SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zoran Kalafatić

SKRIPTNI JEZICI

Materijali za predavanja

Zagreb, 2012

Sadržaj

1 Uvod **1**
1.1 Primjena skriptnih jezika 1
1.2 Ciljevi predmeta 3
1.3 Osobine skriptnih jezika 4
1.4 Klasifikacija programskih jezika 5

2 Ljuska operacijskog sustava **7**
2.1 Ljuska bash 7
2.2 Datoteke i kazala 8
2.2.1 Operacije nad datotekama 8
2.2.2 Uređivanje datoteka 9
2.2.3 Preusmjeravanje 9
2.3 Varijable ljuske 10
2.3.1 Varijable okoline 11
2.4 Koraci obrade naredbenog retka 12
2.4.1 Navodnici i uloga u tumačenju naredbi 12
2.4.2 Širenja u naredbenom retku 13
2.5 Složeniji Unix alati 16
2.5.1 Brojanje riječi, znakova i redaka 16
2.5.2 Pretraživanje tekstnih datoteka 17
2.5.3 Regularni izrazi 18
2.5.4 Sortiranje teksta 19
2.5.5 Pronalaženje ponovljenih redaka 20
2.5.6 Manipulacija tekstom 20
2.5.7 Traženje datoteka 21
2.6 Pisanje i pokretanje skripti 23
2.6.1 Argumenti naredbenog retka 24
2.6.2 Ispitivanje uvjeta 25
2.6.3 Izlazak iz skripte 27
2.6.4 Naredba case 28
2.6.5 Programske petlje 29
2.6.6 Preusmjeravanje u petlji 32
2.6.7 Učitavanje i ispis 32

3 Programski jezik Perl **35**
3.1 Pisanje i pokretanje Perl programa 35
3.2 Skalarni podaci 36
3.2.1 Brojevi 36
3.2.2 Znakovni nizovi 37
3.2.3 Automatska pretvorba između brojeva i nizova 38

3.2.4 Skalarne varijable 38
3.2.5 Operatori i redoslijed primjene 40
3.2.6 Operatori usporedbe 40
3.2.7 Ispitivanje uvjeta 41
3.2.8 Učitavanje podataka 42
3.2.9 Petlja while 42
3.2.10 Nedefinirana vrijednost 42
3.3 Liste i polja 43
3.3.1 Pridruživanje vrijednosti listi 45
3.3.2 Referenciranje polja 46
3.3.3 Interpolacija polja 47
3.3.4 Petlja foreach 47
3.3.5 Izmjena redoslijeda elemenata liste 48
3.3.6 Skalari i liste ovisno o kontekstu 49
3.4 Potprogrami 50
3.4.1 Argumenti 51
3.5 Učitavanje pomoću operatora <> 54
3.6 Ispis na standardni izlaz 55
3.7 Datoteke 57
3.7.1 Otvaranje datoteke 57
3.7.2 Korištenje datoteke 58
3.8 Asocijativna polja 59
3.8.1 Pristup elementima asocijativnog polja 59
3.8.2 Asocijativno polje kao cjelina 60
3.8.3 Operacije s asocijativnim poljima 60
3.9 Regularni izrazi 62
3.9.1 Operacije s regularnim izrazima 63
3.9.2 Operator povezivanja 65
3.9.3 Interpolacija u regularnim izrazima 65
3.9.4 Varijable podudaranja 65
3.9.5 Povezivanje unazad 66
3.10 Perl programi u naredbenom retku 67

4 Python 69
4.1 Izvođenje programa u Pythonu 69
4.2 Moduli 70
4.3 Ugrađeni tipovi podataka 73
4.3.1 Brojevi 73
4.3.2 Dinamičko upravljanje tipovima 78
4.3.3 Znakovni nizovi 80
4.3.4 Nizovi binarnih znakova 90
4.3.5 Liste 91
4.3.6 Rječnici 97
4.3.7 n-torke 102
4.3.8 Skupovi 103
4.3.9 Datoteke 105
4.4 Usporedbe objekata 110
4.5 Naredbe u Pythonu 110
4.6 Funkcije 117
4.6.1 Definiranje i pozivanje funkcija 117

