jer se ne može ponovo koristiti od strane drugih programa.

Rešenje bi se moglo nešto poboljšati modularizacijom, tako što bi se definisala funkcija `IndeksTM(visina, težina)` i višestruko koristila u programima.

No, ako bi trebalo koristiti dodatne podatke o osobama (npr. ime i starost), promenljive bi se kreirale na različitim mestima u programima, a sama funkcija bi se trebala značajno promeniti.

Pogodniji način realizacije programa je smeštanje i funkcija i podataka u jednu međusobno povezanu strukturu, *klasu* IndeksTM, koja je prikazana na UML dijagramu, Sl. 54.



Sl. 54. UML dijagram klase indeksTM

Svi podaci neke osobe smeštaju se u polja podataka ime, starost, visina i tezina. Inicijalizator klase kreira instancu klase, objekt IndeksTM, za zadano ime, starost, težinu i visinu jedne osobe.

Metodi klase računaju indeks telesne mase i vraćaju status osobe (*Normalna težina*, *Povećana težina*, itd.).

Modul IndeksTM sadrži istoimenu klasu, koja računa indeks telesne mase i na osnovu njega određuje status telesne mase osobe:

```

"""
    Modul IndeksTM

    Klasa za računanje indeksa i određivanje statusa
    telesne mase osobe
"""
class IndeksTM:
    # Inicijalizator
    def __init__(self, ime, starost, tezina, visina):
        self.__ime = ime
        self.__starost = starost
        self.__tezina = tezina
        self.__visina = visina

    def getIndeksTM(self):          # indeks telesne mase
        ind = self.__tezina / ((self.__visina/100)**2)
        return round(ind * 100) / 100

    def getStatus(self):           # opis statusa
        ind = self.getIndeksTM()
        if ind < 18.5:
            return "Nedovoljna težina"
        elif ind < 25:
            return "Normalna težina"
        elif ind < 30:
            return "Povećana težina"
        else:
            return "Gojaznost"

    def getIme(self):
        return self.__ime          # skrivena promenljiva

    def getStarost(self):
        return self.__starost      # skrivena promenljiva

    def getTezina(self):
        return self.__tezina       # skrivena promenljiva

    def getVisina(self):
        return self.__visina       # skrivena promenljiva

```

Objektno orijentisani pristup objedinjava dobre strane proceduralnog programiranja (modularnost) i mogućnost integracije podataka i svih operacija nad njima u objekte.

Pogodno definisanu klasu programi mogu višestruko upotrebljavati. Modul IndeksTM može se koristiti na isti način u različitim programima, npr.

```
from indeks_telesne_mase import IndeksTM
...
ind1 = IndeksTM("Jovana Jovanović", 18, 66, 178)
print("Indeks telesne mase za", ind1.getIme(), "je", \
      ind1.getIndeksTM(), ind1.getStatus())
...
ind2 = IndeksTM("Nikola Nikolić", 50, 98, 178)
print("Indeks telesne mase za", ind2.getIme(), "je", \
      ind2.getIndeksTM(), ind2.getStatus())
...
```

12.10.2 Program IndeksTelesne Mase (objektno orijentisana verzija)

Objektno orijentisan program za računanje indeksa i određivanje statusa telesne mase neke osobe može se praktično realizivati pomoću dva modula, IndeksTM i glavnog programa.

Glavni program učitava osnovne podatke o osobi, a zatim računa i prikazuje indeks i status pomoću metoda klase getIndeksTM() i getStatus() iz modula IndeksTM:

```
"""
    Program računa indeks telesne mase osobe
    i štampa status (normalna, povećana, ...)
"""

from indeks_telesne_mase import IndeksTM

# Unos imena, starosti, telesne težine i visine osobe
ime = input("Unesite ime i prezime osobe: ")
starost = int(\
    input("Unesite starost osobe (godina): "))
težina = float(\
    input("Unesite telesnu težinu (kg): "))
visina = float(\
    input("Unesite visinu (cm): "))
```

```
# Kreiranje objekta klase IndeksTM
indeks = IndeksTM(ime, starost, težina, visina)

# Računanje indeksa telesne mase i ispis rezultata
print("Indeks telesne mase za", indeks.getIme(), \
      "je", indeks.getIndeksTM(), indeks.getStatus())
```

Rezultat izvršavanja programa za računanje indeksa i statusa telesne mase dve osobe je:

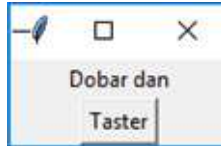
```
Unesite ime i prezime osobe: Petar Petrović
Unesite starost osobe (godina): 18
Unesite telesnu težinu (kg): 66
Unesite visinu (cm): 178
Indeks telesne mase za Petar Petrović je 20.83 Normalna
težina
>>>
Unesite ime i prezime osobe: Ivan Ivanović
Unesite starost osobe (godina): 66
Unesite telesnu težinu (kg): 97
Unesite visinu (cm): 178
Indeks telesne mase za Ivan Ivanović je 30.61 Gojaznost
>>>
```

12.10.3 Osnovni elementi biblioteke Tkinter

Programska biblioteka `tkinter` (od *Tk interface*) [1] sadrži klase, koje se koriste za kreiranje grafičkog korisničkog interfejsa (*Graphical User Interface*, GUI) programa u jeziku Python. U sledećoj tabeli prikazane su klase ove biblioteke koje su osnovni elementi grafičkog korisničkog interfejsa:

Klasa (widget)	Opis
Button	Taster koji se koristi za izvršavanje metoda
Canvas	Komponenta za crtanje slika i grafikona
Checkbutton	Polje za izbor
Entry	Polje za unos teksta (<i>text box</i>)
Frame	Kontajner ostalih vizuelnih komponenti
Label	Prikazuje tekst ili sliku
Menu	Panel za implementaciju menija (<i>pull-down/popup</i>)
Menubutton	Komponenta za implementaciju tastera menija (<i>pull-down</i>)
Message	Prikaz teksta koji se prilagođava obliku prozora
Radiobutton	Polje za izbor koje briše druga povezana polja za izbor
Text	Prikaz formatiranog teksta, koji može da sadrži slike

Osim formi za unos podataka, biblioteka `tkinter` omogućava kreiranje meni sistema i prikaz slika i grafikona. Primer jednostavnog grafičkog interfejsa, koji se sastoji od grafičkog prozora u kome su dva vizuelna objekta, tekstualna poruka i grafički taster (*button*), Sl. 55.



Sl. 55. Primer grafičkog korisničkog interfejsa

Jednostavni interfejs sa Sl. X može se kreirati pomoću svega nekoliko naredbi:

```
from tkinter import * # uvoz svih imena iz biblioteke

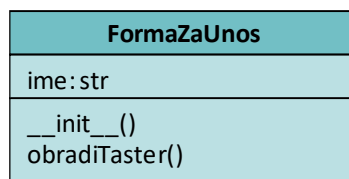
prozor = Tk()         # Kreiranje grafičkog prozora
# Kreiranje grafičkih elemenata za prozor
tekst = Label(prozor, text="Dobar dan") # tekst
taster = Button(prozor, text="Taster")   # taster
# Raspoređivanje elemenata u prozoru (redovi/kolone)
tekst.pack()           # Smeštanje teksta u objekt prozor
taster.pack()          # Smeštanje tastera u objekt prozor

prozor.mainloop() # Osnovna petlja za obradu događaja
```

U primeru metod `Tk()` kreira prozor (*window*), koji predstavlja osnovnu klasu (kontejner) za ostale vizuelne komponente (*widgets*). Osnovna petlja obrađuje događaje objekta prozor, npr. pritisak na taster, pomeranje i zatvaranje prozora.

12.10.4 Program za unos podatka preko grafičkog interfejsa

Korisnička klasa koja predstavlja grafičku formu za unos podatka, može se predstaviti u UML notaciji kao na Sl. 56.



Sl. 56. Struktura klase FormaZaUnos

Klasa ima jedno polje za unos tekstualnog podatka (tipa *string*) i dva metoda, inicijalizator i metod `obradiTaster()`, koji može npr. da omogući preuzimanje podatka iz grafičkog interfejsa i unesenu vrednost dodeli nekoj promenljivoj programa.

