Miloš A. Kovačević

OSNOVE PROGRAMIRANJA u Pajtonu

drugo dopunjeno izdanje

Akademska misao

Beograd 2024.

Miloš A. Kovačević
OSNOVE PROGRAMIRANJA U PAJTONU
Drugo dopunjeno izdanje

Recenzenti:
Dr Jelica Protić
Dr Milo Tomašević
Dr Branislav Bajat

Izdavači:
Akademska misao, Beograd
Univerzitet u Beogradu - Građevinski fakultet, Beograd

Štampa
Akademska misao, Beograd

Tiraž
100 primeraka

Dizajn naslovne strane
Autor

Ilustracija na koricama
Marko i Anđela Kovačević - *Jesen*

Ilustracija na početku svake glave
Luka Kovačević - *Poletanje*

ISBN 978-86-7466-993-8

Mesto i godina izdanja
Beograd, 2024.

© 2024 Miloš A. Kovačević

GRAĐEVINSKI FAKULTET U BEOGRADU
WWW.GRF.BG.AC.RS

majci Ružici i ocu Anđelku,

bez dobrog temelja, kuća je klimava

Tamari,

da dvoje budu jedno

Milošu Cvetojeviću - Abi,

hvala ti za Sveznanje, Galaksiju i podršku tokom odrastanja

Sadržaj

Predgovor i

1 Algoritmi i programski jezici 1

1.1 Programiranje i svet oko nas 1

1.1.1 Sistemi, modeli i procesi 2

1.1.2 Informacije 3

1.2 Model računara 4

1.3 Algoritmi i heuristike 6

1.3.1 Algoritmi 6

1.3.2 Heuristike 9

1.4 Programski jezici 10

1.4.1 Mašinski i asemblerski jezik 11

1.4.2 Viši programski jezik - prevođenje i interpretacija 11

1.4.3 Tipovi programskih jezika 13

1.5 Zašto Pajton? 15

2 Objekti, tipovi i operacije 19

2.1 Interaktivni rad u razvojnom okruženju IDLE 19

2.2 Koliko je težak prvi Pajton program 22

2.2.1 Od oblasti problema do algoritma 22
2.2.2 Testiranje programa u IDLE okruženju 23
2.2.3 Analiza programa 23
2.2.4 Sintaksne i semantičke greške 25

2.3 Predstavljanje podataka i osnovne operacije 26

2.3.1 Objekti 26
2.3.2 Izrazi i promenljive, dodela vrednosti 29
2.3.3 Aritmetičko-logičke operacije i poređenja 33
2.3.4 Konstruktori osnovnih tipova 36
2.3.5 Promenljivost objekata 37

3 Kontrola toka. Iterativni algoritmi 39

3.1 Kontrola toka 39

3.1.1 Grananja 41
3.1.2 Petlje 45

3.2 Iterativni algoritmi 52

3.2.1 Metoda uzastopne aproksimacije 53
3.2.2 Metoda iterativne konstrukcije 54
3.2.3 Iterativna pretraga u prostoru rešenja 59

4 Funkcije i moduli. Rekurzivni algoritmi 69

4.1 Korisničke funkcije 70

4.1.1 Definisanje funkcije 70
4.1.2 Povratne vrednosti 74
4.1.3 Imenovani i opcioni parametri 79
4.1.4 Imenski prostori. Opseg važenja promenljivih 80

4.2 Organizacija izvornog koda **86**
4.2.1 Moduli 86
4.2.2 Paketi 89
4.2.3 Funkcije i moduli kao objekti 89
4.3 Rekurzivni algoritmi **91**
4.3.1 Rekurzija na delu 92
4.3.2 Prednosti i mane rekurzivnog pristupa 98

5 Kolekcije objekata: sekvence, skupovi i rečnici 103
5.1 Sekvence **104**
5.1.1 Tekst - tip `str` 105
5.1.2 Objekti i metode. Metode za rad sa tekstom 114
5.1.3 Liste - tip `list` 117
5.1.4 Tekstualne metode i liste 126
5.1.5 Torke - tip `tuple` 129
5.2 Skupovi i rečnici **133**
5.2.1 Skupovi - tip `set` 133
5.2.2 Rečnici - tip `dict` 138
5.3 Kolekcije i kombinatorne strukture **148**
5.3.1 Kombinacije 149
5.3.2 Permutacije i varijacije 155

6 Algoritmi zasnovani na slučajnim brojevima 163
6.1 Pseudoslučajni brojevi **164**
6.2 Funkcije za slučajni odabir i mešanje **167**
6.3 Probabilistički algoritmi **170**
6.3.1 Verovatnoća događaja i slučajne promenljive 170
6.3.2 Monte Karlo simulacija 173

7 Složenost algoritma. Pretraživanje i sortiranje 181
7.1 Merenje vremena **182**

7.2 **Analiza složenosti algoritma** **186**
7.2.1 Funkcija vremenske složenosti 186
7.2.2 Asimptotske notacije 188
7.3 **Pretraživanje** **190**
7.3.1 Linearno pretraživanje 191
7.3.2 Binarno pretraživanje 194
7.4 **Algoritmi sortiranja** **197**
7.4.1 Counting sort 197
7.4.2 Selection sort 200
7.4.3 Merge sort 203
7.4.4 Quick sort 207
7.5 **Redovi složenosti predefinisanih operacija nad kolekcijama** **210**
7.5.1 Složenost u radu sa listama 212
7.5.2 Složenost u radu sa rečnicima 213

8 Obrada grešaka u programu 217

8.1 **Trag i tipovi grešaka** **217**
8.2 **Defanzivno programiranje** **220**
8.2.1 Provera korektnosti ulaznih podataka 220
8.2.2 Model klijent-server 222
8.2.3 Informacija o stanju obrade u klijent-server modelu 223
8.3 **Greške kao izuzeci** **225**
8.3.1 Kritična sekcija. Kontrolna struktura `try - except` 226
8.3.2 Eksplicitno prosleđivanje izuzetaka. Naredba `raise` 229
8.3.3 Bezuslovno izvršavanje pri obradi izuzetaka. Naredba `finally` . . . 232

9 Tekstualne datoteke 235

9.1 **Pojam i interpretacija** **235**
9.2 **Sistem datoteka** **236**

9.3 Ulazno-izlazne operacije sa tekstualnim datotekama **244**
9.3.1 Čitanje iz tekstualne datoteke . . . 244
9.3.2 Upisivanje u tekstualnu datoteku . . . 246
9.4 Tekstualne datoteke u CSV formatu **254**
9.4.1 Parsiranje CSV datoteke . . . 255
9.4.2 Upisivanje tabelarnih podataka . . . 256
10 Apstraktni tipovi podataka - klase . . . 263
10.1 Apstrahovanje realnog sveta - objekti i klase **264**
10.1.1 Enkapsulacija - atributi i metode . . . 265
10.1.2 Definisanje klase . . . 267
10.1.3 Magične metode i preopterećenje operatora . . . 271
10.1.4 Klasni atributi i statičke metode . . . 279
10.1.5 Sakrivanje informacija . . . 283
10.2 Kompozicija i agregacija **293**
10.3 Nasleđivanje **303**
10.3.1 Pretraga metoda pri pozivu. Prepisivanje metode . . . 307
10.3.2 Red i stek - primeri nasleđivanja . . . 308
11 Pajton za inženjere: NumPy i Matplotlib . . . 317
11.1 Procesiranje višedimenzionalnih nizova brojeva - NumPy **317**
11.1.1 Kreiranje matrice . . . 322
11.1.2 Pristup elementima matrice - osnovno indeksiranje . . . 326
11.1.3 Matrične funkcije i operacije . . . 338
11.1.4 Poređenje, logičke operacije i napredno indeksiranje . . . 350
11.1.5 Spajanje i razdvajanje matrica i rearanžiranje elemenata . . . 359
11.2 Grafičko predstavljanje podataka - Matplotlib **371**
11.2.1 Linijski dijagram i osnovni elementi grafičkog prikaza . . . 371
11.2.2 Pravougaoni i kružni dijagram. Histogram . . . 377
11.2.3 Dijagram površi - grafik realne funkcije dve promenljive . . . 380

Bibliografija 387

Spisak slika 389

Spisak tabela 393

Indeks problema 395

Indeks pojmova 397

Predgovor

Predgovor drugom dopunjenom izdanju

Poštovani čitaoci,

pred vama je drugo dopunjeno izdanje knjige *Osnove programiranja u Pajtonu*. Zahvaljujući ubrzanom razvoju veštačke inteligencije, kao i sve većem značaju nauke o podacima, programski jezik Pajton je, u prethodnih pet godina, povećao svoju popularnost i predstavlja prvi izbor za jezik u kome se izučavaju osnovni koncepti programiranja. Zbog toga sam se odlučio da poboljšam tekst prethodnog izdanja, uklonim uočene greške i dodam glavu koja se bavi programskim paketima *NumPy* i *Matplotlib*. Ovi paketi objedinjavaju neophodne alate za sve one koji se bave kreiranjem i analizom matematičkih modela realnih sistema i analizom numeričkih podataka. NumPy se bavi matričnim izračunavanjima, dok Matplotlib predstavlja skup alata za grafičko predstavljanje podataka.

Pored teksta nove glave, koja nosi ime *Pajton za inženjere: NumPy i Matplotlib*, u okviru ovog izdanja dodati su i delovi teksta vezani za sledeće mogućnosti jezika:

- uslovni izrazi - glava 3,
- oblikovanje teksta pomoću formatirajućeg stringa - glava 5,
- operatori raspakivanja - glava 5,
- slučajno uzorkovanje pomoću odabira sa ponavljanjem - glava 6.

Nadam se da će učinjene izmene pomoći čitaocima da lakše savladaju neophodne tehnike programiranja.

Na kraju, zahvaljujem se kolegi Mladenu Nikoliću, kao i recenzentima Jelici Protić, Milu Tomaševiću i Branislavu Bajatu, koji su svojim vrednim komentarima pomogli da se tekst drugog izdanja što više poboljša. Posebno se zahvaljujem gospodinu Aleksandru Bakoču i kompaniji Daran Group, koji su omogućili štampanje kompletnog tiraža drugog izdanja.

Srećno!

Beograd, decembar 2023. *Autor*

Predgovor prvom izdanju

Poštovani čitaoci,

pred vama je knjiga *Osnove programiranja u Pajtonu*, namenjena kako studentima prve godine Građevinskog fakulteta u Beogradu, tako i svima onima koji žele da pođu na ne tako lak, ali izazovan put ovladavanja veštinom programiranja. Kao i svaka druga veština, programiranje zahteva od početnika *strpljenje* i *upornost*. Olakšavajuća okolnost u procesu učenja ogleda se u mogućnosti da se uloženi rad vrlo brzo testira, tako što se novonastali program izvrši na računaru. Računar će, za razliku od čoveka, uvek izvršiti samo ono što mu se naredi. To često neće biti ono što se želelo, ali problem nije do računara već do programera koji svoje želje nije uspeo da pretoči u ispravan niz naredbi. Zato se u knjizi, pored programskog jezika, naglasak stavlja na osnovne tehnike za rešavanje pojedinih tipova problema.

Algoritmi, koji predstavljaju precizno definisane postupke za rešavanje problema na računaru, ilustrovani su putem kratkih programa realizovanih u Pajtonu. Jezik Pajton izabran je kako zbog svoje jednostavnosti, tako i zbog svoje otvorenosti i opšte prisutnosti, naročito u naučnoj i inženjerskoj zajednici. U knjizi se ne razmatraju sve mogućnosti i specifičnosti Pajtona. Za tu svrhu može se koristiti brojna literatura koja se bavi detaljnom specifikacijom jezika.[1] Slično, rešenja pojedinih problema ne koriste uvek pristup svojstven stilu ekspertskog programiranja u Pajtonu. Ovo je učinjeno kako bi se naglasila opšta ideja nekog postupka rešavanja primenjiva i u drugim modernim jezicima. Algoritamski pristup zahteva i izvesno poznavanje srednjoškolske matematike, što ne bi trebalo da predstavlja nepremostivu prepreku. Pojedini matematički problemi pružaju dobru podlogu za objašnjenje često korišćenih tehnika u inženjerskom programiranju. Jedostavan matematički aparat koristi se i prilikom analize složenosti pojedinih algoritama.

U Pajtonu se *svi* podaci, počevši od broja jedan pa do samog programa, predstavljaju objektima odgovarajućeg tipa. Objektima se mogu pridodati i željena ponašanja pa se uz njihovu pomoć mogu simulirati različiti realni sistemi. Zbog toga, knjiga pored *proceduralnog*, promoviše i *objektno orijentisani* pristup koji u centar rešavanja problema stavlja objekte kao apstrakcije podataka ili delove složenih sistema. Autor se nada da će čitaoci tako steći neophodnu inženjersku veštinu sagledavanja problema (sistema) kroz njegove jednostavnije potprobleme (podsisteme) i uočavanja veza između njih.

Programiranje se najbolje uči na primerima. Knjiga sadrži priličan broj ilustrativnih problema čija rešenja su detaljno objašnjena, a rad pridruženih programa ilustrovan na pogodan način. Čitaoci se ohrabruju da ih sve *obavezno* prostudiraju jer se u okviru njihovih rešenja ilustruju različiti teoretski i praktični koncepti.

Knjiga se ne bavi elementima programiranja koji se odnose na izgradnju korisničkog interfejsa, rad sa bazama podataka i drugim tehnikama za kreiranje poslovnih aplikacija.

[1] Preporučuje se dokumentacija na adresi `http://docs.python.org/reference/` .

Ova odluka učinjena je u nameri da se pažnja čitaoca usmeri na *osnovne tehnike* koje se primenjuju u svim oblastima računarstva. Po savladavanju izložene materije biće daleko jednostavnije upustiti se u dublje programerske vode.

Na kraju, zahvaljujem se asistentima Dušanu Isailoviću i Mariji Petronijević na pomoći pri izradi slika, recenzentima Jelici Protić i Mladenu Nikoliću na uočenim nedostacima i korisnim predlozima u pogledu izložene materije, kao i Milosavi Mijović i Radmili Žeželj Ralić na lekturi i korekturi teksta. Njihovo zalaganje značajno je poboljšalo prvobitnu verziju knjige. Posebno se zahvaljujem Ministarstvu prosvete, nauke i tehnološkog razvoja Republike Srbije, koje je obezbedilo sredstva za štampanje knjige u okviru projekta "Adaptacija master studija geoinformatike u skladu sa savremenim potrebama privrede i tržišta rada".

Srećno!

Beograd, decembar 2017. *Autor*

1. Algoritmi i programski jezici

Poznata knjiga iz oblasti popularne psihologije, *Put kojim se ređe ide*, autora Skota Peka, počinje rečenicom “Život je težak!”. Verovatno bi se većina složila sa ovim iskazom, ne umanjujući time božansku lepotu i duboki smisao života. Pitajući se zašto je to tako, mnogi bi odgovorili da je to zbog velikog broja problema koji se u životu pojavljuju i koje treba rešavati, često u neželjeno vreme i pod ograničavajućim okolnostima. Uspešno rešavanje različitih problema koji se pojavljuju na životnom putu čini život tako očaravajućim iskustvom i vodi ka duhovnom i materijalnom napretku. U nauci i inženjerskim disciplinama, bez izuzetka, rešavanje problema je u osnovi svake uspešne aktivnosti. Ova knjiga bavi se izučavanjem osnovnih metoda programiranja za različite inženjerske potrebe. U tu svrhu izučava se programski jezik Pajton[1], ali se naglasak, pored jezika, stavlja na puteve rešavanja problema uz pomoć kreativnog razmišljanja.

1.1 Programiranje i svet oko nas

Programiranje predstavlja moćan alat kojim se, putem računara, modeluje kako realni, tako i kreira virtuelni svet. U tu svrhu razmatra se kontekst ove mlade, ali dinamične i sveprisutne naučne discipline.

[1] Engl. *Python*.

1.1.1 Sistemi, modeli i procesi

Naš svet je izuzetno složen, bilo da se posmatra njegova materijalna ili nematerijalna priroda. Zbog toga se u nauci, za potrebe izučavanja realnih *sistema*, pribegava njihovom značajnom pojednostavljivanju kroz izradu različitih *modela*. Posmatrani sistem može se shvatiti kao celina sastavljena od povezanih elemenata koji, u međusobnoj interakciji, ostvaruju svrhu njegovog postojanja. Elementi koji čine sistem mogu i sami biti sistemi, kao što i sam sistem može biti element u okviru većeg sistema. Ukoliko se radi o materijalnim elementima (na primer, cigle, malter, ...), onda je sistem *materijalan* (zgrada), a ako su elementi apstraktni pojmovi izvedeni *generalizacijom* iz realnih entiteta, onda je sistem *konceptualan* (na primer, pravni sistem). U zavisnosti od svrhe izučavanja sistema, gradi se odgovarajući model koji zadržava samo one osobine neophodne za njegovu krajnju upotrebu. Svođenje sistema na njegovo pojednostavljenje – model, vrši se putem procesa *apstrahovanja*, pojma koji se često pojavljuje u tekstu knjige (slika 1.1).

Slika 1.1: Različiti nivoi apstrakcije: fizički sistem kuće (levo). Arhitektonska osnova kao model kuće (sredina). Tačka na mapi, sa adresom, modeluje kuću na nivou plana grada (desno). Nivo apstrakcije bira se u zavisnosti od toga gde se model primenjuje.

Posmatra se primer zgrade (materijalni sistem) koja se, između ostalog, sastoji od sledećih elemenata: temelj, ploče, grede, stubovi, zidovi. Temelj sam po sebi predstavlja sistem sastavljen od šipova, naglavnih greda i ploče. Pre gradnje, arhitekta koristi materijalni model – maketu, da bi dočarao izgled budućeg objekta. Građevinski inženjeri koriste različite konceptualne modele zgrade u toku projektovanja: AutoCAD crtež koji definiše oblik, dimenzije i veze sastavnih elemenata, jednačine koje opisuju statičko i dinamičko ponašanje zgrade, predmer, predračun i druge.

Konceptualni model nastaje iz misaone predstave o sistemu u glavi pojedinca. Misaona predstava prelazi u model uz upotrebu opšte prihvaćenih formalnih alata i jezika, poput matematičkih formalizama ili programskih jezika kao što su Pajton, Java ili C. Od interesa za nauku i inženjerske discipline naročito su bitni konceptualni modeli poput matematičkog modela i *računarskog programa* (na primer, simulacije).

Realni sistemi mogu se posmatrati i kroz *procese* koji se odvijaju između pojedinih delova – elemenata sistema. Proces se definiše kao niz aktivnosti koje se, zarad dobijanja odgovarajućih izlaza, primenjuju nad posmatranim ulaznim veličinama. Na primer, u biološkom sistemu biljaka, proces fotosinteze podrazumeva iskorišćavanje svetlosne

energije sunca kako bi se, iz vode i ugljen dioksida pohranjenih u biljci, sintetisali šećeri koji sadrže hemijsku energiju potrebnu za život biljke. Sunce, voda i ugljen dioksid čine ulaze, dok se u hemijskim reakcijama oslobađa kiseonik koji, pored šećera, čini izlaz iz procesa. Procesi se često modeluju na konceptualnom nivou, korišćenjem matematičkih, fizičkih i hemijskih formalizama, ali i uz pomoć simulacije realizovane putem računarskog programa.

1.1.2 Informacije

U toku svog postojanja, sistemi i procesi podrazumevaju razmenu energije i *informacija*, kako između svojih sastavnih elemenata, tako i sa spoljnim okruženjem. Definisanje pojma informacije u opštem slučaju zavisi od oblasti primene. Tako se, u domenu komunikacija, informacija definiše kao poruka koja je primljena i shvaćena, a u domenu menadžmenta, kao saznanje koje čini osnovu za donošenje odluka. Često se pojam informacije pogrešno zamenjuje pojmom *podatka*. Podatak predstavlja sirovi oblik koji, tek pošto se stavi u odgovarajući kontekst, postaje informacija. Na primer, broj 7.8 je sam po sebi podatak, ali ako se odnosi na prosečnu minimalnu temperaturu u celzijusima u Beogradu u mesecu aprilu, onda postaje informacija.

Informacija smanjuje neodređenost sistema iz koga potiče ili koga opisuje. *Stanje* u kome se sistem nalazi, u nekom trenutku vremena, može se opisati informacijama o elementima sistema i trenutnim interakcijama između njih. Empirijske nauke prikupljaju informacije o svetu putem merenja. Na osnovu ovih informacija izgrađuju se konceptualni modeli sistema i procesa. Ovi modeli mogu poslužiti kako za *predviđanje* ponašanja sistema u budućnosti, tako i za *upravljanje* sistemom, njegovim dovođenjem u željeno stanje.

Posmatra se primer vremenske prognoze za koju se pravi model vremena na određenoj geografskoj teritoriji. Stanje vremena može se opisati promenljivim poput temperature, vlažnosti, pritiska ili brzine vetra. Kada instrument meri temperaturu vazduha, on prevodi fizičko opažanje (podizanje žive u mernoj cevi) u predefinisan skup simbola (brojevi) koji kvantifikuju pojavu. Informacija se najčešće zapisuje putem brojeva, teksta, slike i zvuka. Međutim, za potrebe prenosa na daljinu, obrade u računaru ili skladištenja na fizičkom medijumu, ona se *prevodi* u reprezentaciju pogodnu za konkretnu tehnološku implementaciju – *proces kodiranja*. Na primer, u radio prenosu, zvučne informacije se reprezentuju elektromagnetnim talasima odgovarajuće amplitude i frekvencije. Kodirana informacija vraća se, na mestu prijema, u izvorni oblik u procesu *dekodiranja*. Da bi se razumele, dve strane treba da poznaju pravilo po kome se vrši kodiranje (dekodiranje).

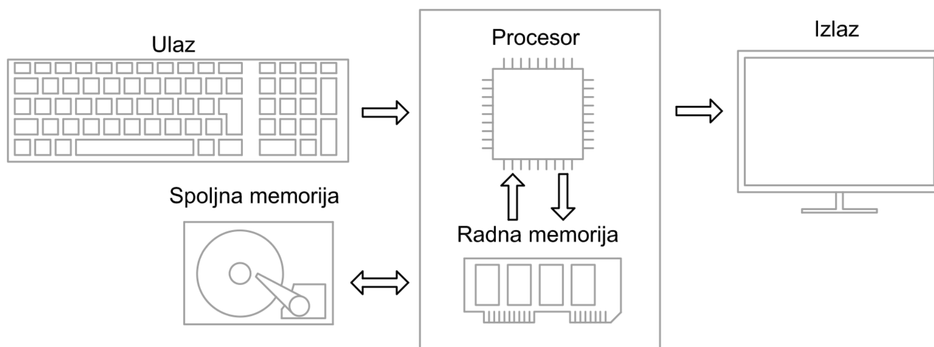
Programiranje, kao put izrade računarskog programa, predstavlja aktivnost modelovanja različitih sistema na konceptualnom nivou.

Prevođenje kompleksnog sistema ili procesa u odgovarajući programski model,

podrazumeva dobro poznavanje njegovog funkcionisanja, jasno uočavanje ograničenja koja definišu upotrebu modela, kao i korišćenje različitih alata u domenu softverskog inženjerstva. Otuda se navod Skota Peka, sa početka ovog teksta, može parafrazirati kao "*Programiranje je teško*". Autor se nada da će ova knjiga olakšati čitaocu savladavanje programiranja kao kreativne inženjerske discipline.

1.2 Model računara

Računar je sistem za obradu podataka (informacija). Način obrade definiše se *programom* koji se i sam može smatrati informacijom o potrebnom postupku obrade. Da bi se bolje shvatila dinamika izvršavanja programa, neophodno je razumeti osnovne principe funkcionisanja računara. Kako se radi o kompleksnom sistemu, za razumevanje se koristi jednostavni konceptualni model predstavljen na slici 1.2. Model je dobio ime po svom tvorcu - fon Nojmanova arhitektura računara.[2]



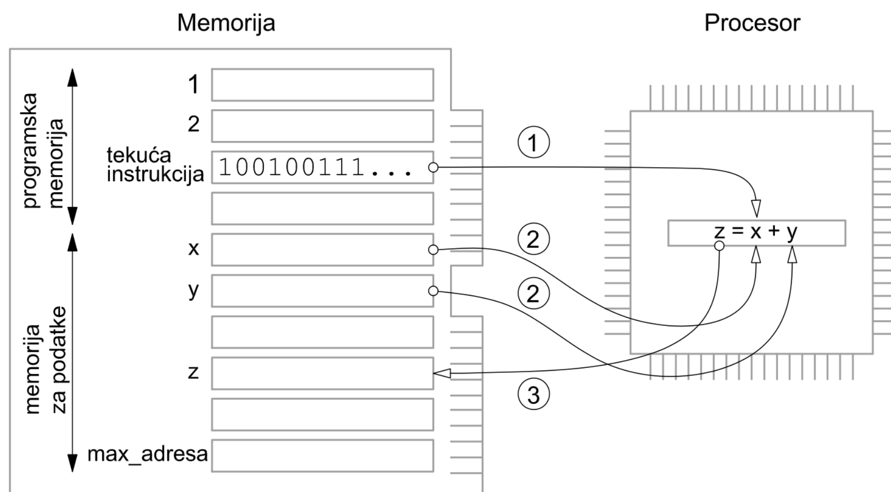
Slika 1.2: Uprošćeni model računara po fon Nojmanovoj arhitekturi.

Centralno mesto u modelu zauzimaju *procesor*[3] i *radna memorija*. U radnoj memoriji su pohranjeni kako podaci, tako i program koji predstavlja recept za željenu obradu. Da bi se informacije mogle obrađivati, potrebno ih je na adekvatan način predstaviti u memoriji. Kako je memorija sastavljena od ćelija koje se mogu naći u dva električna stanja, to se stanje svake ćelije može označiti binarnom cifrom[4] 0 ili 1. Jednim bitom mogu se predstaviti najviše dve mogućnosti (na primer, informacija o položenom ispitu: 0 – student pao, 1 – student položio). Pošto se realni sistemi i procesi, koji se modeluju programom, karakterišu velikim brojem stanja u kojima se mogu naći, to se memorijske ćelije interpretiraju po grupama od po 8, 16, 32, 64 i više ćelija (bita). Kako se sa n bita može predstaviti najviše 2^n različitih kombinacija, to se jednim bajtom (8 bita) može predstaviti $2^8 = 256$ različitih mogućnosti. Jedan bajt dovoljan je da se predstave svi simboli na standardnoj tastaturi računara. Nizovi

[2] John von Neumann (1903-1957) - mađarsko-američki matematičar i fizičar.
[3] Engl. *Central Processing Unit* – CPU.
[4] Engl. *BInary digiT* - bit.

nula i jedinica, pohranjeni u memoriji, bili bi potpuno nerazumljivi kada ne bi bilo poznato pravilo preslikavanja (kodiranje) koje definiše značenje svake kombinacije. Memorija zapravo sadrži određene količine naelektrisanja koje se interpretiraju kao nule i jedinice – evo primera kako se bitima apstrahuje fizičko stanje u stvarnosti na nivou konceptualnog modela!

Memorijske lokacije (celine od najčešće 32 ili 64 bita) označene su *adresama* koje predstavljaju prirodne brojeve u rasponu od 0 do *max_adresa*. Procesor može pristupati svakoj memorijskoj lokaciji navodeći njenu adresu. Brzina pristupanja svakoj lokaciji je *konstantna* i ne zavisi od njene adrese. Prilikom pristupanja, procesor čita staru ili upisuje novu vrednost. On je zadužen za izvršavanje aritmetičkih operacija poput sabiranja ili množenja, odnosno logičkih operacija poput poređenja dve vrednosti. Procesor čita niz instrukcija iz dela memorije u kome se nalazi program. Kada procesor učita tekuću instrukciju, prvo dešifruje njeno značenje na osnovu binarne reprezentacije (dekodiranje), a potom je izvršava. Instrukcije obično sadrže adrese lokacija iz dela memorije sa podacima nad kojima treba obaviti traženu operaciju, kao i adresu lokacije gde treba smestiti novi rezultat (slika 1.3). Redosled izvršavanja instrukcija je sekvencijalan, ali se može promeniti tako što se, u zavisnosti od ishoda odgovarajuće logičke operacije, skače na datu adresu u programskoj memoriji. Na taj način, u zavisnosti od ulaznih vrednosti i međurezultata, mogu se realizovati različite putanje u programu.



Slika 1.3: Izvršavanje programa: procesor dohvata i dekodira instrukcije (1), čitanje vrednosti memorisanih na adresama *x* i *y* (2), izračunavanje i upis rezultata u lokaciju sa adresom *z* (3).

Početni podaci unose se u radnu memoriju putem *ulaznih uređaja* kao što je tastatura, a rezultat rada programa prikazuje se na nekom od *izlaznih uređaja* poput ekrana ili štampača. Kada se isključi napajanje, memorija se briše pa programe i podatke treba upamtiti na nekoj od trajnih (spoljnih) memorija poput čvrstog diska ili fleš memorije. Treba istaći da većina savremenih računarskih sistema radi po *nešto izmenjenom* modelu, ali je, sa stanovišta daljeg izlaganja, ova aproksimacija dovoljno dobra.

1.3 Algoritmi i heuristike

Računari obavljaju *tačno* one operacije koje im se eksplicitno zadaju. Ništa ni više ni manje od toga! Prostor mogućih ishoda pri izvršavanju pojedinih aplikacija toliko je veliki pa se čini da mašina donosi samostalne odluke van uticaja programa. Međutim, da bi se problem rešio, potreban je precizno definisan recept kako se to čini. Recepti mogu biti u formi *algoritma* ili *heuristike*.

1.3.1 Algoritmi

Algoritam je precizno definisani postupak za izvršavanje određenog zadatka ili rešavanje odgovarajućeg problema. Na primer, da bi se broj 10 pomnožio brojem 2, potrebno je rezultat množenja postaviti na nulu, pa onda na njega dodavati broj 10 dva puta. Ovde se recept sastoji od određenih *elementarnih postupaka* (operacija), kao što je postavljanje rezultata na nulu i sabiranje. Pojam *precizno definisanog postupka* odnosi se na konačan skup elementarnih operacija iz koga se bira niz operacija koje vode do rešenja u *konačno* mnogo koraka.

Da bi neki postupak predstavljao algoritam potrebno je dokazati njegovu *korektnost*, što u prethodnom slučaju proizilazi iz same definicije operacije množenja. Očigledno, mnogi problemi mogu korektno da se reše na više načina. U primeru množenja, bilo je moguće postaviti rezultat na nulu, pa onda dodavati na njega deset puta broj 2. Međutim, ovaj algoritam je manje *efikasan* od prve varijante jer treba obaviti osam operacija sabiranja više. Efikasnost se najčešće meri utroškom resursa potrebnih za rešavanje problema. Poželjno je da se vreme izvršavanja održava unutar poznate granice koja zavisi od veličine ulaznih podataka o čemu će više reči biti u glavi 7.

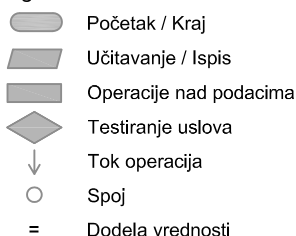
Algoritam za množenje brojeva 2 i 10 nije naročito koristan jer se bavi specijalnim slučajem množenja. Od većeg interesa je *opšti* algoritam koji množi proizvoljne prirodne brojeve x i y. Efikasno rešenje podrazumeva postavljanje rezultata z na nulu, pa potom proveru da li je x veće od y. Ako jeste, y puta bi se ponavljalo dodavanje broja x na broj z. U suprotnom, na z bi se x puta dodavao broj y. *Ulazna informacija* za množenja dva prirodna broja predstavljena je promenljivim veličinama x i y, a *izlazna informacija* promenljivom veličinom z. Algoritam se može zapisati na različite načine, a na slici 1.4 prikazana su dva najčešća: *pseudokod* i *dijagram toka*.

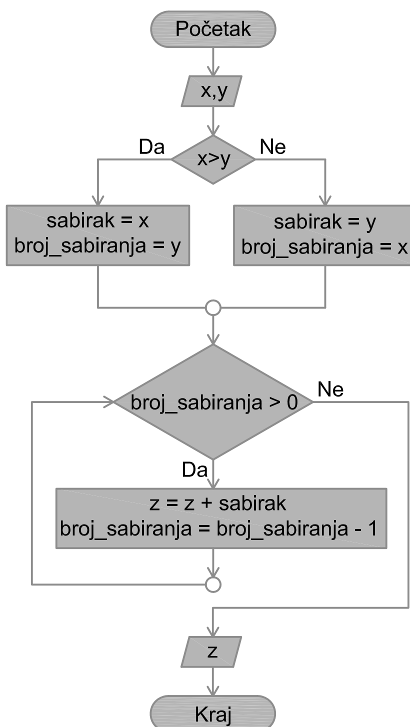
Proces iz realnog sveta modeluje se algoritmom, pri čemu su ulazne i izlazne veličine procesa ujedno i ulazne i izlazne veličine algoritma. Iznalaženje efikasnog algoritma predstavlja centralnu temu kojom se bave računarske nauke.

Problem 1.1 — Pogađanje broja. Osoba zamišlja prirodan broj od 1 do 100. Potom odgovara, sve do pogotka, da li je trenutno ponuđen broj veći, manji ili jednak zamišljenom broju. Predložiti algoritam za pogađanje zamišljenog broja iz što manjeg broja pokušaja. ■

```
Ulaz: x,y
z ← 0
Ako x > y
        sabirak ← x
        broj_sabiranja ← y
Inače
        sabirak ← y
        broj_sabiranja ← x

PonavljajDok broj_sabiranja > 0
        z ← z + sabirak
        broj_sabiranja ← broj_sabiranja-1

Izlaz: z
```

Legenda:





Slika 1.4: Pseudokod kao modifikacija prirodnog jezika (levo) i dijagram toka (desno). Pseudokod obično koristi engleske reči, ali su ovde, zbog razumevanja, navedene srpske. Na slici je dat algoritam množenja dva prirodna broja. Naredbe **Ako-Inače** i **PonavljajDok** omogućavaju alternativno i ponovljeno izvršavanje nizova uvučenih naredbi.

Razmatraju se algoritmi A, B i C, dati pseudokodom sa slike 1.5. Algoritam A podrazumeva izbor brojeva redom od 1 do 100. Ovde osoba odgovara sa "pogodak" ili "broj je veći". U *najnepovoljnijem* slučaju, kada je zamišljen broj 100, potebno je 100 pogađanja. Algoritam koji pronalazi rešenje tako što ispituje *sve* moguće varijante nekog problema, pripada tipu *algoritama grube sile*.[5] On je po pravilu *neefikasan*, sem u jednostavnim slučajevima kada nema puno potencijalnih rešenja. Kada bi se na ovakav način pogađao broj između 1 i 10^6, vrlo brzo bi se odustalo od ovakve igre!

U slučaju B, bira se prvo 10. Ukoliko osoba odgovori sa "broj je manji", biraju se redom brojevi od 1 do 9. U suprotnom, bira se 20 i postupak se ponavlja. Na primer, ako je zamišljen broj 25, bira se sledeća sekvenca: 10, 20, 30, 21, 22, 23, 24, 25. Najnepovoljniji slučaj nastupa za zamišljeni broj 99, kada se pogađa u devetnaestom pokušaju: 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 91, 92, 93, 94, 95, 96, 97, 98, 99. B je

[5] Engl. *Brute force algorithm.*

očigledno efikasniji od A jer se *prostor mogućih rešenja* smanjuje u svakom koraku za deset mogućnosti, sve dok se ne locira desetica u kojoj se nalazi zamišljeni broj.

```
Algoritam A

Ulaz: zamišljen
pokušaj ← 1
broj_pokušaja ← 1
Prikaži pokušaj
PonavljajDok pokušaj ≠ zamišljen
      broj_pokušaja ← broj_pokušaja + 1
      pokušaj ← pokušaj + 1
      Prikaži pokušaj

Izlaz: pokušaj, broj_pokušaja
```

```
Algoritam B

Ulaz: zamišljen
pokušaj ← 10; brojPokušaja ← 1
Prikaži pokušaj
PonavljajDok pokušaj < zamišljen
      broj_pokušaja ← broj_pokušaja + 1
      pokušaj ← pokušaj + 10
      Prikaži pokušaj

Ako pokušaj ≠ zamišljen
      pokušaj ← pokušaj - 10
      PonavljajDok pokušaj ≠ zamišljen
      broj_pokušaja ← broj_pokušaja + 1
            pokušaj = pokušaj + 1
            Prikaži pokušaj

      Izlaz: pokušaj, broj_pokušaja
```

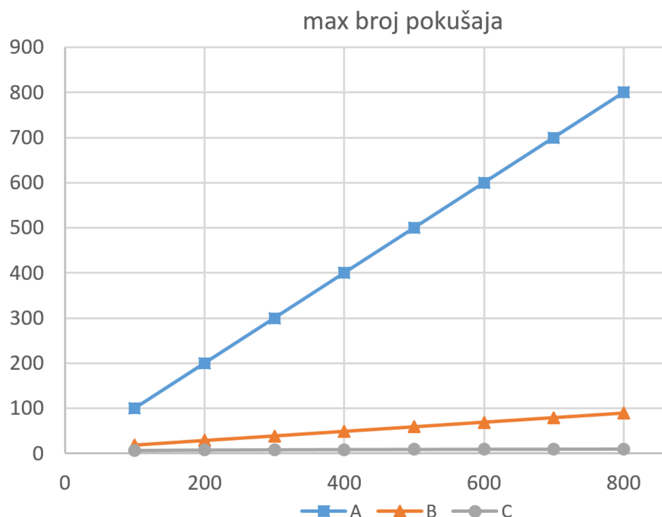
```
Algoritam C

Ulaz: zamišljen
dole ← 1; gore ← 100
pokušaj ← ceo deo od (dole + gore) / 2
broj_pokušaja ← 1
Prikaži pokušaj
PonavljajDok pokušaj ≠ zamišljen
      Ako pokušaj < zamišljen
            dole ← pokušaj + 1
      Inače
            gore ← pokušaj - 1

      pokušaj ← ceo deo od (dole + gore) / 2
      broj_pokušaja ← broj_pokušaja + 1
      Prikaži pokušaj

Izlaz: pokušaj, broj_pokušaja
```

Slika 1.5: Pogađanje broja: A – gruba sila; B – nešto efikasniji algoritam; C – najefikasniji algoritam tipa *podeli pa vladaj*. Primetiti da u slučaju A nije neophodno voditi evidenciju o broju pokušaja jer je on jednak zamišljenom broju.

Najefikasniji je algoritam C. Ovde se prvo proba sa brojem 50. Ako je tačan broj manji, bira se 25, a ako je veći, onda 75. Pri svakom pokušaju, skup mogućih izbora polovi se na dva jednaka dela pa se kompleksnost problema smanjuje. U najnepovoljnijem slučaju, kada je zamišljen broj 100, pogađa se iz sedmog puta: 50, 75, 88, 94, 97, 99, 100. C je karakterističan primer za algoritme tipa *podeli pa vladaj*,[6] koji svode problem na manje potprobleme čije je rešavanje jednostavnije od početnog problema.

U opštem slučaju, kada se pogađa broj od 1 do n, maksimalan broj pokušaja za algoritme A, B i C iznosi n, $\lfloor n/10 \rfloor + 9$, odnosno $\lceil log_2 n \rceil$. Operator $\lfloor x \rfloor$ zaokružuje broj na prvi manji, a $\lceil x \rceil$ na prvi veći ceo broj. Na slici 1.6 ilustrovana je efikasnost predloženih algoritama u funkciji ulazne veličine n koja diktira složenost problema. Algoritmi A i B imaju *linearnu*, a C *logaritamsku* složenost. Osmišljavanje efikasnog algoritma često zahteva veliki intelektualni napor, čak i izvesno nadahnuće.

[6] Engl. *Divide and conquer*.

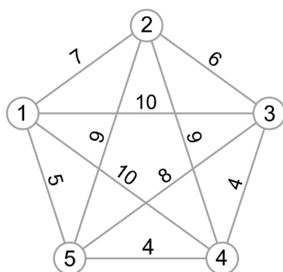


Slika 1.6: Pogađanje broja: efikasnost algoritama A, B i C. Na apscisi su maksimalne vrednosti zamišljenog broja, a na ordinati odgovarajući broj pokušaja u najnepovoljnijem slučaju. Algoritam C ima logaritamsku složenost pa se smatra izuzetno efikasnim rešenjem.

1.3.2 Heuristike

Mnogi problemi nemaju poznato rešenje u opštem slučaju (za sve kombinacije ulaznih veličina) ili je njihovo tačno rešavanje neefikasno. U tom slučaju pribegava se *heuristikama* koje daju približna, ali za kontekst primene zadovoljavajuća rešenja. Heuristika, slično algoritmu, predstavlja niz elementarnih operacija iz konačnog skupa, pri čemu se ne može dokazati njena korektnost u opštem slučaju. Posmatra se sledeći problem:

Problem 1.2 — Trgovački putnik. Za n gradova date su putne razdaljine između svaka dva grada. Trgovački putnik nalazi se u prethodno određenom gradu. On potom treba da poseti sve preostale gradove tačno jednom i da se vrati u polazni grad. Pronaći sekvencu gradova koja daje najkraći ukupni put (slika 1.7) ■



dužina(1 - 2 - 3 - 4 - 5 - 1) = 26
dužina(1 - 3 - 4 - 2 - 5 - 1) = 37
dužina-pohlepna(1 - 5 - 4 - 3 - 2 - 1) = 26

Slika 1.7: Problem trgovačkog putnika. Graf sa pet numerisanih gradova, brojevi na putevima označavaju dužine. Polazi se iz grada označenog brojem 1.

Na prvi pogled problem je jednostavan: gradovi se prvo numerišu od 1 do n. Sekvenca posećivanja definiše se kao uređeni raspored brojeva koji označavaju gradove (svi gradovi zastupljeni u sekvenci tačno jednom). Potom, treba generisati skup svih mogućih sekvenci (S) koje polaze iz početnog grada i pridružiti im njihove dužine. Ovde treba sabrati i dužinu puta od poslednjeg do prvog grada iz sekvence jer se trgovac vraća u početnu poziciju. Prilikom generisanja sekvence iz S, poredi se njena dužina sa do tad najkraćom pronađenom sekvencom. Ako je tekuća sekvenca kraća, treba ažurirati promenljivu *min_put* koja čuva najkraću dužinu, kao i promenljivu *min_sekvenca* koja memoriše sekvencu. Pošto se isprobaju sve sekvence iz S, u promenljivoj *min_sekvenca* nalazi se rešenje problema.

Ponuđeni recept predstavlja tipičan primer algoritma grube sile. Čitaocima se sugeriše da napišu pseudokod za ovaj algoritam, pri čemu mogu koristiti složene operacije poput *generiši_sledeću_sekvencu* ili *ima_još_sekvenci*.

Kako je prvi grad već izabran, broj svih mogućih sekvenci za problem sa n gradova je $(n-1)!$.[7] Otuda je algoritam grube sile primenljiv samo za malo n. Neka procesor ima radni takt od 1GHz. Ako svaka mašinska instrukcija traje jedan takt (a obično traje više), proizilazi da se ona obavi za jednu nanosekundu (10^{-9} s). Kada bi procesor obrađivao jednu sekvencu u samo jednoj instrukciji, za jednu nanosekundu (opet nerealno), algoritmu bi već za $n = 18$ ($17! = 355687428096000$ns), trebalo 4.12 dana da pronađe optimalno rešenje. Za $n = 21$, algoritam bi utrošio 77.1 godina!

Zbog neprimenljivosti algoritma grube sile, za problem trgovačkog putnika može se primeniti neka od heuristika koje značajno redukuju broj obrađenih sekvenci. Ovde ne postoji garancija da će rešenje biti najbolje, ali je često dovoljno dobro pa heuristika nalazi svoju primenu u praksi. Primer najjednostavnije heuristike podrazumevao bi da se, po izboru početnog grada, svaki sledeći u sekvenci bira tako da je *najbliži prethodnom*, u odnosu na sve ostale koji nisu već posećeni. Ovakav tip heuristike (algoritma) naziva se *pohlepnom pretragom*[8] jer se, pri konstrukciji rešenja, uvek bira *najbolji* ishod u odnosu na *trenutnu* poziciju u prostoru svih mogućih izbora. Pohlepna pretraga *ne garantuje* da će konačno rešenje biti i najbolje. U primeru sa slike 1.7, uočava se da je pohlepna putanja ujedno i najbolja.

1.4 Programski jezici

Algoritmi se u računaru realizuju odgovarajućim programom. Program može biti realizovan korišćenjem različitih sistema simbola (jezika), kojima se reprezentuju pojedine operacije podržane od strane računarskog sistema. Ovde će se razmatrati mašinski, asemblerski i viši programski jezik. Biće reči i o različitim tipovima viših programskih jezika.

[7] Broj svih permutacija skupa od k elemenata: $k! = k \cdot (k-1) \cdot \ldots \cdot 2 \cdot 1$

[8] Engl. *Greedy search*.

1.4.1 Mašinski i asemblerski jezik

U uprošćenom modelu računara procesor izvršava program instrukciju po instrukciju, pri čemu se pod instrukcijom podrazumeva jedna od konačno mnogo *elementarnih* operacija podržanih od dotičnog model procesora. Elementarne operacije predstavljene su odgovarajućim nizovima nula i jedinica koje su smeštene u radnoj memoriji računara (u toku izvršavanja) ili na spoljnoj, trajnoj memoriji (kada računar ne izvršava program ili je ugašen). Skup svih elementarnih operacija, sa odgovarajućom binarnom reprezentacijom i pravilima za pristup memorijskim lokacijama, definiše *mašinski jezik* za datu familiju procesora.

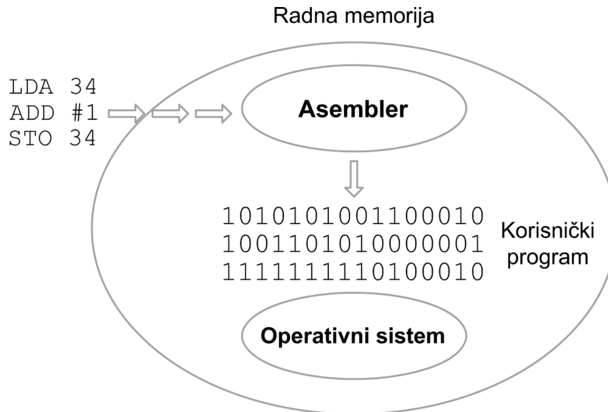
Korišćenje mašinskog jezika za pisanje programa predstavlja izuzetno *neproduktivan* i *greškama podložan* zadatak. Programeri prvih računarskih sistema morali su da pamte nepregledne nizove nula i jedinica da naznače potrebne instrukcije, vrednosti podataka i adrese memorijskih lokacija. Sa druge strane, korišćenjem elementarnih mašinskih operacija, složeniji zadaci se rešavaju mnogo teže nego kada bi se koristili kompleksniji alati. Treba zamisliti koliko bi naporno bilo da se, svaki put kada je u programu potrebno izračunati ostatak celobrojnog deljenja, koriste samo instrukcije sabiranja, promene znaka i poređenja dve vrednosti. Na primer, daleko jednostavnije bi bilo da procesor poseduje elementarnu operaciju $ostatak(x,y)$.

Kako računari nemaju premca u pogledu automatizacije dosadnog posla, rani programeri došli su na ideju da osmisle jezik koji će olakšati proces programiranja tako što bi se program sastojao od skraćenica i brojnih konstanti koje su se odnosile na pojedine instrukcije, podatke i adrese. Ovaj *asemblerski jezik* automatski se prevodio u odgovarajuće binarno kodirane instrukcije. Zbog toga nije bilo potrebno pamtiti nepregledne nizove nula i jedinica, već skraćenice pojedinih, procesorom podržanih operacija. Asemblerski jezik iziskuje postojanje *asemblera*, programa u radnoj memoriji koji prevodi skraćenice jezika u mašinski kod (slika 1.8).

Na slici je prikazan i skup programa pod imenom *operativni sistem*. Operativni sistem automatski počinje sa radom kada se pokrene računar. Njegova uloga je da unapredi vezu između programera i računara: on procesira unos sa tastature, pokreće asembler i prevedeni korisnički program, prikazuje rezultate na nekom od izlaznih uređaja i slično. Uvođenjem asemblera i operativnog sistema, programiranje je postalo jednostavnije, ali je program i dalje ostao *zavisan* od tipa procesora i konkretnog računarskog sistema. Problem *izražajnosti* jezika još uvek nije rešen: programer mora da kombinuje elementarne instrukcije kako bi realizovao složenije operacije.

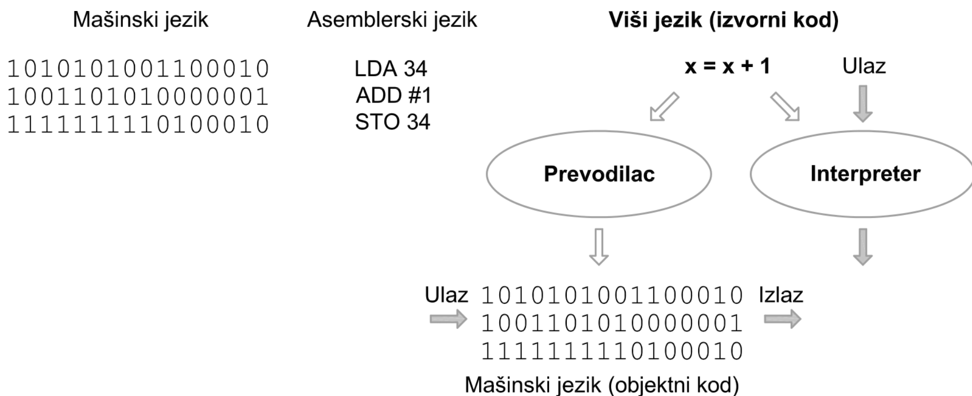
1.4.2 Viši programski jezik - prevođenje i interpretacija

Izražajnost je jedna od ključnih osobina jezika. Idealno bi bilo kada bi računar primao naredbe u formi prirodnog jezika, umesto što mu se mora obrazlagati svaki korak. Međutim, to u opštem slučaju predstavlja izuzetno težak zadatak. U želji da se poveća izražajnost, kao i da se omogući što veća nezavisnost od tipa i organizacije sistema,



Slika 1.8: Asemblerski jezik kao viši nivo apstrakcije u odnosu na mašinski jezik. Program koji uvećava sadržaj memorijske lokacije sa adresom 34 za 1. Prvo se u akumulator procesora učitava vrednost sa željene lokacije (LDA adresa); potom se na nju dodaje broj 1 (ADD #1) pa se rezultat upisuje na isto mesto u memoriji (STO 34). Asembler prevodi skraćenice i dekadne (ili heksadekadne) brojne konstante u odgovarajuće mašinske naredbe.

uvodi se novi nivo apstrakcije: *viši programski jezik*. Viši programski jezici (C, C++, Java, Pajton i drugi) uvode čitljive opise za složenije operacije i koncepte, pribiližavajući se prirodnom jeziku. Ovi opisi se transformišu u niz elementarnih operacija za ciljni računarski sistem (slika 1.9). Na ovaj način ostvaruje se *programska prenosivost*, a programer dobija alat koji omogućava prirodniju realizaciju kompleksnih zadataka.



Slika 1.9: Viši programski jezik uvodi jezičke konstrukcije koje su bliske matematičkom i prirodnom jeziku, čime se postiže nezavisnost od mašinskog koda konkretne mašine. Na svakom sledećem nivou apstrakcije rešenje je značajno kraće i jasnije!

Program napisan u višem programskom jeziku označava se kao *izvorni kod*. Izvorni kod je lako razumljiv za čoveka, ali se mora prevesti u mašinski jezik uz pomoć drugog programa – *prevodioca*. U procesu prevođenja, naredbe višeg programskog jezika

zamenjuju se brojnim mašinskim instrukcijama koje realizuju željeni efekat, a dobijeni rezultat često se zove i *objektni kod*. Prenosivost se u praksi ostvaruje kroz postojanje različitih prevodilaca koji isti jezik prevode u mašinske jezike *ciljnih* računarskih sistema. Kada se program jednom prevede u objektni kod, može se direktno pokretati na računaru bez potrebe ponovnog prevođenja.

Drugi pristup u izvršavanju izvornog koda, umesto prevodioca, koristi program *interpreter*. Interpreter čita naredbe višeg jezika i izvršava ih jednu po jednu, koristeći sopstveni skup operacija. On preuzima ulogu kontrolora koji, u ime programa, zahteva unos podataka sa ulaza i prikazuje rezultate na izlazu. Umesto da se, kao kod prevođenja, program izvršava direktno u procesoru, izvršavaju se direktno instrukcije interpretera koje sprovode željene akcije. Izvršavanje izvornog koda uz pomoć interpretera je, zbog dodatnog nivoa apstrakcije, *sporije* nego u slučaju prevedenog koda! Odakle onda potreba za interpretacijom?

Pristup sa interpreterom omogućava da se program *brže* doteruje jer u *fazi razvoja* postoji česta potreba za testiranjem, kako pojedinih delova, tako i celine. Posle svake izvršene naredbe višeg jezika, ako se pojavi greška, interpreter staje sa radom pa se problematičan deo može odmah ispraviti, a program potom i ponovo pokrenuti. Nema potrebe za prevođenjem u objektni kod pri svakoj izmeni u izvornom kodu. Ovaj pristup je, zbog svoje *interaktivnosti*, zahvalniji i u procesu učenja. Interpretirani jezici često se koriste za izgradnju prototipa. Kada se program potpuno istestira, koriste se prevodioci da proizvedu objektnu verziju koja će se koristiti u *fazi eksploatacije*. Danas su procesori i memorije bitno ubrzani pa se razlika između dva pristupa značajno smanjila. Zato se, često, prednost daje jezicima koji nude *brži razvoj* i *lakše održavanje*.

1.4.3 Tipovi programskih jezika

Viši programski jezici dele se prema načinu kako se program formira na *imperativne* i *deklarativne*. Imperativni jezici opisuju logiku za rešavanje problema navođenjem sekvence naredbi u programu (programski tok). Deklarativni jezici opisuju rešenje problema na formalan način bez navođenja preciznog programskog toka. Pojednostavljeno, jedni opisuju *kako* (imperativni), a drugi *šta* treba izračunati (deklarativni).

Program, prilikom izvršavanja, prolazi kroz određena *stanja* koja su definisana *promenljivim* u memoriji. Programer u imperativnom jeziku određuje *način* na koji pojedine promenljive menjaju svoja stanja, dok se kod deklarativnih jezika ovim pitanjem bave prevodilac ili interpreter na osnovu *opisa* željenog rezultata. Većina programskih jezika ima imperativnu prirodu pa se u ovoj knjizi razmatraju principi imperativnog programiranja. Od poznatijih deklarativnih jezika navode se Prolog i SQL.[9] Prolog je jezik opšte namene koji se najčešće koristi u oblasti veštačke inteligencije, dok se jezik SQL koristi za formiranje *upita* putem kojih se manipuliše bazom podataka.

Imperativni jezici razlikuju se po načinu *organizacije* izvornog koda. Dva domi-

[9] Engl. *Structured Query Language*.

nantna pristupa su *proceduralni* i *objektno orijentisani*. Proceduralni jezici organizuju program po grupama naredbi koje obavljaju određene logičke funkcionalnosti višeg nivoa – *procedure* (iscrtavanje trougla, rešavanje sistema jednačina i slično). Procedura može realizovati svoj zadatak, između ostalog, tako što poziva druge procedure. Na primer, za iscrtavanje trougla, tri puta se poziva procedura za crtanje duži kroz zadate krajnje tačke. Program predstavlja proceduru najvišeg nivoa koja, kombinovanjem ostalih procedura, rešava postavljeni zadatak. Proceduralni pristup omogućava formiranje *biblioteke procedura* za rešavanje određene klase problema, a koja se može *ponovo* upotrebiti u različitim programima kada su potrebne tražene funkcionalnosti. Među poznatijim proceduralnim jezicima su Basic, Cobol, Fortran, Pascal i C.

Ozbiljan nedostatak proceduralnog pristupa je u tome što su procedure, kojima se apstrahuju funkcionalnosti, *odvojene* od podataka. Iako svaka procedura ima svoje *lokalne* promenljive koje samo ona vidi i menja, ključni podaci u programu čuvaju se u *globalnim* promenljivim koje su svima dostupne pa se mogu *neželjeno* promeniti iz svake procedure. Osim sigurnosnog problema, zbog odvojenosti logike obrade od podataka, proceduralni jezici *neadekvatno* modeluju realne sisteme. Oni su pogodni za modelovanje pojedinih procesa koji se odvijaju unutar sistema, ali ne i za kompleksne sisteme u celini.

Objektno orijentisani jezici nastali su iz potrebe da se prevaziđu problemi u modelovanju složenih sistema koji iskrsavaju u proceduralnom pristupu. Osim toga, sedamdesetih godina prošlog veka, bilo je potrebno razviti pristup koji će povećati *produktivnost* u pisanju složenih programskih sistema. Ovde se pre svega misli na načine koji obezbeđuju ponovnu upotrebljivost već napisanog koda, jednostavno proširivanje novim funkcionalnostima i olakšano otklanjanje grešaka.

U objektno orijentisanom pristupu, program se gradi od *objekata* koji imaju svoje *osobine* i *ponašanje*. Osobine objekta (*atributi*) realizuju se kao promenljive koje ukazuju na podatke različitih tipova, a ponašanja kroz procedure (*metode*) koje definišu šta objekat može da radi, odnosno šta može da se radi nad njim. Jedino objektne metode mogu menjati *stanje* objekta koje je definisano vrednošću svih njegovih atributa u određenom vremenskom trenutku. Na ovaj način, objekat *objedinjuje* procesiranje (metode) i podatke (atribute).

Objektni pristup uvodi pojam *klase*. Klasa predstavlja skup objekata sa istim osobinama i istim ponašanjem. Zapravo, klasa predstavlja plan po kome se u programu kreiraju objekti. Posmatra se primer koji opisuje sve studente - klasa *Student*. Ova klasa definiše atribute *ime*, *prezime*, *broj_indeksa* i druge, kao i metode *predstavi_se*, *prijavi_ispit*, *polaži_ispit* i druge. Sada se mogu kreirati objekti (studenti) koji predstavljaju stvarne studente, na primer *Petar_Petrović* ili *Milica_Milić*, a koji imaju iste atribute (sa različitim vrednostima) i isto ponašanje (metode).

Program se izvršava tako što se kreiraju potrebni objekti koji u interakciji, kroz međusobno pozivanje metoda, realizuju potrebne zadatke. Očigledno je da se objektni koncept prirodno uklapa u okvir modelovanja realnog sistema sačinjenog od velikog

broja podsistema. Svaki podsistem modeluje se objektom odgovarajuće klase, a rad celog sistema oponaša se pozivima odgovarajućih metoda koje povezuju objekte onako kako su povezani elementi u sistemu.

Programiranje u objektno orijentisanom jeziku svodi se na slaganje već postojećih kockica (objekata željenih klasa) pa se akcenat stavlja, umesto na algoritamske korake, na *objektni dizajn*. Objektni dizajn razmatra izbor pojedinih klasa i veze između objekata u okviru sistema koji se modeluje. Kada je potrebno napisati nove funkcionalnosti ili modelovati specifične podsisteme, programer *izvodi* nove klase iz postojećih kroz postupak *nasleđivanja*. U postupku izvođenja se dodaju samo one funkcionalnosti koje predstavljaju specifičnost nove klase. Na primer, iz klase *Student* mogla bi se izvesti klasa *Apsolvent* koja zadržava sve atribute i metode studenta (i apsolvent je studnet), ali dodaje i neke specifičnosti – može da polaže u dodatnim rokovima, ide na stručnu praksu i slično. Među najpoznatije objektno orijentisane jezike spadaju C++, C#, Java i Pajton.

Na kraju ovog izlaganja biće pomenut i *funkcionalni* pristup programiranju koji je, po svojim osobinama, bliži deklarativnom nego imperativnom konceptu. U funkcionalnom pristupu program se gradi od funkcija koje su donekle slične procedurama, odnosno metodama. U najčistijoj formi ovog pristupa nema globalnih promenljivih koje čuvaju stanje programa, već se rezultat rada funkcije prosleđuje sledećoj funkciji na dalju obradu. Funkcije se mogu prosleđivati drugim funkcijama, a mogu biti i rezultat rada drugih funkcija. Programi pisani na ovaj način često su *kraći*, a zbog odsustva promenljivih *lakše* je dokazati njihovu korektnost. Međutim, organizacija današnjeg računarskog hardvera, izvedena iz fon Nojmanovog modela (glava 1.2), više pogoduje nefunkcionalnim pristupima. Tipičan predstavnik ove grupacije je Haskel.

Savremeni jezici primenjuju različite principe koji omogućavaju pisanje i organizaciju programa na različite načine. Na primer, u Pajtonu je moguće programirati samo proceduralno, samo objektno ili koristiti kombinaciju oba pristupa. Pajton podržava i pojedine elemente funkcionalnog programiranja koji se mogu mešati sa već pomenutim imperativnim pristupima.

1.5 Zašto Pajton?

Pajton je osmislio holandski programer Guido van Rossum krajem osamdestih godina prošlog veka. Jezik je dobio ime po popularnoj engleskoj seriji *Monty Python*. Pajton je interpretirani, objektno-orijentisani jezik visokog nivoa, namenjen za pravljenje svih vrsta aplikacija – od inženjerskih i naučnih, do poslovnih i web primena. Pravila jezika su jednostavna, a izvorni kod jasan i često dosta kraći nego u slučaju Java ili C/C++ ekvivalenta. Dostupan je za različite operativne sisteme poput Windows-a, Linux-a i Mac OS-a. Uz osnovnu instalaciju dobija se i besplatno *razvojno okruženje* IDLE koje uključuje interaktivni interpreter, editor teksta i alate za testiranje koda.

Programeri imaju na raspolaganju kvalitetnu standardnu biblioteku čiji programski moduli nude gotova rešenja za mnoge probleme. Jezik predstavlja idealnu platformu za *učenje programiranja* jer pruža alate za jednostavno prevođenje misaonog procesa u odgovarajući program. Na slici 1.10 prikazano je kretanje popularnosti pojedinih jezika u januaru 2023. godine, izvor PYPL. Popularnost jezika X izračunata je prema procentualnom učešću broja rezultata na Google upit ”X tutorial”, uzimajući u obzir sve jezike koji su zauzeli mesto na rang listi. Na kraju, treba istaći i veliku popularnost jezika u oblastima analize podataka i veštačke inteligencije.

Rank	Change	Language	Share	Trend
1		Python	27.93 %	-0.9 %
2		Java	16.78 %	-1.3 %
3		JavaScript	9.63 %	+0.5 %
4	↑	C#	6.99 %	-0.3 %
5	↓	C/C++	6.9 %	-0.5 %
6		PHP	5.29 %	-0.8 %
7		R	4.03 %	-0.2 %
8	↑↑↑	TypeScript	2.79 %	+1.0 %
9		Swift	2.23 %	+0.3 %
10	↓↓	Objective-C	2.2 %	-0.1 %
11	↑↑	Go	1.94 %	+0.7 %
12	↑↑↑	Rust	1.9 %	+0.9 %
13	↓	Kotlin	1.81 %	+0.1 %
14	↓↓↓↓	Matlab	1.63 %	-0.1 %
15	↑	Ruby	1.13 %	+0.3 %

Slika 1.10: Popularnost pojedinih programskih jezika u januaru 2023. godine. Pajton je na prvom mestu, ali sa trenutnim relativnim padom u odnosu na prethodnu godinu.

Prva verzija Pajtona 3.0 pojavila se još 2008. godine. Iako verzije 3.x donose značajna poboljšanja u odnosu na verzije 2.x, treba naglasiti da je veliki broj korisnih programa i biblioteka napisan u nekoj od varijanti tipa 2.x. Programi pisani u okruženju verzija 3.x nisu *kompatibilni unazad* sa starijim programima iz verzija 2.x. Pojam kompatibilnosti unazad odnosi se na osobinu programa, pisanog u novijoj varijanti jezika, da može da radi u starijem okruženju ako ne koristi nove, naprednije mogućnosti. Kako je cilj knjige da čitaoca uvede u svet programiranja i upozna ga sa osnovnim principima, autor će se truditi da u svakom sledećem izdanju koristi što noviju verziju tipa 3.x (trenutno 3.10).

ŠTA JE NAUČENO

- Model predstavlja pojednostavljenje realnog sistema. On zadržava samo one osobine koje su potrebne za projektovanu namenu.
- Apstrahovanje je proces kojim se sistem pojednostavljuje na model.
- Proces predstavlja niz interakcija između delova sistema u kojima se, uz utrošak potrebnih resursa, ulazne veličine transformišu u izlazne.
- O stanju u kome se sistem nalazi saznajemo na osnovu informacija koje on odašilje u spoljni svet.
- Podaci postaju informacije tek kada im se pridruži odgovarajući kontekst, odnosno značenje.
- Računar je sistem za obradu podataka. Način obrade definiše se programom koji može da modeluje kako procese, tako i sisteme.
- Računar se sastoji od procesora, radne memorije i ulazno-izlaznih uređaja. Procesor obrađuje podatke iz radne memorije prema programu koji je i sam uskladišten u njoj.
- Algoritam definiše tačan redosled operacija iz predefinisanog skupa, a kojima se rešava određeni zadatak. Najčešće se predstavlja pseudokodom ili dijagramom toka.
- Korektan algoritam rešava problem u opštem slučaju, odnosno za sve moguće kombinacije ulaznih parametara.
- Efikasan algoritam rešava problem u kraćem vremenu i/ili uz manji utrošak memorije.
- Algoritmi grube sile rešavaju problem tako što ispituju svako moguće rešenje.
- Algoritmi tipa "podeli pa vladaj" dele problem (ili prostor mogućih rešenja) na manje kompleksne delove, pa ih onda ponaosob lakše rešavaju.
- Pohlepni algoritmi biraju uvek onu varijantu koja je, u odnosu na tekuću poziciju u prostoru rešenja, najbolja. Na ovaj način se, polazeći iz početne situacije, ne garantuje dolazak do najboljeg rešenja.
- Heuristike su postupci za približno rešavanje problema čije je opšte rešavanje kompleksno ili nepoznato.
- Viši programski jezik definiše raznorodne mehanizme za algoritamsko rešavanje problema. Ovi mehanizimi se imenuju upotrebom pojedinih reči iz govornog jezika ili se koristi modifikovana matematička notacija. Korišćenjem programskog jezika, algoritam se transformiše u izvorni kod.

- Izvorni kod se prevodi u niz mašinskih instrukcija koje se izvršavaju od strane različitih tipova procesora. Prevođenje obavlja softver prevodilac, a prevedeni program može se pokretati proizvoljno mnogo puta.
- Izvorni kod može se, prilikom pokretanja, interpretirati komandu po komandu od strane programa koji se zove interpreter. Ovaj način pogodan je za razvoj prototipa jer se program zaustavlja kada naiđe na grešku, a interpreter ispisuje odgovarajuću poruku o njenom mestu i tipu.
- U zavisnosti od toga kako se program gradi (organizacija izračunavanja i podataka), viši programski jezici mogu biti proceduralni, objektno orijentisani, funkcionalni ili kombinacije prethodna tri tipa.
- Pajton spada u red najpopularnijih programskih jezika. Osobine poput interpretabilnosti, jasnoće i raspoloživosti velikog broja već gotovih funkcionalnosti, čine ga jezikom izbora za izučavanje veštine programiranja.

2. Objekti, tipovi i operacije

Programi obrađuju podatke različitih tipova. U ovoj glavi razmatra se kako se podaci predstavljaju u programu, kako im se pristupa i koje su osnovne operacije nad njima. Prvo se, vrlo kratko, pravi osvrt na interaktivan rad u okviru Pajtonovog razvojnog okruženja IDLE. Analizira se ilustrativni primer koji čitaoce upoznaje sa osnovnim koracima poput unosa sa tastature, ispisa na ekran i dodele vrednosti. Potom se diskutuju objekti i njihove vrednosti, promenljive, izrazi i osnovne aritmetičko logičke operacije.

2.1 Interaktivni rad u razvojnom okruženju IDLE

Programi se najčešće pišu uz pomoć odgovarajućeg softverskog paketa pod imenom *razvojno okruženje*.[1] Razvojno okruženje čini skup *softverskih alata* pomoću kojih se kreira program. U obavezne alate spadaju editor, debager, interpreter ili prevodilac i sistem za pomoć.

Editor je program u okviru koga se unosi tekst izvornog koda. Za razliku od editora opšte namene, razvojni editor omogućava označavanje različitih elemenata jezika različitim bojama, automatsko kompletiranje naredbi, predlaganje pojedinih stilova kodiranja za određene jezičke strukture, označavanje grešaka i slično. Navedene funkcionalnosti imaju za cilj da olakšaju i ubrzaju proces programiranja. Najveći broj

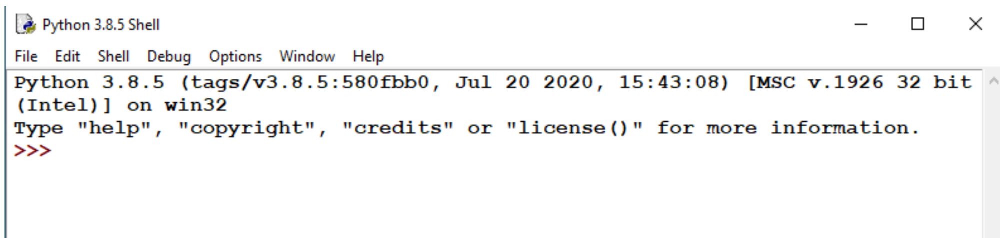
[1] Engl. *Integrated Developement Environment* - IDE.

editora ima ugrađen modul za kreiranje *grafičkog korisničkog interfejsa*.[2] Ovde se, uz pomoć vizuelnih alata, kreiraju forme koje se odnose na prozore, menije, dugmiće, tekstualna polja za unos, padajuće liste i druge grafičke elemente programa. Kreiranje formi propraćeno je *automatskim* generisanjem izvornog koda u okviru editora.

Debager je program kojim se testira rad programa u toku izvršavanja. Programer može pratiti vrednosti svih promenljivih u programu i na taj način uočiti razloge za pojavljivanje greške. Debager predstavlja svojevrsni skener kojim se rad programa razotkriva do najsitnijih detalja.

O pojmovima interpretacije i prevođenja bilo je reči u glavi 1.4. Kako je Pajton jezik koji se interpretira, odgovarajuća razvojna okruženja imaju u svom sastavu ugrađen interpreter. *Sistem za pomoć*[3] omogućava pregledanje dokumenata koji pojašnjavaju korišćenje, kako softverskih alata u okviru razvojnog okruženja, tako i samog jezika. On stavlja na uvid i dokumentaciju za korišćenje pridodatih biblioteka koje uključuju gotove softverske komponente. Često se integriše sa editorom, tako što omogućava programeru da vidi opis pojedinih naredbi jezika u toku njihovog unosa - *kontekstualna pomoć*.

Svi programi u ovoj knjizi realizovani su u razvojnom okruženju IDLE, koje se isporučuje uz besplatnu instalaciju Pajtona. Proces instalacije započinje posećivanjem sajta organizacije *Python Software Foundation*[4]. Potom se, u sekciji *Downloads*, izabira odgovarajuća *3.x* verzija za ciljni operativni sistem. Za Windows sisteme preporučuje se izvršna verzija instalacije sa ekstenzijom **.exe*. Po instalaciji Pajtona i po pokretanju IDLE-a, pojavljuje se ekran kao na sledećem prikazu:



Slika 2.1: IDLE: instalirana verzija može se razlikovati od autorove (3.8.5). Uočiti postojanje menija *Debug* za testiranje rada programa, kao i menija *Help* kojim se aktivira sistem za pomoć na engleskom jeziku.

Simbol >>> označava tekuću liniju za unos komande koja se, po pritisku na taster <Enter>, izvršava od strane interpretera, a njen rezultat ispisuje u sledećem redu. Interaktivni rad u okviru IDLE-a biće ilustrovan primerom u kome se okruženje tretira kao standardni kalkulator:

[2] Engl. *Graphical User Interface* - GUI.
[3] Engl. *Help system* - Help.
[4] `http://www.python.org`.

```
>>> 1 + 2
3
>>> 2 * 3.14 + 1
7.28
>>> 3 * abs(-3)
9
>>> 5 / 2
2.5
```

U gornjem primeru, `abs()` predstavlja matematičku funkciju koja računa apsolutnu vrednost broja. Kako bi se računanje složenijih matematičkih izraza pojednostavilo, u izraze se uvode *promenljive* koje se definišu uz pomoć znaka jednakosti (=):

```
>>> a = 1
>>> b = 2
>>> c = 2 * (a + b)
>>> c
6
>>> a
1
>>> x
Traceback (most recent call last):
  File "<pyshell#20>", line 1, in <module>
    x
NameError: name 'x' is not defined
```

Gornji primer pokazuje da se, u interaktivnom radu, *trenutne vrednosti* pojedinih promenljivih mogu prikazati navođenjem njihovog imena. Ako se navede promenljiva koja prethodno nije definisana, interpreter prijavljuje grešku. O promenljivim, izrazima i mehanizmu dodele vrednosti, biće više reči u narednom tekstu. Kratko izlaganje o interaktivnom okruženju[5] završava se napomenom da se *vezivanja* imena promenljivih za određene vrednosti, na primer `a = 1`, čuvaju u *imenskom prostoru* okruženja o kome će više reči biti u narednom tekstu. Pri izboru opcije *Restart Shell* iz menija *Shell*, ova vezivanja se *nepovratno* gube, što je ilustrovano sledećim primerom:

```
>>> a = 1
>>> a
1
>>>
=============================== RESTART: Shell =====================
>>> a
Traceback (most recent call last):
```

[5] Engl. *Shell*, što u prevodu znači školjka.

```
  File "<pyshell#24>", line 1, in <module>
    a
NameError: name 'a' is not defined
```

Rad u interaktivnom okruženju nije pogodan za izvršavanje složenijeg postupka jer se naredbe, koje ga realizuju, moraju *zapamtiti* i *ponoviti* uvek u istom redosledu. Zato se naredbe *grupišu* u program koji je zapisan u datoteci na disku. Program se može pokrenuti, kako iz IDLE-a tako i van njega, na za to predviđene načine.

2.2 Koliko je težak prvi Pajton program

Programeri veruju da se kroz praktične primere najbolje uči. Sledi jednostavan program u Pajtonu koji rešava sledeći problem:

Problem 2.1 — Težina tela. Izračunati težinu tela na površini Zemlje na osnovu poznate mase u kilogramima. ■

2.2.1 Od oblasti problema do algoritma

Pri rešavanju postavljenog zadatka neophodno je prvo poznavati *oblast problema*, odnosno izabrati podesan model koji ga opisuje. U ovom slučaju, model je opisan jednačinom za silu težine na površini Zemlje koja glasi $F = m \cdot 9.81$. Algoritam za rešavanje problema postaje:

Algoritam 2.1 — Težina tela.

1) uneti masu tela u kg i zapamtiti je u promenljivoj `m`
2) `F ← m * 9.81`
3) prikazati `F`

Pored poznavanja oblasti problema, rešavač treba da vlada veštinom sastavljanja odgovarajućeg algoritma, kao i da poznaje neki od programskih jezika, kako bi se algoritam pretočio u program. Prelazak iz oblasti problema u prostor efikasnih algoritama često predstavlja *najteži* deo zadatka. Program, koji realizuje prethodni algoritam u Pajtonu, dobija sledeći oblik:

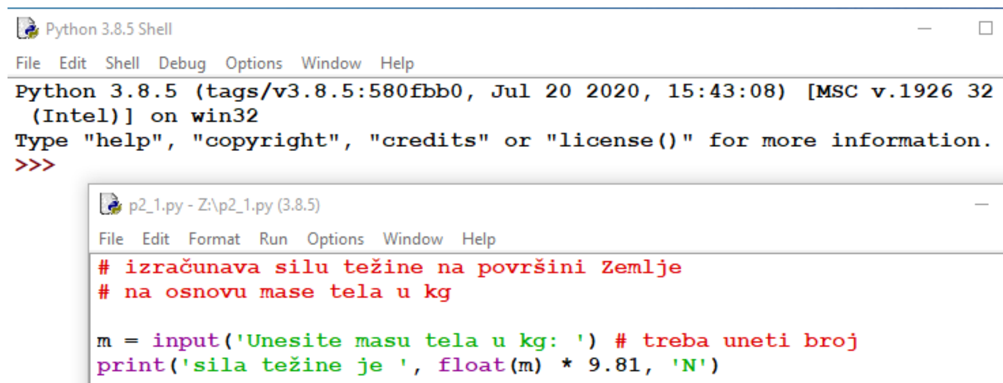
Program 2.1 — Težina tela.

```
1 # izračunava silu težine na površini Zemlje
2 # na osnovu mase tela u kg
3
```

```
4 m = input('Unesite masu tela u kg: ') # treba uneti broj
5 print('sila težine je ', float(m) * 9.81, 'N')
```

2.2.2 Testiranje programa u IDLE okruženju

Testiranje rada prethodnog programa podrazumeva da se, uz pomoć editora u IDLE-u, prvo formira programska datoteka. Editor se pokreće iz menija *File*, u okviru osnovnog prozora razvojnog okruženja, izborom opcije *New File* za novi, odnosno *Open...*, za otvaranje postojećeg programa (slika 2.2).



Slika 2.2: Unos programa u editoru. Pojedini istorodni elementi jezika obojeni su različitim bojama: crno - promenljive i operatori, žuto - naredbe, ljubičasto - ugrađene funkcije, zeleno - tekstualni podaci, crveno - komentari.

Po snimanju na disk, u datoteku sa ekstenzijom *.py* (u primeru *p2_1.py*), program se pokreće izborom opcije *Run Module* iz menija *Run*. Pokretanje se može obaviti i pritiskom na funkcijski taster <F5>. Sledi prikaz rada programa po pokretanju iz editora:

```
Unesite masu tela u kg: 88
sila težine je  863.2800000000001 N
```

2.2.3 Analiza programa

Analiza izvornog koda omogućava razumevanje upotrebljenih jezičkih elemenata. Prva dva reda u programu 2.1, koji počinju simbolom #, predstavljaju *komentare*. Pajton interpreter ne izvršava komentare (deo linije koji počinje sa #). U četvrtom redu komentarisan je deo programa koji se odnosi na unos mase tela. Komentari treba da *pojasne* određene delove programa za osobu koja će održavati program ili nastaviti rad na njemu. To često može biti i autor programa koji će, posle određenog vremenskog perioda, zaboraviti pojedinosti datog rešenja. Međutim, ne treba komentarisati *šta*

ili *kako* program radi, već *zašto* je primenjeno odgovarajuće rešenje. Jasno napisan program *sam po sebi* odgovara na pitanje šta, odnosno kako radi.

Objašnjenje četvrtog reda zahteva upoznavanje sa konceptima *promenljive*, *naredbe dodele vrednosti* i *funkcije*. Koncept promenljive uveden je u algoritam kako bi se omogućilo čuvanje ulaznih i izlaznih podataka za proces koji se modeluje. Slično, u postupku izračunavanja izlaznih promenljivih, često je potrebno sačuvati međurezultate koji se smeštaju u privremene promenljive. Različiti programski jezici *različito* realizuju koncept algoritamske promenljive u operativnoj memoriji. U Pajtonu, promenljiva predstavlja *ime* za podatak na koji se odnosi. Kao što će vrlo brzo postati jasno, ona *pokazuje* na podatak koji imenuje pa se njeno ime navodi uvek kada je potrebno manipulisati tim podatkom (zarad čitanja ili modifikacije). Promenljiva `m`, iz četvrtog reda, odnosi se na masu tela koju unosi korisnik.

Funkcija `input()` predstavlja *ugrađenu funkcionalnost* koja zahteva od korisnika da unese odgovarajuće ulazne podatke sa tastature. U opštem slučaju, funkcija predstavlja imenovani niz komandi jezika koje se, pod tim imenom, mogu pozvati kada je potrebno obaviti istu aktivnost. Može se uočiti sličnost između funkcije u programu i matematičke funkcije, mada ona nije potpuna, kao što će se videti u glavi 4. Kao i u matematici, funkcije obično obrađuju ulazne argumente i tako proizvode odgovarajući izlaz. Argumenti se navode u okviru zagrada koje neposredno slede ime funkcije i, ako ih ima više, razdvajaju se zapetom. U slučaju funkcije `input()`, ulazni argument je tekst koji se navodi između apostrofa, a koji će biti prikazan na ekranu kada interpreter bude izvršavao ovu funkciju. Prilikom izvršavanja funkcije, a po ispisu poruke, program staje sa radom sve dok korisnik ne unese odgovarajući niz znakova koji, u ovom slučaju, treba da bude brojčani podatak koji se odnosi na masu tela. Ako bi se funkcija `input()` pozvala bez argumenata (`m = input()`), interpreter ne bi ispisivao poruku, ali bi čekao korisnika da unese vrednost za masu tela. Ovde se uočava bitna razlika u odnosu na matematičke funkcije koje moraju imati bar jedan argument.

Kada korisnik unese težinu (završetak unosa podrazumeva pritisak na taster <Enter>), izvršava se naredba dodele vrednosti predstavljena simbolom = . Naredba dodele vrednosti izračunava *izraz* sa desne strane jednakosti i njegovu vrednost pridružuje promenljivoj sa leve strane. Izrazi u Pajtonu, kao i u matematici, prave se upotrebom promenljivih i konstanti, uz odgovarajuće operatore i funkcije. Neka `c * abs(teta) + 2.11` predstavlja primer izraza. Izraz čine dve promenljive, `c` i `teta`, funkcija `abs()` koja vraća apsolutnu vrednost broja, realna konstanta 2.11 i operatori množenja i sabiranja. U četvrtom redu programa, izraz sa desne strane jednakosti čini funkcija `input()`, a njegova vrednost postaje odgovarajući korisnički unos.

Poslednji red programa sadrži funkciju za ispis - `print()`. Ona prikazuje svoje argumente na ekranu, po redosledu navođenja, uz *po jedan* blanko znak između argumenata u listi. Prvi i treći argument funkcije `print()` su tekstualne poruke navedene unutar apostrofa, dok drugi argument predstavlja izraz koji se, pre prikazivanja na ekranu, izračunava u realan broj. Izraz `float(m) * 9.81`, sastoji se od funkcije

`float()` koja pretvara vrednost na koju ukazuje promenljiva `m` u realan broj, te operatora množenja i realne konstante 9.81 (ubrzanje zemljine teže). Obratiti pažnju da funkcija `input()` *uvek* vraća podatak unet sa tastature u obliku niza znakova (tekst), pa je tekstualnu reprezentaciju broja potrebno *prevesti* u njegovu realnu vrednost. O tipovima podataka i njihovim vrednostima biće reči u glavi 2.3.

2.2.4 Sintaksne i semantičke greške

Šta bi se desilo ako bi korisnik, umesto realnog broja, uneo neodgovarajući podatak, na primer tekst `'velika masa'`? Program se tom prilikom ponaša na sledeći način:

```
Unesite masu tela u kg: velika masa
Traceback (most recent call last):
  File "C:\programi\p2_1.py", line 5, in <module>
    print('sila težine je ', float(m) * 9.81, 'N')
ValueError: could not convert string to float: 'velika masa'
```

Interpreter prijavljuje grešku tipa `ValueError` (nemogćnost konvertovanja teksta `'velika masa'` u realan broj). U realnom radu sa računarom *neizbežno* je pravljenje grešaka, sem u najtrivijalnijim situacijama. Greške koje se javljaju prilikom izvršavanja programa mogu se svrstati u dve kategorije: *sintaksne greške* i *semantičke greške*.

Pod *sintaksom* jezika podrazumeva se *skup pravila* za kombinovanje osnovnih jezičkih elemenata u ispravne, složenije celine. Na primer, u srpskom jeziku, moguće pravilo za sastavljanje rečenice nalaže sledeći redosled tipova reči: subjekat, predikat, pridev, objekat – `'Jovan piše interesantnu knjigu'`. Sintaksno neispravna rečenica `'piše knjigu Jovan interesantnu'` može se razumeti, ali svakako nije u duhu našeg jezika. U Pajtonu, kao i u drugim jezicima, prisustvo sintaksne greške *lako* se otkriva. Prilikom pokretanja programa interpreter obavezno sprovodi proveru sintaksne ispravnosti pa, ako postoji problem ove vrste, program se ni ne pokreće. Tom prilikom označava se mesto greške u izvornom kodu čime se sugeriše programeru šta treba da ispravi. Slede dva primera sintaksnih grešaka prilikom rada u interaktivnom okruženju:

```
>>> 1a = 100  # ime promenljive ne sme početi brojem
SyntaxError: invalid syntax
>>> a = 3 * 2 +  # nepravilno formiran izraz
SyntaxError: invalid syntax
```

Semantika jezika bavi se *značenjem* sintaksno ispravnih jezičkih konstrukcija. Rečenica `'kamen peva na banderi'` je sintaksno ispravna, ali nema nikakvog smisla. Neke semantičke greške interpreter otkriva u toku izvršavanja programa. Tada se izvršavanje prekida i prijavljuje se greška, kao prilikom unosa niza znakova koji ne predstavljaju validan broj (masu tela). Čest primer semantičke greške je *deljenje nulom*

(`a=b/c`), kada promenljiva delilac (`c`) ukazuje na nulu. Program se od ovakvih grešaka može oporaviti i nastaviti sa radom, ali će o tome biti reči u glavi 8.

Najopasnija situacija u programiranju nastaje kada program nema ni sintaksnih, ni semantičkih grešaka koje se otkrivaju pri izvršavanju, ali ne radi ono što se od njega očekuje! Takve greške potiču od *pogrešnog* algoritma i najteže se otkrivaju.

2.3 Predstavljanje podataka i osnovne operacije

U uvodnom izlaganju napomenuto je da apstrakcija predstavlja pojednostavljenje sistema na nivo modela koji zadržava dovoljno detalja da bi poslužio svojoj nameni. Program, kao model realnog sistema ili procesa, obrađuje podatke kojima se, opet, modeluju informacije koje teku između podsistema ili podprocesa. U Pajtonu se svi podaci apstrahuju *objektima*. Ova glava bavi se objektima osnovnih tipova i elementarnim operacijama u kojima oni učestvuju.

2.3.1 Objekti

Svi celi brojevi kao 0, 1 ili -15, odnosno realni brojevi poput -1.25 ili 3.75, predstavljeni su u programu jedinstvenim objektima. Isto važi i za tekstualne podatke. Na primer, niz znakova `'Programiranje je umetnost!'` tretira se kao jedan objekat. Pored podataka, Pajton tretira kao objekte i funkcije poput `input()` ili `print()` pa i sam program kao jedinstvenu celinu.

Svaki objekat jednoznačno je definisan svojim tipom, identitetom i vrednošću. *Tipom* se određuje skup *svih vrednosti* koje objekti iz tipa mogu imati, kao i skup *svih operacija* u kojima ti objekti mogu učestvovati. U terminologiji objektno orijentisanog programiranja, tip objekta zove se još i *klasom*. Klasa predstavlja sve objekte iste vrste, na primer sve realne brojeve. *Identitet* objekta odnosi se na njegovu memorijsku adresu (lokaciju u memoriji) i ne može se promeniti pošto se objekat prvi put kreira. Tip (klasa) svakog objekta može se odrediti funkcijom `type()`, a identitet funkcijom `id()`, što je ilustrovano sledećim primerom:

```
>>> type(153) # tip (klasa) objekta koji predstavlja broj 153
<class 'int'>
>>> id(153)  # identitet (mem. adresa) objekta sa vrednošću 153
1476919856
>>> type(-10.23401) # tip objekta -10.23401
<class 'float'>
>>> id(-10.23401)   # identitet objekta -10.23401
2440802051560
>>> type('Pajton je zakon!') # tip objekta 'Pajton je zakon!'
<class 'str'>
>>> id('Pajton je zakon!') # identitet objekta 'Pajton je zakon!'
```

```
2440845694576
```

Osnovni predefinisani tipovi

Pajton podržava veliki broj *predefinisanih tipova*. U knjizi se detaljno razmatraju tipovi prikazani u tabeli 2.1. Tip `int` odnosi se na cele brojeve koji se, uz zanemarivanje detalja, predstavljaju u obliku $\pm\sum_{i=0}^{n} a_i \cdot 2^i, a_i \in \{0,1\}$. Realni brojevi predstavljeni su u obliku *zapisa sa pokretnom tačkom*, u dvostrukoj tačnosti – tip `float`.[6] Broj sa pokretnom tačkom može se, uz zanemarivanje detalja, zapisati preko tri cela broja kao $(-1)^s \cdot 1.m \cdot 2^e$, gde $s \in \{0,1\}$ određuje znak, $m \in N_0$ predstavlja mantisu, a $e \in Z$ eksponent broja. Mantisa i eksponent zapisani su u specifičnom binarnom formatu.

tip (ili klasa)	**oznaka**	**domen vrednosti**	**primeri vrednosti**
celobrojni	`int`	celi brojevi	`0, 1, -123, 45_101`
realni	`float`	realni brojevi	`1.25,.02,-146.75`
kompleksni	`complex`	kompleksni brojevi	`-3j, -0.5 - 4.25j`
logički	`bool`	{ False, True }	`False, True`
tekstualni	`str`	niz znakova u Unicode-u	`'Pajton je lagan',''`
torka	`tuple`	nepromenljiv niz proizvoljnih objekata.	`(), (1.25,True,'abc')`
lista	`list`	promenljiv niz proizvoljnih objekata.	`[], [1,2],['A',1,1+2j]`
skup	`set`	neuređen skup proizvoljnih nepromenljivih objekata. Duplikati nisu dozvoljeni.	`{}, {1,2},{'A',{1,2}}`
rečnik	`dict`	parovi ključ-vrednost. Vrednosti proizvoljni objekti, ključevi proizvoljni nepromenljivi objekti.	`{11:'Beograd', 21:'Novi Sad' 22:'Ruma'}`
nepostojeće	`NoneType`	{ None }	`None`

Tabela 2.1: Osnovni tipovi objekata u Pajtonu. Prva četiri reda odnose se na brojne tipove (logički tip interno je realizovan kao broj). Sledećih pet redova predstavljaju objektne kolekcije (objekti koji sadrže druge objekte). Tip "nepostojeće" odnosi se na promenljivu koja, u posmatranom trenutku, ukazuje na objekat sa specijalnom vrednošću `None`. Zapaziti da se simbol `_` može koristiti u brojnom zapisu da poveća čitljivost za velike brojeve (`45_101` je 45101).

[6] Engl. *Double precission floating point*.

Kako je memorija računara *konačna* (sadrži ukupno n bitova), u njoj se može sačuvati najviše 2^n različitih informacija. Čak i kada bi se memorija u celosti koristila za memorisanje brojeva, bilo bi nemoguće preslikati beskonačne skupove celih ili realnih brojeva u konačan skup od 2^n elemenata. Od verzije 3.x, Pajton *ne nameće* ograničenje za maksimalni ceo broj, ali je on određen maksimalnom raspoloživom memorijom. Drugi programski jezici, poput Jave ili C-a, precizno definišu ekstremne vrednosti koje su dovoljno dobre za najveći broj inženjerskih primena. Sa druge strane, tip `float` samo *dobro aproksimira* skup realnih brojeva. Brojevima koji se ne mogu predstaviti kao sume stepena dvojke, poput 0.1, odgovaraju objekti čije su vrednosti približne pa se u programima mogu pojaviti izvesni neželjeni efekti o kojima će biti reči u delu koji se bavi iterativnim algoritmima.

Kompleksni brojevi (tip `complex`) predstavljeni su uređenim parovima sa realnim i imaginarnim delom kao brojevima tipa `float`. Logički tip `bool` odnosi se na *istinitost* pojedinih logičkih izraza poput $5 > 2$ (istina - `True`), ili $5 < 2$ (laž - `False`). Iako posebno izdvojen, ovaj tip predstavljen je interno kao ceo broj (0 za `False`, 1 za `True`) pa se logičke vrednosti mogu koristiti u brojnim izrazima. Na primer, izraz `2 * True + False + 1`, ima brojnu vrednost 3.

Sledeći tip, koji se po svom značaju svrstava uz brojne, je tekstualni[7] – tip `str`. Tekstualni objekti reprezentuju proizvoljne nizove znakova koji mogu da sadrže simbole iz bilo kog svetskog pisma. Vrednost tekstualnog objekta navodi se pod apostrofima ili navodnicima:

```
>>> a = 'Ja sam tekst.'
>>> b = "I ja, i ja!"
>>> print(a, b)
Ja sam tekst. I ja, i ja!
```

Tekstualni objekti mogu imati proizvoljnu dužinu, ograničenu raspoloživom memorijom, a funkcija `len()` daje odgovor koliko ona iznosi. Kada postoji potreba da se u tekstu koriste apostrofi (navodnici), onda se za omeđavanje teksta koriste navodnici (apostrofi):

```
>>> txt = 'Gledao sam juče "Slagalicu" na TV-u'
>>> print('Tekst:', txt, ', ima', len(txt), 'znakova.')
Tekst: Gledao sam juče "Slagalicu" na TV-u, ima 35 znakova.
```

Tipovi (klase) `str`, `tuple`, `list`, `set` i `dict`, predstavljaju *kolekcije* objekata - to su objekti koji *sadrže* druge objekte. Kako je bez kolekcija teško zamisliti iole ozbiljniji program, one će detaljno biti obrađene u glavi 5. Do tada će se, pored brojnih tipova, obilato koristiti samo tekstualni objekti tipa `str`.

---

[7] Engl. *String*.

2.3.2 Izrazi i promenljive, dodela vrednosti

Objekti se, upotrebom različitih operacija, mogu kombinovati u *izraze*. Rezultat izračunavanja izraza takođe predstavlja objekat određenog tipa. Na primer, obim kruga poluprečnika `4.0`, iznosi približno `2 * 4. * 3.14`. Brojni objekti sa vrednostima `4.` i `3.14` su realnog, a `2`, celobrojnog tipa. Operacija množenja označena je operatorom `*`, primenjenim prvo nad operandima `2` i `4.`, a potom nad rezultatom prvog množenja i operandom `3.14`. Po izračunavanju, izraz ima realnu vrednost `25.12` koja se memoriše u novokreiranom objektu sa tom vrednošću. Treba primetiti da se celobrojni objekat sa vrednošću `2` *implicitno konvertuje* u stariji realni tip pre nego što je pomnožen realnim objektom sa vrednošću `3.14`. Uopšte uzevši, celobrojne vrednosti se implicitno konvertuju u realne kada u izrazu ima i jednih i drugih. Slično važi za realne i kompleksne vrednosti, uz napomenu da su kompleksne vrednosti starije. Sledeći primer potvrđuje da izrazi, kao izračunate vrednosti, predstavljaju objekte sa tipom i identitetom:

```
>>> 2 * (3 + 5)  # izraz označava objekat sa vrednošću 16
16
>>> type(2 * (3 + 5)) # celobrojni tip
<class 'int'>
>>> id(2 * (3 + 5))  # identitet (memorijska adresa) objekta
1911877904
```

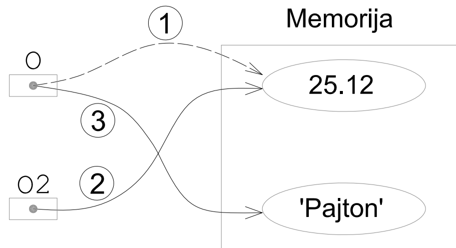
U prethodnom primeru, kao i u matematici, uobičajeni *prioritet* izračunavanja može se promeniti korišćenjem zagrada. Izraz sam po sebi nije dovoljan za kompleksnija izračunavanja, sem ukoliko se ne posmatra u širem kontekstu kada se dodeljuje promenljivoj putem naredbe *dodele vrednosti*.

Imenovanje objekata - promenljive kao objektne reference

Vrednost izraza koji računa obim mogla bi se dodeliti promenljivoj `O`: `O=2*4.*3.14`. Sa obimom se dalje manipuliše tako što se objektu koji čuva njegovu vrednost pristupa preko promenljive `O`. Pri izvršavanju naredbe dodele vrednosti, interpreter izračunava izraz sa desne strane jednakosti i kreira objekat realnog tipa sa vrednošću `25.12`. Potom se ovaj objekat *imenuje* sa `O`. Promenljiva `O` *ne sadrži* objekat, već *adresu* memorijske lokacije na kojoj se objekat nalazi - istu onu adresu koja predstavlja identitet objekta. U tom smislu promenljiva predstavlja *objektnu referencu* putem koje se pristupa objektu. Postupak dodele vrednosti ilustrovan je na slici 2.3, uz pomoć sledeće sekvence:

```
>>> O = 2 * 4.0 * 3.14   # dodela 1
>>> O2 = O   # dodela 2
>>> print(O, O2, id(O), id(O2))
25.12 25.12 1887660059240 1887660059240
>>> O = 'Pajton'   # dodela 3
>>> print(O, O2, id(O), id(O2))
```

```
Pajton 25.12 1887703505640 1887660059240
```



Slika 2.3: Promenljive kao imena za objekte. Objektna referenca označena je strelicom, a nevažeća isprekidanom linijom. Numeracija se odnosi na pojedine dodele iz primera.

U prvom redu primera, izrazom na desnoj strani jednakosti, definisan je objekat realnog tipa čije ime postaje `O`, a vrednost 25.12. Promenljiva sadrži memorijsku adresu objekta - objektnu referencu, što je predstavljeno strelicom sa brojem jedan na slici 2.3. Naredba dodele vrednosti iz drugog reda definiše promenljivu `O2` koja označava isti objekat kao i promenljiva `O`. Dakle, jedan objekat može imati *više* imena u toku rada programa. Treba naglasiti da se ovde ne kreira novi objekat sa vrednošću 25.12, već se kopira memorijska adresa pa obe promenljive, po drugoj dodeli, ukazuju na isti objekat. U to se može uveriti kada se, uz pomoć funkcije `id()`, prikažu identiteti objekata imenovanih sa `O` i `O2` - memorijske adrese su iste, radi se o istom objektu. U nekim jezicima, poput Jave ili C-a, brojni tipovi podataka nisu objekti. Tu odgovarajuće promenljive "sadrže" brojne podatke, a dodela vrednosti *kopira sadržaj* iz jedne lokacije u drugu.

Treća naredba dodele vrednosti raskida vezu između promenljive `O` i realnog objekta. Sada se uspostavlja novo vezivanje promenljive `O` za objekat tekstualnog tipa čija je vrednost niz znakova `'Pajton'`. Ovde se jasno uočava *dinamička* priroda promenljivih u Pajtonu prema tipu objekta na koje ukazuju. U već pomenutoj Javi ili C-u, promenljive se moraju deklarisati po tipu pre upotrebe, a tipovi se ne mogu proizvoljno menjati u toku rada programa. Mogućnost promene tipa objekta na koji promenljiva ukazuje, predstavlja u isto vreme kako slabost, tako i prednost koncepta koji zastupaju dinamički jezici poput Pajtona. Izvorni kod je kraći jer nema naredbi kojima se deklariše tip, ali se zato zahteva pažljivo referenciranje kako ne bi došlo do grešaka u toku rada programa. U slučaju velikih programskih sistema praćenje prirode pojedinih promenljivih je otežano pa je program teže održavati. Sloboda u programiranju, kao i u životu, traži odgovornijeg pojedinca!

Kako promenljive sadrže memorijske adrese objekata (objektne reference), u daljem tekstu, umesto uobičajene formulacije da "promenljiva sadrži", navodiće se da "promenljiva ukazuje na" odgovarajući podatak – objekat određenog tipa.

Imena promenljivih u Pajtonu mogu biti sastavljena od velikih i malih slova engleskog alfabeta, cifara i simbola _ . Imena ne mogu početi cifrom i osetljiva su na velika i mala slova, pa su tako obim i OBIM dve različite promenljive. Savetuje se imenovanje koje odgovara značenju podatka (objekta) na koji promenljiva ukazuje. Preporučuje se da imena počinju *malim* slovima, kao i da ukoliko ime sadrži više reči, da se one razdvajaju simbolom _ (na primer, broj_indeksa). U tabeli 2.2 date su dve sekvence koje računaju površinu i obim kruga, ali je desna manje razumljiva.

Operator **, u tabeli 2.2, označava operaciju stepenovanja. U narednom tekstu biće reči o operacijama koje se mogu koristiti nad različitim brojnim tipovima. U Pajtonu se često koristi naredba *višestruke* dodele vrednosti:

```
>>> a, b, poruka = 3, 4, 'jednostavno zar ne!'
>>> print(a, b, poruka)
3 4 jednostavno zar ne!
```

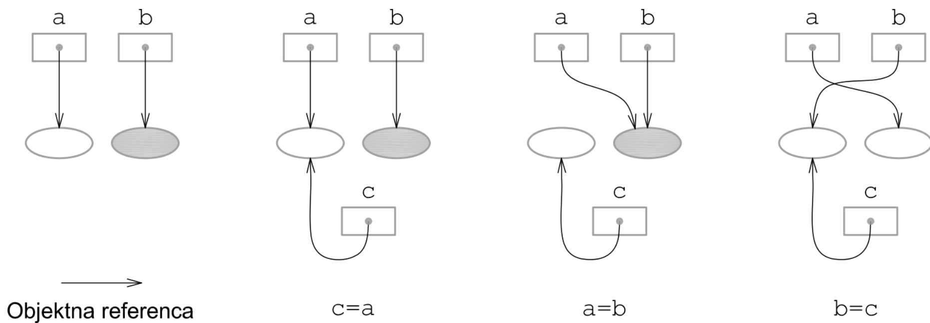
Na levoj strani jednakosti nalazi se lista promenljivih razdvojenih zapetom, a na desnoj, lista izraza čije se izračunate vrednosti pridružuju promenljivim. Dinamika dodele podrazumeva pridruživanje imena izračunatim vrednostima sleva na desno. U primeru se prvo imenuje objekat sa vrednošću 3 (a), pa objekat sa vrednošću 4 (b) i konačno tekstualni objekat (poruka).

razumljivo	**nerazumljivo**
pi = 3.14	a = 3.14
O = 2 * r * pi	b = 2 * x * a
P = r**2 * pi	c = x**2 * a

Tabela 2.2: Promenljive treba da nose imena koja asociraju na značenje podataka koje reprezentuju. Uobičajeno je da imena počinju malim slovima, sem kada označavaju pojmove poput obima iz gornjeg primera.

Problem 2.2 — Razmena vrednosti. Učitati dve vrednosti sa tastature i imenovati ih uz pomoć promenljivih a i b. Razmeniti vrednosti na koje promenljive ukazuju i potom ih prikazati na ekranu. ■

Budući da promenljive predstavljaju objektne reference, razmena vrednosti (objekata) na koje one ukazuju može se obaviti uz pomoć treće promenljive, kao što je prikazano na slici 2.4. Obratiti pažnju da, na kraju procesa, prvi objekat ima dva imena (promenljive koje ukazuju na njega), b i c. Sledi realizacija postupka sa slike 2.4 i prikaz rezultata rada programa.



Slika 2.4: Razmena vrednosti. Prilikom `c = a`, kopiraju se reference (adrese), a ne objekti!

```
1  # razmenjuje učitane vrednosti sa tastature
2  a = input('Prva vrednost = ')
3  b = input('Druga vrednost = ')
4  print('pre razmene a =', a, 'b =', b)
5  # razmena
6  c = a   # 1
7  a = b   # 2
8  b = c   # 3
9
10 print('posle razmene a =', a, 'b =', b)
```

```
Prva vrednost = 10
Druga vrednost = 20
pre razmene a = 10 b = 20
posle razmene a = 20 b = 10
```

Razmena vrednosti može se realizovati preko naredbe višestruke dodele vrednosti što predstavlja optimalno rešenje *u stilu* jezika Pajton.

Program 2.2 — Razmena vrednosti.

```
1  # razmenjuje učitane vrednosti sa tastature
2  a = input('Prva vrednost = ')
3  b = input('Druga vrednost = ')
4
5  print('pre razmene a =', a, 'b =', b)
6  a, b = b, a   # razmena
```

```
7 print('posle razmene a =', a, 'b =', b)
```

Prilikom višestruke dodele, prvo se izračunavaju izrazi sa desne strane (r6).[8] Prvi izraz sa desne strane sastoji se samo od promenljive `b` pa se rezultat njegovog izračunavanja odnosi na objekat sa vrednošću 20. Drugi izraz na desnoj strani, po izračunavanju, ukazuje na objekat sa vrednošću 10. Potom se vrši imenovanje za objekat sa vrednošću 20 sa `a`, odnosno za objekat sa vrednošću 10 sa `b`, počevši od prve promenljive u listi sa leve strane dodele vrednosti. Na ovaj način, dve promenljive su razmenile celobrojne objekte na koje ukazuju.

2.3.3 Aritmetičko-logičke operacije i poređenja

Objekti, kao apstrakcije podataka, mogu imati različite domene dozvoljenih vrednosti (Tabela 2.1). Pored domena vrednosti, tip definiše i operacije koje se mogu obavljati nad objektom. Najčešće operacije, koje se mogu obavljati nad celobrojnim (`int`), realnim (`float`) i logičkim (`bool`) tipovima, prikazane su u tabeli 2.3.

operacija	**opis**
`int` i `float`	
`x + y`	Sabiranje. Ako je bar jedan broj tipa `float`, rezultat je `float`
`x - y`	Oduzimanje. Ako je bar jedan broj tipa `float`, rezultat je `float`
`x * y`	Množenje. Ako je bar jedan broj tipa `float`, rezultat je `float`
`x / y`	Deljenje. Rezultat je uvek `float`
`x // y`	Celobrojno deljenje (7 // 4 je 1)
`x % y`	Ostatak pri celobrojnom deljenju (7 % 4 je 3)
`x ** y`	Stepenovanje x^y. Ako je bar jedan broj tipa `float`, rezultat je `float`
`bool`	
`x and y`	Logičko i. `True` akko oba `True`. Za `x` je `False`, ne računa vrednost `y`
`x or y`	Logičko ili. `False` akko oba `False`. Za `x` je `True`, ne računa vrednost `y`
`not x`	Logičko ne. `True` akko `x` je `False`

Tabela 2.3: Najčešće operacije za brojne i logičke tipove. Logičke vrednosti u aritmetičkim izrazima imaju celobrojne vrednosti, `False` (0) i `True` (1). *Akko* označava “ako i samo ako”.

Problem 2.3 — Tabela istinitosti. *Tabela istinitosti* logičke operacije `op` prikazuje vrednosti izraza `x op y` za sve kombinacije logičkih promenljvih `x` i `y`. Prikazati tabele istinitosti za operacije `and` i `or`. ■

[8] Nadalje će, sa (rX), biti označen red X programa koji se razmatra.

Tabela istinitosti logičkih operacija and i or sadrži četiri moguće kombinacije vrednosti logičkih promenljivih (četiri vrste):

Program 2.3 — Tabela istinitosti.

```
# ispisuje tabelu istinitosti za AND i OR

print('  x   |   y   | x and y | x or y')
print('True  | True  |', True and True, '   |', True or True)
print('True  | False |', True and False, '  |', True or False)
print('False | True  |', False and True, '  |', False or True)
print('False | False |', False and False, '  |', False or False)
```

Zaglavlje tabele ispisano je u r3. Potom sledi ispis za sve četiri kombinacije logičkih promenljivih, pri čemu je rezultat obe operacije prikazan u istoj tabeli (r4-7). Prilikom formiranja poravnanja u tabeli, voditi računa da su vrednosti izraza, odvojenih zapetom u funkciji print(), ispisane sa jednim znakom proreda. Logičko "i" ima vrednost tačno (True – istina) samo kada su oba operanda istinita, a "ili" kada je bar jedan operand istinit. Sledi prikaz rada programa:

```
  x   |   y   | x and y | x or y
True  | True  | True    | True
True  | False | False   | True
False | True  | False   | True
False | False | False   | False
```

Pored operacija iz tabele 2.3, bitan element svakog jezika čine operacije poređenja koje omogućavaju utvrđivanje različitih odnosa između podataka. Poređenja čine osnovu za donošenje odluka i formiranje alternativnih putanja u algoritmu (glava 3). Najčešće korišćeni operatori poređenja prikazani su u tabeli 2.4.

U dosadašnjem izlaganju, za utvrđivanje identičnosti dva objekta, korišćena je funkcija id(). Međutim, za tu svrhu pogodnije je koristiti logički operator is, kao što se vidi iz sledećeg primera:

```
>>> a = 1
>>> b = a
>>> id(a) == id(b)
True
>>> a is b    # da li a i b ukazuju na isti objekat
True
```

poređenja	opis
`int` i `float`	
`x == y`	Jednako. `True`, ako su `x` i `y` jednaki
`x != y`	Različito. `True`, ako su `x` i `y` različiti
`x < y`	Manje. `True`, ako je `x` manje od `y`
`x > y`	Veće. `True`, ako je `x` veće od `y`
`x <= y`	Manje jednako. `True`, ako je `x` manje ili jednako `y`
`x >= y`	Veće jednako. `True`, ako je `x` veće ili jednako `y`
objekti bilo kog tipa	
`x is y`	Provera identiteta. `True`, ako `x` i `y` ukazuju na isti objekat

Tabela 2.4: Operatori poređenja. Izrazi u prvoj koloni izračunavaju se na `True` ili `False`.

Problem 2.4 — Maksimalna vrednost. Napisati program koji ispisuje veći od dva uneta realna broja. ■

Zbog ograničenog izbora do sada izloženih jezičkih konstrukcija, rešenje će biti formulisano u obliku odgovarajućeg izraza koji će, po izračunavanju, imati vrednost većeg broja. Pri tome treba imati na umu da operatori poređenja vraćaju logičku vrednost koja se u aritmetičkim izrazima tretira kao nula ili jedan.

Program 2.4 — Maksimalna vrednost.

```
1 # ispisuje veći od dva uneta broja
2
3 x = float(input('x= '))
4 y = float(input('y= '))
5
6 print('veći broj je', x * (x > y) + y * (x <= y) )
```

Pri unosu brojeva u r3 i r4, argument funkcije `float()` je vrednost koju vraća funkcija `input()`. Interpreter prvo poziva `input()`, pa se tekstualna reprezentacija unetog broja prosleđuje funkciji `float()` koja unos prevodi u realan broj. Ovde postoji analogija sa složenom funkcijom u matematici – na primer, $y = sin(cos(x))$. Obratiti pažnju da prilikom unosa postoji mogućnost za pojavu semantičke greške, ako tekstualni unos ne predstavlja regularni zapis broja.

Ako je `x` veće od `y`, prvi sabirak izraza u r6 postaje `x` - svodi se na `x * True`, odnosno na `x * 1`. Slično, drugi sabirak postaje nula. Ako je `x` manje od `y`, onda

prvi sabirak postaje nula (`x * False`), a drugi `y` (`y * True`). Specijalni slučaj nastaje kada su uneti brojevi jednaki. Tada prvi sabirak postaje nula, a drugi `y`. U slučaju jednakosti brojeva tekstualna poruka nije adekvatna, ali je ispisana vrednost ispravna. Predloženo rešenje predstavlja dosetku koja, zbog smanjene čitljivosti izvornog koda, ne predstavlja dobru programersku praksu. U glavi 3 biće proširen arsenal jezičkih alata koji će omogućiti da se ovakav i slični problemi reše na zadovoljavajući način. Sledi prikaz rada programa:

```
x= 10.25
y= 25.11
veći broj je 25.11
```

2.3.4 Konstruktori osnovnih tipova

Objekti se najčešće kreiraju *implicitno*, prilikom izračunavanja izraza na desnoj strani dodele vrednosti, ili pri korisničkom unosu.[9] Međutim, oni se mogu kreirati i *eksplicitno*, pozivom specijalnih funkcija definisanih unutar svakog tipa, odnosno klase objekata. Ove funkcije zovu se *konstruktori* i o njima će, kao i o korisnički definisanim klasama, više reči biti u glavi 10. Pored odvajanja odgovarajućeg memorijskog prostora za smeštaj objekta, konstruktori obično inicijalizuju objekat na početnu vrednost. Ona se, u toku izvršavanja programa, kod jednih tipova može, a kod drugih ne može menjati.

U prvim problemima ove glave upotrebljeni su konstruktori za celobrojne i realne objekte - `int()` i `float()`. Pored njih, ovde će biti reči i o konstruktoru `str()` kojim se eksplicitno prave tekstualni objekti. Navedeni konstruktori najčešće se koriste kada se podatak osnovnog tipa želi predstaviti objektom drugog tipa - *eksplicitna konverzija*:

```
>>> a = int(5)   # nepotrebno, dovoljno  a = 5
>>> a
5
>>> int(55.9)  # konverzija realnog u celobrojni tip
55
>>> int('25')  # konverzija tekstualnog u celobrojni tip
25
>>> a = float(7.11)  # nepotrebno, dovoljno a = 7.11
>>> a
7.11
>>> float('56.234')  # konverzija tekstualnog u realni tip
56.234
>>> a = str('tekst')  # nepotrebno, dovoljno a = 'tekst'
>>> a
'tekst'
>>> str(56.11)  # konverzija realnog u tekstualni tip
```

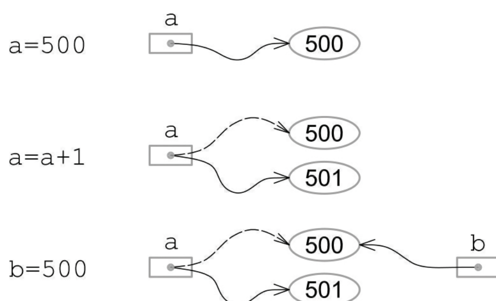
[9] Postoje i druge situacije o kojima je sada rano diskutovati.

```
'56.11'
```

Prilikom konverzije realnog u celobrojni tip sprovodi se *odsecanje* decimalnih cifara. Treba voditi računa da konverzija iz teksta u brojne tipove može rezultovati greškom. Ovo je bitno prilikom korišćenja funkcije `input()`, ako se očekuje unos brojnog podatka.

2.3.5 Promenljivost objekata

Promenljivost objekta odnosi se na *mogućnost promene vrednosti* koju objekat reprezentuje. U tom smislu, objekti mogu biti *promenljivi* ili *nepromenljivi*.[10] Svi brojni tipovi iz tabele 2.1, kao i tekstualni objekti tipa `str`, su nepromenljivi. Nepromenljivim objektima se, pošto su kreirani u memoriji, ne može promeniti vrednost (slika 2.5).



Slika 2.5: Nepromenljivost objekata. Nevažeće reference označene su isprekidanim strelicama.

Pri izvršavanju naredbe dodele vrednosti `a = 500`, u memoriji se, ukoliko već nije postojao, kreira celobrojni objekat sa vrednošću 500. Promenljiva `a` čuva memorijsku adresu objekta (objektna referenca). Prilikom izvršavanja `a = a + 1`, prvo se izračunava vrednost izraza sa desne strane jednakosti. Pristupa se objektu na koji ukazuje promenljiva `a` (500), pa se vrši sabiranje sa jedinicom. Pošto su celobrojni objekti nepromenljivi, potrebno je kreirati *novu* instancu sa vrednošću 501. Potom se `a` menja tako da ukazuje na novokreirani objekat. U većini Pajton implementacija stari objekat sa vrednošću 500 ostaje u memoriji samo ako na njega ukazuje *bar još jedna* promenljiva iz programa. Ako to nije slučaj, objekat se briše u trenutku kada interpreter aktivira proces *čišćenja nereferenciranih objekata* iz memorije, čime se povećava slobodan prostor.[11] U primeru sa slike 2.5, naredba `b = 500` ponovo imenuje stari objekat i na taj način mu *produžava život*.

Nepromenljivi objekti onemogućavaju nenamerne promene stanja programa u situacijama kada veći broj programskih celina pristupa istim podacima. O promenljivim objektima biće reči u glavi 5, kada se budu razmatrale kolekcije.

[10] Engl. *Mutable* i *Immutable*.
[11] Engl. *Garbage collection*.

ŠTA JE NAUČENO

- Svi podaci u Pajtonu posmatraju se kao objekti koji imaju svoj tip, identitet i vrednost.
- Tip objekta određuje skup mogućih vrednosti koje objekat može da ima, kao i koje se sve operacije mogu nad njim obavljati.
- Najčešći tipovi u programu su brojni i tekstualni.
- Identitet objekta predstavlja nepromenljivu adresu memorijske lokacije gde počinje smeštanje objekta u memoriji.
- U memoriji se mogu tačno zapisati samo oni brojevi koji predstavljaju sume stepena dvojke. Ostali brojevi zapisani su približno - zaokruživanje.
- Objekti se kombinuju u izraze korišćenjem različitih operacija. Vrednost izračunatog izraza memoriše se u objektu odgovarajućeg tipa.
- Promenljiva skladišti adresu objekta u memoriji - objektnu referencu. Može se posmatrati i kao ime za objekat na koji trenutno ukazuje. Preko nje se pristupa objektu u programu.
- Objekat može imati više imena - promenljivih koje ukazuju na njega
- Ime promenljive treba da odgovara suštini objekta na koji ona ukazuje.
- Naredba dodele vrednosti imenuje objekat koji nastaje kada se izračuna izraz sa desne strane jednakosti, imenom promenljive sa leve strane.
- Objekti nekih tipova, poput brojnih ili tekstualnih, ne mogu promeniti vrednost po kreiranju.
- Prilikom kopiranja promenljivih, kopiraju se objektne reference, a ne sami objekti.
- Aritmetičke operacije obavljaju se nad brojevima, a logičke nad istinitosnim vrednostima `True` i `False`. Rezultat poređenja objekata je istinitosna vrednost.
- Sintaksne greške nastaju kada se odstupi od pravila za formiranje validnih konstrukcija jezika. Pajton interpreter otkriva sve sintaksne greške neposredno pre izvršavanja prve linije koda.
- Semantičke greške predstavljaju logičke propuste koji najčešće dovode do prekida rada programa. Najopasnije su one semantičke greške koje interpreter ne može da otkrije, a odnose se na nekorektne algoritme.