4.6.2 Prenošenje argumenata u funkcije 121
4.6.3 Bezimene funkcije 124
4.7 Još malo o modulima 125
4.7.1 Moduli i prostori imena 125
4.7.2 Skrivanje podataka u modulu 127
4.8 Razredi 128
4.8.1 Stvaranje razreda 128
4.8.2 Stvaranje primjerka 129
4.8.3 Metode 129
4.8.4 Nasljeđivanje 130

1. Uvod

Ovi materijali su priređeni za kolegij Skriptni jezici, koji se nudi kao izborni predmet na nekoliko modula preddiplomskog studija na Fakultetu elektrotehnike i računarstva Sveučilišta u Zagrebu. Preduvjet mu je predmet Programiranje i programsko inženjerstvo, pa ste se zasigurno već susreli s nekom programskim jezicima, kao i s postupcima oblikovanja i prevođenja programa.

Tema ovog kolegija su jezici koji ne zahtijevaju eksplicitan korak prevođenja programa, što ubrzava ciklus razvoja programa. Radi se o *interpretiranim* jezicima, a kako se takvi jezici najčešće koriste za pisanje relativno kratkih programa visoke razine (*skripti*), nazivaju se i skriptnim jezicima. Pisanje programa visoke razine znači da se programer udaljava od detalja izvedbe programa na ciljnoj platformi, odnosno koristi gotove komponente koje se brinu o tim detaljima. Naravno, to se u nekoj mjeri plaća performansom, no u većini primjena to nije presudno. Stoga su skriptni jezici zauzeli čvrsto mjesto u paleti programskih jezika.

Danas se koristi mnoštvo različitih jezika ove kategorije (JavaScript, VB Script, Ruby, Lua,...), no mi smo se za potrebe ovog kolegija odlučili za tri karakteristična predstavnika: ljuska operacijskog sustava *bash*, *Perl* i *Python*.

1.1 Primjena skriptnih jezika

Da bismo ilustrirali neke primjene skriptnih jezika, poslužit ćemo se s nekoliko jednostavnih primjera. Oni pokazuju kako se neke tipične operacije s kojima se zacijelo susrećete u radu na računalo mogu učinkovitije obaviti pisanjem jednostavnih skripti.

Primjer 1

Kao prvi primjer, zamislimo da želimo preimenovati poveću skupinu datoteka koje imaju karakterističan oblik imena, primjerice fotografije s ljetovanja. Recimo da je u imenu datoteke s fotografijom zapisan datum snimanja, te redni broj fotografije, u obliku:
`DSC_YYYY-MM-DD_NNNN.JPG`.
Pretpostavimo da želimo preimenovati fotografije iz srpnja i kolovoza 2011. godine, uz eliminiranje datuma, u oblik: `Ljeto_YYYY_NNNN.JPG`.

Prvo rješenje koje nam vjerojatno pada na pamet je korištenjem grafičkog korisničkog sustava (GUI) na koji smo navikli. Međutim, to bi značilo označavanje svake pojedine datoteke (desni klik miša) i aktiviranje akcije `Rename`. Čini se da je to rješenje vrlo neprikladno jer zahtijeva ogromnu količinu interakcije uz stalno ponavljanje jednostavnih mehaničkih operacija.

Alternativa bi bila korištenje nekog programa za obradu fotografija koji nudi mogućnost grupnog preimenovanja datoteka (*engl. batch rename*). Međutim, ako i nemamo na raspolaganju takav program, možemo se poslužiti ljuskom operacijskog sustava koji koristimo. Ona omogućuje unošenje naredbi u obliku teksta u naredbenom retku, te zapisivanje takvih naredbi u datoteku – pisanje skripti. Primjerice, ako koristimo operacijski sustav Unix ili Linux, za preimenovanje datoteka primijenit ćemo naredbu `mv`. U najjednostavnijem obliku, naredbi se zadaju izvorno i željeno ime datoteke:

```
$ mv DSC_2011-07-22_1234.JPG Ljeto_2011_1234.JPG
```