Pogodno je klasu definisati u posebnom modulu, npr. `forma_za_unos`, jer se kasnije može višestruko koristiti:

```
from tkinter import * # sva imena iz biblioteke

class FormaZaUnos:
    def __init__(self):
        prozor = Tk() # Kreiranje grafičkog prozora
        prozor.title("Forma za unos") # Naslov prozora
        # Kreiranje i dodavanje okvira 1 u prozor
        frame1 = Frame(prozor)
        frame1.pack()

        # Dodavanje teksta i polja za unos u frame1
        tekst = Label(frame1, text="Unesite svoje ime: ")
        self.ime = StringVar() # def. tipa vrednosti
        poljeIme = Entry(frame1, textvariable = self.ime)

        # Dodavanje tastera za potvrdu unosa u frame1
        tasterGetIme = Button(frame1, text="Potvrdi",\
                               command=self.obradiTaster)
        tekst.grid(row = 1, column = 1)
        poljeIme.grid(row = 1, column = 2)
        tasterGetIme.grid(row = 1, column = 3)

        prozor.mainloop() # Petlja za obradu događaja

    def obradiTaster(self):
        print("Uneli ste vrednost: " + self.ime.get())
```

Program koji koristi ovako definisanu formu za unos omogućava da se podatak unese putem grafičkog umesto konzolnog interfejsa i može imati oblik:

```

""" Jednostavna forma za unos podataka.

    Klasa FormaZaUnos kreira formu za unos podataka
    u grafičkom prozoru.
    Forma se sastoji od polja za unos imena i tastera
    za potvrdu unosa.

"""

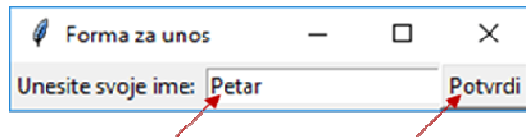
from forma_zu_unos import FormaZaUnos

# Kreiranje grafičkog interfejsa i unos podataka
forma = FormaZaUnos()

print("Putem forme uneseno je ime: ",
      forma.ime.get())

```

Pokretanjem programa kreira se novi grafički prozor u kome se prikazuje forma za unos podataka, Sl. 57. Forma omogućava unos i editovanje tekstualne vrednosti u grafičkom polju i završetak unosa pomoću tastera Potvrdi.



Sl. 57. Primer grafičkog interfejsa za unos podataka

Rezultat unosa program proverava ispisom poruka na konzolni izlaz:

```

Uneli ste vrednost: Petar
Putem forme uneseno je ime: Petar
>>>

```

Poruke prikazuju sadržaj polja forme ime nakon potvrde unosa podatka tasterom Potvrdi; prva poruka je iz metoda obradiTaster(), a druga iz glavnog programa.

Ovo je veoma jednostavan primer forme za unos samo jednog, tekstualnog podatka. Forme za unos mogu da sadrže veći broj različitih elemenata navedene u prethodnoj tabeli, čime se omogućava ne samo direktni unos podataka, već i unos izborom jedne ili više ponuđenih vrednosti.

Pitanja za ponavljanje

1. Koja je razlika između klasičnog i objektno orijentisanog programiranja?
2. Šta je objekt u objektno orijentisanom programiranju i koja su mu osnovna svojstva?
3. Koji su članovi objekta i kako se u jeziku Python vrši pristup članovima objekta?
4. Objasnite odnos klase i objekta u objektno orijentisanom programiranju. Dajte primer i predstavite ga grafički u jeziku UML.
5. Kako se u jeziku Python definiše korisnička klasa?
6. Koja je oblast definisanosti promenljivih kreiranih u definiciji klase kao *self.<naziv_promenljive>*?
7. Kako se i zašto vrši skrivanje polja u definiciji klase i kako se ipak može omogućiti pristupi tim podacima?
8. Šta je nasleđivanje u objektno orijentisanom programiranju i šta se time postiže? Dajte grafički primer nasleđivanja pomoću UML.
9. Kako se u jeziku Python definiše korisnička klasa kao kao proširenje postojeće klase?
10. Objasnite pojam nadjačavanje metoda (*method overriding*) u nasleđivanju klasa.
11. Šta je dinamičko povezivanje (*dynamic binding*) u nasleđivanju klasa?