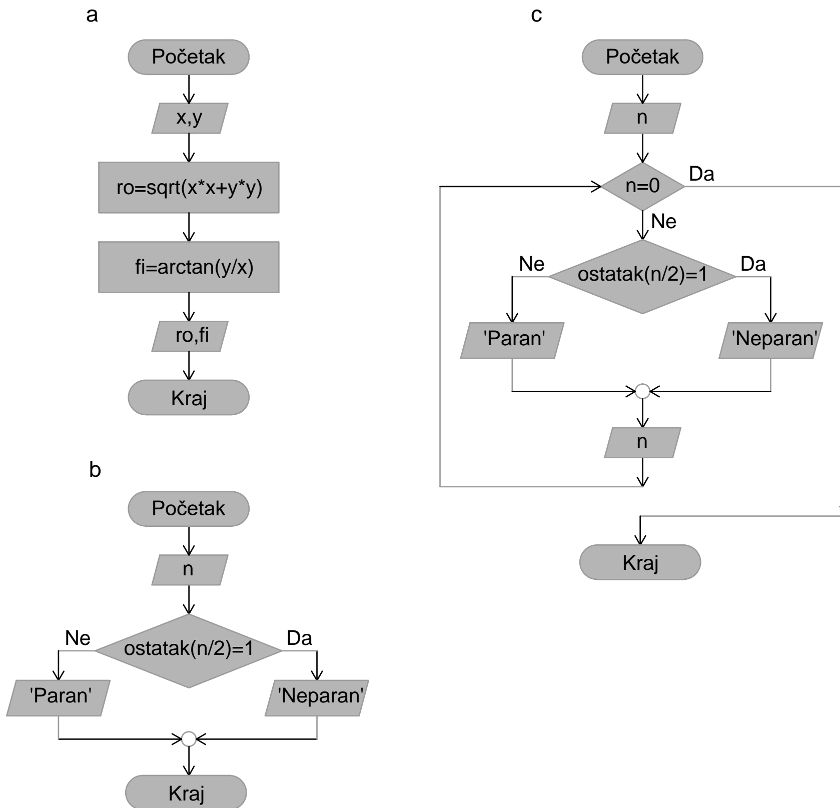
3. Kontrola toka. Iterativni algoritmi

U ovom poglavlju razmatraju se osnovni elementi jezika bez kojih bi rešavanje problema putem računara bilo nemoguće, a odnose se na *izbor* koraka koje treba obaviti u toku rada programa. Ovaj izbor, sem kod najjednostavnijih postupaka, zavisi od ulaznih veličina. Razmatraće se mehanizmi grananja i ponavljanja. Potom će biti reči o iterativnim algoritmima koji nalaze široku primenu u svim inženjerskim disciplinama, a naročito za potrebe kompleksnih matematičkih izračunavanja.

3.1 Kontrola toka

Algoritmi u kojima se, bez obzira na vrednosti ulaznih podataka, svi koraci obavljaju tačno jednom, po redosledu navođenja, nazivaju se *sekvencijalni*. Sekvencijalni *tok izvršavanja* podrazumeva da interpreter, po obradi prethodnog reda u izvornom kodu, prelazi uvek na sledeći, sve dok se ne dođe do kraja programa. Sekvencijalni algoritmi imaju skromne izražajne mogućnosti pa se pomoću njih rešavaju jednostavniji problemi. Na slici 3.1-a prikazan je primer takvog algoritma koji, za zadatu tačku iz prvog kvadranta, transformiše njene pravougle koordinate u polarne.

U najvećem broju slučajeva postupak izračunavanja zahteva da se, u zavisnosti od ulaznih parametara koji opisuju problem, prođe kroz različite putanje u programu. Posmatra se trivijalan zadatak određivanja parnosti proizvoljnog celog broja. Postupak podrazumeva *ispitivanje* ostatka broja pri celobrojnom deljenju sa dva. Ako je ostatak jednak nuli onda je broj paran, u suprotnom je neparan. Programski jezici treba



Slika 3.1: Sekvencijalni (a), razgranati (b) i repetitivni algoritam (c). Značenje pojedinih elemenata u dijagramu toka objašnjeno je u glavi 1, slika 1.4.

da ponude odgovarajući mehanizam kojim se može ispitati ispunjenost potrebnog uslova. Taj mehanizam, u zavisnosti od ishoda ispitivanja, tok programa preusmerava u željenom pravcu. *Razgranati algoritmi*, koji omogućavaju alternativne putanje u izvršavanju, mogu da posluže za rešavanje kompleksnijih problema u odnosu na svoje "sekvencijalne rođake" (slika 3.1-b).

Da bi se opisao proizvoljni postupak izračunavanja, programski jezik treba da poseduje mehanizam koji omogućava *ponavljanje* pojedinih koraka željeni broj puta ili sve dok se ne ispuni potreban uslov. Na primer, treba napisati program koji, za uneti niz prirodnih brojeva, ispituje njihovu parnost i ispisuje odgovarajuće poruke. Postupak se prekida kada korisnik unese nulu. Rešenje podrazumeva da se postupak sa slike 3.1-b ponavlja za svaki uneti broj, sve dok se unose brojevi različiti od nule (slika 3.1-c). Postupak sa slike 3.1-c definiše *repetitivni algoritam* koji prolazi kroz pojedine putanje više puta.

U Pajtonu, kao i u većini programskih jezika, kontrola programskog toka realizuje se uz pomoć naredbi `if`, `while` i `for`.

3.1.1 Grananja

Razgranati algoritmi realizuju se u programu primenom naredbe `if`. Naredba `if` može se javiti u tri oblika: jednogranom, dvogranom i višegranom (slika 3.2).

```
if  uslov:
        blok_1

naredba_posle
```

```
if  uslov:
        blok_1
else:
        blok_2

naredba_posle
```

```
if  uslov_1:
        blok_1
elif uslov_2:
        blok_2
elif uslov_3:
        .....
        .....
elif uslov_n-1:
        blok_n-1
else:
        blok_n

naredba_posle
```

Slika 3.2: Naredba `if` i varijante grananja: jednograna (levo), dvograna (sredina) i višegrana struktura (desno).

Jednograna naredba `if`

Struktura jednograne naredbe `if` prikazana je na slici 3.2, levo. Uslov između ključne reči `if` i dvotačke (`:`) predstavlja *logički izraz* čija vrednost može biti istinita (`True`) ili lažna (`False`). Logički izrazi grade se najčešće uz pomoć relacija poređenja i logičkih operacija `and`, `or` i `not`. Na primer, izraz koji testira da li tačka (x,y) pripada prvom kvadrantu glasi: `x >= 0 and y >= 0`. Da bi izraz bio tačan (tačka pripada prvom kvadrantu), oba uslova povezana operacijom `and` moraju biti ispunjena.

Ako je odgovarajući uslov naredbe `if` ispunjen, tok programa prebacuje se na *blok* naredbi `blok_1`. Blok predstavlja uzastopni niz naredbi koje čine jednu smislenu programsku celinu. Program, kao jedinstvena celina, tretira se kao sveobuhvatni blok. Sve naredbe bloka `if` moraju biti *uvučene udesno* za jednak broj mesta u odnosu na početak reda u kome se nalazi `if`. Prva naredba van bloka, `naredba_posle`, pomerena je sleva u odnosu na blok i poravnata je sa naredbom `if`. Ako uslov nije ispunjen, `blok_1` se preskače i izvršava se `naredba_posle`. Ova naredba izvršava se i kada je uslov ispunjen, ali pošto se prvo izvrše sve naredbe u bloku.

Naglašava se da svaki blok naredbe `if` može da sadrži proizvoljan broj *umetnutih* `if` blokova. Na taj način tok programa nastavlja da se grana u zavisnosti od odgovarajućih logičkih uslova.

Prilikom formiranja novog bloka, po opšteprihvaćenoj *konvenciji*, njegove naredbe pomeraju se za *četiri mesta* u odnosu na roditeljski blok. Po navođenju dvotačke i pritisku na taster <Enter>, IDLE prelazi u novi red i formira novi blok koji je pomeren nadesno za četiri mesta. Program, kao sveobuhvatni blok, mora započeti u *prvoj* koloni.

! Pomeranje (*indentacija*) svih naredbi bloka za jednak broj mesta udesno, u odnosu na roditeljski blok, predstavlja kontroverznu sintaksnu odliku jezika. Većina modernih jezika označava blok korišćenjem vitičastih zagrada (leva za početak, a desna za kraj bloka), čime se ostavlja programeru da po volji započne redove u izvornom kodu. Ako se pravilo indentacije ne poštuje, interpreter prijavljuje sintaksnu grešku!

Problem 3.1 — Par-nepar. Igra par-nepar odvija se tako što računar "pita" igrača za celobrojni ulog koji stavlja u opticaj. Potom računar zamišlja broj x, a igrač pogađa da li je x paran ili ne. Pogodak igraču donosi a promašaj uzima novac u visini uloga. Napisati program koji realizuje ovu igru. ■

Da bi se igra realizovala potrebno je da računar *može* da "zamisli" *slučajan* broj. Slučajni brojevi razmatraće se u glavi 6, a ovde će slučajnost biti *improvizovana* tako što će zamišljeni broj biti jednak unetoj visini uloga. Ovakvo ponašanje programa nije unapred poznato igraču, ali se predloženo pravilo može naslutiti posmatranjem ishoda većeg broja odigranih partija. U idealnom slučaju, kada računar zaista zamišlja slučajan broj, verovatnoća pogotka iznosi 50%. Kako igrači imaju tendenciju da nude "okrugle" uloge (10, 100, 150...), pobednička strategija tipuje na paran broj. Sledi realizacija programa:

```
1  # igra par-nepar
2
3  ulog = int(input('koliki je ulog (ceo broj): '))
4  pokusaj = int(input('par (0) ili nepar (1) '))
5
6  poruka = 'izgubili ste'
7  if pokusaj == ulog % 2:
8      poruka = 'zaradili ste'
9
10 print(poruka, ulog)
```

U programu je inicijalno načinjena pretpostavka da će igrač izgubiti - definisanje poruke koja se ispisuje na kraju igre (r6). Potom se vrši provera da li je igrač pogodio parnost zamišljenog broja (jednak ulogu) (r7). Operacija %, u logičkom izrazu naredbe `if`, daje ostatak pri celobrojnom deljenju. Ako je uslov ispunjen, programski tok prebacuje se na blok sa naredbom dodele vrednosti (r8). Tom prilikom menja se tekstualni objekat na koga ukazuje promenljiva `poruka`. Bez obzira na korisnički unos, u r10 se izvršava funkcija `print()` koja ispisuje odgovarajući tekst. Sledi prikaz rada programa pošto je dva puta pokrenut iz IDLE editora (uz pomoć <F5>):

```
koliki je ulog (ceo broj): 15
par (0) ili nepar (1) 0
izgubili ste 15
>>>
==== RESTART: C:/programi/p3_1a.py ====
koliki je ulog (ceo broj): 10
par (0) ili nepar (1) 0
zaradili ste 10
```

Dvograna naredba `if-else`

Izloženo rešenje igre par-nepar pretpostavlja da korisnički unosi predstavljaju zapise celih brojeva, odnosno da se za par unosi `'0'`, a za nepar `'1'` (r3). Isti problem može da se reši primenom dvograne naredbe `if`, pri čemu se obezbeđuje adekvatna obrada pogrešnog unosa za parnost:

Program 3.1 — Par-nepar.

```
1  # igra par-nepar
2  ulog = int(input('koliki je ulog (ceo broj): '))
3  pokusaj = int(input('par (0) ili nepar (1) '))
4  
5  if pokusaj == 0 or pokusaj == 1:
6      if pokusaj == ulog % 2:
7          print('zaradili ste', ulog)
8      else:
9          print('izgubili ste', ulog)
10 else:
11     print('za parnost se mora uneti 0 ili 1!')
```

Dvograna naredba `if` podrazumeva postojanje bloka naredbe `else` koji se izvršava kada uslov nije ispunjen. Ako unos parnosti nije adekvatan (r5), izvršava se blok `else` u kome se ispisuje poruka upozorenja (r11). Za korektan unos ispituje se parnosti i ispisuje ishod partije (r6-9). Tom prilikom koristi se unutrašnja `if-else` naredba koja je umetnuta u blok prve naredbe `if`.

Višegrana naredba `if-elif-else`

Kada postoji potreba da se, u zavisnosti od složene kombinacije uslova, izvrši *samo jedan* od tri ili više mogućih blokova, koristi se višegrana naredba `if-elif-else`. Njena primena biće ilustrovana na već poznatom problemu iz prethodne glave (2.4), gde je trebalo odrediti veći od dva uneta broja. Nedostatak prikazanog rešenja ogledao

se u nemogućnosti da se prikaže adekvatna poruka za slučaj kada su brojevi jednaki. Varijanta koja koristi višegranu naredbu `if` data je u sledećem prikazu:

```
# ispisuje veći od dva uneta broja
# ako su brojevi jednaki ispisuje prigodnu poruku

x = float(input('x= '))
y = float(input('y= '))

if x > y:
    print('veći broj je', x)
elif x < y:
    print('veći broj je', y)
else:
    print('brojevi su jednaki!')
```

Uslovni izraz

Često je potrebno da se, na osnovu ispunjenosti nekog uslova, promenljivoj pridruži vrednost jednog od dva moguća izraza. Na primer, neka je potrebno izračunati apsolutnu vrednost unetog broja bez korišćenja funkcije `abs()`. Umesto da se koristi dvograna naredba `if-else`, može se upotrebiti *uslovni izraz*:

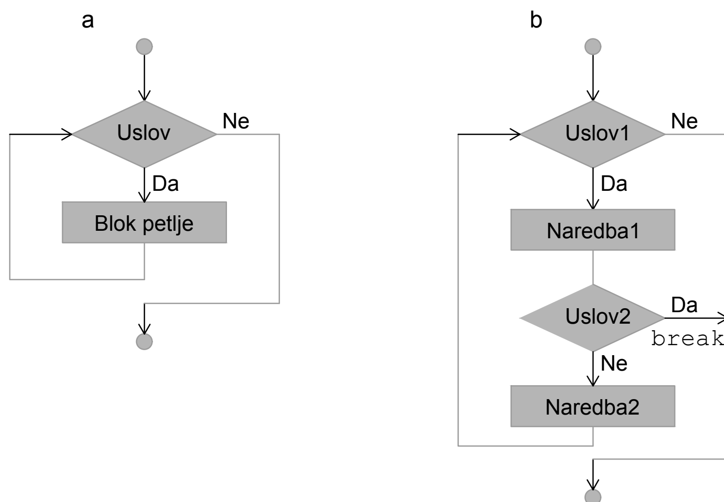
```
# izracunava apsolutnu vrednost unetog broja
x = float(input('x= '))

# umesto:
# if x < 0:
#     abs_x = -x
# else:
#     abs_x = x
#
# bolje je:
abs_x = -x if x < 0 else x
print('apsolutna vrednost za', x, 'je', abs_x)
```

Desna strana jednakosti u r11 predstavlja uslovni izraz čija se vrednost izračunava na sledeći način: ako je navedeni uslov ispunjen (`x<0`), onda se promenljivoj sa leve strane (`abs_x`) pridružuje vrednost izraza levo od naredbe `if` (`-x`); u suprotnom, pridružuje se vrednost izraza posle naredbe `else` (`x`).

3.1.2 Petlje

Postupci koji zahtevaju višestruko ponavljanje pojedinih operacija realizuju se pomoću naredbi `while` i `for`.

Petlja `while`



Slika 3.3: Dijagram toka petlje `while`: (a) testiranje uslova za izvršavanje obavlja se pri vrhu; (b) petlja se može napustiti pre vremena, ako je ispunjen uslov za izvršavanje komande `break`.

Naredba `while` omogućava da se odgovarajući programski blok ponavlja u *petlji* sve dok je ispunjen *uslov za izvršavanje petlje* koji se navodi neposredno iza `while` (slika 3.3-a). Svaki prolaz kroz naredbe bloka petlje zove se *iteracija*. Program može da se zadrži u petlji *beskonačno dugo* ako je uslov za izvršavanje uvek ispunjen. Zbog toga je potrebno obezbediti da se ispunjenost uslova može *menjati* unutar bloka petlje `while`. Petlja se uopšte ne izvršava samo ako uslov nije ispunjen pri prvom testiranju.

Problem 3.2 — Najveći broj u nizu. Napisati program kojim se sa tastature učitava niz brojeva. Unosi se broj po broj, sve dok se ne unese tekst koji označava pozitivnu beskonačnost (`'inf'`). Po završetku unosa treba prikazati najveći uneti broj. ■

Program 3.2 — Najveći broj u nizu.

```
1  # maksimalan broj u nizu
2
3  inf = float('inf') # inf ukazuje na specijalni objekat tipa
4                     # float koji označava + beskonačno(Infinity)
```

```
5  broj = float(input('unesite broj (inf za kraj) '))
6  if broj == inf:
7      print('niz je prazan')
8  else:
9      maks = broj # prvi broj postaje tekuci maksimum
10     while broj != inf:
11         if broj > maks:
12             maks = broj
13         broj = float(input('unesite broj (inf za kraj) '))
14     print('najveći broj u nizu je', maks)
```

U glavi 2.3.4 navedeno je da konstruktor `float()` pretvara tekstualnu reprezentaciju realnog broja u odgovarajući objekat tipa `float`. Ako se pri pozivu konstruktora navede tekst `'inf'`, kreira se objekat koji označava pozitivnu beskonačnost (r3). Ovaj objekat može učestvovati u aritmetičkim i logičkim operacijama. Na primer, izraz `2 * float('inf') - 10` izračunava se na `inf` (beskonačno), a izraz `100 < float('inf')` ima vrednost `True`.

Algoritam za pronalaženje maksimuma realizovan je upotrebom petlje `while` koja omogućava učitavanje i obradu svakog broja ponaosob, sve dok je učitani broj različit od beskonačnosti. Postupak započinje tako što prvi učitani broj (r5), ako nije beskonačan (r6), postaje tekući maksimum (r9). Inače, konstatuje se da je niz prazan i program se završava (r7). Brojevi se obrađuju u petlji `while` (r10-13) tako što se učitani broj poredi sa tekućim maksimumom (r11). Ako je broj veći, maksimum se ažurira (r12). Posle poređenja vrši se novo učitavanje (r13) i postupak se ponavlja. Po izlasku iz petlje, promenljiva `maks` ukazuje na najveći broj u nizu (r14). Da bi se bolje razumeo mehanizam petlje `while`, rad programa biće prikazan pomoću *tabele stanja* relevantnih promenljivih u pojedinim fazama izvršavanja – pustiće se da program "radi na papiru" (tabela 3.1).

vreme	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**izlaz**
red	5	9	13	13	13	13	12	13	14
broj	5		4	3	2	10		inf	
maks		5					10		10

Tabela 3.1: Prikaz vrednosti na koje ukazuju promenljive postavljene u navedenim redovima programa: dat je primer za ulazni niz `[5,4,3,2,10,inf]`. Stanja se posmatraju u diskretnim trenucima koji nisu ekvidistantni.

Sledeći prikaz ilustruje rad programa za niz `[5, 4, 3, 2, 10, inf]`:

```
unesite broj (inf za kraj) inf
niz je prazan
>>>
==== RESTART: C:/programi/p3_2.py ====
unesite broj (inf za kraj) 5
unesite broj (inf za kraj) 4
unesite broj (inf za kraj) 3
unesite broj (inf za kraj) 2
unesite broj (inf za kraj) 10
unesite broj (inf za kraj) inf
najveći broj u nizu je 10.0
```

Drugi pristup za pronalaženje najvećeg broja pretpostavlja da se promenljiva `maks`, umesto na prvi broj iz niza, inicijalizuje na negativnu beskonačnost. Kako je svaki regularan broj veći od $-\infty$, u postupku se, već pri prvom učitavanju, maksimum postavlja na prvi broj. Sada učitavanje pre početka petlje nije potrebno, a modifikovani program glasi:

```
1  # maksimalan broj u nizu
2
3  inf = float('inf')
4  maks = -inf                  # svi brojevi veći od -beskonačno
5
6  while True:                  # beskonačna petlja
7      broj = float(input('unesite broj (inf za kraj)'))
8      if broj == inf:
9          break                # uneto je inf, izađi iz petlje
10     elif broj > maks:
11         maks = broj
12
13 if maks == -inf:              # prvi broj koji je unet bio je inf
14     print('niz je prazan')
15 else:
16     print('najveći broj u nizu je', maks)
```

Uočava se bitna razlika u odnosu na prethodno rešenje: uslov ostanka u petlji `while` (r6) predstavlja nepromenljivu logičku istinu, pa bi se petlja izvršavala beskonačno dugo, pod uslovom da se u r9 ne nalazi naredba `break`. Ako tekući broj predstavlja beskonačnost (r8), petlja se napušta (r9), a tok programa prebacuje na prvu naredbu posle bloka petlje (r13). Kako je moguće da je prvi učitani broj ujedno značio i kraj unosa (prazan niz), neophodno je proveriti vrednost promenljive `maks` (r13).

Naredba `break` (slika 3.3-b) omogućava da se iz petlje izađe u bilo kom trenutku. Uz pomoć ove naredbe može se *simulirati* petlja sa testiranjem na dnu koja je rasprostranjena u modernim jezicima, a koja omogućava da se, za razliku od petlje `while`, kroz blok petlje prođe *bar jednom*. Sledi šablon petlje sa izlaskom na dnu, realizovan uz pomoć naredbi `while`, `if` i `break`:

```
# petlja sa testiranjem na dnu

while True:
    # .....
    # u nekom trenutku uslov izlaska mora biti ispunjen
    # .....
    if uslov:
        break

# ... naredbe posle petlje
```

Prilikom rešavanja raznorodnih problema postoji potreba da se određeni algoritamski koraci ponove *unapred poznati* broj puta. Tada se koriste *brojačke petlje* koje uvode pojam *brojača*. Brojač je promenljiva koja ukazuje na to koliko je puta obavljen željeni blok operacija. U svakoj iteraciji petlje brojač se obično uvećava za jedan, a petlja se završava kada je dostignut potreban broj ponavljanja. Koncept brojačke petlje ilustrovan je na primeru ispisivanja unete poruke željeni broj puta:

```
# papagaj koji ponavlja unesenu poruku n puta

poruka = input('vaša poruka? ')
n = int(input('koliko često? '))

i = 0 # inicijalizacija
while i < n:                 # da li je kraj
    print(i+1, poruka)       # koraci koji se ponavljaju n puta
    i = i + 1                # uvećanje brojača

print('zadovoljni? ')
```

Primer ilustruje tri aktivnosti koje karakterišu svaku brojačku petlju. Prva aktivnost odnosi se na *inicijalizaciju brojača* (promenljiva `i`, r6). Druga aktivnost podrazumeva testiranje *uslova završetka* petlje (r7). Konačno, u bloku petlje definiše se i *prav-*

ilo ažuriranja brojača (r9). Ovde se brojač postavlja na nulu, a potom se u svakoj iteraciji uvećava za jedan. Kako je cilj petlje da ponovi ispis poruke `n` puta, uslov završetka nije ispunjen sve dok je brojač manji od `n` – dozvoljene vrednosti brojača su `0,1,2,...,n-1`. Pored poruke, kao prvi argument funkcije `print()`, ispisuje se i redni broj iteracije (r8). Uočiti da je redni broj veći od tekuće vrednosti brojača za jedan, pošto se prvi ispis realizuje za `i=0`. Rad programa ilustrovan je sledećim prikazom:

```
vaša poruka? Pajton je zakon!
koliko često? 3
1 Pajton je zakon!
2 Pajton je zakon!
3 Pajton je zakon!
zadovoljni?
```

Petlja `for`

Na osnovu prethodnog izlaganja je jasno da petlja `while` predstavlja *najopštiji* način za realizovanje ponavljanja algoritamskih koraka. Može se koristiti kako za situacije kada broj ponavljanja nije poznat, tako i za zadatke u kojima se on unapred zna (brojačke petlje). Međutim, za brojačke petlje pogodnije je koristiti drugi mehanizam – naredbu `for`. Isti zadatak rešava se primenom petlje `for` na sledeći način:

```
1 # papagaj koji ponavlja unesenu poruku n puta
2
3 poruka = input('vaša poruka? ')
4 n = int(input('koliko često? '))
5
6 for i in range(n):          # i = 0,1,2,    ,n-1
7     print(i+1, poruka)      # koraci koji se ponavljaju n puta
8
9 print('zadovoljni? ')
```

U varijanti sa petljom `for`, programer se oslobađa obaveze da eksplicitno inicijalizuje i ažurira brojačku promenljivu pa je rešenje kraće. Funkcija `range()` generiše *sekvencu* celih brojeva, kao što je prikazano u tabeli 3.2. Funkcija je zapravo konstruktor jer se njome *kreira* objekat klase `range` koji obuhvata više celobrojnih objekata. Sekvence se razmatraju u glavi 5 koja se bavi grupisanjem objekata.

Naredba `for`, u r6, čita se na sledeći način: "za svako `i` iz sekvence od `0` do `n-1`, uradi sledeće: ...". Promenljiva `i`, u svakoj iteraciji petlje, ukazuje na jedan celobrojni objekat čija je vrednost u rasponu od `0` do `n-1`. Ako su argumenti za `range()` takvi da je sekvenca prazna, brojačka promenjiva ne ukazuje ni na jednu konkretnu vrednost

poziv funkcije	**generisana sekvenca**
`range(n)`	za `n > 0: 0, 1, 2, ..., n-1`
`range(start, stop)`	za `stop > start:` `start, start+1, start+2, ..., stop-1`
`range(start, stop, korak)`	za `stop > start` i `korak > 0:` `start, start+korak, start+2*korak, ..., n` važi `stop - korak <= n < stop`
`range(start, stop, korak)`	za `stop < start` i `korak < 0:` `start, start+korak, start+2*korak, ..., n` važi `stop - korak >= n > stop`

Tabela 3.2: Generisanje sekvenci celih brojeva: mogu biti kako rastuće, tako i opadajuće.

pa se blok petlje ne izvršava. Brojačka promenljiva najčešće se koristi za različita izračunavanja u bloku petlje, što je ilustrovano u problemu koji sledi.

Problem 3.3 — Kvadratna matrica. Izračunati sumu svih brojeva u celobrojnoj kvadratnoj matrici veličine 2021×2021 i sledećeg oblika:

$$\begin{matrix} 1 & 2 & 3 & \dots & n \\ 2 & 4 & 6 & \dots & 2n \\ 3 & 6 & 9 & \dots & 3n \\ \dots & \dots & \dots & \dots & \dots \\ n & 2n & 3n & \dots & n^2 \end{matrix}$$

■

Da bi se tražena suma izračunala, treba *otkriti zakonitost* po kojoj su brojevi raspoređeni u matrici. Uočava se da je element u preseku i-te vrste i j-te kolone jednak proizvodu $i \cdot j$ pa je suma svih brojeva u matrici data sa:

$$S = \sum_{i=1}^{n} \sum_{j=1}^{n} i \cdot j \tag{3.1}$$

Suma ima n^2 sabiraka oblika $i \cdot j$. Za svaku fiksiranu vrednost i postoji n sabiraka oblika $i \cdot 1, i \cdot 2, \dots, i \cdot n$. Sumiranje se obavlja uz pomoć *petlje u petlji* na sledeći način:

```
1  # računa sumu i*j za sve kombinacije i,j = 1..n
2
3  n = int(input('n = '))
4
```

```
5  suma = 0                     # promenljiva koja akumulira sumu
6  for i in range(1,n+1):       # prvi broj i = 1,2,...,n
7      for j in range(1, n+1): # drugi broj j = 1,2,...,n
8          suma += i*j          # umesto suma = suma + i * j
9
10 print('suma =', suma)
```

Problem je rešen u tipičnom *iterativnom postupku* u kome se suma, zadata sa (3.1), *akumulira* u promenljivoj koja je inicijalizovana u r5. Sve kombinacije za i i j generišu se primenom dve brojačke petlje od kojih se jedna nalazi unutar druge. *Unutrašnja* petlja, koja generiše drugi indeks (r7), izvršava se *brže* od *spoljašnje* petlje (r6) koja odgovara prvom indeksu. Ovde postoji analogija sa kazaljkama na časovniku: mala kazaljka (spoljašnja petlja) odbroji za jedan, kada velika kazaljka (unutrašnja petlja) obrne pun krug (prođe kroz sve moguće vrednosti).

Sabiranje oblika `a = a + b` može se skraćeno zapisati kao `a += b` (r8). Kako se često javlja potreba da se neka promenljiva ažurira na opisani način, to se navedeno sintaksno rešenje primenjuje i u drugim jezicima poput Jave ili C-a. Notacija se može upotrebiti i za oduzimanje, množenje i deljenje. Na primer, `a /= 5` ekvivalentno je sa `a = a / 5`. U r8, usled *prioriteta* aritmetičkih operacija, obavlja se prvo množenje (`i * j`), pa onda sabiranje.

Suma (3.1) ima *simetričnu* prirodu koja proističe iz komutativnosti množenja ($i \cdot j = j \cdot i$) pa se izraz transformiše u sledeći oblik:

$$S = \sum_{i=1}^{n} i^2 + 2 \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} i \cdot j \quad = \quad n^2 + \sum_{i=1}^{n-1} \left\{ i^2 + 2 \sum_{j=i+1}^{n} i \cdot j \right\} \tag{3.2}$$

Simetričnost problema dozvoljava da se broj sabiraka iz sume (3.1), sa n^2, smanji na $n + \frac{n^2 - n}{2}$ u sumi (3.2). Za veliko n, broj sabiraka približno se redukuje na pola. Efikasniji algoritam (3.2) generiše konačnu varijantu rešenja:

Program 3.3 — Kvadratna matrica.

```
1  # računa sumu i*j za sve kombinacije i,j = 1..n
2
3  n = int(input('n = '))
4
5  suma = n**2   # promenljiva koja akumulira sumu
6  for i in range(1, n):   # prvi broj i = 1,2,...,n-1
7      suma += i**2        # skraćena notacija za suma = suma + i**2
8      sij = 0
9      for j in range(i+1, n+1): # drugi broj j = i+1,i+2,...,n
```

```
        sij += i * j  # skraćena notacija za sij = sij + i * j
    suma += 2 * sij   # umesto suma = suma + 2 * sij

print('suma =', suma)
```

Promenljiva `sij`, koja akumulira sumu po brojaču j (3.2), inicijalizuje se u r8. Rešenje je moglo biti i bez upotrebe promenljive `sij` kada bi se, u r6, na promenljivu `suma` dodavalo `2*i*j`. Međutim, ono ne bi bilo efikasnije pošto bi se, umesto jednog, obavljalo mnogo više množenja sa dva. Efekat izvršavanja programa "na papiru", za `n = 3`, prikazan je u tabeli 3.3.

vreme	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**izlaz**
`red`	5	6	7	8	9	10	9	10	11	6	7	8	9	10	11	13
`i`	1								2							
`j`					2		3						3			
`sij`				0		2		5				0		6		
`suma`	9		10						20		24				36	36

Tabela 3.3: Prikaz vrednosti na koje ukazuju promenljive, postavljene u navedenim redovima programa, za `n = 3`. Stanja se posmatraju u diskretnim trenucima koji nisu ekvidistantni.

Tabela stanja može se formirati za proizvoljni program, ali je to praktično izvodljivo samo za najjednostavnije edukativne probleme sa ograničenim skupom ulaznih podataka. Na kraju, odgovor na pitanje iz formulaciji problema dobija se ako se program pokrene za $n = 2021$: tražena suma iznosi 4 174 792 919 361.

3.2 Iterativni algoritmi

Iterativni algoritam predstavlja postupak za rešavanje problema u kome se određeni skup koraka ponavlja sve dok se ne ostvari unapred definisani *stepen ostvarenosti cilja*. Stepen ostvarenosti može biti definisan na različite načine. Na primer, može se zahtevati da rešenje bude približno tačno (definisano određenim brojem značajnih cifara). Naravno, može se tražiti i potpuno ostvarenje cilja, ako je to moguće. Iterativni algoritam *uvek* sadrži izvesne promenljive koje opisuju stanje *trenutne pozicije* u prostoru problema. Ove promenljive ažuriraju se u svakoj sledećoj iteraciji petlje `while` ili `for`, sve dok se ne ispuni odgovarajući stepen ostvarenosti.

3.2.1 Metoda uzastopne aproksimacije

U inženjerskim primenama, za potrebe različitih izračunavanja, često je potrebno sprovesti standardne matematičke zadatke poput nalaženja nula funkcije, integraljenja, rešavanja sistema (ne)linearnih jednačina i sl. Ovi zadaci često se rešavaju *približno*, korišćenjem iterativnih algoritama zasnovanih na metodama numeričke analize.

Problem 3.4 — N-ti koren. Izračunati $\sqrt[n]{a}$ primenom operacija $+, -, /, *$ i $**$. ■

Rešenje problema zasniva se na Njutn-Rafsonovoj metodi za određivanje nula polinoma $P(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$. Njutn je pokazao da, za vrednost x_i *dovoljno blisku* nuli polinoma $P(x)$, važi da je vrednost

$$x_{i+1} = x_i - \frac{P(x_i)}{P'(x_i)} \tag{3.3}$$

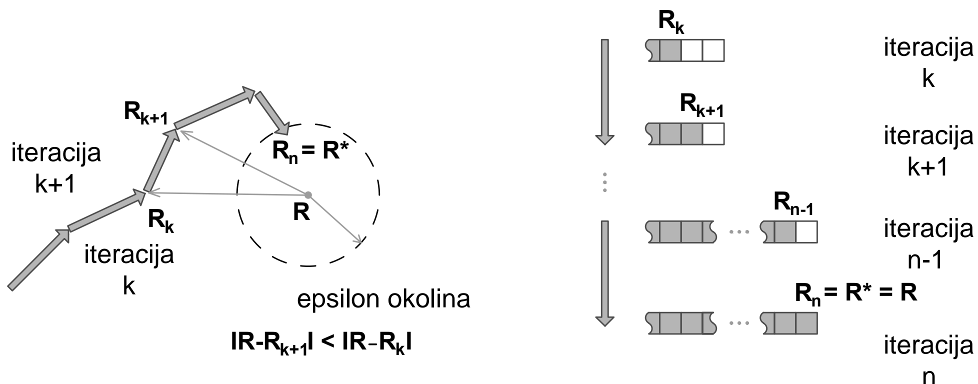
još bliža nuli $P(x)$. Ovde $P'(x_i)$ predstavlja prvi izvod polinoma u tački x_i. Kako je $\sqrt[n]{a}$ nula polinoma $P(x) = x^n - a$, to se izraz (3.3) može upotrebiti za iterativno izračunavanje n-tog korena, pod uslovom da se, u prvoj iteraciji, za x_1 uzme *dovoljno bliska* vrednost. Izraz (3.3) transformiše se u:

$$x_{i+1} = x_i - \frac{x_i^n - a}{n x_i^{n-1}} \tag{3.4}$$

Program 3.4 — N-ti korena broja.

```python
# računa n-ti koren iz a primenom Njutn-Rafsonovog metoda

a = float(input('a = '))
n = int(input('n = '))
delta = float(input('delta = '))  # definiše posrednu meru
                                  # tačnosti rešenja
if n <= 0:
    print('n mora biti veći od nule!')
elif a < 0 and n % 2 == 0:
    print('ne postoji realni parni koren iz negativnog broja')
else:
    koren = a / n   # početna aproksimacija n-tog korena
    while abs(koren**n - a) > delta:
        koren = koren - (koren**n - a) / (n * koren**(n-1))
    print(n, 'koren za', a, 'je', koren)
```

U petom redu programa unosi se *posredna* mera tačnosti izračunavanja kojom se definiše uslov za završetak iterativnog postupka (r13). Strogo govoreći, tačnost predstavlja apsolutnu razliku između približnog i tačnog rešenja, ali se ona ovde ne može koristiti jer je tačno rešenje nepoznato! Za početnu aproksimaciju n-tog korena uzet je n-ti deo vrednosti broja (r12). U svakoj iteraciji petlje `while` tekuće rešenje na koje ukazuje promenljiva `koren` se popravlja pa se postupak zove još i *metoda uzastopnih aproksimacija* (slika 3.4-a). Treba istaći da je navedeni iterativni postupak primenljiv samo u slučajevima kada formule, po kojima se tekuće rešenje prevodi u sledeće, *konvergiraju* ka tačnom rešenju. U suprotnom, postupak nema smisla.



Slika 3.4: Iterativni algoritmi: (a) metoda uzastopne aproksimacije - rešenje se poboljšava kroz iteracije. Sa R je označeno tačno, a sa R* približno rešenje. Definisanjem epsilon okoline (poluprečnik oko R), određuje se željena tačnost. Međutim, kako je R nepoznato, primenjuje se neka od posrednih mera poput one iz problema 3.4. (b) metoda iterativne konstrukcije - rešenje je kompozitno pa se sastavlja kroz iteracije. Primetiti da se, za razliku od metode uzastopne aproksimacije, ovde uvek dobija tačno rešenje.

! Obratiti pažnju kada se dva broja tipa `float` porede po jednakosti. Umesto poređenja `a == b`, uvek je bolje testirati da li važi `abs(a - b) < eps`, odnosno da li su brojevi `a` i `b` dovoljno blizu! Razlog tome je približno predstavljanje relanih brojeva u memoriji o čemu je bilo reči u prethodnoj glavi. Na primer, `3 * 0.1` daje `0.30000000000000004`!

Greške koje se čine pri zaokruživanju u iterativnom postupku najčešće ne predstavljaju veliki problem, ali ih treba imati na umu kada se tumače konačni rezultati.

3.2.2 Metoda iterativne konstrukcije

Iterativni postupak može se primeniti i kod problema koji podrazumevaju tačno rešenje, a čija priroda je takva da se rešenje *konstruiše* iz iteracije u iteraciju (slika 3.4-b).

Problem 3.5 — Kosa crta. Nacrtati kosu crtu primenom simbola $*$, imajući u vidu da su zvezdice raspoređene u kvadratnu šemu dimenzije n (n je uvek neparan broj) na sledeći način (u prikazu, $n = 5$):

```
        *
      * * *
  * * * * *
  * * *
  *
```

Analizom slike iz postavke problema uočavaju se sledeće pravilnosti: kosa crta sastoji se od neparnog broja redova; redovi iz gornje polovine slike počinju praznim mestima pa potom slede zvezdice, dok za redove iz donje polovine važi suprotno; ukupan broj znakova (zvezdice + prazna mesta) iznosi n; broj zvezdica u svakom redu raste za dva, počevši od prvog pa sve do sredine, a potom se smanjuje za dva.

Problem se može rešiti *dekompozicijom* (podelom) na dva manja, jednostavnija potproblema. Prvo će, u iterativnom postupku, biti iscrtana gornja polovina slike, a potom donja. Postupak dekompozicije spada u osnovne tehnike za rešavanje kompleksnih zadataka o čemu će više reči biti u glavi 4.

```python
1  # crta kosu crtu dimenzija n x n (n = 2k+1), uz pomoć *
2  
3  n = int(input('n = '))
4  if n % 2 == 1:
5  
6      sredina = (n + 1) // 2          # broj središnjeg reda
7      zvezdice = -1                   # tekući broj zvezdica u redu
8  
9      for i in range(1, sredina + 1):      # redovi do sredine
10         zvezdice += 2                    # broj zvezdica u redu i
11         for j in range(n - zvezdice):    # blanko deo u redu i
12             print(' ', end='')
13         for j in range(zvezdice):        # zvezdice u redu i
14             print('*', end='')
15         print('')                        # ispiši prazan tekst
16                                          # i pređi u nov red
17     for i in range(sredina + 1, n + 1): # redovi posle sredine
18         zvezdice -= 2                    # broj zvezdica u redu i
19         for j in range(zvezdice):        # zvezdice u redu i
20             print('*', end='')
```

```
21          print('')                           # ispiši prazan tekst
22                                              # i pređi u nov red
23  else:
24      print('n mora biti neparan!')
```

U programu se prvo proverava da li je n neparan broj (r4). Ako je broj paran, program se završava uz poruku upozorenja (r24). Gornja polovina slike crta se red po red u petlji `for` (r9-15). Promenljiva `zvezdice` označava broj zvezdica u tekućem redu i u svakoj iteraciji uvećava se za dva (r10). Pre početka petlje inicijalizovana je na -1 (r7), kako bi pri ispisu prvog reda imala vrednost 1. Prazna mesta i zvezdice crtaju se uz pomoć funkcije `print()` u petljama (r11-12) i (r13-14).

Treba uočiti upotrebu opcionog argumenta `end = ''` koji specificira da se na kraju, posle prikaza željenog teksta, ispiše navedena vrednost (ovde prazan tekst), umesto uobičajenog prelaska u novi red kada je opcioni argument izostavljen (r15). Sličan postupak sprovodi se i za donju polovinu slike, s tim da se prazna mesta ne ispisuju (r17-21).

U Pajtonu postoji mogućnost da se dva teksta *nadovežu* jedan na drugi upotrebom operatora `+`. Na primer, `'abc'+'123'` daje `'abc123'`. Slčno, tekst se može *umnožiti* željeni broj puta na sledeći način: ako `x` predstavlja objekat tekstualnog tipa, onda `n * x` predstavlja drugi tekstualni objekat koji ima vrednost `n` puta ponovljenog teksta `x`. Na primer, `3 * 'da'` daje `'dadada'`. Kombinovanjem ove dve mogućnosti uklanja se potreba za unutrašnjim petljama tipa `for`.

O tekstualnim sekvencama, kao i o operacijama nad njima, više reči biće u glavi 5. Sledi prikaz poboljšane verzije programa i test primer:

Program 3.5 — Kosa crta.

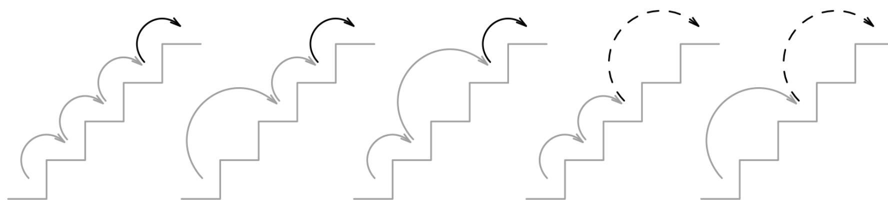
```
1   # crta kosu crtu dimenzija n x n (n = 2k+1), uz pomoć *
2
3   n = int(input('n = '))
4   if n % 2 == 1:
5
6       sredina = (n + 1) // 2        # broj središnjeg reda
7       zvezdice = -1                 # broj zvezdica u tekućem redu
8
9       for i in range(1, sredina + 1):    # redovi  do sredine
10          zvezdice += 2                  # broj zvezdica u redu i
11          print((n - zvezdice) * ' '+ zvezdice * '*') # red i
12
13      for i in range(sredina + 1, n + 1): # redovi posle sredine
```

```
14          zvezdice -= 2                   # broj zvezdica u redu i
15          print(zvezdice * '*')           # prikazuje red i
16
17  else:
18      print('n mora biti neparan!')
```

```
n = 7
      *
     ***
    *****
   *******
   *****
   ***
   *
```

Izlaganje o iterativnoj konstrukciji završava se nešto težim problemom:

Problem 3.6 — Stepenice. Stepenice imaju n stepenika. Na koliko načina se može popeti na vrh (na n-ti stepenik), ako mogu da se prave koraci od jednog ili dva stepenika. Napisati program koji računa broj načina za ulaznu veličinu n. ■



Slika 3.5: Stepenice sa četiri stepenika. Crne strelice označavaju varijante kada se na vrh dolazi jednim, a isprekidane sa dva koraka.

Često se najteži deo u rešavanju nekog zadatka odnosi na preslikavanje iz domena problema u efikasan algoritam. Slika 3.5 ilustruje slučaj sa četiri stepenika i svim mogućim varijantama penjanja na vrh. Uopšte uzevši, ilustrovanje rešenja za *jednostavnije slučajeve* vrlo često pomaže da se dođe do potrebne ideje!

Kako se penjanje obavlja u koracima po jedan, odnosno po dva stepenika, na vrh se može doći jedino sa $n-2$-og, ili $n-1$-og stepenika. Ako se sa f_{n-2} i f_{n-1} označi broj načina na koje se može popeti na stepenike $n-2$ i $n-1$, onda je broj različitih penjanja na vrh (stepenik n) jednak:

$$f_n = f_{n-1} + f_{n-2} \tag{3.5}$$

Formula (3.5) može se dokazati postupkom *matematičke indukcije*. Dokaz polazi od $n = 3$, kada postoje tri načina. Kako je $f_1 = 1$ i $f_2 = 2$, to važi da je $f_3 = f_2 + f_1$. Pretpostavlja se da (3.5) važi za $n \leq k$. Razmatraju se stepenice od $n = k+1$ stepenika. Sa slike se vidi da, kada se penje sa pretposlednjeg na poslednji stepenik $(k+1)$, to se može učiniti na samo jedan način. Dakle, postoji f_k načina da se popne na vrh, pod uslovom da se poslednji korak čini sa pretposlednjeg stepenika. Slično važi i kada se polazi sa $k-1$. stepenika. Tada može da se načini korak od jedan (što se svodi na prethodno razmotreni slučaj), ili korak od dva stepenika. Dakle, suštinski postoji samo jedan izbor koji nije prethodno uračunat pa je broj načina na koji se može stići do vrha, pod uslovom da se polazi sa $k-1$. stepenika, jednak f_{k-1}. Zato je $f_{k+1} = f_k + f_{k-1}$, što je i trebalo dokazati.

Formulom (3.5) generiše se poznata Fibonačijeva sekvenca brojeva:[1] 1, 2, 3, 5, 8, 13, 21, 34, 55, ... Opšte rešenje problema podrazumeva ispisivanje n-tog člana ove sekvence:

Program 3.6 — Stepenice.

```
# broj načina za penjanje na n-ti stepenik
# jednak n-tom članu Fibonačijevog niza 1,2,3,5,8,13,21,...
n = int(input('n = '))
if n == 1:
    print('Jedan stepenik, broj načina: 1')
elif n == 2:
    print('Dva stepenika, broj načina: 2')
else:
    Fi_2, Fi_1 = 1, 2 # F(i-2)=1, F(i-1)=2
    for i in range(3, n+1):
        Fi = Fi_1 + Fi_2   # i-ti član
        # priprema za sl. iteraciju F(i-2)=F(i-1), F(i-1)=F(i)
        Fi_2, Fi_1 = Fi_1, Fi

    print('Broj stepenika:', n, 'broj načina:', Fi)
```

Rešenje se konstruiše u iterativnom postupku tako što se, u svakoj iteraciji, tekući član formira na osnovu prethodna dva (r11). Potom se vrednosti za prethodna dva člana ažuriraju kako bi bile spremne za sledeću iteraciju (r13). Upotrebom naredbe višestruke dodele vrednosti (r9, 13), program se neznatno skraćuje. Uočiti da se u programu *ne pamte* svi, već samo poslednja tri člana sekvence.

[1] Iz originalne Fibonačijeve sekvence izostavljena su prva dva člana, 0 i 1.

3.2.3 Iterativna pretraga u prostoru rešenja

Pretpostaviti da je skup svih *potencijalnih rešenja* problema p označen sa P. Skup P naziva se *prostorom rešenja* problema p. Potencijalno rešenje koje ispunjava zahteve iz postavke problema naziva se *pravo rešenje*. U opštem slučaju, p može imati *više* pravih rešenja pa mu se pridružuje skup $R \subseteq P$ tako da su sva potencijalna rešenja iz R ujedno i prava rešenja za p. Kako bi se u postupku iterativne pretrage prostora P pronašao skup R i na taj način rešio problem p, potrebno je da se svako potencijalno rešenje može *formalno reprezentovati* na pogodan način. Na primer, za problem pronalaženja najvećeg broja u nizu (problem 3.2), sva potencijalna rešenja predstavljaju upravo brojeve iz niza. U problemu trgovačkog putinka (problem 1.2), P se može opisati kao skup svih permutacija rednih brojeva gradova. Prostor rešenja može biti i kontinualan, poput skupa realnih brojeva iz problema 3.4, u kome se izračunava $\sqrt[n]{a}$.

Pored formalne reprezentacije potencijalnog rešenja, potrebno je formulisati i logičku funkciju f kojom se utvrđuje da li je neko potencijalno rešenje ujedno i pravo. Funkcija f preslkava sva potencijalna rešenja iz R u `True`, a iz $P \setminus R$ u `False`. Kao i u slučaju reprezentacije potencijalnog rešenja, funkcija f može poprimiti različite oblike: u slučaju maksimalnog broja u nizu, funkcija bi mogla da testira da li je potencijalno rešenje veće od svih ostalih. Opet, u slučaju $\sqrt[n]{a}$, f bi proveravala da li je razlika između potencijalnog rešenja dignutog na n-ti stepen i broja a manja od prethodno zadate vrednosti. Pristup iterativne pretrage podrazumeva da se, u svakoj iteraciji, generiše određen broj potencijalnih rešenja koja se, uz pomoć f, eventualno ubacuju u skup R.

Algoritmi koji sprovode iterativnu pretragu razlikuju se po *strategiji* kojom se savladava prostor rešenja. Svima je zajednički cilj da se postigne što *efikasnija* pretraga – poželjno je ne ulaziti u one delove prostora za koje se *unapred* zna da ne mogu da sadrže prava rešenja. Neke od strategija bile su pomenute već u glavi 1, u problemu 1.1. Ovde će biti detaljnije ilustrovan algoritam *grube sile* i neka njegova poboljšanja, kao i heuristika *pohlepne pretrage*.[2]

Metoda grube sile

Ako je prostor rešenja *dovoljno mali*, onda se sva rešenja iz skupa R mogu pronaći *iscrpnom* pretragom po *svim* potencijalnim rešenjima iz P – metoda grube sile. Sledi primer:

Problem 3.7 — Delioci broja. Ispisati sve delioce učitanog prirodnog broja n. ■

U problemu sa deliocima broja n, prostor potencijalnih rešenja ima oblik $P = \{1, 2, ..., n\}$. Skup svih delilaca može se dobiti iscrpnom pretragom po P, u postupku grube sile: treba poći od broja jedan i ići sve do n, pa potom testirati da li tekući kandidat deli n bez ostatka. Primetiti da se funkcija f svodi na proveru ostatka celobrojnog deljenja. Pretraga se može ubrzati tako što bi se, umesto do n, išlo do $\lfloor n/2 \rfloor$ –

[2] Engl. *Greedy search*.

brojevi preko polovine sigurno ne mogu biti delioci (osim n). Na ovaj način početni prostor je prepolovljen i glasi $P = \{n, 1, 2, ..., \lfloor n/2 \rfloor\}$. Međutim, *optimalna* strategija podrazumeva da se ispitaju samo potencijalna rešenja od 1 do $\sqrt{n}$, te da se delioci ispisuju po *parovima*. Tom prilikom koristi se lako dokaziva činjenica: ako je $a \geq \sqrt{n}$ delilac broja n, onda je to i n/a koji mora biti manji ili jednak $\sqrt{n}$. Na primer, za $n = 12$, parovi su: (1, 12), (2, 6) i (3, 4).

Program 3.7 — Delioci broja.

```
1  # delioci
2  n = int(input('n > 0? '))
3  if n > 0:
4      print('parovi delilaca:')
5      for i in range(1, int(n**0.5) + 1):
6          if n%i == 0:
7              print(i, n//i) # nasuprot /, // daje celobrojni rez.
8  else:
9      print('n mora biti prirodan broj!')
```

U r5, gornja granica za funkciju `range()` dobija se tako što se na koren broja dodaje jedinica. Naime, kada je broj pun kvadrat (npr. $9 = 3 * 3$), ne bi se uzeo u obzir poslednji par – `range()` generiše celobrojnu sekvencu koja ne uključuje navedenu gornju granicu, već ide do njenog prethodnika! Primetiti da u slučaju punog kvadrata poslednji par sadrži dva ista delioca.

```
n > 0? 1968
parovi delilaca:
1 1968
2 984
3 656
4 492
6 328
8 246
12 164
16 123
24 82
41 48
```

Problem 3.8 — Četiri sabirka. Učitati četiri cela broja, s_1, s_2, s_3 i s_4, kao i ceo broj z. Ispisati sve kombinacije sabiraka s_i za koje je suma manja od z, uz uslov da se svaki od sabiraka u sumi pojavljuje najviše jednom. ■

Neka su dati brojevi 1, 2, 3 i 4; dalje, neka je $z = 5$. Sume čija je vrednost manja od pet ostvaruju se sa po jednim (1, 2, 3, 4), ili sa po dva sabirka (1+2, 1+3). U opštem slučaju, za brojeve s_1, s_2, s_3 i s_4, treba generisati sve moguće zbirove (kombinacije sabiraka). Pri tome se za svaku kombinaciju proverava da li je njena suma manja od z. Prostor svih potencijalnih rešenja P sastoji se od ukupno $2^4 - 1 = 15$ zbirova – u problemu sa n sabiraka ima $\binom{n}{k}$ zbirova sa po k sabiraka, pri čemu je $\sum_{k=0}^{n} \binom{n}{k} = 2^n$. Prilikom računanja ukupnog broja kombinacija treba odbaciti sumu bez sabiraka ($k = 0$). P se može posmatrati i kao partitivni skup[3] skupa $\{s_1, s_2, s_3, s_4\}$.

Svi podskupovi nekog skupa od n elemenata mogu se predstaviti nizom od n bita. Na primer, za skup od četiri sabirka s_1, s_2, s_3, s_4, niska `0000` predstavlja prazan, a niska `1111` skup sa svim sabircima. Niska `0101` predstavlja skup sabiraka s_2, s_4 – jedinica na i-tom mestu označava *prisustvo* sabirka s_i u sumi. Niska od n bita može se generisati u okviru n ugnježđenih petlji `for`, pri čemu svaki brojač, koji uzima vrednost nula ili jedan, označava odsustvo ili prisustvo odgovarajućeg sabirka:

Program 3.8 — Četiri sabirka.

```
1  # četiri sabiraka
2  s1 = int(input('s1? '))
3  s2 = int(input('s2? '))
4  s3 = int(input('s3? '))
5  s4 = int(input('s4? '))
6  z = int(input('z '))
7
8  # 4 indikatorske (0/1) petlje za generisanje svih podskupova
9  for i1 in range(2):
10     for i2 in range(2):
11         for i3 in range(2):
12             for i4 in range(2):
13                 suma = i1*s1 + i2*s2 + i3*s3 + i4*s4
14                 if suma < z:
15                     # ispis pravog rešenja
16                     if i1 == 1:
17                         print(s1, end='  ')
18                     if i2 == 1:
19                         print(s2, end='  ')
20                     if i3 == 1:
21                         print(s3, end='  ')
22                     if i4 == 1:
```

[3] Skup svih podskupova. Na primer, za skup {a, b}, partitivni skup je { ∅, {a}, {b}, {a, b}}.

```
23                  print(s4, end='  ')
24              print(' : ', suma, '<',z)
```

Pošto se unesu ulazni podaci (r2-6), tok izvršavanja prebacuje se u strukturu od četiri ugnježđene petlje `for`. Brojačke promenljive iz ovih petlji određuju jedan podskup sabiraka (potencijalno rešenje) (r9-12). Potom se formira suma od sabiraka iz tekućeg podskupa (r13). Primetiti da sabirci, koji ne pripadaju podskupu, ne učestvuju u sumi (odgovarajući brojači jednaki nuli). Ako suma zadovoljava kriterijum iz formulacije, onda se vrši ispis njenih sabiraka (r14-24). U ispisu se upotrebljava opcioni argument funkcije `print()` kojim se, umesto prelaska u novi red, vrši pomeranje za dva blanko znaka u tekućem redu (`end='  '`). Ostvarena suma ispisuje se posle sabiraka:

```
s1? 1
s2? 2
s3? 3
s4? 4
z 5
4   :  4 < 5
3   :  3 < 5
2   :  2 < 5
1   :  1 < 5
1  3   :  4 < 5
1  2   :  3 < 5
```

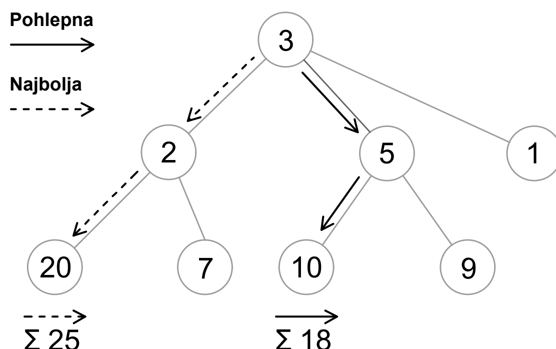
Heuristika pohlepne pretrage

U praksi se često pojavljuju problemi u kojima se, u toku rešavanja, čini niz *odluka* u cilju postizanja optimalnog rešenja. Tom prilikom, odluke se najčešće biraju iz nekog konačnog skupa. Heuristika *pohlepne pretrage* zasniva se na izboru *najbolje* odluke koja se može načiniti iz *tekućeg stanja* iterativnog postupka. U svakoj iteraciji pravi se *lokalno optimalni* izbor koji *ne mora* da vodi do najboljeg konačnog rešenja.

Neka se, za primer, posmatra stablo[4] sa slike 3.6; potrebno je pronaći putanju od početnog čvora sa vrednošću 3 do nekog od čvorova koji nemaju potomke, pri čemu suma vrednosti čvorova na putanji treba da bude maksimalna. Sa formiranjem putanje počinje se iz početnog čvora. Potom se, u sledećoj iteraciji, između čvorova sa vrednostima 1, 2 i 5, bira onaj sa vrednošću 5 – čini se lokalno optimalan izbor pošto se tekuća suma na putanji uvećava za *najveću* moguću vrednost. Konačno, u poslednjem koraku, bira se opet najbolji čvor sa stanovišta tekuće pozicije – čvor sa vrednošću 10. Putanja `3-5-10` ima vrednost od 18 poena. Međutim, ona *nije* najbolja iako je

[4] Stabla odslikavaju hijerarhijske relacije u nekom sistemu. Dobar primer je porodično stablo. Stablo sadrži `n` čvorova (grafički prikaz elemenata skupa) i `n-1` grana (između dva elementa u relaciji). Potomci nekog čvora su čvorovi koji se, u hijerarhiji stabla, nalaze neposredno ispod njega.

postignuti zbir zadovoljavajući. Za razliku od heuristike pohlepne pretrage, algoritam grube sile *pronašao* bi najbolju putanju `3-2-20`, pošto bi razmotrio *svih* pet mogućnosti iz prostora $P = \{$`3-2-20`, `3-2-7`, `3-5-10`, `3-5-9`, `3-1`$\}$.



Slika 3.6: Pohlepna pretraga: u svakom čvoru bira se potomak sa najvećom vrednošću. Najbolja putanja (`3-2-20`), razlikuje se od pohlepne (`3-5-10`).

Pošto pristup vrlo često *ne nalazi* optimalno rešenje, ne može se smatrati algoritmom, već heuristikom – treba se podsetiti da algoritmi uvek daju tačna ili optimalna rešenja (glava 1.3). Pohlepni pristup ilustrovan je prostim praktičnim primerom:

Problem 3.9 — Kusur. Knjiga "Osnove programiranja u Pajtonu" košta 1000 dinara. Student prilikom kupovine daje n dinara. Ako se pretpostavi da je student dao dovoljan iznos, napisati program koji će pomoći prodavcu da vrati kusur sa najmanjim brojem novčanica. Prodavac raspolaže sa dovoljnim brojem novčanica od a, b i c dinara. ■

Pretpostaviti da na najveću vrednost ukazuje promenljiva `a` (`a` dinara). Pohlepni pristup pokušava da ostvari kusur sa najmanjim brojem novčanica tako što, u svakoj iteraciji, dok je to *moguće*, vraća po jednu novčanicu od `a` dinara. Očigledno, za kusur od `k` dinara, novčanica od `a` dinara može se vratiti `k // a` puta (primenjeno celobrojno deljenje). Ostatak od `k % a` dinara treba vratiti uz pomoć preostalih novčanica. Postupak se ponavlja tako što se, uz pretpostavku da je sledeća vrednost po veličini `b`, od ostatka vraća, sve dok je to moguće, novčanica od `b` dinara. Konačno, preostale pare vraćaju se, ako je to moguće, uz pomoć novčanica od `c` dinara.

Da bi se gore pomenuti postupak mogao realizovati, neophodno je prvo *urediti po veličini* učitane vrednosti tako da važi `a > b > c`. Ovo se može učiniti na sledeći način: prvo se porede vrednosti na koje ukazuju `a` i `b`; ako nije `a > b`, onda one treba da *zamene* vrednosti. U drugom koraku, ako nije `b > c`, `b` i `c` zamenjuju vrednosti. Na ovaj način promenljiva `c` *sigurno* ukazuje na najmanju vrednost. Konačno, opet se proverava da li je `a > b`, pa ako nije promenljive menjaju vrednosti na koje ukazuju. Radi ilustracije, neka `a,b` i `c` ukazuju na `1`, `2` i `3` respektivno (niska `123`). Niska `123` prevodi se u željenu nisku `321` tako što se, u prvom koraku opisanog postupka, početna niska transformiše u `213`. Potom, u drugom koraku, niska postaje `231`, da bi posle

trećeg koraka dobila konačan izgled 321. Sledi program:

Program 3.9 — Kusur.

```
# vraćanje kusura (pohlepni pristup)
iznos = int(input('iznos >= 1000 '))
a = int(input('a > 0 '))
b = int(input('b > 0 '))
c = int(input('c > 0 '))

if iznos >= 1000 and a > 0 and b > 0 and c > 0:

    kusur, broj_novcanica = iznos - 1000, 0
    print('kusur je:', kusur, 'din.')

    # sortira a, b, c tako da je a > b > c
    if a < b:
        a, b = b, a
    if b < c:
        b, c = c, b
    if a < b:
        a, b = b, a

    # pohlepni pristup: prvo najveća pa tako redom
    while kusur > 0:
        if a <= kusur:
            vrati = a
        elif b <= kusur:
            vrati = b
        elif c <= kusur:
            vrati = c
        else:
            break

        print(vrati, end=' ')
        kusur -= vrati
        broj_novcanica += 1

    if kusur > 0:
        print('\njoš', kusur, 'din. ne mogu da vratim!')
    else:
        print('\nkusur vraćen sa', broj_novcanica, 'novčanica')
```

```
39  else:
40      print('Unete vrednosti nemaju smisla!')
```

Ako su unete vrednosti smislene, prvo se obavlja uređivanje promenljivih `a`, `b` i `c` u opadajući poredak (r13-18). Promena vrednosti na koje promenljive ukazuju obavljena je uz pomoć naredbe višestruke dodele vrednosti (r14, 16, 18). Višestruka dodela vrednosti upotrebljena je i u r9 pri izračunavanju kusura i inicijalizacije promenljive koja ukazuje na ukupan broj upotrebljenih novčanica. Potom se pohlepno rešenje realizuje u iterativnom postupku (r21-33). Izbor trenutno najveće moguće novčanice za vraćanje kusura obavlja se pomoću višegrane naredbe `if` (r22-29). Kada problem ne može da se reši sa postojećim ulaznim podacima (r29, r35-36), ispisuje se prigodna poruka. Niska `'\n'`, na početku teksta poruke u (r36, 38), označava *kontrolnu sekvencu*[5] za prelazak u novi red. U daljem tekstu biće pomenute i druge kontrolne sekvence poput `'\t'` koja ubacuje predefinisani broj blanko znakova (tabulacija). Ilustracija rada programa za pojedine primere data je ispod:

```
iznos >= 1000: 1260
a > 0: 50
b > 0: 100
c > 0: 200
kusur je: 260 din.
200 50
još 10 din. ne mogu da vratim!
>>>
==== RESTART: C:/programi/p3_9.py ====
iznos >= 1000: 1260
a > 0: 10
b > 0: 50
c > 0: 100
kusur je: 260 din.
100 100 50 10
kusur vraćen sa 4 novčanica
```

Primer ilustruje dve situacije: u prvoj, kusur se ne može vratiti korišćenjem novčanica zadatih vrednosti, a u drugoj je pronađeno optimalno rešenje. Međutim, treba imati na umu da pohlepni pristup *ne garantuje* globalno optimalno rešenje:

```
iznos >= 1000: 1006
a > 0: 1
b > 0: 3
c > 0: 4
```

[5] Engl. *Escape sequence*.

```
kusur je: 6 din.
4 1 1
kusur vraćen sa 3 novčanica
```

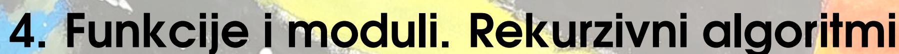
Iako je pronađena niska `411`, kusur od 6 dinara može se vratiti sa manje novčanica: dovoljno je vratiti dva puta po 3 dinara (niska `33`).

> ! Zbog agresivnog pristupa u pretrazi, pohlepna strategija *brzo* pronalazi *neoptimalno* rešenje. Ako je ono u datom kontekstu *zadovoljavajuće*, pristup se može koristiti kada druge metode nisu dovoljno efikasne, ili kada je njihova implementacija izuzetno komplikovana.

ŠTA JE NAUČENO

- Blok predstavlja uzastopni niz naredbi koje pripadaju logičkoj celini programa.
- Program može biti u obliku bloka koji ne sadrži druge blokove, ili obuhvata proizvoljan broj blokova koji i sami mogu da sadrže druge blokove. Sve naredbe bloka moraju biti uvučene sdesna, u odnosu na roditeljski blok, za isti broj mesta.
- Razgranati algoritmi realizuju se upotrebom jednograne, dvograne ili višegrane naredbe `if`.
- Repetitivni algoritmi realizuju se upotrebom petlji `while` i `for`.
- Istinitosne vrednosti logičkih izraza upravljaju grananjima i opredeljuju ostanak u petlji `while`.
- Brojačke petlje, u kojima se odgovarajući blok izvršava predefinisani broj puta, realizuju se uz pomoć naredbe `for`.
- Konstruktor `range()` generiše kako opadajuće, tako i rastuće sekvence celih brojeva koje se koriste u brojačkim petljama.
- Za razliku od petlje `for`, petlja `while` omogućava kako brojačke, tako i petlje koje se izvršavaju sve dok je ispunjen željeni uslov.
- Naredba `break`, u kombinaciji sa naredbom `if`, omogućava prevremeni izlazak iz bloka petlje.
- Iterativni algoritmi rešavaju postavljeni zadatak ponavljanjem određenog postupka, sve dok se ne ispuni željeni stepen ostvarenosti cilja.
- U metodi uzastopnih aproksimacija, koja podrazumeva konvergentnost postupka, stepen ostvarenosti definisan je distancom između približnog i tačnog rešenja. Primer: numerička metoda za izračunavanje korena broja.

- Metoda iterativne konstrukcije oblikuje tačno rešenje tako što se sastavni delovi rešenja formiraju iz iteracije u iteraciju. Primer: crtanje složene slike, red po red.
- Ako se sva potencijalna rešenja nekog problema mogu formalno reprezentovati na pogodan način, te ako postoji funkcija kojom se može utvrditi da li je neko potencijalno rešenje ujedno i pravo, onda se problem može rešiti iterativnom pretragom u prostoru potencijalnih rešenja.
- Algoritam grube sile obavlja iscrpnu pretragu prostora rešenja – prolazi kroz sva moguća potencijalna rešenja. Obično je jednostavan za realizaciju, ali i neupotrebljiv za prostore sa većim brojem potencijalnih rešenja.
- Heuristika pohlepne pretrage, u svakoj iteraciji rešavanja, bira najoptimalniju moguću odluku sa stanovišta trenutne pozicije u pretrazi. Iako lokalno optimalna, heuristika često ne postiže globalno optimalno rešenje. Budući da je veoma brza, koristi se tamo gde drugi pristupi nisu efikasni a neoptimalna rešenja su prihvatljiva.



4. Funkcije i moduli. Rekurzivni algoritmi

U dosadašnjem tekstu izloženi su neophodni jezički elementi pomoću kojih se mogu opisati proizvoljni postupci izračunavanja. Međutim, kako su sistemi i procesi koji se modeluju računarskim programom često veoma složeni, uvedeni mehanizmi grananja i repetitivnosti nisu dovoljni za efikasnu realizaciju kompleksnih modela. Ova glava izučava fundamentalne pojmove računarskih nauka - *dekompoziciju* i *apstrakciju*, pomoću kojih se složeniji problemi lakše rešavaju.

Dekompozicija sistema podrazumeva njegovu podelu na *manje*, *jednostavnije* i *nezavisne* delove koji se lakše modeluju u odnosu na celinu. Ona se, u kontekstu programiranja, može posmatrati na različitim nivoima poput *sistemskog* (klase i objekti), *organizacionog* (biblioteke, moduli) ili *funkcionalnog* nivoa (korisnički definisane funkcije). U ovoj glavi razmatra se *funkcionalna dekompozicija* koja podrazumeva podelu složenijih postupaka izračunavanja na manje procesne celine - *funkcije*.

Apstrakcija predstavlja pojednostavljenje sistema ili procesa na ciljni model koji zadržava samo potrebne osobine za dati kontekst primene. Funkcija predstavlja apstrakciju određenog dela koda koji obavlja neki precizno definisani zadatak. Korisnik funkcije nema potrebu da zna detalje njene implementacije, već samo šta ona radi i kako se poziva.

Pored funkcija razmatraće se i organizacija programa po *modulima* i *paketima* - *organizaciona dekompozicija*. Nešto kasnije, u glavi 10, izučavaće se *sistemska dekompozicija* zasnovana na upotrebi korisnički definisanih klasa.

4.1 Korisničke funkcije

Izlaganje započinje primerom koji objašnjava potrebu uvođenja korisničkih funkcija. Prvo se izlaže postupak definisanja funkcije, a potom će biti reči o povratnim vrednostima, imenovanim i opcionim parametrima i opsegu važenja promenljivih.

4.1.1 Definisanje funkcije

Posmatra se umetnički svet ispunjenih pravougaonika. Slike se iscrtavaju red po red, uz pomoć simbola ispune, poput * ili + (problem 3.5). Treba nacrtati sledeću sliku:

```
         ***
         ***
```

Iznad je iscrtan pravougaonik sa stranicama od dve i tri zvezdice, sa početkom u desetoj koloni. Koristeći mogućnost da se tekstualni objekti mogu spajati (`'a' + 'b'` daje `'ab'`) i multiplikovati (`2 * 'a'` daje `'aa'`), formiran je sledeći program:

```
1 # crta pravougaonik dimenzija 2 x 3
2 # koristi simbol * za ispunu pravougaonika
3 # gornja leva ispuna pravougaonika u 10-toj koloni tekućeg reda
4
5 for red in range(2):
6     print(9 * ' ' + 3 * '*')
```

Neka sada, usled povećanog umetničkog zanosa, treba da se nacrta sledeća slika:

```
         ***
         ***
       |||||||
       |||||||
       |||||||
       |||||||
         +++
         +++
         +++
         +++
```

Prateći logiku prethodnog rešenja, slika se može nacrtati uz pomoć tri sukcesivne petlje `for`. Svaka petlja zadužena je za iscrtavanje jednog od tri pravougaona dela tela (glava, trup i noge):

```
# pravougaona umetnost, crta Glišu
# koristi simbole *, | i + za glavu, trup i noge

# glava
for red in range(2):
    print(9 * ' ' + 3 * '*')

# trup
for red in range(4):
    print(7 * ' ' + 7 * '|')

# noge
for red in range(4):
    print(9 * ' ' + 3 * '+')
```

Kako je slika Gliše naišla na nepodeljeno oduševljenje kritike, autoru je ponuđena samostalna izložba u galeriji Art. Za tu priliku treba iscrtati bar još 30 umetničkih dela koja podsećaju na Glišu. Međutim, to nije nimalo lako jer treba napraviti bar 30 programa nalik na gornji. Pošto je uz veliki napor kolekcija formirana, neko iz uprave galerije moli da se zvezdice *zamene* drugim, prikladnijim simbolom (vlasnik galerije je navijač Partizana). Sada treba proći kroz sve postojeće programe i zameniti neželjene simbole na mestu gde se koriste.

U navedenoj situaciji, kada treba iscrtati *slična* (pravougaona) dela koristeći se *posebnim* programom za svaku sliku, uočavaju se sledeći nedostaci:

- sličan postupak (crtanje pravougaonika različitih dimenzija) *ponavlja se* više puta,
- postupak iscrtavanja složenijih slika postaje sve *kompleksniji*,
- sa porastom kompleksnosti postupka raste i *nepreglednost* odgovarajućeg izvornog koda,
- *otežano* je modifikovanje postojećih ili uvođenje novih mogućnosti u iscrtavanju jer treba intervenisati na *više* različitih mesta.

Navedeni nedostaci uspešno se prevazilaze primenom postupaka dekompozicije i apstrakcije. Slika iz sveta pravougaone umetnosti sastoji se od konačnog broja pravougaonika. Svaki pravougaonik definisan je visinom, širinom, simbolom ispune i pomerajem u odnosu na početak reda. Niz koraka za iscrtavanje pravougaonika navedenih osobina može se smestiti u *korisnički definisanu funkciju*:

```
# linijska grafika

def pravougaonik(a, b, sim, k):
    '''
    Crta pravougaonik dimenzija a x b.
    Koristi simbol sim za ispunu pravougaonika.
    Gornji levi simbol pravougaonika počinje
    od k-te pozicije u tekućem redu'''

    for red in range(a):
        print((k-1) * ' ' + b * sim)

# test primer crta Glišu
pravougaonik(2, 3, '*', 10) # glava
pravougaonik(4, 7, '|', 8)  # trup
pravougaonik(4, 3, '+', 10) # noge
```

Naredbom `def`, u r3, definiše se apstraktni postupak po imenu `pravougaonik` koji iscrtava pravougaonike željenih karakteristika. Iscrtavanje se *parametrizuje* korišćenjem četiri promenljive kojima se definišu visina, širina, simbol ispune i početna pozicija. Deo programa koji poziva funkciju (r14-16) ne zna kako funkcija radi (*implementacija*), već šta radi (*namena*) i kako se poziva (redosled i značenje parametara).

Deo između ključne reči `def` i simbola dvotačke (`:`) predstavlja *potpis funkcije*. Potpis se sastoji od imena funkcije i liste *formalnih parametara* koji se navode između zagrada. Prilikom poziva funkcije (r14-16), umesto formalnih, navode se *stvarni parametri* koji određuju kako treba iscrtati odgovarajuće pravougaonike. Umesto pojma stvarni parametar, često se koristi pojam *argument*. U daljem tekstu, parametri će se odnositi na potpis funkcije, a argumenti na vrednosti koje se navode prilikom njenog poziva! Argumenti mogu biti i izrazi čije se vrednosti izračunavaju pre prosleđivanja u funkciju. Redosled argumenata pri pozivu treba da odgovara, po značenju, redosledu formalnih parametara iz potpisa. Nepoštovanje redosleda može dovesti do neočekivanog rada funkcije, ili do prekida izvršavanja uslovljenog pojavom semantičke greške.

Posle potpisa, sledi blok funkcije (r4-11) koji definiše kako funkcija realizuje svoju namenu. Neobavezni tekst, naveden između *trostrukih* apostrofa ili navodnika (r4-8), zove se još i *docstring*. On predstavlja opis funkcije koji se može dobiti po pokretanju programa kada se, u interaktivnom okruženju, zada komanda `help(ime)`:

```
>>> help(pravougaonik)
Help on function pravougaonik in module __main__:
```

```
pravougaonik(a, b, sim, k)
    Crta pravougaonik dimenzija a x b.
    Koristi simbol sim za ispunu pravougaonika.
    Gornji levi simbol pravougaonika počinje
    od k-te pozicije u tekućem redu.
```

Funkcija `pravougaonik()` može poslužiti za definisanje nove funkcije `gliša()` koja crta Glišu na sledeći način:

```
1  # linijska grafika
2
3  def pravougaonik(a, b, sim, k):
4      # videti prethodni listing
5
6  def gliša():
7      ''' crta Glišu fiksnih dimenzija'''
8      pravougaonik(2, 3, '*', 10) # glava
9      pravougaonik(4, 7, '|', 8)  # telo
10     pravougaonik(4, 3, '+', 10) # noge
11
12 # crta Glišu
13 gliša()
```

Iz potpisa funkcije `gliša()` uočava se odsustvo formalnih parametara, ali bez obzira na ovu činjenicu *uvek* je potrebno navesti zagrade (r6). Korisnički definisana funkcija može pozivati proizvoljan broj drugih korisnički definisanih funkcija, čime se postiže veća fleksibilnost u realizaciji složenijih postupaka.

Uvođenjem koncepta korisnički definisane funkcije prevazilaze se svi problemi navedeni u listi sa početka glave:

- postupak koji se ponavlja (crtanje pravougaonika), apstrahovan je parametrizovanom funkcijom – *izbegnuto* je ponavljanja koda u programima koji iscrtavaju pravougaonik,
- složenije slike iscrtavaju se *kombinovanjem* uvedenih funkcija – *smanjuje* se kompleksnost programa koji crta "izražajnije" slike,
- uvedene funkcije značajno *povećavaju* preglednost odgovarajućeg izvornog koda,
- *olakšano* je modifikovanje postojećih mogućnosti u iscrtavanju – ažuriranje se obavlja samo na potrebnom mestu unutar odgovarajuće funkcije ili, ako je potrebno uvesti novu funkcionalnost, kreira se nova funkcija.

4.1.2 Povratne vrednosti

Funkcija u Pajtonu *vraća* rezultat svoga rada putem objekta određenog tipa koji se formira u funkciji tokom njenog izvršavanja. Rezultat rada funkcije označava se još i kao *povratna vrednost*. Povratna vrednost *prenosi* se iz funkcije u pozivajući program putem naredbe `return`. Naredba `return` *zaustavlja* rad funkcije i prebacuje tok izvršavanja na mesto neposredno posle poziva:

```
1  # konverter: km > milja i obrnuto
2  def konvertuj(d, km):
3      ''' Konvertuje kilometre u milje (km = True),
4          ili milje u kilometre (km = False)'''
5
6      c = 1.609344
7      if km:
8          d_konv = d / c # km > mi
9      else:
10         d_konv = d * c # mi > km
11
12     return d_konv
13
14 # test
15 print('Konverzija: km > mi')
16 for i in range(10):
17     print(10 * i, 'km = ', konvertuj(10 * i, True), 'mi')
```

Funkcija `konvertuj()` transformiše dužinu `d` u kilometrima (`km = True`) u odgovarajući iznos u miljama, odnosno dužinu u miljama (`km = False`) u kilometre. Naredba `return`, iz r12, vraća rezultat (`d_konv`) u pozivajući program - tok se prebacuje u poziv funkcije `print()` gde se vraćena vrednost koristi kao ulazni argument za formiranje ispisa (r17). Ne treba zaboraviti da funkcija zapravo vraća *objektnu referencu* na novoformirani objekat čija vrednost, u ovom slučaju, predstavlja odgovarajuću dužinu u miljama:

```
Konverzija: km > mi
0 km =  0.0 mi
10 km =  6.2137119223733395 mi
20 km =  12.427423844746679 mi
30 km =  18.64113576712002 mi
40 km =  24.854847689493358 mi
50 km =  31.068559611866696 mi
60 km =  37.28227153424004 mi
```

```
70 km =  43.495983456613374 mi
80 km =  49.709695378986716 mi
90 km =  55.92340730136005 mi
```

U primeru funkcije `pravougaonik()`, sa početka ove glave, u izvornom kodu je *izostala* naredba `return`. Kod funkcije koja ne sadrži naredbu `return`, izvršavanje se prekida kada se izvrši *poslednja* naredba u bloku funkcije, a povratna vrednost predstavljena je specijalnim objektom sa vrednošću `None`:[1]

```
>>> a = pravougaonik(2,3,'x', 5)  # nema return pa vraća None!
    xxx
    xxx
>>> print(a)
None
>>> type(a)
<class 'NoneType'>
```

Specijalna vrednost `None`, tipa `NoneType`, dodeljuje se promenljivoj ako se želi naglasiti da ona, u posmatranom tenutku, ne ukazuje ni na šta posebno. Kao što će se kasnije videti, `None` može da nosi informaciju o neuspešnom završetku nekog postupka ili o odsustvu objekta iz neke objektne kolekcije.

Funkcionalna dekompozicija koristi se kada neki složeni postupak treba razbiti na manje, jednostavnije grupe aktivnosti, što povećava čitljivost izvornog koda:

Problem 4.1 — Podniz nula u binarnom zapisu. Odrediti koji prirodan broj iz intervala $[n, m]$ ima najduži podniz nula u svom binarnom zapisu. Na primer, broj 269 (100001101) ima kraći najduži podniz nula od broja 261 (100000101). ■

Realizacija programa podrazumeva nekoliko aktivnosti: učitavanje za n i m, prolazak kroz sve brojeve na intervalu $[n, m]$, prevođenje tekućeg kandidata u binarni zapis i izračunavanje dužine njegovog najdužeg podniza nula, kao i memorisanje onog kandidata koji ima maksimalan podniz nula u svom zapisu. Početni problem će, zbog kompleksnosti, biti dekomponovan na dva manja, jednostavnija potproblema. Sledeći prikaz odnosi se na algoritam u pseudokodu:

Algoritam 4.1 — Podniz nula u binarnom zapisu.

```
Ulaz: n i m
Ako n <= m i n > 0
       max_duzina ← -1
```

[1] Engl. *None* - nijedan.

```
        ZaSvaki x iz [n, m]
               d ← duzina_naj_podniza(x)
               Ako d > max_duzina
                       max_duzina ← d,  rešenje ← x
        Prikaži rešenje, max_duzina
Inače
        Prikaži 'greška'
```

U algoritmu 4.1 uočava se postojanje *nezavisne* celine koja može biti realizovana kao funkcija - `duzina_naj_podniza()`. Ona izračunava dužinu *najdužeg* podniza nula u binarnom zapisu prirodnog broja. Apstrahujući ovu funkciju,[2] algoritam se svodi na pronalaženje maksimalne dužine u nizu dužina, odnosno pamćenje odgovarajućeg prirodnog broja koji sadrži najduži podniz nula u svom binarnom zapisu (videti problem 3.2). Prvo se realizuje funkcija `duzina_naj_podniza()` koja treba da vrati nulu ako binarni zapis broja sadrži sve jedinice.

Prirodan broj x može se zapisati kao sledeća suma:

$$x = \sum_{i=0}^{n} a_i 2^i = \sum_{i=1}^{n} a_i 2^i + a_0, \quad a_i \in \{0,1\} \tag{4.1}$$

Sekvenca $a_n a_{n-1} ... a_1 a_0$ predstavlja *binarni zapis* broja x. Na primer, broj 17 može se predstaviti kao $1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$, pa je njegov binarni zapis jednak 10001. Da bi se detektovao najduži podniz nula, potrebno je izdvojiti sve koeficijente a_i iz sume (4.1). Na osnovu date jednakosti, proizilazi da je ostatak pri celobrojnom deljenju broja x sa dva jednak koeficijentu a_0, a rezultat celobrojnog deljenja svodi se na sumu $\sum_{i=1}^{n} a_i 2^{i-1}$. Očigledno, da bi se dobio a_1, potrebno je ponoviti postupak celobrojnog deljenja i zabeležiti novi ostatak. Postupak se prekida kada se kao rezultat celobrojnog deljenja dobije nula, a ostatak tada predstavlja koeficijent a_n:

```
1  def duzina_naj_podniza(x):
2  
3      '''Vraća dužinu najdužeg podniza nula u binarnom zapisu'''
4      d = 0                     # tekuća dužina podniza nula
5      max_d =  0                # trenutna maks. dužina podniza nula
6      prva_1 = True             # da li će sledeća jedinica biti prva
7                                # posle podniza nula
8      while True:
9          # izdvajanje bin. cifre najmanje težine
10         bin_cifra = x % 2
```

[2] Apstrahovanje se vrši zanemarivanjem implementacije, a uzimanjem u obzir namene i potpisa funkcije.

```
11          # ako je nula uvećaj dužinu tekućeg niza
12          if bin_cifra == 0:
13              d += 1
14              prva_1 = True
15          # ako je prva jedinica posle nule,
16          # da li je podniz nula dosad najduži
17          elif prva_1:
18              prva_1 = False
19              if d > max_d:
20                  max_d = d
21              d = 0
22          # priprema za sledeću cifru
23          x //= 2   # x = x // 2
24          if x == 0:
25              break
26
27      return max_d
```

U svrhu pronalaženja najdužeg podniza nula, definisane su promenljive `d` - dužina aktuelnog podniza, `max_d` - trenutna maksimalna dužina i `prva_1` - promenljiva koja ukazuje da li će sledeća izdvojena jedinica biti prva posle nule (r4-6). Kada se u zapisu pojavi nula, ažurira se dužina tekućeg podniza i priroda sledeće jedinice (r12-14). U suprotnom, ako se u zapisu naiđe na jedinicu, treba završiti sa obradom prethodnog podniza, ali samo ako je ta jedinica nastupila neposredno posle nule (r17-21). Bitnu ulogu u postupku ima logička promenljiva `prva_1` koja nosi informaciju o prirodi svake izdvojene jedinice. Naredba `return`, iz r27, vraća dužinu maksimalnog podniza nula.

Već je istaknuto da funkcije vraćaju reference na objekte koji su proizašli iz njihovog rada. Kako se objektima može pristupati isključivo putem reference, u daljem tekstu će se navoditi da funkcije vraćaju željene objekte (ili vrednosti).

Kako se funkcije *izračunavaju* na odgovarajuće vrednosti, moguće je koristiti ih u okviru drugih, pogodnih izraza. Na primer, izraz `2 * duzina_naj_podniza(261) + 1`, ima vrednost `11`. Sledi direktna implementacija algoritma 4.1 koji pronalazi prirodan broj iz zadatog intervala, a sa maksimalno dugačkim podnizom nula:

Program 4.1 — Podniz nula u binarnom zapisu.

```
1  def duzina_naj_podniza(x):
2      # videti prethodni listing
3
```

```
n = int(input('prirodan broj n: '))
m = int(input('prirodan broj m >= n: '))

if n <= m and n > 0:
    max_duzina = -1
    for x in range(n, m+1):
        d = duzina_naj_podniza(x)
        if d > max_duzina:
            max_duzina = d
            rešenje = x
    print(rešenje, 'ima podniz nula dužine', max_duzina)
else:
    print('greška u unosu!')
```

Rad programa, u intervalu $[1,33]$, prikazan je ispod:

```
prirodan broj n: 1
prirodan broj m >= n: 33
32 ima podniz nula dužine 5
```

Povratne vrednosti različitih tipova

Iz dosadašnjeg izlaganja poznato je da se izvršavanje funkcije prekida nailaskom na naredbu `return`, ili, ako ona nije prisutna, kada programski tok naiđe na kraj funkcijskog bloka. Međutim, izvršavanje funkcije može da se prekine na *više* mesta, pri čemu se u pozivajući program može vratiti više *različitih* tipova vrednosti:

```
# ilustruje različite izlaze i povratne vrednosti
def sve_po_malo(x):

    if x == 5:
        return 5
    elif x < 0:
        return 'negativan broj'
    elif x == 0:
        return  # ovde se vraća None

    # dolaskom u ovu tačku, vraća se None
```

```
>>> sve_po_malo(-1)
'negativan broj'
>>> sve_po_malo(5)
5
>>> sve_po_malo(0)
>>>
>>> sve_po_malo(0) == None
True
>>> sve_po_malo(36) == None
True
```

Definisanje funkcije koja vraća reference na različite tipove objekata *nije preporučljivo*, pošto takva praksa otežava razumevanje izvornog koda i potencijalni je izvor grešaka. Ipak, pored svog osnovnog tipa, funkcije često vraćaju i `None`, kako bi ukazale na neadekvatne ulazne parametre ili na nemogućnost izvršavanja zadatka.

4.1.3 Imenovani i opcioni parametri

Posmatra se sledeća funkcija i njeni mogući pozivi:

```
1  # ilustruje imenovane i opcione parametre
2  def reci_zdravo(ime, broj_puta, jezik='sr'):
3
4      zdravo = 'Zdravo' # podrazumevani jezik
5      if jezik == 'en':
6          zdravo = 'Hello'
7      elif jezik == 'it':
8          zdravo = 'Ciao'
9
10     # ispis pozdrava
11     for i in range(broj_puta):
12         print(zdravo, ime)
```

```
>>> reci_zdravo(broj_puta=2, jezik='it', ime='Miloš')
Ciao Miloš
Ciao Miloš
>>> reci_zdravo(koliko=2, ime='Miloš')
Zdravo Miloš
Zdravo Miloš
>>> reci_zdravo('Miloš', 1)
Zdravo Miloš
```

Argumenti funkcije se, pri pozivu, mogu *imenovati* po formalnim parametrima, što je ilustrovano prvim pozivom funkcije `reci_zdravo()`. *Imenovani parametri* omogućavaju da se, pri njihovom navođenju, ne mora poštovati redosled definisan u potpisu funkcije, što je učinjeno u prvom pozivu.

Ako se, pri definisanju funkcije, uz formalni parametar pojavi i znak jednakosti (r2), onda taj parametar postaje *opcioni*. Opcioni parametri se *ne moraju* navoditi pri pozivu funkcije, kao što je učinjeno u drugom pozivu iz primera. Tada oni dobijaju vrednosti koje su definisane u potpisu funkcije. Treba obratiti pažnju da se opcioni parametri iz potpisa obavezno navode *posle* standardnih parametara. U suprotnom, interpreter će prijaviti odgovarajuću grešku. Pri navođenju izraza za vrednost opcionog parametra, treba imati u vidu da se vrednost opcije izračunava pri *definisanju potpisa*, a *ne* u trenutku poziva. Poslednji poziv ne navodi imena pa se mora poštovati redosled iz potpisa funkcije, uz korišćenje podrazumevanog jezika ispisa.

Opcioni parametri često se koriste u praksi, a podrazumevane vrednosti pažljivo su izabrane tako da pokrivaju najčešće situacije pri pozivu funkcije.

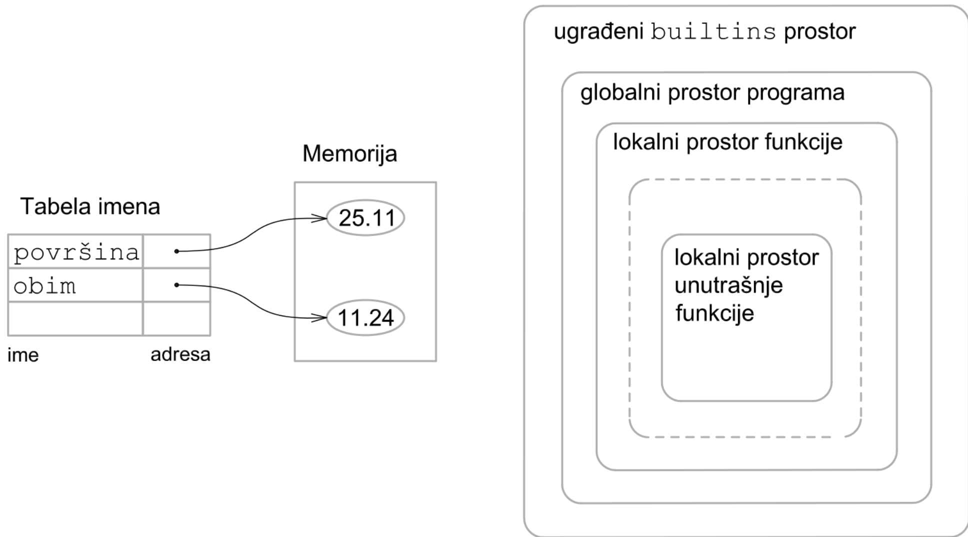
4.1.4 Imenski prostori. Opseg važenja promenljivih

Promenljive predstavljaju imena za objekte. Ime se najčešće vezuje za objekat kroz naredbu dodele vrednosti. Moguća su i druga vezivanja, poput navođenja brojača u petlji `for` ili formalnih parametara u potpisu funkcije. Skup imena koja su definisana u programu naziva se *imenski prostor*[3] programa. Imenski prostor realizuje se *tabelom imena* u kojoj se, svakom imenu iz prostora, pridružuje adresa imenovanog objekta u memoriji (slika 4.1, levo).

U toku izvršavanja programa, u memoriji istovremeno postoje *bar dva* međusobno odvojena imenska prostora. To omogućava da se, u okviru različitih delova koda, različitim objektima daju ista imena. Po pokretanju, interpreter formira imenski prostor *ugrađenog* `builtins` modula koji sadrži predefinisana imena poput `input` ili `print`. Kada se program pokrene, kreira se *globalni* imenski prostor, a pri pozivu svake funkcije, dodatni *lokalni* imenski prostor (slika 4.1, desno). Kasnije će biti reči i o imenskim prostorima modula i klase. Veze između pojedinih imena i objekata, sačuvane u odgovarajućoj tabeli imena, imaju svoj *životni vek*. Tabela imena *ugrađenog* `builtins` modula *postoji* sve dok je aktivna interpreterska sesija. Imenski prostor programa *opstaje* sve dok se program ne završi, a funkcije dok traje njeno izvršavanje.

Opseg važenja promenljive odnosi se na *deo* izvornog koda u kome se promenljiva može koristiti uz navođenje njenog imena. Pitanje životnog veka, odnosno opsega važenja, neraskidivo je povezano sa trenutno formiranim imenskim prostorima. Interpreter primenjuje jednostavno pravilo za izbor tabele u kojoj će se potražiti navedeno ime, odnosno objekat na koji ono ukazuje. Ime se prvo traži u tabeli lokalnog prostora (ako se na promenljivu referiše iz funkcije), pa ako se tu ne pronađe, ide se za po jedan

[3] Engl. *Namespace*.



Slika 4.1: Imenski prostori i tabela imena: imenima promenljivih pridružuju se adrese koje označavaju objekte u memoriji (levo); hijerarhija formiranih imenskih prostora u nekom trenutku izvršavanja. Svakom prostoru pridružena je po jedna tabela imena (desno).

korak naviše u hijerarhiji, sve dok se ne stigne do ugrađenog `builtins` prostora (slika 4.1, desno). Ako se ime ne pronađe ni u jednoj tabeli, interpreter prijavljuje grešku.

Opseg važenja promenljve definisane u nekoj funkciji (*lokalna promenljiva*), odnosi se na tu funkciju i sve eventualne unutrašnje funkcije. Promenljive definisane u programu (*globalne promenljive*) mogu se koristiti kako u programu, tako i u svim pozivajućim funkcijama. Sledi primer koji ilustruje prethodnu diskusiju:

```
def f():
    b, c = 3, 10  # lokalni imenski prostor funkcije
    print('u f(), a =', a, 'b =', b)

a, b = 1, 1  # globalni imenski prostor programa
f()
print('program, a =', a, 'b =', b)
print('program, c =', c) # greška!
```

Kada se gornji program pokrene, dobija se:

```
u f(), a = 1 b = 3
program, a = 1 b = 1
```

```
Traceback (most recent call last):
  File "C:/programi/p4_1scope.py", line 9, in <module>
    print('program, c=', c)
NameError: name 'c' is not defined
```

Globalne promenljive `a` i `b` definisane su u imenskom prostoru programa (r5). U funkciji `f()` definisane su lokalne promenljive `b` i `c` (r2). Prilikom poziva funkcije (r6) ispisuju se vrednosti za `a` i `b` (r3). Promenljiva `a` ne nalazi se u imenskom prostoru funkcije (nije lokalna promenjiva) pa se interpreter "penje" jedan korak naviše u hijerarhiji, pronalazeći vrednost za `a` preko globalne tabele imena (=1). Slično, iako je ime `b` definisano u oba imenska prostora, promenljiva se tretira kao lokalna pa se vrednost pronalazi u imenskom prostoru funkcije (=3).

Posle izvršenja funkcije `f()` njen imenski prostor prestaje da postoji pa se u r7 ispisuju vrednosti na koje ukazuju globalne promenljive `a` i `b`. Prilikom pokušaja ispisa promenljive `c` (r8), interpreter prijavljuje grešku jer navedeno ime ne postoji ni u tabeli globalnog imenskog prostora, ni u tabeli `builtins` modula sa predefinisanim imenima (`c` očigledno nije predefinisano ime ni za jedan objekat).

Prethodni primer pokazuje da se globalno ime (`b` iz r5), ukoliko se u funkciji nađe sa *leve* strane naredbe dodele vrednosti (r2), kopira u lokalni imenski prostor pa sada ukazuje na novi objekat koji reprezentuje vrednost izračunatog izraza sa desne strane dodele. Kada se funkcija izvrši i kontrola toka vrati u pozivajući program, ime promenljive iz globalnog prostora ponovo postaje aktuelno, a promenljiva ukazuje na stari objekat.

Tipična početnička greška, koja nastaje zbog nerazumevanja odnosa globalne i lokalne promenljive, ilustrovana je u sledećem primeru:

```
1  a = 1   # globalna promenljiva
2  
3  def f():
4      a = a + 1   # greška
5      print(a)
6  
7  f()
```

```
Traceback (most recent call last):
  File "C:/programi/p4_1greska.py", line 7, in <module>
    f()
  File "C:/programi/p4_1greska.py", line 4, in f
    a = a + 1  # greška
```

```
UnboundLocalError: local variable 'a' referenced before assignment
```

Greška nastaj u r4, u toku izvršavanja `a = a + 1`. Pošto se globalno ime `a` pojavljuje na levoj strani dodele vrednosti, interpreter kopira ime u lokalni imenski prostor. Međutim, kako to ime još ne ukazuje ni na jedan lokalni objekat, prilikom pokušaja uvećavanja vrednosti za jedan, interpreter prijavljuje grešku. Da je, umesto `a = a + 1`, stajalo `b = a + 1`, onda bi sve bilo u redu. Tada bi se `a` tretirala kao globalna promenljiva, a novonastala lokalna promenljiva `b` ukazivala bi na uvećanu vrednost.

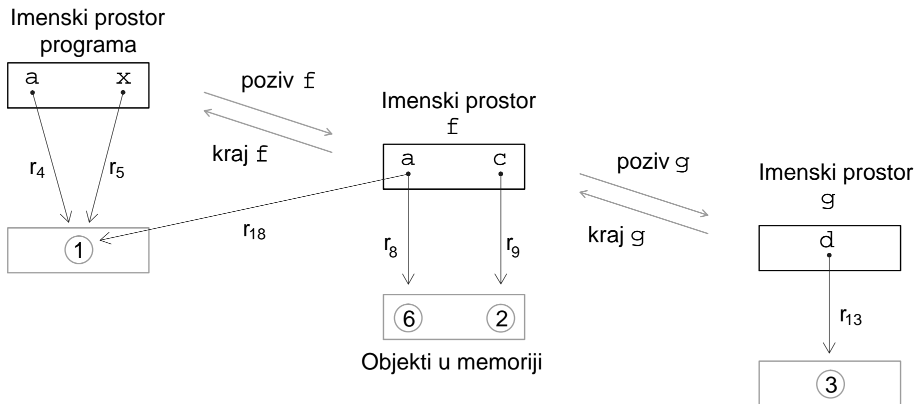
Parametri - opseg važenja i mehanizam prenošenja

Formalni parametri imaju opseg važenja isključivo unutar bloka funkcije (tretiraju se kao lokalne promenljive), a njihov životni vek poklapa se sa njenim trajanjem. Stvarni parametri prenose se u funkciju tako što se, u njen imenski prostor, kopiraju vrednosti objektnih referenci.[4] Zapravo, objekat dobija *još jedno* ime preko koga mu se može pristupati iz funkcije, ali ako se formalni parametar nađe sa leve strane naredbe dodele vrednosti, onda se ta veza *raskida* i referenca dobija lokalni karakter. Ako parametar ukazuje na objekat *promenljivog* tipa, onda se njegova vrednost *može* promeniti u funkciji, o čemu će biti reči u glavi 5 kada se budu izučavali promenljivi tipovi. Sledi ilustrativni primer:

```
1  ## Oblast važenja promenljivih
2
3  # globalne promenljive programa
4  a = 1
5  x = 1
6
7  def f(a):
8      a += 5      # parametar a sakriva globalnu promenljivu
9      c = x + 1   # c je lokalna promenljiva za f
10     print('f: (a, c)', a, c)
11
12     def g():
13         d = c + x   # d je lokalna promenljiva za g
14         print('g: d', d)
15
16     g()
17
18 f(a)
19 print('program: (a,x)', a, x)
```

[4] Engl. *Copy by object reference value*.

Globalne promenljive a i x definišu se u imenskom prostoru programa (slika 4.2). Prilikom poziva funkcije f(), memorijska adresa pridružena promenljivoj a se *kopira* iz imenskog prostora programa u tek kreirani imenski prostor funkcije (r18). Parametar a, koji po kopiranju ima lokalni karakter u funkciji, inicijalno ukazuje na isti objekat kao i globalna promenljiva a (=1). Potom se pokušava sa uvećavanjem vrednosti objekta za pet (r8). Pošto je celobrojni tip nepromenljiv, kreira se novi objekat sa vrednošću šest, a njegova memorijska adresa smešta se u tabelu imenskog prostora funkcije. Sada parametar a ukazuje na novokreirani objekat (slika 4.2).



Slika 4.2: Prenos parametara i opseg važenja promenljive: prilikom poziva funkcije kreira se novi imenski prostor. Oznake r_x označavaju redove u programu u kojima se obavljaju odgovarajuća imenovanja.

U r9 definiše se lokalna promenljiva c koja po izračunavanju ukazuje na vrednost dva. Prilikom njenog izračunavanja bilo je neophodno pristupiti vrednosti na koju ukazuje globalna promenljiva x. U ovom slučaju interpreter je pronašao vrednost na koju ukazuje promenljiva x tako što je prvo potražio ime u tabeli imenskog prostora funkcije f(). Pošto u tom prostoru ime nije pronađeno, interpreter pravi jedan korak unazad u putanji funkcijskih poziva i pronalazi traženo ime u imenskom prostoru programa (slika 4.2). Po pozivu funkcije g() u r16, izračunava se vrednost za promenljivu d. Kako se imena c i x ne nalaze u imenskom prostoru funkcije, interpreter pronalazi njihove vrednosti u odgovarajućim prostorima sa putanje poziva. Program daje sledeći izlaz:

```
f: (a, c) 6 2
g: d 3
program: (a,x) 1 1
```

Promena vrednosti globalne promenljive iz funkcije - naredba global

Globalnim promenljivim može se pristupati iz svih funkcija u programu, ali se njima, kao što je već pokazano, *ne mogu* dodeljivati nove vrednosti. Ako se želi da funkcija

tretira promenljivu globalnom i *posle* dodele vrednosti, onda se ona mora *deklarisati* naredbom `global` pre eventualne dodele:

```
def g():
    global a
    a = a + 1  # a ostaje gloobalna i posle dodele
    print('u g(), a =', a)

a = 1  # globalni imenski prostor programa
g()
print('program posle g(), a =', a)
```

```
u g(), a = 2
program posle g(), a = 2
```

Parametri funkcije *ne mogu* se deklarisati naredbom `global` – interpreter bi tada prijavio grešku.

Sporedni efekat funkcije

Funkcija proizvodi *sporedni efekat*[5] ako se njena projektovana uloga ostvaruje *mimo* povratnog mehanizma naredbe `return`. Sporedni efekat može se desiti u bar dva slučaja: promena vrednosti promenljivih definisanih *van* opsega funkcije i emitovanje informacija u spoljni svet. U prethodnom primeru, menjanjem vrednosti globalne promenljive `a`, funkcija `g()` ostvaruje sporedni efekat. Ovaj slučaj često se smatra *lošom* praksom jer funkcija menja stanje programa mimo svog povratnog mehanizma. Tada je izvorni kod manje čitljiv pa se potencijalne logičke greške teže otklanjaju.

U slučaju kada funkcija šalje informaciju ka eksternim sistemima (ekran, datoteka, štampač, baza podataka, mreža), ostvaruje se *opravdani* sporedni efekat. U takvim situacijama njen rad se ne svodi *samo* na izračnavanje pa se ni rezultat ne može vratiti isključivo putem naredbe `return`. Primer za ovakvo ponšanje je funkcija `pravougaonik()` iz sveta pravougaone umetnosti sa početka glave.

Postupci koji proizvode tačan, efikasan, ali i kod koji je dobro organizovan i čitljiv, smatraju se dobrom praksom. Organizacija i čitljivost značajno olakšavaju otklanjanje grešaka u velikim programskim sistemima. Ovo je naročito bitno ako se ima na umu da se, u toku životnog ciklusa softvera, često menjaju programeri koji ga održavaju.

[5] Engl. *Side effect*.

4.2 Organizacija izvornog koda

Kompleksan programski sistem organizuje se po logičkim celinama, uz poželjno korišćenje već postojećih komponenti. Takvi *modularni* sistemi lakše se nadograđuju i održavaju. Funkcije, kao nezavisne celine, omogućavaju efikasno sprovođenje procesa dekompozicije. Međutim, samo pomoću njih nije moguće jednostavno kreiranje složenijeg softvera pa se zbog toga uvode više hijerarhijske podele poput *modula* i *paketa*.

4.2.1 Moduli

Izvorni kod do sada razmatranih programa bio je smešten u *samo jednu* datoteku. Sada treba pojasniti kako se funkcija, definisana u datoteci *f.py*, može pozivati u programu smeštenom u datoteci *p.py*. U tu svrhu u Pajton se uvodi koncept *modula*. Modul predstavlja datoteku sa ekstenzijom **.py* koja sadrži *proizvoljan* izvorni kod. On najčešće sadrži *definicije funkcija*, ali može da sadrži i *druge* naredbe, poput onih za definisanje globalnih promenljivih. Sledeći primer ilustruje koncept modula:

Problem 4.2 — Površine. Realizovati grupu funkcija koje računaju površine paralelograma, trapeza i kruga. Testirati funkcije u posebnom programu u kome se unosi tip geometrijske slike i neophodni elementi za računanje površine. ■

Grupa funkcija realizovana je kao modul `površine`, smešten u datoteku *površine.py*:

Program 4.2 — Površine.

```
1  # modul površine (datoteka površine.py)
2  paralelogram = 1 #
3  trapez = 2       # globalne promenljive
4  krug = 3         # modula povrsine
5
6  def p_paralelogram(a, b):
7      '''računa površinu paralelograma sa stranicama a i b'''
8      return a * b
9
10 def p_trapez(a, b, h):
11     '''računa površinu trapeza sa osnovama a, b i visinom h'''
12     return h * (a + b)/2
13
14 def p_krug(r):
15     '''računa površinu kruga poluprečnika r'''
16     return 3.14 * r**2
```

Program za testiranje modula `površine` smešten je u datoteku glavnog programa:

```
# unosi tip slike (ceo broj), pa elemente za računanje površine
# unos se prekida kada se unese broj različit od:
# 1 (paralelogram), 2 (trapez), 3 krug
import površine # čini funkcije i promenljive iz povrsine.py
                # vidljivim u programu
while True:
    slika = int(input('1 paralelogram, 2 trapez, 3 krug  '))
    if slika == površine.paralelogram:
        a = float(input('a '))
        b = float(input('b '))
        print('P=', površine.p_paralelogram(a, b))
    elif slika == površine.trapez:
        a = float(input('a '))
        b = float(input('b '))
        h = float(input('h '))
        print('P=', površine.p_trapez(a, b, h))
    elif slika == površine.krug:
        r = float(input('r '))
        print('P=', površine.p_krug(r))
    else:
        break
```

Naredba `import` *uvodi* modul u program (r4).[6] Uvođenje obuhvata učitavanje naredbi modula u operativnu memoriju i kreiranje novog imenskog prostora za modul. Funkcije i globalne promenljive iz ovog prostora postaju *vidljive* u programu ako im se pristupa korišćenjem *notacije sa tačkom* oblika `<ime_modula>.<ime_u_modulu>` (r8, 11, 12, 16, 17, 19). Notacija sa tačkom sprečava moguće *preklapanje* imena definisanih u programu i u uvedenom modulu pa je dozvoljeno koristiti dva ista imena u obe celine. Ako se želi izbeći korišćenje notacije sa tačkom, može se koristiti i sledeća forma: `from <modul> import <ime_1>, <ime_2>, ...` Na ovaj način postaju *direktno* vidljiva *samo* navedena imena, s tim da, u slučaju preklapanja, imena definisana u programu *sakrivaju* imena iz modula. Naredba `import` dozvoljava i upotrebu opcije `as` iza koje sledi kraće, alternativno ime za modul. Na primer, `import površine as p` omogućava da se funkcija može pozvati sa `p.p_paralelogram(a, b)`, umesto `površine.p_paralelogram(a, b)`.

Programi koji su *u celosti* smešteni u jednoj datoteci nazivaju se još i *skripta*. Skripta se, van IDLE okruženja, pokreću u komandnom režimu operativnog sistema zadavanjem

[6] Engl. *Importing*.

komande koja poziva interpreter, uz navođenje njihove pune putanje. Kako su svaka skripta ujedno i modul, ona se, unutar interaktivne sesije u IDLE-u, mogu pokrenuti i sa `import <ime_skripta>`.[7] Prilikom uvođenja u drugi modul, inicijalno se izvršavaju sve naredbe iz bloka uvedenog modula. Da bi se povećala čitljivost glavnog programa, na početku modula `površine` definisane su promenljive koje ukazuju na tip slike čija se površina traži (r2-4). Datoteka sa izvornim kodom nazivaće se modulom *samo* ako nije namenjena za samostalno pokretanje, već služi kao *biblioteka* funkcija za realizaciju složenijih programa.

> ! Naredbe bloka modula izvršavaju se, zbog uštede resursa, *samo* pri prvom uvođenju u okviru jedne interpreterske sesije. Ukoliko se modul modifikuje, promene *neće* biti vidljive sve dok se interpreterska sesija ne restartuje (<ctrl> + <F6>) i modul ponovo ne učita u memoriju!

U daljem razmatranju koristiće se funkcije iz različitih dostupnih modula u okviru standardne instalacije Pajtona. Na primer, upotrebljavaće se modul koji sadrži često korišćene matematičke funkcije – `math`. Sledi primer korišćenja modula `math` i funkcije `dir()` koja prikazuje imena definisana u traženom modulu:

```
>>> import math as m
>>> m.sin(m.pi)
1.2246467991473532e-16
>>> m.sqrt(16)
4.0
>>> dir(m)  # lista imena definisana u modulu
['__doc__', '__loader__', '__name__', '__package__', '__spec__',
'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil',
'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp',
'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum',
'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf',
'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf',
'nan', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan',
'tanh', 'trunc']
```

Uočiti da sinus ugla od 180 stepeni iznosi 1.2246467991473532e-16 (eksponencijalni zapis, e-16 = 10^{-16}). Ovo se dešava jer se najveći broj funkcija izračunava *približno*, upotrebom konačnog broja osnovnih aritmetičkih operacija. Osim toga, već je diskutovano da se realni brojevi, u opštem slučaju, *ne mogu* tačno predstaviti u brojnom sistemu sa osnovom dva.

[7] Važi samo ako se direktorijum, koji je sadrži, nalazi u skupu putanja koje IDLE pretražuje kada učitava module u operativnu memoriju.

4.2.2 Paketi

Paketi organizuju veći broja modula u jedinstvenu imensku hijerarhiju. Pretpostaviti da na disku postoji direktorijum `slike` koji obuhvata datoteke (module) `gif.py`, `jpeg.py` i `png.py`. Svaki od navedenih modula sadrži funkcije koje obrađuju slike istoimenog formata. Funkcionalnosti iz paketa `slike` mogu se, po uvođenju sa `import slike`, koristiti u proizvoljnom programu P. Na primer, iz modula `jpeg` postaje dostupna funkcija `skaliraj()`: `slike.jpeg.skaliraj(0.5)` – umanjuje sliku za 50% po širini i dužini. Uvođenjem paketa kao organizacione celine omogućena je upotreba različitih biblioteka koje sadrže module sa *istim* imenom. Program P može dodatno koristi i funkcionalnosti iz istoimenog modula `jpeg.py`, ali u okviru paketa `prepoznavanje`: `prepoznavanje.jpeg.lice()` – prepoznaje da li se na slici u jpeg formatu nalazi lice osobe.

Da bi se moduli, smešteni u nekom direktorijumu *d*, tretirali kao jedinstveni paket, potrebno je da *d* sadrži specijalno imenovanu datoteku `__init__.py`. U okviru ove datoteke, koja se izvršava prilikom prvog uvođenja paketa, nalazi se izvorni kod kojim se paket inicijalizuje – na primer postavljanje globalnih promenljivih paketa na početne vrednosti. Ukoliko inicijalizacija nije neophodna, datoteka može biti i prazna, ali *mora* da postoji!

Složenije biblioteke mogu biti realizovane kao paketi sa većim brojem potpaketa. Potpaketima odgovara po jedan direktorijum sa `__init__.py` datotekom. Poziv oblika `paket.potpaket.modul.funkcija()`, moguć je pošto se paket uvede u program.

4.2.3 Funkcije i moduli kao objekti

Pajton tretira sve funkcije i module kao objekte tipa `function`, odnosno `module`. Na ovaj način moguće je imenovati ih različitim promenljivim uz pomoć naredbe dodele vrednosti ili prosleđivati kao stvarne parametre u druge funkcije. U sledećem primeru ilustrovana je njihova objektna priroda:

```
>>> import math
>>> print(id(math), type(math)) # identitet i tip objekta
1967655099432 <class 'module'>
>>> sin = math.sin
>>> sin(0)
0.0
>>> print(id(sin), type(sin))
1967655155752 <class 'builtin_function_or_method'>
>>>
```

Po uvođenju modula `math` u imenski prostor interaktivne sesije, prikazani su identitet (memorijska adresa) i tip objekta koji reprezentuje modul. Funkcija `sin()`, iz modula `math`, *dodeljuje* se promenljivoj `sin` iz imenskog prostora interaktivne sesije.

Dodela je moguća jer se funkcije, kao i moduli, tretiraju kao objekti. Sada se funkcija može pozivati preko novog imena bez navođenja notacije sa tačkom. Objektna priroda funkcije potvrđena je ispisivanjem njenog identiteta i tipa.

Ponekad se algoritam za rešavanje određene klase problema ne razlikuje za pojedine probleme iz klase, sem u jednom *nezavisnom* i *varijabilnom* delu. Ako nepromenljivi deo algoritma "vidi" varijabilnu celinu uvek na isti način, kroz unapred poznato ponašanje, onda je moguće da se celina, pored ostalih ulaznih veličina, *unese* u algoritam u obliku funkcije. Koncept se ilustruje sledećim primerom:

Problem 4.3 — Tablica funkcije. Napisati funkciju koja ispisuje sve vrednosti za proizvoljnu funkciju $f(x)$ na intervalu $[a,b]$ sa korakom $delta$. Testirati funkciju za $y = cos(x)$ i $y = e^x$. ■

Ispisivanje tablice vrednosti za *proizvoljnu* funkciju, na intervalu $[a,b]$, podrazumeva *fiksni* iterativni postupak u kome se, sa zadatim korakom, vrši odabir vrednosti nezavisne promenljive. U postupku ispisivanja *varijabilna* je jedino funkcija za koju se tablica formira. Pošto se funkcije tretiraju kao objekti, moguće je *preneti* ih u druge funkcije kao *ulazne* parametre:

Program 4.3 — Tablica funkcije.

```
def tablica_vrednosti(a, b, delta, f):
    ''' Ispisuje tablicu vrednosti f-je f(x)
        na intervalu [a, b], sa korakom delta.
        Podrazumeva se da je f(x) definisana na intervalu'''

    if a < b and delta > 0:
        x = a
        while x <= b:
            print(x, f(x))
            x += delta
    else:
        print('mora biti a < b i delta > 0')

import math
print('cos(x)')
tablica_vrednosti(0, 2 * math.pi, math.pi / 6, math.cos)
print('\nexp(x)') # \n specijalni karakter za prelaz u novi red
tablica_vrednosti(0, 5, 0.5, math.exp)
```

Parametar `f`, iz potpisa funkcije `tablica_vrednosti()` (r1), predstavlja prenetu funkciju čije se vrednosti ispisuju u r9. Po uvođenju modula `math` (r14), prvo se ispisuje tablica za $cos(x)$ (r16), pa potom i za e^x (r18). Izabrana funkcija prenosi se jednostavnim navođenjem imena kao stvarnog parametra. U primeru se koristi notacija sa tačkom, pošto su funkcije koje se prenose definisane unutar imenskog prostora uvedenog modula.

```
cos(x)
0 1.0
0.5235987755982988 0.8660254037844387
1.0471975511965976 0.5000000000000001
1.5707963267948966 6.123233995736766e-17
2.0943951023931953 -0.4999999999999998
2.617993877991494 -0.8660254037844385
3.1415926535897927 -1.0
3.6651914291880914 -0.866025403784439
4.1887902047863905 -0.5000000000000004
4.71238898038469 -1.8369701987210297e-16
5.235987755982989 0.5000000000000001
5.759586531581288 0.8660254037844388

exp(x)
0 1.0
0.5 1.6487212707001282
1.0 2.718281828459045
1.5 4.4816890703380645
2.0 7.38905609893065
2.5 12.182493960703473
3.0 20.085536923187668
3.5 33.11545195869231
4.0 54.598150033144236
4.5 90.01713130052181
5.0 148.4131591025766
```

4.3 Rekurzivni algoritmi

U praksi se često definiše značenje različitih pojmova koji se odnose na materijalni ili apstraktni svet. Ako se pojam koji se objašnjava i sam *koristi* kao sastavni deo objašnjenja, onda se radi o *rekurzivnoj* definiciji. Na primer, validno ime za promenljivu u Pajtonu može se definisati na sledeći način:

1. slovo ili donja crta (_) je validno ime za promenljivu.
2. validno ime iza koga sledi slovo, broj ili donja crta, predstavlja validno ime.