Sada bismo to isto mogli napraviti za svaku od datoteka, uz odgovarajuću prilagodbu imena, no očito bi to bio zamoran posao. No, možemo se poslužiti programskim mogućnostima ljuske, tako da napišemo petlju u kojoj će se pozivati naredba `mv` za svaku od datoteka koje zadovoljavaju naš kriterij.

```
#!/bin/bash

# preimenovati DSC_YYYY-MM-DD_NNNN.JPG
# u Ljeto_YYYY_NNNN.jpg

for ime1 in DSC_2011-07-*.JPG DSC_2011-08-*.JPG; do
  pom=${ime1/-??-??/}   # brisem nepotrebni dio
  ime2=${pom/DSC/Ljeto} # zamjenjujem prefiks
  mv $ime1 $ime2
done
```

Za sada ne moramo ulaziti u detalje skripte, njih ćemo upoznati kasnije. No, možemo uočiti da se koristi petlja `for` koja prolazi kroz listu datoteka u tekućem kazalu čije ime sugerira da se radi o fotografijama snimljenima tijekom srpnja i kolovoza 2011. Nakon toga koriste se naredbe koje isijecaju informacije koje želimo zadržati, te kombiniraju novo ime. Konačno, obavlja se preimenovanje svake od datoteka. Na ovaj način je vrlo lako obaviti i relativno složeno preimenovanje datoteka, neovisno o tome koliko ih je.

Primjer 2

Pretpostavimo da želimo prirediti web stranicu s popisom diplomskih radova, u nekom zadanom formatu. Pritom su podaci o tim radovima zapisani u *excel* tablici. Za potrebe ovog primjera to mogu jednostavno biti prezime i ime studenta, naslov rada, te godina obrane.

Prvo rješenje koje nam pada na pamet bi bilo kopirati podatke u program za obradu teksta (npr. *MS Word*), a zatim uređivati polje po polje teksta. Na kraju tekst možemo pohraniti u datoteku u *html* obliku. Ako lista diplomskih radova nije dugačka, taj posao nije prezahtjevan. No, postavlja se pitanje hoćemo li taj posao morati ponoviti ako se odlučimo za izmjenu formata pojedinih polja.

Drugo rješenje je slično, s tim da podatke umjesto u programu za obradu teksta uređujemo u jednostavnom editoru, izravno u *html* obliku.

No, rješenje koje bismo zasigurno odabrali ako poznajemo neki od skriptnih jezika je mogućnost automatiziranog generiranja dokumenata. Jezik koji je za to naročito pogodan (s takvom je namjenom izvorno i oblikovan) je *Perl*. U toj varijanti bismo podatke iz tablice eksportirali u nekom od tekstnih formata, od kojih je najuobičajeniji *CSV* (*Comma-Separated Values*). Na takvu bismo datoteku primijenili skriptu u Perlu, koja će učitavati redak po redak datoteke, izrezivati polja s podacima i generirati odgovarajuće zapise u *html* formatu.

```
#!/usr/bin/perl -w

# diplomski u html ...
#<UL> <LI>Ime Prezime, <I>Naslov</I>, godina </LI> ... </UL>

print "<UL> \n";
while (<>){
  if(/(.*), (.*), (.*), ([0-9]{4})/){
    print "    <LI> $2 $1, <I>$3</I>, $4 </LI> \n";
  }
}

print "</UL> \n";
```

Ni ovdje se nećemo zamarati detaljima iz skripte, Perl ćemo upoznati kad za to dođe vrijeme, pa će ovi kriptični konstrukti postati jasnima.

Primjer 3

Kao još jednu ilustraciju problema koje je prikladno rješavati skriptnim jezicima pretpostavimo da bismo željeli pristupati nekim *web* stranicama i prikupljati podatke. Primjerice, mogli bismo poželjeti prikupiti linkove na slike koje se pojavljuju na analiziranoj *web* stranici, a koji su obično oblika: `<img src="url" ... >`.