Literatura

1. Liang D., *Introduction to Programming Using Python*, Pearson Education, 2013
2. Dierbach C., *Introduction to Computer Science Using Python: A Computational Problem-Solving Focus*, John Wiley & Sons, 2013
3. Hetland M. L., *Beginning Python: From Novice to Professional*, 3rd Ed, Apress, 2017
4. Knuth D. E., *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, 3rd Ed, Addison-Wesley, 1997
5. Sebesta R. W., *Concepts of Programming Languages*, 11th Ed, Pearson Education, 2016
6. Levitin A., *Introduction to the design & analysis of algorithms*, 3rd Ed, Pearson Education, 2012
7. Hetland M. L., *Python Algorithms: Mastering Basic Algorithms in the Python Language*, 2nd Ed, Apress, 2014
8. Conery J. S., *Explorations in Computing: An Introduction to Computer Science and Python Programming*, Chapman and Hall/CRC Press, 2015
9. Summerfield M., *Programming in Python 3: A Complete Introduction to the Python Language*, 2nd Ed, Pearson Education, 2010
10. Guido van Rossum et al., *The Python Language Reference*, Python Software Foundation, March 2017
11. Downey A. B., *Think Python*, 2nd Ed, O'Reilly Media, 2016
12. Lutz M., *Learning Python*, 5th Ed, O'Reilly, 2013
13. Kalb I., *Learn to Program with Python*, Apress, 2016
14. Goodrich M. T., Tamassia R., Goldwasser M. H., *Data Structures and Algorithms in Python*, John Wiley & Sons, 2013
15. Necaie R. D., *Data Structures and Algorithms Using Python*, John Wiley & Sons, 2011
16. Cormen T. H., Leiserson C. E., Rivest R. L., Stein C., *Introduction to Algorithms*, 3rd Ed, MIT Press, 2009
17. Aho A. V., Hopcroft J. E., Ullman J. D., *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974
18. Rashid T., *Make Your Own Mandelbrot*, Create Space, 2014

19. Vikipedija www.wikipedia.org, 2017
20. Early Office Museum
http://www.officemuseum.com/data_processing_machines.htmPython
www.python.org
21. Python Enhancement Proposal PEP20 on Python website
<https://www.python.org/dev/peps/pep-0020/>
22. What do different aphorisms in The Zen of Python mean? - Quora
<https://www.quora.com/%2FWhat-do-different-aphorisms-in-The-Zen-of-Python-mean&usg=AFQjCNECqX4IRarI0VY3Bq4tevy84KR-bw>
23. The 2017 Top Programming Languages
<http://spectrum.ieee.org/computing/software/the-2017-top-programming-languages>
24. Python zvanični sajt <https://www.python.org>
25. Quora Python software <https://www.quora.com/What-is-the-most-famous-software-written-in-Python>
26. Python 3 referentni priručnik <https://docs.python.org/3/reference/>
27. Gregorian date method <http://alcor.concordia.ca/~gpkatch/gdate-method.html>
28. Kineski zodijak <https://www.powerofpositivity.com/chinese-zodiac-2016/>
29. Dijkstra's algorithm for shortest paths
<http://code.activestate.com/recipes/119466-dijkstras-algorithm-for-shortest-paths/>
30. Codebases: Millions of Lines of Code
<http://www.informationisbeautiful.net/visualizations/million-lines-of-code/>
31. Python - Trees as Lists
<http://www.cs.utsa.edu/~wagner/python/tree.lists/trees.html>
32. Python 3 tutorijel <https://docs.python.org/3/tutorial/>
33. Learn python3 in Y Minutes <https://learnxinyminutes.com/docs/python3/>
34. Stanford CS41 Python <http://stanfordpython.com/>
35. Nauka o podacima (Data Science)
https://en.wikipedia.org/wiki/Data_science
36. Rosetta Code <https://www.rosettacode.org/wiki/Factorial>
37. ASCII tabela <https://www.lookuptables.com>

Prilozi

- A. Programska okruženja za razvoj softvera
- B. Instalacija programskih biblioteka i dodataka
- C. Specifikacije formata

A. Programska okruženja za razvoj softvera

Standardni sistem za razvoj softvera u jeziku Python, koji se koristi u ovom udžbeniku, obuhvata prevodilac (interpreter), standardne programske biblioteke i programski editor IDLE.

Programski editor IDLE omogućava interaktivni unos i izvršavanje pojedinačnih naredbi u režimu interpretera, kao i unos, čuvanje, izvršavanje i testiranje kompletnih programa. Editor ima i neka napredna svojstva, kao što su označavanje sintaksnih elemenata bojama (*syntax highlighting*), automatsko dopunjavanje prilikom unosa naredbi (*auto-completion*), automatsku indentaciju koda, itd.