U definiciji validnog imena promenljive, stav 2, primećuje se rekurzivna priroda formulacije: ako je niz simbola validno ime, onda je to i isti niz proširen za slovo, broj

ili donju crtu. Uz pomoć stava 2 mogu se generisati proizvoljno dugačke sekvence, ali i dalje nije jasno čime treba da započne validno ime. Prvi stav rešava problem početnog simbola – svako slovo ili donja crta predstavlja validno ime za promenljivu. Sada se, primenom drugog stava, može formirati bilo koje validno ime. Prvi stav naziva se još i *bazni slučaj*. Treba primetiti da bi, bez baznog slučaja, definicija postala *nepotpuna* pa samim tim i neupotrebljiva.

4.3.1 Rekurzija na delu

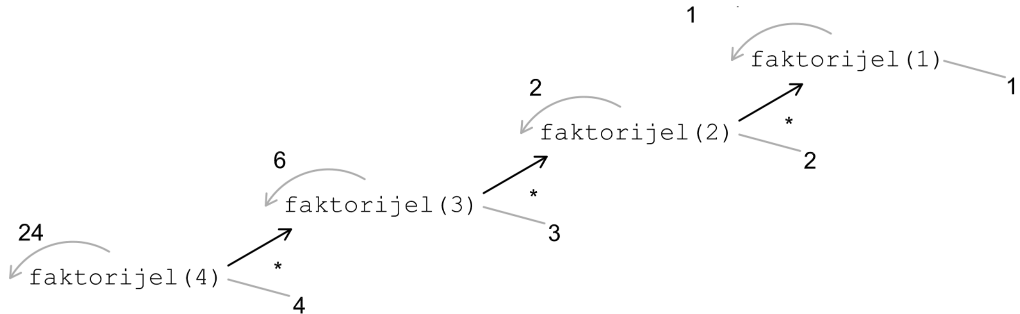
Rekurzivni način rešavanja problema odnosi se na specijalnu klasu algoritama iz porodice *podeli pa vladaj* (glava 1.3.1). Neka se složenost problema može okarakterisati prirodnim brojem n, pri čemu složenijem problemu odgovara veće n. Ako rešenje problema složenosti n zavisi od rešenja *istog problema* složenosti k, pri čemu je $n > k$, te ako je poznato bazno rešenje istog problema (za složenost reda 1), onda se u rešavanju može primeniti *rekurzivni algoritam*. Sledi ilustrativni primer:

Problem 4.4 — Drugovi Faktorijel i Fibonači. Napisati modul sa rekurzivnim funkcijama koje, za zadati prirodni broj n, izračunavaju $n!$ i Fibonačijev broj $F(n)$. Funkcije testirati u posebnom programu za prvih 10 prirodnih brojeva. ■

Program 4.4 — Drugovi Faktorijel i Fibonači.

```
1  def faktorijel(n):
2      '''Izračunava n! u rekurzivnom postupku, n>=1'''
3  
4      if n == 1:
5          return 1  # bazni slučaj
6      else:
7          return n * faktorijel(n - 1) # rekurzija
8  
9  def fibonaci(n):
10     '''Izračunava Fib(n) u rekurzivnom postupku, n>0'''
11 
12     if n == 1:
13         return 0   # bazni slučaj
14     elif n == 2:
15         return 1   # bazni slučaj
16     else:
17         return fibonaci(n - 1) + fibonaci(n - 2) # rekurzija
```

Faktorijel prirodnog broja n definiše se sa $n! = n \cdot (n-1) \cdot \ldots \cdot 2 \cdot 1$, pri čemu je $1! = 1$ (važi i $0! = 1$, ali nula nije prirodan broj). Očigledno, može se zapisati i $n! = n \cdot (n-1)!$ pa se računanje svodi na *manje složenu* verziju *istog* problema (r7). Obrada baznog slučaja obavlja se u (r4-5). Rekurzivni postupak za 4! ilustrovan je *stablom rekurzije*[8] (slika 4.3).



Slika 4.3: Stablo rekurzije: prilikom svakog poziva funkcije `faktorijel()` kreira se novi, lokalni imenski prostor. Na dnu stabla poziva prikazan je bazni slučaj (n=1). Rezultati propagiraju unazad sve dok se ne formira konačni rezultat (4! = 24).

U slučaju izračunavanja n-tog člana Fibonačijevog niza, direktno se primenjuje rekurzivna definicija po kojoj je sledeći broj jednak zbiru prethodna dva (r17): $F(n) = F(n-1) + F(n-2)$. Za razliku od iterativne verzije rešenja iz problema 3.6, ovde se za prva dva člana uzimaju brojevi 0 i 1, a potom se definišu dva bazna slučaja (r12-15). Program za testiranje novokreirane funkcije smešten je u odvojenu datoteku:

```
1  # testira rekurzivne implementacije za n! i Fib(n)
2  import rekurzija
3
4  fak = rekurzija.faktorijel
5  fib = rekurzija.fibonaci
6
7  print('Faktorijeli:')
8  for n in range(1, 11):
9      print(fak(n), end=' ')
10
11 print('\n\nFibonacijevi brojevi:')
12 for n in range(1, 11):
13     print(fib(n), end=' ')
```

Program prvo učitava modul `rekurzija` (r2). Ovaj modul sadrži dve prethodno

[8] O pojmu stabla bilo je reči u glavi 3, kada je razmatrana heuristika pohlepne pretrage.

opisane rekurzivne funkcije. Kako su funkcije po svojoj prirodi objekti, one se mogu dodeliti promenljivima `fak` i `fib` (r4-5). Ovo je učinjeno kako bi se izbeglo navođenje punog imena funkcije koje uključuje ime modula u notaciji sa tačkom. Treba se podsetiti da sekvenca `'\n'` prouzrokuje prelazak u novi red (r11), a da funkcija `print()`, zahvaljujući opcionom parametru `end=' '`, nastavlja ispis u istom redu posle jednog blanko mesta (r9, 13):

```
Faktorijeli:
1 2 6 24 120 720 5040 40320 362880 3628800

Fibonacijevi brojevi:
0 1 1 2 3 5 8 13 21 34
```

Problem 4.5 — Broj redova u sali. Petar stoji na bini pozorišne sale koja je ispunjena do poslednjeg mesta. Kako on može da sazna broj redova u sali, a da se pri tom ne posluži brojanjem? Pretpostaviti da redovi nisu numerisani. ■

Petar bi mogao pretpostaviti da je poslednji red numerisan brojem jedan, onaj do njega brojem dva i tako redom. Tada bi prvi red do bine bio numerisan brojem koji označava broj redova u sali. Sledi algoritam za rešavanje problema u prirodnom jeziku:

Algoritam 4.2 — Broj redova u sali.

Ako neko pita osobu X za njen broj reda onda:

1. X se okreće iza sebe i pita osobu Y za njen broj reda. Kada dobije odgovor od Y, uvećava ga za jedan i rezultat saopštava naglas.
2. ako X, po okretanju, nema iza sebe ni jedan red, onda odgovara naglas – "jedan"!

Petar pita osobu, koja sedi u redu do bine, za njen broj reda. Osoba će mu, uz izvesno kašnjenje, saopštiti koliko ima redova u sali (njen broj reda)!

Treba zapaziti da umesto Petra, kome je brojanje zabranjeno, to čine gledaoci u sali. Pri rešavanju nekog problema, ponekad je potrebno *promeniti perspektivu* (ugao gledanja na problem), kako bi se isti rešio!

Pažljivom analizom algoritma 4.2, uočava se rekurzivna priroda rešenja kojom se problem određivanja broja reda svodi na svoju jednostavniju verziju: stav 1 formuliše određivanje broja reda R preko određivanja broja reda iza R. Ako je taj broj poznat, onda se R numeriše za jedan većim brojem. Redukovanje kompleksnosti problema putem rekurzije *mora* da se završi u nekom od baznih slučajeva. Tada se rešenje

dobija jednostavnim postupkom u kome su svi elementi za rešavanje poznati – stavka 2. Rekurzivno rešavanje problema, po svojoj prirodi, podseća na izvođenje dokaza putem matematičke indukcije (samo obrnutim redosledom). Sledi program koji simulira brojanje na opisani način:

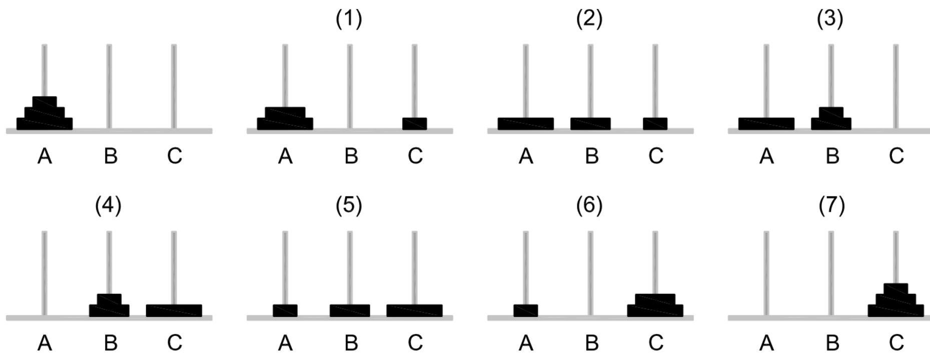
Program 4.5 — Broj redova u sali.

```
# Koliko ima redova u sali, prvi red je najdalje od bine
def pitaj_za_broj_reda(n):
    '''n: nepoznati broj tekućeg reda, vraća br. tekućeg reda'''
    if n == 1:  # bazni slučaj - pri okretanju, iza nema nikoga!
        return 1
    else:
        return 1 + pitaj_za_broj_reda(n-1)

# ukupan broj redova, pravimo se da je nepoznat
nepoznati_broj_redova = int(input('Stvaran broj redova? '))

# Petar postavlja pitanje osobi iz prvog reda do bine
odgovor = pitaj_za_broj_reda(nepoznati_broj_redova)
print('Sala ima', nepoznati_broj_redova, 'redova.')
print('Algoritam pronalazi', odgovor, 'redova.')
```

```
Stvaran broj redova? 50
Sala ima 50 redova.
Algoritam pronalazi 50 redova.
```

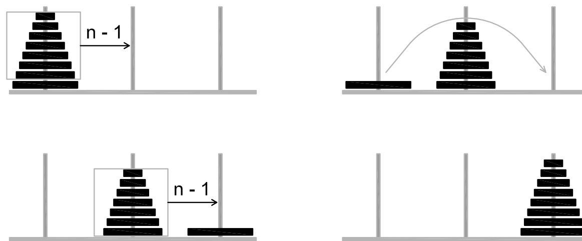
Problem 4.6 — Hanojske kule - Edouard Lucas, 1883. U dvorištu budističkog manastira u Hanoju, Vijetnam, postavljena su tri tanka drvena štapa A, B i C. Na štapu A poslagano je n drvenih diskova koji su bušni u sredini tako da mogu da se navuku na štapove. Diskovi su, u početnom položaju, poslagani od najvećeg ka najmanjem (slika 4.4 - gore levo). Potrebno je prebaciti diskove sa A na C, u svakom koraku po jedan, tako da budu poslagani na ciljni štap kao i u početnom položaju. Pri prebacivanju se koriste svi štapovi, ali tako da, ni u jednom trenutku, veći disk ne može stajati preko manjeg. Napisati funkciju koja ispisuje poteze za rešavanje problema u najmanje koraka. Koliko vremena je potrebno da se reši problem sa n diskova ako prebacivanje diska, sa štapa na štap, traje jednu sekundu? ■

Uočiti da rešavanje problema Hanojskih kula *ne zavisi* od numeracije štapova – ako je poznat postupak za prebacivanje sa A na C, onda se isti postupak može primeniti i na



Slika 4.4: Hanojske kule za $n = 3$: sekvenca rešavanja od 7 poteza.

bilo koji drugi par štapova, pošto se izvrši renumeracija (A će uvek označavati početni, a C krajnji štap). Problem se može rešiti primenom rekurzije tako što se verzija sa n diskova svodi na rešavanje dve verzije sa $n-1$ diskova (slika 4.5).



Slika 4.5: Hanojske kule: rekurzivni postupak za n diskova. Prvo se prebacuje $n-1$ disk sa polaznog (A) na pomoćni štap (B). Potom se najveći disk sa A prebacuje na ciljni štap C. Konačno, treba prebaciti $n-1$ disk sa B na C. Prebacivanje $n-1$ diskova sa A na B, odnosno sa B na C, vrši se na isti način, s tim da polazni, ciljni i pomoćni štap menjaju oznake.

U svakom koraku rekurzije problem se svodi na rešavanje verzije sa jednim diskom manje. Na ovaj način se stiže do trivijalnog baznog slučaja kada više nema diskova za prebacivanje! Uočiti da predloženi postupak vodi ka minimalnom broju poteza jer se, u svakom rekurzivnom koraku, najveći disk prebacuje sa polaznog na ciljni štap u samo jednom potezu, a bazni slučaj ne zahteva nijedan potez! Tvrđenje se može jednostavno dokazati matematičkom indukcijom. Sledi implementacija:

Program 4.6 — Hanojske kule - Edouard Lucas, 1883..

```
1 # Hanojske kule
2
3 def hanoj(n, sa='A', na='C', pomocni='B'):
4     ''' n - broj diskova,
```

```
5            sa - stap SA koga se prebacuje
6            na - stap NA koji se prebacuje
7            pomocni - pomocni stap'''
8        if n > 0:
9            hanoj(n-1, sa=sa, na=pomocni, pomocni=na)
10           print('disk', n, 'sa', sa, 'na', na) # najveci disk
11           hanoj(n-1, sa=pomocni, na=na, pomocni=sa)
12
13   hanoj(3)
```

```
disk 1 sa A na C
disk 2 sa A na B
disk 1 sa C na B
disk 3 sa A na C
disk 1 sa B na A
disk 2 sa B na C
disk 1 sa A na C
```

U svakom rekurzivnom koraku, osim u baznom, funkcija ispisuje redni broj diska koji se prebacuje (r10). Diskovi su označeni od 1 (najmanji) do n (najveći). Svi formalni parametri osim broja diskova su opcioni pa se funkcija poziva u r13 sa oznakama štapova kao na slici 4.4. Osim toga, prilikom rekurzivnih poziva u r9 i r11, ulazni parametri su imenovani radi bolje čitljivosti.

Da bi se odredio minimalni broj poteza za rešavanje problema sa n diskova, funkciju `hanoj()` treba preraditi tako da vraća ukupan broj koraka načinjenih pri svakom pozivu. U svakom pozivu, osim u baznom slučaju, pored poteza kojim se prebacuje i trenutno najveći disk, treba pribrojati i poteze za prebacivanje preostalih diskova u dva rekurzivna poziva. Funkcija vraća nulu za bazni slučaj. Sledi program koji određuje minimalni broj poteza $P(n)$ za nekoliko vrednosti broja diskova n:

```
1    # Hanojske kule, vraća broj poteza
2
3    def hanoj2(n, sa='A', na='C', pomocni='B'):
4        ''' n - broj diskova,
5            sa - stap SA koga se prebacuje
6            na - stap NA koji se prebacuje
7            pomocni - pomocni štap'''
8        if n > 0:
```

```
9          return 1 + hanoj2(n-1, sa=sa, na=pomocni, pomocni=na) \
10                  + hanoj2(n-1, sa=pomocni, na=na, pomocni=sa)
11      else:
12          return 0
13
14  print('n    P(n)')
15  for n in range(1, 6):
16      print(n, '  ', hanoj2(n))
```

Primetiti da je u r9 programska linija prekinuta, a nastavak je u sledećem redu. Za te potrebe koristi se simbol \ na kraju reda.

```
n              P(n)
1              1
2              3
3              7
4              15
5              31
```

Na osnovu dobijenih rezultata može se pretpostaviti zavisnost između n i ukupnog minimalnog broja poteza: $P(n) = 2^n - 1$. Dokaz se sprovodi matematičkom indukcijom. Za $n = 1$, tvrđenje očigledno važi. Pretpostavlja se da tvrđenje važi za $n = k$. U slučaju sa $k+1$ diskova, a na osnovu slike 4.5, za pomeranje k diskova sa A na B, odnosno sa B na C, potrebno je po $P(k)$ poteza. Za pomeranje najvećeg diska sa A na C potreban je jedan potez. Otuda je $P(k+1) = 2P(k)+1$. Kako po pretpostavci važi $P(k) = 2^k - 1$, sledi $P(k+1) = 2(2^k - 1) + 1 = 2^{k+1} - 1$, što je i trebalo dokazati. Na primer, za 32 diska i vreme od jedne sekunde po potezu, problem se rešava za $2^{32} - 1$ sekundi što iznosi nešto više od 136 godina!

Primer Hanojskih kula ilustruje poteškoće u našoj interpretaciji brzine rasta eksponencijalne funkcije. Na primer, za varijantu sa 64 diska i vreme prebacivanja od jedne sekunde po disku, problem bi se rešio za približno 585 milijardi godina! Problemi čije vreme rešavanja raste po eksponencijalnom zakonu, u odnosu na ulazne parametre, predstavljaju izuzetan izazov i za najbrže računare.

4.3.2 Prednosti i mane rekurzivnog pristupa

Na kraju izlaganja o rekurziji izlažu se potencijalne prednosti i mane u odnosu na iterativne postupke iz glave 3. Rekurzivno rešenje često se ostvaruje *kraćim* izvornim kodom pa se smanjuje mogućnost za potencijalne greške. Povećana je i čitljivost (elegancija) koda, što povoljno utiče na jednostavnije održavanje programa. Osim toga, postoje praktični problemi u kojima bi iterativno rešenje bilo dosta složenije pa je rekurzija prirodan izbor (na primer, problem obilaska stabla po dubini).

Pri svakom rekurzivnom pozivu kreira se *novi* imenski prostor sa pripadajućom tabelom imena pa je utrošak memorijskih resursa veći u odnosu na ekvivalentno iterativno rešenje. U slučaju velikog broja rekurzivnih poziva, program bi mogao da prekine sa radom usled prekomerne potrošnje memorijskih resursa. Sa stanovišta vremenske efikasnosti, rekurzivno rešenje obično je *sporije* od iterativnog jer se troši izvesno režijsko vreme za prenos parametara, kreiranje lokalnog imenskog prostora pri pozivu i njegovo uklanjanje po završetku funkcije, te konačno vraćanje rezultata u pozivajuću celinu. Jednostavan primer kojim se izračunava suma prvih n prirodnih brojeva, realizovan u obe varijante, ilustruje tipičan problem koji se odnosi na utrošak memorijskih resursa:

```
1  def suma_it(n):      # iterativna suma
2      s = 0
3      for i in range(n+1):
4          s += i
5      return s
6
7  def suma_re(n):      # rekurzivna suma
8      if n > 0:
9          return n + suma_re(n-1)
10     else:
11         return 0
12
13 print('iterativno rešenje:', suma_it(1000))
14 print('rekurzivno rešenje:', suma_re(1000))
```

```
iterativno rešenje: 500500
Traceback (most recent call last):
--------------- IZOSTAVLJENO ------------------
RecursionError: maximum recursion depth exceeded in comparison
```

Interpreter *ograničava* broj rekurzivnih poziva zbog mogućeg prekomernog utroška memorijskih resursa. U gornjem primeru, program prekida sa radom jer je dostigao *granični broj* rekurzivnih poziva. Ovaj broj *može* se eksplicitno povećati na *sopstveni* rizik. Na kraju, treba istaći da je moguće *transformisati* sva rekurzivna rešenja u odgovarajuće iterativne varijante i obrnuto, ali ovde o tome neće biti reči.

ŠTA JE NAUČENO

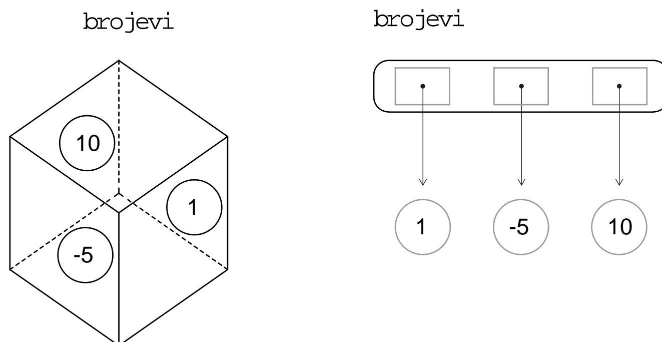
- Funkcijom se, pod jednim imenom, apstrahuje niz algoritamskih koraka za obavljanje nekog posla.
- Funkcije omogućavaju dekompoziciju složenih postupaka na jednostavnije procesne celine.
- Funkcije obično zavise od ulaznih parametara kojima se parametrizuje rešavanje problema.
- Funkcija obavezno vraća rezultate svoga rada putem naredbe `return`, ili ako se obave sve naredbe iz bloka funkcije - tada je rezultat predstavljen objektom tipa `None`.
- Prilikom poziva funkcije, pozivajuća celina nema potrebu da zna kako funkcija radi, već šta radi i kako se poziva. Argumenti se u pozivu navode po redosledu iz potpisa funkcije ili, ako se imenuju po formalnim parametrima iz potpisa, po proizvoljnom redu.
- Opcioni parametri funkcije mogu se pri pozivu izostaviti i tada dobijaju podrazumevane, predefinisane vrednosti.
- Funkcije mogu pozivati druge, kao i definisati unutrašnje funkcije.
- Moduli i paketi predstavljaju više hijerarhijske celine za organizaciju kompleksnih programskih sistema.
- Moduli sadrže srodne funkcije koje se mogu koristiti u različitim aplikacijama, a paketi organizuju module u nezavisne softverske komponente za određenu namenu.
- Promenljiva definisana u modulu ima globalni karakter - može joj se pristupati iz svih funkcija unutar modula, kao i iz onih modula u koje je definišući modul uveden putem naredbe `import`.
- Ista imena u različitim modulima dostupna su uz korišćenje notacije sa tačkom oblika `ime_modula.ime_objekta`. Ime objekta iz modula može se načiniti dostupnim sa `from ime_modula import ime_objekta`. Na ovaj način ime se koristi direktno, bez upotrebe notacije sa tačkom.
- Lokalna promenljiva definiše se u funkciji gde joj se samo i može pristupati.
- Pri svakom pozivu funkcije kreira se novi imenski prostor u kome se, unutar tabele imena, čuvaju preslikavanja između lokalnih imena i objekata na koje ta imena (promenljive) ukazuju.
- Argumenti funkcije prenose se po sistemu kopiranja objektnih referenci. Otuda se nepromenljivi objektni tipovi poput brojeva i teksta, kada su preneti kao argumenti, ponašaju u funkciji kao lokalne promenljive.

- Funkcije imaju objektnu prirodu: mogu se dodeljivati promenljivim i prenositi u druge funkcije kao argumenti.
- Rekurzivni algoritmi rešavaju problem tako što ga, u seriji funkcijskih poziva, redukuju na jednostavnije verzije istog problema, sve dok se početni problem ne svede na bazni slučaj u kome je rešenje trivijalno. Rešenje baznog slučaja propagira se unazad, sve do prvobitno pozvane funkcije koja dobija sve elemente za traženo rešenje.
- Rekurzivna rešenja mogu biti memorijski veoma skupa jer se, prilikom svakog rekurzivnog poziva, kreira novi imenski prostor sa svojim kompletom lokalnih promenljivih.

5. Kolekcije objekata: sekvence, skupovi i rečnici

Iz dosadašnjeg izlaganja poznato je da se podaci apstrahuju objektima. Objekti uzimaju vrednosti iz domena koji je određen njihovim tipom (klasom). Do sada su prevashodno bili razmatrani osnovni brojni tipovi poput `int` ili `float`. Podaci tekstualne prirode, reprezentovani objektima tipa `str`, korišćeni su za formiranje prigodnih poruka pri unosu brojnih podataka, ili pri ispisu rezultata rada programa. Objekti osnovnih brojnih tipova imaju *nedeljivu* prirodu - ne mogu se podeliti na delove koji su smisleni sami po sebi. Svaki takav objekat odnosi se na tačno jednu, jedinstvenu informaciju u procesu ili sistemu koji se modelira programom. Međutim, često je potrebno sačuvati i obraditi *srodne* informacije koje se odnose na istu osobinu različitih entiteta (na primer, ocene iz matematike za sve studente) ili na pojavu koja se posmatra u vremenu (prosečna ocena studenta posle svake godine studiranja). Zbog toga se, u svet objekata, uvodi apstrakcija koja srodne informacije *grupiše* unutar jedinstvene celine i omogućava njihovu obradu, kako na pojedinačnom, tako i na zbirnom nivou. Ova apstrakcija naziva se *kolekcijom objekata*.

Kolekcija objekata realizovana je i sama kao *objekat*. Ona se može shvatiti kao kutija u memoriji koja sadrži pojedinačne objekte (slika 5.1 levo). Pojedinačni objekti iz kolekcije nazivaju se *elementima* kolekcije. Kolekcije zapravo *ne sadrže* objekte direktno, već objektne reference (memorijske adrese) na pripadajuće elemente (slika 5.1 desno)! Kako se objektima isključivo pristupa preko njihovih referenci, stiče se utisak da se oni sami čuvaju unutar kolekcije.

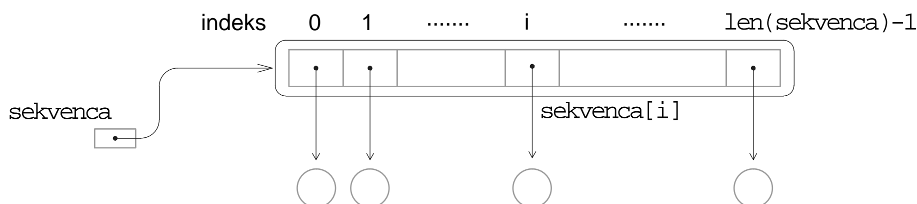


Slika 5.1: Kolekcije sadrže objektne reference: kolekcija `brojevi` obuhvata reference koje ukazuju na tri celobrojna objekta (1, 10 i -5).

Različiti tipovi kolekcija razlikuju se po tome kako su njihovi elementi *organizovani* unutar kolekcije. Tipom kolekcije definiše se koji su *dozvoljeni tipovi* za objekte elemente, kako se *pristupa* pojedinačnim elementima i da li se kolekcija po formiranju može *menjati* ili ne. U ovoj glavi biće reči o tri familije kolekcija u Pajtonu: prvo se razmatraju *sekvence*, potom *skupovi* i na kraju *rečnici*.

5.1 Sekvence

Kolekcije koje predstavljaju *uređene* skupove objekata nazivaju se *sekvence*[1]. Uređenost podrazumeva postojanje redosleda unutar skupa koji čini kolekciju - poznato je koji objekat je prvi, drugi, treći i tako redom do poslednjeg. Uređenost se najčešće definiše redosledom ubacivanja elemenata u kolekciju. Na ovaj način omogućeno je pristupanje pojedinačnim elementima putem mehanizma *indeksiranja*. Indeks je ceo broj koji predstavlja *pomeraj* u odnosu na početak sekvence. Prvom elementu sekvence od n članova pristupa se preko indeksa sa vrednošću 0 (`sekvenca[0]`), a poslednjem sa vrednošću $n-1$ (`sekvenca[n-1]`). Ugrađena funkcija `len()` vraća ukupan broj elemenata, odnosno dužinu prosleđene sekvence (slika 5.2).



Slika 5.2: Sekvence i indeksiranje: `i` je indeks `i+1`-vog elementa. Pristup pojedinačnim elementima obavlja se navođenjem indeksa u uglastim zagradama uz ime sekvence - `[]`.

[1] Engl. *Sequences*.

U glavi 3, prilikom izlaganja o petlji `for`, pomenute su sekvence tipa `range` koje mogu da sadrže samo celobrojne objekte (tip `int`):

```
>>> brojevi = range(25)  # kreira sekvencu brojeva od 0 do 24
>>> type(brojevi)        # tip (klasa) objekta
<class 'range'>
>>> len(brojevi)         # dužina sekvence (broj elemenata)
25
>>> brojevi[0]           # pristup prvom elementu
0
>>> type(brojevi[0])     # pokazuje tip elementa u sekvenci
<class 'int'>
>>> brojevi[24]          # pristup poslednjem elementu
24
>>> brojevi[25]          # početnička greška, element ne postoji
Traceback (most recent call last):
  File "<pyshell#25>", line 1, in <module>
    brojevi[25]
IndexError: range object index out of range
```

Pored sekvenci tipa `range`, u ovoj glavi izučavaju se sekvence tipa `str` (tekst), `list` (lista) i `tuple` (torka).

5.1.1 Tekst - tip `str`

Tekstualni podaci reprezentovani su sekvencama tipa `str`, a tekstualni objekat naziva se još i *string*. Najčešće se kreira pri dodeli vrednosti, uz upotrebu jednostrukih, dvostrukih ili trostrukih navodnika, ali i pomoću konstruktora `str()` (glava 1.3.4):

```
>>> tekst = 'ja sam "kratki" tekst'
>>> type(tekst)
<class 'str'>
>>> tekst2 = "i ja sam tekst, al' sa apostrofom"
>>> tekst3 = """ja se
protežem
u tri reda"""
>>> tekst4 = '''a ja
u dva reda'''
>>> str(4.61)
'4.61'
>>> str(False)
'False'
```

Navođenjem tri jednostruka ili dvostruka navodnika omogućeno je formiranje teksta koji se proteže u više redova. Međutim, rad sa većom količinom teksta postaje praktičan

tek uz upotrebu tekstualnih datoteka o kojima će više reči biti u glavi 9. Kako je svaki tekstualni objekat u Pajtonu po svojoj prirodi kolekcija, postavlja se pitanje šta su njeni elementi? Da bi se shvatila priroda teksta potrebno je uvesti pojam *karaktera*. Karakter predstavlja najmanju moguću tekstualnu jedinicu koja se odnosi na simbol u pisanoj formi nekog jezika. On se u računaru reprezentuje *nizom bita*. Na primer, veliko slovo 'A' moglo bi se predstviti kao 01000001, dok bi se simbol za dolar '$' mogao predstaviti sa 00100100. Karakteri se, pored prirodnih, mogu odnositi i na simbole formalnih jezika kao što je matematički. Tako bi simbol sumacije '$\sum$' mogao da bude zapisan kao 11100010 10001000 10010001.

Kako se sa n bita može predstaviti 2^n različitih simbola, to broj različitih karaktera, iz nekog konačnog skupa koji se želi predstaviti, diktira dužinu reprezentacije u bitima. Način po kome se karakteri iz nekog skupa preslikavaju u bite definiše *sistem kodiranja*[2]. Binarni kodovi za 'A' i '$' dati su u sistemu kodiranja ASCII[3] koji koristi sedam bita u okviru jednog bajta, što podrazumeva $2^7 = 128$ različitih simbola. ASCII *ne obuhvata* karaktere koji se odnose na simbole srpske latinice ili ćirilice. Slično važi i za simbol sumacije. Postoje brojni standardi koji definišu preslikavanja za različite podskupove svetskih jezika, kao i svih jezika zajedno. U Pajtonu se tekst predstavlja putem standarda po imenu *Unicode*.[4]

Unicode standard *odvaja* jedinstvenu identifikaciju simbola od sistema kodiranja u određeni niz bita. Svakom simbolu pridružuje se nenegativan ceo broj po imenu *Unicode code point* (UCP). Trenutno se pokriva više od 128.000 simbola iz 135 jezičkih pisama i skupova simbola, uz moguća proširenja. Postoje različiti sistemi kodiranja za prevođenje UCP brojeva u bitove, a Pajton interno koristi sistem UTF-8 koji podrazumeva jedan, dva, tri ili četiri bajta za kodiranje karaktera. Simbol sumacije iz prethodnog teksta reprezentovan je primenom UTF-8 kodiranja sa tri bajta.

Tekstualni objekat predstavlja sekvencu pojedinačnih karaktera koji odgovaraju UCP brojevima Unicode standarda. U Pajtonu, za razliku od C-a ili Jave, *ne postoji* poseban tip za pojedinačne karaktere pa se svaki karakter tretira kao tekstualna sekvenca dužine jedan. Pojedinačnim karkaterima iz sekvence pristupa se putem indeksiranja:

```
>>> poslovica = 'Ko rano rani dve sreće grabi!'
>>> len(poslovica)    # dužina tekstualne sekvence
29
>>> print(poslovica[0], poslovica[28])  # prvi i poslednji element
K !
>>> type(poslovica[0])  # tip svakog elementa je string dužine 1
<class 'str'>
>>> poslovica[100]   # indeks van sekvence
Traceback (most recent call last):
```

[2] Engl. *Character encoding* ili samo *encoding*; skup karaktera - *character set*.
[3] *American Standard Code for Information Interchange*.
[4] Široko rasprostranjen standard čija se specifikacija nalazi na `http://www.unicode.org`

```
  File "<pyshell#60>", line 1, in <module>
    poslovica[100]
IndexError: string index out of range
>>> ord('K') # za zadati karakter, vraća UCP broj
75
>>> chr(75)  # za zadati UCP broj, vraća karakter
'K'
>>> len('')  # prazan tekst dužine 0 (sekvenca bez elemenata)
0
```

Prilikom indeksiranja treba obratiti pažnju da vrednost indeksa bude u dozvoljenom opsegu. Funkcije `ord()` i `chr()` omogućavaju da se, za navedeni karakter, dobije njegova UCP vrednost i obrnuto. Uočiti specijalan slučaj prazne sekvence koja ima dužinu nula. Tekstualni objekti se u ovoj glavi označavaju kao tekstualne sekvence da bi se naglasila njihova priroda uređene kolekcije.

Problem 5.1 — Šifrovana poruka. Ana i Marko dogovorili su se da razmenjuju šifrovane poruke na sledeći način: jedinstveni Unicode broj svakog karaktera iz originalne poruke treba pomnožiti sa a, pa na rezultat dodati broj b. Potom se broj prevodi u odgovarajući karakter šifrovane poruke. Napisati program koji će, za uneto a i b, šifrovati (dešifrovati) ulaznu poruku. Ako je $a = 2$ i $b = 3$, dešifrovati sledeću Aninu poruku Marku: “ïÕËÕÝáCéÍCËÅßÅéCíCm”. ■

Program 5.1 — Šifrovana poruka.

```
1  # šifruje ili dešifruje poruke koristeći linearno kodiranje
2  poruka = input('poruka? ')
3  a = int(input('a? '))
4  b = int(input('b? '))
5  šifruj = input('šifruj (š) ili dešifruj (d) ') == 'š'
6
7  n = len(poruka)
8  if šifruj:
9      print('šifrovana poruka je ')
10     for i in range(n):
11         print(chr(a * ord(poruka[i]) + b), end='')
12 else:
13     print('originalna poruka je ')
14     for i in range(n):
15         print(chr((ord(poruka[i]) - b) // a), end='')
```

Unos potrebnih podataka obavlja se u (r2-5). Kako funkcija `input()` vraća unos sa tastature u obliku tekstualne sekvence, logička promenljiva `šifruj` ima vrednost `True` samo ako je na ulazu uneto slovo "š" (r5). Uočiti da se tekstovi porede po jednakosti, *karakter po karakter*, korišćenjem relacije ==. Tako je `'Pajton' == 'Pajton'` istinito, ali `'pajton' == 'Pajton'` nije (važi `'pajton' != 'Pajton'`). Dve tekstualne sekvence mogu se porediti i korišćenjem relacija <, >, <= i >=, kada se posmatra *leksikografski poredak*.

Tekstualne sekvence $A = a_1 a_2 ... a_n$ i $B = b_1 b_2 ... b_m$ porede se leksikografski na sledeći način: porede se vrednosti $ord(a_1)$ i $ord(b_1)$; ako su različite, njihov odnos određuje rezultat poređenja (ako $ord(a_1) < ord(b_1)$, onda $A < B$); ako su iste, onda se prelazi na sledeći karakter. Proces se nastavlja sve dok se ne naiđe na *različite* vrednosti čiji odnos određuje rezultat poređenja ili dok se bar jedna sekvenca *ne iscrpi*. Ako sekvenca ostane bez karaktera za poređenje, onda je ona manja od druge. Ako obe sekvence ostanu bez karaktera za poređenje, onda su jednake. Na primer, važi da je $aaa < aab$, $aa < aab$, ali i $aaa < b$.

Dužina unete tekstualne sekvence određuje se uz pomoć ugrađene funkcije `len()` koja se može primeniti na *sve* tipove sekvenci (r7). Bez obzira da li uneta poruka treba da se šifruje ili dešifruje, u petlji `for` generišu se indeksi za pristup svakom karakteru poruke (r10-11, r14-15). Poruka se šifruje i ispisuje karakter po karakter (r11). Prvi argument funkcije `print()` predstavlja realizaciju algoritma šifrovanja iz teksta zadatka. Zahvaljujući opcionom parametru `end=''`, ispis sledećeg šifrovanog karaktera nastavlja se u istom redu. Postupak dešifrovanja je inverzan (r15), s tim da se primenjuje celobrojno deljenje realizovano sa `//`. Kada bi se koristio operator `/`, rezultat deljenja bio bi realan broj tipa `float`. Tada bi, pri pozivu funkcije `char()`, nastala greška jer funkcija očekuje celobrojni argument (UCP broj dešifrovanog karaktera). Sledi prikaz rada programa:

```
poruka? originalna poruka
a? 1
b? 0
šifruj (š) ili dešifruj (d) š
šifrovana poruka je
originalna poruka
>>>
==== RESTART: C:/programi/p5_1.py ====
poruka? šifrovana poruka
a? 1
b? 0
šifruj (š) ili dešifruj (d) d
originalna poruka je
šifrovana poruka
>>>
==== RESTART: C:/programi/p5_1.py ====
```

```
poruka? ïÕËÕÝáCéÎCËÅßÅéCíCm
a? 2
b? 3
šifruj (š) ili dešifruj (d) d
originalna poruka je
vidimo se danas u 5
```

Prethodni prikaz pokazuje da je šifrovana poruka identična originalnoj za $a = 1$ i $b = 0$. Konačno, Ana Marku poručuje: "vidimo se danas u 5".

Svi tipovi kolekcija, pa samim tim i sekvence, omogućavaju proveru pripadnosti kolekciji uz pomoć operatora `in`. U slučaju tekstualne sekvence on proverava pripadnost kako karaktera, tako i proizvoljnog teksta zadatoj sekvenci. Ako tekst t_1 pripada tekstu t_2, onda se kaže da je t_1 *podtekst* (ili *podstring*) od t_2[5]. Pored operatora `in`, sve sekvence, osim tipa `range`, podržavaju i dva operatora *spajanja* (konkatenacije): `+` i `*`. Ovi operatori, od polaznih, uvek kreiraju *nove sekvence*. U slučaju `+`, dve sekvence se spajaju tako što se druga nadovezuje na prvu, a u slučaju `*`, sekvenca se nadovezuje sama na sebe željeni broj puta. Ponašanje operatora `in`, `+` i `*` ilustrovano je u sledećem primeru:

```
>>> tekst = 'Bolje vrabac u ruci nego golub na grani!'
>>> 'v' in tekst     # da li je karakter element sekvence tekst?
True
>>> 'vrabac u ruci' in tekst   # da li je podstring?
True
>>> 'bolje' in tekst   # mala i velika slova prave razliku
False
>>> 'idemo' + ' dalje' # spajanje dva teksta u jedan
'idemo dalje'
>>> 'idemo' + 5 * '!'  # nadoveži tekst od 5 uzvičnika
'idemo!!!!!'
>>> '' + 'AAA'         # nadoveži AAA na prazan tekst
'AAA'
```

Operatori `+` i `*` korišćeni su već u glavi 3, u problemu 3.5. Iz poslednjeg primera uočava se da dodavanje praznog teksta, na bilo koju sekvencu, ne menja istu. Opšte je poznato da se operatori `+` i `*` koriste pri aritmetičkim operacijama sa brojnim tipovima. U pogledu teksta, oni imaju potpuno drugačije ponašanje. Mehanizam u višem programskom jeziku po kome se ponašanje operatora menja, u zavisnosti od tipa operanada koji učestvuju u operaciji, zove se *preopterećenje operatora*[6]. Koristeći novouvedene operatore, problem šifrovanja može se rešiti i na sledeći način:

[5] Engl. t_1 *substring of* t_2.
[6] Engl. *Operator overloading*.

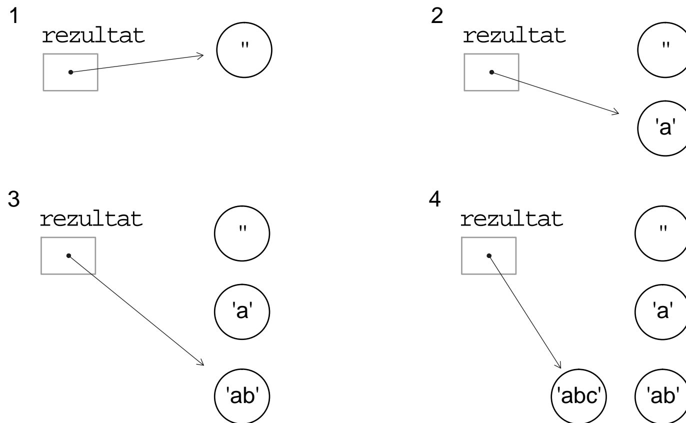
```
1  # šifruje ili dešifruje poruke koristeći linearno kodiranje
2  poruka = input('poruka? ')
3  a = int(input('a? '))
4  b = int(input('b? '))
5  sifruj = input('šifruj (š) ili dešifruj (d) ') == 'š'
6
7  rezultat = ''
8  if sifruj:
9      for k in poruka:
10         rezultat += chr(a * ord(k) + b)
11 else:
12     for k in poruka:
13         rezultat += chr((ord(k) - b) // a)
14
15 print(poruka, '-->', rezultat)
```

Ideja rešenja bazira se na spajanju dve tekstualne sekvence. U r7, promenljiva `rezultat` ukazuje na prazan tekst. Ona će, u svakom koraku iterativnog algoritma, ukazivati na prethodno šifrovanu (dešifrovanu) tekstualnu sekvencu. Prilikom šifrovanja (dešifrovanja), tekućoj sekvenci pridodaje se novoobrađeni karakter (r10, odnosno r13). Treba uočiti da zaglavlje petlje `for` (r9, 12), umesto indeksa, koristi promenljivu `k` koja u svakoj iteraciji ukazuje na po jedan karakter iz unete poruke. Zaglavlje se može pročitati na sledeći način: "za svaki element `k` iz sekvence `poruka`, uradi nešto sa `k`". Na ovaj način izbegnuto je računanje dužine za objekat `poruka` i kreiranje sekvence tipa `range`, čiji su elementi služili kao indeksi za pristupanje ciljnoj sekvenci.

Za dohvatanje svih elemenata iz sekvence *proizvoljnog* tipa koristi se petlja oblika: `for element in sekvenca`. Uočiti da se, u Pajtonu, petlja `for` *uvek* vezuje za neku od sekvenci (podsetiti se da i do sada korišćeni `range()` kreira celobrojnu sekvencu).

U r10 i r13, ažuriranje rezultujuće poruke vrši se tako što se postojećem tekstu pridodaje sledeći karakter. Međutim, tom prilikom se *ne proširuje* ista, već se *formira* nova tekstualna sekvenca! Ovo je posledica *nepromenljive* prirode tekstualnih objekata o čemu je već bilo pomena u glavi 2.3.5. Na primer, neka je konačna rezultujuća poruka `'abc'`. Na slici 5.3, prikazano je formiranje ove sekvence putem četiri različita, nepromenljiva tekstualna objekta.

Da su tekstualne sekvence nepromenljive, svedoči i poruka interpretera prilikom pokušaja promene nekog od pojedinačnih elemenata (karaktera):



Slika 5.3: Nepromenljivost teksta: formiranje sekvence `'abc'` pridodavanjem pojedinačnih karaktera na `rezultat`. Ovom prilikom upotrebljena su četiri tekstualna objekta.

```
>>> tekst = 'Bolje vrabac u ruci nego golub na grani!'
>>> tekst[0] = 'b'
Traceback (most recent call last):
  File "<pyshell#38>", line 1, in <module>
    tekst[0] = 'b'
TypeError: 'str' object does not support item assignment
```

Pored indeksiranja pojedinačnih elemenata, sve sekvence podržavaju *izdvajanje podsekvenci*[7] navođenjem opsega unutar uglastih zagrada `[]`. Navođenjem `sekvenca[i, j]`, formira se nova sekvenca od elemenata određenih indeksom `k`, gde je `i <= k < j`. U slučaju teksta, izdvajanje je ilustrovano sledećim primerom:

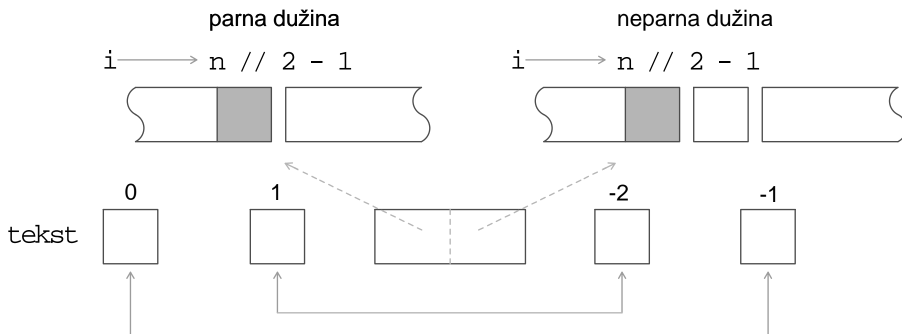
```
>>> tekst = 'Bolje vrabac u ruci nego golub na grani!'
>>> tekst[6:12]
'vrabac'
>>> tekst[:5]
'Bolje'
>>> tekst[5:]
' vrabac u ruci nego golub na grani!'
>>> tekst[-1]
'!'
>>> tekst[6:-7]
'vrabac u ruci nego golub na'
>>> tekst[-6:]
'grani!'
```

[7] Engl. *Slicing*.

```
>>> tekst[6:5]
''
>>> tekst[0:-1:2]
'Blevaa  uing ou agai'
```

Iz gornjeg primera uočava se nekoliko specifičnih situacija. Ako se jedan od indeksa ne navede, onda se biraju elementi počevši od prvog pa sve do elementa neposredno pre navedenog indeksa (`[:j]`), odnosno od elementa sa navedenim indeksom pa sve do kraja sekvence (`[i:]`). Navođenjem negativnog indeksa `i`, bira se `i`-ti element s kraja sekvence. Ako je, pri navođenju opsega, prvi indeks veći ili jednak drugom indeksu, vraća se prazna sekvenca. Prilikom navođenja opsega podržan je i treći argument koji, ako se navede, označava korak po kome se biraju elementi iz sekvence. U poslednjem redu primera formira se sekvenca sa elementima od prvog do pretposlednjeg, uz uslov da se bira svaki drugi karakter (indeksi 0, 2, 4, ...).

Problem 5.2 — Palindromi. Tekst je *palindrom* ako se čita sdesna nalevo isto kao i sleva nadesno (*radar*, *kajak*). Napisati funkciju koja testira da li je neki tekst palindrom. Problem rešiti kako iterativno, tako i rekurzivno. Ispisati sve podtekstove unetog teksta koji predstavljaju palindrome. ■



Slika 5.4: Palindromi: poređenje odgovarajućih karaktera u tekstualnoj sekvenci.

Ideja iterativnog algoritma ilustrovana je na slici 5.4. Za zadatu tekstualnu sekvencu dužine n (promenljiva `tekst`), ispituje se da li je prvi karakter jednak poslednjem. Postupak se prekida ako to nije slučaj, a `tekst` tada ne predstavlja palindrom. U suprotnom, treba ispitati da li je drugi karakter jednak pretposlednjem. Sukcesivna testiranja proveravaju da li je i-ti karakter sa početka jednak i-tom karakteru sa kraja sekvence. U zavisnosti od parnosti broja n, testiranja se obavljaju zaključno sa karakterom na poziciji sa indeksom $n/2-1$ (parna dužina), odnosno $(n-1)/2-1$ (neparna dužina). Neka je uvedena promenljiva `limit=n//2`, gde operacija `//` predstavlja celobrojno deljenje. Sekvenca `tekst` je palindrom ako su karakteri `tekst[i]` i `tekst[-1-i]`

jednaki za svako `i=0,1,...limit-1`, pri čemu negativni indeksi označavaju karaktere s kraja sekvence.

U rekurzivnoj varijanti rešenja ispituje se jednakost prvog i poslednjeg karaktera. Ako oni nisu jednaki, uneta sekvenca ne predstavlja palindrom, a ako jesu, onda se problem redukuje na ispitivanje da li je tekst, koji se dobija *odbacivanjem* prvog i poslednjeg karaktera, palindrom ili ne. Bazni slučaj odnosi se na praznu sekvencu koja, po definiciji, predstavlja palindrom. Opisani algoritmi realizovani su na sledeći način:

Program 5.2 — Palindromi.

```
1  # svi palindromi u tekstu
2
3  def palindrom_it(tekst):
4      ''' ispituje da li je tekst palindorm '''
5      n = len(tekst)
6      limit = n//2
7      for i in range(limit):
8          if tekst[i] != tekst[-i-1]:
9              return False
10     return True
11
12 def palindrom_re(tekst):
13     ''' ispituje da li je tekst palindorm '''
14     if tekst == '':
15         return True
16     elif tekst[0] != tekst[-1]:
17         return False
18     else:
19         return palindrom_re(tekst[1 : -1])
20
21 tekst = input('tekst? ')
22 n = len(tekst)
23 for duzina in range(n, 0, -1):
24     for pocetak in range(0, n-duzina+1):
25         podtekst = tekst[pocetak : pocetak+duzina]
26         if palindrom_re(podtekst): # može i palindrom_it
27             print(podtekst)
```

Funkcije `palindrom_it()` i `palindrom_re()` predstavljaju iterativnu i rekurzivnu varijantu rešenja. U r8 se pristupa odgovarajućim elementima sa početka, odnosno sa

kraja sekvence, dok se u r19 vrši izdvajanje podteksta koji predstavlja novu sekvencu za testiranje. Program koji testira rad opisanih funkcija (r21-27) *generiše* sve podtekstove za dati tekst dužine `n` i ispisuje one sekvence koje predstavljaju palindrome. Sve podsekvence generišu se preko svih mogućih dužina unutar petlje u r23. Sekvenca fiksirane dužine (`duzina`) može započeti na indeksima od 0 do `n-duzina` – unutrašnja petlja u r24. Podtekst sa fiksiranim početkom i dužinom izdvaja se u r25, pa se potom testira da li je palindrom (r26). Palindromi se ispisuju počevši od najdužeg pa do najkraćih – petlja u r23 počinje od `n` i ima negativni korak (-1). Uočiti da se palindrom može detektovati i iz uslova `tekst == tekst[-1::-1]`, što predstavlja najkraće, ali ne i najbrže rešenje (zbog potrebe obrtanja tekstualne sekvence).

5.1.2 Objekti i metode. Metode za rad sa tekstom

Tip (klasa) objekta određuje, pored skupa dozvoljenih vrednosti, operacije koje se mogu obaviti nad objektom. Tako operacija plus (+) obavlja sabiranje dva celobrojna, realna ili kompleksna objekta, dok primenjena na tekst, vrši spajanje dve sekvence. Međutim, ako se pokuša sa sabiranjem dva objekta tipa `range`, doći će do greške. Sa druge strane, funkcija predstavlja imenovani niz naredbi za obavljanje odgovarajućeg zadatka i kao takva *nije vezana* ni za jedan konkretni objekat. Razume se, ulazni parametri utiču na rezultat njene obrade, ali funkcija postoji *nezavisno* od njihovih konkretnih vrednosti.

Posmatra se svet automobila modelovan klasom `Auto`. Svaki auto može se pokrenuti iz mirovanja u procesu koji podrazumeva paljenje motora, ubacivanje u brzinu i otpuštanje kvačila (modeli sa ručnim menjačem). U računarskom modelu ovog sveta mogla bi postojati funkcija `start()` koja, za ulazni parametar, uzima konkretan objekat tipa `Auto`. Ona ga potom pokreće izvršavajući prethodno opisane korake. Tako bi bili mogući sledeći izrazi: `start(mikina_kola)` ili `start(perin_crveni_golf)`. Međutim, na parkingu u realnom svetu ne postoji izolovana funkcija `start()` koja pokreće različite automobile, već svaki auto, po svojoj prirodi, ima *ugrađeno ponašanje* da se pokrene iz stanja mirovanja. Kako je Pajton objektno orijentisan jezik, podrazumeva se da objekti imaju ugrađena ponašanja u zavisnosti od tipa, odnosno da se *nad njima* mogu izvršavati pojedine operacije koje se zovu *metodama*.

Metode predstavljaju funkcije koje su definisane *unutar* objektnog tipa i koje se *vezuju* za konkretan objekat. Na primer, svi automobili iz klase `Auto` imaju ugrađenu metodu `start()` koja se poziva nad konkretnim automobilom u cilju njegovog pokretanja. Metoda se poziva nad objektom u notaciji sa tačkom, poput `mikina_kola.start()` ili `anin_golf.start()`. Metoda `start()` nema ulaznih parametara, već pokreće onaj automobil nad kojim se poziva. Više reči o metodama biće u glavi 10 koja se bavi korisnički definisanim tipovima – klasama.

Izlaganje o tekstualnim sekvencama završava se pregledom najčešće korišćenih metoda nad objektima tipa `str` (tabela 5.1). Sledi primer pozivanja pomenutih metoda:

```
>>> tekst = '   Bolje vrabac u ruci nego golub na grani!    '
>>> tekst.upper()
'   BOLJE VRABAC U RUCI NEGO GOLUB NA GRANI!    '
>>> tekst.lower()
'   bolje vrabac u ruci nego golub na grani!    '
>>> tekst.strip()
'Bolje vrabac u ruci nego golub na grani!'
>>> tekst.count('ra')
2
>>> tekst.replace('g', 'G_')
'   Bolje vrabac u ruci neG_o G_olub na G_rani!    '
>>> tekst.find('ruci')
18
```

metoda	opis
`t.lower()`	vraća novi tekst kao `t`, ali sa svim malim slovima
`t.upper()`	vraća novi tekst kao `t`, ali sa svim velikim slovima
`t.strip()`	vraća novi tekst kao `t`, bez *belih* karaktera na početku i kraju
`t.count(t1)`	vraća broj pojavljivanja podteksta `t1` u `t`
`t.replace(ts, tn)`	vraća novi tekst kao `t`, gde je podtekst `ts` zamenjen sa `tn`
`t.find(t1)`	vraća indeks prvog pojavljivanja podteksta `t1` u `t`

Tabela 5.1: Često korišćene metode tipa `str`.

Prilikom opisa metode `strip()`, u tabeli 5.1 pomenuti su *beli karakteri*.[8] U njih spadaju *blanko* (`' '`) i kontrolni karakteri poput onih za prelazak u novi red (`\n`), tabulaciju (`\t`) i sličnih.

Oblikovanje teksta

Tekst se često *oblikuje* tako da u svoj sastav, pored fiksnih, uključi i promenljive delove koji se odnose na različite tipove podataka. Oblikovanje (ili *formatiranje*) upotrebljava se najčešće prilikom ispisa na ekran ili u tekstualnu datoteku. U tu svrhu koristi se tekstualna metoda `format()` ili, u novijim verzijama Pajtona, [9] *f-string*:

```
>>> a, b, pi = 'Pajton', 'C++', 3.14
>>> '{} je zakon, u odnosu na {}'.format(a, b)
'Pajton je zakon, u odnosu na C++'
>>> f'{a} je zakon, u odnosu na {b}'
'Pajton je zakon, u odnosu na C++'
```

[8] Engl. *Whitespace characters*.
[9] Počevši od verzije 3.6, formatirajući string ili skraćeno f-string.

```
>>> 'pi = {}'.format(pi)
'pi = 3.14'
>>> f'pi = {pi}'
'pi = 3.14'
>>> '{2}-{1}-{0}'.format('prvi', 'drugi', 'treći')
'treći-drugi-prvi'
```

Prethodni primer pokazuje da se argumenti metode `format()` raspoređuju unutar tekstualne sekvence nad kojom je metoda pozvana. Argumenti se raspoređuju unutar *polja* definisanih vitičastim zagradama (`{}`) po redosledu navođenja ili po redosledu koji je određen brojem unutar zagrade (poslednji primer, nulom se označava prvi argument iz liste). Isti efekat postiže se korišćenjem prefiksa `f` neposredno ispred tekstualne sekvence (drugi i četvrti primer). Nadalje će se, zbog kraćeg zapisa, koristiti oblikovanje tipa f-string. Prilikom oblikovanja, moguće je navesti dodatne parametre koji diktiraju *širinu* i *unutrašnje poravnanje* polja:

```
>>> a, b, pi, s = 'Pajton', 'C++', 3.14, 'ste'
>>> f'{a:10} je zakon, u odnosu na {b:>10}'
'Pajton     je zakon, u odnosu na        C++'
>>> f'pi = {pi:8}'
'pi =     3.14'
>>> f'pi = {pi:<8}'
'pi = 3.14    '
>>> f'pi = {pi:^8}'
'pi =   3.14  '
>>> f'Kako {s:1}'
'Kako ste'
>>> f'Kako {s:.1}'
'Kako s'
```

U gornjem prikazu, reč `'Pajton'` (promenljiva `a`) upisana je u polje širine 10 karaktera. Za tekstualne podatke podrazumeva se levo poravnanje. Za desno poravnanje, posle dvotačke se navodi znak `'>'` – promenljiva `b`, reč `'C++'`. Kada se u polja raspoređuju brojevi, onda se primenjuje desno poravnanje (prvi primer sa `3.14`). Za levo poravnanje brojeva koristi se `'<'` (drugi primer sa `3.14`). Centralno poravnanje postiže se upotrebom simbola `'^'` (treći primer sa `3.14`). Ako je podatak *duži* od zadate širine polja, onda se ona ignoriše, sem u slučaju kada se pre broja navede tačka. Tada se podatak odseca kako bi stao u polje tražene širine (poslednja dva primera).

Ako je brojeve potrebno dodatno formatirati (ispis znaka, broj decimalnih mesta i drugo), onda se koriste specifikatori formata `'d'`, `'f'` i `'e'`, za cele, realne i brojeve u eksponencijalnoj notaciji:

```
>>> f'pi = {pi:6.3f}'
'pi =  3.140'
>>> f'x = {30:+6d}'
'x =    +30'
>>> f'x = {1/3:6.2f}'
'x =   0.33'
>>> f'x = {1/6:4.2e}'
'x = 1.67e-01'
```

Za realne i brojeve u eksponencijalnom zapisu, notacija `x.y` desno od dvotačke označava ukupnu širinu polja (`x`) i broj decimalnih mesta (`y`). Zapaziti da se, prilikom interpretacije broja decimalnih mesta, dodaje potreban broj nula sa desne stane (prvi primer), ili vrši neophodno zaokruživanje (poslednji primer). Oblikovanje teksta putem formatirajućeg stringa koristiće se za formiranje preglednijeg ispisa.

5.1.3 Liste - tip `list`

Lista predstavlja sekvencu objekata *proizvoljnih* tipova, sa mogućnošću *brisanja* starih i *dodavanja* novih elemenata. Objekti tipa liste najčešće se kreiraju u naredbi dodele vrednosti pomoću *uglastih* zagrada ili upotrebom konstruktora `list()`:

```
>>> neparni = [1, 3, 5, 7, 9, 11, 13,
               15, 17, 19, 21, 23]
>>> type(neparni)
<class 'list'>
>>> print(neparni, 'ima dužinu', len(neparni))
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23] ima dužinu 12
>>> imena = ['Vlada', 'Pera', 'Stanko']
>>> print(imena, 'ima dužinu', len(imena))
['Vlada', 'Pera', 'Stanko'] ima dužinu 3
>>> prazna_lista = list()  # može i prazna_lista = []
>>> print(prazna_lista, 'ima dužinu', len(prazna))
[] ima dužinu 0
>>> list('Pera i Maja')
['P', 'e', 'r', 'a', ' ', 'i', ' ', 'M', 'a', 'j', 'a']
>>> list(range(5, 15))
[5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
```

Primer ilustruje formiranje dve liste različitih tipova objekata: celobrojna lista `neparni` i lista tekstualnih objekata `imena`. Lista `neparni` ilustruje mogućnost definisanja u više redova, pri čemu se odstupa od strogih pravila za poravnanje izvornog koda. Slično važi i za ostale kolekcije koje se definišu pomoću različitih tipova zagrada: posle navedene leve zagrade, programski red se može *prekinuti* bilo gde “unutar” navođenja, pa potom nastaviti od *proizvoljne* pozicije u sledećim redovima sve do desne zagrade.

Lista prazna_lista ne sadrži ni jedan element pa je njena dužina jednaka nuli. Konačno, pomoću konstruktora list() kreiraju se dve liste čiji su elementi pojedinačni karakteri iz ulazne tekstualne sekvence, odnosno celi brojevi iz zadatog opsega.

Često postoji potreba za grupisanjem različitih tipova pod jednim imenom. Sledeći primer odnosi se na listu koja sadrži raznorodne objekte:

```
>>> razno = [1.11, True, 'Piton', [1, 3, 5, 7, 9], '!']
>>> print(razno, 'je dužine', len(razno))
[1.11, True, 'Piton', [1, 3, 5, 7, 9], '!'] je dužine 5
```

Za razliku od prethodnog primera, lista razno sadrži objekte različitih tipova među kojima je i *unutrašnja* lista celobrojnih objekata. Broj elemenata unutrašnje liste ne utiče na dužinu osnovne liste – svaka unutrašnja lista broji se kao jedan element. Za ispitivanje da li objekat pripada listi ili ne, koristi se operator in:

```
>>> print(3 in neparni, 3 not in imena)
True True
```

Kao i tekstualne sekvence, liste koriste mehanizam indeksiranja za izdvajanje pojedinačnih elemenata. Pozitivni indeksi predstavljaju *pomeraje* u odnosu na prvi element (sa indeksom nula), dok negativni označavaju pomeraje u odnosu na poziciju neposredno iza poslednje (poslednji element ima indeks -1):

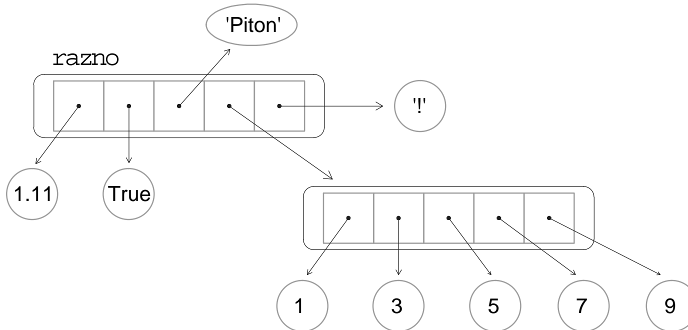
```
>>> x = [1, 3, ['a', 'b'], False]
>>> print(x[1], x[2], x[-3]) # 2., 3. i 3. element s kraja liste
3 ['a', 'b'] 3
>>> x[2][1]  # pristupanje elementima unutrašnje liste
'b'
```

Prilikom pristupanja elementima unutrašnje liste, prvo se pristupa samoj listi (prvi indeks), a potom njenom željenom elementu (drugi indeks). Liste, poput ostalih kolekcija, ne sadrže objekte direktno, već njihove objektne reference – slika 5.5.

Pošto lista sadrži objektne reference, moguće je da ima *veću* dužinu nego što u njoj ima različitih objekata! Situacija je ilustrovana sledećim primerom:

```
>>> x = [1, 2, 1]
>>> print (len(x), x[0] is x[2])
3 True
```

U gornjem primeru korišćen je operator is koji utvrđuje da li dve promenljive (reference) ukazuju na isti objekat u memoriji. Očigledno je da su prvi i poslednji



Slika 5.5: Struktura liste: lista `razno` ne sadrži objekte direktno, već objektne reference. Isto važi i za unutrašnju listu.

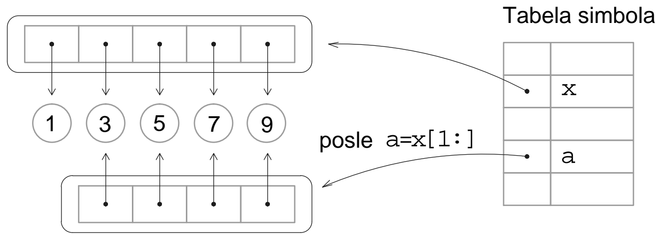
objekat isti. Na ovaj način, kada lista sadrži veliki broj memorijski zahtevnih duplikata (npr. tekstualni objekti), štedi se operativna memorija računara. Podliste se, slično tekstualnim sekvencama, mogu izdvojiti navođenjem opsega oblika `[i:j]`:

```
>>> x = [1, 3, 5, 7, 9]
>>> x[0:2]     # izdvajanje podliste od prva dva elementa
[1, 3]
>>> a = x[1:]  # podlista sa svim elementima osim prvog
>>> a
[3, 5, 7, 9]
>>> x[1::2]    # od 2. elem. pa do kraja, svaki drugi
[3, 7]
>>> x_kopija = x[:]
>>> x_kopija
[1, 3, 5, 7, 9]
>>> print(x_kopija == x, x_kopija is x)
True False
```

Ako se, prilikom izdvajanja, rezultujuća podlista dodeljuje promenljivoj na levoj strani jednakosti, onda se od elemenata podliste formira *nova* lista tako što se *kopiraju* odgovarajuće objektne reference (slika 5.6). Kada se za opseg navede `[:]`, vrši se kopiranje *cele* liste, kao u slučaju `x` i `x_kopija`.

Utvrđivanje *ekvivalentnosti* dve liste (isti elementi na istim mestima), obavlja se pomoću operatora ==. Očigledno je da posle kopiranja `x` u `x_kopija`, obe liste sadže iste elemente na istim mestima, ali da su to dva različita objekta, što se vidi iz poređenja na kraju primera.

Za razliku od dosadašnjih objektnih tipova, liste mogu *menjati* svoj sadržaj u toku rada programa pa ih to čini fleksibilnom strukturom podataka za različite primene:



Slika 5.6: Izdvajanje podliste i dodela vrednosti: prilikom izdvajanja se kopiraju izabrane objektne reference pa podlista ukazuje na iste objekte.

```
>>> x=['Pera', 'Mika', 'Laza', 'Jovan']
>>> x[0]='Ana'  # promena elementa
>>> x
['Ana', 'Mika', 'Laza', 'Jovan']
>>> x[1:3] = [1 , 2]          # promena više elemenata
>>> x
['Ana', 1, 2, 'Jovan']
>>> x.append(100)             # dodavanje na kraj
>>> x
['Ana', 1, 2, 'Jovan', 100]
>>> x.insert(1, [1,2])        # umetanje elementa
>>> x
['Ana', [1, 2], 1, 2, 'Jovan', 100]
>>> del x[1]                  # brisanje elementa
>>> x
['Ana', 1, 2, 'Jovan', 100]
>>> del x[1:3]                # brisanje više elemenata
>>> x
['Ana', 'Jovan', 100]
```

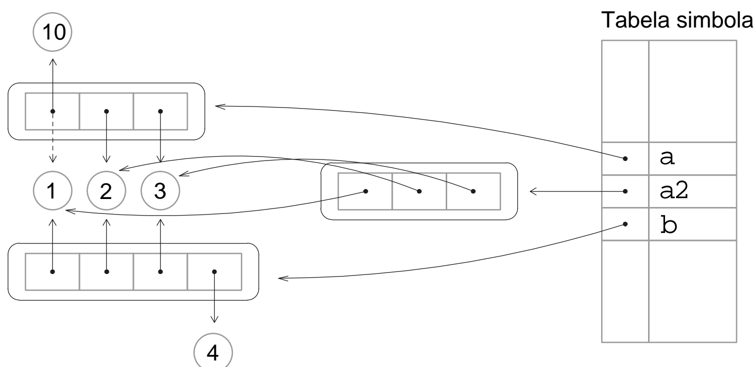
Pored ažuriranja vrednosti na pojedinačnoj poziciji u listi, moguće je promeniti i veći broj članova odjednom. U tom slučaju, na levoj strani dodele vrednosti, navodi se podlista čiji elementi se zamenjuju odgovarajućim objektima iz sekvence na desnoj strani (`x[1:3]=[1,2]`). Za proširivanje liste, *dodavanjem* novog elementa na njen kraj, koristi se metoda `append()` iz objektnog tipa `list`. Kao i kod tekstualnih metoda, ona se poziva *nad* listom koja se proširuje uz upotrebu notacije sa tačkom.

Kada se novi element *ubacuje* na određenu poziciju u listi, koristi se metoda `insert()` koja zahteva unošenje dva parametra: indeks pozicije za umetanje i novi element. Tom prilikom se svim postojećim elementima, počevši od pozicije za umetanje pa nadesno, indeksi uvećavaju za jedan. Za *izbacivanje* elementa iz liste koristi se operator `del`. Gornji primer pokazuje kako se izbacuje jedan ili više elemenata, pri čemu se indeksi odgovarajućih desno pozicioniranih elemenata smanjuju za jedan.

Liste se, kao i tekstualne sekvence, spajaju i umnožavaju pomoću operatora + i $*$:

```
>>> a = [1, 2, 3]
>>> b = a + [4]
>>> print(a, b, 2*a)
[1, 2, 3] [1, 2, 3, 4] [1, 2, 3, 1, 2, 3]
>>> a2 = a[:]  # kopiranje liste
>>> a[0] = 10
>>> print(a, a2)
[10, 2, 3] [1, 2, 3]
```

Prilikom spajanja dve liste ne vrši se proširivanje postojećih, već se formira *nova*, rezultujuća lista. Isto važi i prilikom umnožavanja. Poslednji redovi gornjeg primera ilustruju da, kada se prva lista po kopiranju modifikuje (`a[0]=10`), to nema uticaja na drugu listu (`a2`). Na slici 5.7 prikazani su objekti iz primera u postupku spajanja i kopiranja.



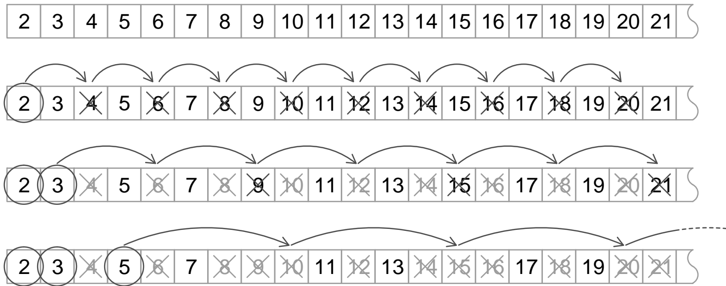
Slika 5.7: Spajanje i kopiranje listi. Prilikom ovih operacija kreiraju se nove liste.

Problem 5.3 — Prosti brojevi. Formirati listu svih prostih brojeva do prirodnog broja n. Prirodan broj je prost ako je deljiv samo jedinicom i samim sobom. Najmanji prost broj je 2, a u prvih deset prirodnih brojeva to su još 3, 5 i 7. ■

Postoji više različitih načina za generisanje sekvence prostih brojeva. Ovde će biti izložen metod koji se pripisuje grčkom matematičaru i geografu Eratostenu.[10] Postupak se može ilustrovati na primeru određivanja svih prostih brojeva do 20 (slika 5.8). Po formiranju liste prirodnih brojeva od 2 do 20, izabere se broj dva kao prvi prost broj iz sekvence. Potom se svaki drugi broj iz liste, budući da je umnožak broja dva, označi kao broj koji nije prost. Može se zamisliti da su brojevi iz početne liste ubačeni u *sito*, te da svi označeni (precrtani) složeni brojevi propadaju kroz sito u prvom prolazu.

[10] Eratosten (276. do 195. godine p. n. e.) je poznat po tome što je prvi odredio obim Zemlje.

U sledećem prolazu bira se broj tri, prvi prost broj veći od dva koji je ostao u situ. Postupak filtriranja ponavlja se tako što se svaki umnožak broja tri označava kao broj koji nije prost. U trećem prolazu, sledeći prost broj iz sita je broj pet. Sada se označavaju svi umnošci broja pet.



Slika 5.8: Eratostenovo sito za proste brojeve do 21. Zaokruženi brojevi ostaju u situ kao prosti. Precrtani brojevi su filtrirani jer predstavljaju celobrojne umnoške prostih brojeva levo od sebe.

U opštem slučaju, na početku svake iteracije filtriranja, izabrani broj je prost jer *nije deljiv* ni sa jednim *manjim* prostim brojem iz liste (da jeste, bio bi označen u nekoj od prethodnih iteracija kao umnožak manjeg prostog broja). Postupak se ponavlja sve dok u listi više ne bude prostih brojeva koji se mogu izabrati (svi preostali brojevi označeni su kao složeni). Sledi realizacija opisanog postupka:

```
def eratosten(n):
    ''' generiše listu prostih brojeva do n'''
    u_situ = [False, False] + (n-1)*[True]
    prosti = []

    for p in range(2, n+1):
        if u_situ[p]:
            prosti.append(p)
            for i in range(2*p, n+1, p):
                u_situ[i] = False
    return prosti

n = int(input('n '))
print(eratosten(n))
```

Sito se realizuje putem liste logičkih vrednosti `u_situ` (r3), koja na poziciji `i` ima vrednost `True` ako je broj `i` trenutno u situ. U početku, svi brojevi od 2 do n nalaze se u situ. Lista `prosti`, koja je inicijalno prazna (r4), na kraju sadrži sve tražene proste

brojeve. Petlja po p testira *kandidate* za proste brojeve iz opsega $[2, n]$ (r6-10). Ako je broj p prost (nalazi se u situ), ubacuje se u `prosti` (r8) uz pomoć metode `append()`, a iz sita se izbacuju svi njegovi celobrojni umnošci (r9-10). Rezultat rada programa, za $n = 100$, dat je u sledećem prikazu:

```
n 100
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59,
61, 67, 71, 73, 79, 83, 89, 97]
```

Prethodno rešenje može se ubrzati ako se, uz pomoć slike 5.8, uoči sledeće: kada se označe svi umnošci broja 5, onda ne treba procesirati preostale neoznačene brojeve jer su i oni prosti (7, 11, 13, 17, 19). U opštem slučaju, petlja po p, umesto do n, može da ide do $\lceil \sqrt{n} \rceil$ (prvi veći ceo broj u odnosu na $\sqrt{n}$). Da bi se ovo poboljšanje pojasnilo treba dokazati sledeće: svaki broj x koji *nije* prost može se zapisati kao $x = ab$, gde je $a \leqslant b$ prost broj za koji važi $a \leqslant \sqrt{x}$. Prema *fundamentalnoj teoremi aritmetike*, svaki složeni broj može se zapisati kao proizvod dva ili više ne nužno različitih prostih faktora. Otuda je $x = p_1 \cdot p_2 \cdot \ldots \cdot p_n$, pri čemu je $p_1 \leqslant p_2 \leqslant \ldots \leqslant p_n$. Očigledno, mora biti $p_1 \leqslant \sqrt{x}$ jer bi u suprotnom svi faktori bili veći pa bi i prozvod bio veći od x. Zaista, dovoljno je označiti (filtrirati) umnoške svih prostih brojeva koji su manji od $\lceil \sqrt{n} \rceil$, kako bi u situ ostali samo prosti brojevi manji ili jednaki broju n.

Iz slike 5.8 uočava se da su neki brojevi označeni više puta (u više iteracija filtriranja). Na primer, 15 je označen kao umnožak prostih brojeva dva i tri. Kako bi se smanjila nepotrebna obeležavanja složenih brojeva (r9-10), petlja se može modifikovati tako da, umesto od 2*p, pođe od p*p jer su manji umnošci već obeleženi u ranijim iteracijama. Optimizovani algoritam realizovan je na sledeći način:

Program 5.3 — Prosti brojevi.

```
1  from math import ceil
2  def eratosten_opt(n):
3      ''' generiše listu prostih brojeva do n'''
4      u_situ = [False, False] + (n-1)*[True]
5      prosti = []
6
7      for p in range(2, ceil(n**0.5) + 1):
8          if u_situ[p]:
9              for i in range(p*p, n+1, p):
10                 u_situ[i] = False
11
12     for i in range(2, n+1):
13         if u_situ[i]:
```

```
14            prosti.append(i)
15    return prosti
```

U r7 koristi se funkcija `ceil()` iz modula `math` (uočiti oblik naredbe `import` u r1). Ona vraća *prvi veći* ceo broj u odnosu na argument. Treba uočiti različitu prirodu listi `u_situ` i `prosti`. Prva lista ima unapred definisanu dužinu, dok se druga *dinamički* formira u toku rada programa. Broj prostih brojeva nije poznat unapred, već zavisi od ulaznog parametra n. Budući da mogu da se skupljaju i proširuju, kolekcije poput liste savršeno odgovaraju ovakvom scenariju.

! U većini programskih jezika postoji struktura podataka pod imenom *niz*. Za razliku od liste, niz ima unapred poznat broj elemenata istog tipa koji se *mora* navesti prilikom njegovog definisanja u programu. Nizovi mogu biti i višedimenzionalni. U inženjerskim primenama često se koriste dvodimenzionalne varijante – *matrice*. Nizovi se u Pajtonu mogu lako realizovati upotrebom liste. Višedimenzionalni nizovi simuliraju se ugnježđenim listama. U glavi 11, razmatra se naučno-tehnički paket *NumPy* koji omogućava jednostavan rad sa višedimenzionalnim nizovima.

Notacija za formiranje liste po unapred poznatom pravilu

Često je potrebno formirati listu elemenata koji zadovoljavaju *unapred poznatu* zakonitost. Na primer, neka je potrebno formirati listu L_1 koja sadrži prvih n kvadrata prirodnih brojeva, kao i listu L_2 sa celim brojevima od a do b koji u svom dekadnom zapisu sadrže cifru 7. Pajton nudi moćan mehanizam koji omogućava kreiranje liste po unapred poznatom pravilu[11] na sledeći način:

```
>>> n = 5
>>> L1 = [i**2 for i in range(n)] # kvadrati prvih n prir. brojeva
>>> L1
[0, 1, 4, 9, 16]
>>> a = 10
>>> b = 75
>>> L2 = [i for i in range(a, b) if '7' in str(i)] # sa cifrom 7
>>> L2
[17, 27, 37, 47, 57, 67, 70, 71, 72, 73, 74]
```

U prvom slučaju lista se formira od celobrojnih objekata sa vrednošću `i**2`, pri čemu promenljiva `i` ukazuje na vrednosti od 0 do `n-1`. U drugom primeru definiše se uslov pod kojim se elementi ubacuju u listu (naredba `if` iza naredbe `for`). Proizvoljan broj *prevodi* se konstruktorom `str()` u svoju tekstualnu reprezentaciju kako bi se, uz

[11] Engl. *List comprehension*.

pomoć operatora in, testiralo da li se tražena cifra nalazi unutar odgovarajuće tekstualne sekvence. Pored dva navedena oblika formiranja liste po zadatom pravilu, postoje i složenije varijante koje se ovde neće navoditi. Mehanizam kreiranja liste po unapred poznatom pravilu može se upotrebiti da se prethodno rešenje za Eratostenovo sito dodatno uprosti:

```
1  from math import ceil
2  def eratosten_opt(n):
3      ''' generiše listu prostih brojeva do n'''
4      u_situ = [False, False] + (n-1)*[True]
5      for p in range(2, ceil(n**0.5) + 1):
6          if u_situ[p]:
7              for i in range(p*p, n+1, p):
8                  u_situ[i] = False
9
10     return [i for i in range(2, n+1) if u_situ[i]]
```

Metode tipa list

Tip list, poput tekstualnih sekvenci, sadrži metode koje se mogu pozivati nad objektima listama u notaciji sa tačkom. Pored već navedenih append() i insert(), u tabeli 5.2 pobrojane su najčešće korišćene metode u okviru ovog tipa.

metoda	opis
l.append(e)	dodaje element e na kraj liste l
l.insert(i, e)	ubacuje element e na poziciju i u listi l
l.pop()	vraća i uklanja poslednji element iz liste l
l.remove(e)	uklanja prvo pojavljivanje elementa e iz liste l
l.count(e)	vraća broj pojavljivanja elementa e u listi l
l.index(e)	vraća indeks prvog pojavljivanja elementa e u listi l
l.reverse()	obrće redosled elemenata u listi l
l.sort()	sortira listu l po rastućim (opadajućim) vrednostima elemenata

Tabela 5.2: Često korišćene metode tipa list.

Metode append(), insert(), remove(), reverse() i sort() vraćaju None pa treba paziti da se *ne koriste* u dodelama oblika lista = lista.metod():

```
>>> parni = [2, 6, 2, 4]
>>> parni = parni.append(22) # nepravilno, podaci se gube!
>>> print(parni)
None
>>> parni = [2, 6, 2, 4]
>>> parni.append(22) # pravilno
>>> print(parni)
[2, 6, 2, 4, 22]
```

Navedene metode modifikuju listu nad kojom se pozivaju. Metode koje modifikuju objekat nad kojim se pozivaju nazivaju se *mutatori*. Upotreba metoda iz tabele 5.2 ilustrovana je sledećim primerom:

```
>>> parni = [2, 6, 2, 4, 22, 22]
>>> parni.count(22)
2
>>> parni.index(2)
0
>>> parni.remove(22)
>>> parni
[2, 6, 2, 4, 22]
>>> poslednji = parni.pop()
>>> print(poslednji, parni)
22 [2, 6, 2, 4]
>>> parni.reverse()
>>> parni
[4, 2, 6, 2]
>>> parni.sort()
>>> parni
[2, 2, 4, 6]
>>> parni.sort(reverse=True) # opadajuci poredak, podrazumevano False
>>> parni
[6, 4, 2, 2]
```

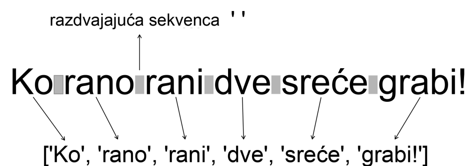
5.1.4 Tekstualne metode i liste

Vrlo često softverski sistemi koriste se u analizi i obradi prirodnih jezika. Tipičan primer sistema iz ove klase je prevodilac iz jezika J_a u jezik J_b, poput *Google Translate*-a[12]. Jedan od osnovnih zadataka u ovoj oblasti jeste rastavljanje rečenice prirodnog jezika na sastavne delove - reči. Programi koji obavljaju ovu i još neke srodne funkcije zovu se *parseri*. Tip `str` poseduje metode `split()` i `join()` koje rastavljaju tekst na sastavne fragmente, odnosno spajaju fragmente teksta u jedinstvenu celinu. U oba slučaja koriste se liste kao uređene kolekcije tekstualnih fragmenata:

[12] `http://translate.google.com`

```
>>> a = "Ko rano rani dve   sreće grabi!"
>>> a.split()
['Ko', 'rano', 'rani', 'dve', 'sreće', 'grabi!']
>>> a.split('a')
['Ko r', 'no r', 'ni dve   sreće gr', 'bi!']
>>> b = a.split(' ')
>>> b
['Ko', 'rano', 'rani', 'dve', '', '', 'sreće', 'grabi!']
>>> ' '.join(b)
'Ko rano rani dve   sreće grabi!'
```

Kada se metoda `split()` pozove nad tekstualnim objektom bez argumenata, formira se lista od svih fragmenata teksta koji ne sadrže bele karaktere. Metoda *ne deli* rečenicu na reči jer pojedini fragmenti mogu obuhvatiti i interpunkcijske znake (`'grabi!'`). Ako se `split()` pozove sa tekstualnim argumentom (2. i 3. poziv), podela se vrši tako što se navedeni tekst posmatra kao *razdvajajuća sekvenca* koja *ne pripada* ni jednom fragmentu (slika 5.9).



Slika 5.9: Podela teksta na fragmente: ukoliko se dve razdvajajuće sekvence nađu jedna pored druge, onda se na tom mestu generiše prazan tekst. Na primer, `'1aa2a3'.split('a')` vraća listu `['1', '', '2', '3']`.

Metoda `join()` poziva se nad tekstualnim objektom koji treba da posluži kao *spajajuća sekvenca*. Ona ima suprotan efekat u odnosu na metodu `split()`: kao što se vidi iz poslednjeg poziva u primeru, iz liste fragmenata konstruiše se polazni tekst.

Problem 5.4 — Reči u rečenici. Formirati listu svih reči za zadatu rečenicu. Reč ne sme da započne ili da se završi sa jednim ili više interpunkcijskih znakova. Pod interpunkcijskim znakovima podrazumeva se skup simbola { . , ! ? : ; ' " }. Na primer, rečenica *Tata kaže: "U laži su kratke noge!"*, transformiše se u sledeću listu reči: `['Tata','kaže','U','laži','su','kratke' ,'noge']`. ■

Problem se može rešiti dekompozicijom na dve funkcionalne celine. Funkcija `reči()`, za datu ulaznu sekvencu, formira listu regularnih reči. U njoj se koristi metoda `split()` i pomoćna funkcija `ukloni_interpunkciju()`. Pomoćna funkcija pronalazi indekse *prvog* neinterpunkcijskog znaka sleva i *poslednjeg* neinterpunkcijskog znaka zdesna, pa na osnovu njih izdvaja podtekst koji predstavlja regularnu reč.

Program 5.4 — Reči u rečenici.

```
def ukloni_interpunkciju(reč):
    ''' uklanja interpunkcijske znake sleva i sdesna'''
    interpunkcija = ['.', ',', '!', '?', ':', ';', '\'', '"']
    n = len(reč)

    levo = 0
    while levo < n and reč[levo] in interpunkcija:
        levo += 1

    desno = n - 1
    while desno > -1 and reč[desno] in interpunkcija:
        desno -= 1

    return reč[levo : desno + 1]

def reči(rečenica):
    ''' kreira listu reči iz rečenice '''
    reči = []
    for reč in rečenica.split():
        r = ukloni_interpunkciju(reč)
        if r:  # isto što i if r != '':
            reči.append(r)
    return reči

print(reči('Tata kaže: ''U laži su kratke noge!'''))
```

```
['Tata', 'kaže', 'U', 'laži', 'su', 'kratke', 'noge']
```

Razmatra se pomoćna funkcija. U r3, formira se lista koja sadrži interpunkcijske znakove iz specifikacije (`interpunkcija`). Leva i desna granica reči utvrđuju se u odgovarajućim petljama `while` (r7-8, 11-12). Prvo se ispituje da li se indeksi, koji označavaju odgovarajuće granice, nalaze u dozvoljenom opsegu (`levo` i `desno`), pa tek onda da li su elementi na koje oni ukazuju interpunkcijski znaci (r7, 11). Promena redosleda testiranja u r7 i r11 proizvela bi grešku za `levo == n`, odnosno `desno == -1`. Ako se reč sastoji *isključivo* od interpunkcijskih znakova, funkcija `ukloni_interpunkciju()` vraća prazan tekst (`levo` ima vrednost `n`, a `desno`, -1). Primetiti da se neprazan tekst tretira kao logička istina, a prazan kao `False` (r21).

> ! Redosled logičkih uslova u naredbama `if` i `while` često je bitan za ispravan rad programa! To se odnosi na situacije kada izraz postaje nedefinisan ako se njegove promenljive, koje mogu dovesti do problema, ne testiraju na vreme.

Liste kao argumenti funkcija i metoda. Sporedni efekat.

Kako je lista, za razliku od svih dosadašnjih tipova, objekat promenljive prirode, prilikom njenog prenošenja u funkciju (metodu) može doći do neželjenog sporednog efekta. Mehanizam prenošenja parametara u funkciju (metodu) zasniva se na principu *kopiranja objektnih referenci* (glava 4.1.4). Ovaj mehanizam čini da se ulazni parametri tretiraju kao lokalne promenljive. Pri prvoj dodeli vrednosti u kojoj se parametar nalazi na *levoj* strani, *raskida* se veza sa objektom na koji je referenca ukazivala, a novo ime ubacuje se u tabelu simbola imenskog prostora funkcije. Međutim, ako se objekat *promenljivog* tipa, poput liste, navede kao ulazni argument, onda se on *može* promeniti unutar funkcije *pre* eventualne dodele vrednosti što će izazvati sporedni efekat:

```
1 def f(lista):
2     lista.append(100) # sporedni efekat
3     lista = []        # lista se redefiniše kao lok. prom.
4     return lista
5
6 lista = [1, 2]
7 print('lista pre poziva', lista)
8 print('rezultat f(lista):', f(lista))
9 print('lista posle poziva', lista)
```

```
lista pre poziva [1, 2]
rezultat f(lista): []
lista posle poziva [1, 2, 100]
```

Da bi se eventualni sporedni efekat sprečio, pri navodjenju stvarnih parametara u pozivu funkcije, lista se može kopirati u novu – umesto `f(lista)`, u r8 treba staviti `f(lista[:])`.

5.1.5 Torke - tip `tuple`

U Pajtonu postoji mogućnost da se objekti proizvoljnih tipova urede unutar sekvence koja, za razliku od liste, *ne može* da se menja po formiranju. Nepromenljive sekvence nazivaju se *torkama*.[13] Iako može da sadrži objekte različitih tipova, listu u praksi

[13] Engl. *Tuples*.

obično čine objekti *iste* vrste čije različite vrednosti potiču iz *jedinstvenog* logičkog konteksta. Kada se podaci *različitih* tipova, koji potiču iz *istog* konteksta, žele objediniti u jedinstvenu strukturu, onda se najčešće koriste torke. Sledi primer:

```
>>> nole = ('Novak', 'Đoković', 1987, 'Srbija', 24)
>>> type(nole)
<class 'tuple'>
>>> nole
('Novak', 'Đoković', 1987, 'Srbija', 24)
>>> rafa = 'Rafael', 'Nadal', 1987, 'Španija', 22
>>> rafa
('Rafael', 'Nadal', 1987, 'Španija', 22)
>>> niko = () # može i niko = tuple()
>>> niko
()
>>> tuple('Pajton')
('P', 'a', 'j', 't', 'o', 'n')
>>> tuple([1, 3, 5])
(1, 3, 5)
>>> nepotpuna = ('Federer',)  #  samo jedan elem. - zapeta obavezna!
>>> nepotpuna
('Federer',)
```

Torke se formiraju navođenjem uređenog niza vrednosti odvojenih zapetama, sa (`nole`) ili bez oblih zagrada (`rafa`). Zagrade se najčešće koriste radi čitljivosti, a obavezne su kada se formira *prazna* torka (`niko`). Torka se može formirati i navođenjem konstruktora `tuple()`, pri čemu se, kao argument, navodi validna sekvenca. Ako torka sadrži samo jedan element neophodno je navesti zapetu (`nepotpuna`). Primer pokazuje da su torke upotrebljene kako bi se raznorodni podaci objedinili u jedinstveni objekat. Ovde se one odnose na isti kontekst – podaci o teniseru.

Za indeksiranje i operacije nad torkama važe pravila o kojima je bilo reči u delu o tekstualnim sekvencama i listama. Ako se pokuša promena na nekoj od pozicija, interpreter će prijaviti grešku kao i u slučaju tekstualne sekvence:

```
>>> nole[4] = 25
Traceback (most recent call last):
  File "<pyshell#52>", line 1, in <module>
    nole[4] = 25
TypeError: 'tuple' object does not support item assignment
```

Da bi se broj Novakovih grand slam titula povećao, treba formirati *novu* torku sa istim vrednostima na svim pozicijama, osim na poslednjoj:

```
>>> nole_posle_2023 = nole[0:3] + (25,)
>>> nole_posle_2023
('Novak', 'Đoković', 1987, 'Srbija', 25)
```

Torke se često koriste za *grupisanje* više povratnih vrednosti iz funkcije, kao što je ilustrovano u sledećem primeru:

Problem 5.5 — Elementarna statistika. Izračunati srednju vrednost i standardno odstupanje uzorka slučajne promenljive x, određenog zadatim nizom realnih brojeva. ■

Srednja vrednost uzorka x data je sa $\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$, a uzoračko standardno odstupanje sa $s_x = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2}$. U slučaju praznog uzorka ($n = 0$), srednja vrednost i standardno odstupanje nisu definisani. Za $n = 1$, srednja vrednost jednaka je jedinom elementu iz uzorka, a standardno odstupanje je nula.[14] Nedefinisane vrednosti biće predstavljene sa `None`. Vrednosti za $\bar{x}$ i s_x izračunavaju se u iterativnim postupku:

Program 5.5 — Elementarna statistika.

```
1  def statistika(uzorak):
2      '''Vraća srednju vrednost i standardno odstupanje uzorka'''
3      so, n = 0, len(uzorak)
4      if n == 0:
5          return (None, None) # nema smisla
6      elif n == 1:
7          return (uzorak[0], 0)
8      else:
9          # srednja vrednost
10         sv = sum(uzorak) / n
11         # uzoračko standardno odstupanje
12         for x in uzorak:
13             so += (x - sv)**2
14         so = (so / (n - 1))**.5
15         return (sv, so)
16 # test
17 x = [1, 2, 3, 4, 5, 4, 3, 2, 1]
18 stat = statistika(x)
19 print(x)
20 print('srednja vrednost', stat[0])
```

[14] Iako po formuli nije definisano, intuitivno je nula – varijacija konstante.

```
21 print('standardno odstupanje', stat[1])
```

```
[1, 2, 3, 4, 5, 4, 3, 2, 1]
srednja vrednost 2.7777777777777777
standardno odstupanje 1.3944333775567928
```

U r3, uz pomoć naredbe višestruke dodele vrednosti, torka sa desne strane dodeljuje se promenljivim sa leve strane. Da bi dodela uspela, potrebno je da broj elemenata u desnoj torci odgovara broju promenljivih sa leve strane. Srednja vrednost uzorka izračunava se u r10 uz pomoć ugrađene funkcije `sum()`. Ona računa sumu elemenata u proizvoljnoj sekvenci, uz uslov da *svi* elementi predstavljaju objekte brojnih tipova. Tražena statistika iznosi se iz funkcije u obliku torke (r5, 7, 15), dok se, u r20 i r21, pristupa pojedinim komponentama rezultata. Rezultat se može vratiti i tako što se pojedine komponente rezultujuće torke *imenuju* putem višestruke dodele vrednosti:

```
1 # test
2 x = [1, 2, 3, 4, 5, 4, 3, 2, 1]
3 sr_vrednost, odstupanje = statistika(x) # imenovanje
4 print(x)
5 print('srednja vrednost', sr_vrednost)
6 print('standardno odstupanje', odstupanje)
```

Mehanizam *raspakivanja* proizvoljne sekvence uz pomoć višestruke dodele vrednosti podrazumeva imenovanje svih elemenata ponaosob, uz uslov da se raspakuje kompletna sekvenca kako ne bi došlo do greške:

```
>>> a, b, c = "ABC"
>>> print(a, b, c)
A B C
>>> a, b, c = [10, 15, True]
>>> print(a, b, c)
10 15 True
>>> a, b, c = [1, 2, 3, 4] # greška
Traceback (most recent call last):  ...
ValueError: too many values to unpack (expected 3)
```

O operatorima za raspakivanje `*` i `**` biće više reči na kraju glave 5.2.2.

5.2 Skupovi i rečnici

Skupovi i rečnici predstavljaju *neuređene* kolekcije objekata. Za razliku od sekvenci, u neuređenoj kolekciji *ne pamti* se redosled ubacivanja pa zbog toga duplikati *nisu* dozvoljeni. Dok se u slučaju *skupa*, koji predstavlja apstrakciju *matematičkog skupa*, elementima ne može pristupati po *unapred* poznatoj šemi, *rečnik* nudi mehanizam pristupanja preko objekta *ključa*. Veza između ključa i objekta *pridruženog* ključu predstavljena je uređenim parom koji se skladišti u rečniku. Redosled pojedinih pridruživanja ne igra nikakvu ulogu u pogledu uređenosti rečnika. Skup se realizuje pomoću objekta tipa `set`, dok se rečnik[15] realizuje objektom tipa `dict`.

5.2.1 Skupovi - tip `set`

Skup predstavlja neuređenu kolekciju objekata *nepromenljivih* tipova koja se može proizvoljno ažurirati po kreiranju. Sledeći primer ilustruje formiranje skupova upotrebom vitičastih zagrada, kao i navođenjem konstruktora `set()`:

```
>>> teniseri = {'Nole', 'Rafa', 'Fed', 'Tipsa'}
>>> type(teniseri)
<class 'set'>
>>> print(teniseri, 'ima', len(teniseri), 'tenisera')
{'Rafa', 'Nole', 'Tipsa', 'Fed'} ima 4 tenisera
>>> 'Nole' in teniseri
True
>>> babe_i_zabe = { 'a', 1, 2, 3,
                    3, 'a', (1, 2)}
>>> babe_i_zabe
{(1, 2), 1, 2, 3, 'a'}
>>>
>>> problematican = { [], [1, 2]} # greška! Liste su prom. objekti
Traceback (most recent call last):
  File "<pyshell#76>", line 1, in <module>
    problematican = { [], [1, 2]}
TypeError: unhashable type: 'list'
>>>
>>> neparni = set(range(1,10,2))
>>> neparni
{1, 3, 5, 9, 7}
>>> samoglasnici = set('aeieou')
>>> samoglasnici
{'i', 'e', 'a', 'u', 'o'}
>>> boje = [ 'crvena', 'plava', 'plava', 'crvena']
>>> set(boje)
```

[15] Engl. *Dictionary*. Rečnik se naziva još i *mapa*. Pojam mapa potiče od engleske reči *mapping* koja znači *preslikavanje* (ključeva u vrednosti).

```
{'crvena', 'plava'}
>>> prazan = set()
>>> prazan
set()
```

Poredak elemenata pri ispisivanju skupova `teniseri` i `babe_i_zabe` razlikuje se od poretka u naredbi dodele vrednosti - redosled ubacivanja je nebitan! Funkcija `len()` vraća broj elemenata, a operator `in` testira pripadnost skupu. Prilikom formiranja skupa `babe_i_zabe`, duplikati se automatski uklanjaju (`'a'` i `3`). Pri pokušaju formiranja skupa koji bi uključivao i objekte nekog od promenljivih tipova, poput liste, interpreter prijavljuje grešku (`problematican`).

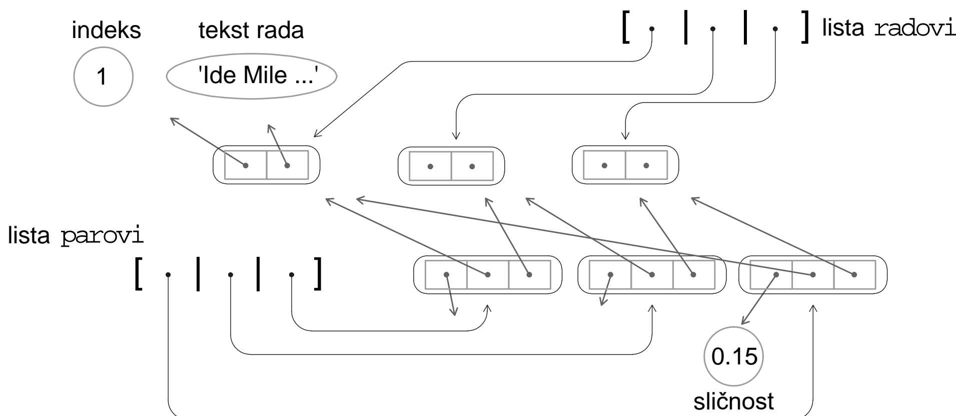
Kada se koristi konstruktor tipa, kao argument se može proslediti bilo koja do sada razmatrana sekvenca. Duplikati iz sekvence automatski se uklanjaju iz rezultujućeg skupa. Prazan skup moguće je formirati *jedino* uz pomoć konstruktora tipa, pošto se notacijom `{}` označava i *prazan rečnik* (glava 5.2.2)!

Dva skupa porede se po jednakosti korišćenjem operatora ==, a njihov odnos utvrđuje se pomoću <= (podskup), < (pravi podskup), >= (nadskup) ili > (pravi nadskup). Pored metoda za dodavanje i uklanjanje elemenata, podržane su i standardne skupovne operacije. Ove operacije ilustrovane su u sledećem primeru:

```
>>> {'crvena', 'plava'} == {'plava', 'crvena'}
True
>>> a, b, c = {1, 2, 3, 4, 5}, {3, 4, 5}, {1, 2}
>>> c.add(3)      # metoda za ubacivanje elementa u skup
>>> c
{1, 2, 3}
>>> c.discard(3)  # uklanja element ako postoji
>>> c
{1, 2}
>>> a & b         # presek skupova
{3, 4, 5}
>>> b & c         # presek je prazan
set()
>>> b | c         # unija skupova
{1, 2, 3, 4, 5}
>>> a - b         # razlika skupova
{1, 2}
>>> a >= b        # a nadskup b
True
>>> b <= a        # b podskup a
True
>>> a.clear()     # uklanja sve elemente iz skupa
>>> a
```

```
set()
```

Problem 5.6 — Prepisivanje na ispitu. Studenti pišu sastav na zadatu temu. Profesor Crnjanski je, po čitanju, primetio da su neki od njih veoma slični. Pomozite profesoru da rangira sve parove studentskih radova po tome koliko su međusobno slični. Pretpostaviti da se svaki studentski rad nalazi u torci oblika (*broj_indeksa*, *tekst_rada*). Sičnost između dva teksta izraziti pomoću broja zajedničkih reči. ■



Slika 5.10: Detekcija sličnosti studentskih radova: lista *radova* sadrži tri uređena para kojima su predstavljena tri rada. Lista *parova radova* sadrži tri moguće kombinacije radova (prvi i drugi, prvi i treći, drugi i treći). Kombinacije su predstavljene uređenim trojkama, gde se prve komponente odnose na stepen sličnosti, a druge dve na odgovarajuće radove iz liste radova.

Sličnost dva teksta može se posmatrati kroz broj *istih* reči koje tekstovi dele. Ako se skup svih različitih reči iz teksta A poklapa sa odgovarajućim skupom reči iz teksta B, onda je sličnost maksimalna. U suprotnom, kada skupovi nemaju istih elemenata, odnosno kada je njihov *presek* prazan skup, sličnost je minimalna. Otuda se, za dati problem, može upotrebiti *Žakarov* (Jaccard) indeks sličnosti skupova:

$$J_{AB} = \frac{|A \cap B|}{|A \cup B|} \tag{5.1}$$

Veličina $|A|$ iz (5.1) označava broj elemenata skupa A. Mera je pogodna zbog svoje *normiranosti*: $0 \leq J_{AB} \leq 1$ (0 - nema sličnosti, 1 - jednaki skupovi).

Pretpostaviti da funkcija `sličnost(A, B)` vraća stepen sličnosti između dva ulazna teksta prema (5.1). Rešavanje problema svodi se na formiranje liste svih *parova radova* iz liste svih radova. Parovi radova mogu se predstaviti tročlanim torkama (*stepen_sličnosti*, *prvi_rad_u_paru*, *drugi_rad_u_paru*). Pri formiranju liste parova treba izračunati sličnost između svaka dva rada koji čine par, pa tako dobijenu listu *sortirati* na osnovu mere sličnosti. Sortiranje liste podrazumeva preuređivanje redosleda

njenih elemenata tako da su oni poređani po rastućim (ili opadajućim) vrednostima osobine po kojoj se vrši sortiranje. Parovi se mogu generisati pomoću dve ugnježđene brojačke petlje čiji indeksi ukazuju na radove iz liste radova. Potrebne strukture podataka date su na slici 5.10.

Naglašava se upotreba torki koje memorišu dve različite *strukture podataka*: radove i parove. Ovakvim izborom pojednostavljuje se sortiranje liste parova prema sličnosti radova jer se, pri sortiranju liste metodom `sort()`, torke porede po vrednosti *prve* komponente koja u ovom slučaju predstavlja stepen sličnosti. Sledi implementacija:

Program 5.6 — Prepisivanje na ispitu.

```
def sličnost(t1, t2):
    '''Vraća meru sličnosti dva teksta putem
       Jaccard-ovog indeksa sličnosti skupova reči'''
    reci_1 = set(t1.lower().split()) # reči iz t1
    reci_2 = set(t2.lower().split()) # reči iz t2
    n = len (reci_1 | reci_2)        # broj svih razl. reči
    if n == 0:
        return 1.0
    else:
        return len(reci_1 & reci_2) / n

def detektor_sličnosti(radovi):
    '''Vraća listu parova sortiranu na osnovu
       sličnosti tekstova u listi radova. '''
    parovi = []
    n = len(radovi)
    for i in range(n-1):
        for j in range(i+1, n):
            ri, rj = radovi[i], radovi[j] # videti sliku 5.10
            parovi.append((sličnost(ri[1], rj[1]), ri, rj))
    parovi.sort(reverse = True) # opadajuća sličnost, može i:
                                # parovi.sort(), pa parovi.reverse()
    return parovi

# test
radovi = [ (1, 'Ide Mile lajkovačkom prugom'),
           (2, 'Ide Mile prugom ka svojoj supruzi'),
           (3, 'Kaži Mile supruzi šta te to muči'),
           (4, "Al' je lep ovaj svet") ]

```

```
31 print('Mera sličnosti\tIndeks 1\tIndeks 2')
32 for par in detektor_sličnosti(radovi):
33     print(f'{par[0]:<14.3}\t{par[1][0]:^8}\t{par[2][0]:^8}')
```

Prilikom računanja sličnosti dva ulazna teksta, prvo se formiraju skupovi svih različitih reči upotrebom konstruktora `set()` (r4, 5). Tom prilikom vrši se *ulančavanje* poziva više metoda (`t1.lower().split()`). Pošto prva metoda vrati objekat određenog tipa, nad njim se može *nadovezati* poziv sledeće metode, ako je ona iz *istog* tipa. Metoda `lower()`, iz tipa `str`, vraća novu tekstualnu sekvencu koja, u odnosu na polaznu, sadrži sva mala slova. Nad vraćenim tekstom poziva se metoda `split()` koja razbija tekst na reči. Ovde je pretpostavljeno da tekst *ne sadrži* znake interpunkcije. U suprotnom, može se koristiti neka naprednija funkcija za podelu na reči, poput one iz problema 5.4. Tekst je ”spušten” kako se, pri ispitivanju sličnosti, ne bi pravila razlika između različito napisanih reči koje imaju isto značenje (na primer: Car, car, CAR).

Postupak u kome se reprezentacija podatka svodi na *osnovni oblik*, pri čemu se ne gubi na njegovom značenju, naziva se *kanonizacija*. U analizi teksta, uobičajena praksa je da se tekst kanonizuje kako bi se smanjila kompleksnost problema koji se razmatra. Na primer, prilikom pretrage teksta po ključnoj reči koju zadaje korisnik, ciljni tekst i ključna reč se transformišu u zapis sa malim slovima.

Prilikom računanja sličnosti, prvo se izračunava broj elemenata u uniji dva skupa reči (r6). Ako se radi o *praznim* tekstovima (r7), onda je njihova sličnost jednaka jedinici (r8). Testiranje u r7 je *neophodno* kako pri računanju izraza (5.1) ne bi došlo do deljenja nulom. Potom se računa broj elemenata u preseku dva skupa reči i vraća odgovarajuća vrednost Žakarovog indeksa (r10). Sve kombinacije parova formiraju se u dvostrukoj petlji `for` (r19). Tom prilikom, svakom paru pridružuje se stepen sličnosti, a formirana kombinacija (uređena trojka) ubacuje se u listu `parovi` (r20). Kako bi se utvrdilo da li pojedini studenti “sarađuju” na ispitu, treba sortirati sve parove radova po stepenu sličnosti u opadajući poredak (r21). Zapaziti da se za opadajući poredak koristi opcioni parametar `reverse = True`. Sada je moguće brzo uočiti sumnjive situacije.

U r31 i r33 realizuje se ispis pregledne tabele sa tri kolone (mera sličnosti, prvi i drugi indeks para). Zaglavlje tabele formira se uz upotrebu tabulatora `\t` kojim se odvajaju naslovi kolona (r31). Redovi tabele oblikuju se uz pomoć formatirajućeg stringa (r33). Širine sva tri polja odgovaraju dužinama naslova svake kolone. Mera sličnosti poravnava se nalevo, a brojevi indeksa centralno. Brojna vrednost za sličnost zaokružena je na tri decimale. Rezultat programa za testiranje ukazuje da su studenti sa indeksima 1 i 2 *možda* nedozvoljeno sarađivali pa ih treba pozvati na usmeni ispit:

```
Mera sličnosti Indeks 1 Indeks 2
0.429              1        2
```

```
0.182          2        3
0.1            1        3
0.0            3        4
0.0            2        4
0.0            1        4
```

> **!** Prethodni primer ukazuje na važno pravilo u programiranju: izborom pogodne reprezentacije podataka, kojom se problem modeluje, može se značajno olakšati postupak njegovog rešavanja. Prilikom razmatranja složenosti pojedinih algoritama, u glavi 7, pokazaće se da izbor strukture podataka bitno utiče i na brzinu rada programa.

5.2.2 Rečnici - tip `dict`

Rečnik predstavlja neuređenu, promenljivu kolekciju, koja skladišti parove objekata *ključ* - *vrednost*, pri čemu ključ mora biti objekat *nepromenljivog* tipa.[16] Za razliku od sekvenci, gde se elementu pristupa preko indeksa pozicije, rečnici omogućavaju pristup preko ključa. Na primer, umesto da se traži element sa indeksom 10, rečnik omogućava da se dobije vrednost koja je *pridružena* ključu `'Beograd'`. U slučaju rečnika sa pridruživanjima oblika *glavni grad* → *broj stanovnika*, mogao bi se dobiti celobrojni objekat sa vrednošću `1 233 796`.[17] Za ključ `'Moskva'`, rečnik daje `10 563 038`.[18] Primer rečnika i njegove organizacije dat je na slici 5.11.

telefonski_imenik

ključ		vrednost
Voja Kešelj	→	123 1234
Aca Vuklić	→	123 4321
Vuk Eremić	→	221 2121
Luka Jankić	→	221 2121

Slika 5.11: Rečnik `telefonski_imenik` sa pridruživanjima oblika *ime* → *telefon*.

Rečnik se formira upotrebom vitičastih zagrada ili uz pomoć konstruktora `dict()`. U prvom slučaju parovi se odvajaju zapetom, a ključ od vrednosti odvaja dvotačkom:

```
>>> kurs = {'EUR':123, 'USD':117, 'CHF':116.3}
>>> type(kurs)
<class 'dict'>
```

[16] Nadalje će se objekat ključ nazivati samo *ključ*, a objekat vrednost – *vrednost*.
[17] Prema popisu iz 2011. godine.
[18] Prema popisu iz 2010. godine.

```
>>> print(kurs, 'ima', len(kurs), 'pridruživanja' )
{'USD': 117, 'CHF': 116.3, 'EUR': 123} ima 3 pridruživanja
>>> prazan = {}   # nije prazan skup već rečnik!
>>> prazan
{}
>>> recept = dict([('jaja', 2), ('brašno (gr)', 100),
                   ('voda (l)', 0.5)])
>>> recept
{'jaja': 2, 'voda (l)': 0.5, 'brašno (gr)': 100}
>>> ocene = dict(pera=10, mika=6, laza=6)
>>> ocene
{'pera': 10, 'laza': 6, 'mika': 6}
```

Rečnici `kurs`, `recept` i `ocene` ilustruju neuređenu prirodu rečnika čiji se redosled pridruživanja pri ispisu razlikuje u odnosu na redosled pri kreiranju. Obratiti pažnju da se, navođenjem zagrada `{}`, formira prazan rečnik a ne prazan skup! Prilikom navođenja konstruktora `dict()`, za argument se navodi lista torki koje predstavljaju parove ključ - vrednost (`recept`). Kada se ključevi navode putem imenovanih parametara, kao u primeru sa ocenama, onda oni treba da predstavljaju *regularna* imena za promenljive. Ako bi ključ, umesto imena, bio broj indeksa (npr. 121/16), tada bi se morao koristiti neki od prethodno pomenutih načina za kreiranje rečnika (jer 121/16 ne predstavlja regularno ime promenljive). Sledeći primer ilustruje *ubacivanje* novog i *ažuriranje* postojećeg pridruživanja:

```
>>> glavni_gradovi = {}
>>> glavni_gradovi['Srbija'] = 'Beograd'
>>> glavni_gradovi['Rusija'] = 'Moskva'
>>> glavni_gradovi
{'Rusija': 'Moskva', 'Srbija': 'Beograd'}
>>> glavni_gradovi['Srbija'] = ('Beograd', 1233796)
>>> glavni_gradovi
{'Rusija': 'Moskva', 'Srbija': ('Beograd', 1233796)}
```

Ubacivanje se obavlja putem naredbe dodele vrednosti tako što se, ispred imena rečnika, u uglastim zagradama navodi ključ, a sa desne strane jednakosti vrednost koja se za taj ključ vezuje. Ako se za vrednost koja se ubacuje navede ključ koji već postoji, onda se stara vrednost ažurira (prepisuje) novom. Pristupanje objektima u rečniku može se realizovati navođenjem ključa u uglastim zagradama ili uz pomoć metode `get()`:

```
>>> tabela = { 1 : 'Partizan', 2 : 'Zvezda', 3 : 'Rad' }
>>> prvi = tabela[1]
>>> prvi
'Partizan'
>>> 4 in tabela
```

```
False
>>> nema = tabela[4]
Traceback (most recent call last):
  File "<pyshell#49>", line 1, in <module>
    prvi = tabela[4]
KeyError: 4
>>> nema = tabela.get(4)
>>> nema == None
True
>>> tabela.get(4, 'Nema')
'Nema'
```

Ako se vrednostima pristupa navođenjem ključa u uglastim zagradama, *poželjno* je prvo proveriti da li željeni ključ postoji (`4 in tabela`). Kada ključ ne postoji u rečniku (kao u primeru), interpreter prijavljuje grešku. Eventualna greška može se izbeći korišćenjem metode `get()` koja vraća specijalnu vrednost `None` ako u rečniku ne postoji traženi ključ. Ipak, na programeru ostaje *odgovornost* da, u delu programa posle pristupanja, proveri da li je vraćena vrednost *validna* ili ne. Zato se često navodi i opcioni parametar koji se vraća kao rezultat ako traženog ključa nema u rečniku (poslednji poziv iz primera). Sledi primer koji ilustruje *brisanje* vrednosti iz rečnika:

```
>>> tabela = { 1 : 'Partizan', 2 : 'Zvezda',
               3 : 'Rad', 4 : 'Čukarički' }
>>> del tabela[1]
>>> tabela
{2 : 'Zvezda', 3 : 'Rad', 4 : 'Čukarički'}
>>> del tabela[5]
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    del tabela[5]
KeyError: 5
>>> print(tabela.pop(2), tabela)
Zvezda {3: 'Rad', 4 : 'Čukarički'}
>>> tabela.pop(5)
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    tabela.pop(5)
KeyError: 5
>>> tabela.pop(5, None)
None
>>> tabela.clear()
>>> tabela
{}
```

Brisanje prdruživanja ključ - vrednost može se realizovati upotrebom operatora `del`, pri čemu interpreter prijavljuje grešku ako navedeni ključ ne postoji. Metoda

`pop()` briše pridruživanje za zadati ključ. Tom prilikom ona vraća vrednost koja se izbacuje iz rečnika. Opet, ako navedeni ključ ne postoji, prijavljuje se greška. Kada se u metodi navede opcioni parametar, greška se ne prijavljuje pa je ovaj način pogodan za izbacivanje iz rečnika u *opštem* slučaju. Metoda `clear()` briše sve elemente iz rečnika.

Prevođenje uz pomoć rečnika

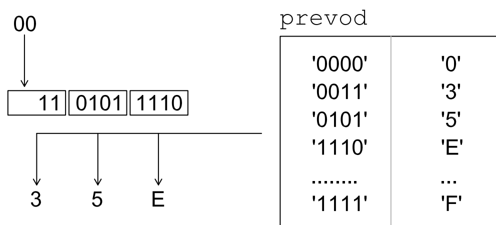
Rečnici nalaze čestu primenu u algoritmima prevođenja kada se vrednost ključa, unutar neke strukture podataka, zamenjuje vrednošću pridruženog objekta. Sledi primer koji prevodi ulaznu tekstualnu sekvencu prema pravilima iz rečnika:

Problem 5.7 — Prevođenje binarnog u heksadekadni zapis. Binarni zapis broja zadat je tekstualnom sekvencom koja sadrži nule i jedinice. Na primer, broj 10111 (23 u dekadnom sistemu), zapisan je kao sekvenca '10111'. Napisati funkciju koja prevodi proizvoljni binarni zapis broja u odgovarajući heksadekadni zapis. Heksadekadni zapis sadrži sledeće cifre: $0,1,2,3,4,5,6,7,8,9,A,B,C,D,E$ i F. Cifra najveće težine, F, u dekadnom sistemu ima vrednost 15. ■

Neka je x dekadni broj čiji je binarni zapis oblika $b_{n-1}...b_1b_0,\ b_i \in \{0,1\}$. Broj x može se zapisati kao:

$$\begin{aligned} x &= b_0 + b_1 2^1 + b_2 2^2 + b_3 2^3 + b_4 2^4 + b_5 2^5 + b_6 2^6 + b_7 2^7 + ... + b_{n-1} 2^{n-1} \\ &= b_0 + b_1 2^1 + b_2 2^2 + b_3 2^3 + 16(b_4 + b_5 2^1 + b_6 2^2 + b_7 2^3) + 16^2(...) + ... \end{aligned} \tag{5.2}$$

Na osnovu (5.2), heksadekadni zapis dobija se *direktno* iz binarnog zapisa tako što se, počevši od b_0, četiri po četiri bita *prevode* u odgovarajuće heksadekadne cifre ($0000 \rightarrow 0, ..., 1111 \rightarrow F$). Prevođenje se može realizovati uz pomoć rečnika koji memoriše preslikavanja iz binarnih kvarteta u heksadekadne cifre. Ako broj bita u binarnom zapisu nije deljiv sa četiri, onda poslednjih nekoliko cifara treba dopuniti nulama sleva do potpunog kvarteta (slika 5.12).