Ne ulazeći u razmatranje kako bismo to mogli napraviti ručno, odmah ćemo predložiti skriptu napisanu u jeziku *Python*. Vidjet ćemo da nam Python nudi jednostavne mehanizme za pristup i analizu *web* stranica.

```
# Python 3.x
import re, sys
from urllib.request import urlopen

# adresu stranice prenosimo kao argument skripte
adr = sys.argv[1].rstrip()
page = urlopen(adr).read().decode('utf8')

# nađimo slike
slike=re.findall('<img.+src="([^"]*)"[^>]*>',page,re.IGNORECASE)
for s in slike:
  print(s)

print("broj slika: "+str(len(slike)))
```

Nakon što smo vidjeli ovih nekoliko primjera, već možemo naslutiti veliku primjenjivost skriptnih jezika. Stoga ne čudi da se oni danas intenzivno koriste razvoju programske podrške, prije svega u razvoju web aplikacija. Druga važna primjena je u području sistemske i mrežne administracije, te automatizacije repetitivnih poslova. Također, često se koristi i mogućnost ugrađivanja interpretera nekog od skriptnih jezika u vlastite aplikacije kako bi se povećala njihova fleksibilnost, odnosno programirljivost od strane krajnjih korisnika.

1.2 Ciljevi predmeta

U okviru predmeta *Skriptni jezici* odlučili smo se obraditi tri cjeline. U prvoj se bavimo ljuskom operacijskog sustava, koja nam može povećati produktivnost automatiziranjem poslova koji se pojavljuju u svakodnevnoj praksi. Ljuska nam je praktički uvijek na raspolaganju, bez obzira na to jesu li na sustavu instalirani i neki drugi jezici i alati. Kao konkretan primjer ljuske odabrali smo *bash*, ljusku koja potječe s Unixa, no danas je nalazimo ili je možemo instalirati i na većini drugih operacijskih sustava. Na inačicama Linuxa i Mac OS X dolazi u paketu, dok je na Windowsima možemo lako instalirati, primjerice u okviru okoline *Cygwin*.

U okviru predmeta upoznat ćemo se s osnovnim konceptima pisanja skripti, s osnovnim Unix naredbama, programskim konstruktima ljuske *bash*, te biste na kraju trebali biti u stanju samostalno napisati jednostavnije skripte. Važno je i moći razumjeti i prilagoditi tuđe skripte, što se često pojavljuje u praksi prilikom instalacije i prilagodbe programske podrške.

U okviru cjeline koja se bavi ljuskom operacijskog sustava upoznat ćemo se i s nekim alatima za automatiziranu obradu podataka. To je jedan zadatak koji se često pojavljuje u inženjerskoj praksi (primjerice, analiza telefonskog prometa od strane operatera), a s njime se često susreću i znanstvenici koji trebaju obrađivati rezultate svojih eksperimenata. Naravno,

u tome im pomažu programeri koji moraju oblikovati učinkovita rješenja. Ti su podaci obično u obliku teksta, a važan element obrade je pronalaženje zadanih uzoraka teksta, te njihova analiza ili pretvaranje u oblik prikladan za daljnju obradu. Stoga ćemo se upoznati s osnovnim konceptima i alatima za obradu teksta u skriptnim jezicima, primjerice s primjenom *regularnih izraza*.

Upravo u području automatizirane obrade teksta, vrlo raširen alat je programski jezik *Perl* (*Practical Extraction and Report Language*), pa ćemo s upoznati i s njegovim osnovama. Njegova raširenost se posebno povećala s razvojem Interneta, jer je vrlo prikladan za generiranje *HTML* dokumenata i oblikovanje dinamičkih web stranica. Perl je jezik specifične sintakse, koji nudi brojne pokrate kojima se česti programski konstrukti mogu sažeto zapisati. No, to istovremeno programe čini manje čitljivima, posebno za programere koji mu nisu vični. I ovdje nam je cilj osposobiti se za pisanje jednostavnijih skripti, te moći pročitati (jednostavniji) tuđi kôd. Ograničavamo se na "jednostavnije" programe, jer Perl omogućuje pisanje sasvim funkcionalnih programa koje ni sam autor nakon nekoliko dana neće razumjeti. Naravno, nastojat ćemo ne pisati takve programe, već se držati pravila preglednog i čitljivog programiranja.

Konačno, treća cjelina obrađuje *Python*, koji je jedan od najpopularnijih programskih jezika današnjice. Radi se o jeziku koji omogućava brzo i izražajno pisanje kratkih skripti, čime lako može poslužiti i kao zamjena za ljusku operacijskog sustava ili Perl, no istovremeno se radi o snažnom objektno orijentiranom jeziku koji omogućuje oblikovanje najsloženijih programskih produkata.