Standardna biblioteka jezika Python obuhvata veliki broj modula, koji se automatski instaliraju u osnovnoj verziji sistema. Biblioteka sadrži podršku za ugrađene tipove i strukture podataka, rad s fajlovima, pristup funkcijama operativnog sistema, Internetu, grafičkom korisničkom interfejsu, funkcijama za testiranje, distribuciju i održavanje softvera itd. U ovom materijalu se, za implementaciju primera programa, uz ugrađene tipove i strukture podataka, funkcije i metode obrade izuzetaka, koriste sledeći moduli iz standardne biblioteke:

- *string* - osnovne operacije sa stringovima;
- *datetime* - osnovni tipovi podataka za datum i vreme;
- *math* - matematičke funkcije;
- *cmath* - funkcije za rad s kompleksnim brojevima;
- *random* - generisanje pseudoslučajnih brojeva;
- *os* - dodatne funkcije operativnog sistema;
- *urllib.request* - biblioteka za pristup Veb objektima.

Detaljni pregled sadržaja standardne Python biblioteke u verziji 3.6 može se naći na <https://docs.python.org/3/library/index.html>.

Ostala razvojna okruženja zasnivaju se na različitim distribucijama i programskim bibliotekama.

Poznatiji alternativni prevodioci jezika Python su:

- *Cython* - pravi prevodilac za proširenje jezika Python, koji omogućava da performanse programa razvijenih pretežno u jeziku Python budu uporedive s programima u jeziku C. Koristi se za razvoj modula za standardnu verziju jezika Python.
- *Iron Python*- implementacija prevodioca jezika Python, razvijena u jeziku C#, koja generiše kod za virtuelne mašine aplikativnih okvira .NET i Mono.
- *Jython* - implementacija prevodioca jezika Python koja generiše kod za virtuelnu mašinu jezika Java.

Popularni programski editori, posebno namenjeni razvoju softvera u jeziku Python su *PyCharm* i Veb sistem *Jupyter/IPython Notebook*.

Podrška razvoju složenih programa u jeziku Python postoji i u obliku dodataka za profesionalna integrisana razvojna okruženja *Eclipse* i *Visual Studio*, koja se koriste za timski razvoj složenih softverskih sistema, često u više programskih jezika istovremeno.

Poznatije nestandardne *distribucije* jezika Python su:

- *Anaconda* - besplatna distribucija jezika Python za Linux, Windows i Mac, najpopularnije programsko okruženje za istraživanja podataka (*Data Science*).
- *Enthought Canopy* - analitičko okruženje za Linux, Windows i Mac, s besplatnom podrškom za laku instalaciju više stotina Python analitičkih paketa i komercijalnom verzijom okruženja za analizu i razvoj softvera.
- *Python(x,y)* - Windows distribucija namenjena naučnim primenama, koja se oslanja na integrisano okruženje Spyder s podrškom za razvoj softvera u više popularnih programskih jezika.
- *WinPython* - besplatna prenosiva Windows distribucija jezika Python namenjena naučnim primenama i edukaciji, s velikim brojem preinstaliranih Python biblioteka.
- *Pyzo* - interaktivno razvojno okruženje Pyzo IDE, koje se oslanja na besplatnu Python distribuciju Anaconda.

B. Instalacija programskih biblioteka i dodataka

Standardno programsko okruženje za razvoj softvera u sistemu Python obuhvata prevodilac (interpreter), standardne biblioteke i programski editor IDLE.

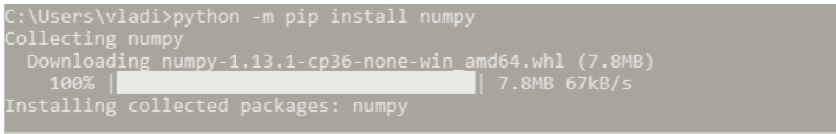
Istovremeno je razvijen i veliki broj drugih programskih biblioteka i dodataka, koji predstavljaju softver otvorenog koda kao i sam standardni Python. Veb sajt PyPI (*Python Package Index*) je repozitorijum softvera u jeziku Python, koji je u vreme završetka ovog materijala imao 116.728 paketa. Instalacija većine ovih programskih modula (biblioteka, paketa) može se izvršiti iz komandne linije, pomoću priloženog instalacionog programa.