Slika 5.12: Prevođenje uz pomoć rečnika: konvertovanje binarnog u heksadekadni zapis.

Program 5.7 — Prevođenje binarnog u heksadekadni zapis.

```
# prevodjenje bin string -> hex string
prevod = { '0000':'0', '0001':'1', '0010':'2', '0011':'3',
           '0100':'4', '0101':'5', '0110':'6', '0111':'7',
           '1000':'8', '1001':'9', '1010':'A', '1011':'B',
           '1100':'C', '1101':'D', '1110':'E', '1111':'F'  }

def bin2hex(bin_cifre):

    n = len(bin_cifre) # broj cifara binarnog broja
    if bin_cifre.count('0') + bin_cifre.count('1') != n:
        return 'Zapis mora da sadrži samo jedinice i nule!'

    hex_cifre = [] # heksa cifre (od najveće težine nadole)
    ost = n % 4 # koliko bita ostaje po odbijanju punih kvarteta
    if ost != 0:
        hex_cifre.append(prevod[(4-ost)*'0' + bin_cifre[0:ost]])

    for i in range(ost, n, 4):
        hex_cifre.append(prevod[ bin_cifre[i:i+4] ])

    return ''.join(hex_cifre)
```

U programu se prvo definiše preslikavanje iz binarnih kvarteta u heksadekadne cifre (r2-5). Potom se proverava da li ulazna sekvenca `bin_cifre` sadrži samo jedinice i nule (r10). U tu svrhu koristi se tekstualna metoda `count()`. Heksadekadne cifre pamte se u listi `hex_cifre` kao tekstualne sekvence dužine jedan. Ulazni bitovi obrađuju se počevši od prve binarne cifre `bin_cifre[0]`, pri čemu ona predstavlja cifru najveće težine.

Na osnovu ostatka deljenja dužine ulazne sekvence sa četiri, određuje se koliko bita pripada heksadekadnoj cifri najveće težine (r14). Kada je ostatak različit od nule, dodaje se potreban broj nula, pa se potom čita prevod za kvartet bita koji odgovara heksadekadnoj cifri najveće težine (r15, 16). Potom se, za kvartete koji odgovaraju heksadekadnim ciframa, repetitivno određuju prevodi počevši od cifre sledeće najveće težine (r18, 19). Rezultujuća tekstualna sekvenca dobija se uz pomoć metode `join()` koja spaja sve cifre iz `hex_cifre` (r21). Zahvaljujući proveri u r10, za svaku nisku ulaznih simbola obezbeđen je *jednoznačan* prevod. Sledi prikaz rada programa:

```
>>> bin2hex('')
''
>>> bin2hex('10011a1')
'Zapis mora da sadrži samo jedinice i nule!'
>>> bin2hex('10011111')
'9F'
>>> bin2hex('1011110101101')
'17AD'
```

Upotreba rečnika za računanje frekvencija i grupisanje objekata

Pored prevođenja, rečnici se često koriste za *prebrojavanje* pojedinih objekata koji se tada posmatraju kao ključevi, a pridružene vrednosti kao njihove celobrojne frekvencije. Rečnici se koriste i za potrebe *grupisanja* objekata sa istim karakteristikama. Tada se, pod zajedničkom karakteristikom koja je predstavljena ključem, formira i dopunjuje kolekcija koja sadrži srodne objekte. Sledi pogodan primer:

Problem 5.8 — Strelci. Nepoznati broj strelaca više puta gađa u metu. Rezultati pristižu *tokom* gađanja u formi uređenog para oblika $(ime_strelca, broj_kruga)$. Po završetku gađanja treba ispisati sve pogotke za svakog strelca. Prebrojati i ispisati koliko puta je, u toku takmičenja, pogođen svaki krug. ■

Frekvencije pojedinih pogodaka mogu se ažurirati po pristizanju rezultata pomoću rečnika `frekvencije` čiji ključevi odgovaraju brojevima pogođenih krugova. Da bi se grupisali pogoci koji pripadaju istom strelcu, koristiće se rečnik `strelci` čiji ključevi odgovaraju imenima, a vrednosti predstavljaju liste pogodaka koje se, tokom takmičenja, proširuju novopristiglim podacima. Kako rezultati pristižu *tokom* gađanja, obrada podataka može se obaviti unutar beskonačne petlje u kojoj se, u svakoj iteraciji, unose ime igrača i njegov tekući pogodak.

Program 5.8 — Strelci.

```
1  # Strelci
2  strelci, frekvencije = {}, {}
3  while True:
4      # učitavanje
5      ime = input('ime ')
6      if ime == '': # ili if not ime: prazna sekv. je logičko False
7          break  # prekid unosa (prazno ime)
8      broj_kruga = input('broj kruga ')
9      # ažuriranje rečnika za grupisanje i frekvencije
10     if ime in strelci:
```

```
11            strelci[ime].append(broj_kruga)
12        else:
13            strelci[ime] = [broj_kruga]
14        frekvencije[broj_kruga] = frekvencije.get(broj_kruga, 0) + 1
15
16    # ispis rečnika
17    for krug in frekvencije:
18        print('krug', krug, 'pogođen', frekvencije[krug], 'puta')
19    for strelac, pogoci in strelci.items():
20        print('strelac', strelac, ' : ', pogoci)
```

Potrebni rečnici formiraju se u r2. Obrada podataka realizovana je unutar beskonačne petlje `while` (r3-14). Ako se za ime strelca unese prazna sekvenca, unos se prekida (r6-7) i prelazi se na ispis traženih informacija (r17-20). Kada se strelac sa datim imenom pronađe u rečniku, treba pristupiti njegovoj listi pogodaka i proširiti je brojem poslednjeg pogođenog kruga (r11). Testiranje prisustva ključa obavlja se uz pomoć operatora `in`, posle koga se navodi ime rečnika (r10). Postoji i alternativni način koji glasi: `if ime in strelci.keys()`. Metod `keys()` vraća *skup* ključeva iz rečnika. Ako dotični strelac nije prisutan u rečniku, ubacuje se *novo* pridruživanje koje imenu strelca pridružuje novu listu. Ona inicijalno skladišti informaciju o prvom pogotku (r13).

Rečnik `frekvencije` ažurira se uz pomoć metode `get()` čiji argumenti su broj pogođenog kruga i opcioni parametar. Vrednost parametra vraća se kao rezultat metode kada ključ *ne postoji* u rečniku pa je, u kontekstu frekvencija, postavljen na nula (r14). Neiskusan programer mogao bi se zavesti načinom na koji je ažururan rečnik `frekvencije` (r14). U želji da sličan postupak primeni i na rečnik `strelci`, mogao bi *pogrešno* staviti: `strelci[ime] = strelci.get(ime, []).append(broj_kruga)`. Kako metoda `append()` umesto ažurirane liste vraća `None`, pridruživanje za navedeni ključ bilo bi potpuno pogrešno!

Rečnici se, po završetku postupka, ispisuju na dva načina: prvi podrazumeva uvođenje petlje po svim ključevima (r17) preko kojih se pristupa odgovarajućim vrednostima (r18). U drugom, u petlji se pristupa svim torkama oblika *(ključ, vrednost)* kojima su definisana pridruživanja iz rečnika. Sve ovakve torke mogu se dobiti putem metode `items()` (r19). Uočiti da su komponente tekuće torke imenovane kao `strelac` i `pogoci`. Ista petlja mogla je biti realizovana i sa `for m in strelci.items()`, s tim da bi `m[0]` označavalo ime, a `m[1]` listu pogodaka za tekućeg strelca. Sledi prikaz rada programa:

```
ime Pera
```

```
broj kruga 5
ime Joca
broj kruga 4
ime Pera
broj kruga 5
ime Joca
broj kruga 5
ime Mika
broj kruga 3
ime
krug 4 pogođen 1 puta
krug 3 pogođen 1 puta
krug 5 pogođen 3 puta
strelac Joca  :  ['4', '5']
strelac Mika  :  ['3']
strelac Pera  :  ['5', '5']
```

Operatori raspakivanja. Funkcije sa proizvoljnim brojem parametara

Na kraju izlaganja u glavi 5.1.5 bilo je reči o tome da se elementi torke (liste) mogu imenovati pomoću naredbe višestruke dodele, ali se mora paziti da broj promenljivih sa leve strane jednakosti bude *jednak* broju elemenata torke (liste) sa desne strane. Operator *raspakivanja* `*` može se koristiti kada sekvenca sa desne strane, koja se raspakuje, ima više elemenata nego što ima promenljivih sa leve strane:

```
>>> lista = [1, 2, 3, 4, 5]
>>> prvi, *nebitno, četvrti, peti = lista
>>> prvi, četvrti, peti
(1, 4, 5)
>>> nebitno
[2, 3]
>>> prvi, drugi, *nebitno = lista
>>> prvi, drugi
(1, 2)
>>> nebitno
[3, 4, 5]
>>> del lista[-1] # briše poslednja
>>> del lista[-1] # tri elementa
>>> del lista[-1] # u listi
>>> prvi, drugi, *nebitno = lista
>>> prvi, drugi, *nebitno
(1, 2)
>>> nebitno
[]
```

Zvezdica ispred imena promenljive nalaže da se svi elementi, koji "pretiču" sa desne

strane, upakuju u listu čije ime ide posle zvezdice (`nebitno`). Ako takvih elemenata nema (posle brisanja tri poslednja elementa), onda je lista prazna. Operator `*` može se koristiti i da se elementi iz dve ili više sekvenci raspakuju u jedinstvenu sekvencu:

```
>>> # raspakivanje sekvenci sa *
>>> brojevi, tekst, imena = (3, 5, 7), "ABC", ["Ana", "Mila"]
>>> zajedno = [*brojevi, *tekst, *imena, "!"]
>>> zajedno
[3, 5, 7, 'A', 'B', 'C', 'Ana', 'Mila', '!']
>>> # raspakivanje rečnika sa **
>>> balkan = {"Srbija":"Beograd", "Grčka":"Atina"}
>>> evropa = {"Nemačka":"Berlin", **balkan}
>>> evropa
{'Nemačka': 'Berlin', 'Srbija': 'Beograd', 'Grčka': 'Atina'}
```

Primer pokazuje da se, pored sekvenci, mogu raspakivati i rečnici – za tu svrhu koristi se operator `**` koji se navodi ispred imena rečnika. Raspakivanje predstavlja još jedan primer za koncept preopterećenja operatora kada se njegov efekat određuje iz konteksta: za numeričke tipove `*` označava množenje, a `**` stepenovanje.

Operatori raspakivanja najčešće se upotrebljavaju prilikom definisanja funkcije sa *proizvoljnim brojem* parametara. Neka je potrebno sastaviti funkciju koja sabira n brojeva. To se može učiniti na uobičajeni način tako što se brojevi sakupe u listu, pa se onda ona prosledi kao argument funkcije. Međutim, zahvaljujući operatoru raspakivanja sekvence, funkcija se može napisati i tako da, prilikom poziva, primi proizvoljan broj brojeva odvojenih zapetom:

```
1  # klasičan pristup
2  def suma_1(brojevi):
3      s = 0
4      for b in brojevi:
5          s += b
6  
7      return s
8  
9  print('1+2+3+4=', suma_1([1, 2, 3, 4]))
10 
11 # pristup sa pakovanjem u listu
12 def suma_2(*brojevi):
13     s = 0
14     for b in brojevi:
15         s += b
```

```
16
17      return s
18
19  print('1+2+3+4=', suma_2(1, 2, 3, 4))
20  print('10+20+30=', suma_2(10, 20, 30))
```

```
1+2+3+4= 10
1+2+3+4= 10
10+20+30= 60
```

Uočiti da su realizacije obe funkcije istovetne (r3-7 i r13-17). Jedina razlika je u načinu definisanja ulaznih parametara i u samom pozivu (r9, r19-20). Ako se u potpisu funkcije koristi operator `**`, onda ona može da primi i proizvoljan broj imenovanih parametara:

```
1   def izraz(*brojevi, **operacije):
2       e = operacije['eksp']
3       vrednost = brojevi[0]**e
4
5       for b in brojevi[1:]:
6           if operacije['glavna'] == '+':
7               vrednost += b**e
8           elif operacije['glavna'] == '*':
9               vrednost *= b**e
10          else:
11              return 'nedefinisano'
12
13      return vrednost
14
15  print('1^2 * 2^2 * 3^2=', izraz(1, 2, 3, glavna='*', eksp=2))
16  print('1^3 + 2^3 * 3^3=', izraz(1, 2, 3, glavna='+', eksp=3))
```

```
1^2 * 2^2 * 3^2= 36
1^3 + 2^3 * 3^3= 36
```

Ulazni parametar `*brojevi` funkcije `izraz()` omogućava navođenje proizvoljnog broja brojeva odvojenih zapetom. Vrednost izraza koju ova funkcija izračunava zavisi od upotrebljene operacije između brojeva, kao i vrednosti eksponenta kojim se svaki broj stepenuje. Operacija i eksponent memorisani su u rečniku `operacije`, pod

ključevima `"glavna"` i `"eksp"`. Kako se u potpisu funkcije koristi operator `**` (r1), to se, prilikom njenog poziva, mogu navesti imenovani argumenti `glavna="*"` i `eksp=2` (r15), odnosno `glavna="+"` i `eksp=3` (r16). Potrebno je samo da imena ulaznih parametara budu *jednaka* vrednostima tekstualnih ključeva koji se koriste unutar funkcije. Osim toga, imenovani parametri, kojima prethodi operator `**`, moraju se navesti *posle* pozicionih.

Prilikom definisanja funkcije, ulazni parametri kojima prethode operatori raspakivanja mogu se koristiti zajedno sa uobičajenim ulaznim parametrima, s tim da se navode *posle* uobičajenih:

```
1  def f(a, b, *c, **d):
2      print(f'pozicioni parametri bez raspakivanja\na={a}, b={b}')
3
4      print('pozicioni parametri sa raspakivanjem')
5      for p in c:
6          print(p)
7
8      print('imenovani parametri sa raspakivanjem')
9      for p in d:
10         print(f'{p}={d[p]}')
11
12 f(1, True, 'a', 'b', 'c', ip1='prvi', ip2=[1, 2])
```

```
pozicioni parametri bez raspakivanja
a=1, b=True
pozicioni parametri sa raspakivanjem
a
b
c
imenovani parametri sa raspakivanjem
ip1=prvi
ip2=[1, 2]
```

5.3 Kolekcije i kombinatorne strukture

Prilikom rešavanja problema metodom iterativne pretrage u prostoru rešenja, često je potrebno generisati sve moguće *kombinacije* sastavnih delova potencijalnog rešenja – bira se k od n objekata iz zadate kolekcije (videti problem 3.8). U slučaju da se potencijalna rešenja predstavljaju kao sekvence unapred poznatih objekata, potrebno je generisati sve moguće rasporede u sekvenci – *permutacije*, ili izabrati određeni broj

objekata za formiranje sekvence – *varijacije*. Za potrebe generisanja kombinacija, permutacija i varijacija iz zadate kolekcije koriste se funkcije iz modula `itertools`.

5.3.1 Kombinacije

Data je kolekcija objekata od n elemenata. Pretpostaviti da se iz kolekcije bira k elemenata *bez vraćanja*, te da redosled unutar jednog izbora *nije bitan*. Svaki od $\binom{n}{k} = \dfrac{n!}{k!(n-k)!}$ mogućih izbora predstavlja jednu *kombinaciju bez ponavljanja* od k elemenata (kaže se još i k-te klase). Na primer, posmatra se kutija u kojoj se nalaze tri kuglice: crvena (c), plava (p) i zelena (z). Dalje, neka se dve kuglice biraju bez vraćanja po načinjenom izboru. Sve moguće kombinacije bez ponavljanja druge klase su: (c, p), (c, z) i (p, z) (uočiti da (c, z) i (z, c) predstavljaju istu kombinaciju).

Ako se objekti biraju *sa vraćanjem*, onda se izbori nazivaju *kombinacijama sa ponavljanjem*. Na primeru kutije sa kuglicama, dvočlani izbori sa ponavljanjem su: (c, c), (c, p),(c, z), (p, p), (p, z) i (z, z). Broj kombinacija sa ponavljanjem k-te klase, u nekom skupu od n elemenata, jednak je $\binom{n+k-1}{k}$. U narednom problemu izložen je *postupak* za generisanje kombinacija bez ponavljanja:

Problem 5.9 — Kombinacije bez ponavljanja. Data je lista od n elemenata i prirodni broj $1 \leq k \leq n$. Generisati sve kombinacije bez ponavljanja k-te klase. Svaku kombinaciju smestiti u torku od k elemenata. ■

Proizvoljna kombinacija od k različitih elemenata može se dobiti tako što se generiše k različitih brojeva u rasponu od 0 do $n-1$: ovi brojevi predstavljaju *indekse* kojima se *odabiraju* elementi iz liste. Na primer, neka treba generisati sve kombinacije treće klase ($k = 3$) iz liste od četiri elementa ($n = 4$). Ovo se može učiniti tako što se generišu sledeće sekvence vrednosti za indekse i, j i m: (0, 1, 2), (0, 1, 3), (0, 2, 3) i (1, 2, 3). U *svakom* trenutku treba da važi $i < j < m$, pri čemu i "ide" od 0 do 1, j od $i+1$ do 2, a m od $j+1$ do 3. Slično važi i za listu proizvoljne dužine n, pri čemu je prva kombinacija određena sa (0, 1, 2), a poslednja sa $(n-3, n-2, n-1)$: i "ide" od 0 do $n-3$, j od $i+1$ do $n-2$, a m od $j+1$ do $n-1$. Algoritam za kombinacije bez ponavljanja *treće* klase realizuje se preko *tri* ugnježđene petlje `for`:

```
1  # kombinacije bez ponavljanja treće klase
2  def kombinacije_3k(lista):
3
4      kombinacije = [] # sve kombinacije treće klase
5      n = len(lista)
6      if n < 3:  # ne postoji, vrati praznu listu
7          return kombinacije
```

```
    for i in range(n-3 + 1): # ide do n-3
        for j in range(i+1, n-2 + 1): # ide do n-2
            for m in range(j+1, n-1 + 1): # ide do n-1
                kombinacije.append((lista[i], lista[j],
                                    lista[m]))
    return kombinacije
# test
boje = ['plava', 'žuta', 'crvena', 'zelena']
tri_boje = kombinacije_3k(boje);
for b in tri_boje:
    print(b)
```

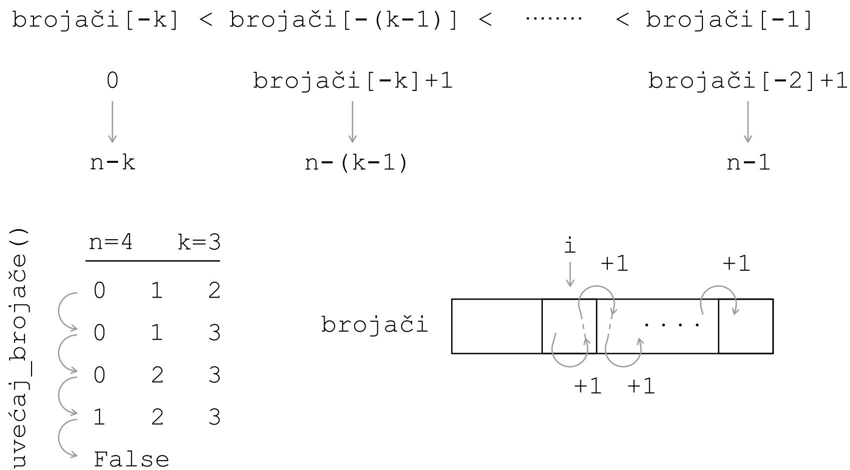
```
('plava', 'žuta', 'crvena')
('plava', 'žuta', 'zelena')
('plava', 'crvena', 'zelena')
('žuta', 'crvena', 'zelena')
```

Po istom principu mogu se generisati i kombinacije bez ponavljanja proizvoljne klase $k \leq n$: umesto tri, potrebno je k ugnježđenih petlji `for`. Međutim, kako je k *nepoznato* u trenutku pisanja programa, ovaj pristup pada u vodu. Ipak, k ugnježđenih petlji, čiji se brojači menjaju po opisanoj šemi, mogu se realizovati putem *liste* od k elemenata. Lista je na početku inicijalizovana sa `[0, 1,..., k-1]`, a na kraju ima vrednosti `[n-k, n-(k-1),..., n-1]` (slika 5.13, gore).

U svakoj iteraciji, vrednosti brojača iz liste `brojači` mogu poslužiti za formiranje jedne kombinacije sastavljene od elemenata iz osnovne liste. Sledeća sekvenca vrednosti za brojačke promenljive može se dobiti pozivom funkcije `uvećaj_brojače()` koja uvećava brojače na potreban način. Na slici 5.13 (dole levo) dat je prikaz rada funkcije za već razmotren primer $n = 4$, $k = 3$. Brojači se uvećavaju s desna na levo – najbrže se uvećava unutrašnja, a najsporije krajnje spoljna petlja. Ako se, idući s desna na levo, ni jedan brojač *ne može* uvećati, onda se prošlo kroz sve kombinacije – funkcija vraća `False`. Pretpostaviti da se brojač na i-toj poziciji *može* uvećati za jedan jer nije dostigao maksimalnu vrednost (5.13, dole desno). Po njegovom uvećavanju, *svi* brojači udesno ažuriraju se tako da se zadrži uređenost njihovih vrednosti, od najmanje do najveće – prvi do i-tog postavlja se na vrednost i-tog uvećanu za jedan, pa tako redom:

Program 5.9 — Kombinacije bez ponavljanja.

```
# kombinacije bez ponavljanja k-te klase
def kombinacije(lista, k):
```



Slika 5.13: Generisanje kombinacija bez ponavljanja: elementi u listi `brojači` referencirani su u odnosu na kraj liste! Prvi element sleva (indeks $-k$) predstavlja brojač krajnje spoljašnje petlje, a poslednji element (indeks -1) brojač unutrašnje petlje koji se najbrže menja. Ovo je učinjeno zbog pogodnije veze između vrednosti indeksa brojača i njegove gornje granice. Na primer, brojač sa indeksom $-(k-1)$ za poslednju vrednost uzima $n-(k-1)$.

```
3
4       def formiraj(brojači):
5           '''formira jednu kombinaciju pomoću vrednosti brojača'''
6           return tuple([lista[b] for b in brojači])
7
8       def uvećaj_brojače(brojači, n):
9           '''uvećava brojače da pripremi sledeću kombinaciju'''
10          for i in range(1, len(brojači) + 1):
11              if brojači[-i] < n - i:
12                  brojači[-i] += 1
13                  for k in range(i-1, 0, -1):
14                      brojači[-k] = brojači[-k-1] + 1
15                  return True
16          return False
17
18      kombinacije, n = [], len(lista)
19      if n < k or k < 1:
20          return kombinacije
21
22      brojači = [i for i in range(k)]
23      kombinacije.append(formiraj(brojači))
24      while uvećaj_brojače(brojači, n):
```

```
25            kombinacije.append(formiraj(brojači))
26
27        return kombinacije
28    # test
29    dve_boje = kombinacije(['plava', 'žuta', 'crvena', 'zelena'], 2)
30    for b in dve_boje:
31        print(b)
```

Parametri funkcije `kombinacije()` su ulazna lista i željena klasa. Ona sadrži dve pomoćne funkcije, `formiraj()` i `uvećaj_brojače()`. Prva formira torku sa objektima iz tekuće kombinacije pomoću notacije za formiranje liste po unapred zadatom kriterijumu (r6). Primetiti da se u r6 koristi promenljiva `lista` koja se, kao lokalna promenljiva (ulazni parametar) funkcije `kombinacije()`, vidi i unutar svih njenih unutrašnjih funkcija (o hijerarhijskoj organizaciji imenskih prostora bilo je reči u glavi 4.1.4). Tekuća kombinacija formira se na osnovu vrednosti brojačkih promenljivih. Funkcija `uvećaj_brojače()` (r8-16) direktno realizuje algoritam sa slike 5.13. Brojači se uvećavaju putem sporednog efekta tako što se prosleđena lista `brojači` menja u unutrašnjosti funkcije. Kao što je već poznato, ovo je moguće jer su liste objekti promenljivih tipova.

Lista `kombinacije`, iz glavne funkcije (r18), memoriše generisane kombinacije. Pošto se brojači, koji simuliraju `k` ugnježđenih petlji, inicijalizuju na početne vrednosti (r22), u listu se dodaje početna kombinacija (r23). Potom se u petlji `while`, sve dok se brojači mogu ažurirati, generišu i dodaju nove kombinacije (r24-25). Program za testiranje realizovan je u (r29-31), a njegov rad prikazan je ispod:

```
('plava', 'žuta')
('plava', 'crvena')
('plava', 'zelena')
('žuta', 'crvena')
('žuta', 'zelena')
('crvena', 'zelena')
```

Modul `itertools`

Kako bi se olakšalo generisanje različitih kombinatorinih struktura sačinjenih od objekata iz različitih kolekcija, standardnoj biblioteci u Pajtonu pridodat je i modul `itertools`. Sledi ilustracija procesa generisanja kombinacija sa i bez ponavljanja pomoću funkcija iz ovog modula:

```
>>> import itertools as it
>>> # kombinacije bez ponavljanja 2. klase
```

```
>>> for kbp in it.combinations(['a', 'b', 'c'], 2):
        print(kbp)


('a', 'b')
('a', 'c')
('b', 'c')
>>> # kombinacije sa ponavljanjem 2. klase
>>> for ksp in it.combinations_with_replacement(['a', 'b', 'c'], 2):
        print(ksp)


('a', 'a')
('a', 'b')
('a', 'c')
('b', 'b')
('b', 'c')
('c', 'c')
>>> list(it.combinations(['a','b','c'], 2))
[('a', 'b'), ('a', 'c'), ('b', 'c')]
>>> tuple(it.combinations({'a','b','c'}, 2))
(('c', 'b'), ('c', 'a'), ('b', 'a'))
```

Po uvođenju modula naredbom `import`, na raspolaganju su funkcije preko kojih se mogu generisati kombinacije bez ponavljanja (`combinations()`) i sa ponavljanjem (`combinations_with_replacement()`). Ove funkcije vraćaju *iterabilne objekte* koji, na *eksplicitan zahtev*, omogućavaju prolazak kroz sekvencu element po element. Sledeći objekat iz sekvence generiše se u *trenutku zahteva*, na osnovu *prethodnog stanja* iterabilnog objekta i *unapred poznatog* pravila. Na ovaj način postiže se *memorijska ušteda* jer se objekti iz sekvence *ne moraju* držati u memoriji, već *samo* tekuće stanje iterabilnog objekta na osnovu koga se generiše sledeći element! Iterabilni objekti se najčešće koriste u okviru petlje `for`. Iterabilni objekti za kombinacije bez i sa ponavljanjem proizvode rezultate u obliku torki.

Sledstveno primeru, ako se iterabilni objekat prosledi konstruktoru neke od kolekcija, sve kombinacije generišu se *odjednom* i smeštaju u dotičnu kolekciju. Ako se kao prvi argument pomenutih funkcija, umesto liste (ili neke druge sekvence), navede *skup*, suštinski se dobija *isti* rezultat jer redosled unutar svake kombinacije, po definiciji, nije bitan! Ipak, zbog upotrebe neuređene kolekcije tipa skupa (poslednji poziv iz primera), kombinacije u torkama imaju *drugačiji* raspored. Ako je ulazna kolekcija *uređena*, onda su kombinacije uređene po *leksikografskom poretku* (videti problem 5.1). Problem 5.9 može se sada rešiti na daleko jednostavniji način:

```
# boje
from itertools import combinations
boje = ['plava', 'žuta', 'crvena', 'zelena']
dve_boje = combinations(boje, 2);# iterabilni objekat, još uvek
                                 # nema komb. u memoriji!
for b in dve_boje:  # u svakoj iteracji generiše se nova komb.
    print(b)
```

Problem 5.10 — Trouglovi. Učitati listu sa dužinama odgovarajućih duži i realne brojeve x i y. Ispisati sve trouglove definisane datim dužima, a čije se površine nalaze u intervalu $[x, y]$. Pored trouglova ispisati i njihove površine. ■

Problem se rešava primenom iterativne pretrage u prostoru rešenja: svaki potencijalni trougao predstavlja se kao jedna kombinacija bez ponavljanja *treće* klase, sastavljena od dužina iz ulazne liste. Međutim, da bi se od stranica dužina a, b i c mogao oformiti trougao, dovoljno je da važe tri *nejednakosti trougla*: $a+b>c$, $a+c>b$ i $b+c>a$. Svaka kombinacija koja ispunjava traženi uslov, pri čemu je površina trougla iz zadatog intervala, predstavlja validno rešenje. Površina trougla, zadatog dužinama stranica, može se izračnati pomoću Heronovog obrasca:

$$P_{abc} = \sqrt{s(s-a)(s-b)(s-c)}, \quad s = \frac{a+b+c}{2}$$

Program 5.10 — Trouglovi.

```
from itertools import combinations

def površina(trougao):
    a, b, c = trougao
    s = (a+b+c) / 2
    return (s * (s-a)*(s-b)*(s-c))**.5

def je_trougao(trougao):
    a, b, c = trougao
    return (a+b > c) and (a+c > b) and (b+c > a)

def trouglovi(stranice, x, y):
    trouglovi = []
    for t in combinations(stranice, 3):
        if je_trougao(t):
            p = površina(t)
```

```
17            if p > x and p < y:
18                trouglovi.append((t, p))
19    return trouglovi
20
21 # test
22 dužine = input('dužine razdvojene razmakom: ')
23 dužine = [float(d) for d in dužine.split()]
24 x = float(input('donja granica: '))
25 y = float(input('gornja granica: '))
26 for t in trouglovi(dužine, x, y):
27     print(t)
```

Trouglovi iz rešenja reprezentovani su *tročlanim torkama* koje sadrže dužine stranica. Funkcija `površina()` računa površinu trougla prema Heronovom obrascu (r3-6). Uočiti kako se elementi torke `trougao` imenuju sa `a`, `b` i `c` (r4). Ako stranice potencijalnog trougla zadovoljavaju potrebne nejednakosti, onda funkcija `je_trougao()` vraća `True`. Primetiti da su potrebna testiranja realizovana putem logičkog izraza bez upotrebe naredbe `if` (r10). Glavni posao obavlja se unutar funkcije `trouglovi()`. Prilikom inicijalizacije petlje `for`, formira se iterabilni objekat pomoću koga će se generisati sve moguće tročlane kombinacije (r14). U svakoj iteraciji generiše se po jedna kombinacija dužina – torka `t`. Potom se proverava da li `t` predstavlja regularni trougao (r15), te da li mu je površina u traženom intervalu (r16-18). Rešenja se ubacuju kao dvočlane torke u listu `trouglovi` (r18) – prvi element torke predstavlja torku sa dužinama stranica, a drugi vrednost površine.

Program za testiranje podrazumeva učitavanje dužina u obliku tekstualne sekvence. Sekvenca sadrži realne vrednosti razdvojene blanko znakom. Potom se, uz upotrebu tekstualne metode `split()`, ulazna sekvenca cepa na deliće i prevodi u realne dužine (r23). Program za testiranje daje sledeći izlaz:

```
dužine razdvojene razmakom: 1 2 3 4 5 6 7 8 9 10
donja granica: 20
gornja granica: 21
((6.0, 7.0, 8.0), 20.33316256758894)
((6.0, 7.0, 9.0), 20.97617696340303)
((6.0, 7.0, 10.0), 20.662465970933866)
```

5.3.2 Permutacije i varijacije

Pored kombinacija (sa i bez ponavljanja), *permutacije* i *varijacije* predstavljaju bitne kombinatorne strukture koje se često koriste u praktičnim problemima. Neka je data *uređena* kolekcija od n elemenata. Permutacije se odnose na sva moguća *preuređenja*

celokupne kolekcije – ima ih ukupno $n!$. Na primer, sve permutacije liste `[1,2,3]`, zapisane u obliku torki, glase: `(1,2,3)`,`(1,3,2)`,`(2,1,3)`,`(2,3,1)`,`(3,1,2)` i $(3,2,1)$ – ima ih ukupno $3! = 6$. Ako se iz neke kolekcije od n elemenata bira k objekata koji se raspoređuju u sekvencu (redosled bitan), onda se radi o *varijacijama*. Varijacije mogu biti *bez ponavljanja* (element se po izboru *ne vraća* u kolekciju) i *sa ponavljanjem* (element se po izboru *vraća* u kolekciju). Broj varijacija bez ponavljanja k-te klase iznosi $n(n-1)...(n-k+1)$, a varijacija sa ponavljanjem iste klase n^k.

Permutacije i varijacije bez ponavljanja mogu se generisati pomoću funkcije `permutations()` iz modula `itertools`. Ova funkcija vraća potrebne iterabilne objekte:

```
>>> import itertools as it
>>> # permutacije
>>> for p in it.permutations(['a','b','c']):
        print (p)


('a', 'b', 'c')
('a', 'c', 'b')
('b', 'a', 'c')
('b', 'c', 'a')
('c', 'a', 'b')
('c', 'b', 'a')
>>> # varijacije bez ponavljanja 2. klase
>>> for vbp in it.permutations(['a','b','c'], 2):
        print (vbp)


('a', 'b')
('a', 'c')
('b', 'a')
('b', 'c')
('c', 'a')
('c', 'b')
```

Pre nego što se izloži postupak za generisanje varijacija sa ponavljanjem, navodi se pojam *Dekartovog proizvoda* dva skupa, $A \times B = \{(a,b) \mid a \in A, b \in B\}$, koji predstavlja sve uređene parove kod kojih je prvi element para iz prvog, a drugi element para iz drugog skupa. Funkcija `product()` vraća iterabilni objekat pomoću koga se mogu dobiti elementi iz Dekartovog proizvoda. Dekartov proizvod može se *višestruko* primeniti na skupu koji se "množi" samim sobom ($A \times A \times ... = A^k$) – ovaj proizvod predstavlja varijacije sa ponavljanjem k-te klase:

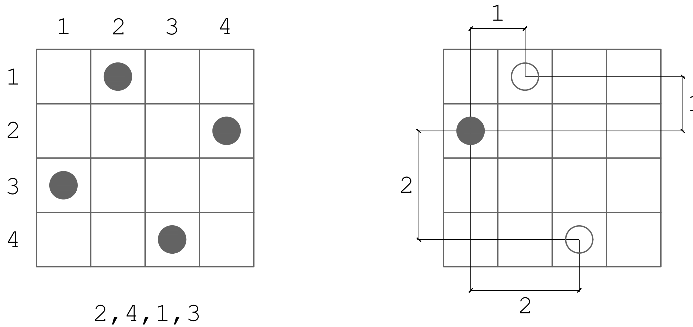
```
>>> # Dekartov proizvod dva skupa
>>> for dp in it.product(['a','b','c'], [1,2]):
        print (dp)


('a', 1)
('a', 2)
('b', 1)
('b', 2)
('c', 1)
('c', 2)
>>> # varijacije sa ponavljanjem 2. klase
>>> for vsp in it.product(['a','b','c'], repeat=2):
        print (vsp)


('a', 'a')
('a', 'b')
('a', 'c')
('b', 'a')
('b', 'b')
('b', 'c')
('c', 'a')
('c', 'b')
('c', 'c')
```

Iz primera se uočava korišćenje opcionog parametra `repeat` kojim se definiše klasa varijacija. Varijacije sa ponavljanjem mogu poslužiti da se *simulira* k ugnježđenih petlji `for`, čiji indeksi uzimaju vrednosti iz *istog* skupa od n elemenata.

Problem 5.11 — Osam dama. Na šahovsku tablu treba rasporediti osam dama tako da se ni jedan par međusobno ne napada. Na primer, ako se dama postavi na polje 'D4', onda se preostale dame ne mogu smestiti ni u jedno polje iste vrste ('4') ili kolone ('D'). Preostale dame ne mogu se smestiti ni na dijagonale koje prolaze kroz 'D4'. Prikazati na pogodan način sve rasporede u kojima dame nisu u međusobnom konfliktu. ■

Problem se može rešiti u opštem slučaju za tablu dimenzija $n \times n$. Posmatra se slika 5.14 (levo) koja ilustruje moguće rešenje za $n = 4$. Prilikom rešavanja problema metodom iterativne pretrage u prostoru rešenja (glava 3.2.3), napomenuta je važnost formalnog opisa potencijalnog rešenja. Sledstveno tome, treba pronaći pogodan način kojim se *kodira* pozicija dama na tabli (reprezentacija potencijalnog rešenja). Kako se u svakoj vrsti i u svakoj koloni *mora* naći *tačno jedna* dama, rešenje se može kodirati brojnom sekvencom dužine n, gde element na i-toj poziciji ima vrednost *broja kolone* u kojoj se, u vrsti i, nalazi postavljena dama. U primeru sa slike 5.14 (levo), rešenje se kodira kao torka `(2,4,1,3)` – u prvoj vrsti dama je u koloni 2, u drugoj u koloni 4, u

trećoj u koloni 1 i u četvrtoj u koloni 3. Očigledno, svaka postavka koja podrazumeva jednu damu po vrsti, odnosno koloni, može se kodirati jednom *permutacijom* sekvence $1, 2, 3, \ldots, n$.



Slika 5.14: Problem četiri dame: moguće rešenje je kodirano permutacijom (2,4,1,3) (levo); uslov za konflikt po dijagonalama – jednako rastojanje između vrsta i kolona (desno).

Posle usvajanja pogodne reprezentacije, problem se može rešiti generisanjem svih permutacija brojeva od 1 do n, pri čemu, za svaku kodiranu poziciju, treba proveriti da se dame međusobno ne napadaju *po dijagonalama*. Ideja o postupku provere dobija se sa slike 5.14 (desno). Uočiti da ako su dve dame na *istoj* dijagonali, onda je *rastojanje* između njihovih vrsta i kolona jednako (jednaka *apsolutna razlika* brojeva njihovih vrsta i kolona). Na osnovu prethodno rečenog, realizovan je sledeći program:

Program 5.11 — Osam dama.

```
1  # n dama
2  from itertools import permutations
3  def dame(n): # tabla nxn
4
5      rešenja = 0
6      for tabla in permutations(range(n)):
7          if not konflikt(tabla):
8              prikaži(tabla)
9              rešenja += 1
10     print(f'za n = {n} ima {rešenja} rešenja')
11
12 def konflikt(tabla): # da li su dame u konfliktu
13
14     n = len(tabla)
15     for i in range(n-1):
16         for j in range(i+1, n):
```

```
17              if abs(tabla[i] - tabla[j]) == abs(i-j):
18                  return True # konflikt!
19      return False # nema konflikta
20
21  def prikaži(tabla): # prikaz table, prazno . , dama D
22
23      n = len(tabla)
24      for kolona in tabla:
25          # ispis reda table
26          print(kolona * '.' + 'd' + (n-kolona-1) * '.')
27      print('') # novi red posle table
28
29  dame(8) # test za standardnu tablu 8x8
```

Pronalaženje i ispisivanje svih mogućih rasporeda, za zadatu dimenziju table, obavlja se u funkciji `dame()` (r3-10). Promenljiva `rešenja` ukazuje na *ukupan broj* rasporeda u kojima n dama mirno koegzistira na tabli. Ovaj broj ispisuje se na kraju postupka (r10). Sve permutacije generišu se uz pomoć iterabilnog objekta iz r6. Uočiti da brojevi u permutaciji idu od nula do $n-1$. U svakoj iteraciji petlje `for` razmatra se po jedan raspored kodiran torkom `tabla` (r7). U tu svrhu koristi se funkcija `konflikt()` koja vraća `True` ako na tabli postoji bar jedan par dama na istoj dijagonali. Kada nema konflikta, poziva se funkcija `prikaži()` koja na pogodan način ispisuje pronađeno rešenje (r8). Uvećavanje brojača dozvoljenih rasporeda obavlja se posle ispisa (r9).

Funkcija `konflikt()` realizuje se prema zapažanju sa slike 5.14 (desno). Unutar petlji po `i` i `j` se proverava da li dame, koje se nalaze u i-toj i j-toj vrsti, dele istu dijagonalu. To se dešava kada je rastojanje između njihovih vrsta jednako rastojanju između njihovih kolona (r17). Primetiti da su petlje tako postavljene da formiraju sve kombinacije vrsta (parove dama), pri čemu je prvi indeks *uvek* manji od drugog. Na primer, ako su testirane dame u trećoj ($i = 2$) i osmoj vrsti ($j = 7$), onda nema potrebe da se testira za $i = 7$ i $j = 2$.

Prilikom ispisa mogućeg rasporeda u funkcji `ispis()`, formira se matrica $n \times n$ sa praznim poljima (`'.'`) i damama (`'d'`) (r21-27). Na ulazu u funkciju prosleđuje se dozvoljena permutacija (`tabla`). Redovi table se ispisuju u petlji pomoću operatora za multiplikaciju i spajanje tekstualnih sekvenci (r26). Sledi prikaz *poslednja dva* od ukupno 92 rešenja za slučaj standardne šahovske table ($n = 8$).

```
.......d
..d.....
d.......
.....d..
```

```
.d......
....d...
......d.
...d....

.......d
...d....
d.......
..d.....
.....d..
.d......
......d.
....d...

za n = 8 ima 92 rešenja
```

Kad god je to moguće, iteratvnu pretragu putem grube sile treba redukovati na prostor sa što manje slučajeva. U primeru sa damama na tabli dimenzije n, postoji $\binom{n^2}{n}$ različitih rasporeda, ali se, uz pomoć permutacija, prostor sveo na $n!$ rasporeda. Za $n = 8$, umesto 4 426 165 368, testirano je samo 40 320 rasporeda!

ŠTA JE NAUČENO

- Kolekcija je objekat koji, pod jednim imenom, grupiše druge objekte – elemente. Zapravo, umesto samih objekata, ona sadrži objektne reference.
- Kolekcije se, prema načinu organizacije elemenata, mogu podeliti na sekvence, skupove i rečnike.
- Sekvenca je uređeni skup objekata sa tačno definisanim redosledom elemenata. Elementima se pristupa putem indeksa koji označavaju pozicije u sekvenci. Indeksi se kreću od 0 do *n* - 1, gde *n* označava dužinu sekvence.
- Sekvenca omogućava pristup svojim delovima (podsekvencama) i njihovo eventualno izdvajanje, navođenjem indeksa kojima se definiše željeni opseg.
- Tekstualna sekvenca sastoji se od karaktera koji reprezentuju slova i simbole iz prirodnih i formalnih jezika. Karakteri se i sami tretiraju kao tekstualne sekvence dužine jedan, a interno su predstavljeni nizovima bita.
- Pajton koristi standard Unicode koji karakterima pridružuje jedinstvene prirodne brojeve - UCP(Unicode Code Point). Za preslikavanje UCP brojeva u nizove bita koristi se tablica preslikavanja po UTF-8 standardu.

- Tekstualna sekvenca je nepromenljiv objekat – ne može menjati svoju vrednost po kreiranju. Promena postojeće sekvence zahteva kreiranje nove.
- Lista je sekvenca potencijalno različitih tipova objekata. Može se menjati po kreiranju tako što se dodaju novi ili brišu stari elementi.
- Iako može objediniti različite objektne tipove, lista je zamišljena da grupiše istovrsne objekte iz istog logičkog konteksta. Na primer, grupiše se niz merenja za neku fizičku veličinu na osnovu prostorne ili vremenske raspodele.
- Lista može da sadrži unutrašnje liste, što omogućava realizovanje višedimenzionalne matrice.
- Torka je nepromenljiva sekvenca potencijalno različitih tipova objekata. Često se koristi za grupisanje različitih tipova podataka koji se odnose na isti kontekst. Na primer, različiti podaci o nekom studentu mogu se opisati torkom.
- Skup je neuređena kolekcija objekata nepromenljivih tipova gde redosled nije bitan. Za skup se samo može utvrditi da li mu neki objekat pripada ili ne.
- Skup ne dozvoljava dodavanje objekata istih vrednosti. Otuda se često koristi za eliminaciju duplikata u podacima.
- Rečnik je neuređena kolekcija parova oblika ključ - vrednost. Ključevi mogu biti nepromenljivi objektni tipovi, a vrednosti objekti proizvoljnih tipova. U drugim jezicima, poput Jave, za rečnik se koristi termin 'mapa'.
- Rečnik omogućava pristup objektima putem ključa i može se menjati tokom vremena. Umesto pristupa na određenoj poziciji, zahteva se element na koga asocira navedeni ključ. Na primer, traži se glavni grad države koja je reprezentovana tekstualnim ključem `'Srbija'`.
- Rečnici se, pored ostalog, koriste za prevođenje izvorne u ciljnu reprezentaciju podataka, za gupisanje objekata sa istim svojstvima ili za prebrojavanje pojedinih objekata u nekom kontekstu.
- Ako se promenljiva kolekcija, poput liste ili rečnika, pojavi kao ulazni parametar, njen sadržaj može se promeniti unutar funkcije – sporedni efekat.
- Metode su funkcije koje se pokreću nad pojedinačnim objektima iz određenog tipa (klase). Pokretanje se obavlja u notaciji sa tačkom.
- Tokom rada, metoda uzima u obzir vrednost objekta nad kojim je pozvana, kao i vrednosti eventualnih ulaznih parametara. Ako je definisana u okviru promenljivog tipa, može modifikovati objekat nad kojim je pozvana.
- Pri rešavanju problema putem iterativne pretrage prostora rešenja, često su potrebne različite kombinatorne strukture objekata iz neke kolekcije. Za generisanje kombinacija, permutacija i varijacija, koriste se funkcije iz modula `itertools`.

6. Algoritmi zasnovani na slučajnim brojevima

Algoritam, koji prilikom ponovljenog izvršavanja za iste ulaze daje uvek isti izlaz, naziva se *deterministički algoritam*. Svi do sada izloženi algoritmi imali su pomenuto svojstvo. Ipak, potpuna određenost u ponašanju determinističkih algoritama *sprečava* njihovu primenu u mnogim oblastima. Realne sisteme odlikuje priličan stepen *neodređenosti*. Na primer, bolje rangirani teniser *obično* pobeđuje na tri dobijena seta, ali *moguć* je i obrnuti ishod. Deterministički algoritam ne može se upotrebiti za *simuliranje* teniskog turnira jer bi ishodi svih mečeva bili unapred poznati.

Za potrebe određivanja ishoda meča, teniski algoritam treba da poseduje mehanizam koji generiše *slučajne* veličine tako da bolje plasirani igrač *češće* pobeđuje nego što gubi. Neka je, praćenjem ranijih rezultata, utvrđeno da bolje plasirani igrač dobija 70% odigranih mečeva. Algoritam primenjuje sledeću taktiku: iz šešira se, na potpuno slučajan način, izvlači jedan broj od 1 do 10. Ako je odabrani broj veći od 7, što se očekuje u 30% izvlačenja, bolje rangirani igrač gubi. Opisani postupak predstavlja primer *nedeterminističkog algoritma*. Zahvaljujući ugrađenom *izvoru slučajnosti*, pri ponovljenom izvršavanju nedeterminističkog algoritma za isti ulaz obično se dobijaju *različiti* izlazi. Izvor slučajnosti realizuje se uz pomoć *generatora slučajnih brojeva*.

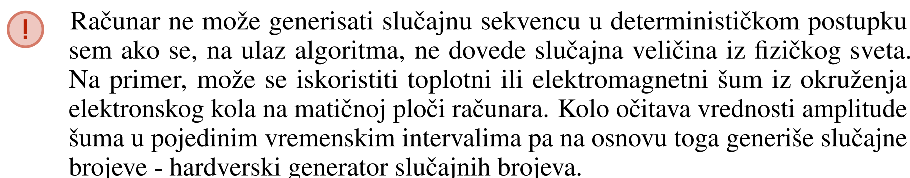
Generatori slučajnih brojeva koriste se i u okviru determinističkog postupka za potrebe generisanja ulaznih podataka. U ovom kontekstu, postupak prevođenja ulaznih u izlazne veličine potpuno je određen, ali se slučajnim ulazom simulira *nepredvidivo* korisničko ponašanje.

6.1 Pseudoslučajni brojevi

Definicija slučajnih brojeva odnosi se na sekvencu brojeva iz nekog skupa S. Sekvenca je slučajna ako su ispunjena sledeća dva uslova:

1. brojevi iz sekvence su *uniformno* raspoređeni u odnosu na S.
2. za svako n važi da, na osnovu prvih $n-1$ članova sekvence, *nije moguće* zaključiti koji broj predstavlja sledeći (n-ti) član.

Uniformna raspoređenost podrazumeva da svi brojevi iz S imaju *jednaku šansu* da budu izabrani. Ovo praktično znači da su, u dovoljno dugačkim sekvencama, frekvencije pojavljivanja svih brojeva iz S jednake ili približno iste.



Računar ne može generisati slučajnu sekvencu u determinističkom postupku sem ako se, na ulaz algoritma, ne dovede slučajna veličina iz fizičkog sveta. Na primer, može se iskoristiti toplotni ili elektromagnetni šum iz okruženja elektronskog kola na matičnoj ploči računara. Kolo očitava vrednosti amplitude šuma u pojedinim vremenskim intervalima pa na osnovu toga generiše slučajne brojeve - hardverski generator slučajnih brojeva.

Hardverski generatori, zbog specifične obrade informacija iz realnih fizičkih izvora, ne mogu da proizvedu dovoljan broj slučajnih brojeva u kratkom vremenskom periodu. U praksi se zato najčešće koriste *pseudoslučajni generatori* čiji brojevi ne ispunjavaju uslov (2) iz definicije o slučajnim sekvencama. Za zadati početni broj, pseudoslučajna sekvenca potpuno je *određena* i *ponavlja* se posle konačno mnogo članova! Početni broj naziva se seme sekvence.[1] Ipak, kako sekvenca ispunjava uslov uniformnosti, te kako je veoma teško uočiti šablon po kome se brojevi ređaju, pseudoslučajni brojevi mogu se primeniti u najvećem broju nedeterminističkih algoritama. Izuzetak su primene koje traže potpunu slučajnost, poput kriptografije ili elektronskih igara na sreću.

funkcija	**opis**
`seed(s)`	inicijalizuje generator semenom s. Ako se s izostavi, za seme se uzima trenutno sistemsko vreme
`random()`	vraća sledeći uniformno raspoređeni realni broj na intervalu $[0,1)$
`uniform(a,b)`	vraća sledeći uniformno raspoređeni realni broj na intervalu $[a,b]$
`gauss(mu, sigma)`	vraća sledeći normalno raspoređeni realni broj iz $N(mu,sigma)$
`randrange(n)`	vraća sledeći uniformno raspoređeni ceo broj na intervalu $[0,n)$
`randint(a,b)`	vraća sledeći uniformno raspoređeni ceo broj na intervalu $[a,b]$

Tabela 6.1: Modul `random`: često korišćene funkcije za generisanje pseudoslučajnih brojeva.

[1] Engl. *Seed*.

Generisanje pseudoslučajnih brojeva u Pajtonu realizuje se uz pomoć funkcija iz modula `random` (tabela 6.1). Sledi primer korišćenja funkcija iz tabele 6.1:

```
>>> import random as r
>>> [r.random() for i in range(3)]
[0.033646681621365304, 0.2320163346900348, 0.8893950894419019]
>>> r.seed(123)
>>> [r.random() for i in range(3)]
[0.052363598850944326, 0.08718667752263232, 0.4072417636703983]
>>> r.seed(123)
>>> [r.random() for i in range(3)]
[0.052363598850944326, 0.08718667752263232, 0.4072417636703983]
>>> [r.uniform(5, 9) for i in range(3)]
[5.15261466440929, 7.144808160135708, 6.328790794038719]
>>> [r.gauss(0, 1) for i in range(3)]
[0.3529214268998316, -0.47259875675165125, -0.4695527782004237]
>>> [r.randrange(10) for i in range(20)]
[0, 6, 8, 6, 8, 9, 7, 8, 6, 5, 8, 0, 2, 1, 7, 4, 2, 5, 5, 8]
>>> [r.randint(2,4) for i in range(20)]
[2, 2, 2, 3, 3, 4, 3, 3, 3, 2, 3, 3, 4, 3, 2, 4, 4, 4, 4, 3]
```

U gornjem primeru korišćena je notacija za formiranje liste po unapred poznatom pravilu (glava 5.1.3). Uočiti da se sekvenca ponavlja pošto se generator inicijalizuje istim semenom (`seed(123)`). Bez obzira na uniformnost generatora, članovi poslednje dve celobrojne liste samo su približno jednako zastupljeni, što je posledica malog broja elemenata u sekvenci (20). Uniformnost se ispoljava tek kada ovaj broj poraste.

Problem 6.1 — Pogodi lutajući broj. Računar "zamišlja" broj od 1 do n, a igrač se trudi da ga pogodi. Ako je igrač neuspešan u pogađanju, računar posle svakih $m < n$ pokušaja bira novi broj iz istog intervala. Realizovati igru pogađanja uz ispis broja pokušaja i mogućnost ponovnog igranja. ■

Računar bira *uniformno* raspoređeni slučajni broj iz intervala $[1,n]$ pomoću funkcije `randint()` iz modula `random`. Kada bi zamišljeni broj bio fiksan, onda bi igrač lakše pogađao broj. Igra je realizovana u beskonačnoj petlji koja se prekida samo ako igrač unese n i m takve da *nije* $1 \leq m < n$:

Program 6.1 — Pogodi lutajući broj.

```
1  from random import randint
2
3  def partija(n, m):
4
```

```
    broj_pokusaja, zamisljen = 0, randint(1, n)

    while True:
        pokusaj = int(input('Vaš broj? '))
        broj_pokusaja += 1

        if pokusaj < 1 or pokusaj > n:
            print(f'Broj treba da je između 1 i {n}!')

        elif pokusaj == zamisljen:
            return broj_pokusaja

        elif broj_pokusaja % m == 0:
            zamisljen = randint(1, n)
            print('Menjam broj!')

# glavna petlja igre
while True:
    n = int(input('Nova igra: n=? '))
    m = int(input('Nova igra: m(m<n)? '))

    if 1 <= m < n:
        rezultat = partija(n, m) # nova partija
        print(f'Pogodili ste iz {rezultat} puta!')

    else:
        print('Bilo je zanimljivo igrati sa vama!')
        break
```

Igra se odvija po *partijama* koje su realizovane funkcijom `partija()` (r3-20). Partija je definisana sa dva ulazna parametra, `n` i `m`, čije je značenje objašnjeno u formulaciji. Funkcija vraća ukupan broj pokušaja na koji ukazuje promenljiva `broj_pokusaja`, a zamišljeni broj iz traženog intervala generiše se pozivom funkcije `randint(1, n)` (r5). Pogađanje se odvija u petlji (r7-19) koja se može napustiti samo kada je igrač pogodio zamišljeni broj (r14-15). Petlja se prekida pomoću naredbe `return` pa se time završava i partija. Ako je uneti broj van traženog intervala (r11-12), ispisuje se upozorenje i igrač nastavlja da pogađa. Ako je broj pokušaja deljiv sa `m` (r17), računar zamišlja novi broj i obaveštava igrača (r18-19).

U glavnoj petlji igre (r22-32), učitavaju se parametri partije (r23-24) i poziva funkcija koja je realizuje (r27). Ukoliko su parametri pogrešni (r26), ispisuje se

prigodna poruka (r31) i igra se prekida pomoću naredbe `break` kojom se izlazi iz glavne petlje (r32). Sledi prikaz jedne igračke sesije:

```
Nova igra: n=? 10
Nova igra: m(m<n)? 4
Vaš broj? 1
Vaš broj? 2
Vaš broj? 3
Vaš broj? 8
Pogodili ste iz 4 puta!
Nova igra: n=? 10
Nova igra: m(m<n)? 2
Vaš broj? -5
Broj treba da je između 1 i 10!
Vaš broj? 8
Menjam broj!
Vaš broj? 1
Vaš broj? 3
Menjam broj!
Vaš broj? 7
Vaš broj? 1
Menjam broj!
Vaš broj? 4
Pogodili ste iz 7 puta!
Nova igra: n=? -1
Nova igra: m(m<n)? 2
Bilo je zanimljivo igrati sa vama!
```

6.2 Funkcije za slučajni odabir i mešanje

Često je potrebno da se iz populacije, zadate odgovarajućom kolekcijom, *izabere* slučajni uzorak. U tu svrhu koriste se funkcije `choice()`, `sample()` i `choices()`. Prva funkcija odabira *tačno jedan* slučajni objekat iz ulazne sekvence. Funkcija `sample()` kreira *novu listu* sa zadatim brojem slučajno izabranih objekata iz navedene *sekvence* ili *skupa*. Svaki element ulazne kolekcije može se odabrati *tačno* jednom - odabiranje *bez* vraćanja. Ako je zadati broj veći od veličine kolekcije, funkcija `sample()` prijavljuje grešku. Uzorak se može kreirati i pomoću odabiranja *sa* vraćanjem - funkcija `choices()`. U ovako kreiranom uzorku svaki element može se ponoviti *više* puta. Sledi primer korišćenja ovih funkcija:

```
>>> import random as r
>>> kockica = [1, 2, 3, 4, 5, 6]
>>> r.choice(kockica)   # jedno bacanje
5
```

```
>>> r.sample(kockica, 3)  # slucajni uzorak od 3 razl. elementa
[1, 4, 3]
>>> r.choices(kockica, k=3) # uzorak sa mogućim ponavljanjem
[4, 2, 2]
>>> r.choices(kockica, k=7) # broj el. može biti veći od populacije
[2, 3, 6, 1, 2, 6, 4]
```

Uočiti da se, prilikom poziva funkcije `choices()`, argument za veličinu uzorka mora imenovati sa `k`. Ako je `k` jednako jedan, onda se ova funkcija ponaša kao `choice()`. Modul `random` sadrži i funkciju za *mešanje* ulazne sekvence - `shuffle()`. Mešanje se primenjuje *samo* na ulazne sekvence promenljivog tipa poput liste:

```
>>> import random as r
>>> samoglasnici = ['a', 'e', 'i', 'o', 'u']
>>> r.shuffle(samoglasnici)
>>> samoglasnici
['o', 'e', 'a', 'u', 'i']
```

Problem 6.2 — Generator (jakih) lozinki. Generisati *jaku* lozinku od n karaktera za prijavu na proizvoljan sistem. Lozinka ne sme biti kraća od osam karaktera i mora da sadrži mala i velika slova, kao i cifre. Lozinka ne sme da sadrži iste karaktere. ■

Kako bi generisana lozinka bila univerzalno primenljiva, poželjno je da se njeni karakteri mogu kodirati uz pomoć ASCII standarda koji je podržan od strane svih operativnih sistema. Zbog toga, karakteri se biraju iz skupa slova engleskog alfabeta i arapskih cifara. Navedene grupe karaktera (mala i velika slova, cifre) mogu se smestiti u tri skupa pa se problem svodi na izbor potrebnog broja elemenata iz svakog od njih. Kako lozinka ne sme da sadrži karaktere koji se ponavljaju, iz skupova se može birati bez vraćanja pomoću funkcije `sample()`. U suprotnom, mogla bi se koristiti i funkcija `choices()`.

Ukupan broj izabranih elemenata iz sva tri skupa mora biti jednak n. U postupku konstrukcije lozinke može se primeniti sledeća šema biranja (veličine *bms* i *bvs* odnose se na izabrani broj malih i velikih slova):

- Prvo se, iz skupa sa malim slovima, slučajno bira *bms* elemenata. Broj *bms* treba slučajano izabrati iz intervala $[1, n-2]$, kako bi se bar po jedan element mogao birati i iz preostala dva skupa.
- Potom se, iz skupa sa velikim slovima, slučajno bira *bvs* elemenata. Broj *bvs* treba slučajano izabrati iz intervala $[1, n - bms - 1]$, kako bi se bar jedan element mogao birati iz skupa sa ciframa.
- Na kraju postupka, iz liste sa ciframa bira se $n - bms - bvs$ elemenata.

- Kako bi lozinka po svojoj prirodi bila što manje pravilna, izabrane karaktere treba dodatno permutovati (promešati) pomoću funkcije `shuffle()`.

Sledi realizacija predloženog postupka u Pajtonu:

Program 6.2 — Generator (jakih) lozinki.

```
from random import randint, sample, shuffle

slova = 'abcdefghijklmnopqrstuvwxyz'
mala_slova = set(slova)
velika_slova = set(slova.upper())
cifre = set('0123456789')

def lozinka(n):
    ''' Kreira jaku lozinku (mala i velika slova, cifre)'''
    karakteri = []
    if n < 8:
        return None
    else:
        bms = randint(1, n - 2)
        bvs = randint(1, n - bms - 1)
        bc = n - bms - bvs

        karakteri.extend(sample(mala_slova, bms))
        karakteri.extend(sample(velika_slova, bvs))
        karakteri.extend(sample(cifre, bc))
        shuffle(karakteri)

        return ''.join(karakteri)

# test
n = int(input('Dužina '))
print('Vaša lozinka je', lozinka(n))
```

Skupovi sa malim i velikim slovima engleskog alfabeta, kao i skup sa arapskim ciframa, definisani su u *globalnom* imenskom prostoru programa (r3-6). Budući da imaju globalni karakter, skupovima se može pristupati u funkciji `lozinka()` (r8-23). Obratiti pažnju na upotrebu konstruktora `set()` koji kreira tražene skupove iz tekstualne sekvence - objekti se eksplicitno kreiraju pozivom konstruktora samo onda kada se prave iz drugog tipa. Osim toga, na ovaj način se postiže ušteda pri pisanju koda u

odnosu na slučaj kada se skupovi inicijalizuju sa pojedinačnim članovima. U funkciji se definiše lista `karakteri` koja će tokom postupka akumulirati sastavne karaktere lozinke (r10). U (r14-15) generišu se slučajni brojevi koji definišu veličine uzoraka za tri navedene grupe karaktera. Metodom `extend()`, iz tipa `list`, lista `karakteri` se proširuje dodavanjem svih elemenata slučajno formiranih uzoraka (r18-20). Ako je tražena dužina lozinke manja od dozvoljene, funkcija vraća `None` (r11-12). U suprotnom, karakteri iz liste spajaju se u tekst lozinke putem metode `join()` i vraćaju kao rezultat funkcije u r23. Kako između karaktera nema dodatnih simbola, za spajajuću sekvencu odabran je prazan string. Rad programa ilustrovan je u sledećem prikazu:

```
Dužina 10
Vaša lozinka je faTqS83wpu
```

6.3 Probabilistički algoritmi

Probabilistički algoritmi predstavljaju nedeterminističke postupke za *približno* rešavanje problema pri čemu je greška, koja se čini u najvećem broju praktičnih situacija, *prihvatljiva*. Treba ih razlikovati u odnosu na metodu uzastopne aproksimacije (glava 3.2.1) koja, prilikom sprovođenja različitih izračunavanja, ne koristi slučajne veličine u svom radu. U praksi se upotrebljavaju kada su tačne metode *nepoznate* ili *veoma kompleksne*, kao i u slučaju modelovanja realnih sistema putem *simulacije*. Za razumevanje prirode probabilističkih algoritama, kao i za procenu prihvatljivosti dobijenih rešenja, neophodno je poznavanje *osnovnih pojmova* iz *teorije verovatnoće* i *statistike*. Sledi kratak pregled osnovnih pojmova, neophodan za razumevanje algoritama iz ove glave.

6.3.1 Verovatnoća događaja i slučajne promenljive

U matematici se izvesnost dešavanja nekog događaja A kvantifikuje realnim brojem $P(A)$, $P(A) \in [0,1]$, koji se zove *verovatnoća* događaja A.[2] Verovatnoća događaja može se neformalno definisati kao odnos broja *povoljnih* i broja *svih mogućih* elementarnih ishoda za događaj. Na primer, prilikom bacanja fer kockice[3], verovatnoća dobijanja *parnog* broja iznosi 0.5 jer od šest mogućih ishoda (1, 2, 3, 4, 5, 6), tačno tri su povoljna (2, 4, 6). *Siguran* događaj ima verovatnoću dešavanja jednaku jedinici (slučajno izabrani parni broj deljiv je sa dva), *nemoguć* događaj verovatnoću nula (prilikom bacanja kockice pašće negativan broj), a svi ostali događaji su izvesni u manjoj ili većoj meri. Što je događaj *izvesniji*, to je njegova verovatnoća *bliža* jedinici.

Neka su sa A i B označena dva događaja. Često je potrebno izračunati $P(AB)$, verovatnoću da se oba događaja dese *istovremeno*. Ova verovatnoća jednaka je količniku broja povoljnih ishoda kojima se ostvaruju *zajedno* A i B, i broja svih mogućih ishoda.

[2] Engl. *Probability*.

[3] Kockica ima ravnomerno raspoređenu masu po zapremini – svi brojevi imaju jednaku šansu pri bacanju.

Na primer, neka se baca fer kockica i neka događaj A glasi "pao je broj veći od tri", a B, "broj je paran". Očigledno, povoljni ishodi za A su brojevi 4, 5 i 6, a za B, brojevi 2, 4 i 6. Povoljni ishodi za događaj AB su brojevi 4 i 6 pa je $P(AB) = 2/6 = 1/3$.

Sledeći bitan pojam odnosi se na *nezavisnost* dva događaja. Dva događaja su nezavisna ako dešavanje prvog *ne utiče* na verovatnoću dešavanja drugog. Na primer, pojava petice u prvom bacanju fer kockice ne utiče na verovatnoću da u drugom bacanju ponovo ispadne pet. Uopšte uzevši, može se smatrati da su uzastopna bacanja fer kockice nezavisna jer ne postoji nikakva *uzročno posledična* veza između njih. Matematička formulacija nezavisnosti glasi: A i B su nezavisni ako i samo ako važi $P(AB) = P(A)P(B)$. Događaji A i B iz prethodnog primera nisu nezavisni jer $P(A) = 0.5$, $P(B) = 0.5$ i $P(AB) = 1/3 \neq 0.5 \cdot 0.5$. Za nezavisne događaje A i B važi da je verovatnoća događaja $A+B$, dešava se ili A ili B, jednaka $P(A+B) = P(A) + P(B)$.

Često se, umesto događaja, posmatraju slučajne veličine kojima se mogu opisati ishodi nekog *slučajnog procesa*. Ove veličine uzimaju brojne vrednosti iz nekog skupa S i zovu se *slučajne promenljive*. Ako je skup S *prebrojiv*,[4] onda se radi o *diskretnoj* slučajnoj promenljivoj. Na primer, u eksperimentu u kome se 10 puta baca novčić, promenljiva X, koja se odnosi na ukupan broj pisama, ima diskretnu prirodu. Za razliku od diskretne, *neprekidna* slučajna promenljiva uzima vrednosti iz *neprebrojivog* skupa S (podskup ili skup realnih brojeva). Očigledno, neprekidna slučajna promenljiva ima *beskonačno* mnogo vrednosti. Na primer, promenljiva Y, koja se odnosi na visinu slučajno izabranog studenta, je neprekidna jer može uzeti bilo koju vrednost iz realnog intervala $[0, 250]$ (za očekivati je da će sve visine biti između 0 i 250 cm). Neprekidne slučajne promenljive pogodne su za opisivanje *fizičkih* veličina iz realnog sveta.

Raspodele

Slučajne promenljive opisuju se *raspodelom verovatnoća*. Raspodela diskretne slučajne promenljive X zadaje se tabelom koja, za svaku vrednost $X = x_i$, prikazuje odgovarajuću verovatnoću njenog ostvarivanja $p_i = P(X = x_i)$. Tabela 6.2 pokazuje raspodelu za slučajnu promenljivu X koja se odnosi na "broj dečaka u tročlanoj porodici". Verovatnoće za pojedine vrednosti X izračunate su uz pretpostavku da je verovatnoća rađanja dečaka jednaka 0.5, kao i da pol deteta *ne zavisi* od pola starije braće i sestara.

X	**0**	**1**	**2**	**3**
$P(X)$	1/8	3/8	3/8	1/8

Tabela 6.2: Diskretna raspodela verovatnoća: broj dečaka u tročlanoj porodici. Na primer, ako je $X = 2$, moglo je biti $110, 101$ ili 011 (1 na i-toj poziciji označava da je i-to rođeno dete dečak). Kako su verovatnoće rođenja dečaka i devojčice jednake, te kako su ishodi na svakoj poziciji međusobno nezavisni, važi $P(X = 2) = P(110) + P(101) + P(011) = 3 \cdot \frac{1}{2}\frac{1}{2}\frac{1}{2} = \frac{3}{8}$.

[4] Mogu se navesti svi elementi u S, ali ne nužno u konačnom vremenu (na primer, skup prirodnih brojeva).

Iz tabele 6.2, uočava se važno svojstvo svake diskretne raspodele: suma svih verovatnoća iz raspodele jednaka je jedinici (slučajna promenljiva *mora* uzeti neku od vrednosti iz raspodele). Diskretna raspodela okarakterisana je sa dva bitna parametra – *matematčko očekivanje* $E[X]$ i *varijansa* $V[X]$:

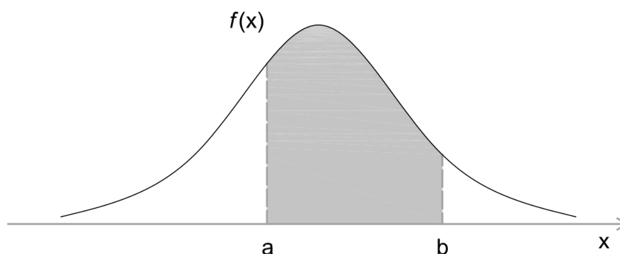
$$E[X] = \mu = \sum x_i p_i \tag{6.1}$$

$$V[X] = \sigma^2 = \sum (x_i - \mu)^2 p_i \tag{6.2}$$

Matematičko očekivanje, za primer sa dečacima, iznosi $\mu = 0 \cdot \frac{1}{8} + 1 \cdot \frac{3}{8} + 2 \cdot \frac{3}{8} + 3 \cdot \frac{1}{8} = 1.5$. Očekivanje predstavlja *srednju* vrednost slučajne promenljive na nivou *svih* mogućih ishoda. Smisao matematičkog očekivanja može se posmatrati i na sledeći način: ako se pretpostavi da je, iz skupa svih tročlanih porodica (*populacija*), na slučajan način odabran *uzorak* od n porodica, onda će *prosečan* broj dečaka u uzorku težiti $\mu = 1.5$ kada $n \to \infty$. Ovo tvrđenje naziva se još i *zakon velikih brojeva*.

Prema (6.2), varijansa za distribuciju iz tabele 6.2 iznosi $\sigma^2 = (0 - 1.5)^2 \cdot \frac{1}{8} + (1 - 1.5)^2 \cdot \frac{3}{8} + (2 - 1.5)^2 \cdot \frac{3}{8} + (3 - 1.5)^2 \cdot \frac{1}{8} = 0.8125$. Varijansa meri prosečno kvadratno *odstupanje* pojedinačnih od očekivane vrednosti μ, a kvadriranjem se eliminiše predznak odstupanja. U praksi se često koristi kvadratni koren varijanse, σ, koji se izražava u istim fizičkim jedinicama kao i slučajna promenljiva X - *standardno odstupanje*.

Raspodela neprekidne slučajne promenljive zadaje se putem funkcije gusitne raspodele (slika 6.1). Za razumevanje koncepta gustine verovatnoće, treba odgovoriti na pitanje "kolika je verovatnoća da neprekidna slučajna promenljiva uzme tačnu vrednost $X = x_i$?" Na primer, treba odrediti verovatnoću da visina slučajno izabranog studenta bude tačno 189 cm. Neka su *sve* visine između 0 i 250 cm. Problem se svodi na izbor tačke $x = 189$ iz realnog intervala $[0, 250]$, načinjen beskonačno tankom iglom. Verovatnoća dobija *geometrijsku* interpretaciju kroz količnik dužine intervala koji sadrži *samo* jednu tačku i dužine intervala $[0, 250]$: $P(X = 189) = 0/250 = 0$.



Slika 6.1: Neprekidna raspodela zadata funkcijom gustine verovatnoće $f(x)$. Verovatnoća za $a \leq X \leq b$, jednaka je površini osenčenog dela ispod krive.

Zbog gore pomenutih razloga, verovatnoća svake *pojedinačne* realizacije neprekidne slučajne promenljive iznosi nula! Međutim, verovatnoća da slučajna promenljiva uzme vrednost iz nekog intervala $[a,b]$ više nije nula, već se može definisati preko gustine raspodele $f(x)$ na sledeći način:

$$P(a \leq X \leq b) = \int_a^b f(x)dx \tag{6.3}$$

Funkcija gustine raspodele *mora* biti nenegativna za sve vrednosti x i pri tome mora biti ispunjeno $P(-\infty \leq X \leq \infty) = \int_{-\infty}^{\infty} f(x)dx = 1$. Matematičko očekivanje i varijansa definišu se na sličan način kao i kod diskretnih raspodela:

$$E[X] = \mu = \int_{-\infty}^{\infty} xf(x)dx \tag{6.4}$$

$$V[X] = \sigma^2 = \int_{-\infty}^{\infty} (x-\mu)^2 f(x)dx \tag{6.5}$$

6.3.2 Monte Karlo simulacija

Posmatra se realni sistem na čije *stanje* i *ponašanje* utiču različite fizičke veličine koje se mogu predstaviti slučajnim promenljivim. Kako se analiza ponašanja sistema često ne može obaviti u realnom svetu (skupa, opasna, dugotrajna, ili neizvodljiva), pribegava se simulaciji na računarskom modelu. Neka su slučajne veličine koje deluju na model označene sa $X_i, i = 1,...,n$. Stanje sistema u nekom trenutku može se opisati promenljivim $S_j = f_j(X_1, X_2, ..., X_n), j = 1,...,m$.[5] Funkcije f_j odnose se na poznata ili pretpostavljena *deterministička* pravila koja prevode ulaze u odgovarajuća stanja. Sistemski izlazi takođe se mogu opisati u funkciji ulaznih i promenljivih stanja.

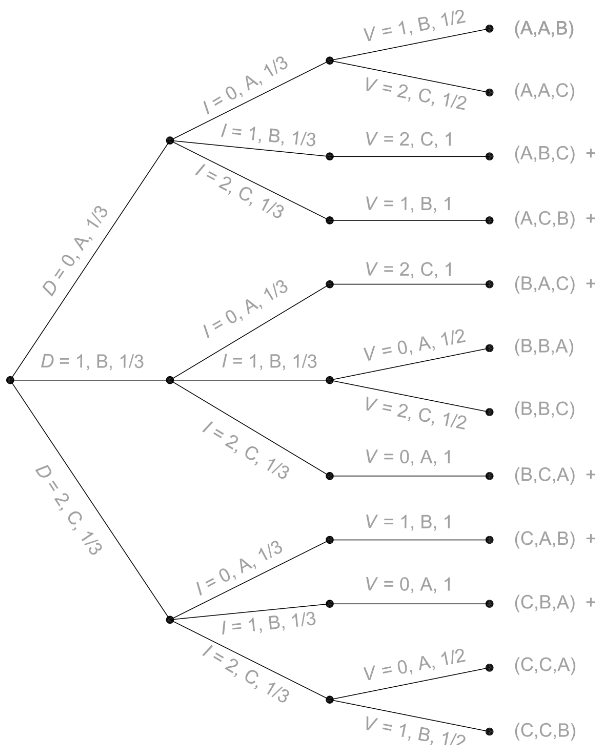
Kada slučajne promenljive X_i dolaze iz *poznatih* ili *pretpostavljenih* raspodela verovatnoća, onda je moguće, uz pomoć *Monte Karlo simulacije*,[6] analizirati ponašanje modela (sistema) pomoću računara. Postupak podrazumeva veliki broj *ponovljenih* opita u kojima se, korišćenjem generatora pseudoslučajnih brojeva, ulazi X_i generišu po odgovarajućim raspodelama. Potom se, u determinističkom postupku, za svaki konkretan skup ulaza izračunava stanje sistema opisano promenljivim S_j. Ako je broj ponovljenih opita dovoljno veliki, stanja (i izlazi) određuju se sa zadovoljavajućom tačnošću. Njima se mogu pridružiti i verovatnoće koje govore o stepenu poverenja da je učinjena greška manja od unapred zadate vrednosti. Sledi primer u kome će rezultati Monte Karlo simulacije biti poređeni sa *analitičkim* modelom koji daje tačno rešenje.

[5] Vremenska dimenzija problema, kao i veze između stanja, zanemareni su zbog jednostavnosti.

[6] Engl. *Monte Carlo simulation*. Metodu je predložio matematičar Stanislav Ulam sredinom XX veka.

Problem 6.3 — Tri kutije i dijamant. Na stolu su tri identične kutije A, B i C, od kojih samo jedna sadrži dijamant. Verovatnoća da se dijamant nađe u nekoj od kutija iznosi $1/3$. Igrač i voditelj započinju igru u kojoj igrač, sa jednakom verovatnoćom, bira neku od kutija. Potom voditelj, koji zna gde je dijamant, otvara jednu (praznu) od dve neizabrane kutije. Ako je igrač odabrao kutiju sa dijamantom, voditelj sa jednakom verovatnoćom bira jednu od preostale dve kutije. U suprotnom, voditelj bira kutiju bez dijamanta. Potom se igrač pita da li menja svoju odluku ili ostaje pri prvobitnom izboru. Po finalnoj odluci, kutija se otvara i igrač zadržava ono što je u njoj. Primenom Monte Karlo metode, analizirati da li je za igrača bolje da promeni početnu odluku. ■

Igra može da se posmatra kao sistem čije krajnje stanje je opisano slučajnim promenljivim D, I i V. Promenljiva D može uzeti jednu od tri diskretne vrednost iz skupa $\{0,1,2\}$, u zavisnosti od toga u kojoj se kutiji nalazi dijamant (0 - A, 1 - B, 2 - C). Slično važi i za preostale slučajne promenljive, s tim da I određuje početni izbor igrača, a V, kutiju koju otvara voditelj. Na primer, za igru u kojoj je $D = 1, I = 1$ i $V = 2$, prelazi se u krajnje stanje (B,B,C) - dijamant se nalazi u kutiji B, igrač je prvo birao B, pa je voditelj otvorio kutiju C. Ako bi igrač promenio svoju odluku da umesto B bira A, onda bi ostao bez dijamanta. Stablo svih mogućih ishoda dato je na slici 6.2.



Slika 6.2: Stablo mogućih ishoda: brojevi označavaju vrednosti slučajnih promenljivih i verovatnoće izbora pojedinih grana. Stanja u kojima igrač dobija dijamant po promeni početne odluke označena su sa +.

U Monte Karlo simulaciji jedne instance igre, bira se prvo slučajna vrednost za D, potom za I i konačno za V. Na ovaj način bira se jedna *putanja* u stablu sa slike 6.2 koja vodi od početnog do krajnjeg čvora. Svi krajnji čvorovi koji su obeleženi sa + predstavljaju dobitak za igrača, pod uslovom da on *promeni* svoju početnu odluku. Da bi se *procenila* verovatnoća dobitka posle promene odluke, treba odigrati *veliki* broj igara i tom prilikom izračunati odnos dobijenih i ukupno odigranih partija. Prilikom izbora vrednosti za slučajne promenljive, treba uočiti da su D i I nezavisne i da se mogu odvojeno generisati pomoću funkcije `randint(0, 2)`. Ova funkcija vraća nula, jedan ili dva, sa jednakom verovatnoćom (1/3), što odgovara definisanim uslovima igre. Međutim, slučajna promenljiva V *nije* nezavisna od prethodne dve promenljive: ako su njihove vrednosti jednake, V uzima jednu od preostale dve vrednosti sa jednakim verovatnoćama (0.5). U suprotnom, V uzima jedinu preostalu vrednost sa verovatnoćom jedan (slika 6.2). Sledi realizacija u Pajtonu:

Program 6.3 — Tri kutije i dijamant.

```
1  # Monte Karlo - gde se krije dijamant
2  from random import randint
3
4  def igra():
5      za_otvaranje = [0, 1, 2]
6      # gde je dijamant (D)
7      D = randint(0,2)
8      za_otvaranje.remove(D)
9      # igracev izbor (I)
10     I = randint(0,2)
11     if I in za_otvaranje:
12         za_otvaranje.remove(I)
13     # voditelj otvara kutiju (V)
14     V = za_otvaranje[randint(0, len(za_otvaranje) - 1)]
15
16     return D, I, V
17
18 # test program
19 print(f'{"n":9}\t{"P_dobitka_po_zameni"}')
20 for n in [100, 1000, 10_000, 100_000, 1_000_000]:
21     zamena_uspela = 0
22     for j in range(n):
23         ishod = igra()
24         if 0 in ishod and 1 in ishod and 2 in ishod:
25             zamena_uspela += 1
26
```

```
27      print(f'{n:<9}\t{zamena_uspela / n}')
```

Igra se simulira funkcijom `igra()` (r4-16) koja vraća stanje definisano slučajnim promenljivim (r16). Uočiti da se, prilikom generisanja vrednosti za slučajnu promenljivu `V`, koristi lista `za_otvaranje` (r5). Iz nje se uklanjaju one vrednosti koje `V` ne može da uzme (r8, 12), a koje su određene vrednostima za `D` i `I`. U programu za testiranje, simulacija igre (r23) ponavlja se `n` puta (r22-25), a na osnovu torke `ishod` određuje se dobitak po promeni početne odluke - igrač dobija dijamant ako su vrednosti svih promenljivih različite (r24-25).

Zavisnost *eksperimentalno* određene verovatnoće dobitka od ukupnog broja simulacija `n` ispisuje se unutar spoljašnje petlje (r27). Tom prilikom formira se pregledna tabela: zaglavljem (r19) je određena širina prve kolone (9) iza koje sledi razmak u vidu tabulatora. Druga kolona nema definisanu širinu. Pri ispisu svakog reda, broj simulacija upisan je u levo poravnato polje širine od 9 karaktera (r27). Zapaziti upotrebu donje crte prilikom navođenja velikih brojeva u r20 – preglednost koda predstavlja bitan cilj u procesu programiranja. Dobijeni su sledeći rezultati:

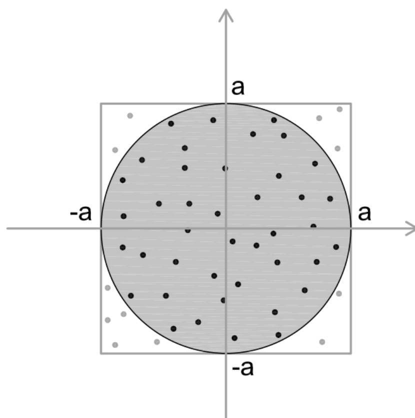
```
n          P_dobitka_po_zameni
100        0.61
1000       0.681
10000      0.6736
100000     0.66811
1000000    0.666421
```

Rezultati pokazuju da, sa porastom broja igara, verovatnoća dobitka pri promeni odluke *konvergira* ka 2/3 (0.6666). Zaključuje se da je bolje *promeniti* početni izbor! Tačna analiza sprovodi se uz pomoć slike 6.2 na sledeći način: verovatnoća uspeha dobija se sabiranjem verovatnoća da se igra završi u nekom od dobitnih čvorova. Verovatnoća stizanja u krajnji čvor jednaka je proizvodu verovatnoća na granama koje vode od startnog do posmatranog čvora. Na primer, $P(AAB) = \frac{1}{3}\frac{1}{3}\frac{1}{2} = \frac{1}{18}$. Tačna verovatnoća uspeha pri promeni odluke iznosi $2/3$, što se slaže sa rezultatom simulacije. Primetiti da bi sledeće razmišljanje bilo *pogrešno*: u stablu sa slike 6.2 ima 12 krajnjih stanja od kojih je 6 (označenih sa +) povoljno za promenu kutije – sledi da je verovatnoća dobitka $6/12 = 0.5$. Greška je u tome da sva stanja *nisu* podjednako verovatna pa količnik $6/12$ nema smisla (na primer, $P(AAB) \neq P(CAB)$)!

Metoda Monte Karlo može se koristiti i za potrebe rešavanja različitih matematičkih problema. Sledi primer izračunavanja površine za zatvorenu konturu u ravni.

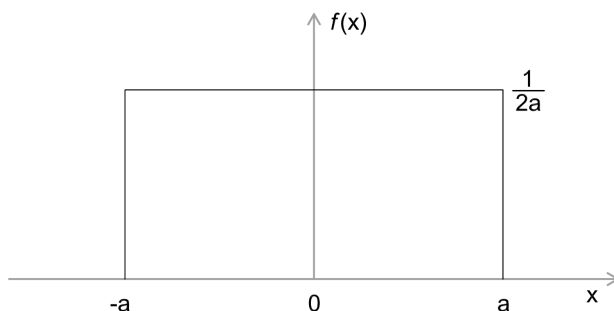
Problem 6.4 — Površina kruga i zaboravljeno π. Dat je krug K poluprečnika a sa centrom u $(0,0)$. Izračunati površinu kruga K bez upotrebe broja π! ■

Površina kruga K može se odrediti na sledeći način: pretpostaviti da krug *u celosti* pripada nekoj oblasti u ravni sa poznatom površinom. Neka to, radi jednostavnosti, bude kvadrat stranice $2a$ koji je opisan oko kruga (slika 6.3). Kada bi odnos između površine kvadrata $P_{kvadrat}$ i površine kruga P_K bio poznat ($c = \frac{P_K}{P_{kvadrat}}$), onda bi se tražena površina odredila uz pomoć izraza $P_K = cP_{kvadrat}$.



Slika 6.3: Određivanje površine kruga bez upotrebe broja π, primenom Monte Karlo metode *pogodak-promašaj*.

Posmatra se slučajno odabrana tačka (x,y) iz opisanog kvadrata. Koordinate slučajne tačke mogu se dobiti uz pomoć generatora pseudoslučajnih brojeva, koristeći se funkcijom `uniform(-a, a)` koja vraća uniformno raspoređeni realni broj iz intervala $[-a,a]$ (neprekidna slučajna promenljiva, slika 6.4). Verovatnoća da ovako izabrana tačka istovremeno pripada i krugu jednaka je odnosu površina kruga i kvadrata.



Slika 6.4: Uniformna raspodela: generisani brojevi ravnomerno su raspoređeni od $-a$ do a. Verovatnoće da broj padne u bilo koji interval širine δ jednake su i iznose $\delta/2a$ (površina pravougaonika sa stranicom δ ispod krive $f(x)$).

Verovatnoća $c = \frac{P_K}{P_{kvadrat}}$ može se proceniti Monte Karlo simulacijom. U simulaciji se prvo generiše veliki broj slučajnih tačaka koje pripadaju kvadratu. Potom se utvrđuje koliko njih pripada istovremeno i krugu K. Prema zakonu velikih bro-

jeva, kada broj generisanih tačaka raste, odnos se *približava* traženoj verovatnoći. Za testiranje pripadnosti tačke (x_i, y_i) krugu $K : x^2 + y^2 = a^2$, koristi se sledeći uslov: $x_i^2 + y_i^2 \leq a^2 \implies (x_i, y_i) \in K$.

Program 6.4 — Površina kruga i zaboravljeno π.

```
# Površina kruga bez PI - Monte Karlo metoda!
from random import uniform
from math import pi

def površina(a, n):
    ''' Monte Karlo metoda: površina kruga poluprečnika a,
        n slučajnih tačaka u [-a, a]'''
    unutra = 0
    for i in range(n):
        x, y = uniform(-a, a), uniform(-a, a)
        if x**2 + y**2 <= a**2:
            unutra += 1
    return 4 * a**2 * unutra / n

# test primer
a = float(input('poluprečnik? '))
p_stvarno = a**2 * pi

print(f'{"n":<9}\t{"Pk":<9}\t{"rel. greška":<9}')
for n in [100, 1000, 10_000, 100_000, 1_000_000]:
    p = površina(a, n)
    rel = abs(p - p_stvarno) / p_stvarno # relativna greška
    print(f'{n:<9}\t{p:<9}\t{rel:<9}')
```

Funkcija `površina()` realizuje opisanu metodu Monte Karlo sa `n` slučajno generisanih tačaka (r5-13). Za svaku generisanu tačku koja pripada krugu K, promenljiva `unutra` uvećava se za jedan (r12). Tražena površina dobija se kao proizvod površine kvadrata i odnosa broja tačaka u krugu i kvadratu (r13). Površina kruga, određena metodom Monte Karlo, razlikuje se od tačne vrednosti izračunate uz pomoć broja π, što se može videti na osnovu rada test programa (r16-23). Broj generisanih tačaka u svakoj iteraciji testa povećava se deset puta pa je za očekivati da se relativna greška smanjuje (r22). Sledi prikaz rada programa:

```
poluprečnik? 10
n          Pk         rel. greška
100        316.0      0.005859240340778516
1000       308.4      0.018332311009189646
10000      314.56     0.0012755779797319375
100000     314.692    0.001695747029494557
1000000    314.212   0.00016785957581230386
```

Opisani program može se upotrebiti za izračunavanje zaboravljene vrednosti za π. Tražena vrednost jednaka je površini jediničnog kruga!

ŠTA JE NAUČENO

- Nedeterministički algoritmi, za razliku od determinističkih, za iste kombinacije ulaznih veličina mogu proizvesti različite rezultate. Tom prilikom oni koriste izvor slučajnosti - generator slučajnih ili pseudoslučajnih brojeva.
- Slučajni brojevi mogu se dobiti iz fizičkih izvora slučajnosti, obradom toplotnog ili elektromagnetnog šuma. Uobičajeno se koriste za šifrovanje podataka u sigurnosnim aplikacijama.
- Sekvenca pseudoslučajnih brojeva generiše se u determinističkom postupku. Pravilo po kome se generišu pseudoslučajni brojevi izuzetno teško se uočava, a brojevi iz sekvence uniformno su raspoređeni unutar željenog intervala.
- U Pajtonu se, za potrebe generisanja pseudoslučajnih brojeva, koristi modul `random`. Ovaj modul pruža i funkcije za slučajni odabir i mešanje uzorka iz populacije objekata koji se nalaze u nekoj kolekciji.
- Probabilistički algoritmi spadaju u klasu nedeterminističkih algoritama. Oni ne garantuju postizanje tačnog rešenja pa se primenjuju za rešavanje kompleksnih problema kada približna rešenja postaju prihvatljiva.
- Ponašanje realnih sistema može se analizirati uz pomoć Monte Karlo simulacije. Monte Karlo metode koriste pseudoslučajne brojeve za modelovanje različito raspoređenih slučajnih veličina kojima se opisuje neki sistem. Izlazi iz simulacije imaju probabilistički karakter, a tačnost analize obično raste sa porastom broja simuliranih događaja.

7. Složenost algoritma. Pretraživanje i sortiranje

Poređenje dva algoritma po *efikasnosti* predstavlja ključnu aktivnost u računarskim naukama. U tu svrhu sprovodi se analiza *algoritamske složenosti*. Algoritamska složenost može se, u zavisnosti od konteksta, definisati na različite načine. Na primer, poželjno je da se trajanje obrade ne produžava *značajno* sa porastom veličine ulaznih podataka, kao i da algoritam troši što manje memorijskih i ostalih računarskih resursa. Centralno mesto u analizi složenosti zauzima *funkcija složenosti* putem koje se, u zavisnosti od veličine ulaznih podataka, opisuje ponašanje proizvoljnog algoritma. Naročito je zanimljivo posmatrati granično ponašanje ove funkcije kada dimenzionalnost problema, okarakterisana veličinom ulaznih podataka, teži beskonačnosti - asimptotske mere složenosti.

Rešavanje praktičnih problema često iziskuje *pretraživanje* odgovarajuće objektne kolekcije, kako bi se pronašli objekti zadatih karakteristika. Bilo da se radi o utvrđivanju prisutnosti traženog broja u listi brojeva ili prikazivanju grupe veb stranica koje sadrže navedeni tekst, program treba da poseduje *efikasan* mehanizam za pronalaženje željenih podataka. Pored pretraživanja, izabrane objekte često treba *sortirati* prema unapred utvrđenom kriterijumu. Sortiranje predstavlja postupak preuređivanja sekvence objekata tako da se elementi poređaju po *rastućim* ili *opadajućim* vrednostima posmatrane osobine. Na primer, sve veb stranice koje sadrže tekst *Python* treba prikazati u opadajućem poretku njihove mere relevantnosti[1] - prvo se prikazuju relevantnije

[1] Internet pretraživači koriste različite mere za ocenu relevantnosti. Google je postao poznat zahvaljujući meri *Page Rank*, koja favorizuje stranice referencirane od strane relevantnijih stranica.

stranice. Sortiranje će se, bez gubitka opštosti, posmatrati u kontekstu preuređivanja proizvoljne liste brojeva u rastući (opadajući) redosled. U ovoj glavi razmatraju se osnovni algoritmi za pretraživanje i sortiranje, a posebna pažnja usmerena je na postupak analize njihove složenosti.

Pored sprovedene analize složenosti koja koristi asimptotske mere, algoritmi pretraživanja i sortiranja testiraće se u realnim uslovima na odgovarajućim skupovima ulaznih podataka. Određivanje trajanja izvršavanja algoritma, za različite kombinacije ulaznih podataka, iziskuje poznavanje postupaka za *merenje vremena*. Izlaganje započinje opisom ovih postupaka.

7.1 Merenje vremena

Analiza složenosti programa, ili pojedinih njegovih delova, podrazumeva mogućnost merenja proteklog vremena između dva događaja u programskom toku. Funkciju merenja vremena u računarskom sistemu obavljaju dva hardverska časovnika: *sat realnog vremena* i *sistemski sat*. Sat realnog vremena (RTC[2]) najčešće je realizovan u obliku integrisanog kola na matičnoj ploči sistema. RTC poseduje bateriju koja omogućava merenje vremena i kada je računar ugašen, a može se, ako je računar u okviru mreže, dodatno usaglasiti sa preciznim spoljnim časovnicima na Internetu. Sistemski sat realizovan je kao programabilni tajmer koji periodično, nakon predefinisanog intervala, generiše signal procesoru da odbroji jednu *jedinicu sistemskog vremena*.[3] Trajanje jedinice sistemskog vremena zavisi od hardverske implementacije i obično iznosi milisekund, mikrosekund ili, čak, nanosekund. Postojanje sistemskog sata omogućava sinhronizovanje (i merenje) raznorodnih procesa u okviru računarskog sistema koji traju znatno kraće nego oni koji se mere uobičajenim časovnikom. Po uključivanju računara, procesor konvertuje vreme očitano sa RTC-a u broj vremenskih jedinica proteklih od referentnog datuma u prošlosti. Potom se sistemski sat, u toku rada sistema, brine o uvećavanju ovog broja. Tako dobijena vrednost reprezentuje *sistemsko vreme*, a referentni trenutak u prošlosti naziva se *epohom*. Mnogi operativni sistemi za epohu uzimaju 1. januar 1970, u 00:00:00 po *univerzalnom vremenu*. Univerzalno vreme ne meri se časovnikom već se određuje astronomskim proračunom pri posmatranju kretanja zvezda.

Merenje vremena u Pajtonu realizuje se uz pomoć modula `time` čije su najčešće korišćene funkcije prikazane u tabeli 7.1. Sledi primer korišćenja ovih funkcija:

```
>>> time.gmtime()   # univerzalno vreme
time.struct_time(tm_year=2017, tm_mon=4, tm_mday=2, tm_hour=17,
tm_min=22, tm_sec=27, tm_wday=6, tm_yday=92, tm_isdst=0)
>>> time.localtime() # lokalno vreme (Beograd)
```

[2] Engl. *Real Time Clock*.
[3] Engl. *Time tick*.

```
time.struct_time(tm_year=2017, tm_mon=4, tm_mday=2, tm_hour=19,
tm_min=31, tm_sec=30, tm_wday=6, tm_yday=92, tm_isdst=1)
>>> time.asctime(time.localtime())  # tekstualni prikaz lok. vremena
'Sun Apr  2 19:31:58 2017'
>>> time.asctime(time.gmtime(0))  # tekstualni prikaz epohe
'Thu Jan  1 00:00:00 1970'
>>> time.time()  # broj proteklih sekundi od epohe
1491154398.6359384
>>> time.sleep(1)  # pauzira sa radom jednu sekundu
>>> time.time(), time.sleep(1), time.time()   # primer za sleep
(1491155266.9192047, None, 1491155267.9215555)
```

funkcija	opis
`gmtime()`	vraća objekat koji reprezentuje tekuće univerzalno vreme
`localtime()`	vraća objekat koji reprezentuje tekuće lokalno vreme
`asctime()`	vraća tekstualnu reprezentaciju vremenskog objekta
`sleep()`	pauzira izvršavanje programa za navedeni broj sekundi
`time()`	vraća broj proteklih sekundi od epohe koja se određuje sa `gmtime(0)`
`perf_counter()`	precizno meri trajanje pojedinih delova programa

Tabela 7.1: Modul `time` - funkcije za merenje vremena.

Funkcije `gmtime()` i `localtime()` vraćaju objekat tipa `struct_time` koji sadrži devet celobrojnih vrednosti: godinu, mesec u godini, dan u mesecu, sat, minut, sekund, dan u nedelji (0 za ponedeljak), dan u godini, te da li se primenjuje korekcija vremena (1 - da, 0 - ne, -1 - nepoznato). Uočava se razlika od plus dva sata za lokalno vreme u Beogradu, u odnosu na univerzalno vreme. Funkcija `asctime()` prevodi objekat tipa `struct_time` u tekstualnu reprezentaciju pogodnu za ispis.

Vreme proteklo između dva događaja može se odrediti uz pomoć funkcije `time()` koja vraća broj sekundi proteklih od epohe (1. januar 1970, u 00:00:00). Njena primena ilustrovana je uz pomoć funkcije `sleep()` koja *zaustavlja* rad programa za navedeno vreme u sekundama. Funkcija `time()` vraća realan broj, ali njena preciznost *nije* veća od sekunde. Ako je vreme između dva događaja *veoma kratko*, preporučuje se upotreba funkcije `perf_counter()` koja meri vreme između dva uzastopna poziva najpreciznijim satom u sistemu. Jedan poziv ove funkcije vraća realnu vrednost koja *nije* vezana za realno vreme, ali se razlika vrednosti iz dva sukcesivna poziva odnosi na trajanje intervala u delovima sekunde. U sledećem primeru poredi se trajanje dva postupka za rekurzivno izračunavanje članova fibonačijevog niza:

Problem 7.1 — Fibonačijevi brojevi - dinamičko programiranje. Rekurentna formula za n-ti član fibonačijevog niza je $F(n) = F(n-1) + F(n-2)$. Porediti trajanje izračunavanja $F(n)$ u verzijama bez i sa pamćenjem prethodno izračunatih članova. ■

Program 7.1 — Fibonačijevi brojevi - dinamičko programiranje.

```
# Poređenje dve rekurzivne verzije za računanje Fib(n)
import time
# klasičan rekurzivni algoritam
def fib(n):
    if n == 0 or n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)

# rekurzivni algoritam sa pamćenjem
memorija = {}
def fib_memo(n):
    if n == 0 or n == 1:
        return 1
    elif n in memorija:
        return memorija[n]
    else:
        memorija[n] = fib_memo(n-1) + fib_memo(n-2)
        return memorija[n]
# unos
n = int(input('unesite n '))
# bez pamćenja
start = time.perf_counter()
f = fib(n)
t1 = time.perf_counter() - start
print(f'F({n}) = {f} za {t1:10.9f} sekundi')

# sa pamćenjem
start = time.perf_counter()
f = fib_memo(n)
t2 = time.perf_counter() - start
print(f'Fmemo({n}) = {f} za {t2:10.9f} sekundi')

print(f'Pamćenje skraćuje trajanje {t1/t2:6.1f} puta!')
```

Standardni rekurzivni postupak za izračunavanje n-tog člana fibonačijevog niza realizovan je uz pomoć funkcije `fib()` (r4-8). Funkcija `fib_memo()` (r12-19) koristi globalni rečnik `memorija` (r11) da *upamti* prethodno izračunate članove. Ubrzanje u

odnosu na klasični pristup postiže se tako što se, pri izračunavanju $F(n)$, proverava da li je traženi n-ti član već izračunat (r15). Ako jeste, funkcija vraća vrednost pohranjenu u rečniku (r16). U suprotnom, $F(n)$ se izračunava uz pomoć prethodna dva člana, a dobijena vrednost stavlja u rečnik (r18) pa potom vraća kao rezultat rada funkcije (r19). Treba uočiti da se globalna promenljiva `memorija` ne deklariše naredbom `global`, pošto se u r18 *ne menja* identitet rečnika na koji ona ukazuje, već se unutar istog ubacuje novo pridruživanje ključ-vrednost.

Testiranje trajanja izvršavanja iziskuje *jednake uslove* za obe funkcije. Zbog toga se merenja obavljaju za istu vrednost n (r21). Prvo se meri trajanje klasičnog pristupa (r23-26), pa potom unapređene verzije koja uključuje pamćenje prethodno izračunatih vrednosti (r29-32). Prikaz rada programa za dve vrednosti n dat je ispod:

```
unesite n 10
F(10) = 89 za 0.000083753 sekundi
Fmemo(10) = 89 za 0.000026162 sekundi
Pamćenje skraćuje trajanje    3.2 puta!
>>>
==== RESTART: C:\programi\p7_1.py ====
unesite n 30
F(30) = 1346269 za 0.240836777 sekundi
Fmemo(30) = 1346269 za 0.000016916 sekundi
Pamćenje skraćuje trajanje 14237.2 puta!
```

Skraćenje trajanja postupka, koje se postiže pamćenjem prethodno izračunatih članova, značajno raste sa porastom broja n.[4] Prethodna realizacija predstavlja primer upotrebe rečnika za ubrzavanje rada programa, a primenjeni pristup u rešavanju problema naziva se još i *dinamičko programiranje*. U dinamičkom programiranju problem se deli na potprobleme čija se rešenja *pamte* pa se, kada je to potrebno, umesto ponovnog rešavanja upotrebljavaju *zapamćene* vrednosti. Dinamičko programiranje može se primeniti samo ako između potproblema postoji izvesno *preklapanje*. Na primer, i za $F(n)$ i za $F(n-1)$, potrebno je izračunati $F(n-2)$.

Pogodno odabrane strukture podataka predstavljaju jedan od ključnih faktora za dobre performanse rešenja. Ipak, skraćivanje vremena izvršavanja dobija se često nauštrb povećanog korišćenja memorijskih resursa.[5]

U prethodnom primeru, uvođenjem rečnika, program troši dodatnu memoriju za pamćenje n celobrojnih objekata.

[4] Prikazana vremena razlikovaće se od računara do računara u zavisnosti od hardverske i softverske konfiguracije. Analiza vremenske kompleksnosti koja ne zavisi od konkretne konfiguracije biće izložena u glavi 7.2.

[5] Engl. *Space-time tradeoff* - prostorno-vremenska nagodba.

7.2 Analiza složenosti algoritma

Cilj analize složenosti je da utvrdi ponašanje algoritma za različite kombinacije ulaznih veličina kako bi se omogućilo poređenje sa drugim rešenjima, u smislu *utroška* vremenskih i operativnih resursa. Tom prilikom određuje se analitička funkcija koja uspostavlja vezu između veličine problema (ulaznih podataka) i vremena izvršavanja (ili utrošenih memorijskih resursa). Ova funkcija naziva se *funkcijom složenosti* algoritma. U daljem tekstu akcenat se stavlja na *vreme izvršavanja*, dok će utrošak memorijskih resursa biti analiziran samo povremeno. Ova odluka opravdava se činjenicom da su uobičajeni kapaciteti standardnih operativnih memorija sve veći, a njihova cena sve niža.

7.2.1 Funkcija vremenske složenosti

Posmatraju se dva algoritma, A i B, za izračunavanje sume prvih n prirodnih brojeva, zajedno sa programom za testiranje kojim se meri trajanje njihovog izvršavanja:

```python
# Poređenje dva algoritma za sumiranje prvih n prir. brojeva
from time import perf_counter

def sumaA(n): # Algoritam A
    return n * (n + 1) // 2

def sumaB(n): # Algoritam B
    suma = 0
    for i in range(1, n+1):
        suma += i
    return suma

# test program za A
for n in [100, 1000, 10_000, 100_000, 1_000_000]:
    start = perf_counter()
    s = sumaA(n)
    t = perf_counter() - start
    print(f'A: n={n:9} s={s:13} za {t:10.9f} sec')

# test program za B
for n in [100, 1000, 10_000, 100_000, 1_000_000]:
    start = perf_counter()
    s = sumaB(n)
    t = perf_counter() - start
    print(f'B: n={n:9} s={s:13} za {t:10.9f} sec')
```

```
A: n=      100 s=         5050 za 0.000006932 sec
A: n=     1000 s=       500500 za 0.000004912 sec
A: n=    10000 s=     50005000 za 0.000005637 sec
A: n=   100000 s=   5000050000 za 0.000006257 sec
A: n=  1000000 s= 500000500000 za 0.000006159 sec
B: n=      100 s=         5050 za 0.000026502 sec
B: n=     1000 s=       500500 za 0.000158585 sec
B: n=    10000 s=     50005000 za 0.001127290 sec
B: n=   100000 s=   5000050000 za 0.008428843 sec
B: n=  1000000 s= 500000500000 za 0.056572881 sec
```

Uočava se da je vreme izvršavanja algoritma A približno *konstantno*, odnosno da *ne zavisi* od n. Algoritam B je očigledno sporiji: kada se n poveća deset puta, njegovo vreme izvršavanja približno se udesetostručuje! Može se još reći da trajanje izvršavanja algoritma B *linearno* raste sa porastom broja n. Funkcija vremenske složenosti, $t(n)$, određuje vezu između trajanja algoritma i veličine ulaznog podatka n. Ona se može odrediti uvidom u izvorni kod. Pretpostavlja se da trajanje aritmetičko-logičkih operacija *ne zavisi* od veličine operanada, kao i da pristup radnoj memoriji, zarad čitanja ili upisa, *ne zavisi* od adrese memorijske lokacije. Posmatra se prvo algoritam A pa potom B.

Funkcija `sumaA()` sastoji se od samo jednog programskog reda. Izračunavanje izraza i vraćanje njegove vrednosti, u r5, obavlja se za neko konstantno vreme c pa je $t_A(n) = c$. Funkcija složenosti za `sumaB()` može se izraziti kao zbir trajanja pojedinačnih redova od kojih se neki ponavljaju, a neki izvršavaju samo jednom. Neka je c_1 trajanje dodeljivanja vrednosti u r8, c_2 trajanje izvršavanja naredbe kojom se realizuje petlja u r9, c_3 trajanje postupka sumiranja u r10, a c_4 trajanje izvršavanja naredbe `return` u r11. Kako se redovi r9 i r10 ponavljaju n puta, funkcija $t_B(n)$ data je sledećim izrazom:

$$t_B(n) = c_1 + n(c_2 + c_3) + c_4 = an + b \tag{7.1}$$

Prilikom poređenja algoritama A i B, porede se njihove funkcije složenosti $t_A(n)$ i $t_B(n)$. Od interesa je porediti *brzine rasta* ove dve funkcije sa porastom ulazne veličine n. Efikasniji je onaj algoritam čija funkcija složenosti *sporije* raste u opštem slučaju, za proizvoljno veliko n. Očigledno, u primeru izračunavanja sume prvih n prirodnih brojeva, važi da je $t_B(n) > t_A(n)$ za $n > \frac{c-b}{a}$. Međutim, kako aritmetičko-logičke operacije različito traju na različitim računarskim sistemima (konstante iz (7.1)), neophodno je definisati *univerzalan* pristup u analizi funkcije složenosti koji *neće zavisiti* od hardverskih i softverskih karakteristika ciljnog sistema.

Univerzalan pristup za formiranje funkcije podrazumeva sledeća pojednostavljenja:

- trajanje izvršavanja algoritma za određenu dimenziju problema treba izraziti preko broja *dominantnih* operacija u algoritmu. U primeru sumiranja, dominantna operacija je sabiranje.

- funkciju složenosti treba posmatrati u *graničnom* slučaju kada brojnost ulaznih podataka teži beskonačnosti.

Algoritmi u praksi barataju sa kompleksnim ulaznim podacima koji, kao što je već spomenuto u glavi 6, dolaze iz različitih raspodela verovatnoće. Zato se funkcije složenosti mogu posmatrati u tri različite situacije: u *najpovoljnijem*, *prosečnom* i *najnepovoljnijem* slučaju. Neka, za primer, treba proveriti da li se traženi broj nalazi u celobrojnoj listi tako što se kroz nju prolazi element po element. Ovaj problem uskoro će biti detaljno razmatran. Najpovoljniji slučaj odnosi se na situaciju u kojoj se traženi broj nalazi na samom početku liste. Ako lista ne sadrži traženi broj, onda nastupa najnepovoljniji slučaj – da bi se to utvrdilo, mora se proći kroz celu listu. Prosečan slučaj podrazumeva da se broj nalazi u središnjem delu liste. Međutim, realni podaci često nisu raspoređeni na način kako se to pretpostavlja u analizi pa se funkcija složenosti za prosečan slučaj najčešće veoma teško izračunava. U daljem izlaganju razmatraće se *najnepovoljnije* ponašanje algoritma. Najnepovoljnije ponašanje određuje krajnji domet algoritma što je od velikog značaja za njegovu praktičnu primenu.

Da bi se prilikom poređenja dva algoritma zanemarili efekti računarske konfiguracije, porede se *redovi rasta* njihovih funkcija složenosti. Neka algoritam A, koji se izvršava na *brzom* računaru, u najnepovoljnijem slučaju ima funkciju složenosti $t_A(n) = an^2 + bn + c$, a algoritam B, koji se izvršava na *sporom* računaru, $t_B(n) = dn + e$. Kako bi se uklonili efekti konstanti koje proizilaze iz odgovarajućeg okruženja, može se posmatrati ponašanje količnika $\frac{t_B(n)}{t_A(n)}$ u *asimptotskom* slučaju kada $n \to \infty$:

$$\lim_{n\to\infty} \frac{t_B(n)}{t_A(n)} = \frac{dn+e}{an^2+bn+c} = 0$$

Iako se B izvršava na sporijem računaru, njegova efikasnost u odnosu na A u najnepovoljnijem slučaju je daleko veća jer, za veliko n, u $t_A(n)$ *dominira* kvadratni član!

! Nezavisno od činjenice da su računari sve brži, pametan programer koji će osmisliti efikasan algoritam i dalje je na ceni! Dobar algoritam i spor računar *bolji* su spoj od lošeg algoritma i brzog računara.

7.2.2 Asimptotske notacije

Ideja da se, prilikom analize funkcije složenosti, posmatraju asimptotski slučajevi biće ukratko izložena u naredne dve sekcije.

Notacija O(n) – "veliko o"

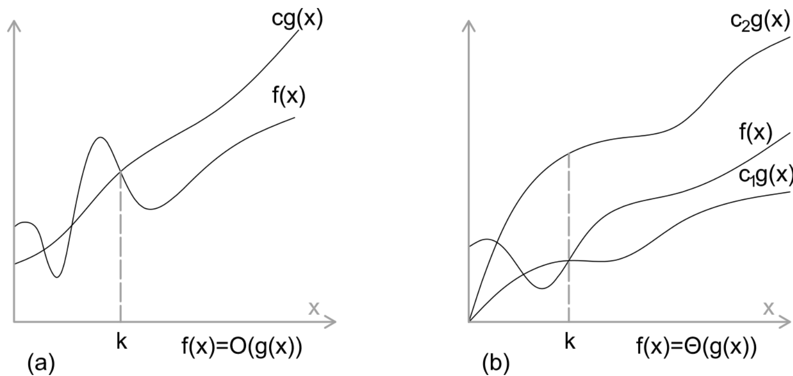
Posmatra se funkcija složenosti algoritma $f : \mathbb{R}^+ \to \mathbb{R}^+$. Oznaka f izabrana je umesto prethodno korišćene oznake t da naglasi opštost analize koja sledi - f može označavati vremensku, ali i neku drugu vrstu složenosti (na primer memorijsku). Domen

funkcije postaje skup pozitivnih realnih brojeva kojim se može kvantifikovati veličina proizvoljnog skupa algoritamskih ulaza. Funkcija f pripada klasi funkcija $\mathrm{O}(g)$ (kaže se još i "f je $\mathrm{O}(g)$"), ako postoje pozitivne relane konstante c i k za koje važi:

$$\forall x > k, \;\; f(x) \leq cg(x) \tag{7.2}$$

Iz izraza (7.2) sledi da je, za dovoljno veliko x, funkcija f *ograničena odozgo* funkcijom $cg(x)$, odnosno da ne raste brže od $cg(x)$ (slika 7.1a).

Vreme izvršavanja algoritma za veličinu ulaza n, a čija funkcija složenosti pripada $\mathrm{O}(g)$, nikad nije veće od $cg(n)$! Na primer, za $t(n) = 2n+3$ i $g(n) = n$, važi $t(n) \leq 3g(n)$ za $n > 3$, pa je t u klasi $\mathrm{O}(n)$.



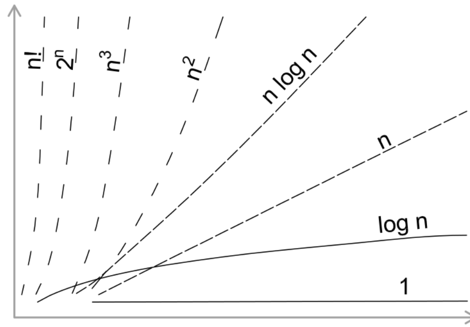
Slika 7.1: Asimptotske notacije: (a) funkcija f pripada klasi $\mathrm{O}(g)$; (b) funkcija f pripada istovremeno klasi $\mathrm{O}(g)$ i klasi $\Omega(g)$, pa se kaže da je f "reda $\Theta(g)$".

Za funkcije složenosti, u pogledu notacije "veliko o", važe sledeći iskazi:

- f je $\mathrm{O}(f)$.
- ako je f u $\mathrm{O}(g)$ i g u $\mathrm{O}(f)$, onda su f i g istog *reda složenosti*.
- ako je f u $\mathrm{O}(g)$ i $\forall x > 0, \;\; h(x) \geq g(x)$, onda važi i f u $\mathrm{O}(h)$.

Iz poslednjeg iskaza sledi da je funkcija reda složenosti $\mathrm{O}(n)$ ujedno i reda $\mathrm{O}(n^2)$, ali ne i obrnuto (slika 7.2). Algoritamsku složenost poželjno je okarakterisati *što jednostavnijom* funkcijom što nižeg reda. Na primer, neka algoritam ima linearnu složenost u najnepovoljnijem slučaju (vreme raste linearno sa povećanjem dimenzije ulaza). Tada je *informativnije* da se ponašanje algoritma okarakteriše redom složenosti $\mathrm{O}(n)$, iako bi bilo tačno i $\mathrm{O}(n^2)$.

Asimptotska analiza tipa "veliko o" može se pojednostaviti tako što se svi članovi, sem dominantnog, mogu ukloniti iz razmatranja. Slično, mogu se ukloniti i sve konstante pa se algoritmi mogu porediti nezavisno od osobina ciljnog računarskog sistema.



Slika 7.2: Redovi složenosti: funkcije složenosti pojedinih familija algoritama i njihov međusobni odnos. Pune linije reprezentuju veoma efikasne, crtkaste - efikasne, a isprekidane - neefikasne algoritme.

Notacije $\Omega(n)$ i $\Theta(n)$

Funkcija složenosti $f : \mathbb{R}^+ \to \mathbb{R}^+$ pripada klasi funkcija $\Omega(g)$ (kaže se još i "f je $\Omega(g)$"), ako postoje realne pozitivne konstante c_1 i k za koje važi:

$$\forall x > k, \quad f(x) \geq c_1 g(x) \tag{7.3}$$

Iz izraza (7.3) sledi da je, za dovoljno veliko x, funkcija f *ograničena odozdo* funkcijom $c_1 g(x)$, odnosno da ne raste sporije od $c_1 g(x)$ (slika 7.1b). Pored notacije "veliko o", u analizi algoritamske složenosti često se koristi i notacija $\Theta(n)$ koja je po svojoj suštini *najinformativnija* jer određuje kako gornju, tako i donju granicu rasta funkcije složenosti. Za funkciju $f : \mathbb{R}^+ \to \mathbb{R}^+$ kaže se da je $\Theta(g)$ ako je u isto vreme i $O(g)$ i $\Omega(g)$, odnosno ako postoje pozitivne realne konstante k, c_1 i c_2 za koje važi:

$$\forall x > k, \quad c_1 g(x) \leq f(x) \leq c_2 g(x) \tag{7.4}$$

U daljem izlaganju najčešće će biti korišćena notacija "veliko o".