Python je zanimljiv i zbog svoje jasne i izražajne sintakse, koja je različita od C-a (odnosno njemu sličnih jezika C++, Java, C#, Perl itd.). Stoga je za iskusnog programera u nekom od takvih jezika ispočetka malo neobičan, no izuzetno se lako i brzo uči jer je vrlo dosljedan. Uz to, specifičnost sintakse je i to što tjera programera na urednost i pregledno pisanje koda, čime su programi pisani u Pythonu inherentno čitljivi. Zbog tih svojih svojstava, Python se sve češće preporučuje kao prvi programski jezik. Također, u mnogim kolegijima i udžbenicima sve se češće, na mjestima gdje se prije obično koristio pseudokôd, pojavljuju programski primjeri pisani u Pythonu.

1.3 Osobine skriptnih jezika

Skripta je program vrlo visoke razine, najčešće vrlo kratak, napisan u skriptnom jeziku. Visoka razina programiranja podrazumijeva da se u programer ne bavi detaljima bliskim arhitekturi računala na kojoj se program izvodi. Važna osobina skriptnih jezika je da se ne prevode prije izvođenja (barem ne eksplicitno), odnosno da se *interpretiraju*. Neke primjere skriptnih jezika smo već spomenuli, a tu ubrajamo ljuske operacijskih sustava, Tcl, Perl, Python, Ruby, Lua, Scheme, Rexx, JavaScript, VBScript i mnoge druge.

Skriptni jezici tipično se koriste za povezivanje više samostalnih programa u cjelinu "lijepljenjem" (*engl. glue language*). Obično imaju ugrađenu podršku za intenzivnu obradu teksta, a najčešće se koriste za pisanje programa usmjerenih specifičnim potrebama korisnika. Tipično se ne radi o složenim programima, što povlači relativno jednostavan razvoj.

Često se skriptni jezici uspoređuju s tzv. *sustavskim* jezicima, u koje se ubrajaju C/C++, Java i slični, a koji omogućavaju upravljanje resursima bliskima arhitekturi računala na kojemu se program izvodi. Prednost skriptnih jezika se prije svega odnosi na tipičnu kratkoću i sažetost programa, zbog čega je razvoj brži i s manje mogućnosti pogreške. S obzirom da je izostavljen eksplicitan korak prevođenja i povezivanja programa, ciklus provjere programa je kraći (naprosto se izmjene sporne naredbe i ponovno pokrene program). U skriptnim jezicima tipično nema deklaracije varijabli, odnosno one se uvode u program kada nam zatrebaju. To je osobina koja olakšava programiranje, no na taj način su izbjegnute provjere koje tipično obavlja program za prevođenje, pa je odgovarajuće provjere potrebno provoditi tijekom izvođenja

programa.

Naravno, ove prednosti ne znače da nam više ne trebaju sustavski jezici. Svaka od ovih kategorija jezika ima svoje područje primjene. Skriptni jezici ciljaju na čim bržu i jednostavniju izgradnju programa, pri čemu se koriste bogate biblioteke gotovih *komponenti* koje se povezuju u cjelinu (*komponentno* programiranje). S druge strane, sustavski jezici nastoje osigurati čim veću učinkovitost konačnog kôda. Stoga nude mogućnost upravljanja svim detaljima izvedbe. Oni su prikladni za izradu komponenti koje će se zatim povezati u skriptnom jeziku, pogotovo za računski zahtjevne dijelove funkcionalnosti.