Ukoliko se umesto standardne instalacije koristi neka od nestandardnih distribucija programskog okruženja Python, najčešće namenjenih istraživanju podataka i drugim naučnim primenama, veliki broj najvažnijih programskih biblioteka je već uključen u samu distribuciju, a za upravljanje instalacijama se koristi poseban interni alat ili funkcija za upravljanje verzijama biblioteka.

U standardnoj distribuciji, najnovija verzija nekog modula, zajedno s neophodnim modulima od kojih on zavisi, instalira se komandom:

```
python -m pip install <naziv_modula>
```

Npr. modul *numpy* instalira se iz komandne linije kao na slici:



```
C:\Users\vladi>python -m pip install numpy
Collecting numpy
  Downloading numpy-1.13.1-cp36-none-win_amd64.whl (7.8MB)
    100% |#####| 7.8MB 67kB/s
Installing collected packages: numpy
```

Može se zahtevati instalacija samo *određene verzije* neke programske biblioteke pomoću relacionog izraza, npr.

```
python -m pip install <naziv_modula>==1.0.4 #određ.ver.
```

Takođe se može zadati i *najniža* ili *najviša verzija* biblioteke za instalaciju pomoću operatora ">" i "<", koji se koriste uz navodnike, npr.

```
python -m pip install "<naziv_modula>>=1.0.4" #min.ver.
```

Ažuriranje instaliranih biblioteka vrši se eksplicitno, naredbom oblika:

```
python -m pip install --upgrade <naziv_modula>
```

Neke programske biblioteke nemaju kompletnu specifikaciju paketa na repozitorijumu *Python Package Index*, pa ih je potrebno instalirati manuelno.

Primer takve biblioteke je *scipy*. Za manuelnu instalaciju, prvo je potrebno pronaći i preuzeti instalacionu verziju biblioteke za instalirani operativni sistem i verziju programskog okruženja Python.

Npr. za 64-bitni operativni sistem Windows i verziju Python 3.6, na lokaciji [1]⁶ pronađe se i preuzme fajl pod nazivom:

```
scipy-0.19.1-cp36-cp36m-win_amd64.whl
```

Instalacija modula vrši se iz komandne linije operativnog sistema. Na kraju instalacije dobija se poruka o uspešnom završetku:

```
C:\Users\vladi\Downloads>pip install scipy-0.19.1-cp36-cp36m-win_amd64.whl
Processing c:\users\vladi\downloads\scipy-0.19.1-cp36-cp36m-win_amd64.whl
Requirement already satisfied: numpy>=1.8.2 in c:\users\vladi\appdata\local\programs\python\python36\lib\site-packages (from scipy==0.19.1)
Installing collected packages: scipy
Successfully installed scipy-0.19.1
```

1. Gohlke C., Unofficial Windows Binaries for Python Extension Packages
<http://www.lfd.uci.edu/~gohlke/pythonlibs/>

⁶ Na ovoj lokaciji nalaze se 32/64-bitne binarne verzije velikog broja naučnih biblioteka (paketa) za standardnu Windows didtribuciju programskog jezika CPython

C. Specifikacije formata

Opšti oblik standardne specifikacije formata u jeziku Python je [1]:

```
<format_spec> ::= [[<fill>]<align>][<sign>][#][0][<width>]
                  [group_option][.precision][type]
<fill>          ::= bilo_koji_znak
<align>         ::= "<" | ">" | "=" | "^"
<sign>          ::= "+" | "-" | " "
<width>         ::= celi_broj
<group_option> ::= "_" | ","
<precision>    ::= celi_broj
<type>         ::= "b" | "c" | "d" | "e" | "E" |
                  "f" | "F" | "g" | "G" | "n" |
                  "o" | "s" | "x" | "X" | "%"
```

Značenje pojedinih opcija standardne specifikacije formata je:

Oznaka	Opcije	Opis
<fill>	<i>znak</i>	prazna mesta se mogu popuniti bilo kojim znakom
<align>	<	levo poravnanje raspoloživog prostora (string)
	>	desno poravnanje (podrazumevano za brojeve)
	=	dopuna nulama između predznaka i prve cifre broja
	^	centriranje u raspoloživom prostoru
<sign>	+	predznak za pozitivne i negativne vrednosti
	-	predznak samo negativne vrednosti
	" "	prazno mesto za pozitivne, a minus za negativne
#		koristi se alternativna format konverzije
0		popuna nulama
<width>	<i>broj</i>	celi broj označava minimalnu dužinu polja
<group_option>	_	separator grupa cifara (hiljade)
	,	separator grupa cifara (hiljade)
<precision>	<i>broj</i>	broj decimalnih mesta za formate f i F, ukupan broj cifara za formate g i G ili broj mesta za nenumeričke
<type>	s	string format (podrazumeva se za stringove)
	b	binarni zapis broja
	c	ispis broja kao Unicode znaka
	d	ispis broja u decimalnom sistemu
	o	ispis broja u oktalnom sistemu
	x	ispis u heksadecimalnom sistemu (mala slova)
	X	ispis u heksadecimalnom sistemu (velika slova)
	n	kao d ili g, ali koristi lokalne separatore znakova
	e	eksponencijalni ispis decimalnog broja (6 decimala)

E	eksponencijalni ispis s velikim E
f	ispis decimalnog broja u fiksnom formatu
F	kao f, samo ispisuje velikim slovima NAN i INF
g	opšti format. Zaokružuje broj na zadani broj decimala i prikazuje u fiksnom ili eksp. formatu, zavisno od raspona vrednosti
G	kao g, samo su sve oznake velikim slovima
%	množi sa 100 i prikazuje u formatu f , uz znak %
none	slično formatu g, najmanje jedno decimalno mesto

Npr. specifikacija za prikaz celog broja "`*<+7,d`" sadrži niz oznaka sa značenjem:

- * - znak za popunu praznih mesta,
- < - levo poravnanje,
- + - prikaz predznaka vrednosti,
- 7 - širina polja (minmalna),
- , - znak razdvajanja grupa cifara (hiljade),
- d - celobrojni tip.

Primer upotrebe:

```
>>> print("{:>*+7,d}".format(1234))
+1,234*
```

Specifikacija za prikaz decimalnog broja "`^-09.3f`" sadrži oznake sa značenjem:

- ^ - poravnanje u centru polja,
- - predznak samo za negativne brojeve,
- 0 - znak za dopunu nulama,
- 9 - ukupna širina polja,
- .3 - broj decimalnih mesta (iza decimalne tačke),
- f - decimalni tip vrednosti, prikaz u fiksnom formatu.

Primer upotrebe:

```
>>> print("{:^-09.3f}".format(123.4))
0123.4000
```


CIP - Каталогизација у публикацији
Народна библиотека Србије, Београд

004.43PYTHON(075.8)
004.42.045(075.8)

МИШКОВИЦ, Владислав, 1957-

Osnove programiranja - Python / Vladislav Miškovic. - 3.
izmenjeno i dopunjeno izd.

- Beograd : Univerzitet Singidunum, 2020 (Beograd : Caligraph).

- XIII, 220 str. : ilustr. ; 24 cm

Na vrhu nasl. str.: Fakultet za informatiku i računarstvo, Tehnički fakultet.

- Tiraž 1.600.

- Bibliografija: str. 212-213.

ISBN 978-86-7912-728-0

a) Програмски језик "Python" b) Објектно оријентисано програмирање

COBISS.SR-ID 19386633

© 2020.

Sva prava zadržana. Nijedan deo ove publikacije ne može biti reprodukovan u bilo kom vidu i putem bilo kog medija, u delovima ili celini bez prethodne pismene saglasnosti izdavača.



Vladislav Mišković



OSNOVE PROGRAMIRANJA PYTHON



Udžbenik je nastao kao osnovna literatura za uvodni kurs programiranja na određenim studijskim programima Fakulteta za informatiku računarstvo i Tehničkog fakulteta Univerziteta Singidunum u Beogradu.

Jezik Python izabran je kao prvi jezik za učenje osnovnih principa programiranja jer manje opterećuje programera obaveznim elementima programskog koda. Minimalistički zamišljen programski jezik omogućava preusmeravanje pažnje programera s problema kodiranja programa u konkretnom programskom jeziku na rešavanje problema, koje uključuje analizu, razvoj algoritama i primenu opštih principa programiranja.

U materijalu se izlažu osnove sintakse i semantike aktuelne verzije programskog jezika Python, kao i tipični primeri njegove upotrebe, kojima se ilustruje pristup rešavanju određenih kategorija problema uz pomoć računara.