7.3 Pretraživanje

Problem pretraživanja može se, u opštem slučaju, formulisati na sledeći način: zadata je kolekcija objekata K i objekat x. Potrebno je ustanoviti da li K sadrži objekat koji je po svojoj vrednosti, ili na drugi zadati način, jednak sa x. U ovoj glavi biće reči o pretraživanju uređene kolekcije tipa liste, a objekti od interesa biće celi brojevi. Preformulisani problem pretraživanja glasi: data je celobrojna lista L i celobrojni objekat x; treba pronaći indeks elementa u L koji je jednak po vrednosti sa x. Ako takav element ne postoji, vraća se `None`. Nadalje se obrazlažu dva osnovna pristupa u rešavanju problema pretraživanja: *linearni* i *binarni*. Dobro poznati operator `in` primenjuje linearnu pretragu kako bi utvrdio prisustvo elementa u sekvenci.

7.3.1 Linearno pretraživanje

Linearno pretraživanje zasniva se na jednostavnoj ideji uzastopne provere elemenata liste, počevši od prvog, kako bi se utvrdilo da li je tekući element jednak traženom broju. Ako se utvrdi jednakost, postupak se završava uz vraćanje indeksa tekućeg elementa. U suprotnom, nastavlja se sa proverom narednih elemenata. Ako se posle provere svih elemenata broj ne pronađe, vraća se objekat tipa `None`. Sledeći program ispituje zavisnost dužine trajanja linearnog pretraživanja od broja elemenata u listi. Razmatraju se tri slučaja: *najnepovoljniji* - traženi broj nije u listi, *prosečni* - traženi broj je na prozvoljnom mestu i *najpovoljniji* - traženi broj je prvi element liste:

Program 7.2 — Linearno pretraživanje.

```
1  # Linearno pretraživanje
2  from random import shuffle
3  from time import perf_counter
4
5  def lin_pretraga(L, x):
6      '''Pronalazi indeks x u L. Vraća None, ako x nije u L'''
7      i, n = 0, len(L)
8      while i < n and L[i] != x:
9          i += 1
10
11     return i if i < n else None
12
13 # testiranje
14 print(f"{'n':6}\t{'Tnep':8}\t{'Tprs':8}\t{'Tpov':8}")
15 for n in [10_000,20_000, 30_000, 40_000, 50_000, 60_000, 70_000]:
16     L = list(range(n))
17
18     # najnepovoljniji slučaj
19     t_start = perf_counter()
20     lin_pretraga(L, n)
21     t_nep = perf_counter() - t_start
22
23     # najpovoljniji slučaj
24     t_start = perf_counter()
25     lin_pretraga(L, 0)
26     t_pov = perf_counter() - t_start
27
28     # procečan slučaj
29     t_prs = 0
```

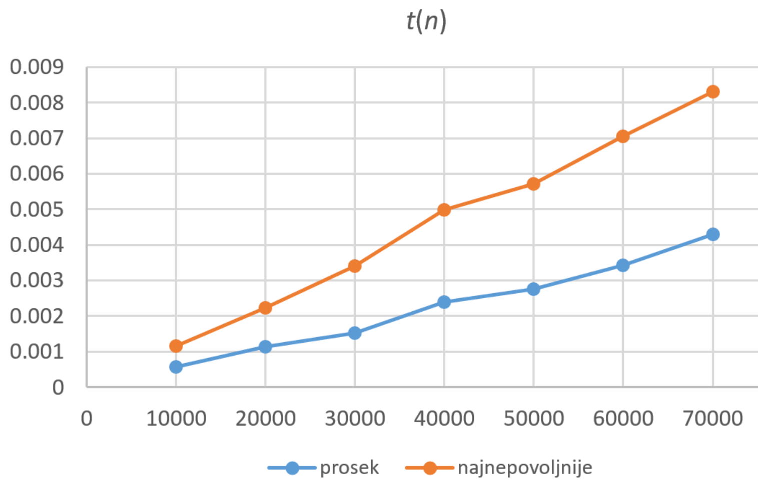
```
30      for j in range(100):
31          shuffle(L)     # promešaj listu
32          t_start = perf_counter()
33          lin_pretraga(L, 1)
34          t_prs += perf_counter() - t_start
35
36      print(
37       f'{n:<6d}\t{t_nep:<8.6f}\t{t_prs/100:<8.6f}\t{t_pov:<8.6f}')
```

U glavnoj petlji programa za testiranje (r15-16) generišu se liste od `n = 10,20,30, 40,50,60 i 70` hiljada elemenata. Formirane liste sadrže sve brojeve od 0 do `n-1`, složene u rastući poredak (r16). Prilikom razmatranja prosečnog trajanja traži se jedinica, a postupak pretraživanja ponavlja se 100 puta za liste svih dužina (r29-34). U svakoj iteraciji upotrebljena je funkcija za mešanje ulazne liste `shuffle()` kako bi se jedinica našla na različitim mestima u listi (r31). Promenljiva `t_prs` akumulira vreme trajanja postupka u svakoj iteraciji pa se na kraju može prikazati *usrednjeno* trajanje (r36-37). Iterativni postupak i mešanje nisu neophodni za određivanje vremena najnepovoljnijeg (r19-21), odnosno najpovoljnijeg slučaja (r24-26). U ovim slučajevima traži se nepostojeći broj (`n`), odnosno prvi element iz liste (`0`). Primetiti upotrebu uslovnog izraza u r11 (videti glavu 3.1.1). Program za testiranje daje sledeći ispis:

```
n       Tnep        Tprs        Tpov
10000   0.001162    0.000573    0.000001
20000   0.002232    0.001134    0.000001
30000   0.003415    0.001518    0.000001
40000   0.004986    0.002394    0.000001
50000   0.005717    0.002765    0.000001
60000   0.007043    0.003435    0.000001
70000   0.008317    0.004305    0.000001
```

Uočava se da, bez obzira na povećanje broja `n`, vreme izvršavanja algoritma u najpovoljnijem slučaj ostaje *konstantno* jer se traženi broj pronalazi na prvoj poziciji u listi. Zato se, bez obzira na veličinu ulazne liste, obavljaju samo tri poređenja (dva u r8 i jedno u r11). Na slici 7.3 prikazani su grafici zavisnosti trajanja izvršavanja od veličine `n` u najnepovoljnijem i prosečnom slučaju.

Grafik sa slike 7.3 sugeriše da, sa porastom broja elemenata u listi, trajanje izvršavanja algoritma raste *linearno* u vremenu. Za očekivati je da brži procesor utiče na kraće izvršavanje i obrnuto. Međutim, analiza vremenske složenosti treba da bude nezavisna od karakteristika sistema. Zato se vreme izvršavanja posmatra kroz broj operacija koje dominantno utiču na brzinu posmatranog algoritma. U funkciji `lin_pretraga()`, dominantna operacija odnosi se na poređenje celobrojnih vrednosti (r8). Ako se traženi



Slika 7.3: Empirijska funkcija vremenske složenost za linearno pretraživanje: x osa - broj elemenata u listi (n), y osa - vreme izvršavanja u sekundama. Uočava se da, sa porastom n, funkcija vremenske složenosti $t(n)$ raste po *linearnom* zakonu.

element nalazi na i-toj poziciji, onda se izvrši $2i+3$ poređenja – dva u r8, za sve elemente na pozicijama $0,1,\ldots i$, te jedno u r11. Ako se traženi broj ne nalazi u listi, obavlja se $2n+2$ poređenja – dva za svaki element u r8, jedno za izlazak iz petlje u r8,[6] te jedno u r11. Funkcija složenosti za najnepovoljniji slučaj može se izraziti kao:

$$t_{najnep}(n) = c_{poredjenje}(2n+2) + c_{ostalo}, \quad c_{poredjenje}, c_{ostalo} > 0 \tag{7.5}$$

Pozitivne konstante $c_{poredjenje}$ i c_{ostalo} iz (7.5) odnose se na trajanje jednog poređenja, odnosno na trajanje svih ostalih operacija koje se ne odnose na dominantna poređenja (na primer, višestruka dodela vrednosti u r7). Konstante imaju *različite* vrednosti u različitim računarskim sistemima.

Da bi se izračunala funkcija složenosti za prosečan slučaj, može se pretpostaviti, bez značajnog umanjenja opštosti, da se traženi broj *nalazi* u listi i da su sve pozicije za njegovu lokaciju *jednako verovatne*. Kako je verovatnoća pojavljivanja na poziciji i jednaka $1/n$, a broj poređenja u tom slučaju $2i+3$, funkcija složenosti može se izraziti sledećom sumom:

$$t_{pros}(n) = c_{poredjenje}\left(\frac{1}{n}\sum_{i=0}^{n-1}(2i+3)\right) + c_{ostalo}, \quad c_{poredjenje}, c_{ostalo} > 0 \tag{7.6}$$

Kako je $\frac{1}{n}\sum_{i=0}^{n-1}(2i+3) = \frac{1}{n}\sum_{i=1}^{n}(2i+1) = \frac{2}{n}\sum_{i=1}^{n} i + 1 = n+2$, izraz (7.6) postaje:

$$t_{pros}(n) = c_{poredjenje}(n+2) + c_{ostalo}, \quad c_{poredjenje}, c_{ostalo} > 0 \tag{7.7}$$

[6] Ako prvi operand za `and` ima vrednost `False`, ne računa se vrednost drugog operanda! Slično važi i za `or`, kada je prvi operand istinit. Videti tabelu 2.3 u glavi 2.3.3.

Iz izraza (7.5) i (7.7) uočava se da, za *veliko* n, odnos $\frac{t_{najnep}(n)}{t_{pros}(n)}$ teži 2 pa je najnepovoljniji slučaj dvostruko sporiji od prosečnog:

$$\lim_{n\to\infty}\frac{t_{najnep}(n)}{t_{pros}(n)} = \frac{2c_{poredjenje}n + 2c_{poredjenje} + c_{ostalo}}{c_{poredjenje}n + 2c_{poredjenje} + c_{ostalo}} = 2$$

Obe funkcije složenosti (7.5 i 7.7) mogu se napisati u obliku $an+b$, gde su a i b pozitivne realne konstante. Koristeći (7.2), lako je pokazati da su obe funkcije u $\mathrm{O}(n)$ jer za bilo koje $c > a$ važi:

$$\forall n > \frac{b}{c-a}, \quad an+b \leq cn$$

Algoritam linearnog pretraživanja ima *linearnu složenost* kako za najnepovoljniji, tako i za prosečan slučaj. U najpovoljnijem slučaju važi $t_{najp}(n) = const$ pa je t_{najp} u O(1) (važi $\forall n > 0, \;\; const < (const+1)\cdot 1$).

7.3.2 Binarno pretraživanje

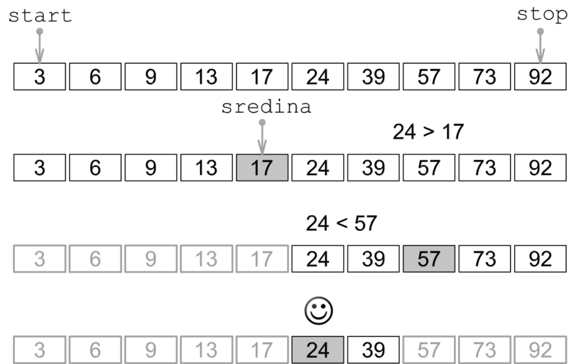
Algoritam binarnog pretraživanja može se upotrebiti *samo* onda kada su objekti u kolekciji *sortirani* po svojoj vrednosti. Prilikom linearnog pretraživanja testiran je jedan po jedan element iz liste počevši od prvog. Na taj način, ukoliko je test bio negativan, iz daljeg razmatranja otpadao je samo po jedan element. Međutim, ako je lista sortirana u rastući poredak, može se prvo testirati da li je traženi broj manji od *središnjeg* elementa u listi. Ako je uslov ispunjen, nema potrebe testirati brojeve desno od središnjeg elementa pošto su oni sigurno veći. U suprotnom, ako je traženi broj veći od središnjeg elementa, prva polovina liste može se odbaciti, a testiranje treba nastaviti počevši od elementa koji dolazi posle središnjeg. Ako je traženi broj jednak središnjem elementu, onda se postupak uspešno završava. Uočiti da se već pri prvom testiranju *polovina* elemenata odbacuje iz razmatranja, a isti postupak ponavlja se sa podlistom levo (ili desno) od središnjeg elementa (slika 7.4). U *najnepovoljnijem* slučaju postupak odbacivanja polovljenjem se ponavlja sve dok se početna lista ne svede na jedan element. Tada je broj lociran ili ga nema u listi.

Binarno pretraživanje spada u algoritme tipa "podeli pa vladaj".[7] Osnovna strategija metode "podeli pa vladaj" ogleda se u *pametnoj redukciji* prostora svih mogućnosti na manje, jednostavnije potprostore. Sledi prikaz funkcije koja realizuje postupak i programa koji meri njeno trajanje u slučaju kada traženi broj nije u listi.

Program 7.3 — Binarno pretraživanje.

```
1 # Binarno pretraživanje
2 def bin_pretraga(L, x):
```

[7] Engl. *Divide and Conquer*.



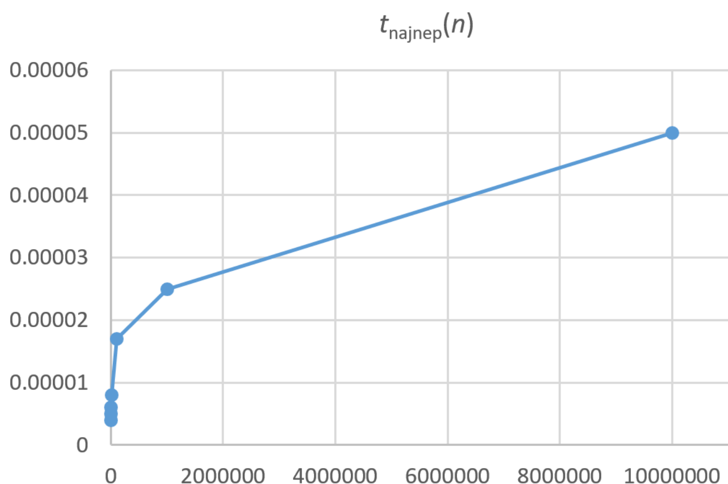
Slika 7.4: Binarno pretraživanje (broja 24): u svakoj iteraciji odbacuje se polovina tekuće podliste. Promenljive `start` i `stop` definišu aktuelnu podlistu, a `sredina` ukazuje na element koji se testira. Odbačeni elementi označeni su sivom bojom.

```
3       '''Pronalazi indeks od x, u L. Vraća None ako x nije u L'''
4       start, stop = 0, len(L)-1
5       while start <= stop:
6           sredina = (start + stop) // 2
7           if x < L[sredina]:
8               stop = sredina - 1
9           elif x > L[sredina]:
10              start = sredina +1
11          else:
12              return sredina
13      return None
14
15  # testiranje za najnepovoljniji slučaj
16  from time import perf_counter
17  print(f"{'n':8}\t{'Tnajnep'}")
18  for n in [10, 100, 1000, 10_000, 100_000, 1_000_000, 10_000_000]:
19      L = list(range(n))
20      t_start = perf_counter()
21      bin_pretraga(L, n)
22      t_najnep = perf_counter() - t_start
23      print(f'{n:<8d}\t{t_najnep:<8.6f}')
```

```
n         Tnajnep
10        0.000004
100       0.000005
```

```
1000     0.000006
10000    0.000008
100000   0.000017
1000000  0.000025
10000000 0.000050
```

Zavisnost trajanja pretrage od velične liste, u najnepovoljnijem slučaju, očigledno nije linearna (slika 7.5). U analizi složenosti najnepovoljnijeg slučaja može se, bez gubitka opštosti, pretpostaviti da je veličina liste $n = 2^k$. Kako se u petlji `while` (r5-12), pored provere uslova `start<=stop`, obavljaju još najviše dva testa (r7 i r9), najnepovoljniji slučaj nastupa kada je broj koji se traži veći od poslednjeg (maksimalnog) elementa u listi.



Slika 7.5: Funkcija složenosti za binarno pretraživanje: postupak ima složenost reda $O(\log_2 n)$.

Budući da se u svakoj iteraciji odbaci polovina preostalih elemenata, broj potrebnih iteracija, koji svodi listu od 2^k elemenata na podlistu od jednog elementa, jednak je k. U poslednjoj, $k+1$-voj iteraciji, promenljive `start` i `stop` iz r5 ukazuju na istu vrednost $(n-1)$ pa `sredina` ukazuje na $n-1$ (r6). Potom `start` postaje veće od `stop` (r10) pa se, nakon provere uslova u r5, petlja prekida i postupak završava vraćanjem objekta `None`. Funkcija složenosti za ovaj slučaj je $t_{najnep}(n) = 3(k+1)+1$, odnosno $t_{najnep}(n) = 3k+4$. Kako je $k = \log_2 n$, funkcija složenosti dobija svoj konačni oblik:

$$t_{najnep}(n) = 3\log_2 n + 4 \qquad (7.8)$$

Kada se (7.8) pojednostavi, primenom pravila iz 7.2.2 (zadržavanje dominantnog člana, odbacivanje konstanti), sledi da algoritam binarne pretrage ima složenost reda $O(\log_2 n)$.

Binarno pretraživanje koristiti se kada je sekvenca objekata sortirana po odgovarajućim vrednostima. Šta raditi kada to nije slučaj? Kao što će se videti u narednom

izlaganju, sortiranje sekvence sa velikim brojem objekata oduzima prilično vremena. Zato treba razmotriti da li se sekvenca često pretražuje ili ažurira. Ako se sekvenca *retko pretražuje* i *često ažurira*, savetuje se upotreba linearnog algoritma. U slučaju kada se sekvenca *retko ažurira* i *često pretražuje*, preporučuje se binarni algoritam uz obavezno prethodno sortiranje.

7.4 Algoritmi sortiranja

Algoritmi sortiranja zauzimaju istaknuto mesto u računarskim naukama. Mogu se klasifikovati na različite načine, ali će ovde biti spomenuta podela na algoritme *unutrašnjeg* i *spoljašnjeg* sortiranja. Unutrašnje sortiranje podrazumeva da se polazna sekvenca u potpunosti nalazi u operativnoj memoriji, dok se kod spoljašnjeg sortiranja, zbog izuzetno velike količine podataka, delovi sekvence nalaze na spoljnoj memoriji (najčešće disk). Imajući u vidu da procesor isključivo radi sa podacima u operativnoj memoriji (Fon Nojmanov model računara, glava 1.2), algoritmi spoljašnjeg sortiranja zahtevaju kompleksnije postupke od unutrašnjih parnjaka. Pošto su savremeni memorijski moduli dovoljno velikog kapaciteta za najveći broj inženjerskih primena, u nastavku će se razmatrati karakteristični algoritmi unutrašnjeg sortiranja. Neki od njih biće rekurzivne prirode, a neke će karakterisati upotreba dodatne memorije. Opet, neki će tokom postupka koristiti operatore poređenja, a neki ne.

U Pajtonu se, za sortiranje, koristi ugrađena funkcija `sorted()` koja vraća *novu* uređenu sekvencu, odnosno metoda `sort()` koja preuređuje *postojeću* listu. Ipak, poznavanje osnovnih tehnika sortiranja izoštrava umeće algoritamskog rešavanja problema, a predstavlja i dobru vežbu za sprovođenje analize algoritamske efikasnosti.

7.4.1 Counting sort

Algoritam *Counting sort*[8] može se primeniti samo ako su svi elementi u sekvenci *pozitivni celi brojevi* ili se mogu *jednoznačno prevesti* u iste. Ideja algoritma izložena je prilikom rešavanja sledećeg problema:

Problem 7.2 — Plate u Patkovgradu. Građani Patkovgrada anketirani su na temu visina mesečnih plata. Plate su zapisivane kao pozitivni celobrojni iznosi, ne veći od 1000 (u talirima, lokalna valuta). Sastaviti program koji na osnovu prikupljenog slučajnog uzorka izračunava prosečnu patkovgradsku platu. ■

Razmatrajući date odgovore, uočava se da je najveći broj ispitanika prijavio visinu plate od oko 30 talira. Kako u gradovima poput Patkovgrada vlada nejednaka raspodela bogatstva, *mali* broj ljudi ima *značajno veće* plate od većine (među ispitanicima je bio i Baja Patak). Aritmetička sredina ovako *nesimetrično* raspoređenog uzorka prikazala bi novčano stanje građana optimističnije nego što je realno. Na primer, neka je data sledeća

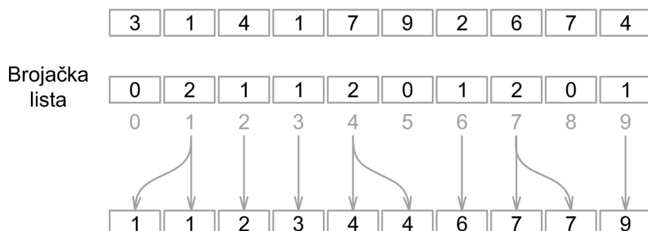
[8] Imena algoritama bez odomaćenog prevoda sa engleskog jezika navodiće se u originalu.

lista plata [20, 15, 10, 20, 25, 30, 50, 30, 50, 1000, 950]. Aritmetička sredina uzorka je 120. Ipak, ona daje neadekvatno visoku procenu za plate jer poslednja dva elementa predstavljaju *izuzetke* u posmatranom primeru.

Za opisivanje pomenutog uzorka pogodnije je koristiti *medijanu*. Medijana uzorka (liste) računa se na sledeći način:

1. sortirati listu plata u neopadajući poredak.
2. ako lista ima neparan broj elemenata, za medijanu se uzima središnji element, inače medijana je aritmetička sredina dva središnja elementa.

U posmatranom primeru medijana iznosi 30. Ključni deo pri računanju medijane predstavlja sortiranje liste plata u neopadajući poredak. Kako su plate pozitivni celobrojni iznosi ne veći od 1000, može se primeniti *Counting sort*.



Slika 7.6: Algoritam *Counting sort*: elementi koji se sortiraju predstavljaju indekse za brojačku listu. Ova lista memoriše broj ponavljanja određenog elementa u polaznoj listi. Na primer, broj 1 se javlja dva puta u polaznoj listi.

Osnovna ideja algoritma je vrlo jednostavna (slika 7.6): prvo se formira brojačka lista od 1001 elemenata inicijalizovana nulama (B). Potom se elementi ulazne liste obrađuju jedan po jedan, počevši od prve pozicije. Ovom prilikom, visina plate koristi se kao *indeks* za ažuriranje odgovarajućeg brojača iz B čija vrednost se povećava za jedan. Konačno, prilikom rednog prolaska kroz B, indeksi nenultih elemenata upisuju se u sortiranu sekvencu onoliko puta na koliko ukazuju odgovarajući brojači.

Program 7.4 — Plate u Patkovgradu.

```
1 # Counting sort
2 def counting_sort(L, maks):
3     '''Sortira L po metodi Counting Sort.
4        maks - najveća očekivana vrednost u L'''
5     B = [0 for i in range(maks+1)]
6     for e in L:
7         B[e] += 1
8     k = 0
```

```
9       for i in range(maks+1):
10          for j in range(B[i]):
11              L[k] = i
12              k += 1
13
14  # Medijana
15  def medijana(L):
16      ''' Vraća medijanu celobrojne liste L'''
17      s = len(L) // 2
18      if s % 2 == 0:
19          return (L[s-1] + L[s])/2
20      else:
21          return L[s]
22
23  # Test: plate u Patkovgradu
24  plate = [20, 15, 10, 20, 25, 30, 50, 30, 50, 1000, 950]
25  maks_plata = max(plate)
26  print('pre sortiranja', plate)
27  counting_sort(plate, maks_plata)
28  print('posle sortiranja', plate)
29  print('prosečna plata u Patkovgradu', medijana(plate))
```

Funkcija `counting_sort()` ima dva formalna parametra: listu koja se sortira (`L`) i maksimalnu vrednost koja se u njoj očekuje (`maks`). Brojačka lista `B` inicijalizuje se na osnovu maksimalno očekivane vrednosti (r5). Odgovarajući brojači ažuriraju se u petlji prilikom prolaska kroz ulaznu listu (r6-7). Na osnovu informacije sadržane u `B`, ulazna lista se sortira tako što se sortirani elementi prepisuju preko početnih vrednosti (r9-12). Promenljiva `k` (r8) ukazuje na tekuću poziciju u ulaznoj listi koja se prepisuje sortiranim elementom. U petlji po `j`, element sa vrednošću `i` upisuje se `B[i]` puta (r10-12). Ako je `B[i]` nula, opseg definisan sa `range(B[i])` je prazan pa se petlja ne izvršava. Treba primetiti da se `L`, unutar funkcije, tretira kao globalna promenljiva. Referenca definisana sa `L` se ne menja unutar funkcije, ali se menja objekat tipa liste na koji ona ukazuje.

Ugrađena funkcija `max()`, upotrebljena u r25, vraća maksimalni element iz ulazne sekvence. Ako se, umesto sekvence, navede niz skalarnih veličina odvojenih zapetom, onda funkcija vraća najveću vrednost iz niza. Na primer, `max(1,2,31.5)` vraća `31.5`. Slično, ugrađena funkcija `min()` vraća najmanju vrednost iz sekvence. Sledi prikaz rada programa:

```
pre sortiranja [20, 15, 10, 20, 25, 30, 50, 30, 50, 1000, 950]
posle sortiranja [10, 15, 20, 20, 25, 30, 30, 50, 50, 950, 1000]
prosečna plata u Patkovgradu 30
```

Sada se razmatra složenost za *Counting sort*, pri čemu se pretpostavlja da je ulazna lista dužine n. Dominantne operacije odnose se na *uvećavanje za jedan* i *dodelu vrednosti* u r7 i r12, odnosno dodelu vrednosti u r11. Prilikom dodele vrednosti iz r7, treba imati na umu da se tom prilikom pristupa e-tom elementu liste, te da ova operacija traje nešto duže nego u slučaju dodele iz r12. Imajući u vidu da se r7, r11 i r12 izvršavaju n puta, bez obzira na ulazne podatke, funkcija složenosti glasi:

$$t(n) = c_{r7}n + c_{r12}n + c_{r11}n + c_{ostalo} = cn + c_{ostalo}$$

Očigledno, funkcija $t(n)$ je reda $\Theta(n)$ - brzina rasta ograničena je i odozgo i odozdo linearnom funkcijom. Ne postoji algoritam sortiranja koji može postići bolji rezultat jer da bi se element postavio u pravu poziciju, mora se bar jednom posetiti! Međutim, pristup ima ozbiljan nedostatak: za sortiranje je *neophodno* postojanje brojačke liste koja može zauzeti puno memorije, čak i za najkraće ulazne liste. Memorijski zahtevi zavise isključivo od maksimalne vrednosti elemenata koji se sortiraju. Ako je maksimalna vrednost unapred nepoznata, ili ako je poznata ali nedopustivo velika, onda se moraju koristiti druge tehnike sortiranja. *Counting sort* ne može se primeniti ni za sortiranje objekata čije se vrednosti ne mogu upotrebiti za indeksiranje brojačke liste.

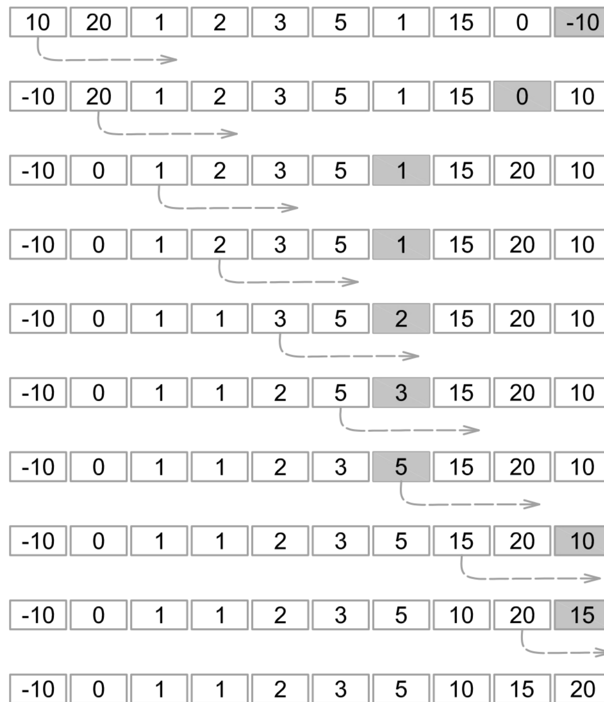
7.4.2 Selection sort

Algoritam *Selection sort* spada u jednostavne i najčešće korišćene postupke sortiranja. Koristi se za ulazne sekvence sa veoma malim brojem elemenata (do par stotina). Razmatra se situacija sortiranja celobrojne liste, pri čemu se napominje da postupak ostaje isti i u slučaju realnih brojeva. U *prvom* prolazu algoritam pronalazi *najmanji* element iz liste i postavlja ga na prvu poziciju. Potom se, počevši od druge pozicije, postupak ponavlja: traži se *sledeći najmanji* element u podlisti koja počinje od druge pozicije.

Neka su, u opštem slučaju, svi brojevi levo od tekuće pozicije sortirani. Sada se na tekuću poziciju i postavlja najmanji element iz podliste koja, počevši od i, obuhvata sve elemente nadesno – i-ti minimum po redu dolazi na svoje mesto (slika 7.7).

Program 7.5 — Selection sort.

```
1  # Selection sort
2  def selection_sort(L):
3      '''Sortira L po metodi Selection sort'''
4      n = len(L)
```



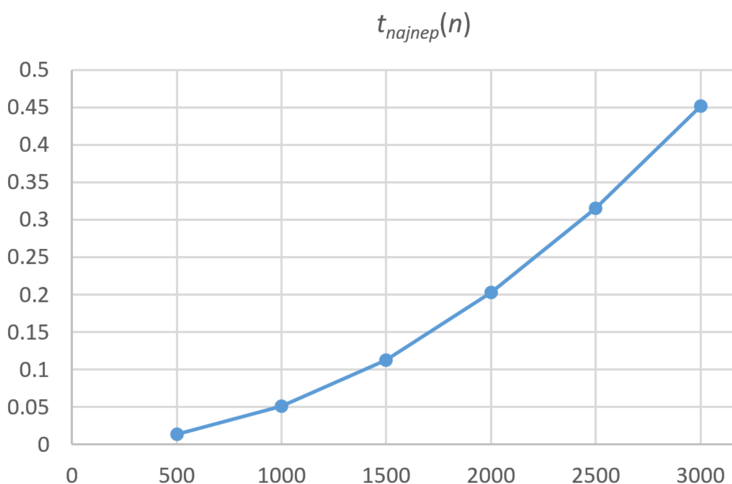
Slika 7.7: Algoritam *Selection sort*: isprekidana linija označava tekuću poziciju, a osenčena ćellija minimalni element u desnoj podlisti. Po prolasku kroz podlistu, najmanji element menja mesto sa elementom na tekućoj poziciji. Posle prve iteracije brojevi -10 i 10 menjaju mesto.

```
    for i in range(n-1):
        tekući_min = i
        for j in range(i+1, n):
            if L[j] < L[tekući_min]:
                tekući_min = j
        L[i], L[tekući_min] = L[tekući_min], L[i]

# test
from time import perf_counter
L = [10, 20, 1, 2, 3, 5, 1, 15, 0, -10]
print('pre ', L)
selection_sort(L)
print('posle ', L)

print(f"{'n':6}\t{'Tnajnep':8}")
for n in [500, 1000, 1500, 2000, 2500, 3000]:
    L = list(range(n, 0, -1))
```

```
22      start = perf_counter()
23      selection_sort(L)
24      t_najnep = perf_counter() - start
25      print(f'{n:<6d}\t{t_najnep:<8.6f}')
```

Glavna petlja po `i` fiksira tekući element, odnosno poziciju na koju treba dovesti `i`-ti minimum (r5-10). Indeks tekućeg minimuma (najmanjeg elementa desno od tekućeg elementa) pamti se uz pomoć promenljive `tekući_minimum` (r6). Indeks elementa koji predstavlja `i`-ti minimum pronalazi se u petlji po `j` (r7-9). U r10, uz pomoć višestruke naredbe dodele, tekući minimum menja mesto sa elementom na tekućoj poziciji. Empirijska analiza složenosti sprovedena je za najnepovoljniji slučaj kada je lista sortirana u opadajućem redosledu, tako što je izmereno trajanja postupka sortiranja (r19-25). Sledi prikaz rada programa i eksperimentalna kriva složenosti (slika 7.8):

```
pre  [10, 20, 1, 2, 3, 5, 1, 15, 0, -10]
posle  [-10, 0, 1, 1, 2, 3, 5, 10, 15, 20]
n         Tnajnep
500       0.013540
1000      0.051029
1500      0.113254
2000      0.203020
2500      0.315468
3000      0.451412
```



Slika 7.8: Složenost algoritma *Selection sort*: u najnepovoljnijem slučaju, funkcija složenosti ponaša se kao kvadratna funkcija.

Da bi se algoritam mogao iskoristiti za sortiranje u opadajući poredak, treba samo *promeniti* operator poređenja iz r8: umesto <, treba staviti >. Tada bi promenljivu `tekući_min` trebalo preimenovati u `tekući_max`, što bi odgovaralo značenju objekta na koji promenljiva ukazuje. Funkcija složenosti, u najnepovoljnijem slučaju, može se definisati na bazi sledećih opažanja:

- obavlja se $n-1$ dodela vrednosti (r6) i isti broj razmena vrednosti (r10),
- obavlja se $(n-1)+(n-2)+...+2+1=\frac{(n-1)n}{2}$ dodela vrednosti (r9).

Ako se pretpostavi da su c_{dodela} i $c_{razmena}$ vremena potrebna za izvršavanje jedne dodele i jedne razmene vrednosti, a c_{ostalo} vreme trajanja ostalih operacija u programu, može se formirati funkcija složenosti za najnepovoljniji slučaj:

$$t_{najnep}(n)=\frac{(n-1)n}{2}c_{dodela}+(n-1)c_{dodela}+(n-1)c_{razmena}+c_{ostalo}=c_1n^2+c_2n+c_3$$

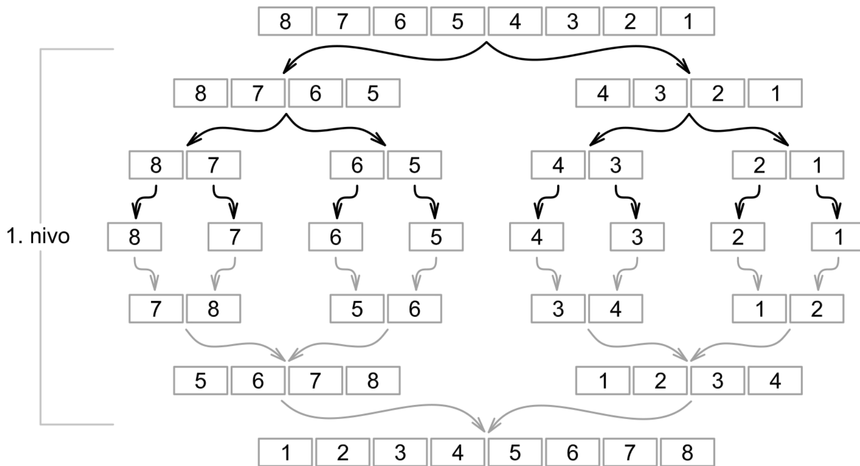
Očigledno je da *Selction sort* ima vremensku složenost reda O(n^2), što se slaže sa rezultatom dobijenim u eksperimentalnoj analizi (slika 7.8). Algoritam *nije* memorijski zahtevan jer se sortiranje, za razliku od postupka *Counting sort*, obavlja *u okviru* zadate liste. Za sekvence sa većim brojem elemenata primenjuju se efikasniji algoritmi, poput *Merge sort* ili *Quick sort* postupaka, koji su opisani u sledećem delu.

7.4.3 Merge sort

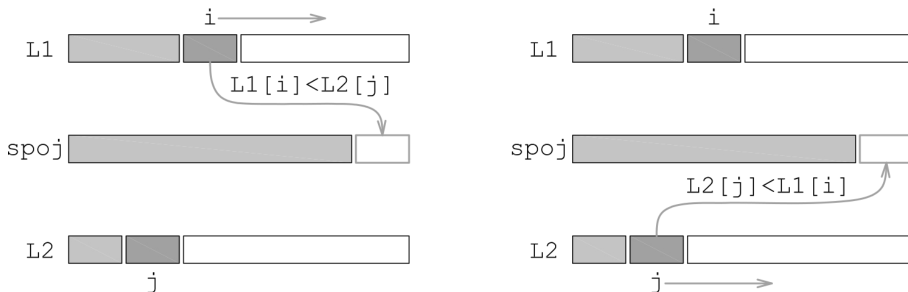
Metoda sortiranja *Merge sort* pripada familiji algoritama *podeli pa vladaj* koja početni problem razlaže na svoje *jednostavnije* verzije koje se obično rešavaju u *rekurzivnom* maniru. Ulazna lista se deli na *dva* jednaka dela koja se rekurzivno sortiraju pa se potom sortirane polovine *spajaju*.[9] U svakom rekurzivnom pozivu prosleđena podlista deli se na polovine koje se sortiraju i potom spajaju. Bazni slučaj rekurzije nastupa kada lista koju treba sortirati sadrži samo jedan element – takva lista je već sortirana. Rekurzivni postupak ilustrovan je pogodnim primerom na slici 7.9.

Postupak spajanja dve *već sortirane* polovine `L1` i `L2` liste `L` ima sledeći oblik: neka su podliste `L1` i `L2` sortirane u rastući poredak. Lista `L` sortira se tako što se `L1` i `L2` spoje u rezultujuću listu `spoj` na način prikazan na slici 7.10. Neka se svi elementi levo od `L1[i]` i `L2[j]` već nalaze u listi `spoj` tako da formiraju rastući poredak. Sada se razmatra uslov `L1[i]<L2[j]` (slika 7.10 levo). Ako je uslov ispunjen, `L1[i]`, kao najmanji od svih preostalih elemenata u `L1` i `L2`, treba dodati na kraj liste `spoj` i uvećati indeks `i` za jedan. Slično rezonovanje primenjuje se i ako uslov nije ispunjen kada se u `spoj` dodaje element `L2[j]`, a indeks `j` uvećava za jedan (slika 7.10 desno). Postupak se ponavlja sve dok se jedna od listi potpuno ne iscrpi. Tada se preostali deo druge liste dodaje na kraj liste `spoj`.

[9] Engl. *to merge* - spojiti.



Slika 7.9: Algoritam *Merge sort*: stablo rekurzivnih poziva. Lista se prvo deli na polovine, pa na polovine polovina i tako redom sve do jednoelementnih listi. Potom se, obrnutim redosledom, vrše spajanja sortiranih delova. Podele su ilustrovane crnim, a spajanja sivim strelicama.



Slika 7.10: Algoritam *Merge sort*: faza spajanja.

Opisani način sortiranja *ne obavlja* se u mestu (u istoj listi u kojoj se nalaze podaci), već je potrebno odvojiti onoliko dodatnog memorijskog prostora koliko zauzima i sama lista L! Izloženi postupak može se realizovati u Pajtonu na sledeći način:

Program 7.6 — Merge sort.

```
# Merge sort
def merge_sort(L, levo, desno):
    '''Sortira podlistu L[levo:desno] po metodi Merge sort'''
    if levo + 1 < desno:
        sredina = (levo + desno) // 2
        L1 = merge_sort(L, levo, sredina) # sortira 1. polov. L
        L2 = merge_sort(L, sredina, desno) # sortira 2. polov. L
        return spajanje(L1, L2)      # spaja sortirane polovine
```

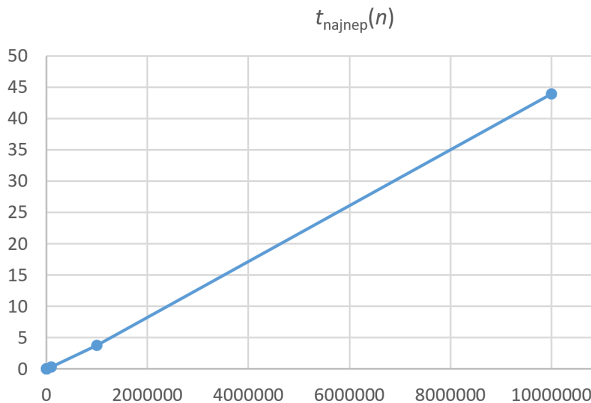
```
9	    else:
10	        return [L[levo]]
11	
12	def spajanje(L1, L2):
13	    ''' spaja sortirane liste L1 i L2 u novoj listi'''
14	    spoj = []
15	    i, j = 0, 0
16	    n1, n2 = len(L1), len(L2)
17	    while i < n1 and j < n2:
18	        if L1[i] < L2[j]:
19	            spoj.append(L1[i])
20	            i += 1
21	        else:
22	            spoj.append(L2[j])
23	            j += 1
24	
25	    spoj.extend(L1[i:]) # za i=len(L1) (prazna podl.) ne dodaje
26	    spoj.extend(L2[j:]) # za j=len(L2) (prazna podl.) ne dodaje
27	
28	    return spoj
29	
30	# test
31	from time import perf_counter
32	L = [8, 7, 6, 5, 4, 3, 2, 1]
33	print('pre ', L)
34	print('posle ', merge_sort(L, 0, len(L)))
35	
36	print(f"{'n':9}\t{'Tnajnep':9}")
37	for n in [10, 100, 1000, 10_000, 100_000, 1_000_000, 10_000_000]:
38	    L = list(range(n, 0, -1))
39	    start = perf_counter()
40	    merge_sort(L, 0, n)
41	    delta = perf_counter() - start
42	    print(f'{n:<9d}\t{delta:<9.6f}')
```

Funkcija `merge_sort()`, pored liste `L`, prima i formalne parametre `levo` i `desno` (r2). Parametri definišu početak i kraj podliste koju treba sortirati. Inicijalni poziv podrazumeva navođenje opsega kojim se definiše početna lista (r34, 40). Po određivanju pozicije središnjeg elementa (`sredina`), pristupa se sortiranju prve (r6), odnosno druge polovine inicijalne liste (r7). Sortirane podliste se spajaju pomoću funkcije `spajanje()`, čime se tekući poziv za `merge_sort` završava (r8). Ako ulazna lista

sadrži samo jedan element, od njega se formira jednoelementna lista (r10), a rekurzivni pozivi na toj grani pozivnog stabla se prekidaju (slika 7.9).

U funkciji `spajanje()`, po završetku petlje `while` (r17-23), jedna od podlisti obrađena je u celosti. Preostali elementi nepotpuno obrađene podliste dodaju se na kraj liste `spoj`. Dodavanje se, *bez testiranja* o kojoj podlisti se radi, obavlja putem dva proširenja liste `spoj` pomoću metode `extend()` (r25-26). Rezultat rada programa za testiranje (r31-42) i eksperimentalna analiza složenosti, dati su u sledećem prikazu i na slici 7.11:

```
pre  [8, 7, 6, 5, 4, 3, 2, 1]
posle  [1, 2, 3, 4, 5, 6, 7, 8]
n         Tnajnep
10        0.000058
100       0.000420
1000      0.004109
10000     0.031143
100000    0.337171
1000000   3.819061
10000000  43.912488
```



Slika 7.11: Složenost algoritma *Merge sort*: funkcija složenosti ima nešto brži rast od linearnog. Primenom precizne analize utvrđen je red složenosti O($n \cdot log_2 n$).

Ključna obrada u algoritmu obavlja se u funkciji `spajanje()`. Dominantni *par* operacija čine *pristup* odgovarajućem elementu neke od listi, te *dodavanje* tog elementa na kraj rezultujuće liste (r19, 22, 25, 26). Za liste `L1` i `L2`, čije dužine iznose po $m/2$, u procesu spajanja treba izvršiti m dominantnih parova - funkcija vremenske složenosti za `spoj()` može se izraziti kao $c_{dodavanje} m + c_{ostalo}$. Veličina $c_{dodavanje}$ označava vreme izvršavanja dominantnog para operacija, m - dužinu rezultujuće liste, a c_{ostalo} - vreme izvršavanja svih ostalih operacija u funkciji.

Da bi se definisala funkcija složenosti kompletnog postupka, može se, kao i prilikom

binarnog pretraživanja, bez gubitka opštosti pretpostaviti da je broj elemenata u listi $n = 2^k$. Uočiti bilo koju putanju u stablu sa slike 7.9, od početne do jednoelementne liste (na primer, krajnje levu koja vodi do [8]). Da bi se stiglo do jednoelementne liste, na svakoj od putanja treba obaviti k sukcesivnih podela (k nivoa u stablu). U prvom nivou, ispod polazne liste, obavlja se samo jedno spajanje sortiranih podlista od po $n/2$ elemenata. Zato ova faza u sortiranju ima složenost reda $O(n)$.

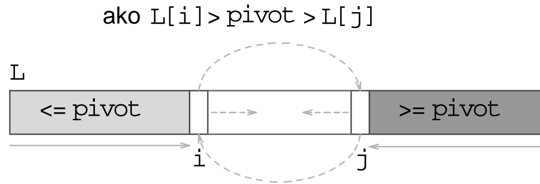
U drugom nivou, obavljaju se dva spajanja u rezultujuće liste koje imaju po $n/2$ elemenata, što odgovara funkciji vremenske složenosti $2(c_{dodavanje}n/2 + c_{ostalo}) = c_{dodavanje}n + 2c_{ostalo}$. Očigledno, ova faza ima složenost reda $O(n)$. Sličnom analizom zaključuje se da je, u svakoj fazi sortiranja, funkcija vremenske složenosti reda $O(n)$. Pošto se u kompletnom postupku obavi $k = \log_2 n$ ovakvih faza, može se zaključiti da red složenosti algoritma *Merge sort* iznosi $O(n \cdot \log_2 n)$. Za algoritme navedenog reda složenosti kaže se da se izvršavaju u *kvazilinearnom* vremenu. Kvazilinearni algoritmi sporiji su u odnosu na linearne, ali i dosta brži u odnosu na kvadratne pa se problemi, koji se na ovaj način mogu rešiti, smatraju *efikasno* rešivim. Treba istaći da se u algoritmima za sortiranje koji se zasnivaju na *poređenjima*, a koji se mogu primeniti na *bilo koju* listu objekata (što *Counting sort* svakako nije), *ne može* postići složenost manja od $O(n \cdot \log_2 n)$!

Očigledna prednost metode *Merge sort*, u odnosu na *Selection sort*, ogleda se u brzini izvršavanja prilikom velikog broja podataka. Međutim, treba uočiti i nedostatak koji se odnosi na dodatnu memoriju koja je, za razliku od metode *Selection sort*, u ovom slučaju neophodna.

Naučna i inženjerska praksa pokazuju da se rešenje iole ozbiljnijeg problema *ne može* smatrati *univerzalno* dobrim u svim prilikama i za svaku primenu. Izborom rešenja automatski se pristaje na odgovarajući kompromis, ali se tom prilikom treba truditi da se, u željenoj aplikaciji, *najefikasnije* pokriju *najbitniji* slučajevi.

7.4.4 Quick sort

Metoda sortiranja *Quick sort* pripada familiji algoritama "podeli pa vladaj". Neka se početna lista L sortira u rastući poredak. Lista L se prvo podeli na dve podliste tako da su *svi* elementi prve *manji ili jednaki* u odnosu na elemente druge podliste. Potom se svaka podlista, ako ima *bar dva* elementa, rekurzivno sortira tako što se podeli na isti način. U baznom slučaju, kada podlista ima jedan ili nula elemenata, sortiranje nije potrebno. Pomenuta podela liste L postiže se izborom *proizvoljnog* elementa koji se naziva *pivot* (obično središnjeg). U postupku podele uvode se dva indeksa koji polaze sa različitih krajeva liste (levi s početka, a desni s kraja). Potom se za svaki element koji je veći od pivota, a na koga ukazuje levi indeks, odnosno za svaki element koji je manji od pivota, a na koga ukazuje desni indeks, vrši zamena mesta (slika 7.12). Na ovaj način održava se svojstvo da su svi elementi leve *ne veći* od elemenata u desnoj podlisti. Postupak podele završava se kada levi indeks postane veći ili jednak desnom.



Slika 7.12: Algoritam *Quick sort*: postupak podele na dve podliste. Svi elementi leve podliste (svetlo sivo), manji su ili jednaki u odnosu na sve elemente desne podliste (tamno sivo). Pivot, u odnosu na koga se formira podela, može biti proizvoljan element iz liste.

Algoritam *Quick sort* može se realizovati uz pomoć glavne funkcije `quick_sort()` i pomoćne funkcije `podeli()` na sledeći način:

Program 7.7 — Quick sort.

```
1  # Quick sort
2  def quick_sort(L, levo, desno):
3      '''Sortira podlistu L[levo:desno+1] po metodi Quick sort'''
4      if levo < desno:
5          p = podeli(L, levo, desno)
6          quick_sort(L, levo, p)
7          quick_sort(L, p+1, desno)
8
9  def podeli(L, levo, desno):
10     '''Deli L tako da su prvo sve manji ili jednaki,
11        pa sve veći ili jednaki pivotu. Pivot - srednji elm.
12        Vraća poziciju poslednjeg elementa prvog podniza'''
13     i, j, pivot = levo, desno, L[(levo + desno) // 2]
14     while True:
15         while L[i] < pivot:
16             i += 1
17         while L[j] > pivot:
18             j -= 1
19
20         if i < j:
21             L[i], L[j] = L[j], L[i]
22             i, j = i + 1, j - 1
23
24         else:
25             return j
26
```

```
27  # test prosečnog slučaja
28  from time import perf_counter
29  from random import randint
30  from mergesort import merge_sort
31
32  print(f"{'n':8}\t{'Merge s.':9}\t{'Quick s.':9}")
33  for n in [100, 1000, 10_000, 100_000]:
34
35      ms, qs = 0, 0
36      for j in range(100):
37
38          L1 = [randint(1, n) for k in range(n)]
39          L2 = L1[:]   # kopiranje L1 u L2
40
41          start = perf_counter()
42          merge_sort(L1, 0, n)
43          ms = perf_counter() - start + ms
44
45          start = perf_counter()
46          quick_sort(L2, 0, n-1)
47          qs = perf_counter() - start + qs
48
49      print(f'{n:<8d}\t{ms/100:<9.6f}\t{qs/100:<9.6f}')
```

Funkciji `quick_sort()` (r2-7), prosleđuje se lista za sortiranje (`L`), kao i granice u okviru kojih se vrši sortiranje (`levo` i `desno`). Desna granica odnosi se na *poslednji* element u listi pa se sortiranje vrši sa `quick_sort(L,0,n-1)`, gde `n` označava dužinu liste. Ako lista ima bar dva elementa (r4), onda se u r5 poziva funkcija `podeli()` koja preuređuje listu na način sa slike 7.12 (r9-25). Tom prilikom vraća se vrednost pozicije koja predstavlja *kraj* leve podliste (`p`). Potom se vrši rekurzivno sortiranje leve (r6), odnosno desne podliste (r7). U prikazanoj implementaciji za pivot se uzima središnji element liste (r13). Međutim, uočiti da postupak *ne zavisi* od načina izbora pivota!

U programu za testiranje poredi se prosečno vreme izvršavanja algoritama *Quick sort* i *Mege sort* (r28-49). Algoritam *Mege sort*, realizovan prema postupku iz 7.4.3, smešten je u modul `mergesort` (r30). Petljom po `n` fiksira se ukupan broj elemenata liste (r33), pa se potom generiše 100 listi sa po `n` slučajnih celih brojeva iz intervala `[1,n]` (r36, 38, 39). Uočiti da se generisana lista `L1`, koja će se sortirati uz pomoć algoritma *Mege sort* (r41-43), kopira u r39 kako bi se pod istim uslovima testiralo vreme izvršavanja za algoritam *Quick sort* (r45-47). Da je u r39, umesto `L2 = L1[:]`, stavljeno `L2 = L1`, kopirala bi se vrednost objektne reference pa bi se *Quick sort*

primenio na već sortiranu listu (glava 5.1.3). Usrednjeno vreme izvršavanja za oba postupka prikazuje se u r49. Dobijeni su sledeći rezultati:

```
n         Merge s.  Quick s.
100       0.000289  0.000171
1000      0.003751  0.002384
10000     0.047270  0.029838
100000    0.570616  0.369056
```

Algoritam *Quick sort* spada u najefikasnije postupke za sortiranje u najvećem broju praktičnih situacija. Njegov red složenosti u *prosečnom* slučaju iznosi $O(n \cdot \log_2 n)$. Ipak, zbog manjih vrednosti konstanti u funkciji složenosti za prosečni slučaj, on je dosta brži od *Merge sort*-a. U *najnepovoljnijem* slučaju njegov red složenosti iznosi $O(n^2)$, što je *lošije* od *Merge sort*-a. Međutim, važna prednost *Quick sort*-a u odnosu na *Merge sort* je da *ne koristi* dodatnu memoriju (sortira u mestu).

Interesantno je napomenuti da najgori slučaj zavisi od *vrednosti* izabranog pivota (uvek se bira minimalni ili maksimalni element podliste koja se sortira). U slučaju da se za pivot uzima središnji element, jedan od mogućih najnepovoljnijih slučajeva ne odnosi se na opadajuću listu, već na listu u kojoj su svi elementi isti! Kada je lista opadajuća, i-ti elementi s početka i kraja menjaju mesta već posle prve podele, što rezultuje sortiranom listom. Međutim, ako su svi elementi jednaki onda, pri svakom pozivu funkcije `podeli()`, svi parovi elemenata s početka i kraja zamenjuju mesta, uz vraćanje središnjeg indeksa kao kraja levog podniza. Tada se, u svim rekurzivnim pozivima, obavlja maksimalan broj zamena.

7.5 Redovi složenosti predefinisanih operacija nad kolekcijama

Prilikom analize nekog programa potrebno je poznavati vremensku složenost upotrebljenih operacija i metoda jer se iza njih često *krije* kompleksan postupak izračunavanja. Ovo naročito važi za metode koje se pozivaju nad objektima koji predstavljaju kolekcije. Posmatra se sledeći ilustrativni primer:

```
1  # briše listu element po element na loš način
2  def briši_loše(lista):
3      n = len(lista)
4      for i in range (n):
5          del lista[0]
6
7  # briše listu element po element na dobar način
8  def briši_dobro(lista):
```

```
9      n = len(lista)
10     for i in range (n-1, -1, -1):
11         del lista[i]
12
13 # test program
14 from time import perf_counter
15 a = list(range(100_000))
16 b = a[:] # kopija
17
18 start = perf_counter()
19 briši_loše(a)
20 print(a, 'briši_loše za', perf_counter() - start)
21
22 start = perf_counter()
23 briši_dobro(b)
24 print(b, 'briši_dobro za', perf_counter() - start)
```

Program testira dva pristupa za brisanje *svih* elemenata iz proizvoljne liste objekata. Na prvi pogled, moglo bi se zaključiti da su funkcije `briši_loše()` i `briši_dobro()` podjednako efikasne - operacija `del` poziva se n puta pa bi vremenska složenost trebalo da bude reda $\Theta(n)$ – postupak se odvija *nezavisno* od vrednosti elemenata u listi pa su gornja i donja granica brzine rasta istog reda. Međutim, po izvršavanju programa dobija se sledeći prikaz:

```
[] briši_loše za 1.1180672180931737
[] briši_dobro za 0.007525126883800137
```

Očigledno, prilikom brisanja elemenata s kraja liste, postiže se daleko bolje vreme nego kada se elementi brišu počevši od prve pozicije. Ovakvo ponašanje posledica je prirode algoritma kojim se realizuje operacija `del`. Prilikom brisanja elementa sa k-te pozicije, sve reference na elemente desno od njega *pomeraju* se za jedno mesto ulevo.[10] Zato `del` ima složenost reda $O(n)$. Ako se elementi brišu s kraja liste, *nema* nikakvih pomeranja pa je red složenosti za funkciju `briši_dobro()` jednak $\Theta(n)$. U slučaju brisanja s početka, *uvek* se vrši pomeranje referenci na sve preostale članove liste za jedno mesto ulevo. Zato funkcija `briši_loše()` ima red složenosti $\Theta(n^2)$. U daljem tekstu navode se složenosti pojedinih operacija i metoda za objekte najčešće korišćenih kolekcija tipa liste i rečnika.

[10] U glavi 5 je pomenuto da liste, kao i druge kolekcije, ne sadrže same objekte već reference na njih – videti slike 5.1 i 5.5.

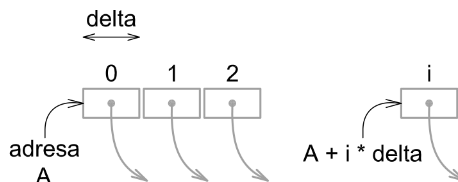
7.5.1 Složenost u radu sa listama

Prikaz složenosti pojedinih operacija/metoda u radu sa listama dat je u tabeli 7.2. Navedeni redovi složenosti odnose se na prosečan i najnepovoljniji slučaj.

operacija	prosečno	najnepovoljnije
`n = len(L)`	O(1)	O(1)
`a = L[i]`	O(1)	O(1)
`L[i] = a`	O(1)	O(1)
`L.append(a)`	O(1)	O(1)
`L.insert(k, a)`	O(n)	O(n)
`del L[i]`	O(n)	O(n)
`L1 = L[i:i+k]`	O(k)	O(k)
`a = min(L), a = max(L)`	O(n)	O(n)
`a in L`	O(n)	O(n)
`L1 = k * L`	O(kn)	O(kn)
`L.sort()`	O($n \cdot \log_2 n$)	O($n \cdot \log_2 n$)

Tabela 7.2: Vremenska složenost u radu sa listama. Prilikom sortiranja, Pajton koristi algoritam *Timsort* [`http://wiki.c2.com/?TimSort`].

Ovde se diskutuje operacija `L1 = L[i:i+k]`, a čitaocu se ostavlja da razmotri ostale slučajeve iz tabele 7.2. Od ranije je poznato da lista ne sadrži objekte direktno, već njihove početne adrese u memoriji - reference. Reference su smeštene počevši od startne adrese liste, označene sa A (slika 7.13).

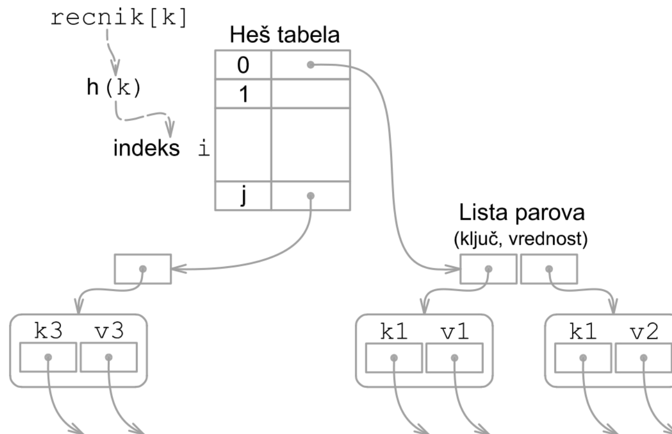


Slika 7.13: Uprošćeni prikaz interne organizacije liste. Svaka referenca zauzima *delta* bajta.

Kako reference imaju fiksnu dužinu u bajtima (*delta*), kada se pristupa elementu sa indeksom i, adresa njegove reference dobija se kao $A + i \cdot delta$. Izračunavanje adrese za pristup elementu obavlja se u konstantnom vremenu nezavisno od dužine liste. U slučaju `L1 = L[i:i+k]`, vrši se k pristupanja listi `L`, kao i k kopiranja odgovarajućih objektnih referenci u `L1`, što rezultuje složenošću reda O(k).

7.5.2 Složenost u radu sa rečnicima

Red složenosti pojedinih operacija nad rečnikom može se objasniti na osnovu njegove interne memorijske organizacije (slika 7.14). Ključni deo memorijske strukture rečnika predstavlja *heš tabela*. Tabela sadrži uređene parove celobrojnih indeksa i referenci. Reference ukazuju na liste sa parovima oblika *ključ - vrednost*. Prilikom upisa ili čitanja odgovarajućeg objekta, vrednost ključa k preslikava se pomoću *heš funkcije* h u celobrojni indeks iz heš tabele.



Slika 7.14: Uprošćeni prikaz interne organizacije rečnika: po izračunavanju heš funkcije, pronalazi se odgovarajuća lista (za upis ili pretragu objekta sa navedenim ključem).

U opštem slučaju, heš funkcija h preslikava skup sa *velikim* brojem elemenata u *ograničeni*, po brojnosti daleko manji skup. Na primer, skup svih mogućih tekstualnih sekvenci (beskonačno mnogo članova), može se preslikati u celobrojni skup od 2^{32} vrednosti - indeksi u tabeli od 0 do $2^{32}-1$. Što je kardinalnost domena h veća, to je veći i broj elemenata iz domena koji se preslikavaju u *isti* indeks. U primeru sa tekst sekvencama, u svaki indeks iz posmatranog opsega preslikava se beskonačno mnogo sekvenci. Pri realizaciji rečnika, *pogodna* heš funkcija pridružuje ključevima različite indekse, svaki indeks sa približno *jednakom* verovatnoćom - svojstvo *uniformnosti*.

Neka se u rečnik upisuje vrednost sa ključem k. Po izračunavanju indeksa $i = h(k)$ mogu nastupiti dva slučaja: (1) na i-toj poziciji u tabeli nema ni jedne memorisane vrednosti; (2) postoji lista sa bar jednim elementom na koju ukazuje odgovarajuća referenca iz tabele. U prvom slučaju formira se jednoelementna lista sa ključem k i pridruženom vrednošću. U drugom slučaju, u već postojeću listu dodaje se par ključ-vrednost, ali *samo* ako se ključ k ne nalazi u listi. U suprotnom, par sa ključem k ažurira se novom vrednošću. Ako je broj parova u rečniku jednak n, u *najnepovoljnijem* slučaju svi parovi nalaze se u istoj listi pa se prilikom upisa mora proći kroz sve ključeve - obavlja se linearna pretraga. Kako izračunavanje heš funkcije ima konstantno trajanje bez obzira na popunjenost rečnika, red složenosti pri upisu u najnepovoljnijem slučaju jednak je $O(n)$.

Kada heš tabela nije potpuno popunjena, uz korišćenje uniformne heš funkcije, može se *očekivati* da će najveći broj upisa u rečnik biti obavljen u vrlo malom broju pristupa odgovarajućoj listi parova. U praksi se, za popunjenost do 75 procenata, očekuje jedan do dva pristupa pa red složenosti pri upisu, u *prosečnom* slučaju, iznosi $O(1)$. Prikaz složenosti pojedinih operacija/metoda u radu sa rečnicima dat je u tabeli 7.3. Navedeni redovi složenosti odnose se na prosečan i najnepovoljniji slučaj. Ostavlja se čitaocu da sam razmotri redove složenosti ostalih operacija iz tabele.

operacija	**prosečno**	**najnepovoljnije**
`n = len(D)`	O(1)	O(1)
`v = D[k]`	O(1)	O(n)
`D[k] = v`	O(1)	O(n)
`del D[k]`	O(1)	O(n)
`D2 = D.copy()`	O(n)	O(n)

Tabela 7.3: Vremenska složenost u radu sa rečnicima. Kopiranje rečnika predstavlja najskuplju operaciju.

Iz prethodnog izlaganja postaje jasno zašto ključevi *moraju* da budu objekti *nepromenljivih* tipova. Kako heš funkcija preslikava istu vrednost ključa u isti indeks, ako bi se ključ menjao tokom rada programa, bilo bi nemoguće pronaći listu koja memoriše odgovarajući objekat.

ŠTA JE NAUČENO

- U računaru se mere dva vremena - realno i sistemsko. Sistemsko vreme izražava se brojem proteklih sekundi u odnosu na referentni datum u prošlosti.
- Pojedini algoritmi ubrzavaju se pamćenjem često korišćenih međurezultata izračunavanja u rečniku. U dinamičkom programiranju problem se rešava podelom na preklapajuće potprobleme čija se rešenja pamte i ponovo koriste.
- Analiza složenosti algoritma ispituje vezu između veličine ulaznih podataka (dimenzije problema) i utroška resursa potrebnih za rešavanje problema. Najčešće se razmatra utrošak vremena i memorijskih resursa.
- Vremenska funkcija složenosti izražava vezu između veličine ulaza i trajanja izvršavanja. Efikasnost algoritma određuje se preko brzine rasta funkcije složenosti kada veličina ulaza raste i teži beskonačnosti - asimptotska analiza.

- Asimptotska analiza složenosti omogućava da se algoritmi mogu porediti nezavisno od osobina konkretnih računarskih sistema na kojima se izvršavaju.

- U programiranju se najčešće analizira najnepovoljniji slučaj kada su ulazni podaci raspodeljeni tako da se algoritam najduže izvršava. Algoritam ima red složenosti O(g), ako njegova funkcija složenosti ne raste brže od g, kada veličina ulaza teži beskonačnosti.

- Linearno pretraživanje liste od n elemenata ima red složenosti O(n), a binarno O($\log_2 n$). Binarno pretraživanje može se upotrebiti samo kada je lista sortirana.

- Metoda sortiranja *Counting sort* upotrebljava se ako su elementi zadate liste prirodni brojevi iz ograničenog opsega. Njen red složenosti je O(n), što predstavlja najbolji mogući rezultat za sortiranje. U opštem slučaju, metode koje se zasnivaju na poređenju ne mogu imati niži red složenosti od O($n\log_2 n$).

- Metoda sortiranja *Selection sort* sortira zadatu listu u mestu - ne zahteva se dodatni memorijski prostor. Njen red složenosti je O(n^2).

- Metoda sortiranja *Merge sort* zahteva dodatni memorijski prostor jednak zadatoj listi, ali zato postiže red složenosti od O($n\log_2 n$).

- Metoda sortiranja *Quick sort* smatra se najefikasnijim postupkom sortiranja u mestu za najveći broj praktičnih situacija. Njen red složenosti u prosečnom slučaju iznosi O($n\log_2 n$). Ona je tada, zbog povoljnijih konstanti, brža od metode *Merge sort*. U najnepovoljnijem slučaju, njen red složenosti iznosi O(n^2).

- Da bi se neki program mogao uspešno analizirati sa stanovišta složenosti, treba poznavati i redove složenosti bibliotečkih funkcija i metoda koje se u njemu koriste.

8. Obrada grešaka u programu

Prilikom upoznavanja sa osnovnim elementima jezika Pajton, pomenuta su dva tipa grešaka koje se mogu pojaviti u programu: *sintaksne* greške odnose se na nepravilnu upotrebu jezičkih konstrukcija poput naredbi, imena promenljivih ili pravila za indentaciju koda;[1] *semantičke* greške predstavljaju logičke nedostatke u radu algoritma (glava 2.2.4). Interpreter ne dozvoljava pokretanje sintaksno neispravnih programa pa se ovaj tip grešaka lako detektuje i uklanja. Nasuprot njima, semantičke greške prekinuće rad programa u toku izvršavanja - *greške pri izvršavanju*,[2] ili će program nastaviti sa radom, ali će proizvesti nekorektan izlaz - *logičke greške*. U ovoj glavi razmatraju se načini kojima se realizuje detektovanje i obrada grešaka pri izvršavanju.

8.1 Trag i tipovi grešaka

Program koji ne obrađuje informacije iz spoljnog sveta nije naročito koristan. Ulazni podaci mogu dospeti u program na različite načine: sa tastature, putem grafičkog korisničkog okruženja, iz ulazne datoteke sa diska, preko mreže ili na neki drugi način. Bez obzira na izvor, određene vrednosti ulaznih veličina mogu izazvati greške pri izvršavanju koje prekidaju rad programa.

[1] Pravila indentacije važe samo za jezik Pajton. Drugi jezici ne nameću ograničenje po tom pitanju.
[2] Engl. *Runtime errors*.

! Radno okruženje računarskih programa predstavlja svet u kome vlada Marfijev zakon. Zakon kaže da *sve loše što može da se desi, desiće se*! Programeri, koji očekuju da korisnici poštuju specifikaciju u pogledu ulaznih veličina, biće razočarani odmah pošto program napusti svog tvorca i nađe se u realnom svetu.

Interpreter izvršava program tako što obrađuje liniju po liniju u izvornom kodu. Dat je primer koji učitava dva cela broja i ispisuje njihov količnik:

```
1 x = int(input('x= '))
2 y = int(input('y= '))
3 print('x/y=', x/y)
```

```
x= 2
y= 3.1
Traceback (most recent call last):
  File "C:/programi/p8_1a.py", line 2, in <module>
    y = int(input('y= '))
ValueError: invalid literal for int() with base 10: '3.1'
>>>
==== RESTART: C:/programi/p8_1a.py ====
x= 10
y= 0
Traceback (most recent call last):
  File "C:/programi/p8_1a.py", line 3, in <module>
    print('x/y=', x/y)
ZeroDivisionError: division by zero
```

U prethodnom programu, ako bar jedan od unetih brojeva nije ceo, nastaje greška pri izvršavanju tipa `ValueError`. Interpreter smešta podatke o grešci u *objekat* klase `ValueError` koji, između ostalog, sadrži i red izvornog koda u kome je greška nastala (u primeru, r2), kao i njen opis. U drugom pokušaju, korisnik je uneo celobrojne podatke, ali je uneti delilac jednak nuli. Zato se, prilikom poziva funkcije `print()` u r3, pojavljuje greška tipa `ZeroDivisionError`.

Objekat greške sadrži *trag*[3] kojim se navodi niz poziva funkcija (ili metoda) koji je doveo do greške, uključujući broj reda svakog poziva i broj reda u kojoj je greška nastala. Kako je u gornjem primeru izvorni kod organizovan u obliku skripta (modula) bez funkcija, to trag sadrži sledeću informaciju:

```
File "C:/programi/p8_1a.py", line 3, in <module>
```

Sledeći primer ilustruje nešto komplikovaniji trag za isti tip (klasu) greške:

[3] Engl. *Trace*.

```
def podeli(x,y):
    return x/y

x, y = 1, 0
z = podeli(x,y) # deljenje nulom
print('x/y', z)
```

```
Traceback (most recent call last):
  File "C:/programi/p8_1b.py", line 5, in <module>
    z = podeli(x,y)
  File "C:/programi/p8_1b.py", line 2, in podeli
    return x/y
ZeroDivisionError: division by zero
```

Trag pokazuje da sekvenca, koja je proizvela grešku, započinje pozivom funkcije `podeli()` u petom redu skripta. Potom se, u samoj funkciji, greška pojavljuje prilikom deljenja u naredbi `return` (r2). Navedeni brojevi redova odnose se na istu datoteku (*p8_1b.py*). Kada bi se funkcije nalazile u različitim modulima, datoteke iz traga bile bi različite. U tabeli 8.1 nabrojane su najčešće predefinisane klase grešaka u Pajtonu.

tip	**dešava se**
`ValueError`	kada je argument operacije ili funkcije ispravnog tipa, ali ima pogrešnu vrednost
`NameError`	prilikom pokušaja pristupanja objektu preko nepostojećeg imena
`TypeError`	prilikom pokušaja izvršavanja operacije, funkcije ili metode sa argumentom pogrešnog tipa
`IndexError`	prilikom pristupanja sekvenci objekata kada je navedeni indeks van trenutnog opsega
`OverflowError`	prilikom izračunavanja realnog izraza kada je vrednost prevelika
`ZeroDivisionError`	prilikom deljenja sa nulom
`IOError`	pri neuspeloj ulazno-izlaznoj operaciji (na primer, pri čitanju iz nepostojeće datoteke)
`KeybordInterrupt`	kada korisnik sa tastature unese sekvencu <Ctrl><C>

Tabela 8.1: Najčešće korišćeni predefinisani tipovi (klase) grešaka u Pajtonu. Tip iz poslednje vrste nema sufiks *Error*. On nastaje u situaciju kada korisnik *zahteva* prekid rada programa (engl. *Interrupt* - prekid).

8.2 Defanzivno programiranje

Greške u programu predstavljaju najveći izvor frustracije, kako za korisnika, tako i za programera. Ako se izuzmu logičke greške koje nastaju kao posledica nekorektnog algoritma, otklanjanje grešaka u izvršavanju, putem *defanzivnog pristupa*, zasniva se na principu *bolje sprečiti nego lečiti*.

8.2.1 Provera korektnosti ulaznih podataka

Obezbeđivanje programa od neadekvatnog korisničkog unosa može se ostvariti primenom defanzivnog pristupa pri obradi ulaznih veličina. U ovom pristupu *proverava se* da li *sve* ulazne veličine programa odgovaraju po tipu i vrednosti prethodno utvrđenoj *specifikaciji*. Specifikacijom se zahteva da uneti podaci imaju smisla za posmatrani problem. Na primer, specifikacija nalaže da uneti tekst predstavlja validnu reprezentaciju celog broja. Slično, prilikom realizovanja funkcije, potrebno je proveriti da li svi ulazni parametri zadovoljavaju njenu specifikaciju.

Problem 8.1 — Deljenje u defanzivi. Uneti sa tastature dva cela broja i ispisati njihov količnik. U slučaju neadekvatnog unosa, obavestiti korisnika primerenom porukom. ■

Program koji deli unete celobrojne veličine mogao bi se realizovati u defanzivnom maniru na sledeći način:

```
1  # proverava da li je tekst validan ceo broj
2  def je_ceo(txt):
3      if not txt:            # isto kao len(txt) == 0
4          return False
5      elif txt[0] == '-':
6          return txt[1:].isdigit()
7      else:
8          return txt.isdigit()
9  
10 x, y = input('x= '), input('y= ')
11 if je_ceo(x) and je_ceo(y):
12     x, y = int(x), int(y)
13     if y:                   # isto kao y != 0
14         print('x/y=', x/y)
15     else:
16         print('y ne sme biti 0!')
17 else:
18     print('uneti brojevi moraju biti celi!')
```

```
x= 10
y= 1.1
uneti brojevi moraju biti celi!
>>>
==== RESTART: C:/programi/p8_1c.py ====
x= 10
y= 0
y ne sme biti 0!
>>>
==== RESTART: C:/programi/p8_1c.py ====
x= -3
y= 1
x/y= -3.0
```

Funkcija `je_ceo()` vraća `True` ako i samo ako njen tekstualni argument predstavlja validan zapis celog broja (r2-8). U slučaju prazne tekstualne sekvence vraća se `False` (r3-4). Primetiti da se neprazna tekstualna sekvenca tretira kao logička istina, a prazna kao neistina - ovo važi i za sve ostale sekvence! Ako sekvenca počinje znakom minus (r5), onda se testira da li je ostatak sekvence validan niz cifara (r6). U tu svrhu koristi se tekstualna metoda `isdigit()`. Uočiti da, za jednočlanu sekvencu, izraz `x[1:]` vraća prazan tekst koji ne predstavlja validan zapis broja. Slično se postupa i kada sekvenca ne započinje znakom `"-"` (r8).

U samom programu, po proveri valjanosti tekstualnog unosa, vrši se konverzija u celobrojne objekte (r12). Jednostavniji deo postupka provere odnosi se na ispitivanje da li je delilac različit od nule (r13) - primetiti da objekti brojčanih tipova različiti od nule predstavljaju logičku istinu! Prilikom neregularnog unosa, program jednostavno ispisuje poruku o grešci i prekida sa radom (r16, 18). U drugoj varijanti rešenja, posle pogrešnog unosa, korisnik može da pokuša ponovo, pri čemu se ceo postupak provere smešta u petlju:

Program 8.1 — Deljenje u defanzivi.

```
1  # proverava da li je tekst validan ceo broj
2  def je_ceo(txt):
3      # kao u prethodnom primeru ...
4
5  unos_ok = False
6  # unos se ponavlja sve do ispravno unetih podataka
7  while not unos_ok:
8      x, y = input('x= '), input('y= ')
9      if je_ceo(x) and je_ceo(y):
10         x, y = int(x), int(y)
```

```
11         if y:
12             unos_ok = True
13             print('x/y=', x/y)
14         else:
15             print('y ne sme biti 0!')
16     else:
17         print('uneti brojevi moraju biti celi!')
```

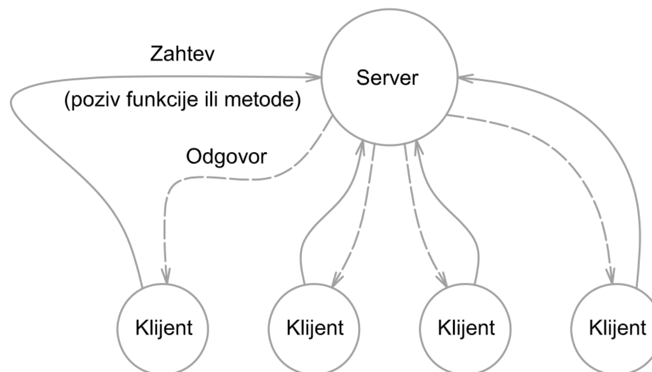
U ovoj verziji programa korisnik se, uz pomoć petlje `while` (r7-17), vraća na “popravni” sve do ispravnog unosa traženih podataka. Kontrolu ostanka u petlji preuzima logička promenljiva `unos_ok`, koja inicijalno ukazuje na `False` (r5). Petlja se napušta samo ako su uneti podaci ispravni, pošto se `unos_ok` postavi na `True` (r12). Naravno, ukoliko korisniku dosadi da unosi pogrešne podatke, on *može prekinuti* rad programa uz pomoć kontolne sekvence <Ctrl><C> :

```
x= 1
y= 0
y ne sme biti 0!
x= q
y= 2
uneti brojevi moraju biti celi!
x= 1
y= 2
x/y= 0.5
x= 1
y=
Traceback (most recent call last):
  File "C:/programi/p8_1.py", line 8, in <module>
    x, y = input('x= '), input('y= ')
  File "C:\Python35\lib\idlelib\PyShell.py", line 1386, in readline
    line = self._line_buffer or self.shell.readline()
KeyboardInterrupt
```

8.2.2 Model klijent-server

Informacije potrebne za rad programa ne nastaju uvek kao posledica korisničkih akcija. Vrlo često, *interakcija* između programa i čoveka može se zameniti interakcijom između programa i drugih programa. Obrada grešaka se tada komplikuje jer izvor greške ne ispisuje poruku za korisnika kao u slučaju programa za deljenje. Na primer, neka se program A, specijalizovan za isporučivanje stranica nekog sajta, izvršava na određenom računaru na Internetu. Program A zove se još i *veb server*. Njegove usluge mogu koristiti čitači poput Chorme-a ili Internet Explorer-a, koji šalju zahteve preko mreže kako bi dobili tražene stranice i prikazali ih svojim korisnicima.

U opštem slučaju, interakciju između različitih programa možemo predstaviti modelom *klijent-server* ilustrovanim na slici 8.1. *Server* je program koji *pruža* usluge (servise) strani koja te usluge potražuje - *klijentu*. Klijent može biti kako korisnik, tako i drugi program. Već spomenuti program A predstavlja serversku komponentu, dok se svaki čitač nalazi u ulozi klijenta. Čitač iscrtava dobijenu stranicu na računaru korisnika. Sa stanovišta korisnika koji zadaje željenu adresu (klijent), čitač je u ulozi servera koji prikazuje zahtevane stranice. Očigledno je da se jedan program može naći u obe uloge, u zavisnosti da li pruža ili potražuje uslugu.



Slika 8.1: Klijent-server model. Klijent šalje zahtev za uslugu serverskoj strani. Tom prilikom, on navodi određene argumente koji dodatno određuju traženu uslugu. Jedan server može opsluživati veći broj klijenata istovremeno.

8.2.3 Informacija o stanju obrade u klijent-server modelu

Klijent-server interakcija može se posmatrati i na *mikro* nivou: funkcija u Pajtonu predstavlja serversku komponentu koja se poziva iz drugih, klijentskih delova nekog programa. Postavlja se pitanje kako funkcija, u kojoj je nastao problem, prosleđuje informaciju o grešci u pozivajući deo programa? Na prvi pogled, čini se da je dovoljno da funkcija vrati *specijalnu* vrednost kao indikaciju nastanka greške. Na primer, funkcija koja deli dva broja može vratiti `None` u slučaju neregularnosti. Međutim, kako u funkciji mogu da se jave različiti tipovi grešaka (deljenje nulom, neregularno `x`, neregularno `y`, neregularni `x` i `y`), informaciju o *stanju izračunavanja* (*status*) treba *pogodno kodirati* i vratiti *odvojeno* od osnovne povratne vrednosti.

Pozivajuća (klijentska) strana *dužna* je, prema defanzivnom pristupu, da testira statusnu informaciju pre nego što upotrebi povratni rezultat. Sledi primer koji funkcionalnost deljenja realizuje u serverskom modulu *deli_server.py*, a klijentski program smešta u skript *deli_klijent.py*:

```
# server: deli_server.py
# proverava da li je tekst validan ceo broj
def je_ceo(txt):
    # .... kao u prethodnom listingu

# serverska funkcija
# ako je status: 0 - sve ok, 1 - deljenje sa 0
#                2 - x ne valja, 3 - y ne valja
def deli(x,y):
    if je_ceo(x):
        if je_ceo(y):
            x, y = int(x), int(y)
            if y:
                return (0, x/y)
            else:
                return (1, 0)
        else:
            return (3, 0)
    else:
        return (2, 0)
```

```
# test klijent: deli_klijent.py
import deli_server as server

# prekida rad sa ctrl-c
while True:
    x, y = input('x '), input('y ')
    status, z = server.deli(x, y)
    if status == 0:
        print ('x/y', z)
    elif status == 1:
        print('deljenje nulom!')
    elif status == 2:
        print('ne valja x!')
    else:
        print('ne valja y!')
```

Funkcija `deli()`, iz modula `deli_server.py`, sadrži tri naredbe `if-else` (r10-

20) putem kojih se detektuju četiri mogućnosti: operacija uspela (`status=0`), deljenje nulom (`status=1`), prvi argument neregularan (`status=2`) i drugi argument neregularan (`status=3`). Funkcija vraća torku u kojoj prvi element označava status, a drugi, vrednost količnika za uspešnu operaciju. Pri pogrešnom unosu, drugi element nije bitan (ovde 0).

Klijentski deo aplikacije učitava vrednosti sa tastature (r6), poziva serversku funkciju i raspakuje elemente rezultujuće torke (r7). Potom se, prilikom tumačenja povratnih vrednosti, primenjuje defanzivan pristup (r8-15). Program insistira na ponovnom unosu, sve dok korisnik ne izazove prekid rada sa tastature:

```
x 1
y 2.3
ne valja y!
x qqq
y 1
ne valja x!
x 1
y 0
deljenje nulom!
x 15
y -3
x/y -5.0
x
Traceback (most recent call last):
  File "C:/programi/deli_klijent.py", line 6, in <module>
    x, y = input('x '), input('y ')
  File "C:\Python35\lib\idlelib\PyShell.py", line 1386, in readline
    line = self._line_buffer or self.shell.readline()
KeyboardInterrupt
```

8.3 Greške kao izuzeci

U defanzivnoj varijanti serverske funkcije `deli_server.deli(x,y)`, bilo je nužno sprovesti složenu proceduru kontrole ulaznih parametara `x` i `y`. Struktura serverskog koda opterećena je brojnim naredbama `if` koje proveravaju valjanost parametara u cilju formiranja odgovarajuće statusne informacije. Klijentski kod nije ništa jednostavniji – posle poziva funkcije, treba proveriti vraćenu statusnu informaciju. Pored *povećane* složenosti rešenja, defanzivno programiranje donekle *usporava* rad kako klijenta, tako i servera. Usporavanje potiče od toga što se, bez obzira na valjanost unetih vrednosti, vrše brojne provere podataka. Ako se pretpostavi da se greške pri unosu ne dešavaju često, onda je efekat usporavanja još očigledniji. Greške pri izvršavanju nazivaju se još i *izuzeci*[4] jer se *ne očekuje* da se često dešavaju.

[4] Engl. *Exceptions*.

Postoje brojni problemi u kojima je *veoma teško* proveriti sve ulazne kombinacije podataka. Nekada je to i *nemoguće*. Na primer, posmatra se klijent koji komunicira sa serverom preko mreže. Šta se dešava kada se, usled razloga koje klijent *ne može* da kontroliše, mrežna veza prekine? Deo klijentskog koda koji poziva serversku funkciju prouzrokovaće prestanak rada programa. Kako programer treba da obezbedi svoj program u ovom slučaju? Slična situacija dešava se i kada klijent piše u datoteku na disku na kome više nema slobodnog prostora ili kada klijent nema dovoljna ovlašćenja u datom direktorijumu.

U pomenutim okolnostima, defanzivno programiranje postaje izuzetno složeno pa bi programer došao u iskušenje da *odustane* od provere. Zbog toga se u više programske jezike uvodi novi koncept – *obrada izuzetaka*[5] po potrebi. Umesto principa "bolje sprečiti nego lečiti", koncept se zasniva na *traženju oproštaja* kada se problem desi – u ovim *retkim* situacijama obavlja se odgovarajuća *procedura oporavka*.

8.3.1 Kritična sekcija. Kontrolna struktura `try - except`

Obrada izuzetaka oslanja se na to da se pojedini delovi koda, u kojima može doći do greške (npr. ulazno-izlazne operacije), obeleže kao *kritične sekcije* u programu. Ako greška zaista i nastupi, interpreter preusmerava tok izvršavanja iz kritične sekcije u deo programa koji obrađuje presretnutu grešku. Sledeći primer ilustruje koncept:

Problem 8.2 — Deljenje sa izuzecima. Kreirati serversku funkciju koja deli dva prirodna broja, a eventualne greške tretirati preko obrade izuzetaka. U slučaju neregularnog unosa, funkcija treba da vrati pogodnu poruku. ■

```
1  # obrada izuzetaka
2  def deli(x,y):
3      '''Serverska funkcija koja deli dva prirodna broja '''
4      poruka, količnik = None, 0
5      try:
6          količnik = int(x)/int(y)
7      except:
8          poruka = 'Greška u podacima'
9
10     return poruka, količnik
11
12 # test klijent (ctrl-c za prekid)
13 while True:
```

[5] Engl. *Exception handling*.

```
14      x, y = input('x = '), input('y = ')
15      greška, k = deli(x, y)
16
17      if not greška:        # može i if greška == None
18          print(f'{x}/{y} = {k}')
19      else:
20          print(greška)
```

Funkcija `deli()` vraća dvočlanu torku u kojoj se prvi objekat odnosi na statusnu poruku, a drugi na količnik prosleđenih brojeva (r10). Kada su navedeni parametri ispravni, poruka ima vrednost `None`.

Deo koda između ključnih reči `try` i `except` predstavlja kritičnu sekciju u kojoj se *pretpostavlja* mogućnost pojave greške (r6), a blok naredbe `except` definiše postupak za njenu obradu (r8). Oba bloka mogu da sadrže proizvoljan broj naredbi. Ako se u bloku naredbe `try` pojavi greška, usled loše tekstualne reprezentacije broja ili zbog toga što je delilac jednak nuli, programski tok se *prekida* na mestu greške i *preusmerava* na prvu naredbu bloka `except`. Promenljiva `poruka` opisuje prirodu greške (r8), a program se nastavlja prvom naredbom van bloka naredbe `except` (r10). Ako se izuzetak ne desi, preskače se blok naredbe `except` i tok programa, sa naredbe u r6, prelazi na naredbu u r10. Prilikom nastanka greške u izračunavanju izraza na desnoj strani r6, promenljiva `količnik` će i dalje ukazivati na nulu.

Klijentski deo programa realizovan je pomoću defanzivnog programiranja s tim da se ovde, po pozivu funkcije `deli()`, proverava vrednost vraćene poruke i postupa u skladu sa njom (r17-20). Uočiti da se vrednost `None` u logičkim izrazima tretira kao neistina (r17). Unos je smešten u beskonačnu petlju (r13-20), a izvršavanje se može prekinuti sa tastature:

```
x = 2
y = 3.1
Greška u podacima
x = 2
y = 0
Greška u podacima
x = 1
y = 2
1/2 = 0.5
x =
Traceback (most recent call last):
  File "C:/programi/p8_2a.py", line 15, in <module>
    x = input('x = ')
  .... ....
KeyboardInterrupt
```

Navedeni način obrade izuzetaka *nije* preporučljiv jer se, bez obziran na tip, greške obrađuju na isti način. Poznato je da interpreter smešta informaciju o grešci u objekat određenog tipa (tabela 8.1). Kako bi se različite greške obradile na njima svojstven način, koristiti se drugi oblik iste kontrolne strukture sa više blokova naredbe `except`:

Program 8.2 — Deljenje sa izuzecima.

```
# obrada izuzetaka
def deli(x,y):
    '''Server za deljenje prirodnih brojeva'''
    poruka, količnik = None, 0
    try:
        količnik = int(x)/int(y)
    except ValueError:
        poruka = 'x i y moraju biti celi!'
    except ZeroDivisionError:
        poruka = 'y ne sme biti 0!'
    except:
        poruka = 'nepoznata greška!'

    return poruka, količnik

# test klijent (ctrl-c za prekid)
try:
    while True:
        x, y = input('x = '), input('y = ')
        greška, k = deli(x, y)

        if not greška:      # može i if greška == none
            print(f'{x}/{y} = {k}')
        else:
            print(greška)

except KeyboardInterrupt:
    print('zdravo!')
```

U poslednjem rešenju, umesto jednog, navedena su tri bloka naredbe `except` od kojih se, ako dođe do greške, može izvršiti *samo* jedan. Ako se prilikom izvršavanja kritične naredbe (r6) ispostavi da neki od argumenata ne predstavlja zapis celog broja, onda interpreter generiše grešku tipa `ValueError` i izvršava naredbu u r8. Kada je

konverzija uspešna, ali je delilac jednak nuli, izvršava se naredba u bloku greške `ZeroDivisionError`. Konačno, ako se prilikom izvršavanja pojavi greška nekog drugog tipa, onda se izvršava naredba iz poslednjeg bloka `except` (r12).

U slučajevima kada je potrebno obraditi *više* tipova greški na *isti* način, umesto ponavljanja pojedinačnih blokova, može se navesti sledeća forma naredbe `except`:

```
except (TipGreške1, TipGreške2,..., TipGreškeN):
```

Na pimer, sa `except (TypeError, NameError):`, obe greške biće obrađene na isti način. U zagradama se može navesti proizvoljan broj tipova odvojenih zapetama.

Izvorni kod programa za testiranje (klijent) promenjen je tako da, prilikom prekidanja programa sa tastature, interpreter ne ispisuje poruku o grešci, već prigodnu poruku za kraj rada. Zbog toga je petlja `while` (r18-25) smeštena u blok naredbe `try`, a prigodna poruka ispisuje se kada se generiše `KeyboardInterrupt`:

```
x = 1
y = 0
y ne sme biti 0!
x = qqq
y = 12
x i y moraju biti celi!
x = 10
y = 5
10/5 = 2.0
x =
zdravo!
```

Kako je u programu za testiranje izostavljen *opšti* oblik naredbe `except` koji ne navodi eksplicitno tip greške, presreće se samo `KeyboardInterrupt`. Drugi mogući izuzeci *ne bi* bili obrađeni, već bi interpreter prekinuo izvršavanje i ispisao poruku o grešci. U opštem slučaju, kada se izuzetak ne obradi u funkciji u kojoj je nastao, objekat greške se *prosleđuje* u pozivajuću celinu na *mesto poziva* funkcije. Ako se taj poziv ne nalazi u okviru strukture `try-except` koja obrađuje presretnutu grešku, ona se prosleđuje na sledeći nivo u hijerarhiji poziva. Ako se greška ne obradi ni u okviru glavnog programa, onda se rad prekida, a interpreter ispisuje trag greške.

8.3.2 Eksplicitno prosleđivanje izuzetaka. Naredba `raise`

Pozvana funkcija (server) često *nema* dovoljno informacija o stanju celokupnog sistema pa se unutar nje ne može izvršiti adekvatna obrada greške. Na primer, kako postupiti u funkciji koja šalje podatke preko mreže kada se veza prekine? U funkciji se može obaviti delimična obrada poput zapisivanja vremena i razloga greške u dnevničku datoteku. Sada je na pozivajućoj strani da utvrdi kako će se program ponašati u toj prilici: pokušati ponovo sa slanjem, pitati korisnika za potrebnu akciju i slično. U ovakvim situacijama,

izuzetak se može *eksplicitno* proslediti u pozivajuću celinu upotrebom naredbe `raise`:

```
# obrada izuzetaka
from time import asctime
def deli(x,y):
    '''Server koji deli dva prirodna broja'''
    try:
        return int(x) / int(y)
    except:
        print('serverski dnevnik:', asctime(), 'greška!')
        raise

# test klijent (ctrl-c za prekid)
try:
    while True:
        x, y = input('x = '), input('y = ')
        try:
            print(f'{x}/{y} = {deli(x, y)}')
        except ValueError:
            print('program: x i y moraju biti celi!')
        except ZeroDivisionError:
            print('program: y ne sme biti 0!')

except KeyboardInterrupt:
    print('zdravo!')
```

Ako je naredba `raise` samostalno navedena bez tipa greške (r9), onda se izuzetak prosleđuje u pozivajuću funkciju ili program bez obzira na tip. Program za testiranje obavlja obradu greške koja je prosleđena iz funkcije `deli()`. Primer ilustruje situaciju u kojoj jedna kritična sekcija (r13-20), *obuhvata* drugu (r16):

```
x = 1
y = 0
serverski dnevnik: Mon Jun 19 15:09:03 2017 greška!
program: y ne sme biti 0!
x = qq
y = 1
serverski dnevnik: Mon Jun 19 15:09:09 2017 greška!
program: x i y moraju biti celi!
x = 2
y = 5
```

```
2/5 = 0.4
x =
zdravo!
```

U serverskoj funkciji je upotrebljena funkcija `asctime()` iz modula `time` (r8). Ako je pozvana bez argumenata, ona vraća tekstualni opis sistemskog vremena (glava 6.1). Naredbom `raise` (r9), greška koja je nastala u serverskoj funkciji prosleđuje se u pozivajući program. Naredba `raise` može da prosledi i grešku odabranog tipa:

```
>>> raise ValueError
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    raise ValueError
ValueError
```

Prosleđivanje izuzetka sa predefinisanom porukom

Pajton pruža mogućnost da korisnik definiše *nove* tipove izuzetaka prema potrebi. O mehanizmu definisanja novih tipova objekata biće reči u glavi 10. Ovde se naglašava mogućnost korišćenja već postojećeg tipa `Exception`, kako bi se defnisao izuzetak sa odgovarajućom porukom koju interpreter ispisuje ako se izuzetak ne obradi. U tu svrhu se koristi konstruktor predefinisanog tipa `Exception` kome se prosleđuje odgovarajuća poruka:

```
>>> raise Exception('veoma izuzetan izuzetak!')
Traceback (most recent call last):
  File "<pyshell#39>", line 1, in <module>
    raise Exception('veoma izuzetan izuzetak!')
Exception: veoma izuzetan izuzetak!
```

Poruka se može ispisati, bez dodatnih informacija iz traga, na sledeći način:

```
>>> try:
       # ....
       raise Exception('veoma izuzetan izuzetak!')
    except Exception as e:
       print(e)   # ispis predefinisane poruke

veoma izuzetan izuzetak!
```

Konstrukcija `except Exception as e:` omogućava da se objekat greške imenuje promenljivom `e` i na taj način postane direktno dostupan u programu. Sada se, sa `e`, mogu obavljati različite operacije među kojima je i ispis predefinisane poruke. Imenovanje se može primeniti na objekte svih tipova izuzetaka.

8.3.3 Bezuslovno izvršavanje pri obradi izuzetaka. Naredba `finally`

Prilikom obrade izuzetaka, često treba *bezuslovno* izvršiti određene akcije, bez obzira na to da li se izuzetak desio ili ne. Pojam bezuslovnog izvršavanja postaće jasniji iz sledećeg primera:

```
# obrada izuzetaka - finally

def f(x):

    try:
        print(int(x))
    except ValueError:
        print('nije ceo broj!')
    finally:
        print('gotovo')

def g(x):

    try:
        print(int(x))
    except ValueError:
        print('nije ceo broj!')

    print('gotovo')

def h(x):

    try:
        print(int(x))
        return
    except ValueError:
        print('nije ceo broj!')
    finally:
        print('gotovo')
```

Funkcija `f()` obavlja jednostavno izračunavanje: uneti argument se konvertuje u ceo broj i potom ispisuje (r6). Ako se prilikom konverzije desi greška tipa `ValueError`, ispisuje se odgovarajuća poruka u bloku naredbe `except` (r8). Blok naredbe `finally` izvršava se *bez obzira* da li je do greške došlo ili ne, što rezultuje ispisom reči `'gotovo'` na kraju izvršavanja funkcije:

```
>>> f('aa')   # sa greškom
nije ceo broj!
gotovo
>>> f('-10')  # bez greške
-10
gotovo
```

Postavlja se pitanje zašto se u jezik uvodi naredba `finally` kada se pomenuto ponašanje postiže i naredbama iz funkcije `g()`? Odgovor je povezan sa pojmom bezuslovnog izvršavanja. Neka se, prilikom izvršavanja kritične sekcije u funkciji `g()` (r15), dogodi neka nepredviđena greška koja nije tipa `ValueError`. Tada bi se izuzetak smatrao *neobrađenim*, funkcija bi prekinula sa radom, a interpreter bi grešku prosledio u pozivajuću celinu. U takvom scenariju, naredba iz r19 ne bi bila izvršena! Isto se dešava i ako se u toku izvršavanja bloka naredbe `try` naiđe na naredbu `return`, a da pri tome nije nastupila greška. Funkcija bi tada bila okončana bez izvršavanja naredbi posle strukture `try-except`.

U funkciji koja radi sa eksternim sistemima poput diska ili mreže, a za potrebe nesmetane komunikacije, vrši se alokacija određenih sistemskih resursa. Na primer, prilikom čitanja ili upisa u bazu podataka, potrebno je *prvo* ostvariti vezu (*konekciju*) sa određenim korisničkim pravima nad podacima. Svaka konekcija reprezentuje se u memoriji odgovarajućom strukturom podataka. Pre nego što funkcija završi sa radom, *neophodno* je da se konekcija *bezuslovno zatvori*, odnosno da se *oslobode* privremeno zauzeti memorijski resursi. Prilikom zatvaranja konekcije, klijent sugeriše serverskoj aplikaciji (bazi podataka) da oslobodi resurse koji su bili pridruženi klijentu i da smanji broj tekućih konekcija za jedan. Ako server to ne bi učinio, pojedine "mrtve" konekcije nepotrebno bi umanjile broj dozvoljenih istovremenih pristupa ka bazi. Slično važi i u radu sa datotekama na disku. Čitaoci se ovde podsećaju na analogiju sa telefonskom mrežom: po završenom razgovoru spušta se slušalica da bi drugi mogli da nas dobiju!

Zbog svega rečenog, u radu sa eksternim sistemima neophodno je bezuslovno oslobađanje resursa, što se čini u bloku naredbe `finally`. Primer funkcije `h()` ilustruje bezuslovno izvršavanje ispisa (r29), uprkos izlasku iz funkcije (r25):

```
>>> h(21)
21
gotovo
```

ŠTA JE NAUČENO

- Greška koja prekida rad programa naziva se još i izuzetak. Tom prilikom, interpreter smešta informacije o grešci u objekat odgovarajućeg predefinisanog tipa.
- Defanzivno programiranje podrazumeva eksplicitnu proveru ulaznih podataka programa ili funkcije u smislu odstupanja od očekivane specifikacije. Funkcije generišu dodatnu statusnu informaciju kojom se signalizira pozivajućoj celini da li je i kakva je greška nastupila.
- Defanzivni pristup povećava dužinu izvornog koda, a ponekad je komplikovano ili čak neizvodljivo testirati sve problematične situacije (rad sa ulazno-izlaznim sistemima poput diska ili mreže).
- Aplikacija koja pruža usluge drugim aplikacijama naziva se server. Aplikacije koje koriste usluge servera, često preko računarske mreže, predstavljaju klijente. Paradigma klijent - server može se posmatrati i na nižem nivou: funkcije i metode za pružanje različitih usluga nazivaju se serverima, a pozivajuće celine klijentima.
- Sistem obrade izuzetaka zasniva se na presretanju greške na mestu nastanka, uz preusmeravanje kontrole toka na delove programa za njenu adekvatnu obradu. Ovo se postiže upotrebom kontrolne strukture `try-except-finally`.
- U programskom toku mogu se, uz pomoć naredbe `raise`, generisati različiti tipovi izuzetaka koji se onda prosleđuju u nadređene programske celine (na primer, iz funkcije u pozivajući program).
- Ako se izuzetak nigde ne obradi, onda interpreter prekida rad i ispisuje trag greške. Trag sadrži informaciju o sekvenci poziva funkcija i/ili metoda koja je proizvela grešku.

9. Tekstualne datoteke

U ovoj glavi razmatra se jedan od ključnih pojmova u računarstvu - pojam *datoteke*. Datoteke omogućavaju čuvanje srodnih podataka na spoljnoj memoriji računara, kao jedinstvenih entiteta koji opstaju i kada program prestane sa radom. Sadržaj datoteke može da se odnosi na različite tipove podataka, od korisnički definisanih informacija (tekst, slika, zvuk, video), preko različitih sistemskih podešavanja (konfiguracione datoteke) pa do samih programa u višem programskom jeziku (izvorni kod) ili njihovih ekvivalenata u izvršnom formatu (mašinski kod).

9.1 Pojam i interpretacija

Objekti različitih tipova smešteni su tokom rada programa u operativnu memoriju računara. Međutim, kako se sadržaj operativne memorije gubi po prestanku električnog napajanja, podatke je potrebno uskladištiti na nekom od trajnih memorijskih medijuma poput čvrstog (HD) ili poluprovodničkog diska (SSD).

> **!** Trajnost memorijskih medijuma odnosi se na period koji se meri u godinama. Na primer, *očekivani* životni vek HD diska, pod *standradnim* opterećenjem i uslovima čuvanja, iznosi 5 godina! Životni vek SSD diskova i fleš memorija smanjuje se sa porastom količine *upisanih* podataka na medijum. Pod uslovima koji su navedeni za HD diskove, imaju očekivani vek od 10 godina. Stoga je neophodno praviti *kopije* važnih podataka, pri čemu sadržaj starih diskova treba povremeno *prebacivati* na nove, u skladu sa pomenutim trajanjem!

Datoteka[1] predstavlja *imenovani* skup informacija koji je uskladišten na trajnom medijumu poput diska ili fleš memorije. Ime datoteke čini tekstualna sekvenca koja obično ima oblik `s1.s2` (`s1` i `s2` sekvence). Sekvenca `s2` naziva se *ekstenzijom* i asocira na tip datoteke. Na primer, datoteka `merge_sort.py` mogla bi da sadrži izvorni kod u Pajtonu koji obavlja sortiranje po algoritmu *Merge sort*. Iako su sve datoteke reprezentovane niskama bitova, sa stanovišta kako program *interpretira* ove niske, mogu se podeliti u dva osnovna tipa: *binarne* i *tekstualne* datoteke.

Binarne datoteke oblikovane su tako da program, u zavisnosti od konteksta primene, različite grupe bitova tretira na unapred definisan način. Na primer, binarna datoteka može da započne sa k bajtova koji označavaju broj zapisa o studentima, pri čemu je za informacije o svakom studentu (jedan zapis) potrebno n bajtova. Zapisi o studentima slede posle informacije o broju studenata. Ako program ne poznaje pravilo za grupisanje bitova, kao ni njihovo značenje, onda podaci u datoteci nemaju nikakvog smisla.

Program tumači sekvencu bitova tekstualne datoteke kao niz *redova teksta* odvojenih niskom bitova koja predstavlja simbol za novi red. Svaki red sastoji se od niza bitova koji reprezentuju pojedinačne karaktere. Da bi tekstualna datoteka bila čitljiva, program treba da poznaje *sistem kodiranja* pomoću koga je datoteka zapisana na spoljnu memoriju. U glavi 5.1.1, diskutovani su sistemi kodiranja i pridružene tablice preslikavanja putem kojih se binarne niske prevode u konkretne simbole. Čitaocima se savetuje da novokreirane tekstualne informacije sačuvaju korišćenjem `Unicode` standarda, čime se postiže *čitljivost* datoteka u različitim računarskim sistemima.

Iako po pravilu zauzimaju *više* mesta, tekstualne datoteke su, za razliku od binarnih, *čitljive* u različitim editorima teksta i mogu se lako ažurirati. Osim toga, jednostavnije je rekonstruisati ih ako dođe do greške u jednom delu podataka. Zbog toga će nadalje biti razmatrane samo tekstualne datoteke.

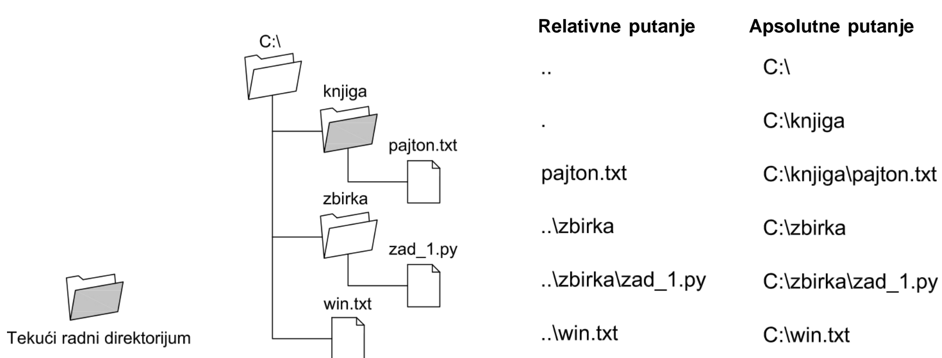
9.2 Sistem datoteka

Datoteke su organizovane u *sistem datoteka*[2] koji se, sa stanovišta korisnika ili programa, može posmatrati kao hijerarhijska struktura *direktorijuma*. Direktorijumi, pored datoteka, mogu da sadrže i druge direktorijume - *poddirektorijumi*. Direktorijum koji u spomenutoj hijerarhiji *nije* ujedno i poddirektorijum, naziva se *koreni* direktorijum.[3] Sistem datoteka može imati više korenih direktorijuma (na Windows-u, to su početni direktorijumi memorijskih medijuma poput `C:\` ili `D:\`).

Svaka datoteka može se identifikovati putem već pomenutog imena, kao i *apsolutne putanje* direktorijuma koji je neposredno sadrži. Apsolutna putanja direktorijuma `d` je tekstualna sekvenca koja uključuje imena svih direktorijuma na putanji od korenog direktorijuma pa do `d`, uključujući i njega. Imena u putanji razdvojena su *separatorom*

[1] Engl. *File*.
[2] Engl. *File system*.
[3] Engl. *Root directory*.

putanje koji je, u slučaju operativnog sistema Windows, predstavljen simbolom `\`. Apsolutna putanja datoteke uključuje i njeno ime. Na primer, neka se datoteka pod imenom `pajton.txt` nalazi u direktorijumu `knjiga` koji se, opet, nalazi u `C:\`. Apsolutna putanja za `pajton.txt` glasi: `C:\knjiga\pajton.txt` (slika 9.1).



Slika 9.1: Relativne i apsolutne putanje: primer se odnosi na operativni sistem Windows. Kada se nađe u relativnoj putanji, simbol "`.`" se odnosi na tekući, a "`..`" na nadređeni direktorijum.

Za lociranje datoteka, pored apsolutne, koristi se i *relativna putanja*. Relativna putanja predstavlja putanju od tekućeg *radnog direktorijuma* do posmatrane datoteke, uključujući i njeno ime. Radni direktorijum ukazuje na *eksplicitno zadato* mesto u sistemu datoteka, a inicijalno se odnosi na direktorijum u kome je pokrenut program. Ako bi radni direktorijum bio postavljen na `C:\`, onda bi relativna putanja za `pajton.txt` bila `knjiga\pajton.txt` (slika 9.1). Za potrebe rada sa sistemom datoteka koristi se modul `os` (tabela 9.1).

funkcija	**opis**
`os.getcwd()`	vraća putanju tekućeg radnog direktorijuma
`os.chdir(p)`	postavlja tekući radni direktorijum zadat putanjom `p`
`os.listdir(p)`	vraća listu direktorijuma i datoteka iz direktorijuma sa putanjom `p`
`os.mkdir(p)`	kreira direktorijum zadat putanjom (ako već ne postoji)
`os.rmdir(p)`	uklanja direktorijum zadat putanjom (ako je prazan)
`os.mkdirs(p)`	kreira sve direktorijume iz putanje (ako već ne postoje)
`os.rmdirs(p)`	uklanja sve direktorijume iz putanje (ako su prazni)
`os.remove(p)`	uklanja datoteku zadatu putanjom

Tabela 9.1: Česte operacije sa sistemom datoteka. Putanja `p` može biti kako apsolutna, tako i relativna (u odnosu na tekući radni direktorijum).

Najčešće korišćene operacije iz modula `os`, navedene u tabeli 9.1, biće ilustrovane u okviru interaktivne sesije:

```
>>> import os
>>> os.getcwd()  # tekući radni direktorijum
'C:\\Users\\Andjelko\\AppData\\Local\\Programs\\Python\\Python35'
>>> os.chdir('c:\\papers')  # promeni tekući radni direktorijum
>>> os.getcwd()
'c:\\papers'
>>> os.listdir()  # izlistaj sadržaj radnog dir.
['2016', '2017']
>>> os.listdir('c:\\papers\\2017')  # izlistaj zadati dir.
['jcce']
>>> os.mkdir('c:\\papers\\a')  # napravi zadati dir.
>>> os.listdir('c:\\papers)
['2016', '2017', 'a']
>>> os.rmdir('c:\\papers\\a')  # ukloni dir. (samo ako je prazan!)
>>> os.listdir('c:\\papers')
['2016', '2017']
>>> # napravi dir. i sve iz putanje, ako treba
>>> os.makedirs('c:\\papers\\a\\b\\x')
>>> # ukloni sve prazne dir. iz putanje počev od x!
>>> os.removedirs('c:\\papers\\a\\b\\x')
>>> os.listdir('c:\\papers')
['2016', '2017']
>>> # pošto je napravljena nova datoteka iz Windows-a
>>> os.listdir('c:\\papers')
['2016', '2017', 'novi.txt']
>>> os.remove('c:\\papers\\novi.txt') # ukloni datoteku
>>> os.listdir('c:\\papers')
['2016', '2017']
```

Pri zadavanju putanje u operativnom sistemu Windows, separator putanje *mora* da sadrži dodatno “\” jer bi se, u suprotnom, karakter posle “\” shvatio kao kontrolna sekvenca.[4] Prilikom listanja sadržaja direktorijuma vraća se *lista* relativnih putanja. Pored modula `os`, za potrebe rada sa putanjama, koristi se i modul `os.path` čije su često korišćene funkcije prikazane u tabeli 9.2. Sledi primer za navedene operacije iz modula `os.path`:

```
>>> import os.path as osp
>>> os.chdir('C:\\') # promena za tek. rad. dir.
>>> osp.isabs('c:\\Users')
True
>>> osp.abspath('a\\b\\c.txt') # u odnosu na tek. rad. dir.
'C:\\a\\b\\c.txt'
>>> osp.dirname('C:\\a\\b\\c.txt'), osp.basename('C:\\a\\b\\c.txt')
```

[4] U Linux/Unix operativnim sistemima ovo nije potrebno jer se za separator koristi simbol “/”. Na primer, ispravna apsolutna putanja mogla bi da bude: `/home/milos/pajton.txt`.

```
('C:\\a\\b', 'c.txt')
>>> osp.join('a','b','c.txt')  # spaja fragmente putanje.
'a\\b\\c.txt'
>>> osp.exists('C:\\ne_postoji.txt')
False
>>> osp.isfile('devlist.txt'), osp.isdir('devlist.txt')
(True, False)
>>> osp.getctime('devlist.txt'), osp.getmtime('devlist.txt')
(1455880136.8817973, 1455879611.5920193)
>>> osp.getsize('devlist.txt'), osp.getsize('C:\\Users')
(12019, 4096)
```

funkcija	**opis**
`os.path.abspath(p)`	vraća apsolutnu putanju za p
`os.path.isabs(p)`	ispituje da li je p apsolutna
`os.path.relpath(p)`	vraća relativnu putanju za p
`os.path.dirname(p)`	vraća putanju direktorijuma za p
`os.path.basename(p)`	vraća ime datoteke za p
`os.path.join(f1,...,fn)`	pravi putanju od tekstualnih fragmenata `f1`,...,`fn`
`os.path.getsize(p)`	vraća veličinu datoteke (direktorijuma) u bajtima
`os.path.getctime(p)`	vraća sistemsko vreme nastanka p
`os.path.getmtime(p)`	vraća sistemsko vreme poslednjeg ažuriranja p
`os.path.getatime(p)`	vraća sistemsko vreme poslednjeg pristupanja p
`os.path.exists(p)`	ispituje da li p postoji u sistemu datoteka
`os.path.isfile(p)`	ispituje da li je p datoteka
`os.path.isdir(p)`	ispituje da li je p direktorijum

Tabela 9.2: Česte operacije sa putanjama. Putanja p može biti kako apsolutna, tako i relativna (u odnosu na tekući radni direktorijum).

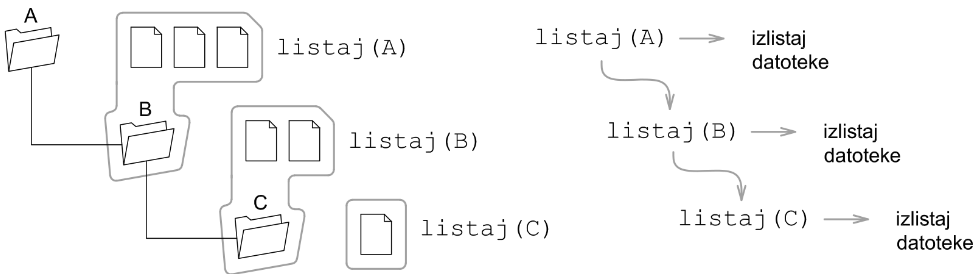
Prilikom korišćenja funkcije `join()`, argumenti su *tekstualni fragmenti* putanje odvojeni zapetama. Ova funkcija koristi sistemski separator putanje koji se razlikuje između operativnih sistema (na Windows-u, "\"). Putanja se može sastaviti iz fragmenata korišćenjem metode `join()` iz klase `str` (videti glavu 5.1), ali bi se tada za spajajuću sekvencu morao navesti sistemski zavisan separator, što bi *nepovoljno* uticalo na *prenosivost* programa na druge operativne sisteme.

Prenosivost (ili portabilnost) programa odnosi se na mogućnost da program, bez izmene izvornog koda, može da radi pod različitim operativnim sistemima. Kad god je to moguće, program treba pisati tako da bude prenosiv!

Vreme koje vraćaju funkcije `getatime()`, `getctime()` i `getmtime()` odnosi se na broj proteklih sekundi u odnosu na epohu (glava 6.1.1). Iz primera je interesantno uočiti da *pojedinačni* direktorijumi, poput datoteka, imaju odgovarajuću veličinu u bajtovima. Ona se ne odnosi na datoteke koje se nalaze unutar direktorijuma ili datoteke koje se nalaze niže u hijerarhji sistema datoteka, već na informacije poput vremena kreiranja, prava pristupa i slično.

Problem 9.1 — Stablo direktorijuma. Kreirati funkciju koja prikazuje informacije o datotekama koje se nalaze u i ispod navedenog direktorijuma u hijerarhiji sistema datoteka. Prikazati informacije o veličini i vremenu kreiranja svake datoteke. ■

Problem se rešava obaveznim *posećivanjem* svih datoteka iz navedenog direktorijuma, pri čemu se, uz pomoć njihovih apsolutnih putanja i uz upotrebu odgovarajućih funkcija iz tabele 9.2, ispisuju tražene informacije. Međutim, kako svaki direktorijum može da sadrži više poddirektorijuma, postupak treba *rekurzivno* ponoviti za svaki poddirektorijum (slika 9.2).



Slika 9.2: Rekurzivno posećivanje datoteka u stablu direktorijuma. Funkcija `listaj(A)` posećuje sve datoteke u direktorijumu `A`. Kada se naiđe na direktorijum `B`, čini se rekurzivni poziv `listaj(B)`. Prilikom posete datoteka u `B`, nailazi se na direktorijum `C` pa se poziva `listaj(C)`. Kako `C` ne sadrži poddirektorijume, poslednji poziv predstavlja bazni slučaj rekurzije.

```
1  # listanje datoteka + veličine i vreme kreiranja
2  from os import listdir
3  import os.path as osp
4  from math import ceil
5  from time import asctime, localtime
6
7  def listaj(ul_dir, dubina):
8
9      for p in listdir(ul_dir):
10         p = osp.join(ul_dir, p)
11         if osp.isdir(p):
```

```
12                print(f"+{dubina * '--'} {p}")
13                listaj(p, dubina + 1)
14            elif osp.isfile(p):
15                print('+{} {:20}\t{:6}K\t{}'.format(
16                       dubina * '--',
17                       osp.basename(p),
18                       ceil(osp.getsize(p)/1024),
19                       asctime(localtime(osp.getctime(p)))))
20  # test program
21  putanja = osp.abspath(input('unesite putanju direktorijuma: '))
22  if osp.exists(putanja) and osp.isdir(putanja):
23      listaj(putanja, 0)
24  else:
25      print('pogrešan unos!')
```

Ulazni parametar funkcije `listaj()` predstavlja *apsolutnu* putanju za početni direktorijum (`ul_dir`). Posećivanje datoteka u tekućem pozivu ostvareno je u petlji (r9-19). Promenljiva `p` ukazuje na putanje iz prosleđenog direktorijuma. Putanje se dobijaju pozivom funkcije `listdir()` iz modula `os` (r9). Kako su dobijene putanje relativne u odnosu na prosleđeni direktorijum, potrebno je načiniti ih apsolutnim (r10). Za svaku putanju ispituje se da li označava direktorijum ili datoteku (r11, 14). Ako se radi o direktorijumu (r11), ispisuje se potrebna informacija (r12) i čini sledeći rekurzivni poziv (r13). Informacije o datotekama ispisuju se upotrebom odgovarajućih funkcija (r15-19). Ovom prilikom, osim funkcija iz tabele 9.2, korišćene su funkcije `math.ceil()` (zaokruživanje na prvi veći ceo broj), `time.asctime()` (tekstualna reprezentacija vremenskog objekta) i `time.localtime()` (vraća vremenski objekat za navedeno sistemsko vreme).