Ovakav pristup programiranju možemo ilustrirati jednim primjerom. Recimo da želimo napisati program koji učitava niz slika neke video snimke i postupcima računalnog vida pronalazi i prati objekte. Budući da su postupci računalnog vida računski intenzivni, a važno je da obrada pojedine slike bude brza (za tipični video pojavljuje se 25 slika u sekundi, što znači da bismo za rad u stvarnom vremenu svaku sliku trebali obraditi u 40 ms). Stoga ćemo komponentu za analizu slike napisati u programskom jeziku C. Pritom program može slike učitavati sa standardnog ulaza, na koji se pak može preusmjeriti sadržaj niza slikovnih datoteka ili pak slike dobivene s digitalizatora slike. Rezultat (slika s označenim objektom) može se zapisati na standardni izlaz, koji se po potrebi može preusmjeriti u datoteku ili u program za prikaz slike. Kako bismo olakšali pokretanje programa, možemo napisati kratku skriptu:

```
cat slika*.ras | prati_objekt | display
```

koju možemo pozivati s:

```
$ ./pokreni_pracenje
```

Za pristupanje digitalizatoru slike možemo također napisati program (u C-u). Program će iz digitalizatora slike preuzimati sliku po sliku, te će ih u odgovarajućem formatu prosljeđivati na standardni izlaz. Nakon toga je, korištenjem ljuske operacijskog sustava, jednostavno ulančati program za digitalizaciju s ulazom programa za praćenje objekata:

```
grab | prati_objekt | display
```

U ovom pristupu, možemo napisati i dodatne komponente za obradu slike, kao programe u C-u. Primjerice, možemo napisati komponentu za glađenje slike Gaussovim filtrom zadanog parametra σ, koja sa standardnog ulaza čita sliku, a glađenu sliku zapisuje na standardni izlaz:

```
grab | gladixy -s 1.5 | prati_objekt | display
```

Ako se pojavi potreba za demonstracijom programa, lako je pripremiti skriptu koja učitava nekoliko karakterističnih sekvenci iz datoteka, i pokreće program za praćenje s odgovarajućim parametrima, te ispisuje na ekranu odgovarajuće poruke. Također, možemo dodati i grafičko korisničko sučelje, koje možemo jednostavno oblikovati u Pythonu.

1.4 Klasifikacija programskih jezika

Do sada smo već spomenuli podjelu programskih jezika na skriptne i sustavske jezike. Također smo spomenuli prevođene i interpretirane jezike. Razredba se može temeljiti na različitim kriterijima, pa je se često u obzir uzima upravljanje tipovima podataka. Razlikujemo jezike sa statičkim (*engl. statically typed*) i dinamičkim (*engl. dynamically typed*) dodjeljivanjem tipova.

Kod dinamički tipiziranih jezika, kojima tipično pripadaju skriptni jezici, dodjeljivanje tipa varijablama zbiva se tijekom izvršavanja programa. Stoga varijabla može u različitim dijelovima programa sadržavati objekte različitih tipova. Primjerice u sljedećem fragmentu programa u Pythonu, varijabla c u jednom trenutku pokazuje na cijeli broj, dok joj je u sljedećoj naredbi dodijeljena lista.

```
c = 1         # c je integer
c = [1,2,3]   # c je lista
```

U statički tipiziranim jezicima to nije moguće. Tip varijable određuje se pri prevođenju programa i samim time ne može se mijenjati. Stoga će sljedeći programski odsječak u C-u izazvati pogrešku pri prevođenju.

```
    double c; c = 5.2;    /* c prima samo double */
    c = "a string...";    /* compiler error */
```

Sljedeća podjela je na slabo (*engl. weakly typed*) i strogo (*engl. strongly typed*) tipizirane jezike. Ta razlika se odnosi na mogućnost miješanja tipova u izrazima i njihovu implicitnu pretvorbu, a manifestira se i kod različitih skriptnih jezika. Primjerice, Perl je slabo tipiziran, pa se miješanje tipova dozvoljava:

```
$b = '1.2'
$c = 5*$b;  # implicitna pretvorba tipova:
            # '1.2' -> 1.2
```

U gornjem primjeru, znakovni niz `'1.2'` pojavljuje se na mjestu na kojem se očekuje broj. Perl obavlja odgovarajuću pretvorbu i izračunava izraz. Nasuprot tome, Python je strogo tipiziran i ne dozvoljava ovakve zamjene. Sličan programski fragment će u Pythonu izazvati drugačiji rezultat:

```
b = '1.2'
c = 5*b    # nema implicitne pretvorbe tipova
           # rezultat je ponavljanje niza
           # '1.21.21.21.21.2'
```

Zapravo, u ovom primjeru se radi o polimorfizmu operatora množenja, koji kod Pythona u slučaju da su oba operanda brojevi doista obavlja množenje, dok za kombinaciju znakovnog niza i broja obavlja ponavljanje znakovnog niza. Stoga je rezultat gornjeg izraza znakovni niz `'1.21.21.21.21.2'`.