Da bi ispis bio pregledan (u obliku stabla), koristi se ulazni parametar `dubina` koji označava *dubinu* prosleđenog direktorijuma u stablu čiji je koren početno zadati dirketorijum. Sve datoteke iz početnog direktorijuma imaju dubinu jednaku nula (r23), u poddirektorijumima jedan i tako dalje. Zbog toga se, prilikom svakog poziva u r13, ovaj parametar uvećava za jedan. Na osnovu dubine tekućeg poziva, odgovarajući ispis se pomera za željeni broj mesta udesno (r12, 15), a u taj prostor se ubacuje prigodna ispuna koja asocira na stablo (`dubina * '--'`). Budući da u r15 format string ne bi stao u jedan programski red, iskorišćena je tekstualna metoda `format()` koja proizvodi isti efekat, a može se smestiti u više redova:

```
unesite putanju direktorijuma: c:\papers\2017
+ c:\papers\2017\jcce
+-- jcce.7z               15752K Wed Jun 21 19:45:26 2017
+-- oldPaper.docx          3233K Wed Jun 21 19:45:26 2017
```

```
+-- paper.rar            21028K Wed Jun 21 19:45:26 2017
+-- response.doc            75K Wed Jun 21 19:45:26 2017
+-- revised.docx            96K Wed Jun 21 19:45:26 2017
+-- revised2.docx           95K Wed Jun 21 19:45:26 2017
+-- c:\papers\2017\jcce\slikePdf
+---- Fig. 1.pdf              30K Wed Jun 21 19:45:26 2017
+---- Fig. 10.pdf            135K Wed Jun 21 19:45:26 2017
+---- Fig. 11.pdf            326K Wed Jun 21 19:45:26 2017
+---- Fig. 2.pdf              48K Wed Jun 21 19:45:26 2017
+---- Fig. 3.pdf              49K Wed Jun 21 19:45:26 2017
+---- Fig. 4.pdf              17K Wed Jun 21 19:45:26 2017
+---- Fig. 5.pdf              53K Wed Jun 21 19:45:26 2017
+---- Fig. 6.pdf            2235K Wed Jun 21 19:45:26 2017
+---- Fig. 7.pdf              42K Wed Jun 21 19:45:26 2017
+---- Fig. 8.pdf           15114K Wed Jun 21 19:45:26 2017
+---- Fig. 9.pdf             185K Wed Jun 21 19:45:27 2017
+ test.txt                 1K Fri Jun 23 13:28:44 2017
```

Budući da predstavlja *detalj realizacije* za kontrolu izgleda ispisa, drugi parametar funkcije `listaj()` "kvari" njen prirodan potpis. On se može izbeći uz pomoć druge, *unutrašnje* funkcije na sledeći način:

```
1  # listanje datoteka + veličine i vreme kreiranja
2  from os import listdir
3  import os.path as osp
4  from math import ceil
5  from time import asctime, localtime
6
7  def listaj(ul_dir):
8      def listaj_rek(ul_dir, dubina):
9          for p in listdir(ul_dir):
10             p = osp.join(ul_dir, p)
11             if osp.isdir(p):
12                 print(f"+{dubina * '--'} {p}")
13                 listaj_rek(p, dubina + 1)
14             elif osp.isfile(p):
15                 print('+{} {:20}\t{:6}K\t{}'.format(
16                     dubina * '--',
17                     osp.basename(p),
18                     ceil(osp.getsize(p)/1024),
19                     asctime(localtime(osp.getctime(p)))))
20
21     listaj_rek(ul_dir, 0)
```

Funkcija `listaj_rek()` mogla je biti smeštena i *van* funkcije `listaj()`, ali bi tada, posle uvođenja programskog modula, bila *dostupna* u drugim programima. Sledi konačno rešenje problema koje, po ispisu stabla, prikazuje *zbirnu* veličinu svih datoteka u stablu početnog direktorijuma:

Program 9.1 — Stablo direktorijuma.

```
1  # listanje datoteka + veličine i vreme kreiranja
2  from os import listdir
3  import os.path as osp
4  from math import ceil
5  from time import asctime, localtime
6
7  def listaj(ul_dir):
8
9      def listaj_rek(ul_dir, dubina):
10         total = 0
11         for p in listdir(ul_dir):
12             p = osp.join(ul_dir, p)
13             if osp.isdir(p):
14                 print(f"+{dubina * '--'} {p}")
15                 total += listaj_rek(p, dubina + 1)
16             elif osp.isfile(p):
17                 veličina = ceil(osp.getsize(p)/1024)
18                 total += veličina
19                 print('+{} {:20}\t{:6}K\t{}'.format(
20                     dubina * '--',
21                     osp.basename(p),
22                     veličina,
23                     asctime(localtime(osp.getctime(p)))))
24
25         return total
26
27     return listaj_rek(ul_dir, 0)
28
29 # test program
30 putanja = osp.abspath(input('unesite putanju direktorijuma: '))
31 if osp.exists(putanja) and osp.isdir(putanja):
32     print(f'Ukupno {listaj(putanja)}K')
33 else:
34     print('pogrešan unos!')
```

Novi izgled ispisa za isti primer (samo poslednja dva reda) dat je sa:

```
+ test.txt                  1K Fri Jun 23 13:28:44 2017
Ukupno 58514K
```

9.3 Ulazno-izlazne operacije sa tekstualnim datotekama

Funkcije iz modula os pristupaju datoteci preko apsolutne ili relativne putanje koja, po svom tipu, predstavlja tekstualni objekat. Međutim, kada se datoteci pristupa kako bi se iz nje čitalo ili u nju upisivalo, *prvo* treba pribaviti objekat kojim se datoteka predstavlja. Ovo se čini upotrebom funkcije `open()` koja "otvara" datoteku navedenu putanjom i vraća objekat za pristup. Tekstualne datoteke, o kojima će nadalje biti reči, mogu se otvoriti za *čitanje*, *pisanje* ili *pisanje/dodavanje*.

9.3.1 Čitanje iz tekstualne datoteke

Prilikom čitanja sadržaja iz datoteke, ona se otvara na sledeći način:

```
>>> d = open('c:\\papers\\pesma.txt')  # otvoreno za čitanje
>>> type(d)
<class '_io.TextIOWrapper'>
```

Promenljiva `d` ukazuje na objekat klase `_io.TextIOWrapper` kojim se predstavlja datoteka sa apsolutnom putanjom `'c:\\papers\\pesma.txt'` – evo još jednog primera kako se objektima apstrahuju ne samo jednostavni podaci poput brojeva, već i složeni entiteti kao što su datoteke na disku (apstrahovanje, glava 1.1.1). Ako se pri otvaranju navede samo putanja, onda se sadržaj datoteke može *samo* čitati:

```
>>> d.read()  # čita ceo sadržaj
'Ja znam sva tvoja lica, svako šta hoce, šta nosi,\ngledao sam
 sve tvoje oci, razumem šta kažu, šta kriju.\nJa mislim tvoju
 misao za celom ti u kosi,\nja znam tvoja usta šta ljube, šta piju.'
>>>
>>> d.read()  # pročitana cela datoteka, vraća prazan tekst
''
>>> d.close() # zatvara datoteku
>>> d.read()  # posle zatvaranja, nema čitanja!
Traceback (most recent call last):
  File "<pyshell#102>", line 1, in <module>
    d.read()
ValueError: I/O operation on closed file.
```

Metoda `read()`, *bez* navedenog argumenta, čita *celokupan* sadržaj datoteke i vraća

ga kao tekstualnu sekvencu. Treba se podsetiti da je tekstualna datoteka organizovana kao niz redova teksta koji su odvojeni separatorom za novi red. Iz gornjeg primera uočava se da je separator kontrolna sekvenca `\n`, kao i da je nakon poslednjeg reda ovaj separator *izostavljen*. Svaki sledeći poziv metode `read()` vraća praznu sekvencu. Posle završetka rada, datoteku *obavezno* treba zatvoriti upotrebom metode `close()`! Pokušaj ponovnog čitanja nad zatvorenom datotekom uvek dovodi do greške.

Ulazno-izlazne operacije sa datotekama (UI) zasnivaju se na *sistemskim* pozivima koje interpreter prosleđuje operativnom sistemu (OS) računara. Tom prilikom, OS menja stanje internih struktura u radnoj memoriji, pristupa fizičkom uređaju eksterne memorije i sl. Ako se datoteka otvorena za čitanje eksplicitno ne zatvori, OS *ne može* da oslobodi memorijske resurse koje je odvojio za tu UI operaciju. Pored toga, na računaru se izvršava više *istovremenih* procesa koji mogu da dele iste resurse. Ako se datoteka otvorena za *upisivanje* ne zatvori, drugi procesi *neće* moći da menjaju njen sadržaj sve dok interpreter ne prekine sa radom.

Celovito čitanje tekstualne datoteke, svi redovi odjednom, može se primeniti *samo* ako je kapacitet slobodne memorije *dovoljno* veliki da primi celokupan sadržaj. Zato se obrada podataka najčešče obavlja u režimu čitanja *red po red*. Za tu svrhu se koristi petlja `for` koja ima sledeći oblik:

```
>>> d = open('c:\\papers\\pesma.txt')
>>> for red in d:
        print(red)

Ja znam sva tvoja lica, svako šta hoce, šta nosi,

gledao sam sve tvoje oci, razumem šta kažu, šta kriju.

Ja mislim tvoju misao za celom ti u kosi,

ja znam tvoja usta šta ljube, šta piju.
>>> d.close()
```

U gornjem primeru, u svakoj iteraciji petlje, promenljiva `red` ukazuje na pojedinačne redove iz tekstualne datoteke. Primećuje se da su redovi ispisani sa proredom jer se, prilikom učitavanja, na kraju svake linije nalazi kontrolna sekvenca za novi red `\n`. Osim toga, prikazani tekst odstupa od originalnih stihova jer fale pojedini karakteri (č i ć). Ovo se dešava jer je datoteka `pesma.txt` sačuvana u *podrazumevanom* sistemskom formatu koji ne podržava sva naša slova.[5] U glavi 5.1 bilo je pomena da se, za potrebe čuvanja slova i simbola iz različitih svetskih jezika, može koristiti standard `Unicode` sa kodiranjem UTF-8. Ako su originalni stihovi smešteni u datoteci

[5] Na autorovom računaru, sa sistemom Windows 10, sistemsko kodiranje je `Windows cp-1252`.

zapisanoj u formatu UTF-8 (kao što je to u ovom slučaju), onda se prilikom otvaranja mora navesti odgovarajući opcioni parametar:

```
>>> d = open('c:\\papers\\pesma_sr.txt', encoding='utf-8')
>>> for red in d:
        print(red.strip()) # uklanja \n s kraja reda

Ja znam sva tvoja lica, svako šta hoće, šta nosi,
gledao sam sve tvoje oči, razumem šta kažu, šta kriju.
Ja mislim tvoju misao za čelom ti u kosi,
ja znam tvoja usta šta ljube, šta piju.
>>> d.close()
>>> d = open('c:\\papers\\pesma.txt')
>>> d.read(7) # čita prvih 7 karaktera
'Ja znam'
>>> d.read(4) # pa sledeća 4
' sva'
```

Na kraju prethodnog primera ilustrovana je i mogućnost da se iz tekstualne datoteke pročita traženi broj karaktera (`d.read(7)`). Sledeće čitanje nastavlja od pozicije koja je definisana prethodnim (`d.read(4)`). Ako bi se prilikom otvaranja datoteke navelo pogrešno kodiranje (različito od onog pod kojim je datoteka zapisana), interpreter bi prijavio grešku pri prvom čitanju:

```
>>> d = open('c:\\papers\\pesma.txt', encoding='utf-8')
>>> d.read()
Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    d.read()
  File "C:\Python35\lib\codecs.py", line 321, in decode
    (result, consumed) = self._buffer_decode(data, self.errors, final)
UnicodeDecodeError: 'utf-8' codec can't decode byte 0x9a
in position 30: invalid start byte
```

9.3.2 Upisivanje u tekstualnu datoteku

Prilikom upisivanja novog sadržaja postoje dve mogućnosti: tekst za upisivanje se zapisuje *preko postojećeg* sadržaja koji se *gubi* ili se *dodaje* na kraj već postojećeg:

```
>>> # upisivanje, kreira novu datoteku
>>> d = open('c:\\papers\\nova.txt', 'w')
>>> d.write('Prvi red\nDrugi red')
18
>>> d.close()
```

```
>>> # dodavanje
>>> d = open('c:\\papers\\nova.txt', 'a')
>>> d.write('\nTreci red')
10
>>> d.close()
>>> # čitanje
>>> d = open('c:\\papers\\nova.txt') # opciono 'r' se podrazumeva!
>>> print(d.read())
Prvi red
Drugi red
Treci red
>>> d.close()
>>> # upisivanje, briše stari sadržaj
>>> d = open('c:\\papers\\nova.txt', 'w')
>>> d.write('Cetvrti red\n')
12
>>> d.close()
>>> # čitanje
>>> d = open('c:\\papers\\nova.txt')
>>> d.read()
'Cetvrti red\n'
>>> d.close()
```

Funkcija `open()` prima opcioni parametar kojim se navodi *način* na koji se datoteka otvara: `'w'` podrazumeva da je datoteka otvorena za upis. Ako datoteka ne postoji, na navedenoj putanji (`nova.txt`) formira se *nova*. Metoda `write()` upisuje navedeni tekst u novu datoteku i tom prilikom vraća broj uspešno upisanih karaktera. Uočiti da `write()` ne dodaje automatski separator za novi red, već da se to čini njegovim navođenjem u okviru tekstualne sekvence. Ako se umesto `'w'` stavi `'a'`, onda se novi tekst *dodaje na kraj* postojeće datoteke. Međutim, kada datoteka ne bi postojala, onda bi se kreirala nova. Ako se datoteka otvori uz opciju `'w'`, a da pri tome već postoji na disku, onda se njen sadržaj *zamenjuje* novim što se vidi iz primera.

Problem 9.2 — Kopiranje. Kreirati funkciju koja kopira sadržaj tekstualne datoteke uz adekvatnu obradu mogućih izuzetaka. ■

Funkcija `kopiraj()` osmišljena je da primi dva obavezna i jedan opcioni ulazni parametar: `ulaz` i `izlaz` predstavljaju putanje za izvornu i datoteku kopiju, a parametar `kodiranje` navodi ulazni (i izlazni) sistem kodiranja teksta. Ako se drugačije ne navede, podrazumevaće se standard `Unicode` sa tablicom preslikavanja UTF-8:

```
1 # kopiranje
2 def kopiraj(ulaz, izlaz, kodiranje='utf-8'):
```

```
3
4       try:
5           udat = open(ulaz, encoding=kodiranje)
6       except:
7           print('Pogrešna putanja!')
8           return
9
10      try:
11          idat = open(izlaz, 'w', encoding=kodiranje)
12      except:
13          print('Pogrešna putanja!')
14          return
15
16      try:
17          for red in udat:
18              idat.write(red)
19          print(f'{ulaz} -> {izlaz}  OK!')
20
21      except UnicodeDecodeError:
22          print(kodiranje, 'pogrešno kodiranje!')
23      except:
24          print('Greška pri kopiranju!')
25      finally:
26          udat.close()
27          idat.close()
28
29  # test program
30  kopiraj('c:\\papers\\test.txt','c:\\papers\\test2.txt')
31  kopiraj('c:\\papers\\test.txt','c:\\papers\\test2.txt','cp1252')
```

U prvom delu funkcije otvaraju se potrebne datoteke i pribavljaju odgovarajući objekti za pristup (r5, 11). Funkcije za otvaranje smeštene su u kontrolne strukture `try-except` (r4-8, r10-14), što omogućava da se, u slučaju neispravnih putanja, ispiše odgovarajuća poruka (r7, 13) i funkcija završi sa radom (r8, 14).

Kopiranje se obavlja u petlji po sistemu red po red (r17-18). Prilikom upisa tekućeg reda u datoteku kopiju (r18), nema potrebe za dodvanjem separatora novog reda jer se on, osim poslednjeg, *već nalazi* na kraju svakog učitanog reda. Petlja je smeštena u strukturu `try-except-finally` kako bi se mogli obraditi eventualni izuzeci. Posebno je prepoznata greška tipa `UnicodeDecodeError` (r21) koja se javlja kada se navedeni i primenjeni sistem kodiranja *razlikuju*. Po završetku postupka, bez obzira na izuzetke, datoteke se zatvaraju u bloku naredbe `finally` (r26-27). Sledi prikaz rada programa za testiranje:

```
utf-8 pogrešno kodiranje!
c:\papers\test.txt -> c:\papers\test2.txt  OK!
```

Datoteka `c:\papers\test.txt` je zapisana u podrazumevanom sistemskom formatu (*Windows-1252*, oznaka `cp1252`). U prvom primeru navedeno je neadekvatno kodiranje (UTF-8, r30). Zato se, pri pokušaju čitanja prvog reda u r17, pojavljuje greška koja se obrađuje u bloku naredbe `except` (r21-22). Potom se izvršava i blok naredbe `finally` (r26-27). Operacija kopiranja uspeva u drugom pokušaju pošto se zada ispravno kodiranje (r31). Primetiti da rešenje podrazumeva *isto* ulazno i izlazno kodiranje. U opštem slučaju, može se uvesti još jedan ulazni parametar koji bi označavao dodatno kodiranje (umesto `kodiranje`, `ul_kod` i `iz_kod`).

Prikazano rešenje ilustruje negativnu stranu mehanizma za obradu izuzetaka koja se ogleda u povećanom broju naredbi `try` i `except`. Početnici bi mogli doći u iskušenje da *izbace* “višak” naredbi na sledeći način:

```
1  # kopiranje (neispravna varijanta)
2  def kopiraj(ulaz, izlaz, kodiranje='utf-8'):
3
4      try:
5          udat = open(ulaz, encoding=kodiranje)
6          idat = open(izlaz, 'w', encoding=kodiranje)
7          for red in udat:
8              idat.write(red)
9          print(f'{ulaz} -> {izlaz}  OK!')
10
11     except FileNotFoundError:
12         print('Loša putanja!')
13     except UnicodeDecodeError:
14         print(kodiranje, 'pogrešno kodiranje!')
15     except:
16         print('Greška pri kopiranju!')
17     finally:
18         udat.close()
19         idat.close()
20 # test program
21 kopiraj('c:\\papers\\nema.txt','c:\\papers\\test2.txt','cp1252')
```

Problem u ovoj realizaciji nastaje u r18, u bloku naredbe `finally`, kada se referencira objekat koji *nije* uspeo da se kreira u r5:

```
Loša putanja!
Traceback (most recent call last):
  File "C:/programi/p9_2b.py", line 22, in <module>
    kopiraj('c:\\papers\\nema.txt','c:\\papers\\test2.txt','cp1252')
  File "C:/programi/p9_2b.py", line 18, in kopiraj
    udat.close()
UnboundLocalError: local variable 'udat' referenced before assignment
```

Kako se u praksi datoteke vrlo često koriste, Pajton nudi mogućnost da se obavezni kod skrati primenom naredbe `with`:

Program 9.2 — Kopiranje.

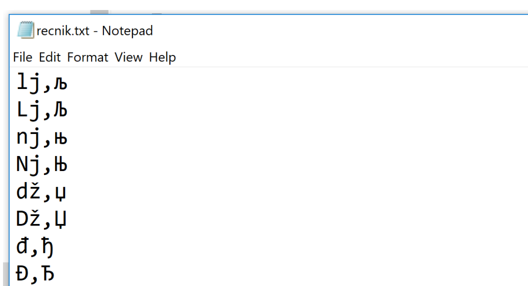
```
1  # kopiranje
2  def kopiraj(ulaz, izlaz, kodiranje='utf-8'):
3
4      try:
5          with open(ulaz, encoding=kodiranje) as udat, \
6                open(izlaz, 'w', encoding=kodiranje) as idat:
7
8              for red in udat:
9                  idat.write(red)
10             print(f'{ulaz} -> {izlaz}  OK!')
11
12     except UnicodeDecodeError:
13         print(kodiranje, 'pogrešno kodiranje!')
14     except FileNotFoundError:
15         print('Pogrešna putanja!')
16     except:
17         print('Greška pri kopiranju!')
18 # test
19 kopiraj('c:\\papers\\test.txt','c:\\papers\\test2.txt')
20 kopiraj('c:\\papers\\test.txt','c:\\papers\\test2.txt','cp1252')
```

Naredba `with` omogućava da se u njenom zaglavlju otvori više datoteka odjednom, pri čemu se svakoj datoteci pridružuje objekat odgovarajućeg imena (r5-6). Zapaziti upotrebu simbola “\”, koji omogućava da se programski red *nastavi* u sledećem redu izvornog koda. Osim toga, naredba `with` obezbeđuje da se pri napuštanju njenog bloka, iz bilo kog razloga (regularan kraj ili izuzetak), sve datoteke *automatski zatvore*! Ako pri izvršavanju naredbi iz (r5-10) dođe do greške, otvorene datoteke iz zaglavlja naredbe

`with` se zatvaraju, a programski tok se prebacuje u odgovarajući blok naredbe `except`.

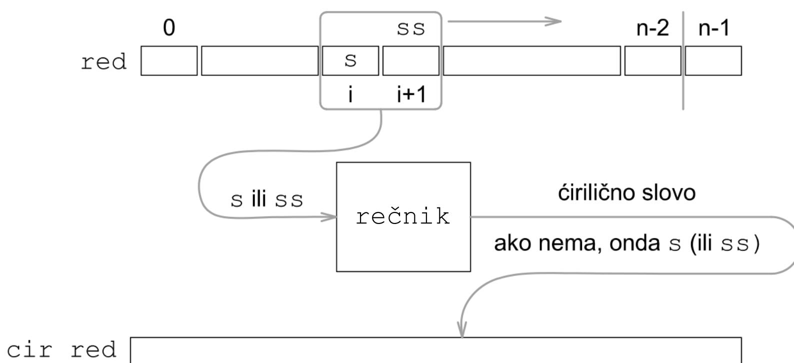
Problem 9.3 — Ćirilica. Kreirati funkciju koja kopira sadržaj tekstualne datoteke uz prevođenje latiničnog u ćirilični tekst. ■

Za potrebe preslovljavanja koristiće se rečnik koji sadrži prevod latiničnih u ćirilična slova (videti problem 5.7). Za razliku od prevođenja u obrnutom smeru, zbog postojanja dvoslovnih grupa (`'lj'`,`'nj'`,`'dž'`), preslikavanje *nije* uvek jednoznačno. Rečnik će, radi vežbe, biti smešten u posebnoj datoteci kodiranoj u formatu UTF-8. U svakom redu datoteke smešteno je po jedno preslikavanje oblika x,y. Ovde x predstavlja latiničnu grupu od jednog ili dva slova, a y odgovarajuće ćirilično slovo (slika 9.3).



Slika 9.3: Rečnik preslikavanja: datoteka ima 60 redova, po dva za svako slovo (malo i veliko).

Problem se rešava dekompozicijom na jednostavnije celine. Funkcija `preslovi()` *učitava* red po red iz izvorne datoteke, vrši *preslovljavanje* svakog reda i *upisuje* preslovljeni red u odredišnu datoteku. U tu svrhu prosleđuje joj se rečnik koji definiše potrebna preslikavanja. Rečnik se učitava u funkciji `učitaj_rečnik()` koja vraća objekat tipa `dict`. Prevođenje pojedinačnog reda obavlja se u pomoćnoj funkciji `preslovi_red()`. Logika obrade prikazana je na slici 9.4.



Slika 9.4: Postupak preslovljavanja: prolazak kroz sekvencu `red`, sve do *pretposlednjeg* slova, uz pomoć dve promenljive, `s` (tekuće slovo) i `ss` (tekuća dva slova). Prevod iz rečnika traži se prvo za tekuća dva pa, ako ga nema, za tekuće slovo. Prevodi se dodaju u listu `cir_red`.

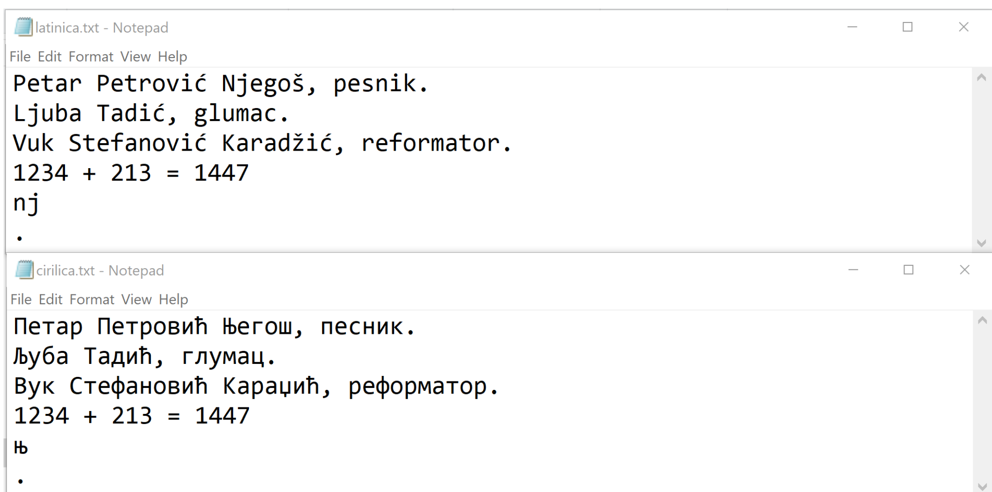
Program 9.3 — Ćirilica.

```
# preslovljavanje
def preslovi(lat, cir, rečnik):

    try:
        with open(lat, encoding='utf-8') as dlat, \
             open(cir, 'w', encoding='utf-8') as dcir:

            for red in dlat:
                dcir.write(preslovi_red(red, rečnik))
    except:
        print('Greška pri prevođenju')

def preslovi_red(red, rečnik):

    lj_nj_dz = {'lj', 'Lj', 'nj', 'Nj', 'dž', 'Dž'}
    cir_red, i, n = [], 0, len(red)

    while i < n-1:
        s, ss = red[i], red[i:i+2]
        if ss in lj_nj_dz:
            cir_red.append(rečnik.get(ss, ss))
            i += 2
        else:
            cir_red.append(rečnik.get(s, s))
            i += 1
    if i == n-1:
        cir_red.append(rečnik.get(red[i], red[i]))

    return ''.join(cir_red)

def učitaj_rečnik(p):

    r = {}
    try:
        with open(p, encoding='utf-8') as rečnik:
            for red in rečnik:
                par = red.strip().split(',')
                r[par[0]] = par[1]
    except:
```

```
40                print('Greška pri učitavanju rečnika!')
41                r = {}
42        return r
43
44    # test program
45    r = učitaj_rečnik('c:\\papers\\recnik.txt')
46    if r:   # može i if len(r) > 0
47        preslovi('c:\\papers\\latinica.txt',
48                 'c:\\papers\\cirilica.txt', r)
```

Prevođenje se obavlja u petlji `while` prema logici sa slike 9.4 (r18-25). Skup `lj_nj_dz` je upotrebljen za detektovanje dvoslovnih latiničnih grupa koje se prevode u jedno ćirilično slovo (r15, 20). U slučaju postojanja prevoda za dvoslovnu grupu, promenljiva koja ukazuje na tekuće slovo uvećava se za dva (r22), inače za jedan (r25). Ako odgovarajući prevod ne postoji u rečniku, metoda `get()` vraća tekuća dva, odnosno tekuće slovo (r21, 24) – dešava se za simbole koji nisu slova. U slučaju da poslednje slovo nije deo dvočlane grupe sa postojećim prevodom, prevodi se posebno na kraju postupka (r26-27). Preslovljeni red formira se tako što se lista, sa pojedinačnim ćiriličnim slovima, spoji u tekstualnu sekvencu pomoću metode `join()` (r29).

Prilikom čitanja iz rečničke datoteke koriste se tekstualne metode `strip()` i `split()`. Ove metode upotrebljene su da uklone separator novog reda s kraja učitane sekvence, odnosno da razdvoje latiničnu grupu od odgovarajućeg prevoda (r37). U slučaju neregularnosti, vraća se prazan rečnik (r41). Efekti rada programa ilustrovani su na slici 9.5.



Slika 9.5: Efekti preslovljavanja: izgled izvorne i odredišne datoteke.

9.4 Tekstualne datoteke u CSV formatu

Podaci se u inženjerskim ili poslovnim primenama *često* zapisuju u obliku *tabela*. Skup istovrsnih podataka može se okarakterisati odgovarajućim osobinama koje određuju svaku instancu iz skupa. Na primer, skup studenata nekog fakulteta mogao bi se opisati sledećim osobinama: ime, prezime, broj indeksa, smer, godina studiranja i prosek. Svaki student iz skupa može se predstaviti kao *torka* konkretnih vrednosti koje odgovaraju pojedinačnim osobinama (npr. *Petar*, *Kralj*, *1/17*, *MTI*, *3*, *7.55*). Osobine su opisane odgovarajućim tipovima. Tip definiše domen mogućih vrednosti, kao i operacije koje se mogu obavljati nad njima. Ovakvi podaci najčešće se predstavljaju *tabelarno*, pri čemu *redovi* odgovaraju pojedinačnim instancama (na primer, studentima), a *kolone* se odnose na vrednosti pojedinih osobina (na primer, ime).

Za skladištenje i obradu tabelarnih podataka koriste se programi poput Microsoft Excel-a, kao i relacione baze podataka. Često postoji potreba da se ovakvi podaci prebacuju iz jednog programskog sistema u drugi ili ih treba obraditi u posebnom programu, u skladu sa konkretnim problemom. U tu svrhu, oni se iz matičnih sistema mogu *izvesti* (ili *eksportovati*) u tekstualnu datoteku u CSV[6] formatu. Prema ovom formatu, svaki red izvorne tabele zapisuje se u odgovarajući red tekstualne datoteke pri čemu se, u tekstualnom zapisu, konkretne vrednosti osobina razdvajaju *zapetom*.[7] Prvi red CSV datoteke obično, ali ne i nužno, predstavlja *zaglavlje*. Zaglavlje sadrži imena osobina razdvojena zapetom ili naznačenim separatorom - slika 9.6.

ime	prezime	broj indeksa	godina studiranja	prosek
Stevan	Stevanović	1/15	2	8.5
Milan	Milić	2/15	2	7.5
Rade	Radović	3/15	2	6
Ana	Aničić	4/15	2	6
Višnja	Višnjić	5/15	1	6.5

studenti.csv - Notepad

File Edit Format View Help

```
ime,prezime,broj indeksa,godina studiranja,prosek
Stevan,Stevanović,1/15,2,8.5
Milan,Milić,2/15,2,7.5
Rade,Radović,3/15,2,6
Ana,Aničić,4/15,2,6
Višnja,Višnjić,5/15,1,6.5
```

Slika 9.6: Izgled datoteke u CSV formatu po eksportovanju iz Excel-a. U pozadini je prikazana izvorna Excel tabela. Tekstualna polja mogu se, prilikom eksportovanja, staviti pod navodnike pa se na taj način između navodnika može naći i simbol za separator (na primer, zapeta).

[6] Engl. *Comma Separated Values*.
[7] Separator može biti proizvoljan eksplicitno navedeni simbol. Najčešće je to zapeta ili tabulator.

Kako se datoteke u CSV formatu vrlo često obrađuju u praksi, u Pajtonu postoji modul sa funkcionalnostima za izdvajanje, odnosno upisivanje tabelarnih podataka u pomenutom formatu – modul `csv`.

Tekstualni podaci u CSV formatu *ne sadrže* informaciju o tipovima osobina. Program je obavezan da, u skladu sa specifikacijom, pravilno interpretira i obradi vrednosti iz tekstualnog reda. Na primer, da vrednost `'01-01-2000'` u koloni A tretira kao datum, a istu vrednost u koloni B kao tekst.

9.4.1 Parsiranje CSV datoteke

U računarskim naukama, *parsiranje* podrazumeva proces podele podataka na *manje celine* radi lakše interpretacije i korišćenja. Podela se obavlja na osnovu *predefinisanih pravila* koja zavise od konteksta primene. U glavi 5.1.4, pomenuti su programi koji vrše parsiranje tekstualnih rečenica iz prirodnog jezika. Oni razbijaju rečenice na pojedinačne reči. Parsiranje se najčešće i primenjuje nad tekstualnim podacima kako bi program bolje "shvatio" značenje navedenog teksta. Na primer, rečenice se mogu rastaviti na reči koje su, po odgovarajućoj interpretaciji, klasifikovne kao imenice, pridevi, glagoli i slično. Kao još jedan primer, navodi se parsiranje programskog reda od strane interpretera u Pajtonu: tekst reda treba podeliti na celine koje se interpretiraju kao brojevi, ključne reči, separatori i slično. Prilikom parsiranja, interpreter sprovodi i *sintaksnu analizu* čiji je cilj da utvrdi da li je red sintaksno ispravan – na primer, da li se potpis funkcije završava dvotačkom.

Proces izdvajanja tabelarnih podataka iz CSV datoteke naziva se još i *parsiranje CSV datoteke*. Sledi primer interaktivne sesije u IDLE-u koja koristi modul `csv` da putem parsiranja formira kolekciju tipa liste:

```
>>> from csv import reader
>>> with open('c:\\papers\\studenti.csv', encoding='utf-8') as d:
        tabela = list(reader(d))
        tabela2 = list(reader(d))
        print(tabela)
        print(tabela2)

[['ime', 'prezime', 'broj indeksa', 'godina studiranja', 'prosek'],
 ['Stevan', 'Stevanović', '1/15', '2', '8.5'],
 ['Milan', 'Milić', '2/15', '2', '7.5'],
 ['Rade', 'Radović', '3/15', '2', '6'],
 ['Ana', 'Aničić', '4/15', '2', '6'],
 ['Višnja', 'Višnjić', '5/15', '1', '6.5']]
[]
>>> tabela[0]  # zaglavlje
['ime', 'prezime', 'broj indeksa', 'godina studiranja', 'prosek']
```

```
>>> tabela[1][2]
'1/15'
```

Funkcija `csv.reader()` vraća objekat *čitač*[8] koji je po svojoj prirodi iterator, a pomoću koga se parsira navedena CSV datoteka `d` – o iteratorima je bilo reči u glavi 5.3.1 kada se razmatrao modul `itertools`. Navođenjem čitača, kao argumenta konstruktora liste, vrši se čitanje podataka iz datoteke i njihovo izdvajanje u listu: `tabela = list(csv.reader(d))`. Elementi ove liste odnose se na redove iz tabele. Svaki red i sam je predstavljen listom pa se pojedinačnim ćelijama pristupa putem dvostrukog indeksiranja – na primer, `tabela[1][2]`. Prvi indeks označava redni broj reda, a drugi, redni broj kolone u tabeli.

Po završenom parsiranju, CSV datoteka obavezno treba da se zatvori. U gornjem primeru to se postiže obradom u bloku naredbe `with`. Parsiranje se, dok je odgovarajuća datoteka otvorena, može obaviti *samo* jednom. Ako se pak ponovi (`tabela2`), dobija se prazna lista. U slučaju velike datoteke, koja prilikom parsiranja ne može da stane u radnu memoriju, učitavanje se može obaviti red po red:

```
>>> from csv import reader
>>> with open('c:\\papers\\studenti.csv', encoding='utf-8') as d:
        tabela = reader(d)
        for red in tabela:
            print(red)

['ime', 'prezime', 'broj indeksa', 'godina studiranja', 'prosek']
['Stevan', 'Stevanović', '1/15', '2', '8.5']
['Milan', 'Milić', '2/15', '2', '7.5']
['Rade', 'Radović', '3/15', '2', '6']
['Ana', 'Aničić', '4/15', '2', '6']
['Višnja', 'Višnjić', '5/15', '1', '6.5']
```

9.4.2 Upisivanje tabelarnih podataka

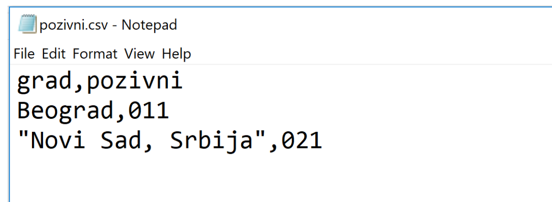
Ako su tabelarni podaci u programu već pripremljeni u obliku *liste listi*, onda se oni, uz pomoć funkcije `writer()`, mogu direktno upisati u CSV datoteku. Ova funkcija vraća objekat *upisivač*,[9] putem koga se tabelarni podaci upisuju odjednom ili red po red:

```
>>> from csv import writer
>>> tabela = [['grad', 'pozivni'],
              ['Beograd', '011'],
              ['Novi Sad, Srbija', '021']]
```

[8] Engl. *Reader*.
[9] Engl. *Writer*.

```
>>> # upis odjednom
>>> with open('c:\\papers\\pozivni.csv', 'w', encoding='utf-8', \
               newline='') as d:
        w = writer(d)
        w.writerows(tabela)

>>> # upis red po red
>>> with open('c:\\papers\\pozivni.csv', 'w', encoding='utf-8', \
               newline='') as d:
        w = writer(d)
        for red in tabela:
            w.writerow(red)

14
13
14
```

Opcioni argument funkcije `open()`, `newline=''`, omogućava da redovi u izlaznoj datoteci `pozivni.csv` budu jedan ispod drugog umesto sa po jednim redom proreda (slika 9.7).[10] Metoda `writerow()` vraća broj *upisanih* karaktera iz reda.



Slika 9.7: Formiranje CSV datoteke: modul `csv` pravilno razlikuje zapete u podacima od zapeta u funkciji separatora. Tekst sa zapetama "uokviren" je duplim znacima navoda.

Na kraju, treba naglasiti da se upotrebom opcionog parametra `delimiter`, u funkcijama `reader()` i `writer()`, može promeniti simbol separatora za podatke. Na primer, `csv.writer(dat, delimiter='\t')`, rezultovaće formiranjem datoteke u kojoj su podaci u redu odvojeni *tabulatorom* (kontrolna sekvenca `\t`).

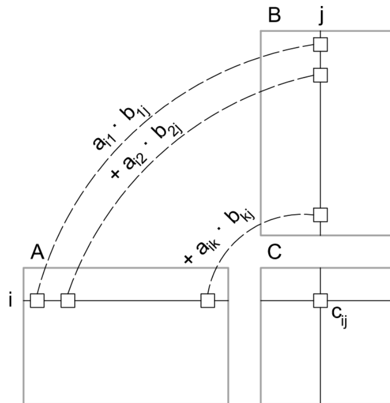
Problem 9.4 — Množenje matrica. Učitati saglasne matrice $\mathbf{A}$ i $\mathbf{B}$ iz datoteka u CSV formatu. Formirati matricu $\mathbf{C} = \mathbf{AB}$ i smestiti je u izlaznu datoteku u CSV formatu. ■

Posmatraju se saglasne matrice $\mathbf{A}_{m\times k}$ i $\mathbf{B}_{k\times n}$. Matrica $\mathbf{C}_{m\times n} = \mathbf{AB}$, određena je preko svog opšteg člana $c_{i,j}$:

$$c_{i,j} = \sum_{p=1}^{k} a_{i,p} b_{p,j}, \quad i = 1..m, \; j = 1..n \tag{9.1}$$

[10] Razlog za dodavanje opcionog argumenta pri otvaranju datoteke odnosi se na propust u dizajnu modula `csv`. U uobičajenom radu sa datotekama, ovaj opcioni argument nije potrebno navoditi!

Na osnovu (9.1), element $c_{i,j}$ formira se kao suma proizvoda odgovarajućih elemenata iz i-te vrste matrice **A** i j-te kolone matrice **B** (slika 9.8). Suma se može izračunati u iterativnom postupku pomoću tri ugnježđene petlje `for`. Prve dve petlje opredeljuju tekući element rezultujuće matrice (indeksi `i` i `j`), a unutrašnja petlja (po `p`) realizuje sumu iz (9.1).



Slika 9.8: Množenje matrica: algoritam za formiranje opšteg člana matice $\mathbf{C} = \mathbf{AB}$.

Prilikom rešavanja postavljenog problema, definišu se dve funkcije koje rade sa datotekama u CSV formatu: `učitaj_matricu(putanja)` učitava matricu navedenu argumentom `putanja`; `zapiši_matricu(putanja, M)` zapisuje matricu `M` u navedenu datoteku. Algoritam množenja, definisan sa (9.1), realizovan je u funkciji `množi_matrice(A, B)`:

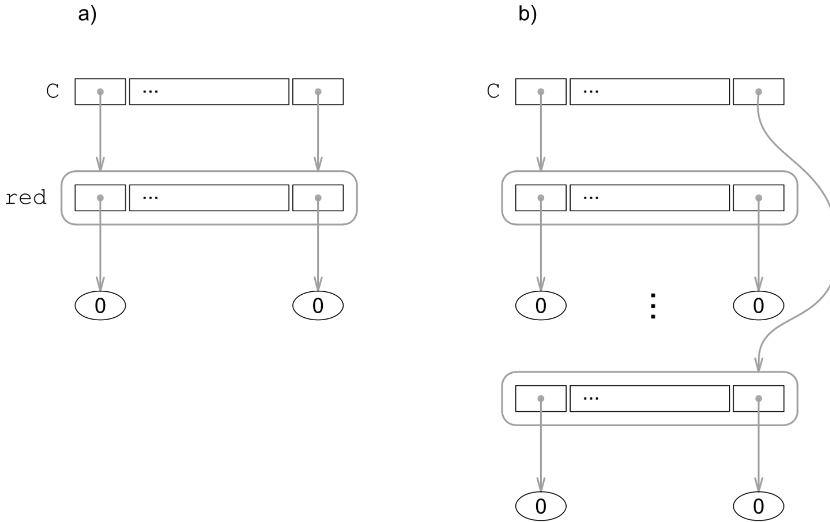
Program 9.4 — Množenje matrica.

```
# množenje matrica
import csv
def množi_matrice(A, B):

    m, k = len(A), len(B)
    if m == 0 or k == 0:
        raise Exception('A(B) mora imati bar jedan element')

    k1, n = len(A[0]), len(B[0])
    if k != k1:
        raise Exception('A i B nisu saglasne!')

    try:
        C = [ n * [0] for i in range(m)] # pravi C sa nulama
```

```
15          for i in range(m): # redovi C
16              for j in range(n):  #kolone C
17                  for p in range(k):
18                      C[i][j] += float(A[i][p]) * float(B[p][j])
19          return C
20
21      except:
22          raise Exception('Problem u formatu matrice')
23
24  def učitaj_matricu(putanja):
25
26      try:
27          with open(putanja) as d:
28              M = list(csv.reader(d))
29          return M
30      except:
31          raise Exception('Problem pri učitavanju ' + putanja)
32
33  def zapiši_matricu(putanja, M):
34
35      try:
36          with open(putanja, 'w', newline='') as d:
37              csv.writer(d).writerows(M)
38      except:
39          raise Exception('Greška pri upisu ' + putanja)
40
41  # test program
42  try:
43      A = učitaj_matricu(input('datoteka A '))
44      B = učitaj_matricu(input('datoteka B '))
45      zapiši_matricu(input('datoteka C=AB '), množi_matrice(A,B))
46
47  except Exception as e:
48      print(e)
```

U funkciji `množi_matrice()` prvo se proverava da li je neka od ulaznih matrica prazna (r5-6). Ako jesete, generiše se izuzetak sa odgovarajućom porukom (r7). Kako se ovaj izuzetak ne obrađuje, funkcija prekida sa radom, a izuzetak se prosleđuje u pozivajući program (r45). Zahvaljujući specifičnom obliku primenjene strukture `try-catch` (r42-48), prosleđena greška predstavljena je objektom na koji ukazuje promenljiva `e` (r47), što omogućava ispis poruke o grešci (r48).

U istoj funkciji se proverava da li su matrice saglasne (r9-11). Algoritam množenja realizovan je u kritičnoj sekciji (r14-19). Uočiti da se, za potrebe iterativnog postupka, rezultujuća matrica `C` inicijalizuje nulama (r14). Neiskusan programer mogao je, umesto (r14), doći u iskušenje da napiše `red = n * [0]`, pa `C = [red for i in range(m)]`. Tada bi se `C` *pogrešno* formirala kao matrica kod koje sve vrste ukazuju na isti red (slika 9.9-a), umesto ispravne varijante sa slike 9.9-b.



Slika 9.9: Inicijalizacija matrice `C=AB`: (a) neispravni oblik matrice; (b) ispravni oblik matrice. Neispravni oblik nastaje kao posledica kopiranja *iste* objektne reference `red` u `C = [red for i in range(m)]`.

Funkcije koje učitavaju (r24-31) i upisuju odgovarajuće matrice (r33-39), koriste funkcionalnosti iz modula `csv`. U glavi 11, biće reči o paketu *NumPy* koji omogućava da se matrične operacije obave na *prirodniji* način nego što to dozvoljavaju matrice predstavljene listama. Rad programa dat je u sledećem prikazu i na slici 9.10:

```
datoteka A c:\papers\ne_postoji.csv
Problem pri učitavanju c:\papers\ne_postoji.csv
>>>
==== RESTART: C:/programi/p9_4.py ====
datoteka A c:\papers\matB.csv
datoteka B c:\papers\matA.csv
datoteka C=AB c:\papers\matC.csv
A i B nisu saglasne!
>>>
==== RESTART: C:/programi/p9_4.py ====
datoteka A c:\papers\matA.csv
datoteka B c:\papers\matB.csv
datoteka C=AB c:\papers\matC.csv
```

matA

	A	B	C	D	E
1	1	2	3	-1	
2	4	1	1	2	
3	5	0	2	-3	
4					

matB

	A	B	C
1	2	2	
2	1	1	
3	-1	0	
4	0	5	
5			

matC

	A	B	C	D	E
1	1	-1			
2	8	19			
3	8	-5			
4					

Slika 9.10: Množenje matrica: izgled ulaznih i izlazne CSV datoteke u Excel-u.

ŠTA JE NAUČENO

- Datoteke predstavljaju imenovane skupove podataka, uskladištene na trajnoj memoriji poput diska. Sadržaj binarne datoteke tumači se prema specifikaciji koju, za potrebe aplikacije, definiše programer. Bitovi tekstualne datoteke tumače se kao niske redova teksta, odvojenih separatorom za novi red.
- Sistem datoteka odnosi se na hijerarhijsku strukturu direktorijuma i datoteka u kome se svakom članu pristupa preko apsolutne ili relativne putanje. Za rad sa sistemom datoteka koristi se modul `os`.
- Apsolutna putanja je tekstualna sekvenca koja opisuje putanju od direktorijuma na vrhu hijerarhije pa do posmatrane datoteke (direktorijuma). Relativna putanja odnosi se na putanju koja polazi od tekućeg radnog direktorijuma. Za rad sa putanjama koristi se modul `os.path`.
- Tekstualne datoteke mogu se otvoriti za čitanje, upisivanje ili dodavanje. Pri otvaranju datoteke funkcijom `open()`, ona se predstavlja datotečkim objektom. Čitanje i upisivanje podataka vrši se uz pomoć metoda ovog objekta.
- Prilikom otvaranja datoteke za čitanje ili pisanje, treba paziti na izvorni (odredišni) sistem kodiranja kojim se navodi kako se niske bita preslikavaju u odgovarajuće karaktere. Ako se prilikom otvaranja ne navede sistem kodiranja, podrazumeva se preslikavanje definisano u okviru operativnog sistema.
- Ulazno-izlazne operacije mogu se obavljati u režimu karakter po karakter, red po red ili, ako to dozvoljava slobodna radna memorija, celokupan sadržaj.

- Datoteka se, po završetku rada, obavezno mora zatvoriti korišćenjem metode `close()`. Preporučuje se zatvaranje u bloku naredbe `finally` ili automatski, pri napuštanju bloka naredbe `with` u kome je datoteka otvorena.

- U praksi se često koriste tabelarni podaci u CSV formatu. Tabela se sastoji od redova (instance podataka) i kolona (osobine instanci). Vrednosti osobina svake instance razdvojene su, u okviru reda, zapetom ili nekim drugim naznačenim separatorom.

- Modul `csv` pruža funkcionalnosti za rad sa tabelarnim podacima. Tabela se, prilikom učitavanja, tretira kao lista listi koje sadrže tekstualne objekte. Na programu je da, u zavisnosti od tipa osobine, ispravno konvertuje tekstualne vrednosti u objekte odgovarajućih tipova.

10. Apstraktni tipovi podataka - klase

Svi problemi u dosadašnjem izlaganju rešavani su na *proceduralni* način. Centralno mesto u proceduralnom programiranju zauzima *procedura* (u Pajtonu, funkcija), kojom se definiše niz koraka za transformisanje ulaznih podataka u željene izlaze. U proceduralnom pristupu, objekti predefinisanih tipova najčešće su korišćeni za predstavljanje brojnih i tekstualnih podataka. Uvođenjem objektnih kolekcija, u glavi 5, značajno je proširen skup problema koji se mogao rešiti jednostavnim programom. Pored kolekcija, izučavani su i objektni tipovi kojima je apstrahovano sistemsko vreme, tekstualne datoteke i greške u izvršavanju programa. Svaki od ovih tipova *definiše* odgovarajući skup metoda koje se mogu primeniti nad pripadajućim objektima. Na primer, objektni tip kojim su reprezentovane tekstualne datoteke omogućava čitanje njihovog sadržaja putem `read()`, a upisivanje novog, putem `write()` metode.

Objektno orijentisani pristup *omogućava* definisanje novih *apstraktnih* tipova, slično kao što se u proceduralnom pristupu, putem funkcija, definišu nove funkcionalnosti. Međutim, za razliku od funkcija, tipovima se definišu skupovi *istovrsnih* objekata koji sadrže pridružene *podatke* i nad kojima se definišu odgovarajuće *operacije*. Uz pomoć objekata, program može *bolje* da modeluje domen problema. Objekti različitih tipova prirodno modeluju različite podsisteme realnog sistema, a sam sistem modeluje se programom u kome se ostvaruje *interakcija* između objekata. Objektni tipovi nadalje će se nazivati *klasama*.

10.1 Apstrahovanje realnog sveta - objekti i klase

Objekti su *svuda* oko nas. Na primer, kratkotrajni pogled na gradsku ulicu otkriva mnogobrojne *ljude* i *automobile* koji se užurbano kreću na sve strane. Svaki pojedinačni čovek (automobil) može se posmatrati kao *poseban* entitet - *objekat*. Objekti se mogu međusobno razlikovati po *osobinama* koje ih opisuju. Dva slučajna prolaznika, između ostalog, razlikuju se po imenu, polu, rodnom mestu, težini ili po profesiji kojom se bave. Automobili se mogu razlikovati po proizvođaču, modelu, boji, jačini motora ili po količini goriva u rezervoaru. Navedene karakteristike predstavljaju *svojstva* ili osobine uočenih objekata.

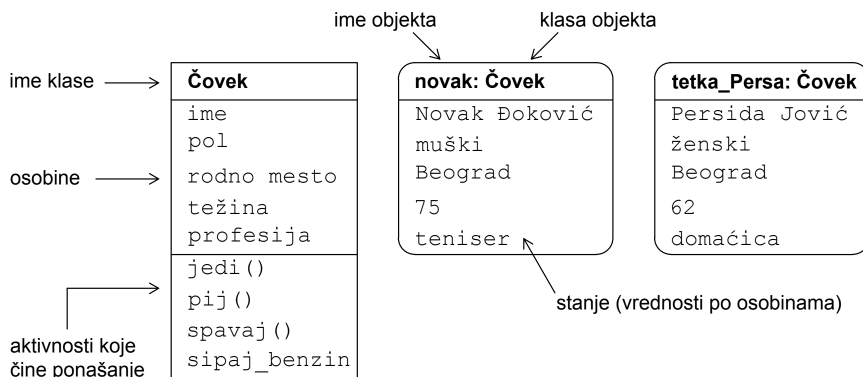
Objekat se, u svakom trenutku vremena, nalazi u određenom *stanju* koje je okarakterisano *trenutnim* vrednostima *svih* njegovih osobina. Ako se stanje objekta može promeniti, tako što se promeni vrednost bar jedne od njegovih osobina, onda je objekat *promenljivog* tipa. Na primer, težina svakog čoveka je promenljiva osobina pa su ljudi objekti promenljive prirode (bilo bi dobro da se menjaju i po nekim drugim aspektima ponašanja). Nasuprot ljudima, objekti brojnih tipova u Pajtonu su nepromenljivi.

Pored osobina, objekti se razlikuju i po sebi svojstvenom *ponašanju*. Skup svih *svojstvenih aktivnosti* koje jedan objekat može da obavlja, definiše njegovo ponašanje u svetu kome pripada. Prilikom obavljanja neke od aktivnosti, objekat može da *promeni* svoje stanje, kao i stanje objekata iz okolnog sveta. Na primer, svaki čovek, putem *ishrane* koju svakodnevno obavlja, menja svoju težinu. Ishrana je aktivnost koja se, po prirodi stvari, pridružuje čoveku, ali ne i automobilu. Slično, čovek *sipa* benzin u svoj automobil i na taj način menja količinu goriva u rezervoaru (promena stanja drugog objekta). Sa druge strane, aktivnost koja se obavlja pri pokretanju motora obuhvata niz tehničkih operacija unutar sistema vozila i svojstvena je automobilu. Treba primetiti da se neke aktivnosti pokreću *samoinicijativno* (ishrana), a neke *eksterno*, od strane drugih objekata (pokretanje automobila od strane čoveka).

U primeru gradske ulice uočava se postojanje *značajnije* razlike između slučajnog prolaznika i nasumičnog automobila nego što je to slučaj sa pojedinačnim prolaznicima ili pojedinačnim automobilima. Očigledno, svi prolaznici predstavljaju objekte *iste prirode* jer dele *iste* osobine i *ista* ponašanja. Skup svih objekata sa istim osobinama i ponašanjem naziva se *klasa*. Pojedinačni prolaznici mogu se tretirati kao objekti ili *instance* klase `Čovek`, dok se svi automobili mogu tretirati kao objekti klase `Automobil`.

Na slici 10.1 prikazana je jedna od mogućih realizacija klase `Čovek`, zajedno sa dve konkretne instance, uz pomoć simbola definisanih po UML specifikacji.[1] Iako instance iz klase imaju iste osobine i ponašanje, one se u svakom trenutku *razlikuju* po stanju u kome se nalaze. Međutim, nije nemoguće da se dva objekta iste klase nađu u istom stanju. Tada su oni ekvivalentni po sadržaju, u smislu jednakih vrednosti po osobinama.

[1] Engl. *Unified Modeling Language*. UML se koristi za modelovanje kompleksnih sistema. U daljem tekstu biće korišćeni samo pojedini elementi jezika. Za detaljnu specifikaciju videti `http://www.omg.org/spec/UML/`.



Slika 10.1: Klasa i objekti: klasa `Čovek` je apstraktan pojam kojim je definisan skup svih ljudi. Objekti iz klase predstavljaju stvarne, pojedinačne ljude, poput tetka Perse ili Novaka Đokovića. Klasa se šematski predstavlja pravougaonikom, a objekti pravougaonicima sa zaobljenim krajevima. Ime klase *uobičajeno* počinje *velikim* slovom. Prikaz klase, pored imena i svih osobina, navodi i sve aktivnosti koje određuju ponašanje njenih objekata. Prikaz instance sadrži konkretne vrednosti osobina u posmatranom trenutku - stanje objekta.

U objektno orijentisanom programu, klase predstavljaju *apstrakcije* (uprošćenja) stvarnih klasa, a objekti apstrakcije stvarnih objekata. Na programeru je da, u zavisnosti od *namene* programa, odredi potreban nivo apstrakcije. Posmatra se aplikacija za evidenciju izdatih vozila u okviru rent-a-car sistema. Objekti klase `Automobil` mogli bi se opisati osobinama poput broja registarske tablice, proizvođača, modela, broja sedišta, tipa i snage motora. Sa druge strane, u aplikaciji za simulaciju ponašanja automobila na putu, osobina poput broja registarske tablice nije od interesa. Opet, raspon između točkova, ili visina najniže tačke na vozilu u odnosu na put, od bitnog su značaja za primenu. Slično važi i za usvojeno ponašanje objekata iz klase.

10.1.1 Enkapsulacija - atributi i metode

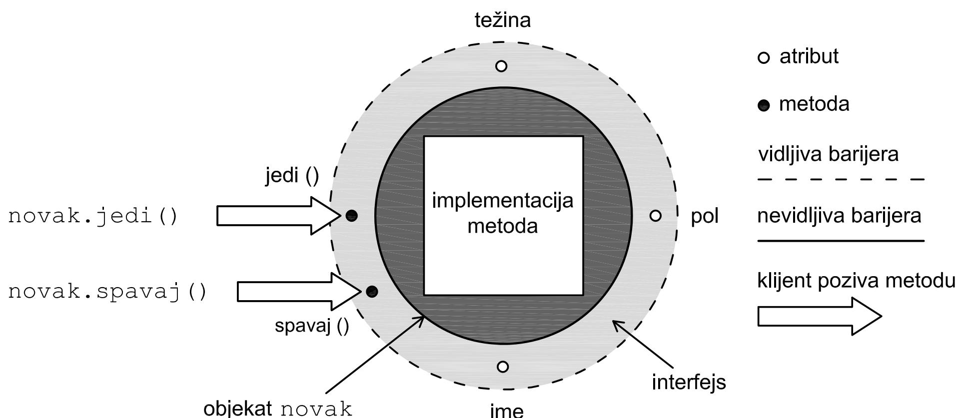
Po završenom procesu apstrahovanja, posmatrani sistem prevodi se u *klasni model* koji predstavlja skup klasa sa odabranim osobinama i ponašanjem. U objektno orijentisanom programu, osobine definisane u specifikaciji klase realizuju se na nivou *svakog* objekta preko *promenljivih* odgovarajućih tipova podataka. U primeru sa slike 10.1, u klasi `Čovek`, definisana je osobina `ime`. Ovoj osobini, na nivou svakog objekta, odgovara *istoimena* tekstualna promenljiva koja ukazuje na ime konkretnog čoveka. Tako se čovek, predstavljen objektom `novak`, zove `'Novak Đoković'`, a objektom `tetka_Persa`, `'Persida Jović'`. Kako svakoj osobini odgovara po jedna promenljiva koja se pridružuje svakoj instanci, ove promenljive zovu se *promenljivim instance* ili *atributima*.[2] Stanje objekta definisano je vrednostima svih njegovih atributa.

[2] Engl. *Instance variables*, ili *Attributes*.

U Pajtonu, kao i u većini drugih jezika, atributima se pristupa u *notaciji sa tačkom*, pri čemu su atributi, kao i ostale promenljive, zapravo reference na objekte odgovarajućih tipova. Tako bi se, sa `novak.ime`, pristupilo imenu osobe iz klase `Čovek` na koju ukazuje promenljiva `novak`.

Sve aktivnosti koje definišu ponašanje objekta jedne klase realizovane su uz pomoć specijalnih funkcija - *metoda*. Metode se pozivaju *isključivo nad* objektima iz klase pa se nazivaju i *metodama instanci*.[3] O njima je bilo reči u glavi 5.1.2 i od tada su često korišćene u radu sa objektima predefinisanih klasa poput `str` ili `list`. Prilikom poziva se koristi notacija sa tačkom. Na primer, sa `novak.jedi()`, naređuje se Novaku, a sa `tetka_Persa.jedi()`, tetka Persi da nešto pojede. Poziv može biti samoinicijativan (u metodi se poziva druga metoda), ili od spolja (objekat iste ili druge klase aktivira metodu). Za razliku od prikaza sa slike 10.1, u prikazu sa više detalja mogu se navesti i mogući ulazni parametri metoda, zajedno sa odgovarajućim tipovima.

Zahvaljujući konceptu klase sa atributima i metodama, svi neophodni podaci i operacije *enkapsulirani* su u okviru objekta kome se pristupa preko njegove reference. Proces enkapsulacije, kao jedan od osnovnih principa objektno orijentisanog programiranja, predstavlja *smeštanje* svih relevantnih podataka (atributa) i ponašanja (metode), u *jednistvenu* komponentu - objekat. Sa objektima se, od spolja, može raditi samo ono što je *predviđeno* metodama iz klase, a detalji implementacije ostaju *sakriveni* (slika 10.2). Sve *dostupne* metode i atributi čine *interfejs* klase. Njeni klijenti treba da znaju *šta* mogu da rade sa objektima (interfejs), a ne kako pojedine operacije rade (implementacija). Na primer, za slanje Novaka na osmočasovni počinak, klijent navodi `novak.spavaj(8)`. Tada mu je nebitno da li će Novak brojati ovce ili udisati duboko. Princip *sakrivanja informacija* omogućava da se, prilikom definisanja atributa i metoda, može naznačiti da li pripadaju interfejsu klase ili njenoj implementaciji (glava 10.1.5).

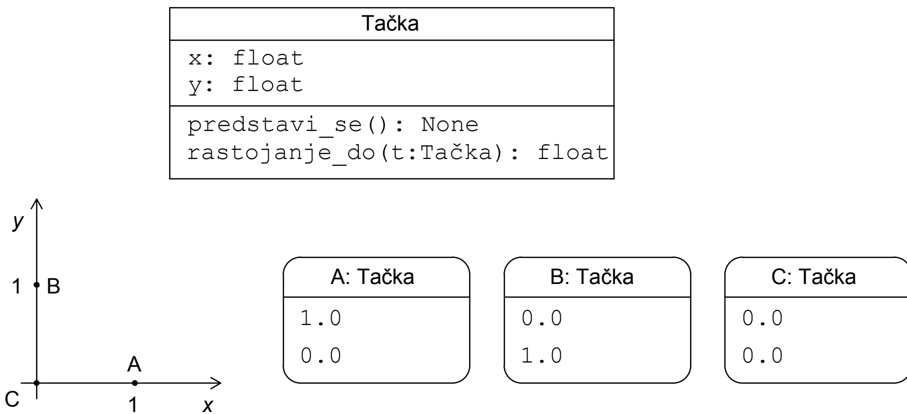


Slika 10.2: Enkapsulacija: interfejs klase `Čovek` čine svi atributi i metode iz klase. Poziv metode od spolja nalikuje "pritisku na dugme" na površini neprobojnog omotača objekta.

[3] Engl. *Instance methods*.

10.1.2 Definisanje klase

Postupak definisanja klase u programu biće prikazan na jednostavnom primeru klase `Tačka` kojom se modeluju tačke u ravni. Sve tačke opisane su sa dva atributa `x` i `y` koja predstavljaju odgovarajuće koordinate u ravni. Ponašanje objekata u klasi definisano je metodama `predstavi_se()` i `rastojanje_do()`. Prva metoda ispisuje informacije (uređeni par koordinata) o konkretnoj tački (objektu) nad kojom je pokrenuta. Druga metoda vraća realni broj koji se odnosi na rastojanje od tačke (objekta) nad kojom se poziva, do tačke (objekta) koja se prosleđuje metodi kao parametar (slika 10.3).



Slika 10.3: Klasa `Tačka`: u prikazu klase, pored atributa navedeni su njihovi tipovi, a pored metoda tipovi povratnih vrednosti i tipovi ulaznih parametara. Za trougao definisan temenima A, B i C u koordinatnom sistemu, navedeni su konkretni objekti (tačke) kojima se on realizuje.

Klasa `Tačka` definisana je u modulu `tacke.py` na sledeći način:[4]

```
1  from math import sqrt
2  # definiše apstraktni tip koji predstavlja sve tačke u ravni
3  class Tačka:
4      ''' Predstavlja tačke u ravni definisane sa x i y koord'''
5
6      # poziva se iz konstruktora pri kreiranju objekta
7      def __init__(self, x = 0, y = 0):
8
9          self.x = x       # definisanje atributa x
10         self.y = y       # definisanje atributa y
```

[4] Za razliku od nekih jezika poput Jave, u jednoj datoteci (modulu) može se definisati *proizvoljan* broj klasa. Ime datoteke *ne mora* da odgovara imenu klase koja se definiše.

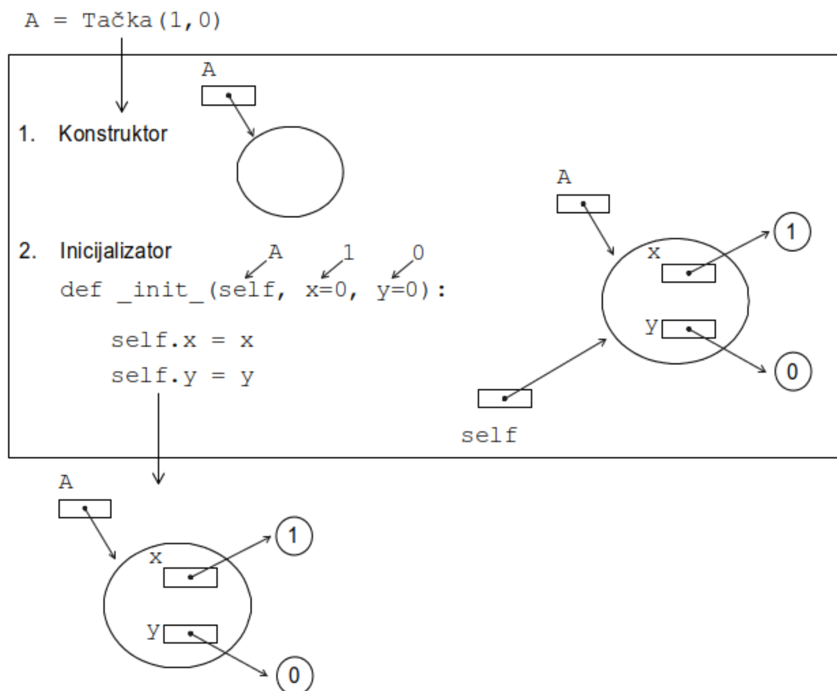
```
11
12      # ispisuje podatke o objektu
13      def predstavi_se(self):
14
15          print(f'({self.x:<f}, {self.y:<f})')
16
17      # računa rastojanje do druge tačke t
18      def rastojanje_do(self, t):
19
20          dx = self.x - t.x
21          dy = self.y - t.y
22          return sqrt(dx**2 + dy**2)
```

Definisanje klase započinje upotrebom službene reči `class`, a njeno ime navodi se pre simbola dvotačke (`Tačka`, r3). U bloku klase definišu se dve metode sa slike 10.3, kao i specijalna metoda `__init()__` - poziva se *svaki* put kada se kreira *novi* objekat:

```
>>> from tacke import Tačka
>>> A = Tačka(1,0)  # poziv konstruktora klase Tačka
>>> type(A)
<class 'tacke.Tačka'>
>>> A.x                 # pristupanje atributu x
1
>>> A.y                 # pristupanje atributu y
0
>>> C = Tačka()  # poziv konstruktora sa podrazumevanim vrednostima
>>> print(C.x, C.y)
0 0
```

Prilikom izvršavanja reda `A = Tačka(1,0)`, poziva se *konstruktor* klase `Tačka` kojim se kreira novi objekat. O konstruktorima predefinisanih tipova, poput `int()` ili `str()`, bilo je reči u glavi 2.3.4. Treba upamtiti da konstruktor *ne inicijalizuje* stanje objekta, već kreira potrebnu *memorijsku strukturu* za čuvanje budućeg sadržaja. Međutim, ukoliko se u definiciji klase pojavi metoda sa specijalnim imenom `__init()__`, onda je konstruktor *obavezno* poziva i prosleđuje joj navedene argumente (r7-10, u pozivu konstruktora za tačku `A` argumenti su `1` i `0`).

Potpisi svih metoda u klasi, uključujući i `__init()__`, za *prvi* formalni parametar imaju referencu na konkretan objekat *nad kojim* se poziv izvršava. Uobičajeno ime za ovu referencu je `self` (r7, 13, 18). Po kreiranju objekta, konstruktor poziva *inicijalizator* (metoda `__init()__`) kome *automatski* prosleđuje, kao prvi argument, stvarnu referencu na kreirani objekat. U inicijalizatoru se postavljaju vrednosti za atribute `x` (`self.x`) i `y` (`self.y`), uz pomoć notacije sa tačkom (r9, 10) - slika 10.4.



Slika 10.4: Postupak kreiranja novog objekta: konstruktor prvo kreira prazan objekat pa potom, ako je definisana u klasi, poziva metodu `__init()__` koja formira i inicijalizuje atribute. Prilikom poziva inicijalizatora, referenca `A` kopira se u referencu `self` pa obe pokazuju na novokreirani objekat. U inicijalizatoru se atributima pristupa u notaciji sa tačkom preko reference `self`.

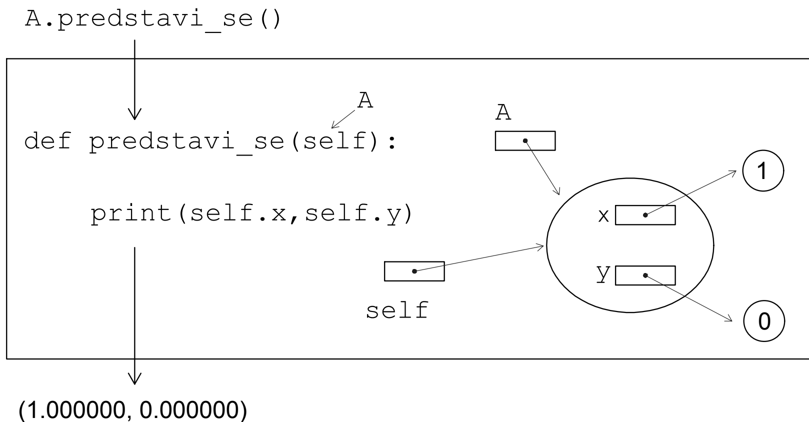
Atributi se, poput ostalih promenljivih u Pajtonu, definišu *dinamički* kada se njihova imena, sa prefiksom `self`, prvi put pojave u naredbi dodele vrednosti (na primer, `self.ja = 'Miloš'`). Po kreiranju objekta `A`, sa `A.x` i `A.y` pristupa se navedenim atributima za potrebe čitanja, odnosno promene vrednosti. Ako se u konstruktoru ne navedu koordinate tačke (`C = Tačka()`), onda se u inicijalizatoru za parametre `x` i `y` uzimaju podrazumevane vrednosti (r7).

! Dinamičko definisanje atributa omogućava njihovo uvođenje *u hodu* unutar proizvoljne metode u klasi. Međutim, autor ovo smatra *lošom* praksom koja smanjuje čitljivost programa! Imajući u vidu unapred poznatu specifikaciju klase, logično je da se svi atributi definišu u inicijalizatoru u trenutku kreiranja objekta.

U gornjem primeru, kreirana su dva objekta (tačke) sa istim atributima, ali sa različitim stanjem (vrednosti atributa). U opštem slučaju, moguće je kreirati onoliko instanci neke klase koliko dozvoljava radna memorija, pri čemu stanja pojedinih objekata odgovaraju željenoj primeni. Model sa slike 10.3 realizuje se na sledeći način:

```
>>> A = Tačka(1,0)
>>> B = Tačka(0,1)
>>> C = Tačka()
>>> A.predstavi_se()
(1.000000, 0.000000)
>>> A.rastojanje_do(C)
1.0
>>> A.rastojanje_do(B)
1.4142135623730951
```

Metoda `predstavi_se()` ispisuje koordinate tačke nad kojom je pozvana (r13-15). Prilikom njenog poziva se *ne navode* argumenti, iako njen potpis sadrži ulazni parametar `self` (r13). Uopšte, prilikom poziva metode instance nad objektom `x`, referenca `x` *automatski* se *kopira* u *prvi* argument metode. Na primer, u toku izvršavanja `A.predstavi_se()`, referenca `self` ukazuje na isti objekat kao i `A` (slika 10.5). Tom prilikom se u r15 ispisuju vrednosti koordinata tačke na koju ukazuje `self` (`self.x` i `self.y`). U opštem slučaju, potpis metode instance *uvek* sadrži jedan parametar više (`self`), u odnosu na broj argumenata u pozivu.



Slika 10.5: Automatsko kopiranje objektne reference u parametar `self`. Atributima objekta, u okviru metode instance, pristupa se navođenjem reference `self` u notaciji sa tačkom. Format ispisa u metodi `predstavi_se()` uprošćen je u odnosu na originalni program.

> **!** Objektna referenca preko koje se pristupa atributima instance *ne mora* da se zove `self`, ali je to opšte prihvaćena konvencija među programerima u Pajtonu. Ako želite da vas drugi lakše razumeju, poštujte konvenciju!

Metoda `rastojanje_do()`, pored reference `self`, prima i parametar `t` koji ukazuje na tačku do koje se računa rastojanje (r18). Promenljive `self.x` i `self.y` ukazuju na

koordinate tačke nad kojom je metoda pozvana, a `t.x` i `t.y` na koordinate tačke koja je predstavljena parametrom `t` (r20, 21). Rastojanje se računa prema formuli za euklidsko rastojanje koje se u ravni svodi na Pitagorinu teoremu (r22). Primetiti da se, prilikom poziva metode, navodi samo referenca na drugu tačku.

10.1.3 Magične metode i preopterećenje operatora

Metode čija imena počinju i završavaju se sa dva simbola `'_'`, poput `__init__()`, nazivaju se magične[5] i imaju *poseban* tretman prilikom izvršavanja programa u Pajtonu. One se mogu pozivati nad objektima na uobičajeni način, ali ih, za razliku od ostalih metoda, *najčešće* poziva interpreter kada se za to steknu određeni uslovi. Na primer, metodu `__init__()` interpreter poziva iz konstruktora kako bi se sprovela korisnički definisana inicijalizacija novokreiranog objekta. Koncept se dodatno ilustruje na primeru često korišćenih metoda `__str__()`, `__eq__()` i `__add__()`, koje su dodate postojećoj klasi `Tačka`:

```
1  from math import sqrt
2  # definiše apstraktni tip koji predstavlja sve tačke u ravni
3  class Tačka:
4      ''' Predstavlja tačke u ravni definisane sa x i y koord'''
5      # poziva se iz konstruktora pri kreiranju objekta
6      def __init__(self, x = 0, y = 0):
7          # ... kao u prethodnom prikazu
8
9      # računa rastojanje do druge tačke t
10     def rastojanje_do(self, t):
11         # ... kao u prethodnom prikazu
12
13     # tekstualna reprezentacija objekta, za print(t) ili str(t)
14     def __str__(self):
15
16         return f'({self.x:<f}, {self.y:<f})'
17
18     # poredi da li su dve tačke jednake
19     def __eq__(self, t):
20
21         if isinstance(t, Tačka):
22             return self.x == t.x and self.y == t.y
23         return False
```

[5] Engl. *Magic methods*.

```
24
25      # sabira dva vektora definisana tačkama
26      def __add__(self, t):
27
28          return Tačka(self.x + t.x, self.y + t.y)
```

Tekstualna reprezentacija objekta

Umesto metode `predstavi_se()`, u proširenoj definiciji klase `Tačka` pojavljuje se metoda `__str__()` koja vraća *korisnički* definisanu tekstualnu reprezentaciju tačke (r14-16). Poziva se indirektno u funkciji `print()` ili pri kreiranju tekstualne sekvence u konstruktoru `str()`:

```
>>> from tacke import Tačka
>>> A = Tačka(1,0)
>>> print(A)
(1.000000, 0.000000)
>>> 'Tačka A je ' + str(A)
'Tačka A je (1.000000, 0.000000)
```

U zavisnosti od prirode klase i na osnovu značenja pojedinih atributa, programer je u mogućnosti da definiše pogodnu tekstualnu reprezentaciju objekta. Međutim, ako se metoda `__str__()` ne definiše, onda reprezentacija nosi informaciju o *tipu* objekta i njegovoj *heksadekadnoj adresi* u memoriji:

```
>>> print(A)
<tacke.Tačka object at 0x000001C0CDFE4978>
>>> # dokaz za adresu, id() vraća dekadnu adresu
>>> print(hex(id(A)))  # hex() pretvara dek. broj u heksadek. broj
0x1c0cdfe4978
```

Poređenje objekata po sadržaju

Iz dosadašnjeg izlaganja poznato je da se poređenje dve reference, po tome da li ukazuju na isti objekat, obavlja operatorom `is` (na primer, `if a is b:`). Prilikom poređenja objekata korisnički definisanih klasa po *jednakosti sadržaja* (stanja), potrebno je *redefinisati* značenje operatora ==. Ovo se čini u metodi `__eq__()` koju interpreter poziva kada izračunava istinitosnu vrednost izraza `a == b` (r19-23). Tom prilikom, ova metoda poziva se iz klase *levog* operanda (`a`), pod uslovom da je u njoj definisana:

```
>>> A = Tačka(1,0)
>>> B = Tačka(0,1)
>>> X = Tačka(1,0)
```

```
>>> A == B # poziv __eq__() iz klase objekta A (a ne klase B!)
False
>>> A == None
False
>>> A == X
True
```

Implementacija metode definiše jednakost između tačaka prema tome da li se podudaraju (imaju iste koordinate). Prvo se testira da li je objekat sa kojim se vrši poređenje iz klase `Tačka` (r21). U tu svrhu koristi se funkcija `isinstance()` koja vraća `True` ako je njen prvi argument iz klase čije ime je zadato drugim. Ako je uslov ispunjen, porede se odgovarajuće koordinate (r22). U suprotnom, vraća se `False` (r23).

Preopterećenje operatora

Realizovanjem metode `__eq__()` izvršeno je *preopterećenje* binarnog operatora ==, tako da se on ponaša *različito* za različite klase levog operanda. O preopterećenju je već bilo reči u glavi 5.1.1, kada je diskutovano ponašanje operatora + i * u kontekstu primene nad tekstualnim sekvencama. Standardno ponašanje operatora == podrazumeva poređenje objektnih referenci – testira da li dve promenljive ukazuju na isti objekat (kao `is`). U slučaju do sada pominjanih predefinisanih tipova, ponašanje operatora == redefinisano je na sledeći način:

- brojni tipovi se porede po vrednosti odgovarajućih objekata,
- tekstualne sekvence se porede po jednakosti karaktera na istim pozicijama,
- torke i liste se porede po jednakosti elemenata na istim pozicijama,
- skupovi se porede po jednakosti elemenata koje sadrže,
- rečnici se porede po jednakosti parova ključ - vrednost.

Preopterećenje operatora dodatno je ilustrovano na primeru metode `__add__()` koja se poziva prilikom izračunavanja izraza `a + b` (r26-28). Slično metodi `__eq__()`, aktivira se ako postoji u klasi levog operanda. Zbir dve tačke definiše se kao *nova* tačka čije koordinate su jednake zbiru odgovarajućih koordinata polaznih tačaka (r28):

```
>>> A = Tačka(1,0)
>>> B = Tačka(0,1)
>>> D = A + B
>>> print(D)
(1.000000, 1.000000)
>>> # može i ovako
>>> print(Tačka(1,0) + Tačka(0,1) + D) # redosled s leva na desno
(2.000000, 2.000000)
```

U gornjem primeru, pri sabiranju u okviru funkcije `print()`, pored već kreirane tačke `D` koriste se i dva *anonimna objekta* koja se kreiraju u toku izračunavanja izraza. Anonimni objekti imaju *privremeni* karakter pa se ne imenuju. U tabeli 10.1 prikazane su neke od često korišćenih magičnih metoda pomoću kojih se redefiniše standardno ponašanje pojedinih operatora.

operator	**magična metoda**
unarni	
`-`	`__neg__(self)`
`+`	`__pos__(self)`
binarni	
`-`	`__sub__(self, obj)`
`+`	`__add__(self, obj)`
`*`	`__mul__(self, obj)`
`/`	`__truediv__(self, obj)`
`//`	`__floordiv__(self, obj)`
`%`	`__mod__(self, obj)`
`**`	`__pov__(self, obj)`
poređenja	
`<`	`__lt__(self, obj)`
`<=`	`__le__(self, obj)`
`>`	`__gt__(self, obj)`
`>=`	`__ge__(self, obj)`
`==`	`__eq__(self, obj)`
`!=`	`__ne__(self, obj)`

Tabela 10.1: Preopterećenje operatora i često korišćene magične metode. Za binarne operatore podrazumeva se izbor metode iz klase levog operanda (desni je `obj`). Pajton poseduje i magične metode koje se aktiviraju na osnovu klase desnog operanda, ali o tome ovde neće biti reči.

Problem 10.1 — Razlomci. Realizovati klasu kojom se predstavljaju razlomci definisani celobrojnim brojiocem i imeniocem. Predvideti osnovne operacije sa razlomcima poput sabiranja ili množenja. Prilikom kreiranja objekta, razlomak predstaviti u nesvodljivoj formi (brojilac i imenilac uzajamno prosti). Na primer, $\frac{6}{4}$ svesti na $\frac{3}{2}$. ■

Svaki razlomak može se predstaviti pomoću dva cela broja kao $\frac{a}{b}$, $a \in Z$, $b \in N$, pri čemu su brojilac a i imenilac b uzajamno prosti – nemaju zajedničkih delilaca osim jedinice. U predloženoj reprezentaciji brojilac *čuva* informaciju o znaku razlomka. U klasi će biti definisana metoda `vrednost()` koja vraća realnu vrednost razlomka,

kao i magične metode kojima se redefinišu operacije sabiranja, oduzimanja, množenja i deljenja razlomaka. Pored njih, klasa sadrži metode za inicijalizaciju i formiranje tekstualne reprezentacije, `__init__()` i `__str__()`, čija realizacija se *preporučuje* za sve korisnički definisane apstraktne tipove.

Prilikom inicijalizacije objekta proizvoljnim parom celih brojeva treba obezbediti da brojilac čuva znak razlomka, a da imenilac bude uvek pozitivan. Potom treba obaviti *skraćivanje* razlomka *najvećim* zajedničkim deliocem brojioca i imenioca – objekti predstavljaju razlomke koji su u nesvodljivoj formi. U tu svrhu, u modul koji sadrži definiciju klase, dodaje se funkcija za izračunavanje najvećeg zajedničkog delioca - `nzd()`. Tokom obavljanja aritmetičkih operacija treba kreirati nove razlomke.

Razmatra se prvo izračunavanje najvećeg zajedničkog delioca dva prirodna broja a i b, $nzd(a,b)$, pri čemu je $a \geqslant b$. Rešenje se zasniva na sledećoj jednakosti:

$$nzd(a,b) = nzd(b, a \bmod b) \tag{10.1}$$

U jednakosti (10.1), vrednost $a \bmod b$ predstavlja ostatak pri celobrojnom deljenju a sa b, koji se može zapisati i kao $o = a - bp$, $0 \leqslant o < b$. Na primer, važi $nzd(24,10) = 2$, ali i $nzd(10, 24 \bmod 10) = nzd(10,4) = 2$. Sledi dokaz za (10.1), pri čemu se sa $x|y$ označava da x deli y bez ostatka:

$$\begin{aligned}
& nzd(a,b) \mid a \quad \wedge \quad nzd(a,b) \mid b \implies \\
& nzd(a,b) \mid (a - bp) \implies \\
& nzd(a,b) \mid o \implies \\
& nzd(a,b) \leqslant nzd(b,o)
\end{aligned}$$

ali važi i obrnuto:

$$\begin{aligned}
& nzd(a,b) \leqslant nzd(b,o) \quad \wedge \quad nzd(b,o) \mid o \implies \\
& nzd(b,o) \mid (o + bp) \implies \\
& nzd(b,o) \mid a \implies \\
& nzd(b,o) \leqslant nzd(a,b)
\end{aligned}$$

pa je:

$$\begin{aligned}
& nzd(a,b) \leqslant nzd(b,o) \quad \wedge \quad nzd(b,o) \leqslant nzd(a,b) \implies \\
& nzd(a,b) = nzd(b,o) \implies \\
& nzd(a,b) = nzd(b, a \bmod b)
\end{aligned}$$

Uz pomoć izraza (10.1), `nzd()` se može realizovati kao rekurzivna funkcija, pri čemu se bazni slučaj odnosi na situaciju $nzd(x,0) = x$ (x deli nulu bez ostatka). Na primer, $nzd(24,10) = nzd(10,4) = nzd(4,2) = nzd(2,0) = 2$. Kako je ostatak pri celobrojnom deljenju uvek manji od delioca, te kako se u svakom koraku rekurzije smanjuje,

bazni slučaj dostiže se u *konačno mnogo* koraka. Ako u startu nije ispunjen uslov $a \geqslant b$, brojevi zauzimaju pretpostavljeni položaj već nakon prvog koraka. Na primer, za $nzd(10, 24)$, $b = 24$ i $o = 10 \bmod 24 = 10$, pa posle prvog koraka treba izračunati $nzd(24, 10)$. Prikazani postupak predstavlja modifikovanu varijantu Euklidovog algoritma[6] koja vrlo brzo dolazi do traženog rešenja.

Program 10.1 — Razlomci.

```
# najveći zajednički delilac za prirodne brojeve x i y
def nzd(x, y):

    if y == 0:
        return x
    else:
        return nzd(y, x % y)

class Razlomak:
    '''Definiše razlomak zadat broijocem (a) i imeniocem (b)'''
    # poziva se iz konstruktora pri kreiranju razlomka
    def __init__(self, a, b):

        try:
            a, b = int(a), int(b)
        except:
            raise Exception('uneti brojevi nisu celobrojni!')

        if b == 0:
            raise Exception('imenilac je 0')
        else:
            a = -abs(a) if a*b < 0 else abs(a)
            b = abs(b)
            k = nzd(abs(a), b)
            self.a = a // k
            self.b = b // k

    # tekstualna reprezentacija razlomka
    def __str__(self):

        return f'{self.a:<d}/{self.b:<d}'

```

[6] Starogrčki matematičar Euklid opisao je u svojoj knjizi *Elementi*, oko 300. godine p.n.e, postupak za iznalaženje najvećeg zajedničkog delioca.

```
33      # vraća realnu vrednost
34      def vrednost(self):
35
36          return self.a / self.b
37
38      # redefinisanje operacije +
39      def __add__(self, r):
40
41          return Razlomak(self.a * r.b + r.a * self.b,
42                          self.b * r.b)
43
44      # redefinisanje operacije -
45      def __sub__(self, r):
46
47          return Razlomak(self.a * r.b - r.a * self.b,
48                          self.b * r.b)
49
50      # redefinisanje operacije *
51      def __mul__(self, r):
52
53          return Razlomak(self.a * r.a, self.b * r.b)
54
55      # redefinisanje operacije /
56      def __truediv__(self, r):
57
58          return Razlomak(self.a * r.b, r.a * self.b)
59
60  # test program
61  x = Razlomak(-3, 5)
62  y = Razlomak(5, 3)
63  print(f'{x} + {y} = {x + y}')
64  print(f'{x} - {y} = {x - y}')
65  print(f'{x} * {y} = {x * y}')
66  print(f'{x} / {y} = {x/y}')
```

Kako se funkcija `nzd()` nalazi van definicije klase, njeno ime pripada globalnom imenskom prostoru modula pa je dostupna iz svake metode unutar klase (r2-7). U inicijalizatoru se prvo proverava da li su uneti brojevi celobrojni (r14-17), odnosno da li je imenilac jednak nuli (r19). Celobrojnost se proverava *indirektno*, u kritičnoj sekciji, tako što se pokušava sa konverzijom ulaznih parametara u celobrojne vrednosti. Ako su ulazni parametri neispravni, prekida se sa inicijalizacijom i generiše se odgovarajući

izuzetak sa predefinisanom porukom (r17, 20). Potom se obezbeđuje da brojilac nosi informaciju o znaku razlomka i da imenilac uvek bude pozitivan ceo broj (r22, 23). Na primer, pri navođenju konstruktora `Razlomak(-5,-7)`, brojilac i imenilac bili bi transformisani u 5 i 7. U nastavku, razlomak se skraćuje najvećim zajedničkim deliocem navedenih brojeva (r24-26). Budući da su vrlo slične, od metoda kojima se menja ponašanje aritmetičkih operatora, u daljoj diskusiji biće pomenuta ona koja se odnosi na sabiranje (r39-42).

Prilikom sabiranja dva razlomka kreira se treći sa odgovarajućim brojiocem i imeniocem, a prema izrazu $\frac{a}{b} + \frac{c}{d} = \frac{ad+cb}{bd}$ (r41-42). Još jednom se podvlači da se, prlikom kreiranja novog objekta, brojilac i imenilac prevode u nesvodljivu formu primenom skraćivanja sa najvećim zajedničkim deliocem. Rezultati programa za testiranje (r61-66) dati su ispod:

```
 -3/5 + 5/3 = 16/15
-3/5 - 5/3 = -34/15
-3/5 * 5/3 = -1/1
-3/5 / 5/3 = -9/25
```

Testiranje rada modula uz pomoć atributa `__name__`

Prilikom uvođenja modula `M` u drugi modul, sa `import M`, interpreter izvršava naredbe iz `M` (glava 4.2.1). U prethodnom primeru sa razlomcima, po uvođenju modula koji sadrži klasu `Razlomak` i posle obrade definicije klase, interpreter izvršava naredbe za testiranje (r61-66). Takvo ponašanje poželjno je *samo* u situaciji kada se *testira* rad modula. Međutim, prilikom korišćenja modula u drugim programima koji u svojoj realizaciji koriste razlomke, izvršavanje sekvence za testiranje bilo bi *nepoželjno*.

U glavi 4.2.3 navedeno je da se svaki modul, po uvođenju u druge celine, tretira kao *objekat* klase `module`. U okviru ove klase definisan je atribut `__name__` koji ukazuje na ime uvedenog modula – ime datoteke, bez ekstenzije `.py`. Kada se modul izvršava kao skript (na primer, pri pokretanju u IDLE-u sa <F5>), pomenuti atribut ima vrednost `'__main__'`. Ovo se može upotrebiti za *sprečavanje* izvršavanja pojedinih *delova* modula kada se on ne pokreće kao skript, već se uvodi u druge module:

```
1  # Modul osobe, smešten u datoteku osobe.py
2  class Osoba:
3      '''Definiše osobu sa imenom i prezimenom'''
4  
5      def __init__(self, ime, prezime):
6  
7          self.ime, self.prezime = ime, prezime
8  
```

```
9       def __str__(self):
10
11          return f'{self.ime} {self.prezime}'
12
13  # test sekvenca
14  print(__name__) # izvršava se uvek
15  if __name__ == '__main__':
16      # samo ako je pokrenut kao skript (npr. F5 u IDLE-u)
17      pera = Osoba('Petar', 'Petrović')
18      print(pera)
```

```
==== RESTART: C:/programi/osobe.py ====
__main__
Petar Petrović
>>> import osobe # sada sekvenca za testiranje neće biti pokrenuta
osobe
>>> mile = osobe.Osoba('Mile', 'Kitić')
>>> print(mile)
Mile Kitić
```

Modul `osobe` sadrži definiciju klase `Osoba` sa atributima koji čuvaju informaciju o imenu i prezimenu. Prilikom poziva modula kao skripta (sa <F5>), vrši se ponovno pokretanje interpreterske sesije (`RESTART`) i potom se modul izvršava. Prvo se ispisuje vrednost atributa `__name__` (r14). Kako je ona jednaka `'__main__'`, izvršava se sekvenca za testiranje u kojoj se kreira i ispisuje jedan objekat iz klase (r17-18). Potom se isti modul uvodi naredbom `import osobe`. Primetiti da je sada vrednost za `__name__` jednaka imenu modula pa se sekvenca za testiranje ne izvršava. U nastavku interaktivne sesije kreira se i ispisuje jedan (raspevani) objekat.

10.1.4 Klasni atributi i statičke metode

U dosadašnjem izlaganju razmatrani su atributi koji opisuju pojedinačne objekte (atributi instanci), kao i metode koje se pozivaju *isključivo* nad objektima (metode instanci). Međutim, ponekad je potrebno da svi objekti neke klase imaju pristup *zajedničkoj* promenljivoj, kao i da pozivaju *zajedničku* funkciju koja se *ne vezuje* ni za jednu pojedinačnu instancu. Pri tome se zajednička promenljiva (funkcija), po svojoj prirodi, odnosi na objekte iz klase pa je poželjno da se definiše *unutar* nje. Sledi ilustrativan primer: treba proširiti klasu `Tačka` iz poglavlja 10.1.2 tako da se u okviru klase vodi evidencija o broju kreiranih tačaka i vremenu poslednjeg kreiranja. Dodatno, potrebno je realizovati funkciju koja prigodno prikazuje ove informacije. Prikaz može biti na srpskom ili engleskom jeziku, u zavisnosti od navedenog ulaznog parametra.

Postavlja se pitanje *gde* čuvati informaciju o ukupnom broju kreiranih tačaka?

Na nivou pojedinačnih instanci to nije preporučljivo jer broj kreiranih objekata ne treba dovoditi u vezu sa stanjem bilo koje od tačaka. Ista konstatacija važi i za vreme kreiranja poslednjeg objekta (vreme poziva konstruktora). Postojanje *klasnih atributa* omogućava da se pojedine informacije, koje se *ne vezuju* za konkretne objekte, sačuvaju na nivou klase:

```
from time import sleep, asctime, localtime

class Tačka:
    ''' Klasa koja definiše tačke u ravni '''
    br_tačaka = 0         # klasni atribut
    t_kreiranja = None    # klasni atribut

    @staticmethod
    def statistika(jezik):

        if jezik == 'e':
            p1 = '# of created points:'
            p2 = 'last point created:'
        else:
            p1 = 'broj kreiranih tačaka:'
            p2 = 'poslednja tačka kreirana:'

        print(p1, Tačka.br_tačaka)
        if Tačka.t_kreiranja == None:
            Tačka.t_kreiranja = ' - '
        print(p2, Tačka.t_kreiranja, '\n')

    def __init__(self, x = 0, y = 0):

        # inicijalizacija instance
        self.x, self.y = x, y

        # statistika vezana za klasu (sve tačke)
        Tačka.br_tačaka += 1
        Tačka.t_kreiranja = asctime(localtime())

    def __str__(self):

        return f'({self.x:<f}, {self.y:<f})'

```

```
36         # ... slično kao u prethodnom listingu iste klase
37
38     # Test program
39     if __name__ == '__main__':
40
41         Tačka.statistika('s') # poziv statičkog metoda
42         a = Tačka()
43         print(a)
44         a.statistika('e')      # opet, ali nad instancom
45         sleep(1)
46         print(Tačka(1, -1))
47         Tačka.statistika('s') # pa opet
```

Klasni atributi `br_tačaka` i `t_kreiranja` definisani su u r5 i r6, *van* svih metoda u klasi. Za razliku od atributa instance, čije se vrednosti mogu razlikovati od objekta do objekta, klasni atribut ukazuje na *tačno jedan* objekat u radnoj memoriji. Klasni atribut dostupan je u metodi instance putem notacije sa tačkom: `ime_klase.atribut`. Prilikom kreiranja nove tačke u gornjem primeru, klasni atribut `Tačka.br_tačaka` uvećava se za jedan (r29), a pamti se i tekuće lokalno vreme (r30). Po izvršavanju redova r29 i r30, sve instance "vide" isti broj kreiranih tačaka, kao i vreme poslednjeg poziva inicijalizatora.

Metoda `statistika()` predstavlja *statičku* metodu kojom se realizuje prikazivanje vrednosti klasnih atributa uz korišćenje odabranog jezika (r8-21). Treba uočiti da se u bloku statičke metode *ne pojavljuje* referenca `self`. Dalje, iako potpis metode sadrži parametar `jezik` (r9), zahvaljujući oznaci `@staticmethod` (r8), on se *ne odnosi* na objekat nad kojim bi se metoda pozivala, kao što bi to bio slučaj da se radi o metodi instance koja ima `self` za prvi argument. Statičke metode najčešće se pozivaju nad *klasom*, umesto nad objektom (r41, 47). Upotrebljeni *dekorator*[7] proizvodi sledeći efekat: ako se metoda pozove nad objektom (r44), onda se objektna referenca *ne kopira* automatski u prvi parametar metode, kako bi se to očekivalo kod metode instance. Zato, prilikom poziva statičke metode, prvi argument *uvek* odgovara prvom ulaznom parametru iz potpisa, isto kao i kod funkcija. Ako se metoda pozove nad objektom (r45), ona ne može promeniti njegovo stanje, sem u slučaju kada bi se objektna referenca eksplicitno navela kao jedan od argumenata poziva. Sledi ilustracija rada modula koji je pokrenut kao skript:

```
broj kreiranih tačaka: 0
poslednja tačka kreirana:  -
```

[7] Konstrukcija koja počinje sa `@` naziva se *dekorator*. Dekoratori spadaju u domen funkcionalne paradigme jezika, te kao takvi neće biti razmatrani u ovoj knjizi.

```
(0.000000, 0.000000)
# of created points: 1
last point created: Mon Aug 14 23:12:10 2017

(1.000000, -1.000000)
broj kreiranih tačaka: 2
poslednja tačka kreirana: Mon Aug 14 23:12:11 2017
```

Iz dosadašnjeg izlaganja može se zaključiti da je funkcija `nzd()`, iz problema 10.1, mogla biti realizovana i kao statička metoda. U praksi se klasni atributi i statičke metode koriste znatno manje u odnosu na svoje parnjake na nivou instance.

Imenski prostori klase i objekta

U glavi 4.1.4 uveden je pojam imenskog prostora. On se realizuje preko *tabele imena* koja sadrži vezivanja između pojedinih promenljivih i konkretnih objekata u memoriji. Pomenuti su imenski prostori programa, funkcije, uvedenih i ugrađenog `builtins` modula. U toku izvršavanja, interpreter pronalazi objekat na osnovu imena promenljive koja na njega ukazuje i tabele imena iz odgovarajućeg prostora. Pored pomenutih prostora, u Pajtonu postoje i *imenski prostor klase* i *imenski prostor objekta*.

Prilikom obrade koda koji sadrži definiciju klase, interpreter kreira objekat koji reprezentuje *samu klasu*. Klasni objekat sadrži određene atribute koji opisuju klasu (*metapodaci*[8]), a njegov tip je `type`:

```
>>> from tacke import Tačka
>>> type(Tačka)
<class 'type'>
>>> Tačka.__dict__  # imenski prostor klase
mappingproxy({'br_tačaka': 2, '__str__': <function Tačka.__str__ at
0x00000225B43C7D90>, '__doc__':' Klasa koja definiše tačke u ravni ',
'__init__': <function Tačka.__init__ at 0x00000225B43C7D08>,
'statistika':<staticmethod object at 0x00000225B4373C18>,
'__module__': '__main__','__dict__': <attribute '__dict__' of 'Tačka'
objects>, '__weakref__':<attribute '__weakref__' of 'Tačka' objects>,
't_kreiranja': 'Fri Aug 18 18:48:08 2017'})
```

Svakom objektu klase `type` pridružen je specijalni atribut `__dict__`. On predstavlja *rečnik* kojim je realizovana tabela imena iz imenskog prostora klase. U primeru iznad, sa `Tačka.__dict__`, prikazana su pridruživanja u imenskom prostoru klase `Tačka`. Imenski prostor sadrži imena klasnih atributa `br_tačaka` i `t_kreiranja`, kao i imena svih metoda u klasi.

[8] Metapodaci su podaci koji opisuju druge podatke. Na primer, jedan Word dokument, pored sadržaja (glavna informacija), sadrži i metapodatke o vremenu kreiranja, veličini, vlasniku datoteke i slično.

Za novokreiranu instancu vezuje se *novi* imenski prostor objekta koji čuva imena svih atributa instance. Slično klasnom objektu, objekti apstraktnih tipova poseduju atribut `__dict__` kojim se realizuje odgovarajući imenski prostor objekta:

```
>>> a = Tačka(1,2)
>>> a.__dict__              # imenski prostor objekta
{'y': 2, 'x': 1}
>>> a.br_tačaka             # iz imenskog prostora klase
2
>>> a.br_tačaka = 3         # postaje atribut na nivou objekta
>>> Tačka.br_tačaka         # klasni atribut nepromenjen
2
>>> a.__dict__
{'y': 2, 'x': 1, 'br_tačaka': 3}
```

Primer pokazuje da imenski prostor objekta a inicijalno sadrži dva atributa. Ako se ime atributa ne pronađe u imenskom prostoru objekta, onda se ono traži u imenskom prostoru klase (`a.br_tačaka`), a ako ga nema ni tamo, interpreter prijavljuje grešku. Obratiti pažnju da se sa `a.br_tačaka = 3` *ne menja* vrednost klasnog atributa, već se u imenski prostor objekta a *dodaje* novi atribut instance. Atributi instance mogu se dinamički dodavati i van inicijalizatora, ali je već naglašeno da se to smatra *lošom* praksom. Promena vrednosti klasnog atributa iz objektne metode može se obaviti *jedino* sa prefiksom imena klase u notaciji sa tačkom (`Tačka.br_tačaka` u r29 ili `Tačka.t_kreiranja` u r30). Konačno, unutar objektne metode, a slično funkciji, važeći je *lokalni* imenski prostor metode. Imenima iz lokalnog prostora pristupa se *direktno*, bez upotrebe notacije sa tačkom.

10.1.5 Sakrivanje informacija

Ako ime atributa (metode) *počinje* sa jednom ili dve donje crte (npr. `_x` ili `__x`), onda se on smatra *privatnim* članom klase koji *nije* namenjen za korišćenje od spolja. Privatna imena *sakrivena* su unutar implementacije objekta pa *ne spadaju* u njegov interfejs (slika 10.2). Atributi i metode koji čine interfejs nazivaju se još i *javnim* članovima klase.

Da bi se shvatio značaj sakrivanja pojedinih atributa (metoda), posmatra se nekoliko verzija klase `Tačka_1k` koja predstavlja tačke iz prvog kvadranta u ravni ($x, y > 0$):

```
1 # modul t1k
2 # Verzija 1: bez privatnih imena
3 class Tačka_1k:
4     ''' Klasa kojom se definišu tačke u 1. kvadrantu ravni '''
```

```
5
6       def __init__(self, x, y):
7
8           if x > 0 and y > 0:
9               self.x, self.y = x, y
10          else:
11              raise Exception('x i/ili y pogrešni!')
12
13      def __str__(self):
14
15          return f'({self.x:<f}, {self.y:<f})'
16
17      # ... ostale metode
```

```
>>> from t1k import Tačka_1k
>>> a = Tačka_1k(-1, 1) # problem!
Traceback (most recent call last):
  File "<pyshell#43>", line 1, in <module>
    a = Tačka_1k(-1, 1)
  File "C:/programi\t1k.py", line 11, in __init__
    raise Exception('x i/ili y pogrešni!')
Exception: x i/ili y pogrešni!
>>> a = Tačka_1k(1, 1)
>>> a.x = -5  # problem!
>>> print(a)
(-5.000000, 1.000000)
```

Prilikom poziva konstruktora sa neadekvatnim argumentima (`Tačka_1k(-1, 1)`), generiše se izuzetak koji prekida rad konstruktora (r11). Tada se objekat ne kreira. Međutim, kako su svi atributi vidljivi spolja, korisnik klase (klijent) može *eksplicitno* menjati njihove vrednosti. Tako se, sa `a.x = -5`, objekat a dovodi u *pogrešno* stanje - tačka više ne pripada prvom kvadrantu! Ako se atributima promeni ime *dodavanjem* donje crte na početku (npr. `_x`), a u klasu ubace metode za *pristup* i *promenu vrednosti* atributa, dobija se sledeća varijanta:

```
1   # modul t1k
2   # Verzija 2: sa privatnim imenima tipa _ime
3   class Tačka_1k:
4       ''' Klasa kojom se definišu tačke u 1. kvadrantu ravni '''
5
```

```
6       def __init__(self, x, y):
7
8           if x > 0 and y > 0:
9               self._x, self._y = x, y # privatni atributi
10          else:
11              raise Exception('x i/ili y pogrešni!')
12
13      def daj_x(self):  # pristup atributu _x
14
15          return self._x
16
17      def postavi_x(self, novo_x): # menja atribut _x
18
19          if x > 0:
20              self._x = novo_x
21          else:
22              raise Exception('x <=0 !')
23
24      def daj_y(self):  # pristup atributu _y
25
26          return self._y
27
28      def postavi_y(self, novo_y): # menja atribut _y
29
30          if y > 0:
31              self._y = novo_y
32          else:
33              raise Exception('y <=0 !')
34
35      def __str__(self):
36
37          return f'({self._x:<f}, {self._y:<f})'
38      # ... ostale metode
```

Koordinate se, sa `self._x` i `self._y`, deklarišu kao *privatni* atributi koji čine deo implementacije klase (r9). Njima *ne bi trebalo* pristupati jer bi se narušilo stanje objekta. Privatnim atributima pristupa se uz pomoć metoda `daj_x()` i `daj_y()`, a vrednost im se postavlja sa `postavi_x()` i `postavi_y()`. Prednost ovakvog pristupa je u tome što se može *kontrolisati* na koji način se postavlja nova vrednost. Na primer, u `postavi_x()` (r17-22), ako vrednost koordinate nije veća od nule, generiše se izuzetak (r22), a objekat ostaje u ispravnom stanju. Uopšte uzevši, čitanje i postavljanje novih vrednosti enkapsulirano je u metode pa klijenti, koji *ispravno* koriste ovu klasu, ne

mogu da učine ništa mimo projektovanog ponašanja. Sledi prikaz rada druge verzije:

```
>>> a = Tačka_1k(1, 1)
>>> a.daj_x()
1
>>> a.postavi_x(-5)
Traceback (most recent call last):
  File "<pyshell#57>", line 1, in <module>
    a.postavi_x(-5)
  File "C:/programi\t1k.py", line 22, in postavi_x
    raise Exception('x <=0 !')
Exception: x <=0 !
>>> print(a)
(1.000000, 1.000000)
>>> a._x = -5  # opet problem!
>>> print(a)
(-5.000000, 1.000000)
```

Primer pokazuje da klijent *ipak može* da pristupi privatnom atributu i prevede objekat u neregularno stanje (`a._x = -5`). Pajton, za razliku od ostalih jezika poput Jave ili C++, *ne sprečava* klijenta da zanemari konvenciju i pristupi atributima (metodama) čija imena počinju sa _, ali onda klijent *preuzima odgovornost* za moguće posledice! Ipak, moguće je učiniti sledeće (varijanta 3):

```
1  # modul t1k
2  # Verzija 3: sa privatnim imenima tipa __ime
3  class Tačka_1k:
4      ''' Klasa kojom se definišu tačke u 1. kvadrantu ravni '''
5  
6      def __init__(self, x, y):
7  
8          if x > 0 and y > 0:
9              self.__x, self.__y = x, y # privatni atrib. sa dva _
10         else:
11             raise Exception('x i/ili y pogrešni!')
12 
13     def daj_x(self):  # pristup atributu __x
14 
15         return self.__x
16 
17     def postavi_x(self, novo_x): # menja atribut __x
18 
```

```
19          if x > 0:
20              self.__x = novo_x
21          else:
22              raise Exception('x <=0 !')
23
24      # ... ostale metode
```

U trećoj verziji, umesto `self._x` i `self._y`, stoji `self.__x` i `self.__y`. Na ovaj način, interpreter *otežava* klijentu da direktno pristupa sakrivenim atributima:

```
>>> a = Tačka_1k(1, 1)
>>> a.__x                   # ne može!
Traceback (most recent call last):
  File "<pyshell#72>", line 1, in <module>
    a.__x
AttributeError: 'Tačka_1k' object has no attribute '__x'
>>> a.__x = 5               # pokušaj promene definiše novi atribut!!!
>>> print(a)                # ipak je sve u redu
(1.000000, 1.000000)
>>> a.__dict__              #  imenski prostor objekta a
{'__x': 5, '_Tačka_1k__y': 1, '_Tačka_1k__x': 1}
>>> b = Tačka_1k(2, 2)
>>> b.__dict__              # imenski prostor objekta b
{'_Tačka_1k__y': 2, '_Tačka_1k__x': 2}
>>> a._Tačka_1k__x = -5  # ovo je problem!
>>> print(a)
(-5.000000, 1.000000)
```

Interpreter prijavljuje grešku kada se pokuša sa čitanjem vrednosti sakrivenog atributa `__x`. Kada se pokuša sa eksplicitnom dodelom vrednosti (`a.__x = 5`), greška se ne prijavljuje, ali vrednost atributa *nije* promenjena, što se vidi po ispisu objekta `a`. Po dodeli vrednosti, dinamički se kreira *novi* atribut na nivou instance `a`. Ovo se može proveriti tako što se, sa `a.__dict__`, prikaže tabela imena imenskog prostora objekta. Objekat `a` ima tri defnisana atributa: `__x`, `_Tačka_1k__x`, i `_Tačka_1k__y`. Prvi se odnosi na atribut koji je dinamički kreiran sa `a.__x = 5`. Druga dva predstavljaju koordinate definisane u inicijalizatoru. Uočava se da im vrednosti nisu promenjene jer prilikom definisanja atributa, čija imena počinju sa `__`, interpreter vrši *transformaciju* njihovih imena kako bi atributi ostali sakriveni (dodaje se prefiks `_ime_klase`). Već je istaknuto da usputno definisanje novih atributa ne predstavlja dobru praksu i da atribute treba definisati isključivo unutar inicijalizatora.

Kada se kreira tačka `b`, novi imenski prostor vezuje se za novi objekat. Rečnik `b.__dict__` otkriva da atributi novog objekta ukazuju na koordinate, te da su im imena modifikovana kako bi se obeshrabrio direktan pristup. Ipak, i ovde se vrednost privatnog

atributa *može* promeniti sa `a._Tačka_1k__x = -5`. Klijent opet sam snosi posledice za eventualno narušavanje stanja objekta jer se ne može reći da je promena nastala iz nehata.

Mehanizam sakrivanja informacija u Pajtonu svodi se na konvenciju: atributi i metode, čija imena počinju sa `_` ili `__`, smatraju se privatnim detaljima implementacije. Konvencija se može narušiti na sopstvenu odgovornost!

Problem 10.2 — Vicomat. Kreirati klasu koja reprezentuje hipotetički automat za prodavanje viceva - *vicomat*. Prilikom kreiranja vicomata navodi se cena po vicu (5, 10 ili 20 dinara), a automat se puni vicevima iz tekstualne datoteke. U eksploataciji, vicomat prima komande za prikaz cene, ubacivanje novca (apoeni od 5, 10 i 20 dinara), prikaz vica na ekranu automata (ako je korisnik ubacio dovoljno novca) i vraćanje kusura (ako postoji). Vicevi se biraju po slučajnom principu. Testirati rad vicomata simuliranjem jedne kupoprodajne sesije. ■

Rešavanje postavljenog problema započinje analizom atributa koji opisuju jedan vicomat. Automat treba da sadrži kolekciju koja čuva viceve kao tekstualne objekte pa se za prvi atribut usvaja lista viceva: `__vicevi`. Funkcijom `choice()`, iz modula `random`, može se odabrati slučajni vic iz liste.[9] Drugi nezaobilazni atribut je celobrojni objekat `__cena`. Prilikom kreiranja, u konstruktoru se navodi cena po kojoj će se vicevi prodavati (5, 10 ili 20 dinara). Vicomat u centru grada može prodavati viceve po višoj ceni od onog koji se nalazi na periferiji. Konačno, vicomat treba da ima i atribut koji čuva informaciju o trenutno ubačenom iznosu od strane potencijalnog kupca: `__iznos`. Atributi, poput kolekcije `__vicevi`, *mogu* da sadrže *više* drugih objekata. Na taj način omogućeno je formiranje *složenog objekta* iz već postojećih komponenti o čemu će više reči biti u sledećoj glavi.

Jedna kupoprodajna sesija odvija se na sledeći način: pošto se *informiše* o ceni vica, kupac *ubacuje* 5, 10 ili 20 dinara i na taj način uvećava `__iznos` za vrednost ubačenog novca. Novac se može ubacivati u automat više puta uzastopce, ako treba da se dostigne cena vica, ili ako se kupuje veći broj viceva. Kada kupac aktivira metod za *kupovinu*, proverava se da li je postignuti `__iznos` dovoljan (veći ili jednak `__cena`). Ako jeste, `__iznos` se umanjuje za `__cena`, a vic se prikazuje na ekranu automata (ili štampa na papir). U suprotnom, kupac se obaveštava koliko mu fali do potrebnog iznosa. U svakom trenutku može se tražiti *povraćaj* novca – `__iznos` se postavlja na nula, a dotadašnja vrednost ubačenog novca vraća se kao kusur.

Iz prethodne analize uočava se potreba za sledećim metodama koje *enkapsuliraju* traženo ponašanje automata: `cena_vica()`, `ubaci_novac()`, `kupi_vic()` i `vrati_kusur()`. Kako se sve neophodne operacije obavljaju preko pomenutih metoda,

[9] Izbor se može napraviti i pristupanjem elementu liste sa indeksom koji se generiše kao pseudoslučajni broj iz $[0, n-1]$ (n - ukupan broj viceva u listi).

atributi se sakrivaju od kupca ispod nevidljive barijere objekta (slika 10.2). Kako bi se olakšalo razumevanje implementacije klase, prvo će biti prikazano rešenje koje podrazumeva da se vicevi *već* nalaze u listi:

```
from random import choice
class Vicomat:
    ''' Klasa kojom se modelira ponašanje vicomata '''

    @staticmethod
    def iznos(iznos): # vraća iznos kao int 5, 10 ili 20

        if iznos != 5 and iznos != 10 and iznos != 20:
            raise Exception('Vicomat prima 5, 10 ili 20 din.')
        return int(iznos)

    def __init__(self, cena):

        self.__cena = Vicomat.iznos(cena)
        self.__vicevi = ['prvi vic', 'drugi vic', 'treći vic']
        self.__iznos = 0

    def cena_vica(self): # prikazuje cenu jednog vica

        print('Cena vica je', self.__cena, 'din.')

    def ubaci_novac(self, iznos): # ubacuje 5, 10 ili 20 din.

        self.__iznos += Vicomat.iznos(iznos)
        print('Do sada ste ubacili', self.__iznos, 'din.')

    def kupi_vic(self):  # prikaži vic, ako iznos >= cena.

        if self.__iznos >= self.__cena:
            self.__iznos -= self.__cena
            print(choice(self.__vicevi))
            print('-------')
            print('Ostalo je još', self.__iznos, 'din.')
        else:
            print('Fali', self.__cena - self.__iznos, 'din.')

    def vrati_kusur(self): # vraća kusur
```

```
38
39          kusur = self.__iznos
40          self.__iznos = 0
41          return kusur
```

Metoda `iznos()` proverava da li vrednost navedenog argumenta predstavlja regularni apoen (r8). Ako je argument pogrešan, generiše se izuzetak sa odgovarajućom porukom (r9). U suprotnom, vraća se celobrojni objekat čija je vrednost jednaka ulaznom parametru funkcije (r10). Eksplicitna konverzija u ceo broj je *neophodna* ako je argument realni broj poput 5.0, 10.0 ili 20.0. Logički uslov iz r8 tada je ispunjen (na primer, 5.0 je jednako sa 5). Metoda se deklariše kao statička jer se ne vezuje ni za jedan konkretan objekat iz klase. Upotrebljena je na dva mesta: u inicijalizatoru, prilikom postavljanja sakrivenog atributa `__cena` (r14), kao i prilikom uvećavanja iznosa (r24), u metodi `ubaci_novac()`. Lokalna promenljiva `kusur` (r39) omogućava da se uloženi novac vrati i, u isto vreme, atribut `__iznos` postavi na nula. Slede primeri korišćenja klase `Vicomat`:

```
>>> from vicevi import Vicomat
>>> perica = Vicomat(10.1)
Traceback (most recent call last):
  File "<pyshell#71>", line 1, in <module>
    perica = Vicomat(10.1)
  File "C:\programi\vicevi.py", line 14, in __init__
    self.__cena = Vicomat.iznos(cena)
  File "C:\programi\vicevi.py", line 9, in iznos
    raise Exception('Vicomat prima 5, 10 ili 20 din.')
Exception: Vicomat prima 5, 10 ili 20 din.
>>> perica.cena_vica()
Traceback (most recent call last):
  File "<pyshell#72>", line 1, in <module>
    perica.cena_vica()
NameError: name 'perica' is not defined
```

Prilikom pokušaja kreiranja automata sa nedozvoljenom cenom za vic, metoda `__init__()` prekida se u r14, jer se, prilikom postavljanja sakrivenog atributa `__cena`, u metodi `iznos()` generiše izuzetak (r9). Kako `Vicomat()` automatski poziva magičnu metodu za inicijalizaciju, to se konstruktor ne izvršava do kraja pa objekat `perica` nije ni kreiran. Ovo se vidi pri pozivu `perica.cena_vica()`, kada se generiše novi izuzetak tipa `NameError`. Uspešno kreiranje objekta i sekvenca poziva koja simulira jednu kupoprodajnu sesiju data je ispod:

```
>>> perica = Vicomat(10)
>>> perica.cena_vica()
```

```
Cena vica je 10 din.
>>> perica.kupi_vic()
Fali 10 din.
>>> perica.ubaci_novac(10)
Do sada ste ubacili 10 din.
>>> perica.ubaci_novac(20)
Do sada ste ubacili 30 din.
>>> perica.kupi_vic()
drugi vic
-------
Ostalo je još 20 din.
>>> perica.kupi_vic()
prvi vic
-------
Ostalo je još 10 din.
>>> perica.vrati_kusur()
10
>>> perica.kupi_vic()
Fali 10 din.
```

Prilikom realizacije verzije koja učitava viceve iz tekstualne datoteke, neophodno je promeniti *samo* metodu `__init__()`. Inicijalizator treba da ima dodatni ulazni parametar koji predstavlja putanju do datoteke sa vicevima. Način parsiranja teksta u datoteci zavisi od *formata* po kome su vicevi zapisani, a koji se definiše od strane programera klase `Vicomat`. Može se pretpostaviti da su vicevi zapisani jedan iza drugog, pri čemu se između dva vica nalazi *jedan* prazan red. Ovaj format omogućava da pojedinačni vicevi mogu da sadrže više redova teksta, što i jeste najčešći slučaj. Pretpostaviti da datoteka ima sledeći izgled (ViRj označava j-ti red i-tog vica):

```
V1R1
V1R2

V2R1
V2R2
```

Ako se sadržaj gornje datoteke posmatra kao jedna tekstualna sekvenca, onda ona ima sledeći izgled: `'V1R1\nV1R2\n\nV2R1\nV2R2'`. Ovde `'\n'` označava specijalnu sekvencu koja označava prelazak u novi red. Ako `tekst` predstavlja sadržaj tekstualne sekvence, onda se sa `tekst.split('\n\n')` može dobiti lista sa pojedinačnim vicevima (videti glavu 5.1.4):

Program 10.2 — Vicomat.

```
1 from random import choice
```

```
2  class Vicomat:
3      ''' Klasa kojom se modelira ponašanje vicomata '''
4  
5      @staticmethod
6      def iznos(iznos): # vraća iznos kao int 5, 10 ili 20
7  
8          if iznos != 5 and iznos != 10 and iznos != 20:
9              raise Exception('Vicomat prima 5, 10 ili 20 din.')
10         return int(iznos)
11 
12     def __init__(self, datoteka, cena):
13 
14         self.__cena = Vicomat.iznos(cena)
15         self.__iznos = 0
16         # učitavanje viceva iz datoteke
17         try:
18             with open(datoteka, encoding = 'utf8') as ulaz:
19                 tekst = ulaz.read()
20             self.__vicevi = [v for v in tekst.split('\n\n')]
21         except:
22             raise Exception('Problem pri učitavanju datoteke!')
23 
24     # ostale metode kao u prethodnom listingu
```

Datoteka sa vicevima učitava se u celosti u bloku naredbe `with` koja omogućava njeno automatsko zatvaranje pri napuštanju bloka (r18, 19). Uočiti izgled petlje `for` iz r20. Metoda `split()`, kojom se kreira lista viceva, poziva se *samo jednom* pri ulasku u petlju, kada se izračunava izraz kojim se definiše kolekcija kroz koju treba proći. Evo i vica o Perici koji je kupljen za 10 dinara:

```
>>> from vicevi import Vicomat
>>> perica = Vicomat('c:\\papers\\vicevi.txt', 10)
>>> perica.ubaci_novac(10)
Do sada ste ubacili 10 din.
>>> perica.kupi_vic()
U restoranu, Pericin otac se obraća konobaru:
- Zapakujte nam ovo meso što nismo pojeli pa da ponesemo psu.
A Perica će k'o iz topa:
- Tata, zar ćeš mi kupiti psa?!
-------
Ostalo je još 0 din.
```

10.2 Kompozicija i agregacija

Objekti u realnom svetu najčešće su *sastavljeni* od drugih objekata. Na primer, svaka instanca klase `Čovek` sastoji se od tačno po jednog objekta iz klasa `Glava`, `Vrat` i `Trup`. Dodatno, svaki čovek sadrži nula, jednu ili dve instance klase `Ruka`, odnosno `Noga`. Opet, svaka glava mogla bi se sastojati od delova poput ušiju, kose, lica i slično. Naravno, moguće su i druge vrste *dekompozicije* čoveka na potrebne sastavne delove, što zavisi od konkretnog modela koji se izučava.

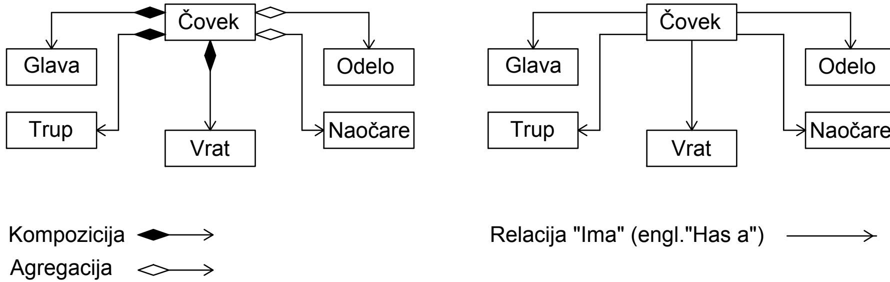
U klasnom modelu posmatrane pojave, procesa ili realnog sistema, objekti sastavni delovi predstavljeni su kao *posebne* osobine. Tako bi se u klasi `Čovek`, pored osobina poput visine ili težine, našle i osobine kao što su leva ili desna ruka, glava, trup i druge. U Pajtonu, visina i težina bile bi predstavljene realnim atributima iz predefinisane klase `float`. Sa druge strane, leva (desna) ruka može se predstaviti atributom iz klase `Ruka` koja se *posebno* definiše na način opisan u prethodnoj glavi.

> ! Do kog nivoa detalja se ide u procesu dekompozicije, zavisi od primene. Klasni model treba da bude *što jednostavniji*, a opet da pokriva *sve željene* osobine i ponašanje. Fleksibilnost objektno orijentisanog pristupa ogleda se u tome da se, po potrebi, mogu dodavati novi atributi (objekti delovi) i proširivati skup metoda koje čine ponašanje objekata iz klase.

Treba uočiti da se navedeni objekti, koji čine konkretnog čoveka, *ne mogu* posmatrati odvojeno od njegovog postojanja. Objekat klase `Glava` nema mnogo smisla sam za sebe, sem ukoliko se ne posmatra kao sredstvo za zaplašivanje pobunjene mase! Zbog toga se između klasa `Čovek` i `Glava` može uspostaviti *jaka* veza koja se, u terminologiji objektno orijentisane analize, naziva *kompozicija*. Jedan objekat klase `Čovek` komponovan je od navedenih objekata delova. On ne može postojati bez njih, kao što ni oni nemaju mnogo smisla bez njega!

Klasa `Čovek` mogla bi da sadrži i atribute poput odela ili naočara. Ovi atributi bili bi predstavljeni objektima odgovarajućih klasa (`Odelo` i `Naočare`). Za razliku od glave ili trupa, odelo ili naočare mogu da postoje *nezavisno* od svog vlasnika. Na primer, kada je vlasnik u inostranstvu, naočare i odelo može nositi njegov rođak. Slično, objekat iz klase `Akumulator` pripada određenom automobilu, ali se može zameniti i staviti u druga kola. Štaviše, akumulator je, dok nije dospeo u automobil, postojao nezavisno na polici radnje za auto delove. Veza između klasa `Čovek` i `Odelo`, ili `Automobil` i `Akumulator`, *slabija* je nego u slučaju kompozicije i zove se *agregacija*. Svaki čovek može imati i konkretno odelo, ali to nije nužno. Isto važi za automobil i akumulator.

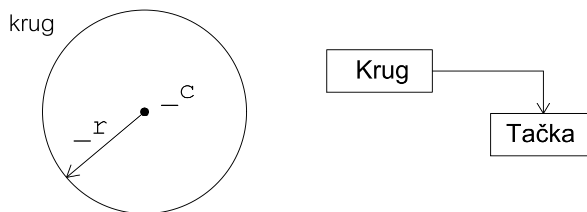
Putem kompozicije i agregacije, od prostijih objekata mogu se graditi složeniji pa se modelovanje realnog sveta značajno uprošćava. Objekat, koji u svoj sastav uključuje objekte drugih klasa, *proširuje* svoje osobine i ponašanje osobinama i ponašanjem tih klasa. Na primer, stanje nekog automobila okarakterisano je i naponom na klemama njegovog akumulatora.



Slika 10.6: Kompozicija i agregacija. U klasnom dijagramu, oba tipa veze mogu se prikazati i kao jedinstvena relacija "Ima".

Odnosi između klasa u modelu mogu se šematski prikazati uz pomoć *klasnog dijagrama* (slika 10.6). Klasni dijagram *ne sadrži* objekte, već samo klase koje se prikazuju prema konvenciji sa slike 10.1. Obično se, prilikom prikazivanja odnosa između klasa, u dijagramu izostavljaju atributi i metode, a navode samo imena klasa. Ako se želi naglasiti da objekti klase `A` sadrže u svom sastavu objekte klase `B`, bez navođenja tipa veze, onda se koristi relacija "Ima",[10] kao što je ilustrovano u desnom delu slike 10.6. Objekat klase `Čovek` u svom sastavu ima objekte klasa `Glava`, `Vrat`, `Trup`, `Odelo` i `Naočare`.

Prethodno izlaganje biće ilustrovano jednostavnim primerom koji se odnosi na klasu `Krug`. Ova klasa reprezentuje krugove u ravni. U primeru se koristi već poznata klasa `Tačka` koja može poslužiti za predstavljanje *centra* proizvoljnog kruga. Pored centra (`_c`), svaki krug opisan je i realnim atributom koji se odnosi na poluprečnik (`_r`). Očigledno, između klasa `Krug` i `Tačka` važi relacija "Ima" (slika 10.7). Tačnije, ova relacija ima prirodu *kompozicije* jer proizvoljan krug *ne može* biti jednoznačno određen bez centra. Prilikom formiranja klasnog dijagrama, *uobičajeno* je da se prikazuju *samo* korisnički definisane klase, dok se predefinisani tipovi ne prikazuju. U suprotnom, između klasa `Krug` i `float` bila bi ucrtana veza tipa kompozicije.



Slika 10.7: Klase `Krug` i `Tačka`: svaki krug, pored poluprečnika, u svom sastavu ima jedan objekat klase `Tačka`.

[10] Engl. *Has-a*.

Reprezentacija kruga iz primera *razlikuje se* od geometrijskog poimanja koje podrazumeva beskonačno mnogo tačaka. Može se pretpostaviti da se klasa koristi u okviru programa za crtanje u ravni, poput AutoCAD-a, baziranom na *vektorskoj grafici*. U vektorskom pristupu, grafički objekti poput duži, kruga ili mnogougla, predstavljaju se karakterističnim osobinama koje omogućavaju da se objekat iscrta u realnom vremenu pomoću jednačina analitičke geometrije. Sa druge strane, u *rasterskom pristupu*, slika se pamti kao matrica određene rezolucije, pri čemu se njeni pojedini elementi (*pikseli*) reprezentuju nizovima bajtova. Niz bajtova pridružen pikselu odnosi se na njegovu boju. U slučaju crno-bele slike dovoljan je jedan bit - ovako zapamćena slika naziva se još i *bit mapa*. Sledi realizacija klase `Krug`:

```
1  from tacke import Tačka
2  from random import uniform
3  # definiše apstraktni tip koji predstavlja sve krugove u ravni
4  class Krug:
5      ''' Predstavlja krugove u ravni definisane
6          sa centrom i poluprečnikom'''
7
8      PI = 3.14159265359 # klasni atribut, konstanta
9
10     def __init__(self, x = 0, y = 0, r = 1):
11
12         self._c = Tačka(x, y) # definisanje centra kao tačke
13         self._r = r           # definisanje poluprečnika
14
15     # tekstualna reprezentacija objekta.
16     def __str__(self):
17
18         return f'c= {str(self._c)}\nr= {self._r:<f}'
19
20     # ispituje da li je tačka t u krugu
21     def unutra(self, t):
22
23         return self._c.rastojanje_do(t) <= self._r
24
25     # obim i površina
26     def obim_površina(self):
27
28         return (2 * Krug.PI * self._r, self._r**2 * Krug.PI)
29
30 # test program
```

```
31 if __name__ == '__main__':
32 
33     # pravi jedinični krug k, ispisuje njegove karakteristike
34     k = Krug()
35     print(k)
36     o, p = k.obim_površina()
37     print(f'o= {o:<f}\np= {p:<f}')
38     # generiše 7 sl. tačaka, ispisuje da li su u k ili ne
39     for i in range(7):
40         tt = Tačka(uniform(-2, 2), uniform(-2, 2))
41         if k.unutra(tt):
42             print('u:', tt)
43         else:
44             print('van:', tt)
```

Klasa `Krug` započinje definisanjem klasnog atributa koji se odnosi na vrednost broja π (`PI`), a koji se koristi prilikom izračunavanja obima i površine kruga (r8). Klasni atribut zapisan je korišćenjem velikih slova, što po konvenciji označava *konstantu* čija se vrednost *neće* menjati u programu. Za razliku od drugih jezika, poput C++ ili Jave, Pajton *ne sprečava* promenu njene vrednosti sa `Krug.PI=...`, ali bi to bila loša programerska praksa. Prilikom kreiranja kruga navode se tri realna parametra koja označavaju koordinate centra i poluprečnik (r10). Pozivom konstruktora iz klase `Tačka` (r12), u inicijalizatoru se inicijalizuje atribut za centar kruga (`_c`) i tako, između klasa `Krug` i `Tačka`, uspostavlja relacija “Ima”. Na ovaj način, novoformirani objekat klase `Tačka` postaje *sastavni deo* kruga koji se kreira.

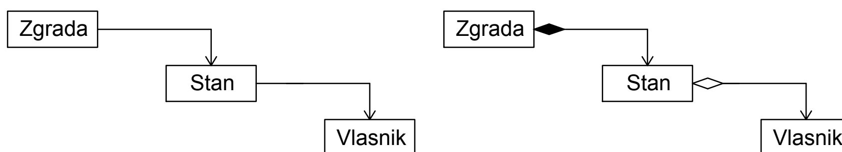
Tekstualna reprezentacija objekta formira se pomoću *već postojeće* metode iz druge klase: sa `str(self._c)`, u r18, poziva se metoda `__str__()` iz klase `Tačka`. Slično, prilikom ispitivanja da li neka tačka pripada krugu (`unutra()`, r21-23), koristi se metoda koja utvrđuje rastojanje između dve tačke (r23). Ako je rastojanje između centra kruga i navedene tačke manje od vrednosti poluprečnika, odgovor je potvrdan (`True`). Primetiti da se obim i površina kruga vraćaju kao dvoelementna torka (r28). Umesto metode `obim_površina()`, mogle su biti definisane i dve, jedna za obim i jedna za površinu.

U programu za testiranje prvo se kreira jedinični krug (r34), pa se potom ispisuju njegove karakteristike (r35-37). Uočiti konstrukciju `o, p = k.obim_površina()`, kojom se komponente dvočlane torke imenuju sa `o` i `p`. Ovo je svakako *čitljivije* od `op = k.obim_površina()`, gde bi se obimu pristupalo preko `op[0]`, a površini sa `op[1]`. U petlji se potom, na slučajan način, generišu tačke čije koordinate pripadaju intervalu $[-2,2]$ i ispisuje njihova tekstualna reprezentacija (r39-44). Tačke koje pripadaju jediničnom krugu ispisuju se sa prefiksom `'u'`, a ostale sa prefiksom `'van'`:

```
c= (0.000000, 0.000000)
r= 1.000000
o= 6.283185
p= 3.141593
van: (-1.431078, -1.909463)
van: (1.240213, -0.204659)
van: (1.740264, 1.657738)
van: (-0.851863, 1.500134)
u: (0.728460, -0.176299)
u: (0.489073, -0.124433)
van: (-1.804330, -0.896103)
```

Objektno orijentisano programiranje podseća na slaganje Lego kockica! Treba samo odabrati objekte odgovarajućih klasa i potom ih, uz dodatak sopstvenog koda, spojiti u željenu celinu. Tom prilikom, neophodno je dobro poznavanje interfejsa (skup javnih metoda) svake upotrebljene klase.

Problem 10.3 — Nekretnine. Modelovati stambenu zgradu koja se sastoji od izvesnog broja stanova. Model se koristi za potrebe sistema evidencije o nekretninama. Stanovi mogu, ali ne moraju imati vlasnika. Jedan stan ne može imati više od jednog vlasnika. Predvideti neophodne klase i snabdeti ih odgovarajućim metodama koje enkapsuliraju potrebno ponašanje, uz sakrivanje detalja implementacije. ■



Slika 10.8: Zgrada, stanovi i vlasnici: klasni dijagram. Relacija "Ima" (levo) i preciznije određenje veza (desno). Zgrada se sastoji od stanova koji mogu, ali ne moraju imati vlasnika.

Imajući u vidu da se sistem koristi za evidenciju nekretnina, *zgrada* se, pored osnovnih podataka poput adrese, može predstaviti listom *stanova* koji su njeni sastavni delovi. Očigledno, potrebno je definisati klase `Zgrada` i `Stan`. Jedan stan može se predstaviti na različite načine. U predloženom rešenju, stanovi se predstavljaju preko tri sakrivena atributa: prvi atribut odnosi se na površinu stana u kvadratnim metrima (`_m2`); drugi atribut nosi informaciju o spratu na kome se stan nalazi (`_sprat`); konačno, svakom stanu pridružen je i njegov *vlasnik* (`_vlasnik`). Kako podaci o vlasniku mogu da sadrže raznorodne informacije poput imena, matičnog broja i drugih, prirodno je da

se one *spakuju* u jedan objekat klase `Vlasnik` (slika 10.8). Na ovaj način moguće je kasnije *dodavati* nove osobine i ponašanje u odgovarajuću klasu, a da pri tome delovi koda koji se odnose na druge klase ostanu *nepromenjeni*. Na primer, u klasi `Zgrada`, umesto tekstualnog atributa `adresa`, mogao bi se dodati objekat klase `Lokacija` koji, pored adrese, sadrži i takve podatke poput udaljenosti od pojedinih bitnih objekata (škola, pijaca, centar grada i sl.).

Sa slike 10.8, treba uočiti da zgrada ne može postojati bez stanova (kompozicija), a oni najčešće imaju vlasnike koji, opet, mogu postojati nezavisno od njih (agregacija). Prilikom realizacije u Pajtonu, redosled definisanja pojedinih klasa ide od najnižih ka najvišim, prateći relaciju "Ima" u dijagramu. Ovakav pristup realizaciji (analizi) klasnog modela, u kome se prvo razmatraju sastavni delovi pa potom složenije celine, naziva se *pristup odozdo na gore*.[11] Sve klase smeštene su u modul `zgrada`:

Program 10.3 — Nekretnine.

```
1  # definiše apstraktni tip koji predstavlja vlasnike stanova
2  class Vlasnik:
3
4      def __init__(self, ime, prezime, jmbg):
5
6          self._ime = ime
7          self._prezime = prezime
8          self._jmbg = jmbg
9
10     def __str__(self):
11
12         return f'{self._ime} {self._prezime} {self._jmbg}'
13
14     def jmbg(self):
15
16         return self._jmbg
17
18     # da li ime ili prezime sadrži zadati tekst
19     def ime_sadrži(self, txt):
20
21         txt = txt.lower()
22         return txt in self._ime.lower() or \
23                txt in self._prezime.lower()
24
```

[11] Engl. *Bottom-up approach*.

```
25      # ostale metode ....
26
27  # definiše apstraktni tip koji predstavlja stanove
28  class Stan:
29
30      def __init__(self, m2, sprat):
31
32          self._m2 = m2
33          self._sprat = sprat
34          self._vlasnik = None
35
36      def __str__(self):
37
38          st = f'{self._m2}m2 spr. {self._sprat}'
39          return f'{st}\nvl. {str(self._vlasnik)}'
40
41      def vlasnik(self):
42
43          return self._vlasnik
44
45      def promeni_vlasnika(self, novi_vlasnik):
46
47          self._vlasnik = novi_vlasnik
48
49      def površina(self):
50
51          return self._m2
52
53      # ostale metode ....
54
55  # definiše apstraktni tip koji predstavlja stambenu zgradu
56  class Zgrada:
57
58      def __init__(self, adresa, stanovi):
59
60          self._adresa = adresa
61          self._stanovi = stanovi
62
63      def __str__(self):
64
65          opis = '\n\n'.join([str(s) for s in self._stanovi])
66          return f'{self._adresa}\n---\n{opis}\n---\n'
```

```
67
68     def dodaj_stan(self, stan):
69
70         self._stanovi.append(stan)
71
72     # vraća rečnik sa vlasnicima koji zadovoljavaju upit
73     # i njihovim stanovima. Ključ je matični broj vlasnika
74     def stanovi_vlasnika(self, upit):
75
76         vlasnik_stanovi = {} # rečnik sa stanovima po vlasniku
77         for stan in self._stanovi:
78             v = stan.vlasnik()
79             if v and v.ime_sadrži(upit):
80                 if v.jmbg() in vlasnik_stanovi:
81                     vlasnik_stanovi[v.jmbg()].append(stan)
82                 else:
83                     vlasnik_stanovi[v.jmbg()] = [stan]
84
85         return vlasnik_stanovi
86
87     # ostale metode ....
```

Metoda `ime_sadrži()`, u klasi `Vlasnik`, vraća `True` ako ime ili prezime vlasnika *sadrži* tekst koji se prosleđuje metodi kao argument `txt` (r19-23). Pre provere pripadnosti teksta imenu ili prezimenu sa operatorom `in`, tekstovi se prevode u mala slova korišćenjem funkcije `lower()`. U klasi `Stan`, vredna pomena je metoda `promeni_vlasnika()` kojom se menja postojeći vlasnik stana (r45-47). Prilikom kreiranja, novi stan još uvek nema vlasnika pa se atribut `_vlasnik` postavlja na `None` (r34). Uočiti da se, pri formiranju tekstualne reprezentacije za stan, implicitno poziva metoda `__str__()` iz klase `Vlasnik` (`str(self._vlasnik)`, r39).

Na vrhu hijerarhije u klasnom dijagramu nalazi se klasa `Zgrada`. Objekti ove klase kreiraju se tako što se konstruktoru prosleđuje lista sa *već formiranim* stanovima (r58). Klijent ove klase *može* proslediti i praznu listu pa se stanovi u dotičnu zgradu mogu ubacivati i putem metode `dodaj_stan()` (r68-70). Ova metoda može se koristiti i da simulira naknadnu nadogradnju, što se u Beogradu vrlo često dešava. Generisanje tekstualne reprezentacije za zgradu podrazumeva da se, pored adrese, uključe i opisi svih stanova. Zato se u metodi `__str__()`, na desnoj strani izraza u r65, definiše anonimna lista sa opisima stanova. Ovom prilikom korišćena je notacija za kreiranje liste po unapred poznatom pravilu. Lista opisa prevodi se, putem tekstualne metode `join()`, u jedinstveni tekst u kome su pojedinačni opisi razdvojeni praznim redom.

Metoda `stanovi_vlasnika()` predstavlja najsloženiji deo modula `zgrada` (r74-85). Ona vraća rečnik sa pridruživanjima oblika *jmbg vlasnika - lista njegovih stanova*. Svi vlasnici, čiji se matični brojevi nalaze u rečniku, *sadrže* tekst `upit` kao deo imena ili prezimena. Rezultujući rečnik u početku je prazan (r76). Potom se, u glavnoj petlji (r77-83), prolazi kroz sve stanove u zgradi i za svaki stan utvrđuje njegov vlasnik (r78). Ako stan ima vlasnika koji u imenu ili prezimenu sadrži tekst iz upita, onda se proverava da li on već postoji u rečniku (vlasnik je bar još jednog od prethodno obrađenih stanova). Ako postoji, u listu njegovih stanova ubacuje se tekući stan (r81). U suprotnom, formira se novo pridruživanje u rečniku koje, za datog vlasnika (njegov matični broj), vezuje jednoelementnu listu sa tekućim stanom (r83). Uočiti da metoda vraća podatke o svim vlasnicima i njihovim stanovima ako se za tekst upita izabere prazan tekst (`''`). Ovo proishodi iz činjenice da prazan tekst pripada *svakoj* tekstualnoj sekvenci. Sledi program koji ilustruje rad sa klasama iz modula `zgrada`:

```
from zgrada import Vlasnik, Stan, Zgrada

# Kreira tri vlasnika
raja = Vlasnik('Raja', 'Pataković', '123456')
gaja = Vlasnik('Gaja', 'Pataković', '123457')
vlada = Vlasnik('Vladimir', 'Putinić', '123458')

# Kreira četiri stana u zgradi P+1
s0_1 = Stan(50, 0) # sprat 0 je prizemlje
s0_2 = Stan(100, 0)
s1_1 = Stan(75, 1)
s1_2 = Stan(75, 1)
zgrada = Zgrada('Radnička 4', [s0_1, s0_2, s1_1, s1_2])

# Posle kupovine stanova
s0_1.promeni_vlasnika(raja)
s0_2.promeni_vlasnika(gaja)
s1_1.promeni_vlasnika(vlada)
s1_2.promeni_vlasnika(raja) # opet Raja

# Posle velikog interesovanja, ide nadgradnja
s2_1 = Stan(150, 2)
zgrada.dodaj_stan(s2_1)

# Ispis podataka o zgradi
print(zgrada)

```

```
28 # Prikazuje stanove vlasnika koji u imenu imaju 'aja'
29 # Prikazuje ukupan broj kvadrata po traženim vlasnicima
30 print('Vlasnici sa "aja" u imenu:\n')
31 for stanovi in zgrada.stanovi_vlasnika('aja').values():
32     total = 0
33     for stan in stanovi:
34         total += stan.površina()
35         print(stan)
36     print('Ukupno', total, 'm2\n')
```

U delu programa r1-23, ilustrovan je primer kreiranja nekoliko vlasnika, stanova i zgrade koja ih sadrži (stan od $150m^2$, na drugom spratu, nema vlasnika). Potom se ispisuju svi podaci o novoformiranoj zgradi (r26). Najinteresantniji deo programa za testiranje ispisuje podatke o stanovima onih vlasnika koji, u svom imenu ili prezimenu, imaju reč `'aja'` (r30-36). U petlji, u r31, upotrebljeno je *nadovezivanje* metoda u notaciji sa tačkom. Sa `zgrada.stanovi_vlasnika('aja')`, kao što je već diskutovano, dobija se rečnik koji sadrži tražene vlasnike i liste sa njihovim stanovima. Međutim, kako je vraćeni objekat klase `dict`, nad njim se može pozvati bilo koja metoda iz te klase, poput `values()`. Metoda `values()`, za sva pridruživanja oblika *ključ-vrednost*, vraća sve objekte vrednosti. Kao što je već pomenuto u glavi 5.2.2, metoda `keyes()` vraća sve ključeve, a metoda `items()` pridruživanja u obliku dvoelementne torke.

U svakoj iteraciji petlje (r31), promenljiva `stanovi` ukazuje na stanove konkretnog vlasnika koji zadovoljava kriterijum pretrage. U daljoj obradi (r32-36), prikazuje se informacija o svakom od njegovih stanova i pri tome računa ukupna površina koju on poseduje. Rad programa ilustrovan je ispod:

```
Radnička 4
---
50m2 spr. 0
vl. Raja Pataković 123456

100m2 spr. 0
vl. Gaja Pataković 123457

75m2 spr. 1
vl. Vladimir Putinić 123458

75m2 spr. 1
vl. Raja Pataković 123456

150m2 spr. 2
vl. None
---
```

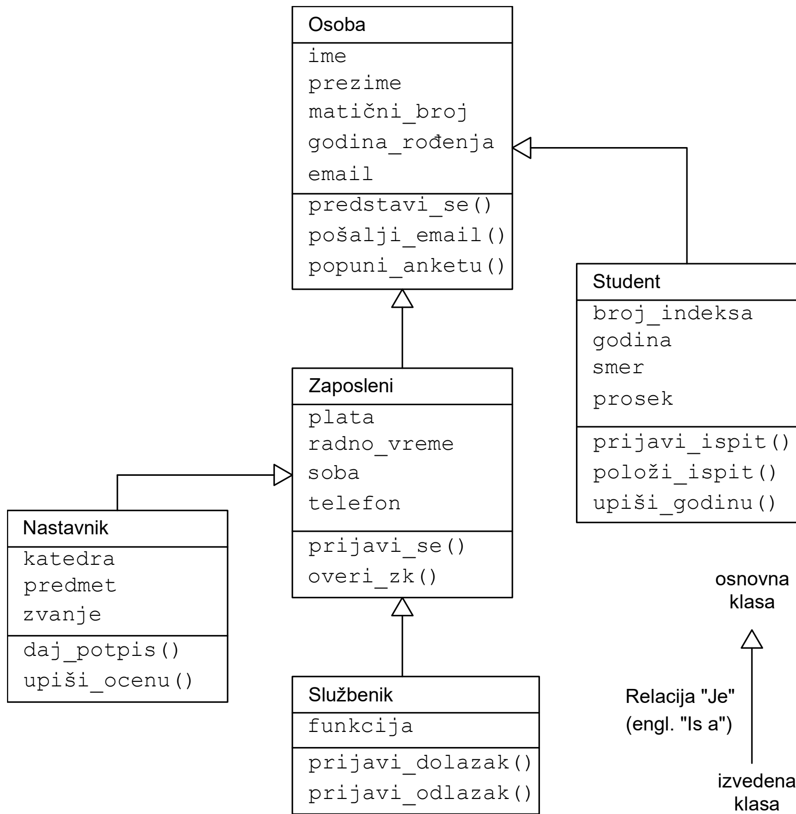
```
Vlasnici sa "aja" u imenu:

50m2 spr. 0
vl. Raja Pataković 123456
75m2 spr. 1
vl. Raja Pataković 123456
Ukupno 125 m2

100m2 spr. 0
vl. Gaja Pataković 123457
Ukupno 100 m2
```

10.3 Nasleđivanje

U toku analize složenijeg modela sa većim brojem klasa često se dešava da se, u pojedinim klasama, određene osobine i elementi ponašanja *podudaraju*. Posmatra se primer modela informacionog sistema fakulteta. U sistemu se, između ostalog, prepoznaju osobe koje su *zaposlene* na fakultetu, kao i *studenti* koji ga pohađaju. Dalje, zaposleni se mogu podeliti na *nastavno* i *nenastavno* osoblje. Pretpostavlja se da sve *osobe* u sistemu *dele* iste atribute koji se odnose na ime i prezime, matični broj, godinu rođenja, adresu fakultetske e-pošte i slično. Nastavno i nenastavno osoblje deli atribute poput visine plate, radnog vremena, broja kancelarije ili broja telefona na poslu. Sve osobe mogu da se *predstave* (kažu osnovne informacije o sebi), *pošalju* e-poštu preko fakultetskog servera, *popune* fakultetsku anketu i slično. Svi zaposleni mogu da se *prijave* na interni fakultetski sajt ili da *overe* svoje zdravstvene knjižice.

Mehanizam *nasleđivanja* omogućava da se klasa *izvede* iz druge klase i na taj način *automatski preuzme* (nasledi) njene atribute i metode. Nad objektom izvedene klase mogu se pozivati sve metode iz klase koja je poslužila za izvođenje. Klasa iz koje se vrši izvođenje naziva se *osnovna* klasa, a izvedena klasa još i *potklasa*. U primeru informacionog sistema, svi zajednički atributi i metode smešteni su u osnovnu klasu `Osoba`. Iz nje bi se mogle izvesti klase `Zaposleni` i `Student`, dok bi se iz klase `Zaposleni` mogle izvesti klase `Nastavnik` i `Službenik`. Klasni dijagram sa slike 10.9 prikazuje *hijerarhiju* klasa za pomenuti primer, uspostavljenu uz pomoć relacije nasleđivanja - “Je”.

Između osnovne (`A`) i izvedene klase (`B`) uspostavlja se relacija “Je”, u oznaci `B je A`. Zaista, svi objekti klase `B` *istovremeno su* i objekti klase `A` pošto sa njima dele iste atribute i metode. Naravno, objekti klase `B` imaju i svoje *specifične* osobine i elemente ponašanja koje ih razlikuju od ostalih objekata drugih klasa. Relacija “Je” je *tranzitivna*: iz `C je B` i `B je A`, sledi `C je A`. Zbog toga, klasa `C` nasleđuje sve osobine i ponašanja od klasa `A` i `B`. U opštem slučaju, klasa `X` nasleđuje od *svih* klasa do kojih se može stići u klasnom dijagramu, polazeći od `X` i prateći strelicu relacije “Je”.



Slika 10.9: Nasleđivanje. Studenti i zaposleni nasleđuju zajedničke osobine i ponašanje iz osnovne klase `Osoba`, a nastavno i nenastavno osoblje, iz klase `Zaposleni`.

Sa slike 10.9, uočava se da svi nastavnici i službenici preuzimaju atribute i metode kako iz klase `Zaposleni`, tako i iz klase `Osoba`.

Realizacija koncepta nasleđivanja u Pajtonu biće objašnjena na primeru pojednostavljene varijante informacionog sistema fakulteta koja obuhvata uprošćene klase `Osoba`, `Zaposleni` i `Student`:

```
1  # predstavlja sve osobe u informacionom sistemu fakulteta
2  class Osoba:
3
4      def __init__(self, ime, prezime, jmbg):
5
6          self._ime = ime
7          self._prezime = prezime
8          self._jmbg = jmbg
```

```
9
10      def __str__(self):
11
12          return f'{self._ime} {self._prezime} {self._jmbg}'
13
14      def predstavi_se(self):
15
16          print('Ja sam', self._ime, self._prezime)
17
18  # predstavlja osobe zaposlene na fakultetu
19  class Zaposleni(Osoba):    # izvođenje iz klase Osoba
20
21      def __init__(self, ime, prezime, jmbg, plata, soba, telefon):
22
23          super().__init__(ime, prezime, jmbg) # priroda osobe
24          self._plata = plata
25          self._soba = soba
26          self._telefon = telefon
27
28      def __str__(self):
29
30          return f'{super().__str__()}\n{str(self._plata)} ' + \
31                 f'soba: {self._soba} tel: {self._telefon}'
32
33      def promeni_platu(self, nova):
34
35          self._plata = nova
36
37  # predstavlja studente na fakultetu
38  class Student(Osoba):    # izvođenje iz klase Osoba
39
40      def __init__(self, ime, prezime, jmbg, indeks, smer):
41
42          super().__init__(ime, prezime, jmbg) # priroda osobe
43          self._indeks = indeks
44          self._godina = 1
45          self._smer = smer
46
47      def __str__(self):
48
49          return f'{super().__str__()}\n{self._indeks} ' + \
50                 f'godina: {str(self._godina)} smer: {self._smer}'
```

```
51
52      def upisi_godinu(self, godina):
53
54          if godina < 6 and (godina == self._godina or # max 5 god
55                            godina == self._godina + 1):
56              self._godina = godina
```

Klasa `Osoba` (r2-16) sadrži tri zajednička atributa za sve kategorije ljudi u informacionom sistemu (ime, prezime i matični broj osobe), kao i metod kojim se osoba predstavlja ako se to od nje zahteva. Klasa `Zaposleni` izvodi se iz klase `Osoba` tako što se, u potpisu klase u zagradi posle imena, navodi klasa iz koje se vrši izvođenje (r19). Isto se čini i u potpisu klase `Student` (r38). Tako svi zaposleni i studenti automatski dobijaju sve atribute i metode iz osnovne klase, bez potrebe da se oni definišu u izvedenim klasama. Efekti nasleđivanja mogu se videti iz sledeće sekvence poziva:

```
>>> import fakultet as f
>>> z = f.Zaposleni('Aca', 'Vukić', '12153', 95000, '3a', '4218-589')
>>> s = f.Student('Mitar', 'Mirić', '12131', '119/17', 'MTI')
>>> print(isinstance(s, f.Student), isinstance(s, f.Osoba))
True True
>>> z.predstavi_se()
Ja sam Aca Vukić
>>> s.predstavi_se()
Ja sam Mitar Mirić
>>> print(z)
Aca Vukić 12153
95000 soba: 3a tel: 4218-589
>>> print(s)
Mitar Mirić 12131
119/17 godina: 1 smer: MTI
```

U gornjem primeru prvo se kreiraju dva objekta izvedenih klasa, `z` i `s`. Uočiti da se, prilikom poziva konstruktora izvedenih klasa, navode *svi* potrebni podaci za inicijalizaciju, kako onog dela objekta koji predstavlja prirodu osnovne klase, tako i onog dela koji se odnosi na specifičnost potklase. Tako se, prilikom kreiranja osobe koja radi na fakultetu, pored plate, sobe i telefona, navode i ime, prezime i matični broj.

Posmatra se inicijalizator u klasi `Zaposleni`. Prvo se inicijalizuje onaj deo objekta koji se odnosi na klasu `Osoba` (r23, zaposleni *jeste* osoba). Tom prilikom poziva se eksplicitno inicijalizator iz osnovne klase, što se postiže konstrukcijom `super().__init__()`. Funkcija `super()` koristi se kada u metodi izvedene klase treba pozvati metodu osnovne klase sa *istim potpisom*. Potom se postavljaju vrednosti atributa koji se odnose na specifična svojstva zaposlenih osoba (r24-26). Slično važi i prilikom kreiranja objekata klase `Student` (uz izuzetak da se novokreirani stu-

dent uvek smešta u prvu godinu, r44). Pošto su objekti kreirani, upotrebom funkcije `isinstance()` potvrđuje se da je `s` objekat klase `Student`, ali i klase `Osoba`. Na kraju, treba uočiti da se formatirani tekstovi sabiraju poput običnih (r30-31, r49-50).

> **!** Ako se prilikom inicijalizacije objekta izvedene klase ne sprovede inicijalizacija njegove opštije prirode (sa `super().__init__()`), objekat će se naći u *nepotpuno* definisanom stanju pa se ispravan rad programa dovodi u pitanje!

10.3.1 Pretraga metoda pri pozivu. Prepisivanje metode

Metoda `predstavi_se()` dostupna je po osnovu nasleđivanja iz osnovne klase, što pokazuju uspešno realizovani pozivi `z.predstavi_se()` i `s.predstavi_se()`. U opštem slučaju, pri pozivu metode `m()` nad objektom `o` iz klase `K`, *prvo* se proverava da li je `m()` definisana u klasi `K`. Ako jeste, poziva se `m()` iz `K`. U suprotnom, proverava se da li je `m()` definisana u prvoj sledećoj klasi koja se nalazi iznad `K` u hijerarhiji nasleđivanja. Pretraga se nastavlja sve dok se ne pronađe klasa iz hijerarhije koja sadrži metodu. Ako se metoda ne pronađe ni u vršnoj klasi hijerarhije, prijavljuje se greška.

U Pajtonu 3.x, *sve* klase su *podrazumevano izvedene* iz ugrađene klase `object`! U metodama klase `object` objedinjeno je *minimalno* zajedničko ponašanje za sve objekte. Tako je podrazumevana tekstualna reprezentacija, za klase koje ne definišu metodu `__str__()`, preuzeta kao rezultat rada istoimene metode iz klase `object` (koja prikazuje ime klase i memorijsku adresu objekta).

Kreirani objekti ispisuju se uz pomoć funkcije `print()` koja implicitno poziva odgovarajuću metodu `__str__()`. Analizirajući realizaciju ispisa iz prethodnog primera, treba primetiti da je metoda `__str__()` definisana kako u izvedenim (r28-31, r47-50), tako i u osnovnoj klasi (r10-12). Ako se u potklasi definiše metoda sa *istim* potpisom kao i u osnovnoj klasi, onda ona *prepisuje* metodu osnovne klase (jer prilikom poziva, a na osnovu opisanog načina pretrage, ona ima prvenstvo u odnosu na metodu osnovne klase). Međutim, često je potrebno da, umesto prepisivanja, metoda potklase *proširi* ponašanje definisano u metodi osnovne klase tako što joj *pridodaje* akcije koje su specifične za potklasu. Tada se u prepisanoj metodi, uz upotrebu funkcije `super()`, poziva ista metoda iz osnovne klase. Ovo je učinjeno prilikom formiranja tekstualne reprezentacije (r28-31, r47-50), kada se, pored specifičnih podataka za izvedenu klasu, pridodaju i opšti podaci o osobi (`super().__str__()`). Konačno, treba uočiti da student može upisati istu (pao), ili sledeću godinu, pod uslovom da navedena godina nije veća od 5 (r52-56).

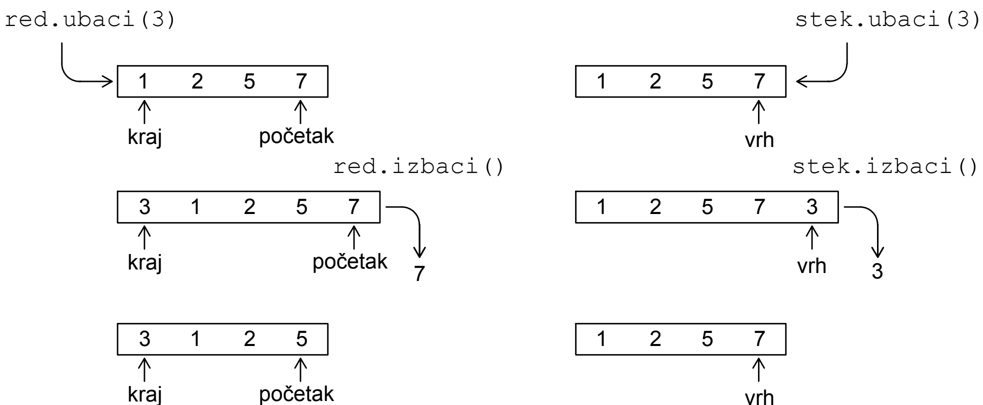
> **!** U programiranju ne treba izmišljati rupu na saksiji! Koncept nasleđivanja omogućava da se *ponovo upotrebe* ranije testirani apstraktni tipovi podataka (klase), uz eventualne dodatke koji su svojstveni konkretnoj primeni. Na ovaj način se *povećava produktivnost* u izradi novih aplikacija, odnosno *smanjuje mogućnost greške* u radu novog sistema.

Pajton dozvoljava *višestruko* nasleđivanje po kome izvedena klasa preuzima atribute i metode od više osnovnih klasa. Na primer, u aplikaciji sistema sportskih prodavnica, klasa `Bicikl` može se izvesti iz klasa `PrevoznoSredstvo`[12] i `Artikal` jer se svaki bicikl, pored osnovne namene, tretira i kao roba za prodaju. Prilikom realizacije u Pajtonu, višestruko nasleđivanje bilo bi uspostavljeno sledećim klasnim potpisom: `class Bicikl(PrevoznoSredstvo, Artikal): .`

10.3.2 Red i stek - primeri nasleđivanja

Uređena *linearna* kolekcija objekata poput liste, kod koje se novi elementi ubacuju na jednom, a stari elementi izbacuju na drugom kraju, naziva se *red*.[13] Pristupanje elementima reda realizuje se po principu FIFO (*First-In, First-Out*) koji podrazumeva da element koji je prvi ubačen prvi i napušta kolekciju (slika 10.10, levo).

Redom se može modelirati niz dolazećih zahteva koji čekaju ispred procesnog elementa (servera), kako bi bili obrađeni. Na primer, u računarskoj mreži sa više računara i mrežnim štampačem, u memoriji štampača realizuje se *red zahteva* za štampu (datoteke koje treba odštampati, a koje su poslate sa različitih računara). Softver mrežnog štampača pristupa redu zahteva po principu FIFO - prvi pristigli zahtev ide prvi na štampu, pa tako redom po redosledu pristizanja.



Slika 10.10: Red (levo): novi element (3) ide na kraj, a obrađuje se prvi koji je ušao u red (7). Prilikom ubacivanja, elementi se pomeraju za jedno mesto udesno. Stek (desno): novi element ide na vrh (3), a obrađuje se poslednji koji je došao na stek (3).

Pored reda, u programiranju se često koristi linearna kolekcija pod imenom *stek*,[14] gde se ubacivanje novih i izbacivanje starih elemenata obavlja na *istom* kraju. Pristupanje elementima steka realizuje se po principu LIFO (*Last-In, First-Out*) koji podrazumeva da se poslednji pristigli element obrađuje prvi (slika 10.10, desno). Na

[12] Po konvenciji, reči u imenu klase pišu se bez razmaka, a svaka reč započinje velikim slovom.
[13] Engl. *Queue*.
[14] Engl. *Stack*.

primer, prilikom svakog poziva funkcije, imenski prostor (tabela imena) pozivajuće celine ubacuje se na vrh steka, a novokreirani lokalni imenski prostor funkcije postaje aktivan. Kada funkcija završi sa radom, iz steka se uklanja tabela imena, čime imenski prostor pozivajuće celine ponovo postaje aktivan. Jasno je da se na ovaj način, uz pomoć steka, pamte reference na vrednosti iz različitih rekurzivnih poziva iste funkcije.

Problem 10.4 — U red i na stek. Definisati klase kojima se predstavljaju redovi i stekovi. Kreirati metode pomoću kojih se pristupa ovim kolekcijama. Predvideti i metode kojima se ispituje da li je odgovarajuća struktura prazna i koliko ima elemenata, odnosno metodu koja vraća tekstualnu reprezentaciju objekta iz klase. Prilikom implementacije, izbeći nepotrebno ponavljanje koda. Ilustrovati korišćenje obe klase u interaktivnoj sesiji, kao i na primeru provere da li je tekstualna sekvenca palindrom. ■

Red i stek mogu se realizovati uz pomoć objekta predefinisane klase `list`, zahvaljujući metodama koje umeću (`insert()`), dodaju na kraj (`append()`) ili uzimaju s kraja (`pop()`) elemente u listi (videti tabelu 5.2). Kako bi se sprečilo da se listi pristupa direktno, mimo FIFO ili LIFO redosleda, ona se može enkapsulirati unutar objekta klase `Kontejner`. Ova klasa može da sadrži i sve elemente ponašanja koji su zajednički za klase `Red` i `Stek`: utvrđivanje broja elemenata u kontejneru, testiranje da li je kontejner prazan i formiranje tekstualne reprezentacije. Na osnovu slike 10.10, jasno je da se realizacija za izbacivanje elemenata iz reda i steka *ne razlikuje* pa se može uvrstiti u klasu `Kontejner`. Sada se klase `Red` i `Stek` mogu izvesti iz klase `Kontejner` na sledeći način:

Program 10.4 — U red i na stek.

```
1  # predstavlja listu objekata sa minimalnom logikom
2  class Kontejner:
3
4      def __init__(self):
5
6          self._niz = []
7
8      def __str__(self):
9
10         txt = [str(x) for x in self._niz]
11         return ','.join(txt)
12
13     def je_prazan(self):
14
15         return len(self._niz) == 0
16
```

```
    def koliko(self):

        return len(self._niz)

    def izbaci(self):

        if self.je_prazan():
            return None
        else:
            return self._niz.pop()

# predstavlja Fifo strukturu (prvi unutra, prvi napolje)
class Red(Kontejner):

    def __init__(self):

        super().__init__()

    def ubaci(self, x):

        self._niz.insert(0, x)
        return self # omogućava ulančavanje za ubaci

# predstavlja Lifo strukturu (poslednji unutra, prvi napolje)
class Stek(Kontejner):

    def __init__(self):

        super().__init__()

    def ubaci(self, x):

        self._niz.append(x)
        return self # omogućava ulančavanje za ubaci
```

Prilikom kreiranja kontejnerskog objekta, definiše se prazna lista koja će čuvati elemente reda ili steka (r6). Atribut `_niz` sakriven je, prema konvenciji, upotrebom donje crte u imenu promenljive. U glavi 10.1.5 naglašeno je da atributi, čija imena počinju sa jednom ili dve donje crte, predstavljaju detalje implementacije klase, s tim da je varijanta sa dve donje crte nešto restriktivnija kada se pristupa u notaciji sa tačkom. U slučajevima kada se primenjuje koncept nasleđivanja, imena iz osnovne klase koja

počinju sa *dve* donje crte *nisu direktno* vidljiva u izvedenoj klasi. Na primer, ako se u osnovnoj klasi definiše atribut `__niz`, onda mu se u metodama izvedene klase *ne može* pristupati sa `self.__niz`. Zbog toga se, u situacijama kada se *planira* nasleđivanje iz klase, sakrivanje njenih atributa obavlja upotrebom *jedne* donje crte, kao što je učinjeno u gornjem primeru.

Prilikom formiranja tekstualne reprezentacije kontejnerskog objekta, primenjena je notacija za formiranje liste po unapred zadatom pravilu (r10). Lista `txt` sastoji se od tekstualnih opisa objekata iz kontejnera. pri čemu će ovi opisi biti razdvojeni zapetama u jedinstvenom tekstu (r11). U metodi `izbaci()` se proverava da li je lista prazna pa ako nije, *izbacuje* se poslednji element koji predstavlja povratnu vrednost (r26). Ako je lista prazna, vraća se `None`. Ovde treba uočiti *interno* pozivanje metode od strane druge metode iz iste klase (r23). Ako se neka metoda u klasi poziva *isključivo* interno, onda je *treba sakriti* upotrebom simbola donje crte – očigledno, takva metoda pripada implementaciji, a ne interfejsu klase.

Klase `Red` (r29-38) i `Stek` (r41-50) izvode se iz klase `Kontejner`. One sadrže implementacije samo onih metoda koje ih čine *posebnim* u odnosu na objekte kontejnere. Pored inicijalizatora koji poziva svog parnjaka iz osnovne klase (r33, r45), u ovim klasama realizuje se metoda `ubaci()` kojom se određuje priroda kolekcije (red ili stek). U slučaju reda, prilikom umetanja na početnu poziciju, svi elementi se pomeraju za jedno mesto udesno - metoda liste `insert()` (r37). Primetiti da je atribut `self._niz` vidljiv iako nije definisan u ovoj klasi (efekat nasleđivanja). Prilikom ubacivanja elemenata u stek,[15] koristi se metoda liste `append()` (r49). U obe klase, metoda `ubaci()` vraća referencu na samu instancu (red ili stek), što se postiže izrazom `return self`. Ovo je učinjeno kako bi se, po ubacivanju novog elementa u red (stek), moglo primeniti ulančavanje u notaciji sa tačkom. Tako se, u okviru jednog izraza, može ubaciti više elemenata odjednom, kao što je ilustrovano u sledećem primeru:

```
>>> import redstek as rs
>>> red = rs.Red()
>>> isinstance(red, rs.Kontejner)
True
>>> red.je_prazan()
True
>>> red.ubaci(1).ubaci(2).ubaci(3)
<redstek.Red object at 0x0000026E733F1A20>
>>> print(red)
3,2,1
>>> red.izbaci()
1
>>> print(red)
3,2
```

[15] Kaže se i “na stek” jer se elementi slažu jedan na drugi, s tim da poslednji ide na vrh

```
>>> red.koliko()
2
>>> stek = rs.Stek()
>>> None == stek.izbaci()
True
>>> stek.ubaci(1).ubaci(2).ubaci(3)
<redstek.Stek object at 0x0000026E733F1978>
>>> print(stek)
1,2,3
>>> stek.izbaci()
3
>>> stek.izbaci()
2
>>> print(stek)
1
```

Na osnovu analize prethodnog koda uočava se da postupak izbacivanja elementa iz obe kolekcije ima reda složenosti O(1). Sa druge strane, složenost pri ubacivanju elementa u red iznosi O(n) - prilikom ubacivanja na prvu poziciju treba pomeriti n elemenata za jedno mesto nadesno. Kako se u ovoj situaciji ne razlikuju najpovoljniji i najnepovoljniji slučaj, može se pisati i $\Theta(1)$, odnosno $\Theta(n)$ - videti glavu 7.2. Ubacivanje novog elementa u stek, budući da on uvek ide na vrh, ima red složenosti $\Theta(1)$. Algoritam koji testira da li je unešeni tekst palindrom (čita se isto sa obe strane), izložen je u problemu 5.2. Sledi prikaz postupka realizovanog pomoću reda i steka:

```
1  from redstek import Red, Stek
2  tekst = input('Unesite tekst ')
3
4  # test za palindrom, uz pomoć reda
5  red = Red()
6  for s in tekst:
7      red.ubaci(s)
8
9  if tekst == str(red).replace(',', ''):
10     print(tekst, 'je palindrom (red)')
11 else:
12     print(tekst, 'nije palindrom (red)')
13
14 # test za palindrom, uz pomoć steka
15 stek = Stek()
16 for s in tekst:
17     stek.ubaci(s)
```

```
18
19 obrnut = [stek.izbaci() for i in range(len(tekst))]
20 if tekst == ''.join(obrnut):
21     print(tekst, 'je palindrom (stek)')
22 else:
23     print(tekst, 'nije palindrom (stek)')
```

U prvoj varijanti, pojedinačni karakteri ubacuju se u `red` tako da prvi karakter odlazi na poslednje, a poslednji na prvo mesto u listi kojom je red realizovan (r6-7). Na ovaj način, čitajući karaktere u redu s leva na desno, dobija se obrnuti zapis unetog teksta. Poređenje originalnog teksta i njegovog obrnutog zapisa obavlja se u r9. Treba primetiti da su u tekstualnoj reprezentaciji, onako kako je to definisano u klasi `Red`, pojedinačna slova razdvojena zapetom. Na primer, za ulazni tekst `'Pajton'`, tekstualna reprezentacija reda glasi `'n,o,t,j,a,P'`. Zato se zapete izbacuju primenom tekstualne metode `replace()` koja ih menja praznim tekstom (r9).

U varijanti sa stekom, pojedinačni karakteri ubacuju se u `stek` tako da prvi karakter odlazi na prvo, a poslednji na poslednje mesto u listi kojom je stek realizovan (r16-17). Tekstualna reprezentacija steka razlikuje se od originalnog teksta samo po tome što se između svaka dva karaktera nalazi zapeta. Na primer, za ulazni tekst `'Pajton'`, tekstualna reprezentacija steka glasi `'P,a,j,t,o,n'`. Zato se ulazni tekst "obrće" tako što se karakteri izbacuju sa steka i ubacuju u listu `obrnut` (r19). Skreće se pažnja na notaciju kojom se lista kreira po unapred poznatom pravilu. Finalna provera obavlja se u r20, gde se pojedinačni karakteri obrnutog teksta spajaju u tekstualnu sekvencu uz pomoć metode `join()`. Interesantno je primetiti da, iako izgleda komplikovanije, varijanta sa stekom radi *brže* za duže sekvence, što je posledica nižeg reda složenosti prilikom ubacivanja u stek ($\Theta(1)$, u odnosu na $\Theta(n)$ kod reda).

ŠTA JE NAUČENO

- Realni svet može se modelovati objektima koji su opisani osobinama i ponašanjem. Skup svih objekata sa istim osobinama i ponašanjem naziva se klasa. Objekti se nazivaju i instancama klase.

- Objekat se, u svakom trenutku, nalazi u određenom stanju koje je definisano vrednostima svih njegovih osobina. Stanje objekta može se promeniti delovanjem akcija koje spadaju u njegovo ponašanje. Akcija može biti pokrenuta nad objektom od spolja ili samoinicijativno.

- Objektno orijentisani program realizuje se kroz međusobnu interakciju programskih objekata. Oni predstavljaju apstrakcije realnih objekata čije su osobine realizovane atributima, a akcije, koje čine ponašanje, metodama.
- Atributi se realizuju preko promenljivih različitih tipova. Metode predstavljaju specijalne funkcije koje se pokreću nad objektima u cilju promene vrednosti njihovih atributa (stanje objekta).
- Princip enkapsulacije podrazumeva da su svi neophodni podaci, kao i operacije koje se nad njima obavljaju, spakovani u jedinstvenu celinu - objekat.
- Sa objektom iz klase može se raditi samo ono što je definisano u interfejsu klase. Interfejs predstavlja skup javnih metoda i atributa. Detalji implementacije, koji se odnose na to kako metode rade i kako su podaci predstavljeni, sakriveni su od spoljnjeg sveta.
- Sakrivanje informacija podrazumeva da se eksplicitno može naznačiti koji atributi i metode pripadaju interfejsu, a koji implementaciji. U Pajtonu se sakrivanje realizuje putem konvencije o imenovanju atributa i metoda. Nepoštovanje konvencije može dovesti do narušavanja stanja objekta.
- Princip enkapsulacije omogućava da se kompleksni sistemi jednostavnije razvijaju i održavaju jer postoji jasna separacija nadležnosti između objekata različitih klasa.
- Pored atributa instanci, koji se vezuju za pojedinačne objekte, postoje i klasni atributi koji se vezuju za klasu. Nasuprot atributu instance koji ima različite vrednosti od objekta do objekta, klasni atribut ima jedinstvenu vrednost deljenu između svih objekata.
- Statičke metode ne vezuju se za konkretne objekte, već predstavljaju funkcije koje po svojoj nameni prirodno pripadaju klasi. One ne menjaju vrednosti atributa instanci.
- Metode koje interpreter najčešće poziva indirektno, kada se steknu odgovarajući uslovi u programu, nazivaju se magične. Na primer, inicijalizator `__init__()` poziva se iz konstruktora da postavi novokreirani objekat u početno stanje. Metoda `__str__()` generiše tekstualnu reprezentaciju objekta i poziva se unutar funkcija `print()` i `str()`.
- Preopterećenje operatora podrazumeva da se ponašanje pojedinih operatora, poput + ili `*`, menja kada se menjaju klase odgovarajućih operanada. Ponašanje se definiše putem odgovarajućih magičnih metoda.
- Objekti mogu da sadrže druge objekte istih ili različitih klasa. Objekti delovi realizuju se kao atributi odgovarajućih tipova.
- Ako objekat klase `A` ne može da postoji bez objekta klase `B` u njegovom sastavu, onda između `A` i `B` postoji jaka veza tipa kompozicije. U slučaju da objekti klase `B` mogu da postoje i nezavisno, veza je slabijeg karaktera - agregacija. U oba slučaja kaže se da `A` ima u svom sastavu `B` - relacija "Ima": `A ima B`.

- Odnosi između klasa prikazuju se klasnim dijagramom. Klase se predstavljaju kao pravougaonici sa imenom, spiskom atributa i spiskom metoda. Relacije između klasa, poput "Ima", crtaju se odgovarajućim tipovima linija.

- Princip nasleđivanja omogućava izvođenje novih klasa iz već postojećih. Izvedena klasa, pored svojih specifičnosti, automatski preuzima atribute i metode iz osnovne klase. Nasleđivanjem se podstiče ponovna upotrebljivost već razvijenih komponenti, što povećava produktivnost u programiranju.

- Između izvedene klase `A` i osnovne klase `B` uspostavlja se relacija "Je" - `A je B`. U Pajtonu, izvedena klasa može se izvesti iz većeg broja osnovnih klasa.

- Objektno orijentisano programiranje podseća na slaganje lego kockica. Nove tipove kockica treba razvijati od početka ili putem nasleđivanja samo ako na tržištu ne postoje već gotove.

11. Pajton za inženjere: NumPy i Matplotlib

U dosadašnjem izlaganju čitaoci su se upoznali sa osnovnim elementima jezika Pajton i najvažnijim principima proceduralnog i objektno-orijentisanog programiranja. Kako je knjiga prvenstveno namenjena budućim inženjerima, u ovoj glavi predstavljeni su paketi *NumPy* i *Matplotlib*. Oni omogućavaju efikasnu realizaciju različitih numeričkih algoritama koji procesiraju *višedimenzionalne nizove* brojeva (NumPy), kao i grafički prikaz podataka (Matplotlib). Budući da su ovi paketi besplatni, njihova kombinacija često se koristi kao *alternativa* za komercijalno naučno-tehničko okruženje i programski jezik *Matlab*.

11.1 Procesiranje višedimenzionalnih nizova brojeva - NumPy

U mnogim naučno-tehničkim problemima potrebno je obrađivati složene strukture brojeva koji opisuju sisteme i procese. Na primer, treba analizirati temperature registrovane u različitim vremenskim trenucima (niz brojeva), obrađivati crno-bele slike (pravougaone šeme brojeva koji nose informaciju o intenzitetu crne boje po određenom elementu slike - pikselu), ili fotografije u boji (trodimenzionalni nizovi brojeva koji predstavljaju intenzitete različitih osnovnih boja po svakom pikselu). Paket NumPy predstavlja skup klasa i funkcija za *efikasan* rad sa višedimenzionalnim nizovima brojeva[1]. U glavi 5 razmatrana je sekvencijalna kolekcija objekata tipa liste. Matrice proizvoljnih dimenzija mogu se predstaviti listama na sledeći način:

[1] U daljem tekstu, višedimenzionalni nizovi nazivaju se *matricama*, a jednodimenzionalni *nizovima*.

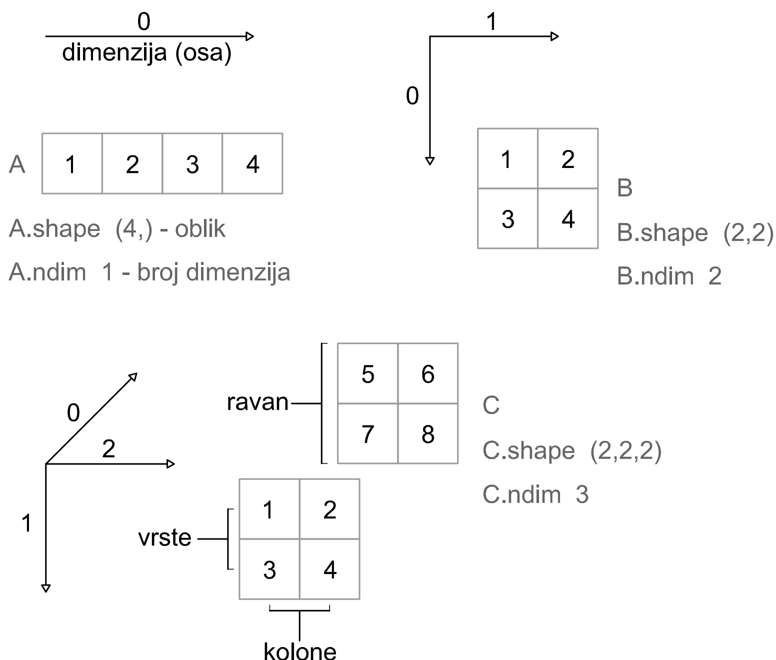
```
>>> A = [[1, 2, 3], [4, 5, 6], [7, 8, 9]] # kvadratna matrica 3x3
>>> A
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> A[2][2] # Element u preseku 3. vrste i 3. kolone
9
>>> B = [      # matrica 3x2x2
  [
    [1, 2],
    [3, 4]
  ],
  [
    [1, 0],
    [0, 1]
  ],
  [
    [2, 2],
    [1, 3]
  ]
]
>>> B[1][0][1] # 2. ravan, presek 1. vrste i 2. kolone
0
```

Predstavljanje matrice pomoću liste ima sledeće ozbiljne nedostatke: *nepostojanje* matričnih operacija u klasi `list`, *sporost* u pristupanju elementima, kao i *veliki* memorijski utrošak pri skladištenju brojnih podataka iz liste. Zato, za predstavljanje višedimenzionalnih matrica, NumPy uvodi novu strukturu podataka - objekat tipa `numpy.ndarray`:

```
>>> import numpy as np
>>> A = np.array([1, 2, 3, 4]) # 1D matrica (ili niz)
>>> A
array([1, 2, 3, 4])
>>> type(A)
<class 'numpy.ndarray'>
>>> A.ndim, A.shape, A.size, A.dtype
(1, (4,), 4, dtype('int64'))
>>> B = np.array([[1, 2], [3, 4]]) # 2D matrica
>>> B
array([[1, 2],
       [3, 4]])
>>> B.ndim, B.shape, B.size, B.dtype
(2, (2, 2), 4, dtype('int64'))
>>> C = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]]) # 3D matrica
>>> C
array([[[1, 2],
        [3, 4]],
```

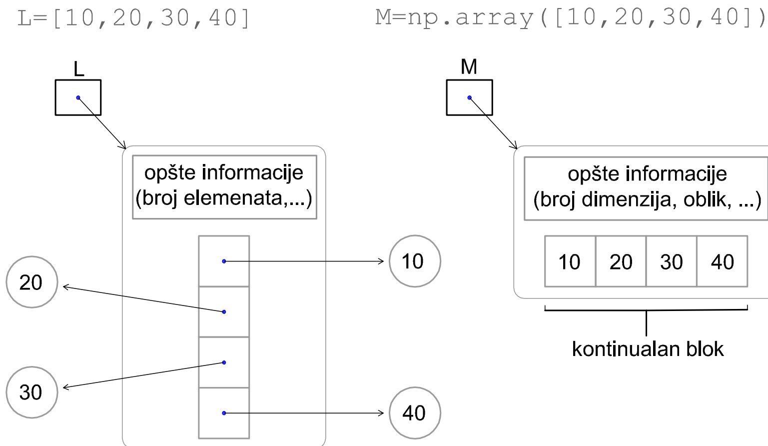
```
       [[5, 6],
        [7, 8]]])
>>> C.ndim, C.shape, C.size, C.dtype
(3, (2, 2, 2), 8, dtype('int64'))
```

Posle uvođenja paketa `numpy`,[2] jednodimenzionalna (`A`), dvodimenzionalna (`B`) i trodimenzionalna matrica `C` kreiraju se pomoću konstruktora `array()`. Konstruktoru se prosleđuje lista koja reprezentuje željenu matricu – na primer, u slučaju matrice `B`, elementi ove liste su liste koje predstavljaju njene *vrste*. Objekti klase `numpy.ndarray` opisani su sa četiri atributa: `shape` opisuje oblik matrice `X` (za `B`, dve vrste i dve kolone), `ndim` govori o broju dimenzija (jednak `len(X.shape)`), `size` daje ukupan broj elemenata (jednak proizvodu elemenata iz torke `X.shape`), a `dtype` se odnosi na njihov tip. Navedeni pojmovi ilustrovani su na slici 11.1. O načinima kreiranja matrice biće više reči u sledećem poglavlju. Na slici 11.2 prikazano je kako se liste i matrice smeštaju u memoriju.



Slika 11.1: Višedimenzionalne matrice: oblik i dimenzije (ose).

[2] Paket inicijalno nije dostupan u okviru IDLE-a, već ga treba instalirati. Na operativnom sistemu Windows, na kome je Pajton već instaliran, pokrenuti iz komandne linije `pip install numpy`. Slično, na operativnom sistemu Linux, pokrenuti iz komandne linije `python3.x -m pip install numpy` (ovde se x odnosi na instaliranu verziju Pajtona).



Slika 11.2: Memorijska organizacija liste (levo) i matrice u NumPy-ju (desno). Za razliku od liste, elementi NymPy matrice su istog tipa i smešteni su, kad god je to moguće, jedan pored drugog u kontinualnom memorijskom bloku.

Sa slike 11.2 se uočava da, za razliku od listi, matrice zauzimaju *kontinualan* prostor u radnoj memoriji. Zbog toga se, pri pozivu konstruktora ili neke druge funkcije koja vraća novoformiranu matricu, *mora* navesti *ukupan broj* i *tip* njenih elemenata! Sa druge strane, elementi liste mogu biti objekti proizvoljnih tipova. Nasuprot matricama, broj elemenata u listi obično se često menja. Zato su liste organizovane sa dodatnim nivoom indirekcije - one zapravo sadrže reference (memorijske adrese) objekata elemenata koji su *razbacani* po memoriji, ne nužno po susednim memorijskim lokacijama. Zbog dodatnog nivoa indirekcije, liste zauzimaju više prostora nego matrice. Opet, zbog dodatnog pristupa memoriji, rad sa listama je sporiji od rada sa matricama. Uzimajući u obzir da moderni procesori *keširaju* susedne podatke,[3] elementima u matrici se, u odnosu na iste podatke u listi, još više ubrzava pristup. Prednost korišćenja paketa NumPy ilustrovana je u sledećem primeru:

```
1  import numpy as np
2  from time import perf_counter
3  from random import randint
4
5  def lista_protiv_numpy_matrice(n):
6
7      A = [[randint(0, 10) for __ in range(n)] for _ in range(n)]
```

[3] Prilikom čitanja podatka iz radne memorije, procesor dohvata kontinualni blok i smešta ga u brzu procesorsku memoriju - keš (*cache memory*). Prilikom čitanja susednih podataka, nema potrebe za njihovim dohvatanjem iz radne memorije jer su oni već u kešu.

```
8       B = [[randint(0, 10) for __ in range(n)] for _ in range(n)]
9
10      # C = A + B, liste
11      s = perf_counter()
12      C = []
13      for i in range(n):
14          vrsta = []
15          for j in range(n):
16              vrsta.append(A[i][j] + B[i][j])
17          C.append(vrsta)
18      dt1 = perf_counter() - s
19      print("C=A+B: Liste. Gotovo za", dt1)
20
21      # C = A + B, NumPy matrice
22      A = np.array(A)
23      B = np.array(B)
24      s = perf_counter()
25      C = A + B
26      dt2 = perf_counter() - s
27      print("C=A+B: NumPy matrice. Gotovo za", dt2)
28
29      print("NumPy matrice brže", dt1/dt2, "puta")
30
```

```
>>> lista_protiv_numpy_matrice(100)
C=A+B: Liste. Gotovo za 0.00166 sec.
C=A+B: NumPy matrice. Gotovo za 0.00007 sec.
NumPy matrice brže 21.11 puta
>>> lista_protiv_numpy_matrice(1000)
C=A+B: Liste. Gotovo za 0.09319 sec.
C=A+B: NumPy matrice. Gotovo za 0.00259 sec.
NumPy matrice brže 35.97 puta
>>> lista_protiv_numpy_matrice(10000)
C=A+B: Liste. Gotovo za 11.12602 sec.
C=A+B: NumPy matrice. Gotovo za 0.22243 sec.
NumPy matrice brže 50.02 puta
```

Primer ilustruje sabiranje matrica predstavljenih preko listi (r7-8, r12-17) i korišćenjem objekata tipa `numpy.ndarray` (r22-23, r25). Sabiranje matrica u paketu NumPy obavlja se na prirodan način (`C = A + B`) bez upotrebe većeg broja naredbi kao u slučaju sa listama. Iz demonstracije rada programa uočava se da, sa porastom reda kvadratne matrice, raste i ubrzanje koje se postiže upotrebom paketa NumPy.

11.1.1 Kreiranje matrice

U prethodnom tekstu je pokazano kako se, pomoću konstruktora `array()`, matrica kreira iz standardne liste. Pored liste, konstruktor prima i opcioni argument `dtype`, kojim se navodi tip objekata elemenata (*skalarni* tip):

```
>>> import numpy as np
>>> A = np.array([1, 2, 3], dtype=np.float32)  # 1D-matrica (niz)
>>> print(f'{A}\n{A.shape} {A.dtype}')
[1. 2. 3.]
(3,) float32
>>> B = np.array([[1.1, 2.6, 3]], dtype=np.int32)  # 2D-matrica vrsta
>>> print(f'{B}\n{B.shape} {B.dtype}')
[[1 2 3]]
(1, 3) int32
>>> C = np.array([[1.1], [2.2], [3.6]])  # 2D-matrica kolona
>>> print(f'{C}\n{C.shape} {C.dtype}')
[[1.1]
 [2.2]
 [3.6]]
(3, 1) float64
```

Jednodimenzionalna matrica `A` predstavlja niz od tri elementa - torka `A.shape` sadrži samo jedan element (broj elemenata po jedinoj dimenziji matrice). Uočiti da su elementi matrice *konvertovani* iz celobrojnog u realni tip `np.float32`. Ovaj tip predstavlja realne brojeve pomoću 32 bita (4 bajta). Dvodimenzionalna matrica `B` je zapravo *vektor-vrsta* jer se 2D matrice opisuju listama čiji su elementi liste koje predstavljaju vrste. Iako su u polaznoj listi navedeni realni brojevi, matrica sadrži cele brojeve predstavljene sa 32 bita jer je pri kreiranju naveden tip `np.int32`. Primetiti da prilikom konverzije nije došlo do zaokruživanja, već je izvršeno *odsecanje* decimalnih mesta (`2.6` u `2`). Konačno, matrica `C` predstavlja *vektor-kolonu*. Ako se ne navede opcioni tip, konstruktor `array()` kreira matricu čiji tip elemenata zavisi od tipa elemenata ulazne liste: ako su brojevi u listi tipa `int`, onda su elementi matrice tipa `np.int64`, a ako su brojevi u listi tipa `float`, onda matrica sadrži brojeve tipa `np.float64`. U tabeli 11.1 dat je prikaz najčešće korišćenih skalarnih tipova iz paketa NumPy. Pri navođenju skalarnog tipa mogu se koristiti i ekvivalentni tipovi u Pajtonu poput `int`, `float` ili `complex`, što će se u daljem tekstu i činiti.

Pored konstruktora `array()`, paket NumPy nudi brojne funkcije za kreiranje 1D, 2D i višedimenzionalnih matrica. Od funkcija koje kreiraju nizove, ovde se pominju `arange()`, `linspace()` i `logspace()`:

```
>>> np.arange(10)
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

tip u NumPy-u	ekvivalent u Pajtonu	opis
`numpy.bool8`	`bool`	`True` ili `False`
`numpy.uint8`	–	neoznačeni 8-bitni celi brojevi
`numpy.uint16`	–	neoznačeni 16-bitni celi brojevi
`numpy.uint32`	–	neoznačeni 32-bitni celi brojevi
`numpy.uint64`	–	neoznačeni 64-bitni celi brojevi
`numpy.int8`	–	označeni 8-bitni celi brojevi
`numpy.int16`	–	označeni 16-bitni celi brojevi
`numpy.int32`	–	označeni 32-bitni celi brojevi
`numpy.int64`	`int`	označeni 64-bitni celi brojevi
`numpy.float16`	–	realni brojevi u polovičnoj tačnosti
`numpy.float32`	–	realni brojevi u jednostrukoj tačnosti
`numpy.float64`	`float`	realni brojevi u dvostrukoj tačnosti
`numpy.complex64`	`complex`	kompleksni brojevi u dvostrukoj tačnosti

Tabela 11.1: Najčešće korišćeni skalarni tipovi u NumPy-u. Odgovarajući tip bira se u zavisnosti od potrebne tačnosti i raspoložive memorije - precizniji tipovi reprezentuju se sa više bita.

```
>>> np.arange(10, dtype=float)
array([0., 1., 2., 3., 4., 5., 6., 7., 8., 9.])
>>> np.arange(5, 10)
array([5, 6, 7, 8, 9])
>>> np.arange(1, 3, 0.3)
array([1. , 1.3, 1.6, 1.9, 2.2, 2.5, 2.8])
>>> np.linspace(1, 3, 5)
array([1. , 1.5, 2. , 2.5, 3. ])
>>> np.logspace(1, 3, 5) # početak 10**1, kraj 10**3
array([  10.        ,   31.6227766 ,  100.        ,  316.22776602,
        1000.        ])
```

Funkcija `arange()` generiše niz *ekvidistantnih* brojeva ($a_{i+1} - a_i = const$) iz navedenog intervala i pri tome prima jedan (kraj intervala), dva (početak i kraj) ili tri argumenta (početak, kraj i korak *const*). Može se navesti i opcioni argument `dtype` kojim se navodi željeni tip elemenata niza. Slično ponašanje ima i funkcija `linspace()`, s tim da se ovde, umesto koraka, kao treći argument navodi broj elemenata niza. Za razliku od funkcije `arange()`, `linspace()` garantuje da će poslednji element biti uvršten u rezultujući niz. Funkcija `logspace()` slična je sa `linspace()`, osim što se primenjuje logaritamska skala sa navedenom osnovom (opcioni parametar `base`, ako se ne navede `base=10`) i važi $a_{i+1}/a_i = const$.

Uobičajene funkcije za kreiranje dvodimenzionalnih matrica su `eye()` i `diag()`:

```
>>> np.eye(3) # jedinična matrica 3x3
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
>>> np.eye(2, 3, dtype=int) # jedinična matrica 2x3
array([[1, 0, 0],
       [0, 1, 0]])
>>> np.diag([1, -1.1]) # dijagonalna matrica 2x2
array([[ 1. ,  0. ],
       [ 0. , -1.1]])
>>> np.diag([1, 2], 1) # elementi idu na 1. dijagonalu iznad glavne
array([[0, 1, 0],
       [0, 0, 2],
       [0, 0, 0]])
>>> np.diag([1, 2], -2) # elementi idu na 2. dijagonalu ispod glavne
array([[0, 0, 0, 0],
       [0, 0, 0, 0],
       [1, 0, 0, 0],
       [0, 2, 0, 0]])
>>> A = np.array([[1, 1], [2, -1]])
>>> np.diag(A) # vraća niz brojeva sa glavne dijagonale
array([ 1, -1])
>>> np.diag(A, -1) # vraća brojeve sa 1. dijagonale ispod glavne
array([2])
```

Primer pokazuje da, ako je argument za `diag()` dvodimenzionalna matrica, onda funkcija vraća niz brojeva sa glavne dijagonale (ili neke druge ako se ona specificira drugim argumentom). Od najčešće korišćenih funkcija za kreiranje matrica proizvoljnih dimenzija navode se `zeros()`, `ones()` i `random()`:

```
>>> np.zeros((2, 2), dtype=int) # 2x2
array([[0, 0],
       [0, 0]])
>>> np.zeros((2, 2, 3)) # 2x2x3
array([[[0., 0., 0.],
        [0., 0., 0.]],

       [[0., 0., 0.],
        [0., 0., 0.]]])
>>> np.ones((4,), dtype=int) # niz jedinica
array([1, 1, 1, 1])
>>> np.ones((3, 3, 3)) # 3x3x3
array([[[1., 1., 1.],
```

```
         [1., 1., 1.],
         [1., 1., 1.]],

        [[1., 1., 1.],
         [1., 1., 1.],
         [1., 1., 1.]],

        [[1., 1., 1.],
         [1., 1., 1.],
         [1., 1., 1.]]])
>>> np.random.random((2, 3)) # slučajna matrica 2x3
array([[0.1866016 , 0.34780019, 0.94597572],
       [0.5958919 , 0.72284352, 0.02520067]])
```

Funkcije `zeros()` i `ones()` formiraju matrice čiji su svi elementi jednaki nuli, odnosno jedinici. Primetiti da se oblik matrice navodi kao prvi argument u obliku torke sa brojem elemenata po svakoj od dimenzija matrice.[4] Za formiranje proizvoljne matrice čiji su elementi uniformno raspoređeni pseudoslučajini brojevi iz intervala $[0,1)$, koristi se funkcija `random()` iz modula `np.random`. O tome kako se kreira matrica sa elementima iz intervala $[a,b)$, biće reči u problemu 11.2.

Konačno, matrice se mogu kreirati i učitavanjem podataka iz tekstualne datoteke u formatu CSV, pomoću funkcije `loadtxt()`:

```
Neka je izgled datoteke "C:\matrica.txt" sledeći:
x, y, x + y
0, 0, 0
1, 1, 2
2, 2, 4
3, 4, 7

>>> A = np.loadtxt('c:\\matrica.txt', delimiter=',', skiprows=1)
>>> A
array([[0., 0., 0.],
       [1., 1., 2.],
       [2., 2., 4.],
       [3., 4., 7.]])
```

Prvi argument funkcije `loadtxt()` je obavezan i označava putanju do tražene datoteke. Opcioni parametar `delimiter` pojašnjava koji karakter ima ulogu separatora u formatu CSV, a `skiprows` navodi koliko početnih redova treba preskočiti kako bi se došlo do prve vrste matrice. Za navođenje tipa elemenata može se koristiti parametar `dtype`. Obratiti pažnju da funkcija *uvek* vraća dvodimenzionalnu matricu. Za tretiranje

[4] NumPy često, ali ne uvek, dozvoljava i upotrebu liste na mestima gde se očekuju torke. Tako se oblik matrice može navesti i u listi.

učitane pravougaone šeme brojeva kao višedimenzionalne matrice, neophodno je da se ona *preoblikuje* o čemu će više reči biti kasnije.

11.1.2 Pristup elementima matrice - osnovno indeksiranje

Za potrebe čitanja i upisa, elementima matrice `A` može se pristupiti kako pojedinačno, tako i grupno sa `A[obj]`. U zavisnosti od tipa navedenog objekta `obj`, postoje tri načina indeksiranja: *osnovno*, *napredno* i *kombinovano*. U ovoj glavi razmatra se osnovno indeksiranje.

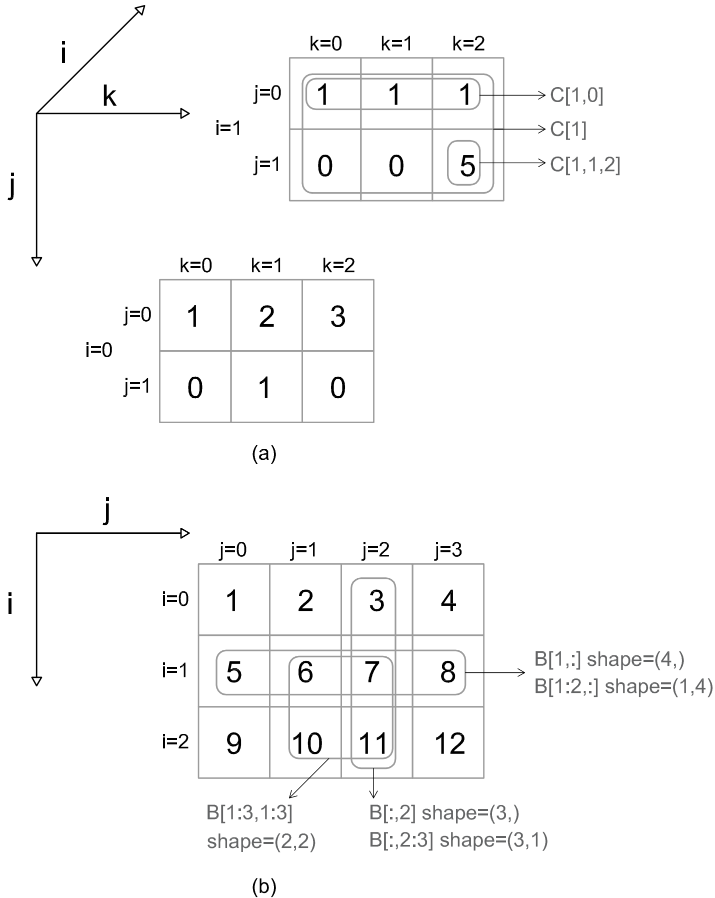
Osnovno indeksiranje podrazumeva da je objekat `obj` iz `A[obj]` celobrojan, tipa `slice` (zadaje se u obliku opsega `start:stop:korak`)[5], ili torka objekata tipa `slice` i/ili celih brojeva. Razmatraju se prvo celobrojni indeksi i indeksi definisani celobrojnim torkama:

```
>>> A, B = np.array([1, 2, 3, 4]), np.array([[1, 2], [3, 4]])
>>> print(A)
[1 2 3 4]
>>> print(A[0], A[-1]) # prvi i poslednji element niza A
1 4
>>> print(B)
[[1 2]
 [3 4]]
>>> # indeksiranje celobrojnim torkama
>>> print(B[1, 1], B[-2, -1]) # moglo je i B[(1, 1)], B[(-2, -1)]
4 2
>>> C = np.array([[[1, 2, 3], [0, 1, 0]], [[1, 1, 1], [0, 0, 5]]])
>>> print(C)
[[[1 2 3]
  [0 1 0]]

 [[1 1 1]
  [0 0 5]]]
>>> C[1, 1, 2]
5
>>> print(C[1]) # pristupa se drugoj ravni 3D matrice
[[1 1 1]
 [0 0 5]]
>>> print(C[1, 0]) # druga ravan, prva vrsta
[1 1 1]
>>> C[1, 1, 2] = 100 # dodela vrednosti
>>> print(C)
[[[  1   2   3]
  [  0   1   0]]
```

[5] O zadavanju opsega već je diskutovano u glavi 5, deo o indeksiranju sekvenci.

```
[[  1   1   1]
 [  0   0 100]]]
```

Iz primera se uočava da se elementima matrice može pristupiti navođenjem torke celih brojeva, što nije bilo moguće sa standardnim sekvencama. Na primer, umesto `B[1][1]`, koristi se prirodniji zapis `B[1, 1]`.[6] Kao i u slučaju sekvenci, negativni indeksi označavaju elemente sa kraja odgovarajuće dimenzije: -1 za poslednji, -2 za pretposlednji... Ako se neki indeksi izostave (`C[1]` ili `C[1, 0]`), pristupa se traženoj *podmatrici* dimenzionalnosti $n-k$, gde n označava broj dimenzija matrice, a k broj navedenih celobrojnih indeksa iz *prvih k* dimenzija. U gornjem primeru, matrica `C` je trodimenzionalna ($n=3$) pa se navođenjem `C[1]` ($k=1$, prva dimenzija), vraća dvodimenzionalna, a navođenjem `C[1, 0]` ($k=2$, prve dve dimenzije), vraća jednodimenzionalna podmatrica. Osnovno indeksiranje ilustrovano je na slici 11.3.



Slika 11.3: Osnovno indeksiranje u NumPy-u.

[6] Moglo je i `B[(1, 1)]` - zagrade koje označavaju torku mogu se izostaviti pa će se nadalje tako i navoditi.

Promena vrednosti na pojedinačnoj poziciji obavlja se pomoću operatora dodele (`C[1, 1, 2] = 100`), s tim da se dodeljena vrednost *direktno* upisuje u odgovarajuću memorijsku lokaciju, bez upotrebe objektnih referenci (videti sliku 11.2).

U okviru osnovnog indeksiranja mogu se koristiti i opsezi definisani izrazom `start:stop:korak`, pri čemu se `start` uključuje, a `stop` isključuje kada se definišu granice opsega. Kao i u slučaju standardnih sekvenci, neki od parametara `start`, `stop` i `korak` mogu se izostaviti. Ako se izostavi `start`, podrazumeva se da je 0, a ako se izostavi `stop`, smatra se da je jednako broju elemenata u posmatranoj dimenziji. Kada se izostavi `korak`, podrazumeva se da je 1 i tada se druga dvotačka može izostaviti:

```
>>> A = np.array([1, 2, 3, 4, 5])
>>> print(A)
[1 2 3 4 5]
>>> print(A[:2], A[2:], A[1:3], A[1:4:2], A[::2], A[-1:1:-1])
[1 2] [3 4 5] [2 3] [2 4] [1 3 5] [5 4 3]
>>> B = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
>>> print(B)
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
>>> # podmatrice kao nizovi
>>> print(B[:, 2], B[1, :], B[0, 1:3], B[-1, 1:4:2], B[0, -1::-1])
[ 3  7 11] [5 6 7 8] [2 3] [10 12] [4 3 2 1]
>>> # podmatrice kao 2D matrice
>>> print(B[:, 2:3], B[1:2, :], B[0:1, 1:3], B[-1:-2:-1, 1:4:2], \
      B[0:1, -1::-1])
[[ 3]
 [ 7]
 [11]] [[5 6 7 8]] [[2 3]] [[10 12]] [[4 3 2 1]]
>>> print(B[1:3, 1:3])
[[ 6  7]
 [10 11]]
>>> print(B[:, 1::2])
[[ 2  4]
 [ 6  8]
 [10 12]]
```

Uočiti iz prethodnog primera sa matricom `B` da, ako se za jedan od indeksa upotrebi ceo broj, a za drugi opseg, onda se kao rezultat dobija niz brojeva. Ako su oba indeksa zadata opsezima, dobijaju se dvodimenzionalne matrice (videti sliku 11.3). Ovakvo ponašanje važi i u slučaju većeg broja dimenzija:

```
>>> C = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
>>> print(C, 'oblik matrice C je', C.shape)
```

```
[[[1 2]
  [3 4]]

 [[5 6]
  [7 8]]] oblik matrice C je (2, 2, 2)
>>> print(C[1, 1, :], C[1, 1:2, :], C[1:2, 1:2, :])
[7 8] [[7 8]] [[[7 8]]]
```

Na kraju izlaganja o osnovnom indeksiranju treba pomenuti i upotrebu operatora izostavljanja (...),[7] kojim se proširuje selekciona torka tako da se obuhvate sve dimenzije matrice. Proširivanje se vrši dvotačkom – indeksiraju se svi elementi u izostavljenim dimenzijama:

```
>>> print(C)
[[[1 2]
  [3 4]]

 [[5 6]
  [7 8]]]
>>> print(C[..., 1]) # isto kao C[:, :, 1], 2. kolona iz svake ravni
[[2 4]
 [6 8]]
>>> print(C[..., 1:]) # kao C[:, :, 1:], isti broj dimenzija kao C
[[[2]
  [4]]

 [[6]
  [8]]]
>>> print(C[0:1, ...]) # kao C[0:1, :, :], prva ravan
[[[1 2]
  [3 4]]]
```

Promena oblika matrice, pogledi i kopije

U inženjerskoj praksi često je potrebno da se matrica transformiše po svom obliku, bez brisanja ili dodavanja novih elemenata. Jedan od primera je *transponovanje* dvodimenzionalne matrice, kada vrste i kolone menjaju mesta. Drugi primer je metoda `reshape()`, iz klase `numpy.ndarray`, koja menja oblik matrice na osnovu novozadatih dimenzija:

```
>>> A = np.array([[1, 2, 3], [4, 5, 6]])
>>> print(A)
[[1 2 3]
```

[7] Engl. *Ellipsis* - trotačka.

```
 [4 5 6]]
>>> print(f'oblik: {A.shape}, broj elemenata: {A.size}, \
a_ij zauzima: {A.itemsize} bajta')
oblik: (2, 3), broj elemenata: 6, a_ij zauzima: 8 bajta
>>> At = A.T # transponovanje, može i At = A.transpose()
>>> print(At, '\noblik:', At.shape)
[[1 4]
 [2 5]
 [3 6]]
oblik: (3, 2)
>>> B, C  = A.reshape(3, 2), A.reshape(6) # preoblikovanje
>>> print(B, '\noblik:', B.shape)
[[1 2]
 [3 4]
 [5 6]]
oblik: (3, 2)
>>> print(C, '\noblik:', C.shape)
[1 2 3 4 5 6]
oblik: (6,)
```

Primetiti da se transponovanoj matrici može prići preko atributa matričnog objekta (`A.T`), ili korišćenjem matrične metode `transpose()`. Prilikom navođenja novih dimenzija matrice, u okviru metode `reshape()`, treba voditi računa da proizvod navedenih dimenzija bude *jednak* ukupnom broju elemenata matrice (`A.size`). Kako bi se operacije preoblikovanja obavljale efikasno, *bez kopiranja* elemenata polazne matrice, NumPy tretira matrične objekte kao *poglede*[8] na originalnu memorijsku zonu koja skladišti elemente. Pogled sadrži sve potrebne informacije o obliku matrice, odnosno o broju elemenata po svakoj dimenziji. Posle preoblikovanja originalne matrice, dobijena matrica uzima elemente iz *iste* memorijske zone kao i polazna matrica. Zbog toga treba voditi računa da, prilikom ažuriranja matrice, *sve* matrice koje sa njom dele istu memorijsku zonu bivaju *promenjene*! Sledi primer koji ilustruje ovu činjenicu na prethodno definisanim matricama `A`, `At`, `B` i `C`. Kako su nastale preoblikovanjem `A` putem transponovanja, odnosno metode `reshape()`, matrice `At`, `B` i `C` dele istu memorijsku zonu sa `A`:

```
>>> A[0, 2] = 100 # matrica A definisana u prethodnom primeru
>>> print(B) # B je pogled na A pa je promenjena
[[  1   2]
 [100   4]
 [  5   6]]
>>> print(C) # C je pogled na A pa je promenjena
[  1   2 100   4   5   6]
>>> C[0] = -100 # promena u pogledu menja i originalnu matricu
```

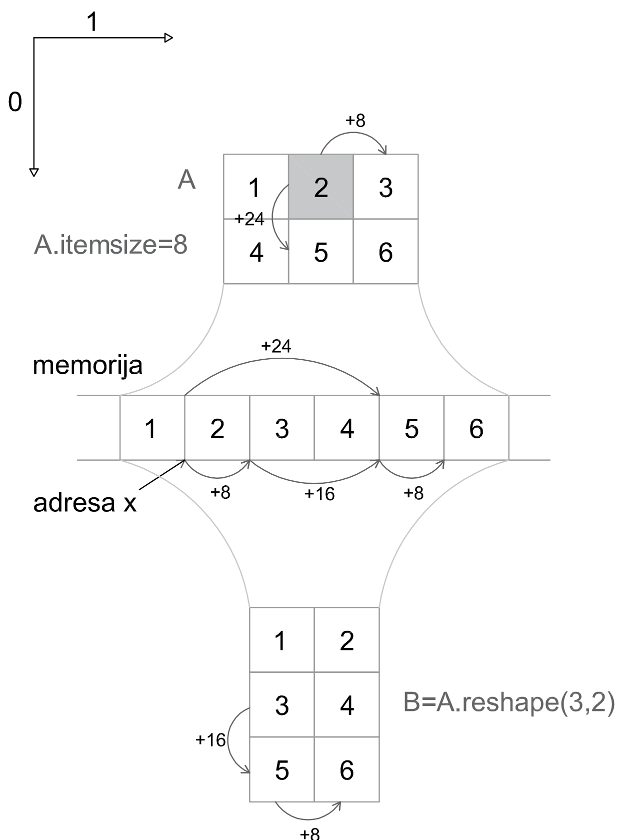
[8] Engl. *View* - pogled.

```
>>> print(A)
[[-100    2  100]
 [   4    5    6]]
>>> print(At) # At je pogled na A
[[-100    4]
 [   2    5]
 [ 100    6]]
```

Poglede dodatno pojašnjava sledeće razmatranje. Matrice se skladište u *kontinualnu* memorijsku zonu po *vrstama*. Na primer, matrica

$$A_{2\times 3} = \begin{bmatrix} 1 & \boxed{2} & 3 \\ 4 & 5 & 6 \end{bmatrix},$$

smešta se u memoriju po vrstama na način kako je to prikazano na slici 11.4.



Slika 11.4: Pogledi u NumPy-u.

Na osnovu početne adrese memorijske zone koja skladišti matricu, oblika matrice (`A.shape`), veličine u bajtima pojedinačnih elemenata (`A.itemsize`) i zadatih indeksa,

izračunava se memorijska adresa elementa kome se želi pristupiti. Pošto su brojevi u matrici celobrojni, svaki od njih zauzima 8 bajtova. Pri kretanju kroz prvu dimenziju (osa 0, označava vrste), treba praviti korake od $24 = 3 * 8$ bajtova. Na primer, ako se broj 2 nalazi na adresi x (slika 11.4), onda se sledeći element po prvoj dimenziji, broj 5, nalazi na adresi $x + 24$. Sledeći element po drugoj dimenziji (osa 1, označava kolone) je broj 3, a kroz ovu dimenziju kreće se u pomerajima od po 8 bajtova – adresa za 3 je $x + 8$. Kada se sa `B = A.reshape(3, 2)` matrica preoblikuje, dobija se pogled nad istim podacima, ali sada brojevi 1 i 2 čine prvu, 3 i 4 drugu, a 5 i 6 treću vrstu matrice `B` (slika 11.4, dole). U okviru ovog pogleda, kroz prvu dimenziju kreće se u koracima od po 16 bajtova, a kroz drugu koracima od 8 bajtova. Matrični atribut `base` govori o tome da li je matrica nastala preoblikovanjem neke druge matrice, a atribut `strides` nosi informaciju o pomerajima po dimenzijama:

```
>>> A = np.arange(6)
>>> print(A, A.shape, A.base, A.strides)
[0 1 2 3 4 5] (6,) None (8,)
>>> B = A.reshape(2, 3)
>>> print(B, B.shape, B.base, B.strides)
[[0 1 2]
 [3 4 5]] (2, 3) [0 1 2 3 4 5] (24, 8)
>>> C = A.copy() # kopiranje matrice
>>> print(C, C.shape, C.base, C.strides)
[0 1 2 3 4 5] (6,) None (8,)
>>> C[0] = 10
>>> print(A)
[0 1 2 3 4 5]
>>> B[0, 0] = 10
>>> print(A)
[10  1  2  3  4  5]
>>> np.shares_memory(A, B)
True
>>> np.shares_memory(A, C)
False
```

Kako je niz `A` kreiran putem funkcije `arange`, a ne preoblikovanjem, vrednost atributa `A.base` jednaka je `None`. Pošto je niz jednodimenzionalna matrica, torka `strides` ima samo jednu komponentu, a kako su u pitanju celobrojni elementi, pomeraj iznosi 8 bajtova (susedni elementi u nizu su susedni i u memoriji). Matrica `B` nastaje preoblikovanjem niza `A` sa kojim deli istu memorijsku zonu – zato je vrednost za `B.base` upravo niz `A`. Međutim, budući da je `B` dvodimenzionalna, pomeraji po dimenzijama su 24 i 8 (u prvoj dimenziji uzima se svaki treći, a u drugoj svaki sledeći element kontinualne zone).

NumPy nudi mogućnost *kopiranja* matrice pomoću metode `copy()`, čime se formira nova matrica sa elementima duplikatima u novokreiranoj memorijskoj zoni. Tada nema

opasnosti da se, po promeni vrednosti kopirane matrice, promeni originalna i obrnuto (`C[0]=10` ne utiče na originalni niz `A`). Sa druge strane, promena u `B` utiče i na niz `A`. Na kraju, primer ilustruje upotrebu funkcije `shares_memory()` koja utvrđuje da li dve matrice dele istu memorijsku zonu.

! Prilikom osnovnog indeksiranja, kreira se *pogled* na matricu kojoj se pristupa.

```
>>> # osnovno indeksiranje i pogledi
>>> A = np.zeros((3, 3))
>>> print(A)
[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
>>> B = A[:2, :] # B je pogled nastao putem osnovnog indeksiranja!
>>> print(B)
[[0. 0. 0.]
 [0. 0. 0.]]
>>> np.shares_memory(A, B)
True
>>> B[0, 0] = 1 # promena u B utiče na A
>>> print(A)
[[1. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
>>> A[1, 1] = 2  # ... i obrnuto
>>> print(B)
[[1. 0. 0.]
 [0. 2. 0.]]
```

Prolazak kroz matricu

Sistem prolaska kroz sve elemente jedne matrice(podmatrice) zasniva se na činjenici da se elementi smeštaju u memoriju *po vrstama*. Sledeći primer ilustruje prolazak kroz trodimenzionalnu matricu:

```
>>> A = np.array([[[1, 2, 3],[4, 5, 6]], [[7, 8, 9],[10, 11, 12]]])
>>> A
array([[[ 1,  2,  3],
        [ 4,  5,  6]],

       [[ 7,  8,  9],
        [10, 11, 12]]])
>>> for ravan in A:
        for vrsta in ravan:
```

```
                for el in vrsta:
                        print(el, end=' ') # nastavi, isti red


1 2 3 4 5 6 7 8 9 10 11 12
>>> for i in np.ndenumerate(A):
        print(i)


((0, 0, 0), 1)
((0, 0, 1), 2)
((0, 0, 2), 3)
((0, 1, 0), 4)
((0, 1, 1), 5)
((0, 1, 2), 6)
((1, 0, 0), 7)
((1, 0, 1), 8)
((1, 0, 2), 9)
((1, 1, 0), 10)
((1, 1, 1), 11)
((1, 1, 2), 12)
```

Uočiti da se pomoću petlje `for` prvog nivoa prolazi kroz ravni, drugog nivoa kroz vrste, a trećeg nivoa kroz pojedinačne elemente matrice. Ovo potvrđuje i primer sa funkcijom `ndenumerate()` koja dohvata elemente zajedno sa njihovim indeksima. Primetiti da se najbrže menja indeks krajnje desne, a najsporije indeks krajnje leve dimenzije. Pomoću osnovnog indeksiranja može se sprovesti proizvoljan redosled obilaska matrice. Sledi primer u kome se elementi unutar svake ravni obilaze po kolonama:

```
>>> # obilazak po kolonama unutar svake ravni
>>> dim0, dim1, dim2 = A.shape
>>> for i in range(dim0):
        for j in range(dim2):
                for k in range(dim1):
                        print(A[i, k, j], end=' ')


1 4 2 5 3 6 7 10 8 11 9 12
```

Grupna dodela vrednosti

U dosadašnjem izlaganju pomenuto je kako se, navođenjem celobrojnih indeksa po *svim* dimenzijama, vrši pojedinačna dodela vrednosti. Međutim, često je potrebno da se većem broju pozicija u matrici *istovremeno* dodeli jedna ili više različitih vrednosti. Sledeći primer pokazuje kako se to postiže pomoću osnovnog indeksiranja:

```
>>> # A, B, C - nula matrice
>>> A, B, C = np.zeros((8,)), np.zeros((2, 3)), np.zeros((2, 2, 2))
>>> A[1::2] = 1 #  svaki drugi počevši od drugog elementa
>>> A
array([0., 1., 0., 1., 0., 1., 0., 1.])
>>> B[:, 1:] = 1 # druga i treća kolona
>>> B
array([[0., 1., 1.],
       [0., 1., 1.]])
>>> C[1, 0, :] = 1 # prva vrsta druge ravni
>>> C
array([[[0., 0.],
        [0., 0.]],

       [[1., 1.],
        [0., 0.]]])
```

Funkcija `zeros()` podrazumevano kreira realnu matricu. Primer pokazuje da se vrednost sa desne strane jednakosti upisuje na *sve* pozicije podmatrice određene indeksima sa leve strane. Podmatrici neke matrice može se dodeliti i veći broj (potencijalno različitih) vrednosti tako što se, na desnoj strani jednakosti, navede matrica koja ih sadrži, a čije su dimenzije *kompatibilne* sa podmatricom na levoj strani. Kompatibilnost podrazumeva da podmatrica sa leve i matrica sa desne strane imaju *isti oblik* ili se oblik desne matrice može, određenim ponavljanjem njenih elemenata, *misaono proširiti* tako da bude jednak obliku leve podmatrice – ovaj mehanizam zove se matrično emitovanje[9] o čemu će više reči biti u sledećoj glavi. Evo primera kada dve matrice imaju isti oblik:

```
>>> A = np.zeros((5, 5), dtype=int)
>>> V = np.array([[-1, -2, -1], [-2, -3, -5]]) # vrednosti za dodelu
>>> A[1::2, 0::2] = V
>>> A
array([[ 0,  0,  0,  0,  0],
       [-1,  0, -2,  0, -1],
       [ 0,  0,  0,  0,  0],
       [-2,  0, -3,  0, -5],
       [ 0,  0,  0,  0,  0]])
>>> # pored matrica mogu i sekvence odgovarajućeg oblika
>>> A[0, 1:3] = 10, 15 # 1D-podmatrica
>>> A
array([[ 0, 10, 15,  0,  0],
       [-1,  0, -2,  0, -1],
       [ 0,  0,  0,  0,  0],
       [-2,  0, -3,  0, -5],
```

[9] Engl. *Matrix Broadcasting* - matrično emitovanje.

```
       [ 0,  0,  0,  0,  0]])
>>> A[:2, :2] = [[7, 8], [8, 7]] # 2D-podmatrica
>>> A
array([[ 7,  8, 15,  0,  0],
       [ 8,  7, -2,  0, -1],
       [ 0,  0,  0,  0,  0],
       [-2,  0, -3,  0, -5],
       [ 0,  0,  0,  0,  0]])
```

U prvoj dodeli, podmatrica matrice `A`, kojoj se dodeljuju vrednosti sa desne strane, sastoji se od elemenata iz preseka druge i četvrte vrste sa prvom, trećom i petom kolonom – njen oblik je 2x3. Matrica `V`, koja objedinjuje vrednosti za dodelu, ima isti oblik pa se njeni odgovarajući elementi upisuju u podmatricu `A`. U nastavku primera vidi se da se grupa vrednosti za dodelu može predstaviti i standardnom sekvencom (torka u drugoj, lista listi u trećoj dodeli). Treba samo obratiti pažnju da se sekvence sa desne strane mogu prevesti u matrice čije dimenzije odgovaraju selekciji sa leve strane.

Problem 11.1 — Blok matrica. Za uneto parno $n\,(n = 2k)$, formirati kvadratnu matricu $\mathbf{M} = \begin{bmatrix} \mathbf{A} & \mathbf{1} \\ \mathbf{0} & \mathbf{B} \end{bmatrix}$ reda n, gde su $\mathbf{1}$ i $\mathbf{0}$ odgovarajuće kvadratne matrice ispunjene jedinicama i nulama, a $\mathbf{A}$ i $\mathbf{B}$ kvadratne matrice reda k:

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & 2 & \dots & 0 \\ . & . & \dots & . \\ 0 & 0 & \dots & k \end{bmatrix}, \mathbf{B} = \begin{bmatrix} k & k & \dots & k \\ k & k & \dots & k \\ . & . & \dots & . \\ k & k & \dots & k \end{bmatrix}.$$

Porediti rešenje koje se zasniva na pojedninačnoj dodeli elemenata sa petljom `for` i rešenje koje koristi osnovno indeksiranje i odgovarajuće funkcije iz paketa NumPy. ■

Formiraju se dve funkcije: `pojedinačna_dodela()` i `grupna_dodela()`. Prva funkcija prvo formira nula matricu reda n pa onda, u okviru petlje `for`, postavlja elemente iz matrica $\mathbf{A}$, $\mathbf{B}$ i $\mathbf{1}$ na njihove krajnje pozicije u matrici $\mathbf{M}$. Druga funkcija koristi funkciju `diag()` i grupnu dodelu vrednosti primenom osnovnog indeksiranja. Obe funkcije vraćaju rezultujuću matricu i ispisuju vreme potrebno za izvršenje zadatka:

Program 11.1 — Blok matrica.

```
1 # Blok matrica
2 import numpy as np
3 from time import perf_counter
4
5 # klasičan pristup
6 def pojedinacna_dodela(n):
```

```
7
8       s = perf_counter()
9       k = n // 2
10      M = np.zeros((n, n), dtype=int)
11      for i in range(n):
12          for j in range(n):
13              if i == j and i < k:   # elementi matrice A
14                  M[i, j] = i + 1
15              elif i < k and j >= k: # elementi matrice 1
16                  M[i, j] = 1
17              elif i >= k and j >= k: # elementi matrice B
18                  M[i, j] = k
19      print('pojedinačna dodela, for petlja', perf_counter() - s)
20      return M
21
22  # NumPy pristup
23  def grupna_dodela(n):
24
25      s = perf_counter()
26      k = n // 2
27      M = np.zeros((n, n), dtype=int)
28      M[:k, :k] = np.diag([i for i in range(1, k+1)]) # mat. A
29      M[:k, k:] = 1   # mat. 1
30      M[k:, k:] = k   # mat. B
31      print('grupna dodela', perf_counter() - s)
32      return M
33
34  n = int(input('unesite parno n '))
35  print(pojedinacna_dodela(n))
36  print()
37  print(grupna_dodela(n))
```

Oba pristupa podrazumevaju formiranje početne matrice sa nulama pomoću funkcije `zeros()` – (r10, 27). U klasičnom rešenju (r11-18), uz pomoć dvostruke petlje, prolazi se kroz sve pozicije kvadratne matrice `M`. Ispunjenost logičkih uslova u višegranoj naredbi `if` definiše u kojoj se zoni matrice vrši upis pa samim tim i čiji se elementi upisuju (iz matrica `A`, `1` ili `B`). U pristupu koji ne koristi pojedinačne dodele (r28-30), koristi se funkcija `diag` i grupna dodela vrednosti na nivou traženih podmatrica. Primetiti upotrebu notacije za brzo definsanje liste po unapred zadatom kriterijumu (r28). Sledi primer rada programa:

```
unesite parno n 1000
pojedinačna dodela, for petlja 0.12230506300693378
[[  1   0   0 ...   1   1   1]
 [  0   2   0 ...   1   1   1]
 [  0   0   3 ...   1   1   1]
 ...
 [  0   0   0 ... 500 500 500]
 [  0   0   0 ... 500 500 500]
 [  0   0   0 ... 500 500 500]]

grupna dodela 0.0038664619787596166
[[  1   0   0 ...   1   1   1]
 [  0   2   0 ...   1   1   1]
 [  0   0   3 ...   1   1   1]
 ...
 [  0   0   0 ... 500 500 500]
 [  0   0   0 ... 500 500 500]
 [  0   0   0 ... 500 500 500]]
```

Iz prethodnog se zaključuje da je "NumPy pristup" ne samo kraći (elegantniji), već i znatno brži. Uočava se i da se velike matrice ispisuju u *skraćenoj* formi – za ispis *svih* elemenata velike matrice *pre* ispisa treba pozvati funkciju `np.printops(threshold=np.inf)` – promenljiva `np.inf` ukazuje na pozitivnu beskonačnost.

11.1.3 Matrične funkcije i operacije

Na početku izlaganja o paketu NumPy napomenuto je da matrica zauzima kontinualan prostor u radnoj memoriji, te da svi njeni elementi predstavljaju objekte *iste* klase. Zbog toga, prilikom obavljanja matričnih operacija, NumPy upotrebljava *vektorske instrukcije* modernih procesora.

Paket NumPy napisan je u jeziku C, uz optimalno korišćenje resursa modernih procesora. Matrične operacije koriste vektorske SIMD instrukcije (*Single Instruction Multiple Data*), koje omogućavaju da se u *istom* procesorskom taktu obavi jedna operacija nad *više istorodnih* podataka. Na taj način se postiže ubrzanje u odnosu na slučaj kada se koriste standardne ne-vektorske instrukcije.

Unarne funkcije

Unarne funkcije primaju samo jedan matrični argument, a kao rezultat vraćaju drugu, *novokreiranu* matricu. Neke često korišćene unarne funkcije prikazane su u tabeli 11.2.

Prilikom izračunavanja unarne funkcije Y$= f(\mathrm{X})$ važi $y_{ij\ldots k} = f(x_{ij\ldots k})$, odnosno funkcija "ulazi" u matricu i primenjuje se na *svaki* element *ponaosob*:

Unarna matrična funkcija	Funkcija u NumPy-u
$\vert X \vert$	`np.absolute(X)`
$\sqrt{X}$	`np.sqrt(X)`
$\sin X$	`np.sin(X)`
$\cos X$	`np.cos(X)`
$\tan X$	`np.tan(X)`
$\exp X$	`np.exp(X)`
$\log X$	`np.log(X)`
$\log_{10} X$	`np.log10(X)`

Tabela 11.2: Često korišćene unarne funkcije u NumPy-u.

```
>>> A = np.array([i**2 for i in range(9)]).reshape(3, 3)
>>> A
array([[ 0,  1,  4],
       [ 9, 16, 25],
       [36, 49, 64]])
>>> np.sqrt(A)
array([[0., 1., 2.],
       [3., 4., 5.],
       [6., 7., 8.]])
>>> np.exp(A[0, :]) # primena po osnovnom indeksiranju
array([ 1.        ,  2.71828183, 54.59815003])
```

Binarne funkcije i operacije

Binarne funkcije primaju dva matrična, jedan matrični i jedan skalarni ili dva skalarna argumenta, a kao rezultat vraćaju *novokreiranu* matricu (ili skalar). Neke često korišćene binarne funkcije prikazane su u tabeli 11.3. Primetiti da se mnoge funkcije mogu izračunati i putem standardnih matematičkih operatora poput + ili $*$. O pojmu preopterećenja operatora, koji se odnosi na obavljanje različitih operacija u zavisnosti od tipa argumenata, bilo je reči i ranije. Na primer, u slučaju operatora +, kod numeričkih tipova obavlja se sabiranje, a kod tekstualnih objekata spajanje. U glavi 10 je pokazano kako se značenje operatora može redefinisati upotrebom odgovarajućih magičnih metoda. Na primer, klasa koja redefiniše sabiranje treba da implementira metodu `__add__()`.

Prilikom izračunavanja binarne funkcije $Z = f(X, Y)$, ako su oba argumenta (ili operanda u slučaju operacije) matrice *istog* oblika, važi $z_{ij...k} = f(x_{ij...k}, y_{ij...k})$, odnosno funkcija "ulazi" u matrice i primenjuje se na *svaki* par odgovarajućih elemenata *ponaosob*:

Binarna operacija/funkcija	**Funkcija u NumPy-u**	**Operator**
$X+Y$	`np.add(X, Y)`	`X + Y`
$X-Y$	`np.subtract(X, Y)`	`X - Y`
$X \cdot Y$	`np.multiply(X, Y)`	`X * Y`
$X \div Y$	`np.divide(X, Y)`	`X / Y`
$X \mod Y$	`np.mod(X, Y)`	`X % Y`
X^Y	`np.power(X, Y)`	`X**Y`
$\max(X,Y)$	`np.maximum(X, Y)`	
$\min(X,Y)$	`np.minimum(X, Y)`	

Tabela 11.3: Često korišćene binarne funkcije i operacije u NumPy-u. U daljem tekstu, kada se funkcija može pozvati i putem redefinisanog operatora, koristiće se operator.

```
>>> A = np.arange(10).reshape((2, 5))
>>> A
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
>>> B = np.arange(9, -1, -1).reshape((2, 5))
>>> B
array([[9, 8, 7, 6, 5],
       [4, 3, 2, 1, 0]])
>>> np.add(A, B) # može i A + B
array([[9, 9, 9, 9, 9],
       [9, 9, 9, 9, 9]])
>>> A + B
array([[9, 9, 9, 9, 9],
       [9, 9, 9, 9, 9]])
>>> A * B
array([[ 0,  8, 14, 18, 20],
       [20, 18, 14,  8,  0]])
>>> np.maximum(A, B)
array([[9, 8, 7, 6, 5],
       [5, 6, 7, 8, 9]])
```

Ako je jedan od argumenata (operanada) u izrazu $Z = f(X,Y)$ *skalarna* veličina c, odnosno $Z = f(X,c)$, onda je opšti član matrice Z, kao i u linearnoj algebri, jednak $z_{ij...k} = f(x_{ij...k}, c)$:

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

```
>>> B
array([[9, 8, 7, 6, 5],
       [4, 3, 2, 1, 0]])
>>> A * 2 # isto kao 2 * A
array([[ 0,  2,  4,  6,  8],
       [10, 12, 14, 16, 18]])
>>> A + 2 # isto kao 2 + A
array([[ 2,  3,  4,  5,  6],
       [ 7,  8,  9, 10, 11]])
>>> A**2
array([[ 0,  1,  4,  9, 16],
       [25, 36, 49, 64, 81]])
>>> A * B + 2 * A  # matrični izraz
array([[ 0, 10, 18, 24, 28],
       [30, 30, 28, 24, 18]])
>>> np.power(2, 3) # oba skalara
8
```

Primer pokazuje kako se na jednostavan način kreiraju složeniji matrični izrazi. Ovde treba voditi računa da se oblici matrica međurezultata slažu po dimenzijama, ili da postoji mogućnost transformacije nekompatibilne matrice u kompatibilnu putem *emitovanja*. O postupku emitovanja biće reči nešto kasnije. Poslednji primer pokazuje da se funkcije, ako su pozvane sa samo skalarnim argumentima, ponašaju na očekivani način dajući za rezultat skalare.

U inženjerskoj praksi često treba pomnožiti *saglasne* dvodimenzionalne matrice na način kako je definisano matrično množenje (videti problem 9.4, jednačina 9.1). Za ovu potrebu može se koristiti funkcija `np.matmul()` ili binarni operator `@`:

```
>>> A = np.arange(6).reshape((3, 2))
>>> A
array([[0, 1],
       [2, 3],
       [4, 5]])
>>> B = np.arange(6).reshape((2, 3))
>>> B
array([[0, 1, 2],
       [3, 4, 5]])
>>> np.matmul(A, B) # isto kao A @ B
array([[ 3,  4,  5],
       [ 9, 14, 19],
       [15, 24, 33]])
>>> A @ B
array([[ 3,  4,  5],
       [ 9, 14, 19],
       [15, 24, 33]])
```

```
>>> E = np.eye(3, dtype=int) # jedinična matrica 3x3
>>> E
array([[1, 0, 0],
       [0, 1, 0],
       [0, 0, 1]])
>>> E @ A  # =A, A @ E ne može zbog neusaglašenih dimenzija!
array([[0, 1],
       [2, 3],
       [4, 5]])
```

Postupak emitovanja u matričnim operacijama

Matrične operacije između matrica *nejednakih* oblika moguće su samo onda ako su dve matrice *kompatibilne* za emitovanje. Da bi se ovaj postupak pojasnio navodi se sledeći primer:

```
>>> A, B = np.arange(6).reshape((2, 3)), np.arange(3)
>>> A # A.shape = (2, 3)
array([[0, 1, 2],
       [3, 4, 5]])
>>> B # B.shape = (3,)
array([0, 1, 2])
>>> A * B  # pre množenja po parovima elemenata vrši se emitovanje
array([[ 0,  1,  4],
       [ 0,  4, 10]])
```

Primer pokazuje dve matrice čiji se oblici razlikuju – matrica `A` je dvodimenzionalna, a `B` je jednodimenzionalni niz. Ipak, ako se pokuša sa množenjem po pojedinačnim elementima, dobija se rezultat koji odgovara množenju `A` sa matricom istog oblika koja nastaje iz niza `B` na sledeći način:

$$\begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \end{bmatrix} \cdot \begin{bmatrix} 0 & 1 & 2 \end{bmatrix} \xrightarrow{po\ emitovanju} \begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \end{bmatrix} \cdot \begin{bmatrix} 0 & 1 & 2 \\ 0 & 1 & 2 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 4 \\ 0 & 4 & 10 \end{bmatrix}$$

Emitovanjem se postiže efekat *proširivanja* niza `B` u dvodimenzionalnu matricu čije vrste su jednake sa `B`.

> ! Postupak emitovanja zapravo *ne proširuje* matrice u memoriji jer bi to usporilo matrične operacije i utrošilo dosta memorijskog prostora. Analogija sa proširivanjem je korišćena da bi se efekat operacije koja koristi emitovanje lakše pojasnio.

Dve matrice su kompatibilne za emitovanje ako su ispunjeni određeni uslovi. Neka je oblik matrice `X`, `X.shape`, jednak `(l, ..., d)` i neka `l` označava krajnje levu, a `d`

krajnje desnu dimenziju. Dve matrice su, nakon *poravnavanja* njihovih oblika po krajnje *desnoj* dimenziji, kompatibilne za emitovanje ako su sve *uparene* dimenzije *jednake* ili je neka od njih *jednaka jedinici*. Neuparene dimenzije (kada se broj dimenzija dve matrice razlikuje) ne utiču na kompatibilnost i biće prepisane u oblik rezultujuće matrice. Evo primera koji ovo ilustruje, koristeći se oblicima matrica `X` i `Y`, te matricom `R` koja nastaje nakon izvršavanja binarne funkcije (operacije):

```
X.shape: 3 x 2
Y.shape:     2
R.shape: 3 x 2

X.shape:         2
Y.shape: 4 x 3 x 2
R.shape: 4 x 3 x 2

X.shape:     5 x 4
Y.shape: 4 x 3 x 4
R.shape: NEKOMPATIBILNO

X.shape: 5 x 1 x 5
Y.shape: 5 x 3 x 5
R.shape: 5 x 3 x 5

X.shape: 5 x 1 x 5 x 2
Y.shape:     3 x 1 x 2
R.shape: 5 x 3 x 5 x 2
```

Izlaganje o emitovanju zaključuje sledeći primer:

```
>>> import numpy as np
>>> A, B = np.arange(3).reshape((3, 1)), np.arange(4)
>>> A
array([[0],
       [1],
       [2]])
>>> B
array([0, 1, 2, 3])
>>> A + B
array([[0, 1, 2, 3],
       [1, 2, 3, 4],
       [2, 3, 4, 5]])
```

U navedenom primeru, *obe* matrice emituju svoje vrednosti na sledeći način:

$$\begin{bmatrix}0\\1\\2\end{bmatrix} + \begin{bmatrix}0 & 1 & 2 & 3\end{bmatrix} \xrightarrow{po\ emitovanju} \begin{bmatrix}0 & 0 & 0 & 0\\1 & 1 & 1 & 1\\2 & 2 & 2 & 2\end{bmatrix} + \begin{bmatrix}0 & 1 & 2 & 3\\0 & 1 & 2 & 3\\0 & 1 & 2 & 3\end{bmatrix} = \begin{bmatrix}0 & 1 & 2 & 3\\1 & 2 & 3 & 4\\2 & 3 & 4 & 5\end{bmatrix}$$

Agregatne funkcije

Agregatne funkcije obično izračunavaju neko *numeričko obeležje* grupe brojeva, te kao takve predstavljaju nezamenljiv alat za analizu podataka. U svom osnovnom obliku prevode n-dimenzionalnu matricu u broj, a uz korišćenje opcionog argumenta `axis`, u matricu čija je dimenzionalnost *najviše* reda $n-1$. Često korišćene agregatne funkcije date su u tabeli 11.4.

Agregatna funkcija	Funkcija u NumPy-u
suma	`np.sum`
proizvod	`np.prod`
srednja vrednost	`np.mean`
medijana	`np.median`
varijansa	`np.var`
standardna devijacija	`np.dev`
maksimalna vrednost	`np.max`
minimalna vrednost	`np.min`
indeks maksimalne vrednosti	`np.argmax`
indeks minimalne vrednosti	`np.argmin`

Tabela 11.4: Često korišćene agregatne funkcije u NumPy-u. Isti efekat može se dobiti i pozivanjem istoimenih metoda nad matričnim objektima pa će se tako činiti u daljem tekstu.

```
>>> A
array([[[ 1,  2,  3],
        [ 4,  5,  6]],

       [[ 7,  8,  9],
        [10, 11, 12]]])
>>> np.sum(A)
78
>>> A.sum() == np.sum(A) # poziv metode nad A obavlja istu funkciju!
True
>>> # srednja vrednost u 1. ravni
>>> A[0].mean() # isto kao A[0, ...].mean()
3.5
>>> A.max(), A.min(), A.argmax(), A.argmin()
```

```
(12, 1, 11, 0)
```

Primetiti da se indeksi za maksimalnu i minimalnu vrednost odnose na *redni broj* elementa u memorijskom bloku koji skladišti matricu. Navođenjem opcionog argumenta `axis`, vrši se agregacija po jednoj ili više dimenzija:

```
>>> A
array([[[ 1,  2,  3],
        [ 4,  5,  6]],

       [[ 7,  8,  9],
        [10, 11, 12]]])
>>> B = A[0, ...]
>>> B
array([[1, 2, 3],
       [4, 5, 6]])
>>> # primer agregacije 2D matrice
>>> B.sum(axis=0)
array([5, 7, 9])
>>> B.sum(axis=1)
array([ 6, 15])
>>> # primer agregacije 3D matrice
>>> A.sum(axis=0)
array([[ 8, 10, 12],
       [14, 16, 18]])
>>> A.sum(axis=1)
array([[ 5,  7,  9],
       [17, 19, 21]])
>>> A.sum(axis=2)
array([[ 6, 15],
       [24, 33]])
>>> A.sum(axis=(0, 1)) # agregacija po prve dve ose
array([22, 26, 30])
```

Primer sa matricom `B` pokazuje kako se može dobiti niz suma po kolonama tako što se agregira po prvoj dimenziji (`axis=0`) – prilikom sumiranja u pravcu nulte ose, za svaku fiksiranu kolonu menja se indeks vrste (za konvenciju o dimenzijama i osama videti sliku 11.1). Slično važi i za sumu po vrstama u pravcu prve ose kada se, za fiksirani indeks vrste, menja indeks kolone. Primer sa matricom `A` pokazuje kako se agregacija obavlja u slučaju trodimenzionalne matrice, kao i šta se dešava ako se ona sprovodi po više dimenzija (osa). Za veći broj dimenzija, agregacija se obavlja osu po osu, s tim da se svaka sledeća osa procesira na matrici koja je nastala agregacijom po prethodnoj osi. U opštem slučaju, agregacijom n-dimenzionalne matrice `X` u pravcu k navedenih osa, nastaje $n-k$-dimenzionalna matrica čiji se oblik dobija izostavljanjem onih dimenzija iz `X.shape` koje odgovaraju navedenim osama. Na primer, posle

`A.sum(axis=(0, 1))`, kako je `A.ndim` jednako 3, te kako su navedene prve dve ose, nastaje niz oblika `(3, )` jer su iz početnog oblika `(2, 2, 3)` izostavljene prve dve dimenzije.

Problem 11.2 — Uspeh na studijama. Kreirati dvodimenzionalnu matricu koja čuva prosečne godišnje ocene n studenata tokom petogodišnjih studija. Svakom studentu odgovara tačno jedna vrsta, a godini tačno jedna kolona. Godišnji prosek studenta je realan broj iz intervala $[6, 10]$ koji se, radi simulacije, generiše na slučajan način iz uniformne raspodele. Ispisati pregledno formiranu matricu pa prikazati prosečnu ocenu svih studenata po svakoj od godina. Odrediti studente sa minimalnim i maksimalnim ukupnim prosecima i ispisati te proseke. Broj studenata učitati sa tastature. ■

Program 11.2 — Uspeh na studijama.

```
1  import numpy as np
2
3  def ispis(studenti):
4      '''Formira pregledni ispis za matricu studenata'''
5      i, tekst = 1, []
6      for student in studenti: # ocene tekućeg studenta
7          ocene = " ".join([f'{o:4.2f}' for o in student])
8          tekst.append(f'student {i}: {ocene}')
9          i += 1
10     print('\n'.join(tekst) + '\n')
11
12 n = int(input('broj studenata? '))
13 # Kreira i ispisuje slučajnu matricu n x 5
14 # sa prosecima po godinama
15 studenti = 6 + 4 * np.random.random((n, 5))
16 studenti = studenti.round(decimals=2)
17 ispis(studenti)
18
19 # prosečne ocene po godinama
20 god = 1
21 for prosek in studenti.mean(axis=0):
22     print(f'{god}. godina, prosek: {prosek:4.2f}')
23     god += 1
24
25 # student sa minimalnim prosekom + prosek
26 proseci = studenti.mean(axis=1)  # niz
27 i = proseci.argmin()
```

```
28 print(f'\nstudent {i + 1} ima min. prosek {proseci[i]:4.2f}')
29 
30 # student sa maksimalnim prosekom + prosek
31 i = proseci.argmax()
32 print(f'student {i + 1} ima max. prosek {proseci[i]:4.2f}')
```

U programu se prvo definiše funkcija koja ispisuje matricu godišnjih proseka (r3-10). Prilikom prolaska kroz vrste matrice (r6-9), u redu 7 formira se linija teksta (`ocene`) za studenta koji se trenutno obrađuje. Primetiti upotrebu tekstualne metode `join()` koja spaja prosečne ocene po godinama pomoću blanko znaka kao separatora. Po kreiranju tekstualne linije, ona se ubacuje u listu `tekst` (r8) sa idejom da se, prilikom traženog ispisa (r10), sve linije spoje korišćenjem separatora za novi red. Sve ocene smeštene su u polja širine četiri i prikazane sa dva decimalna mesta.

Nakon učitavanja broja studenata `n`, u redu 15 kreira se matrica proseka po godinama oblika `(n, 5)`. Sve ocene su uniformno raspoređeni pseudo-slučajni brojevi iz intervala $[6,10]$. Ovde se koriste svojstva sabiranja i množenja matrice skalarom, opisana u prethodnom izlaganju. U opštem slučaju, ako je potrebno kreirati slučajnu matricu $M_{m,n}$ čiji se elementi nalaze u intervalu $[a,b)$, treba navesti `M = a + (b - a) * np.random.random((m, n))`. Primetiti da matrica iz reda 15, zbog prirode funkcije `random.random()`, vraća brojeve iz $[0,1)$ i ne može da sadrži prosećnu ocenu 10. Zato se svi brojevi u matrici `studenti` zaokružuju na dve decimale primenom metode `round()`. Ova metoda kreira *novu* matricu tako što deluje na sve elemente u matrici nad kojom je pozvana.

Primena agregatnih funkcija ilustrovana je u redovima 21, 26, 27 i 31. Proseci po godinama dobijaju se agregiranjem matrice `studenti` po kolonama (`axis=0`), a ukupne prosečne ocene studenata agregiranjem iste matrice po vrstama (`axis=1`). Pozicije studenata sa najlošijom i najboljom ukupnom prosečnom ocenom dobijaju se pomoću funkcija `argmin()` i `argmax()`. Sledi prikaz rada programa za pet studenata:

```
broj studenata? 5
student 1: 6.05 7.25 8.60 9.81 9.08
student 2: 8.03 6.48 9.85 6.96 9.29
student 3: 6.49 6.64 6.78 9.80 8.20
student 4: 9.95 8.81 7.91 8.83 7.90
student 5: 7.18 8.48 6.09 7.50 7.43

1. godina, prosek: 7.54
2. godina, prosek: 7.53
3. godina, prosek: 7.85
4. godina, prosek: 8.58
5. godina, prosek: 8.38
```

```
student 5 ima min. prosek 7.34
student 4 ima max. prosek 8.68
```

Grupna dodela vrednosti (opet) i ažuriranje

U prethodnom tekstu bilo je reči kako o pojedinačnoj, tako i o grupnoj dodeli vrednosti, kada je broj elemenata matrice sa desne strane jednak broju elemenata podmatrice sa leve strane. Međutim, zahvaljujući matričnom emitovanju, grupna dodela je moguća i u slučaju kada su leva i desna matrica kompatibilne za emitovanje:

```
>>> A = np.zeros((3, 3), dtype=int)
>>> A
array([[0, 0, 0],
       [0, 0, 0],
       [0, 0, 0]])
>>> B = np.arange(1, 4).reshape(3, 1)
>>> B
array([[1],
       [2],
       [3]])
>>> A[:, 0:2] = B # dodela 1: 3x2 i 3x1 --> 3x2
>>> A
array([[1, 1, 0],
       [2, 2, 0],
       [3, 3, 0]])
>>> A[0:2, 1:] = np.array([-1, -1]) # dodela 2: 2x2 i 2 --> 2x2
>>> A
array([[ 1, -1, -1],
       [ 2, -1, -1],
       [ 3,  3,  0]])
>>> A[0:2, 1:] = [-2, -2] # može i sekvenca odgovarajuće dimenzije
>>> A
array([[ 1, -2, -2],
       [ 2, -2, -2],
       [ 3,  3,  0]])
```

U prvoj dodeli, podmatrica `A[:, 0:2]` ima oblik (3, 2), a matrica `B` oblik (3, 1). Po poravnaju krajnje desnih dimenzija (2 i 1), preostale dimenzije levo od poslednje su jednake (3 i 3) pa se matrica `B` može misaono proširiti na matricu oblika (3, 2) tako što joj se dodaje još jedna kolona poput prve. Još jednom se podseća da se matrica `B` pri emitovanju *ne proširuje*, ali je efekat dodele isti kao i kada bi ona bila proširena.

Slično se dešava i tokom druge dodele, kada se niz sa desne strane emituje u kvadratnu matricu sa dve istovetne vrste. Konačno, moguća je i dodela koja na desnoj strani ima standardnu sekvencu *odgovarajućeg* oblika pogodnog za emitovanje. U

sledećem primeru pokazano je kako se matrica *ažurira u mestu*:[10]

```
>>> A
array([[ 1, -2, -2],
       [ 2, -2, -2],
       [ 3,  3,  0]])
>>> B = A
>>> A = A + 1 # pravi se novi objekat A!
>>> A
array([[ 2, -1, -1],
       [ 3, -1, -1],
       [ 4,  4,  1]])
>>> B
array([[ 1, -2, -2],
       [ 2, -2, -2],
       [ 3,  3,  0]])
>>> A += 1 # A se ažurira u mestu!
>>> A
array([[3, 0, 0],
       [4, 0, 0],
       [5, 5, 2]])
>>> A *= B # A se ažurira u mestu!
>>> A
array([[ 3,  0,  0],
       [ 8,  0,  0],
       [15, 15,  0]])
```

Primetiti da se sa `A = A + 1` kreira nova matrica `A`, dok matrica `B` ostaje nepromenjena i ukazuje na stari matrični objekat. Ako se želi ažuriranje u mestu, bez kreiranja novog matričnog objekta, onda je najjednostavnije koristiti skraćene operatore poput +=, *= i drugih. Konačno, treba pokazati kako se ažuriranje u mestu može obaviti pomoću mnogih matričnih funkcija:

```
>>> import numpy as np
>>> A = np.arange(9).reshape(3, 3)
>>> A
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
>>> B = np.arange(9, 0, -1).reshape(3, 3)
>>> B
array([[9, 8, 7],
       [6, 5, 4],
       [3, 2, 1]])
```

[10] Engl. *In place* - u mestu, odnosno ne kreira se nova matrica.

```
>>> np.add(A, B, out=A) # isto kao A += B
array([[9, 9, 9],
       [9, 9, 9],
       [9, 9, 9]])
>>> A
array([[9, 9, 9],
       [9, 9, 9],
       [9, 9, 9]])
>>> C = np.random.random((3, 3))
>>> C
array([[0.33639682, 0.87264013, 0.23715884],
       [0.28232152, 0.79846351, 0.97632443],
       [0.49749375, 0.82859531, 0.4917901 ]])
>>> C.round(out=C) # zaokružuje elemente matrice
array([[0., 1., 0.],
       [0., 1., 1.],
       [0., 1., 0.]])
>>> C
array([[0., 1., 0.],
       [0., 1., 1.],
       [0., 1., 0.]])
```

Navođenjem opcionog argumenta `out` određuje se *postojeća* matrica u koju će biti *upisan* rezultat. Ako je navedena matrica jedna od polaznih, onda se njene vrednosti ažuriraju bez kreiranja novog matričnog objekta.

11.1.4 Poređenje, logičke operacije i napredno indeksiranje

U praksi se često javljaju zadaci poput sledećeg: izračunati prosečnu vrednost svih elemenata koji su manji od neke zadate vrednosti. Prilikom rešavanja ovog problema, uočavaju se dve bitne aktivnosti: prvo treba *pronaći* one elemente, ili zone u matrici, koje *zadovoljavaju* odgovarajući uslov (manji od zadate vrednosti); potom treba *pristupiti* ovim elementima radi njihovog izdvajanja i tražene obrade (računanje proseka). Problem pronalaženja odgovarajučih elemenata rešava se poređenjem i logičkim operacijama na nivou matrica, dok se njihovo izdvajanje obavlja pomoću naprednog indeksiranja.

Poređenje i logičke operacije

NumPy omogućava poređenje između matrica istog oblika, matrica kompatibilnih za emitovanje, kao i matrice i skalara. Poređenjem nastaju matrice čiji su elementi logičke vrednosti `True` i `False` (objekti tipa `bool`). Nad matricama koje sadrže logičke vrednosti mogu se obavljati standardne logičke operacije:

```
>>> A = (10 * np.random.random((3, 3))).round()
>>> B = (10 * np.random.random((3, 3))).round()
>>> A
array([[0., 3., 5.],
       [8., 4., 4.],
       [9., 2., 7.]])
>>> B
array([[2., 1., 9.],
       [6., 2., 1.],
       [6., 2., 2.]])
>>> A == B # poređenje po jednakosti
array([[False, False, False],
       [False, False, False],
       [False,  True, False]])
>>> A < B
array([[ True, False,  True],
       [False, False, False],
       [False, False, False]])
>>> A < B[0, :] # primenjuje se emitovanje za prvu vrstu matrice B
array([[ True, False,  True],
       [False, False,  True],
       [False, False,  True]])
>>> A > 3 # poređenje sa skalarom
array([[False, False,  True],
       [ True,  True,  True],
       [ True, False,  True]])
>>> (A > 3) & (A < 7) # operacija "i"
array([[False, False,  True],
       [False,  True,  True],
       [False, False, False]])
>>> (A == 0) | (A == 7) # operacija "ili"
array([[ True, False, False],
       [False, False, False],
       [False, False,  True]])
```

Prilikom poređenja dve matrice jednakih oblika, nastaje matrica čiji su elementi dobijeni kao rezultat poređenja odgovarajućih elemenata polaznih matrica. Na primer, za `A == B`, rezultujuća matrica ima vrednost `True` na onim pozicijama na kojima su elementi dve matrice jednaki. Za slučaj da dve matrice nemaju isti oblik, poređenje se obavlja *samo* ako su matrice kompatibilne za emitovanje (`A < B[0, :]`). Prilikom poređenja matrice sa skalarom, formira se matrica čiji su elementi rezultat poređenja skalara sa odgovarajućim elementima iz polazne matrice (`A > 3`). Za formiranje složenih logičkih izraza koriste se operatori poput `&` (i), `|` (ili) i `~` (ne), s tim da se elementarni logički izrazi *obavezno* stavljaju u zagrade. Na primer, izraz `(A > 3) & (A < 7)`, kreira logičku matricu koja ima vrednost `True` na onim mestima gde su elementi matrice `A` između 3 i 7.

! Prilikom formiranja logičkih izraza sa matricama, *ne mogu* se koristiti standardne logičke operacije poput `and`, `or` i `not`. Standardne operacije očekuju da njihovi operandi imaju jedinstvenu logičku vrednost, a matrice, čak i kada sadrže elemente tipa `bool`, ne mogu imati jedinstveno definisanu logičku vrednost.

Često je potrebno utvrditi da li su *svi* elementi u nekoj matrici jednaki `True` - funkcija `all()`, odnosno da li je *bar jedan* jednak `True` - funkcija `any()`. U opštem slučaju, uz pomoć funkcije `count_nonzero()`, prebrojava se *koliko* elemenata zadovoljava unapred definisani uslov, dok se pomoću funkcije `nonzero()` dobijaju indeksi *nenultih* elemenata:

```
>>> A = np.arange(1, 10).reshape((3, 3))
>>> A
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
>>> A < 100
array([[ True,  True,  True],
       [ True,  True,  True],
       [ True,  True,  True]])
>>> np.all(A < 100) # da li su svi elementi manji od 100
True
>>> np.any(A > 9) # da li je bar jedan veći od 9
False
>>> np.count_nonzero((A < 5) & (A > 1)) # koliko ih ima izmedju 1 i 5
3
>>> np.nonzero((A < 5) & (A > 1)) # pozicije elemenata izmedju 1 i 5
(array([0, 0, 1]), array([1, 2, 0]))
```

Primetiti da funkcije `count_nonzero()` i `nonzero()` koriste osobinu Pajtona da `False`, u aritmetičkim izrazima, tumače kao nula. Konačno, treba pomenuti i funkcije `isnan()` i `isinf()` koje utvrđuju da li u matrici postoje vrednosti "nije broj" (`np.nan`) i "beskonačno" (`np.inf`):

```
>>> B = 1 / np.log(np.array([3, 1, -1]))

Warning (from warnings module):
  File "<pyshell#3>", line 1
RuntimeWarning: invalid value encountered in log
>>> B
array([0.91023923,        inf,        nan])
>>> np.isnan(B)
array([False, False,  True])
>>> np.isinf(B)
array([False,  True, False])
```

Vrednost `np.nan` dobija se u slučaju kada matematička operacija nije definisana za date ulazne vrednosti. U gornjem primeru, „nije broj" se dobija prilikom pokušaja logaritmovanja negativne vrednosti -1. Slično, svaki izraz koji uključuje `np.nan` ne može se izračunati i ima vrednost `np.nan`. Treba primetiti da ni deljenje nulom, koje u primeru nastaje pri pokušaju izračunavanja $1/\log(1)$, nije definisano. Ipak, prema IEEE 754 standardu za aritmetiku u pokretnom zarezu, pri deljenju nulom dobija se `np.inf`. Ni ova vrednost ne predstavlja konkretan broj, ali Pajton omogućava njeno poređenje sa drugim brojevima pa se može smatrati da je svaki broj manji od nje. U problemu 3.2 bilo je reči da se objekat koji predstavlja pozitivnu beskonačnost može kreirati sa `float('inf')`. Važi da `float("inf") == np.inf` vraća `True`!

Napredno indeksiranje

Korišćenjem osnovnog indeksiranja *ne može* se definisati potpuno *proizvoljan* skup pozicija, već se izbor odnosi na neku *pravilnu podmatricu*. Kako elementi koji zadovoljavaju navedeni uslov mogu biti razbacani unutar matrice na proizvoljan način, NumPy pruža mogućnost njihovog izbora putem *naprednog indeksiranja*. Ono podrazumeva da selekcioni objekat `obj`, iz `A[obj]`, može biti celobrojna sekvenca koja nije torka, matrica celobrojnog ili logičkog tipa, ili torka u kojoj je bar jedan objekat tipa sekvence ili matrice celobrojnog ili logičkog tipa. Za razliku od osnovnog indeksiranja koje vraća pogled na izabrane elemente, napredno indeksiranje vraća matricu *kopiju*. Razmatra se prvo celobrojna varijanta naprednog indeksiranja:

```
>>> # celobrojno napredno indeksiranje
>>> A = np.arange(1, 17)
>>> A
array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16])
>>> A[[1, 2, 2, -1]] # primer 1
array([ 2,  3,  3, 16])
>>> ind = np.array([2, 3, 0])
>>> A[ind] # primer 2
array([3, 4, 1])
>>> B = A.reshape(4, 4)
>>> B
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12],
       [13, 14, 15, 16]])
>>> B[(0, 1, 2), (0, 1, 2)] # primer 3
array([1, 6, 11])
>>> B[ind] # primer 4
array([[ 9, 10, 11, 12],
       [13, 14, 15, 16],
       [ 1,  2,  3,  4]])
>>> B[ind, 2] # primer 5
```

```
array([11, 15,  3])
>>> x, y = np.eye(2, dtype=int), np.zeros((2, 2), dtype=int)
>>> x
array([[1, 0],
       [0, 1]])
>>> y
array([[0, 0],
       [0, 0]])
>>> B[x, y] # primer 6
array([[5, 1],
       [1, 5]])
```

Prvi primer pokazuje šta se dešava kada je selekcioni objekat celobrojna sekvenca koja nije torka – biraju se elementi niza prema pozicijama navedenim u listi (ovde drugi, treći, ponovo treći, pa poslednji element niza `A`). Drugi primer pokazuje kako se elementi niza mogu izabrati navođenjem celobrojne matrice. Ako je matrica koja se indeksira višedimenzionalna (primer 3), onda se za svaku dimenziju mogu navesti sekvence/matrice kojima se određuje željeni izbor elemenata. Konkretno, u pimeru 3, prvi element prve torke (0) uparuje se sa prvim elementom druge torke (0), drugi element prve torke (1) uparuje se sa drugim elementom druge torke (1) i tako dalje. Na ovaj način formiran je niz od prva tri elementa sa glavne dijagonale matrice `B`.

Kada se, prilikom naprednog indeksiranja, izostavi celobrojna sekvenca/matrica za neku od dimenzija, postižu se slični efekti kao i u slučaju izostavljanja indeksa u osnovnom indeksiranju (videti 11.1.2). U četvrtom primeru odabrane su treća, četvrta i prva vrsta matrice `B`. Ako se u nekoj dimenziji, umesto celobrojne sekvence/matrice, navede broj, onda se on uparuje sa svim vrednostima iz ostalih dimenzija – u petom primeru biraju se treći, četvrti i prvi element iz treće kolone. Konačno, ako se unutar svake dimenzije navedu celobrojne indeksne matrice jednakih oblika, rezultujuća matrica poprima taj oblik, a njeni elementi su određeni uparivanjem odgovarajućih vrednosti iz indeksnih matrica. U šestom primeru, kako su `x` i `y` matrice oblika `(2, 2)`, to `B[x, y]` postaje matrica oblika `(2, 2)` . Sada je `B[x, y][0, 0]` element koji se u originalnoj matrici nalazi na poziciji `B[1, 0]`. Pomoću naprednog indeksiranja vraća se kopija selektovanog dela matrice. To potvrđuje primer izdvajanja prva dva elementa niza `A` putem oba tipa indeksiranja:

```
>>> A = np.arange(1, 17)
>>> A
array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16])
>>> C_pogled = A[0:2] # osnovno indeksiranje
>>> C_pogled, np.shares_memory(A, C_pogled)
(array([1, 2]), True)
>>> C_kopija = A[[0, 1]] # napredno indeksiranje
>>> C_kopija, np.shares_memory(A, C_kopija)
```

```
(array([1, 2]), False)
>>> C_kopija[0] = 100
>>> A # nema promene
array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16])
>>> C_pogled[0] = 100
>>> A # ima promene
array([100, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16])
```

Napredno indeksiranje pomoću logičkih vrednosti `True` i `False` ilustrovano je u sledećem primeru:

```
>>> # napredno logičko indeksiranje
>>> A = np.arange(1, 10).reshape(3, 3) - np.diag([1, 5, 9])
>>> A
array([[0, 2, 3],
       [4, 0, 6],
       [7, 8, 0]])
>>> B = A[A != 0] # izdvaja elemente iz A različite od 0
>>> B
array([2, 3, 4, 6, 7, 8])
>>> B[0] = 10 # B je kopija, promena ne utiče na A
>>> A
array([[0, 2, 3],
       [4, 0, 6],
       [7, 8, 0]])
>>> A[A == 0] = 1 # postavlja sve nule u A na 1
>>> A
array([[1, 2, 3],
       [4, 1, 6],
       [7, 8, 1]])
>>> A[A%2 == 0].sum() # vraća sumu parnih brojeva u A
20
>>> studenti = ["123/21", "45/21", "3/20", "12/19", "1/21"]
>>> proseci = np.array([6.11, 7.44, 9.15, 8.23, 9.05])
>>> gen_21 = ["/21" in s for s in studenti]
>>> gen_21
[True, True, False, False, True]
>>> proseci[gen_21].mean() # srednji prosek generacije 21
7.533333333333334
>>> A[[True, False, True]]
array([[1, 2, 3],
       [7, 8, 1]])
```

U gornjem primeru ilustrovana je "moć" naprednog indeksiranja pomoću logičkih vrednosti. Indeksna sekvenca/matrica, koja je istog oblika kao i matrica koja se indek-

sira, odabira elemente iz polazne matrice na osnovu logičkih vrednosti za ekvivalentne pozicije – vrednost `True` odabira referentni element. Kao i u slučaju sa celobrojnim indeksima, nije neophodno navesti logičke indekse po svim dimenzijama. U poslednjem primeru biraju se one vrste matrice `A` za koje su odgovarajuće logičke vrednosti u indeksnoj sekvenci istinite. U ovoj glavi navedene su najčešće korišćene forme naprednog indeksiranja. Za ostale varijante naprednog (i osnovnog) indeksiranja, čitaoci se upućuju na referencu o biblioteci NumPy navedenu u dodatku Bibliografija.

Kombinovano indeksiranje

Osnovno i napredno indeksiranje mogu se *kombinovati* tako što se, po različitim dimenzijama matrice, primenjuju oba pristupa. Sledi nekoliko primera koji ilustruju kombinovani pristup:

```
>>> A = np.arange(27).reshape(3, 3, 3)
>>> A
array([[[ 0,  1,  2],
        [ 3,  4,  5],
        [ 6,  7,  8]],

       [[ 9, 10, 11],
        [12, 13, 14],
        [15, 16, 17]],

       [[18, 19, 20],
        [21, 22, 23],
        [24, 25, 26]]])
>>> A[0, [0, 1, 2], [1, 1, 0]] # primer 1
array([1, 4, 6])
>>> A[[0, 2], :, 2] # primer 2
array([[ 2,  5,  8],
       [20, 23, 26]])
>>> A[1, [True, False, True], 1:] # primer 3
array([[10, 11],
       [16, 17]])
>>> B = A[:, [0, 2], ::2] # primer 4
>>> B
array([[[ 0,  2],
        [ 6,  8]],

       [[ 9, 11],
        [15, 17]],

       [[18, 20],
        [24, 26]]])
>>> B[0, 0, 0] = 100 # B je kopija, A se ne menja
```

```
>>> A
array([[[ 0,  1,  2],
        [ 3,  4,  5],
        [ 6,  7,  8]],

       [[ 9, 10, 11],
        [12, 13, 14],
        [15, 16, 17]],

       [[18, 19, 20],
        [21, 22, 23],
        [24, 25, 26]]])
```

Prvi primer primenjuje osnovno indeksiranje u prvoj dimenziji matrice `A` kako bi odabrao prvu ravan. Potom se, pomoću celobrojnog naprednog indeksiranja, biraju elementi rezultujućeg *niza* brojeva. U drugom primeru, napredno celobrojno indeksiranje koristi se u prvoj, a osnovno u druge dve dimenzije matrice. Naprednim indeksiranjem odabrane su prva i treća ravan, dok se osnovnim indeksiranjem biraju svi elementi druge kolone. Primetiti da je broj dimenzija dobijene matrice za jedan manji od početne. Treći primer koristi napredno logičko indeksiranje u drugoj dimenziji kako bi se odabrale prva i treća vrsta druge ravni. Zadavanjem opsega u trećoj dimenziji, biraju se svi elementi iz odabranih vrsta počevši od drugog. Kao i u prethodnom slučaju, dobija se dvodimenzionalna matrica. Četvrti primer pokazuje kako se dobija trodimenzionalna matrica `B` sastavljena od ugaonih elemenata prve i treće ravni. Konačno, pokazano je da se kombinovanim adresiranjem, za razliku od osnovnog, dobija matrica *kopija*.

Problem 11.3 — Sopstveni vektori. Napisati funkciju koja izračunava normu vektora $\mathbf{x}$ dužine n prema formuli $\|\mathbf{x}\| = \sqrt{\sum_{i=1}^{n} x_i^2}$. Potom napisati funkciju kojom se, za zadatu kvadratnu matricu $\mathbf{A}$ reda n i vektor $\mathbf{x}$ dužine n, ispituje da li je $\mathbf{x}$ sopstveni vektor matrice $\mathbf{A}$ za koji je $\mathbf{Ax}= \lambda\mathbf{x}$, $\lambda \in \mathbb{R}$. Pretpostaviti, radi jednostavnosti, da sopstveni vektor ne sadrži ni jednu nulu! Testirati funkcije u programu koji učitava dve kvadratne matrice istih dimenzija, $\mathbf{A}$ i $\mathbf{B}$, iz datoteka `matA.txt` i `matB.txt`. Program treba da ispita da li je neka od kolona matrice $\mathbf{B}$ sopstveni vektor od $\mathbf{A}$. Među takvim kolonama ispisati onu koja ima najmanju normu. ■

U rešavanju ovog problema pretpostavlja se da su sve kvadratne matrice oblika $n \times n$. Oblici učitanih podataka neće se proveravati, već će se koristiti princip *obrade izuzetaka* pomoću kontrolne strukture `try - except`. Dalje, pretpostavlja se da su matrice $\mathbf{A}$ i $\mathbf{B}$ sačuvane u odgovarajućim datotekama po vrstama, bez zaglavlja i uz korišćenje zapete kao separatora u vrsti (redu datoteke).

Neka je $\mathbf{Z} = \mathbf{Ax}$. Ako je $\mathbf{x}$ sopstveni vektor matrice $\mathbf{A}$, onda važi $\mathbf{Z} = \mathbf{Ax} = \lambda\mathbf{x}$. Budući da, po uslovu zadatka, sopstveni vektor ima sve elemente različite od nule, provera se svodi na formiranje količnika $\frac{z_i}{x_i}$. Ako je ovaj količnik *jednak* λ za *sve*

vrednosti i, onda je $\mathbf{x}$ sopstveni vektor matrice $\mathbf{A}$:

Program 11.3 — Sopstvene vrednosti.

```
import numpy as np

def norma(x):
    """ Računa L2 normu vektora X """
    return np.sqrt((x * x).sum())

def je_sopstveni_vektor(A, x):
    """ Ispituje da li je x sopstveni vektor od A """
    if np.any(x == 0):
        return False
    else:
        Z = A @ x
        Lambda = Z / x
        return np.all(np.abs(Lambda - Lambda[0]) < 0.0001)

# program za tesiranje
try:
    A = np.loadtxt("matA.txt", skiprows=0, delimiter=",")
    B = np.loadtxt("matB.txt", skiprows=0, delimiter=",")
    min_sv, min_norma = None, np.inf

    for i in range(B.shape[0]):
        if je_sopstveni_vektor(A, B[:, i:i+1]):
            n = norma(B[:, i])
            if n < min_norma:
                min_norma, min_sv = n, B[:, i]

    if min_sv is not None:
        print("Sopstveni vektor", min_sv)
        print("ima najmanju normu", min_norma)
    else:
        print("Nema sopstvenih vektora!")
except:
    print("Proveriti da li su putanje datoteka ispravne")
    print("i/ili da li su matrice oblika n x n!")
```

Funkcija `norma()` računa euklidsku (ili L2) normu prema zadatoj formuli. Izraz u r5 daje tačan rezultat bez obzira da li je vektor `x` zadat kao niz brojeva, vektor-vrsta ili vektor-kolona: svaka komponenta množi se sama sa sobom, rezultujuća matrica sumira se po svim elementima, pa se računa kvadratni koren iz sume. Funkcija `je_sopstveni_vektor()` očekuje da vektor `x` ima oblik $n \times 1$. Zbog toga se, u r23, i-ta kolona matrice `B` indeksira navođenjem dva opsega (`B[:, i:i+1]`). Kada bi se navelo `B[:, i]`, onda bi drugi argument funkcije bio niz a ne vektor-kolona! U r9 se provera da li je bar jedan element vektora `x` jednak nuli. Ako su sve komponente vektora različite od nula, onda se, primenom operatora matričnog množenja, formira vektor `Z` oblika $n \times 1$ (r12). Vektor `Lambda` sadrži sve količnike $\frac{z_i}{x_i}$ (r13). Provera jednakosti svih količnika *ne obavlja* se pomoću logičke matrice `Lambda == Lambda[0]`, kao što bi se na prvi pogled moglo pretpostaviti. Treba se podsetiti da brojevi u pokretnom zarezu samo *približno* predstavljaju realne brojeve, te da ih je najbolje porediti tako što se utvrđuje da li su dovoljno *bliski* jedan drugome (videti poglavlje 3.2.1). Matrica `Lambda[0]` emituje se u oblik jednak matrici `Lambda` pa se potom utvrđuje bliskost svih količnika sa prvim – r14.

U glavnom programu, pomoću funkcije `loadtxt()`, učitavaju se kvadratne matrice `A` i `B` (r18-19). Potom se u glavnoj petlji prolazi kroz kolone matrice `B` (r22-26) i testira da li one predstavljaju sopstvene vektore (r23). Ako je odgovor potvrdan, računa se norma (r24) i pamti najmanji dotadašnji sopstveni vektor (r26). Pri pozivu funkcije `norma()`, i-ta kolona navodi se kao niz, a ne kao vektor-kolona. Kao što je već naglašeno, to ne utiče na ispravan rad funkcije koja računa normu. Ako postoji, minimalni sopstveni vektor ispisuje se kao niz brojeva:

```
# izgled matA.txt
5,4,2
4,5,2
2,2,2

# izgled matB.txt
1,-1,2
1,1,2
1,0,1

# po pokretanju programa
Sopstveni vektor [2. 2. 1.]
ima najmanju normu 3.0
>>>
```

11.1.5 Spajanje i razdvajanje matrica i rearanžiranje elemenata

Ova glava razmatra često korišćene operacije za *spajanje* i *razdvajanje* matrica, kao i operacije pomoću kojih se elementi matrice *rearanžiraju* po zadatom pravilu.

Spajanje i razdvajanje matrica

Razmatra se prvo spajanje matrica sa *jednakim* brojem dimenzija po nekoj od *postojećih* osa, što za rezultat daje *istodimenzionalnu* matricu ili niz:

```
>>> import numpy as np
>>> A, B = np.array([[1, 2], [3, 4]]), np.array([[5, 6], [7, 8]])
>>> A
array([[1, 2],
       [3, 4]])
>>> B
array([[5, 6],
       [7, 8]])
>>> np.concatenate((A, B), axis=0)
array([[1, 2],
       [3, 4],
       [5, 6],
       [7, 8]])
>>> np.concatenate((A, B), axis=1)
array([[1, 2, 5, 6],
       [3, 4, 7, 8]])
>>> np.hstack((A, B)) # kao concatenate za axis=1
array([[1, 2, 5, 6],
       [3, 4, 7, 8]])
>>> np.vstack((A, B)) # kao concatenate za axis=0
array([[1, 2],
       [3, 4],
       [5, 6],
       [7, 8]])
>>> np.concatenate((A, B), axis=None) # dobija se niz
array([1, 2, 3, 4, 5, 6, 7, 8])
>>> C = np.array([[10, 20]])
>>> C
array([[10, 20]])
>>> np.concatenate((A, C), axis=0)
array([[ 1,  2],
       [ 3,  4],
       [10, 20]])
>>> np.concatenate((A, C), axis=1) # greska!
Traceback (most recent call last): ...
```

Funkcija `concatenate()` spaja matrice iz torke prvog argumenta po dimenziji (osi) navedenoj parametrom `axis`. Ako se navede `axis=None`, kao rezultat se dobija niz pri čemu se elementi navedenih matrica uzimaju po vrstama. Ovde treba voditi računa da se matrice mogu spojiti po naznačenim osama – prilikom spajanja se ne obavlja emitovanje kao kod matričnih operacija! Funkcije `hstack()` i `vstack()` obavljaju sličan posao, s tim da se u prvom slučaju spaja po prvoj, a u drugom po nultoj osi. U

praksi je često potrebno kreirati *blok matricu* iz nekoliko polaznih matrica, a to se može učiniti putem funkcije `block()`:

```
>>> A
array([[1, 2],
       [3, 4]])
>>> B
array([[5, 6],
       [7, 8]])
>>> C
array([0, 1, 2, 3])
>>> np.block([[A, B], [C]])
array([[1, 2, 5, 6],
       [3, 4, 7, 8],
       [0, 1, 2, 3]])
>>> np.block([[A, B], [B, A]])
array([[1, 2, 5, 6],
       [3, 4, 7, 8],
       [5, 6, 1, 2],
       [7, 8, 3, 4]])
```

Funkcija `block()` prvo spaja matrice u najdubljim listama duž poslednje dimenzije, zatim po listama sledećeg nivoa duž pretposlednje, pa sve do spoljašnje liste gde se spaja po nultoj osi. Pomoću funkcije `stack()` može se spajati po *novoj* dimenziji:

```
>>> A, B = np.arange(6).reshape(2,3), np.arange(6, 12).reshape(2,3)
>>> A
array([[0, 1, 2],
       [3, 4, 5]])
>>> B
array([[ 6,  7,  8],
       [ 9, 10, 11]])
>>> np.stack((A, A, B), axis=0)
array([[[ 0,  1,  2],
        [ 3,  4,  5]],

       [[ 0,  1,  2],
        [ 3,  4,  5]],

       [[ 6,  7,  8],
        [ 9, 10, 11]]])
>>> np.stack((A, B), axis=1)
array([[[ 0,  1,  2],
        [ 6,  7,  8]],

       [[ 3,  4,  5],
```

```
        [ 9, 10, 11]]])
>>> np.stack((A, B), axis=2)
array([[[ 0,  6],
        [ 1,  7],
        [ 2,  8]],

       [[ 3,  9],
        [ 4, 10],
        [ 5, 11]]])
```

Problem 11.1 razmatrao je kreiranje kvadratne matrice oblika $\mathbf{M} = \begin{bmatrix} \mathbf{A} & \mathbf{1} \\ \mathbf{0} & \mathbf{B} \end{bmatrix}$ reda $n = 2k$, gde su $\mathbf{A} = \begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & 2 & \dots & 0 \\ . & . & \dots & . \\ 0 & 0 & \dots & k \end{bmatrix}$ i $\mathbf{B} = \begin{bmatrix} k & k & \dots & k \\ k & k & \dots & k \\ . & . & \dots & . \\ k & k & \dots & k \end{bmatrix}$ kvadratne matrice reda k, a $\mathbf{1}$ i $\mathbf{0}$ odgovarajuće kvadratne matrice ispunjene jedinicama, odnosno nulama. Problem se, korišćenjem operacija spajanja, može jednostavno rešiti na sledeći način:

```python
# Blok matrica
import numpy as np

n = int(input("Unesite parno n "))
if n % 2 == 0:
    k = n // 2
    M = np.block([
        [np.diag([i for i in range(1, k+1)]), np.ones((k, k))],
        [np.zeros((k, k)), k * np.ones((k, k))]
    ])
    print(f"M:\n{M}")
```

```
Unesite parno n 6
M:
[[1. 0. 0. 1. 1. 1.]
 [0. 2. 0. 1. 1. 1.]
 [0. 0. 3. 1. 1. 1.]
 [0. 0. 0. 3. 3. 3.]
 [0. 0. 0. 3. 3. 3.]
 [0. 0. 0. 3. 3. 3.]]
```

Pored spajanja, matrica se može *razdvojiti* na dve ili više podmatrica korišćenjem

funkcije `split()`. Ova funkcija vraća podmatrice *poglede*:

```
>>> C = np.block([[A, B], [B, A]]) # A, B iz prethodnog primera
>>> C
array([[ 0,  1,  2,  6,  7,  8],
       [ 3,  4,  5,  9, 10, 11],
       [ 6,  7,  8,  0,  1,  2],
       [ 9, 10, 11,  3,  4,  5]])
>>> np.split(C, 3, axis=1) # 1. razdvajanje
[array([[ 0,  1],
       [ 3,  4],
       [ 6,  7],
       [ 9, 10]]), array([[ 2,  6],
       [ 5,  9],
       [ 8,  0],
       [11,  3]]), array([[ 7,  8],
       [10, 11],
       [ 1,  2],
       [ 4,  5]])]
>>> X, Y = np.split(C, 2, axis=0) # 2. razdvajanje
>>> X, Y
(array([[ 0,  1,  2,  6,  7,  8],
       [ 3,  4,  5,  9, 10, 11]]), array([[ 6,  7,  8,  0,  1,  2],
       [ 9, 10, 11,  3,  4,  5]]))
>>> X[0, 0] = 100 # razdvajanje vraća poglede
>>> C
array([[100,   1,   2,   6,   7,   8],
       [  3,   4,   5,   9,  10,  11],
       [  6,   7,   8,   0,   1,   2],
       [  9,  10,  11,   3,   4,   5]])
>>> np.split(C, (1, 4), axis=1) # 3. razdvajanje
[array([[100],
       [  3],
       [  6],
       [  9]]), array([[ 1,  2,  6],
       [ 4,  5,  9],
       [ 7,  8,  0],
       [10, 11,  3]]), array([[ 7,  8],
       [10, 11],
       [ 1,  2],
       [ 4,  5]])]
```

Prvo razdvajanje iz prethodnog primera pokazuje kako se matrica `C` deli na tri *jednaka* dela po vertikali. Podmatrice se vraćaju kao elementi standardne liste. Prilikom navođenja drugog argumenta treba voditi računa o tome da se matrica može razdvojiti na navedeni broj jednakih delova jer, u suprotnom, interpreter prijavljuje grešku – u prvom

razdvajanju to znači da je broj kolona matrice C deljiv sa 3.[11] U drugom razdvajanju ilustruje se podela matrice C na dve jednake podmatrice po horizontali. Potom se pokazuje da funkcija `split()` kreira podmatrice poglede, te da će promena u nekoj od podmatrica rezultovati i promenom u polaznoj matrici. Konačno, treće razdvajanje pokazuje kako se navođenjem celobrojne sekvence (ili niza) brojeva za drugi argument, matrica može podeliti na više mesta određenih brojevima iz sekvence (niza). Konkretno, matrica C deli se vertikalno tako da rezultujuće podmatrice sadrže kolone specificirane torkom `(1, 4)`: prva - kolone sa indeksima od 0 do 1 ne uključujući 1, druga - kolone sa indeksima od 1 do 4 ne uključujući 4, te poslednja sa kolonama počevši od indeksa 4 pa do kraja.

Rearanžiranje elemenata matrice

U praksi se često javlja potreba da se elementi matrice preslože na određeni način tako da rezultujuća matrica sadrži *sve* polazne elemente, pri čemu se oblik matrice može ali i ne mora promeniti. U dosadašnjem izlaganju često je korišćena metoda `reshape()` koja menja oblik matrice vraćajući pogled na polaznu matricu. Slično, pominjana je i metoda `transpose()` za transponovanje, koja vraća pogled na polaznu matricu. Isti efekat postiže se i sa `X.T`. Sada će biti pomenute još neke funkcije i metode koje rearanžiraju elemente unutar matrice.

Funkcija `ravel()` i metoda `flatten()` preoblikuju matricu u *niz*, s tim da prva vraća pogled, a druga matricu kopiju:

```
>>> A = np.arange(4).reshape(2, 2)
>>> A
array([[0, 1],
       [2, 3]])
>>> x = np.ravel(A) # ravnanje po vrstama, x je pogled na A
>>> x
array([0, 1, 2, 3])
>>> x[0] = 10 #
>>> A
array([[10,  1],
       [ 2,  3]])
>>> y = A.flatten() # ravnanje po vrstama, y je kopija od A
>>> y
array([10,  1,  2,  3])
>>> y[0] = 0
>>> A
array([[10,  1],
       [ 2,  3]])
>>> np.ravel(A, order='F') # ravnanje po kolonama
```

[11] Funkcija `array_split()` ponaša se isto kao i funkcija `split()`, osim što dozvoljava da poslednja podela ne mora biti jednaka prethodnim – drugi argument može biti proizvoljan prirodni broj.

```
array([10,  2,  1,  3])
>>> A.reshape(-1) # kao ravel
array([10,  1,  2,  3])
```

Primer pokazuje da se u metodi `flatten()`, pri navođenju opcionog argumenta `order='F'`, preoblikovanje vrši po kolonama. Isti opcioni agument može se primeniti i na funkciju `ravel()`. Umesto funkcije `ravel()`, može se upotrebiti i `reshape(-1)`. Ako se *najviše na jednom* mestu, umesto pozitivnog broja koji predstavlja broj elemenata unutar posmatrane dimenzije, stavi -1, funkcija `reshape()` *sama preračunava* broj elemenata u toj dimenziji.

Često korišćene funkcije za rearanžiranje elemenata matrice, bez promene oblika, su `flip()` i `roll()`. Razmatra se prvo funkcija `flip()` koja *obrće* (invertuje) redosled elemenata u navedenim dimenzijama, vraćajući pri tome *pogled* na polaznu matricu:

```
>>> A = np.arange(9).reshape(3, 3)
>>> A
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
>>> np.flip(A, axis=0) # obrtanje po vertikali
array([[6, 7, 8],
       [3, 4, 5],
       [0, 1, 2]])
>>> np.flip(A, axis=1) # obrtanje po horizontali
array([[2, 1, 0],
       [5, 4, 3],
       [8, 7, 6]])
>>> B = np.flip(A) # obrtanje po obe dimenzije
>>> B
array([[8, 7, 6],
       [5, 4, 3],
       [2, 1, 0]])
>>> B[0, 0] = 100 # obrtanje rezultuje pogledom!
>>> A
array([[  0,   1,   2],
       [  3,   4,   5],
       [  6,   7, 100]])
>>> np.flip(A) == np.flip(A, axis=(0, 1))
array([[ True,  True,  True],
       [ True,  True,  True],
       [ True,  True,  True]])
```

Primer pokazuje kako se matrica obrće po vrstama (`axis=0`), kolonama (`axis=1`), ili po obe dimenzije (izostavljen drugi argument). U opštem slučaju, redosled elemenata može se obrnuti po svakoj od više dimenzija tako što se za opcioni argument navodi

torka sa indeksima odgovarajućih osa (videti poslednju operaciju u primeru).

Funkcija `roll()` vrši *kružno* pomeranje elemenata matrice u zadatim dimenzijama željeni broj puta:

```
>>> A = np.arange(9).reshape(3, 3)
>>> A
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
>>> np.roll(A, 1, axis=0) # pomeranje po vertikali
array([[6, 7, 8],
       [0, 1, 2],
       [3, 4, 5]])
>>> np.roll(A, -2, axis=1) # pomeranje po horizontali
array([[2, 0, 1],
       [5, 3, 4],
       [8, 6, 7]])
>>> np.roll(A, 1) # pomeranje uz prethodno ravnanje
array([[8, 0, 1],
       [2, 3, 4],
       [5, 6, 7]])
>>> B = np.roll(A, 1) # B je kopija od A!
>>> B[0, 0] = 100
>>> A
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
>>> np.roll(A, (1, 1), axis=(0, 1)) # pomeranje po obe dimenzije
array([[8, 6, 7],
       [2, 0, 1],
       [5, 3, 4]])
```

Kao što se iz primera vidi, pomeranje za jedan po vrstama rezultuje matricom čija je druga vrsta jednaka prvoj vrsti matrice `A`, treća vrsta jednaka drugoj i tako redom. Prva vrsta rezultujuće matrice jednaka je poslednjoj vrsti matrice `A`. Pomeraj može biti i negativan: pomeranje za -2 po kolonama rezultuje matricom čija je prva kolona jednaka trećoj koloni matrice `A`. Ako se osa izostavi, matrica se prvo *preoblikuje* u niz pa se pomeranje vrši unutar niza. Potom se dobijeni niz ponovo preoblikuje u matricu polaznog oblika (pokazano na matrici manjih dimenzija zbog prostora):

$$\begin{bmatrix} 0 & 1 \\ 2 & 3 \end{bmatrix} \xrightarrow{ravnanje} \begin{bmatrix} 0 & 1 & 2 & 3 \end{bmatrix} \xrightarrow{pomeranje +1} \begin{bmatrix} 3 & 0 & 1 & 2 \end{bmatrix} \xrightarrow{preoblikovanje} \begin{bmatrix} 3 & 0 \\ 1 & 2 \end{bmatrix}$$

Kraj prethodnog primera ilustruje da se pri kružnom pomeranju vraća matrica *kopija*. Ako se pomera po više dimenzija, te ako se pomeraji razlikuju, onda se za odgovarajuće argumente stavljaju torke jednakih dužina, na primer `np.roll(A, (1, 1), axis=(0,`

1)). Pomeranje se vrši po prvoj navedenoj osi za prvi navedeni pomeraj, potom po drugoj navedenoj osi za drugi navedeni pomeraj i tako redom.

Elementi matrice mogu se zarotirati pomoću funkcije rot90(). Funkcija vraća *rotirani* pogled na polaznu matricu, pri čemu se rotacija obavlja u ravni definisanoj dvema osama (podrazumevane ose su 0 i 1):

```
>>> A = np.arange(8).reshape(2, 4)
>>> A
array([[0, 1, 2, 3],
       [4, 5, 6, 7]])
>>> np.rot90(A, 1) # rotacija za 90 stepeni
array([[3, 7],
       [2, 6],
       [1, 5],
       [0, 4]])
>>> np.rot90(A, 2) # rotacija za 180 stepeni
array([[7, 6, 5, 4],
       [3, 2, 1, 0]])
>>> B = np.rot90(A, -1) # rotacija za -90 stepeni
>>> B
array([[4, 0],
       [5, 1],
       [6, 2],
       [7, 3]])
>>> B[0, 0] = 100
>>> A
array([[  0,   1,   2,   3],
       [100,   5,   6,   7]])
>>> C = np.stack((A, A), axis=0)
>>> C
array([[[  0,   1,   2,   3],
        [100,   5,   6,   7]],

       [[  0,   1,   2,   3],
        [100,   5,   6,   7]]])
>>> np.rot90(C, 1, axes=(0, 2)) # rotacija kvadra u horiz. ravni
array([[[  3,   3],
        [  7,   7]],

       [[  2,   2],
        [  6,   6]],

       [[  1,   1],
        [  5,   5]],

       [[  0,   0],
```

```
       [100, 100]]])
```

Prethodni primer pokazuje da se smer ugla određuje isto kao i u matematici, odnosno da je pozitivan ugao orijentisan u *suprotnom* smeru od kretanja kazaljke na časovniku. Poslednja rotacija u primeru obavlja se u ravni određenoj osama 0 i 2. Na kraju izlaganja o mogućim načinima za rearanžiranje elemenata matrice, navode se funkcije `tril()` i `triu()`. Ove funkcije *ne menjaju* oblik i *ne rearanžiraju* elemente matrice, već vraćaju donju, odnosno gornju *trougaonu matricu* kopiju:

```
>>> A = np.ones((4, 4))
>>> A
array([[1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.]])
>>> np.tril(A)
array([[1., 0., 0., 0.],
       [1., 1., 0., 0.],
       [1., 1., 1., 0.],
       [1., 1., 1., 1.]])
>>> np.tril(A, 1)
array([[1., 1., 0., 0.],
       [1., 1., 1., 0.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.]])
>>> np.triu(A)
array([[1., 1., 1., 1.],
       [0., 1., 1., 1.],
       [0., 0., 1., 1.],
       [0., 0., 0., 1.]])
>>> np.triu(A, -1)
array([[1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [0., 1., 1., 1.],
       [0., 0., 1., 1.]])
```

Slično funkciji `diag()`, funkcije `tril()` i `triu()` primaju i drugi (opcioni) argument kojim se definiše ispod koje, odnosno iznad koje dijagonale su nenulti elementi (podrazuemvana vrednost je 0 - glavna dijagonala). Razmatranje paketa NumPy završava se ilustrativnim primerom:

Problem 11.4 — Uglovi između vektora. Data je matrica $\mathbf{X}$ koja čuva m vektora iz $\mathbb{R}^n$. Ispisati sve parove vektora koji zaklapaju ugao veći (ili manji) od zadatog ugla u stepenima. ■

Neka su $\mathbf{x} =< x_1, \ldots, x_n >$ i $\mathbf{y} =< y_1, \ldots, y_n >$ dva vektora iz $\mathbb{R}^n$. Ugao između vektora izračunava se prema formuli:

$$\sphericalangle(\mathbf{x}, \mathbf{y}) = \arccos \frac{\mathbf{x} \cdot \mathbf{y}}{\|\mathbf{x}\| \cdot \|\mathbf{y}\|} \tag{11.1}$$

Skalarni proizvod dva vektora iz izraza (11.1) računa se kao $\mathbf{x} \cdot \mathbf{y} = \sum_{i=1}^{n} x_i y_i$, a norma kao $\|\mathbf{x}\| = \sqrt{\sum_{i=1}^{n} x_i^2}$. Imajući u vidu izraz (11.1), te da su vektori $\mathbf{x}_k$ smešteni u matricu $\mathbf{X}$ *po vrstama*, problem se može rešiti na sledeći način:

Program 11.4 — Uglovi između vektora.

```
import numpy as np

def uglovi(X):
    """ Kreira matricu uglova U, U[i,j] je ugao između
        vektora X[i] i X[j]. Vektori u vrstama matrice X. """

    # normalizacija vektora
    norme = np.sqrt((X**2).sum(axis=1)).reshape(-1, 1)
    X1 = X / norme

    # uglovi
    U = np.rad2deg(np.arccos(X1 @ X1.T))
    U[U < 0.01] = 0 # uglovi manji od stotog dela stepena - 0

    return U

def parovi_vektora(X, veće=True, alfa=90):
    """ Prikazuje vektore koji zaklapaju uglove
        veće (manje) od alfa. """

    U = uglovi(X)
    # pozicije vektora koji ispunjavaju uslov
    i_poz, j_poz = np.nonzero(U > alfa if veće else U < alfa)
    # ispis
    print(f"Ugao {'veći' if veće else 'manji'} od {alfa}")
    for k in range(len(i_poz)):
        if i_poz[k] < j_poz[k]:
            vi, vj = X[i_poz[k]], X[j_poz[k]]
            uij = U[i_poz[k], j_poz[k]]
            print(f"v1: {vi}\nv2: {vj}\nu: {uij}\n")

```

```
32 X = np.array([[1., 1.], [-1., 0.], [1., 0.5]])
33 parovi_vektora(X)
34 parovi_vektora(X, veće=False, alfa=75)
```

Problem se rešava pomoću funkcija `uglovi()` i `parovi_vektora()`. Prva funkcija formira matricu uglova **U** iz ulazne matrice **X**, pri čemu u_{ij} predstavlja ugao između vektora $\mathbf{x}_i$ i $\mathbf{x}_j$ (r3-15). Funkcija `parovi_vektora()` ispisuje sve parove koji zadovoljavaju uslov definisan pomoću dva opciona parametra: relacijom (ugao veći ili manji) i uglom (r17-31).

Da bi se izračunavanje uglova prema izrazu (11.1) svelo na izračunavanje skalarnih proizvoda između vektora, potrebno je da se svaki od njih prvo normalizuje (podeli svojim intenzitetom - normom). U r8 se formira niz koji čuva norme vektora. Pošto su vektori iz **X** memorisani po vrstama, sumiranje se obavlja po osi jedan. Kako bi se koordinate svakog vektora podelile odgovarajućom normom (r9), uz primenu mehanizma emitovanja, niz sa normama preoblikovan je u vektor kolonu (r8). Svi potrebni skalarni proizvodi sada se mogu dobiti množenjem matrice **X1**, koja sadrži normirane polazne vektore, sa njenom transponovanom varijantom (r12). Funkcija `arccos()` vraća ugao između 0 i π radijana pa je u r12 upotrebljena funkcija koja radijane prebacuje u stepene – `rad2deg()`. Svi uglovi manji od stotog dela stepena su, radi preglednijeg ispisa, postavljeni na nulu (r13).

Pozicije uglova koji zadovoljavaju traženi uslov dobijaju se pomoću funkcije `nonzero()` (r23). Podsetiti se da su elementi matrice `U > alfa` (ili `U < alfa`) logičke vrednosti, te da se logička istina (`True`) tretira kao nenulta vrednost. Nizovi `i_poz` i `j_poz` čuvaju redne brojeve vrsta, odnosno kolona za uglove koji zadovoljavaju traženi uslov. U isto vreme, ovi indeksi označavaju i parove vektora koji se štampaju u (r26-31). Uslov iz r28 je tako postavljen da se biraju samo uglovi čiji je indeks vrste manji od indeksa kolone, odnosno uglovi *iznad* glavne dijagonale matrice `U`. Ovako je učinjeno jer je matrica uglova *simetrična* – ugao između vektora $\mathbf{x}_i$ i $\mathbf{x}_j$ jednak je uglu između vektora $\mathbf{x}_j$ i $\mathbf{x}_i$. Sledi prikaz rada programa za tri vektora i slučajeve kada je traženi ugao veći od 90, odnosno manji od 75 stepeni:

```
Ugao veći od 90
v1: [1. 1.]
v2: [-1.  0.]
u: 135.0

v1: [-1.  0.]
v2: [1.  0.5]
u: 153.434948822922
```

```
Ugao manji od 75
v1: [1. 1.]
v2: [1.  0.5]
u: 18.434948822922035
```

11.2 Grafičko predstavljanje podataka - Matplotlib

Česta aktivnost u analizi podataka primenom računara odnosi se na njihovo *grafičko predstavljanje*. Pomoću adekvatnih grafika lakše se otkrivaju pravilnosti i trendovi unutar podataka pa eksperti mogu lakše da ih interpretiraju. Za tu svrhu najčešće se koristi paket *Matplotlib*.[12] Njega čine klase i funkcije pomoću kojih se podaci grafički predstavljaju nekim od standardnih grafikona poput linijskog, pravougaonog ili kružnog dijagrama. Ovde će biti pokazane osnovne tehnike prikaza podataka bazirane na funkcijama potpaketa *Pyplot*, koji simulira grafičke alate komercijalnog softvera Matlab.

! Podaci se, po svojoj prirodi, dele na *numeričke*, *kategorijske* i *ordinalne*. Numerički podaci (fizičke veličine, prihodi, trajanja, ...) imaju *kvantitativnu* prirodu. Njihove vrednosti mogu da se međusobno razlikuju i porede, kao i da se nad njima obavljaju matematičke operacije. Kategorijski podaci (nacionalnost, boja, profesija, ...) imaju *kvalitativnu* prirodu i jedino se mogu međusobno razlikovati. Ordinalni podaci (rangiranja, preferencije, ...) su po karakteristikama između prva dva tipa i mogu da se razlikuju i porede.

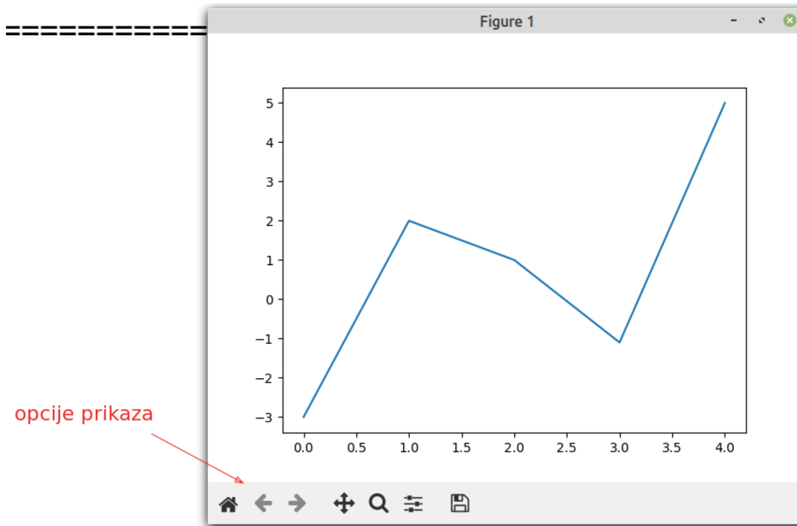
11.2.1 Linijski dijagram i osnovni elementi grafičkog prikaza

Linijski dijagram obično predstavlja *promenu* posmatranog numeričkog podatka u vremenu, ili *funkcionalnu zavisnost* između dve numeričke veličine. Čini ga niza tačaka u ravni zadatih x i y koordinatom, pri čemu su tačke spojene linijskim segmentima. Sledeći primer ilustruje kreiranje najprostijeg linijskog dijagrama:

```
1  import matplotlib.pyplot as plt
2  
3  x = [-3, 2, 1, -1.1, 5] # podaci za prikaz
4  plt.plot(x) # kreira linijski dijagram
5  plt.show() # otvara graficki prozor
```

[12] Paket inicijalno nije dostupan u okviru IDLE-a, već ga treba instalirati. Na operativnom sistemu Windows, na kome je Pajton već instaliran, pokrenuti iz komandne linije `pip install matplotlib`. Slično, na operativnom sistemu Linux, pokrenuti iz komandne linije `python3.x -m pip install matplotlib` (ovde se x odnosi na instaliranu verziju Pajtona).

Potpaket Pyplot uvodi se u program u r1 – uobičajeno je da se tom prilikom koristi alijas `plt`. *Serija* numeričkih podataka za prikaz memorisana je u listi `x`. Funkcija `plot()` formira linijski dijagram koji, pod rednim brojevima $0, 1, 2, 3$ i 4, prikazuje vrednosti iz serije: $-3, 2, 1, -1.1$ i 5 (r4). Linijski dijagram formira se u *memoriji*, ali se ne prikazuje sve dok se u r5 ne pozove funkcija `show()`. Slika 11.5 ilustruje efekat funkcije `show()` – interpreter kreira prozor sa grafikonom i *opcijama prikaza*. Opcije omogućavaju kretanje kroz prikaz upotrebom miša (engl. *Pan*), uvećanje (umanjenje) prikaza (engl. *Zoom*) i snimanje prikaza u nekom od formata za čuvanje slika.



Slika 11.5: Jednostavni linijski dijagram. Pored alatki *Pan*, *Zoom* i *Save* (četvrta, peta i sedma), opcije prikaza sadrže i alatke za navigaciju kroz različite prikaze – inicijalni (prva), prethodni (druga) i sledeći (treća alatka).

Funkcija `plot()` najčešće prima x i y koordinate tačaka koje će biti povezane linijskim segmentima i *tekstualni opis* boje i tipa linije i *markera* za prikaz tačaka, što je ilustrovano u sledećem primeru:

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  x = np.arange(1, 13)
5  y = 116 + 2 * np.random.rand(12)
6
7  plt.plot(x, y, "go-") # kreira linijski dijagram
8  plt.xlabel("mesec") # opis x ose
9  plt.ylabel("kurs evra") # opis y ose
```

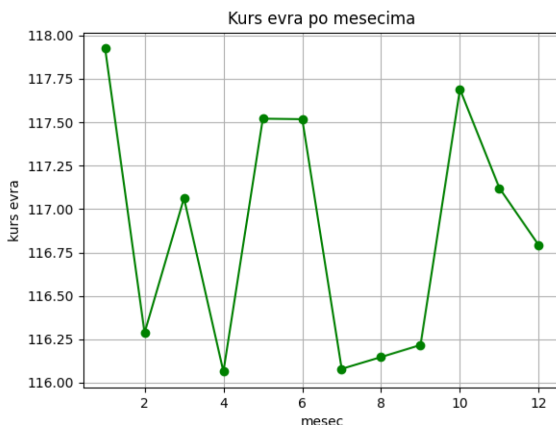
```
10 plt.title("Kurs evra po mesecima") # ime grafikona
11 plt.grid() # mreža pomoćnih linija
12 plt.show()
```

Redovima 4 i 5 definisane su koordinate tačaka koje treba grafički predstaviti. Pored standardnih sekvenci u Pajtonu, funkcija `plot()` prima i nizove iz paketa NumPy (r7). Tekstualni opis `"go-"` ima sledeće značenje: `"g"` predstavlja liniju zelene boje, `"o"` postavlja marker tačke na simbol za krug, a `"-"` čini da linija bude kontinualna. U opisu se, kada se on navede, mogu izostaviti najviše dva elementa specifikacije. Na primer, podrazumevana vrednost ovog opcionog parametra je `"b-"`: plava kontinualna linija bez markera tačke. Često korišćeni opisi prikazani su u tabeli 11.5.

Boja		Marker		Tip linije	
`b`	plava	`o`	krug	`-`	kontinualna
`r`	crvena	`^`	trougao	`:`	tačkasta
`g`	zelena	`*`	zvezda	`--`	crtkasta
`y`	žuta	`s`	kvadrat	`-.`	crta-tačka
`k`	crna	`d`	dijamant		

Tabela 11.5: Često korišćeni opisi linijskog prikaza.

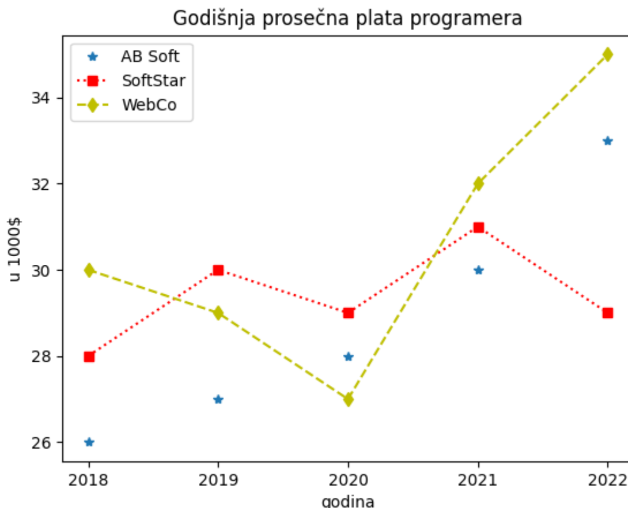
U r8-11 definišu se često korišćeni elementi grafičkog prikaza: tekstualna oznaka za x (r8) i y osu (r9), ime grafikona (r10) i uključivanje mreže pomoćnih linija (r11). Kada su svi elementi definisani, grafički prozor aktivira se pomoću funkcije `show()` - slika 11.6.



Slika 11.6: Elementi linijskog dijagrama.

U okviru jednog prikaza može se prikazati veći broj linijskih dijagrama tako što se, pre poziva funkcije `show()`, navede više poziva funkcije `plot()`. Sledeći primer ilustruje tri linijska dijagrama sa različitim elementima prikaza:

```
import matplotlib.pyplot as plt

godine = range(2018, 2023)
p1, p2 = [26, 27, 28, 30, 33], [28, 30, 29, 31, 29]
p3 = [30, 29, 27, 32, 35]

plt.plot(godine, p1, "*", label="AB Soft")
plt.plot(godine, p2, "rs:", label="SoftStar")
plt.plot(godine, p3, "yd--", label="WebCo")
plt.title("Godišnja prosečna plata programera")
plt.xlabel("godina") # opis x ose
plt.ylabel("u 1000$") # opis y ose
plt.xticks(godine) # gde su labele na x osi
plt.legend() # legenda
plt.show()
```



Slika 11.7: Višestruki linijski prikaz sa legendom.

Funkcija `plot()` omogućava da se, preko opcionog parametra `label` (r7-9), definiše naziv koji će biti naveden u okviru *legende* dijagrama. Slično mreži pomoćnih linija, prikazuje se na zahtev pozivom funkcije `legend()` u r14. Pored navedenih, `plot()`

može da primi i druge opcione parametre – na primer, `linewidth` za debljinu linije. Konačno, funkcijom `xticks()` može se uticati na mesta na kojima će se pojaviti labele na x osi (r13) – u primeru se one pojavljuju samo za godine definisane opsegom iz r3. Za detaljniji opis opcija kod linijskog (kao i drugih) dijagrama, čitalac se upućuje na dodatnu literaturu o Matplotlib-u navedenu u odeljku Bibliografija.

Grafik funkcionalne zavisnosti

Prikaz funkcionalne zavisnosti dve numeričke veličine svodi se na iscrtavanje grafika funkcije $y = f(x), x \in [a, b]$. Prikaz se jednostavno realizuje linijskim dijagramom uz korišćenje potrebnih funkcija iz paketa NumPy. Često se, u okviru iste slike, formira *više* prikaza sa različitim prikaznim elementima poput naslova ili oznaka osa. Ovi prikazi mogu da sadrže i različite tipove dijagrama. Kao primer navodi se formiranje slike koja sadrži linijske dijagrame četiri realne funkcije u četiri *odvojena* dela grafičkog prozora:

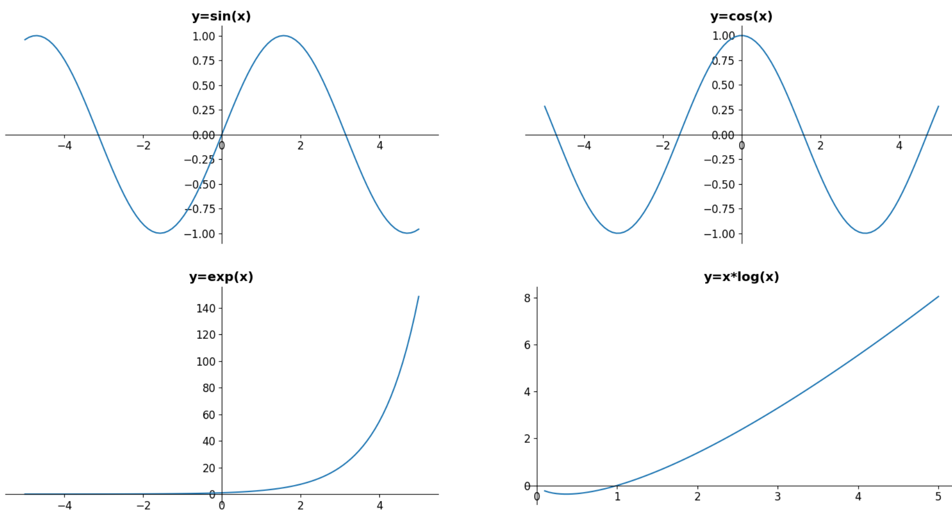
```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  def grafik(prikaz, naslov, x, y):
5      """Crta grafik funkcije zadate vrednostima za x i y"""
6      # postavlja koordinatne ose
7      prikaz.spines['left'].set_position('zero')
8      prikaz.spines['right'].set_color('none')
9      prikaz.spines['bottom'].set_position('zero')
10     prikaz.spines['top'].set_color('none')
11     plt.title(naslov)
12     plt.plot(x, y)
13
14 x1, x2 = np.linspace(-5, 5, 100), np.linspace(0.1, 5, 100)
15
16 plt.rc("font", size=12) # postavlja veličinu fonta
17 plt.rc("axes", titleweight="bold") # naslov je podebljan
18
19 gore_levo = plt.subplot(221)  # prvi grafikon
20 grafik(gore_levo, "y=sin(x)", x1, np.sin(x1))
21
22 gore_desno = plt.subplot(222) # drugi grafikon
23 grafik(gore_desno, "y=cos(x)", x1, np.cos(x1))
24
25 dole_levo = plt.subplot(223)  # treći grafikon
```

```
26 grafik(dole_levo, "y=exp(x)", x1, np.exp(x1))
27
28 dole_desno = plt.subplot(224) # četvrti grafikon
29 grafik(dole_desno, "y=x*log(x)", x2, x2 * np.log(x2))
30
31 plt.show()
```

Dva niza od 100 jednako udaljenih tačaka između -5 i 5, odnosno 0.1 i 5, definisana su u r14. Pomoću funkcije `rc()` postavlja se veličina fonta, stil ispisa naslova, kao i druge osobine teksta u prozoru (r16-17). Funkcija `subplot()` poziva se u r19, r22, r25 i r28 kako bi se, u okviru iste slike, formirala četiri odvojena prikaza. Funkcija vraća referencu na *tekući prikaz* u okviru slike sa matričnom strukturom – na primer, za argument `221`, vraćena referenca odnosi se na prvi prikaz slike podeljene na dve vrste i dve kolone (numeracija po vrstama počevši od prve). Tekući prikaz prosleđuje se funkciji `grafik()` (r4-12) koja crta funkcije definisane vrednostima x i y (r20, r23, r26, r29). Kako se svi delovi slike formiraju u memoriji, grafički prozor se "otvara" u r31.

Linijski dijagram crta se u tekućem prikazu (parametar `prikaz`) na već poznat način (r12). Primetiti da svaki prikaz ima drugačiji naslov (r11). Kada bi se grafici funkcija crtali na istom prikazu, ne bi bilo moguće definisati različite naslove. Slično, različiti rasponi za vrednosti x i y mogli bi da poremete čitljivost zajedničkog prikaza.

U r7-10 postavljaju se koordinatne ose tekućeg prikaza. Matplotlib podržava *četiri* ose koje ograničavaju grafički prikaz podataka i koje se mogu postaviti na proizvoljne vrednosti. Leva i donja osa iz primera postavljene su na nulu, a gornja i desna su načinjene nevidljivim. Rad programa ilustrovan je na slici 11.8.



Slika 11.8: Grafik realne funkcije jedne promenljive.

11.2.2 Pravougaoni i kružni dijagram. Histogram

Pravougaoni dijagram[13] najčešće prikazuje numeričke vrednosti pridružene pojedinim *kategorijskim* podacima. Njime se mogu *porediti* i numeričke vrednosti dodeljene različitim objektima unutar iste kategorijske podele. Sledeći primer ilustruje obe primene, uz korišćenje horizontalnog i vertikalnog pravougaonog dijagrama:

```
import matplotlib.pyplot as plt
import numpy as np

kategorije = ["Aus. Open", "Rol. Garros", "Wimbledon", "US Open"]
nole, rafa = [10, 3, 7, 4], [2, 14, 2, 4]

plt.rc("font", size=12)
plt.subplot(131)
plt.title("Nole")
plt.barh(kategorije, nole, height=0.15)

plt.subplot(132)
plt.title("Rafa")
plt.bar(kategorije, rafa, color="red", width=0.25)

plt.subplot(133)
plt.title("Nole vs Rafa")
x = np.arange(4)
plt.bar(x-0.2, nole, width=0.4, label="nole")
plt.bar(x+0.2, rafa, width=0.4, label="rafa")
plt.xticks(x, kategorije)
plt.legend()

plt.show()
```

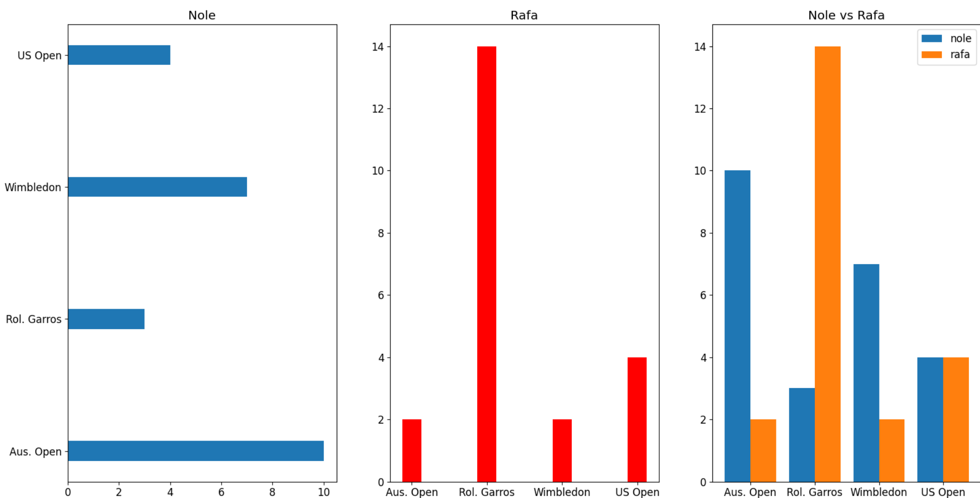
U primeru se iscrtavaju grafici koji pokazuju raspodelu Grand Slam titula za Novaka Đokovića i Rafaela Nadala. Kategorijski podaci koji predstavljaju Grand Slam turnire, a koji idu na x osu za vertikalni i y osu za horizontalni pravougaoni dijagram, definisani su u r4. Numerički podaci koji su pridruženi turnirima predstavljaju broj titula za oba tenisera (r5).

Primer ilustruje upotrebu tri dijagrama u okviru iste slike, organizovane u obliku matrice sa jednom vrstom i tri kolone. Za postavljanje tekućeg prikaza za iscrtavanje koristi se već pomenuta funkcija `subplot()` – (r8, r12 i r16). Prvi dijagram prikazuje

[13] Engl. *bar chart*.

raspodelu Novakovih titula pomoću horizontalnih pravougaonika (r10) – slika 11.9, levo. Funkcija `barh()` prihvata bar dva parametra, sekvencu sa kategorijama i sekvencu sa pridruženim numeričkim vrednostima. Parametar `height` definiše visinu horizontalnog pravougaonika. Slično, funkcija `bar()` iscrtava vertikalni dijagram koji pokazuje raspodelu Nadalovih titula (r14) – slika 11.9, sredina. Parametar `width` definiše širinu vertikalnog pravougaonika, a `color` upotrebljenu boju ispune.

Vertikalni dijagram, koji poredi Novakove i Nadalove titule u okviru jednog prikaza, realizovan je u r16-22 – slika 11.9, desno. Funkcija `bar()` poziva se dva puta, prvo da prikaže Novakove titule (r19), a potom i Nadalove (r20). Kada se boja ispune ne navede, a `bar()` se pozove više puta u okviru istog prikaza, Matplotlib menja tekuću boju. Kategorijske vrednosti (imena turnira) ispisuju se na x osi u tačkama 0, 1, 2 i 3 (r21). Ako `xticks()` primi dva argumenta, onda drugi označava labele koje se pojavljuju na koordinatama definisanim prvim. Kako bi se odgovarajući pravougaonici *dodirivali*, potrebno je pažljivo proračunati x koordinate njihovih *centralnih* tačaka (centralna tačka polovi osnovu pravougaonika). Zbog širine od 0.4, centralne tačke Novakovih pravougaonika postavljene su u $0-0.2, 1-0.2, 2-0.2$ i $3-0.2$ (r19), a Nadalovih u $0+0.2, 1+0.2, 2+0.2$ i $3+0.2$ (r20).



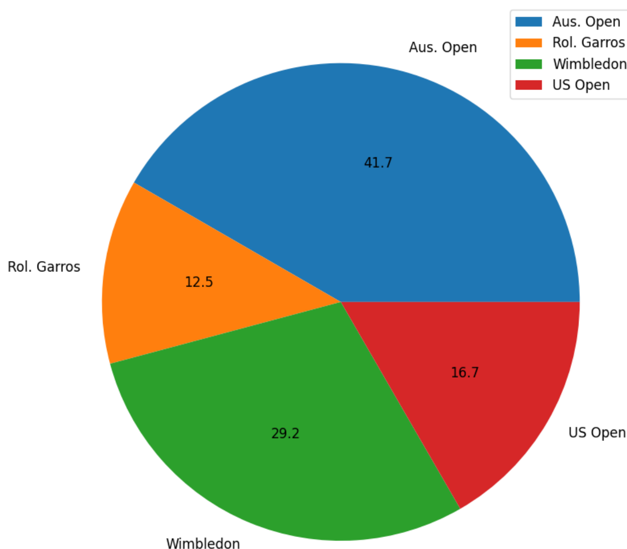
Slika 11.9: Horizontalni, vertikalni i pravougaoni dijagram sa dve grupe podataka.

Sledeći dijagram koji se razmatra je *kružni dijagram*.[14] On prikazuje *procentualno učešće* posmatranih kategorija u odnosu na neku numeričku veličinu. Procentualno učešće svakog od četiri najbitija turnira u Novakovoj kolekciji Grand Slam titula može se prikazati kružnim dijagramom na sledeći način:

[14] Engl. *pie chart*.

```
import matplotlib.pyplot as plt

kategorije = ["Aus. Open", "Rol. Garros", "Wimbledon", "US Open"]
gs_titule = [10, 3, 7, 4]

plt.rc("font", size=12)
plt.pie(gs_titule, labels=kategorije, autopct="%3.1f")
plt.legend()
plt.show()
```

Funkcija `pie()`, pozvana u r7, prima tri argumenta: prvi predstavlja numeričku seriju čiji *zbir* predstavlja 100% površine kružnog dijagrama, drugi memoriše kategorijske labele, a treći se odnosi na format ispisa za vrednosti (procente) na kružnim odsečcima. Format `%3.1f` odnosi se na decimalni broj sa tri cifre od kojih je jedna iza decimalne tačke. Rezultat rada programa prikazan je na slici 11.10.



Slika 11.10: Kružni dijagram: procentualno učešće za sve Novakove grand slam titule.

Za razliku od pravougaonog dijagrama koji prikazuje frekvencije kategorijskih podataka, *histogram* predstavlja grafikon frekvencija za *kontinualne* numeričke podatke. On uključuje *opsege* podataka grupisane po intervalima[15] na x osi, sa pridruženim frekvencijama na y osi. Sledi tipičan primer za upotrebu histograma koji prikazuje raspodelu visina u okviru izabrane populacije ljudi:

[15] Intervali formiraju *korpe podataka* čije se vrednosti posmatrane veličine nalaze u zadatom intervalu – na engleskom se ove korpe nazivaju *data bins*.

```
import matplotlib.pyplot as plt
import numpy as np

visine = np.random.normal(172, 10, 1000)
x = np.round(np.linspace(visine.min(), visine.max(), 11), 1)

plt.rc('font', size=13)
plt.subplot(121)
plt.title("Visine - raspodela frekvencija")
plt.hist(visine, bins=10, color="red")
plt.xticks(x)

plt.subplot(122)
plt.title("Visine - kumulativna kriva")
plt.hist(visine, bins=10, cumulative=True)
plt.xticks(x)
plt.show()
```

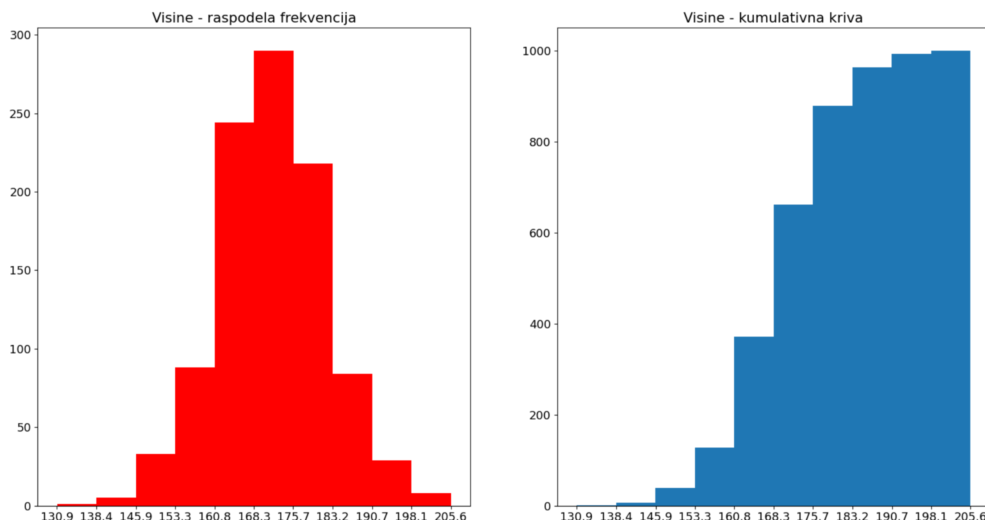
U programu se prvo formira niz od 1000 visina koje podležu normalnoj raspodeli sa očekivanom visinom od 172 cm i standardnom devijacijom od 10 cm (r4). Potom se, putem niza `x`, definišu granice 10 jednakih intervala širine $\frac{\max visine - \min visine}{10}$ (r5). Zbog preglednijeg ispisa, granice intervala se zaokružuju na jednu decimalu. Potom se, upotrebom funkcije `hist()`, formiraju dva histograma: prvi prikazuje raspodelu visina po deset intervala (parametar `bins`), uz upotrebu crvene boje (parametar `color`, r10); drugi histogram je kumulativan (parametar `cumulative=True`, r15) – slika 11.11.

Slika pokazuje da u populaciji ima približno 240 ljudi čija se visina kreće između 160.8 cm i 168.3 cm, odnosno da približno 380 ljudi ima visinu do 168.3 cm (kumulativni dijagram). Podela na x osi se postavlja tako da se ispisuju granice intervala (r11, 16). Da to nije učinjeno, podela bi bila sačinjena od celobrojnih umnožaka broja 10 iz intervala $[\min visine, \max visine]$.

Primetiti da se, za razliku od pravougaonog dijagrama, kod histograma pravougaonici *dodiruju*. To je zbog prirode podataka na x osi - podaci su kontinualni, a intervali koji se nastavljaju jedan na drugi su *sortirani* u rastući redosled. Kod pravougaonog dijagrama redosled kategorija, pa samim tim i pravougaonika, može se proizvoljno menjati.

11.2.3 Dijagram površi - grafik realne funkcije dve promenljive

Grafik realne funkcije $z = f(x,y)$, $x \in [a,b]$, $y \in [c,d]$, predstavlja površ u prostoru prikazanu *dijagramom površi*. Da bi se grafik iscrtao, neophodno je izračunati vred-



Slika 11.11: Histogram: raspodela frekvencija i kumulativni dijagram visina u populaciji.

nost funkcije f za sve kombinacije vrednosti x i y iz navedenog pravougaonika u xOy ravni. Kako traženi intervali sadrže beskonačno mnogo vrednosti, pribegava se njihovoj *diskretizaciji*: vrednosti za x i y dobijaju se upotrebom funkcija `linspace()` ili `arange()` iz paketa NumPy, kao i u slučaju crtanja funkcije jedne promenljive. Tačke iz pravougaonika $x \in [a,b]$, $y \in [c,d]$, u kojima se računa vrednost $z = f(x,y)$, čine *pravougaonu mrežu*, a njihove koordinate dobijaju se primenom NumPy funkcije `meshgrid()`:

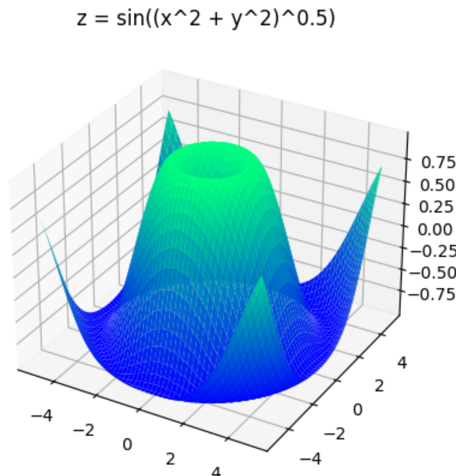
```
>>> import numpy as np
>>> x, y = np.arange(3), np.arange(3, 5)
>>> X, Y = np.meshgrid(x, y)
>>> X
array([[0, 1, 2],
       [0, 1, 2]])
>>> Y
array([[3, 3, 3],
       [4, 4, 4]])
```

Funkcija `meshgrid()` vraća dve matrice (`X` i `Y`) koje čuvaju koordinate tačaka iz pravougaone mreže – tačka iz i-te vrste i j-te kolone mreže ima koordinate `X[i, j]`, `Y[i, j]`. Program za kreiranje dijagrama površi koristi metodu `plot_surface()` klase koja modeluje tekući prikaz u okviru slike:

```
1 import matplotlib.pyplot as plt
```

```
2  import numpy as np
3
4  x, y = np.linspace(-5, 5, 100), np.linspace(-5, 5, 100)
5  X, Y = np.meshgrid(x, y)
6  Z = np.sin(np.sqrt(X**2 + Y**2))
7
8  tekuci = plt.subplot(projection="3d")
9  plt.title("z = sin((x^2 + y^2)^0.5)")
10 tekuci.plot_surface(X, Y, Z, cmap="winter")
11 plt.show()
```

Funkcija $z = \sin\sqrt{x^2+y^2}$, $x,y \in [-5,5]$, definisana je matricama `X`, `Y` i `Z` (r4-6). Intervali za promenljive veličine podeljeni su na 99 jednakih delova pa se kvadratna mreža u xOy ravni sastoji od 100×100 tačaka. Tekući prikaz kreira se primenom već poznate funkcije `subplot()` (r8). Opcioni parametar `projection="3d"` definiše trodimenzionalnu prirodu prikaza. Prilikom crtanja površinskog dijagrama koristi se *paleta boja* „winter“ (r10). Paleta omogućava da se tačke na površi prikažu različitim bojama (ili nijansama iste boje) u zavisnosti od vrednosti funkcije $z = f(x,y)$. Matplotlib podržava različite palete boja čija se imena (i karakteristike) mogu pronaći u svakom uputstvu za korišćenje paketa. Ako se u metodi `plot_surface()` ne navede neka od podržanih paleta, podrazumeva se da će sve tačke biti obojene *istom* nijansom *plave* boje. Efekat rada programa prikazan je na slici 11.12.



Slika 11.12: Grafik realne funkcije dve promenljive.

ŠTA JE NAUČENO

- Programski paket NumPy predstavlja skup klasa i funkcija za rad sa matricama. Matrica se predstavlja kao objekat klase `numpy.ndarray`.
- Matrice se karakterišu brojem dimenzija (dimenzionalnost), brojem elemenata po svakoj dimenziji (oblik) i jedinstvenim tipom svojih elemenata (celobrojne, realne, kompleksne, logičke i druge matrice).
- Jednodimenzionalne matrice nazivaju se još i nizovima.
- Budući da su elementi matrice istorodni objekti smešteni u kontinualni memorijski prostor, te da se operacije mogu vršiti nad delovima ili celim matricama istovremeno, rad sa matricama je značajno brži nego ako se one realizuju pomoću listi.
- Pojedinačnim elementima matrice može se pristupiti pomoću osnovnog indeksiranja navođenjem njihove pozicije u okviru svake od dimenzija. Na primer, `A[2, 3, 0]` označava element koji se nalazi u trećoj ravni, u preseku četvrte vrste i prve kolone (broji se od nule).
- Različitim podmatricama matrice pristupa se najčešće pomoću osnovnog ili naprednog indeksiranja. Prilikom osnovnog indeksiranja koriste se opsezi definisani dvotačkom. Na primer, sa `A[:, 2]` pristupa se trećoj koloni matrice, dok se, sa `A[2:, 1::2]`, pristupa podmatrici koja obuhvata sve vrste počevši od treće i svaku drugu kolonu počevši od druge.
- Napredno indeksiranje za indekse koristi sekvence u Pajtonu ili druge NumPy matrice. Elementi ovih sekvenci/matrica mogu biti i logičke vrednosti `True` i `False`. Na primer, iz niza `A` se, sa `A[[0, 3, 4]]`, izdvaja podniz koji se sastoji od prvog, četvrtog i petog elementa, a sa `B[B > 0]`, iz matrice `B` izdvajaju se u niz svi pozitivni elementi – `B > 0` je logička matrica čiji su elementi jednaki `True` ako su odgovarajući elementi u `B` pozitivni.
- NumPy nudi brojne funkcije za kreiranje, ažuriranje i transformaciju matrica, te različite matrične operacije poznate iz linearne algebre.
- Prilikom obavljanja matričnih operacija može se koristiti funkcionalni ili objektni pristup. Na primer, kada se sumiraju svi elementi matrice `A`, može se pisati `numpy.sum(A)` ili `A.sum()`. Moguća je i upotreba standardnih operatora poput `+` (sabiranje), `*` (množenje elemenata na istim pozicijama), ili `@` (matrično množenje).
- Ako su dve matrice istog oblika, onda se mnoge matrične operacije izvode tako što se operacija primenjuje na sve parove elemenata na istim pozicijama.
- Različite funkcije, koje za argument uzimaju neku matricu, primenjuju se pojedinačno na sve njene elemente. Na primer, `numpy.sin(A)` rezultuje matricom čiji su elementi sinusi elemenata matrice `A`.

- Operacije u kojima učestvuju matrice i skalari obavljaju se prema pravilima linearne algebre. Na primer, `2 * A + 3` daje matricu čiji su svi elementi jednaki odgovarajućim elementima matrice `A` pomnoženim sa dva pa uvećanim za tri.

- Kao rezultat najvećeg broja matričnih funkcija, metoda i operacija, nastaju matrice kopije. Tom prilikom kreiraju se novi matrični objekti čiji se elementi nalaze u kontinualnoj memoriji odvojeno od polaznih matrica. Na primer, za realnu matricu `A`, `A.round(2)` rezultuje matricom kopijom čiji su elementi zaokruženi na dve decimale, a polazna matrica `A` ostaje nepromenjena.

- Neke funkcije, primenjene na matrice, vraćaju matrice poglede. Pogled čuva informaciju o dimenzionalnosti i novom obliku matrice, a elementi mu se nalaze u istoj memorijskoj zoni koja odgovara polaznoj matrici. Na primer, ako je `A` matrica sa četiri vrste i četiri kolone, posle `B=A.reshape(2, 8)`, matrica `B` je pogled sa dve vrste i osam kolona. Ako bi se neki element matrice `B` promenio, onda bi se promenio i odgovarajući element polazne matrice `A`!

- Osnovno indeksiranje vraća matrice poglede, a napredno matrice kopije. Na primer, neka je `A` niz elemenata `[1,2,3,2,1]`. Tada je `B=A[:2]` pogled koga čine prva dva elementa iz `A`. Sada bi `B[0]=10` promenilo ne samo `B`, već i `A` (sada `[10,2,3,2,1]`).

- Ako prilikom izvođenja neke operacije dve matrice nisu istog oblika, onda se operacija može obaviti samo pod uslovom da su dve matrice kompatibilne za postupak emitovanja. U postupku emitovanja, efekat je kao da se jedna od matrica proširuje (ili obe) tako da dve rezultujuće matrice imaju isti oblik:

$$\begin{bmatrix} 0 \\ 1 \end{bmatrix} + \begin{bmatrix} 0 & 1 \end{bmatrix} \xrightarrow{po\ emitovanju} \begin{bmatrix} 0 & 0 \\ 1 & 1 \end{bmatrix} + \begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 2 \end{bmatrix}$$

- Neke matrične metode/funkcije primaju opcioni argument `axis` kojim se navodi po kojoj će osi (dimenziji) operacija biti obavljena. Na primer, za dvodimenzionalnu matricu `A`, `A.sum(axis=0)` vraća niz čiji su elementi sume po kolonama matrice – za fiksiranu kolonu menjaju se indeksi unutar prve dimenzije (osa 0, indeksi vrsta). Slično, `A.sum(axis=1)` vraća niz čiji su elementi sume po vrstama matrice.

- Programski paket Matplotlib predstavlja skup klasa i funkcija za kreiranje grafičkog prikaza podataka.

- Podaci mogu biti numerički (razlikuju se, porede se i nad njima se mogu obavljati matematičke operacije), kategorijski (razlikuju se) i ordinalni (razlikuju se i porede se).

- Linijski dijagram prikazuje promenu neke numeričke veličine u odnosu na neku drugu numeričku ili ordinalnu veličinu.

- Pravougaoni dijagram prikazuje numeričke veličine pridružene pojedinim kategorijskim podacima, a najčešće su to frekvencije kategorijskih podataka.

- Kružni dijagram prikazuje procentualno učešće posmatranih kategorija u

odnosu na neku numeričku veličinu.

- Histogram prikazuje frekvencije za kontinualne numeričke podatke.

Bibliografija

1. Guido van Rossum, *Python Tutorial*, Python Software Foundation, 2013

2. John V. Guttag, *Introduction to Computation and Programming Using Python: with application to computational modeling and understanding data*, 2021 edition, MIT Press, 2021

3. Bradley N. Miller, David L. Ranum, *Problem Solving with Algorithms and Data Structures using Python*, second edition, Franklin Beedle Publishers, 2011

4. Robert Johansson, *Numerical Python : Scientific Computing and Data Science Applications with Numpy, SciPy and Matplotlib*, second edition, Apress®, 2019

5. *NumPy user guide*, `https://numpy.org/doc/stable/user/index.html`, 2022

6. *Matplotlib user guide*, `https://matplotlib.org/stable/users/index.html`, 2022

Slike

1.1 Sistem i model - različiti nivoi apstrakcije 2
1.2 Fon Nojmanov model računara 4
1.3 Izvršavanje programa u procesoru 5
1.4 Pseudokod i dijagram toka 7
1.5 Pogađanje broja - gruba sila i podeli pa vladaj 8
1.6 Pogađanje broja - efikasnost algoritma 9
1.7 Problem trgovačkog putnika 9
1.8 Asemblerski i mašinski jezik 12
1.9 Viši programski jezik 12
1.10 Popularnost programskih jezika u januaru 2023. godine 16

2.1 Razvojno okruženje IDLE 20
2.2 Unos programa u editoru 23
2.3 Promenljive kao objektne reference 30
2.4 Razmena vrednosti 32
2.5 Nepromenljivost objekata 37

3.1 Sekvencijalni, razgranati i repetitivni algoritmi 40
3.2 Varijante grananja 41
3.3 Dijagram toka petlje while 45
3.4 Iterativni algoritmi 54
3.5 Stepenice 57

3.6 Pohlepna pretraga 63

4.1 Imenski prostori i tabela imena 81
4.2 Prenos parametara i opseg važenja promenljive 84
4.3 Stablo rekurzije 93
4.4 Hanojske kule - tri diska 96
4.5 Hanojske kule - rekurzivni postupak 96

5.1 Kolekcije i objektne reference 104
5.2 Sekvence i indeksiranje 104
5.3 Nepromenljivost teksta 111
5.4 Palindromi 112
5.5 Struktura liste 119
5.6 Izdvajanje podliste i dodela vrednosti 120
5.7 Spajanje i kopiranje listi 121
5.8 Eratostenovo sito 122
5.9 Podela teksta na fragmente 127
5.10 Detekcija sličnosti studentskih radova 135
5.11 Rečnici: primer telefonskog imenika 138
5.12 Prevođenje uz pomoć rečnika 141
5.13 Generisanje kombinacija bez ponavljanja 151
5.14 Problem četiri dame 158

6.1 Funkcija gustine raspodele 172
6.2 Stablo igre "gde je dijamant?" 174
6.3 Monte Karlo: do površine kruga bez upotrebe broja π 177
6.4 Uniformna raspodela 177

7.1 Asimptotske notacije 189
7.2 Redovi složenosti 190
7.3 Empirijska vremenska složenost za linearno pretraživanje 193
7.4 Binarno pretraživanje 195
7.5 Funkcija složenosti za binarno pretraživanje 196
7.6 Algoritam Counting sort 198
7.7 Algoritam Selection sort 201
7.8 Složenost Selection sort algoritma 202
7.9 Algoritam Merge sort: stablo rekurzivnih poziva 204
7.10 Algoritam Merge sort: faza spajanja 204
7.11 Složenost algoritma Merge sort 206
7.12 Algoritam Quick sort: postupak podele na dve podliste 208
7.13 Interna organizacija liste 212
7.14 Interna organizacija rečnika 213

8.1 Klijent-server model 223

9.1 Relativne i apsolutne putanje 237
9.2 Rekurzivno posećivanje datoteka u stablu direktorijuma 240
9.3 Rečnik preslikavanja latiničnih u ćirilična slova 251
9.4 Postupak preslovljavanja 251
9.5 Efekti preslovljavanja 253
9.6 Izgled datoteke u CSV formatu 254
9.7 Formiranje CSV datoteke 257
9.8 Množenje matrica 258
9.9 Inicijalizacija matrice C=AB 260
9.10 Množenje matrica: izgled ulaznih i izlazne CSV datoteke 261

10.1 Klasa i objekti 265
10.2 Enkapsulacija 266
10.3 Klasa `Tačka` 267
10.4 Postupak kreiranja novog objekta 269
10.5 Automatsko kopiranje objektne reference u parametar `self` 270
10.6 Kompozicija i agregacija 294
10.7 Klase `Krug` i `Tačka` 294
10.8 Zgrada, stanovi i vlasnici: klasni dijagram 297
10.9 Nasleđivanje 304
10.10 Red i stek 308

11.1 Višedimenzionalne matrice. 319
11.2 Memorijska organizacija liste i matrice u NumPy-u 320
11.3 Osnovno indeksiranje u NumPy-u 327
11.4 Pogledi u NumPy-u 331
11.5 Jednostavni linijski dijagram. 372
11.6 Elementi linijskog dijagrama. 373
11.7 Višestruki linijski prikaz sa legendom. 374
11.8 Grafik realne funkcije jedne promenljive. 376
11.9 Horizontalni i vertikalni pravougaoni dijagram. 378
11.10 Kružni dijagram. 379
11.11 Histogram: raspodela frekvencija i kumulativni dijagram. . . . 381
11.12 Grafik realne funkcije dve promenljive. 382

Tabele

2.1 Osnovni tipovi objekata . . . 27
2.2 Primerena imena za promenljive . . . 31
2.3 Najčešće operacije za brojne i logičke tipove . . . 33
2.4 Operatori poređenja . . . 35

3.1 Najveći broj u nizu - tabela stanja . . . 46
3.2 Generisanje sekvenci celih brojeva . . . 50
3.3 Kvadratna matrica - tabela stanja . . . 52

5.1 Često korišćene metode tipa `str`. . . . 115
5.2 Često korišćene metode tipa `list`. . . . 125

6.1 Modul `random`: generisanje pseudoslučajnih brojeva . . . 164
6.2 Diskretna raspodela verovatnoća . . . 171

7.1 Modul time - funkcije za merenje vremena . . . 183
7.2 Vremenska složenost u radu sa listama . . . 212
7.3 Vremenska složenost u radu sa rečnicima . . . 214

8.1 Predefinisani tipovi grešaka . . . 219

9.1 Česte operacije sa sistemom datoteka . . . 237

9.2 Česte operacije sa putanjama 239

10.1 Preopterećenje operatora i često korišćene magične metode 274

11.1 Najčešće korišćeni skalarni tipovi u NumPy-u. 323
11.2 Često korišćene unarne funkcije u NumPy-u. 339
11.3 Često korišćene binarne funkcije i operacije u NumPy-u. 340
11.4 Često korišćene agregatne funkcije u NumPy-u. 344
11.5 Često korišćeni opisi linijskog prikaza. 373

Indeks problema

G1
- Pogađanje broja, 6
- Trgovački putnik, 9

G2
- Težina tela, 22
- Razmena vrednosti, 31
- Tabela istinitosti, 33
- Maksimalna vrednost, 35

G3
- Par-nepar, 42
- Najveći broj u nizu, 45
- Kvadratna matrica, 50
- N-ti koren, 53
- Kosa crta, 55
- Stepenice, 57
- Delioci broja, 59
- Četiri sabirka, 60
- Kusur, 63

G4
- Podniz nula u binarnom zapisu, 75
- Površine, 86
- Tablica funkcije, 90
- Drugovi Faktorijel i Fibonači, 92
- Broj redova u sali, 94
- Hanojske kule, 95

G5
- Šifrovana poruka, 107
- Palindromi, 112
- Prosti brojevi, 121
- Reči u rečenici, 127
- Elementarna statistika, 131
- Prepisivanje na ispitu, 135
- Prevođenje binarnog u heksadekadni zapis, 141
- Strelci, 143
- Kombinacije bez ponavljanja, 149
- Trouglovi, 154
- Osam dama, 157

G6
- Pogodi lutajući broj, 165
- Generator (jakih) lozinki, 168
- Tri kutije i dijamant, 174

Površina kruga i zaboravljeno pi, 176
G7
Fibonačijevi brojevi - dinamičko programiranje, 183
Plate u Patkovgradu, 197
G8
Deljenje u defanzivi, 220
Deljenje sa izuzecima, 226
G9
Stablo direktorijuma, 240
Kopiranje, 247
Ćirilica, 251
Množenje matrica, 257
G10
Razlomci, 274
Vicomat, 288
Nekretnine, 297
U red i na stek, 309
G11
Blok matrica, 336
Uspeh na studijama, 346
Sopstveni vektori, 357
Uglovi između vektora, 368

Indeks

adresa, 5
agregacija, 293
agregatne funkcije, 344
algoritam, 6
algoritamska složenost, 181
anonimni objekat, 274
apsolutna putanja, 236
apstrahovanje, 2
apstrakcija, 69
apstraktni tip podataka, 263
argument, 72
aritmetičko-logičke operacije, 33
ASCII standard, 106
asemblerski jezik, 11
asimptotska notacija, 188
atribut, 265
ažuriranje, 348

bajt, 4
bazni slučaj, 92
beli karakteri, 115
biblioteka procedura, 14
binarna datoteka, 236
binarna pretraga, 194
binarni zapis, 76
bit, 4
blok, 41
brojač, 48
brojačka petlja, 48

Counting sort, 197
CSV format, 254

datoteka, 235
datotečki objekat, 244
debager, 20
defanzivno programiranje, 220
Dekartov proizvod, 156
deklarativni jezik, 13
dekodiranje informacija, 3
dekompozicija, 55, 69
dekorator, 281
deterministički algoritam, 163

dijagram površi, 380
dijagram toka, 6
dinamičko programiranje, 185
direktorijum, 236
docstring, 72
dodela vrednosti, 24, 29

editor, 19
efikasnost algoritma, 6, 181
eksplicitna konverzija tipa, 36
ekstenzija, 236
emitovanje, 341
enkapsulacija, 266
epoha, 182
Euklidov algoritam, 276

f-string, 115
FIFO, 308
fon Nojmanov model računara, 4
formalni parametar, 72
formatiranje, 115
funkcija, 24
funkcija `__isinstance()__`, 273
funkcija složenosti, 181, 186
funkcija vremenske složenosti, 187
funkcije sa proizvoljnim brojem parametara, 145
funkcionalna dekompozicija, 69
funkcionalno programiranje, 15

generalizacija, 2
generator slučajnih brojeva, 163
globalna promenljiva, 14, 81, 84
globalni imenski prostor, 80
grafičko predstavljanje podataka, 371
grananje, 41
greška pri izvršavanju, 217
gruba sila, 7, 59
grupna dodela vrednosti, 348

heuristika, 9
heš funkcija, 213
heš tabela, 213
histogram, 377, 379

identitet, 26
ime promenljive, 31
imenovani parametar, 80
imenski prostor, 21, 80, 99
imenski prostor klase, 282
imenski prostor objekta, 282
imperativni jezik, 13
implicitna konverzija tipa, 29
indeksiranje, 104
indentacija, 42
informacija, 3
inicijalizator, 268
interaktivno okruženje, 20
interfejs klase, 266
interpreter, 13
iterabilni objekat, 153
iteracija, 45
iterativna pretraga prostora rešenja, 59
iterativni algoritam, 52
izdvajanje podsekvenci, 111
izlazni uređaji, 5
izraz, 24, 29
izuzetak, 225
izvorni kod, 12
izvođenje, 303

javni član klase, 283

kanonizacija, 137
karakter, 106
kategorijski podaci, 371
keširanje, 320
klasa, 14, 26, 263, 264
klasna dekompozicija, 293
klasni atribut, 280
klasni dijagram, 294
klasni model, 265
klijent, 223
ključ, 133, 138
koceptualni model, 2
kodiranje informacija, 3
kodiranje karaktera, 106
kolekcija objekata, 28, 103

kombinacije, 148
kombinacije bez ponavljanja, 149
kombinacije sa ponavljanjem, 149
kombinatorna struktura, 148
kombinovano indeksiranje, 356
komentar, 23
kompatibilnost unazad, 16
kompozicija, 293
konekcija, 233
konkatenacija, 109
konstruktor, 36, 268
kontrola toka, 40
kontrolni karakter, 115
kopiranje objektnih referenci, 83, 129
korektnost algoritma, 6
korisnički definisana funkcija, 71
kreiranje matrice, 322
kritična sekcija, 226
kružni dijagram, 377
kvazilinearna složenost, 207

leksikografski poredak, 108
LIFO, 308
linearna pretraga, 191
linearna složenost, 8
linijski dijagram, 371
lista, 117
logaritamska složenost, 8
logička greška, 217
logičke operacije, 350
logički izraz, 41
lokalna promenljiva, 14, 81
lokalni imenski prostor, 80
lokalno optimalno rešenje, 62

magična metoda, 271
Matplotlib, 371
matrica, 124, 257, 317
matrične funkcije, 338
matrične operacije, 338
matrični pogledi, 329
matrično emitovanje, 342
mašinski jezik, 11

merenje vremena, 182
Merge sort, 203
metapodaci, 282
metoda, 114, 266
metoda `__eq()__`, 272
metoda `__init()__`, 268
metoda `__str()__`, 272
metoda instance, 266
metoda iterativne konstrukcije, 54
metoda uzastopnih aproksimacija, 54
model, 2
model klijent-server, 223
modul, 69, 86
modul `builtins`, 80
modul `csv`, 255
modul `itertools`, 149
modul `math`, 88
modul `np.random`, 325
modul `random`, 165
modul `time`, 182
Monte Karlo, 173
Monte Karlo, pogodak-promašaj, 177
mutator, 126

napredno indeksirnje, 353
naredba `break`, 47
naredba `class`, 268
naredba `def`, 72
naredba `finally`, 232
naredba `for`, 49
naredba `global`, 85
naredba `if-elif-else`, 43
naredba `if-else`, 43
naredba `if`, 41
naredba `import`, 87
naredba `raise`, 229
naredba `return`, 74
naredba `try-except`, 227
naredba `while`, 45
naredba `with`, 250
nasleđivanje, 303
nedeterministički algoritam, 163
niz, 124, 317

notacija sa tačkom, 87, 266
numerički podaci, 371
NumPy, 317

objekat, 14, 26, 264
objekat `None`, 75
objekat greške, 218
objektna referenca, 29
objektni kod, 13
objektno orijentisani jezik, 13
oblast problema, 22
obrada grešaka, 217
obrada izuzetaka, 225
odozdo na gore, 298
opcioni parametar, 80
operacije poređenja, 34
operativni sistem, 11
operator izostavljanja, 329
operatori raspakivanja, 145
opseg važenja, 80
ordinalni podaci, 371
osnovna klasa, 303
osnovno indeksiranje matrice, 326
osobine objekta, 264
otvaranje datoteke, 244

paket, 69, 89
paket `numpy`, 317
paleta boja, 382
parser, 126
parsiranje, 255
permutacije, 148, 155
petlja, 45
podatak, 3
podeli pa vladaj, 8, 92, 194, 203, 207
podmatrica, 327
podtekst, 109
pogledi, 329
pohlepna pretraga, 10, 62
ponašanje objekta, 264
poređenje matrica, 350
potencijalno rešenje, 59
potklasa, 303
potpis funkcije, 72
povratna vrednost, 74
pravo rešenje, 59
pravougaoni dijagram, 377
predefinisane greške, 219
predefinisani tipovi, 27
prenosivost, 239
preopterećenje operatora, 109, 146, 273
prepisivanje metoda, 307
pretraživanje, 190
prevodilac, 12
privatni član klase, 283
probabilistički algoritam, 170
proceduralni jezik, 13
proces, 2
proces čišćenja memorije, 37
procesor, 4
program, 2, 4, 22
promenljiva, 21, 24, 29
promenljiva instance, 265
promenljivost objekata, 37
prostor rešenja, 59
prostorno-vremenska nagodba, 185
pseudokod, 6
pseudoslučajni brojevi, 164

Quick sort, 207

radna memorija, 4
radni direktorijum, 237
raspakivanje sekvence, 132
raspodela verovatnoća, 171
rasterska grafika, 295
razdvajajuća sekvenca, 127
razdvajanje matrica, 360
razgranati algoritam, 40
razumevanje liste, 124
razvojno okruženje, 15, 19
računar, 4
rearanžiranje elemenata matrice, 364
red, 308
referenca `self`, 270
rekurzija, 91

rekurzivni algoritam, 92
relativna putanja, 237
repetitivni algoritam, 40
rečnik, 133, 138

sakrivanje informacija, 283
sat realnog vremena, 182
sekvenca, 49, 104
sekvencijalni algoritam, 39
Selection sort, 200
semantika, 25
semantička greška, 25, 217
separator putanje, 237
server, 223
SIMD instrukcije, 338
simulacija, 173
sintaksa, 25
sintaksna analiza, 255
sintaksna greška, 25, 217
sistem, 2
sistem datoteka, 236
sistem kodiranja, 236
sistem za pomoć, 20
sistemski sat, 182
sistemsko kodiranje, 245
sistemsko vreme, 182
skalarni tip, 322
skalarni tipovi u NumPy-u, 322
skripta, 87
skup, 133
slučajna promenljiva, 171
sortiranje, 135, 181
spajajuća sekvenca, 127
spajanje matrica, 360
spajanje teksta, 109
spoljašnje sortiranje, 197
sporedni efekat, 85, 129
stablo, 62, 174
stablo direktorijuma, 240
stanje izračunavanja, 223
stanje objekta, 264
stanje sistema, 3
statička metoda, 281
stek, 308
stepenovanje, 31
string, 105
stvarni parametar, 72

tabela imena, 80
tabela istinitosti, 33
tabela stanja, 46
tabulator, 257
tekstualna datoteka, 236
tekstualna sekvenca, 105
tip, 26
tip `bool`, 28
tip `complex`, 28
tip `dict`, 138
tip `float`, 27
tip `int`, 27
tip `list`, 117
tip `NoneType`, 75
tip `numpy.ndarray`, 318
tip `range`, 105
tip `set`, 133
tip `str`, 28, 105
tip `tuple`, 129
torka, 129
trag greške, 218
transponovanje, 329

ulančavanje poziva, 137
ulazni uređaji, 5
UML, 264
Unicode, 106, 236
Unicode code point - UCP, 106
univerzalno vreme, 182
unutrašnje sortiranje, 197
uslovni izraz, 44

varijacije, 148, 155
varijacije bez ponavljanja, 156
varijacije sa ponavljanjem, 156
vektor-kolona, 322
vektor-vrsta, 322
vektorska grafika, 295
vektorske instrukcije, 338

veliko O, 188
veliko Omega, 190
veliko Teta, 190
verovatnoća, 170
višestruka dodela vrednosti, 31
viši programski jezik, 12