Ako pak pokušamo u Pythonu izračunati izraz: `'5'*'1.2'`, dogodit će se iznimka. U Perlu će rezultat evaluacije tog istog izraza biti `6`.

Ponekad se jezici dijele na jezike visoke i jezike niske razine. Ta se podjela odnosi na udaljenost programera od implementacijskih detalja vezanih za arhitekturu računala na kojem se program izvodi, odnosno na razinu apstrakcije tih detalja. Tako programer u C-u može izravno pristupati memorijskim lokacijama, ali zato mora i voditi brigu o upravljanju memorijskim prostorom kako bi zauzete resurse uredno vratio sustavu. S druge strane, programer u Pythonu nema na raspolaganju takve mehanizme, ali im se zato ne mora ni opterećivati – o tome brine sam Python. Naravno da automatska briga o resursima predstavlja i određeno računsko opterećenje sustava, pa su programi pisani u jeziku visoke razine nešto sporiji nego u jeziku niske razine. K tome, ako je programer u jeziku niske razine vrlo vješt, može ugađanjem detalja izvedbe postići znatno ubrzanje programa za neki konkretan problem. Naravno, takav je problem značajno teže napisati, a u velikom broju primjena brzina izvođenja nije kritična, već je puno važnija brzina implementacije (*engl. time-to-market*).

2. Ljuska operacijskog sustava

Ljuska operacijskog sustava predstavlja fleksibilno tekstualno sučelje prema operacijskom sustavu. Danas operacijski sustavi dolaze s dopadljivim grafičkim korisničkim sučeljima (*GUI*), no ranije je osnovni način pokretanja operacija bio upravo kroz ljusku operacijskog sustava. To vrijedi ne samo za Unix, već i za ostale ranije operacijske sustave kao što su bili MS-DOS, CPM i VMS. Primjeri operacija koje su se pokretale kroz ljusku su pristupanje datotečnom sustavu (kreiranje mapa, pregledavanje sadržaja mapa, kopiranje datoteka), pokretanje korisničkih programa i raznih ugrađenih alata. Bez obzira na raširenost grafičkih korisničkih sučelja, i danas svi operacijski sustavi nude neku vrstu ljuske. Korištenjem naredbenog retka (*engl. command line interpreter*), iskusan korisnik može traženu operaciju pokrenuti brže nego traženjem odgovarajuće naredbe ili opcije u menijima. U slučaju velikog broja naredbi, te bogatog skupa opcija, dobitak u učinkovitosti može biti i značajan. Uz to, niz naredbi može se zapisati u tekstualnu datoteku iz koje se zatim pokreću (*engl. batch obrada*).

U osnovi, ljuska je program koji posreduje između korisnika i operacijskog sustava. Ona nije dio jezgre operacijskog sustava već korisnički program koji koristi pozive prema jezgri OS-a. Ljuska naprosto prihvaća naredbe koje korisnik unosi u naredbenom retku i prevodi ih u pozive jezgre OS-a. To bi značilo da bi ”svatko” mogao napisati svoju ljusku. Međutim, radi kompatibilnosti s drugim korisnicima i drugim sustavima, jasno je da je puno bolje koristiti neku standardnu ljusku.

Jedna od najvažnijih značajki Unix-a je velika zbirka programa – standardno se isporučuje više od 200 osnovnih naredbi (alata). Međutim, još je i važnija lakoća kojom se te naredbe mogu kombinirati za obavljanje sofisticiranih zadaća, u čemu je ljuska presudna. Tipična ljuska podržava programske konstrukte koji omogućavaju ispitivanje uvjeta, formiranje programskih petlji, te korištenje varijabli za pohranu vrijednosti.

Standardne ljuske koje se isporučuju s Unix/Linux sustavima izvedene su iz ljuske *sh* koju je 1978. razvio Stephen Bourne u AT&T Bell Labs. Najpoznatiji sljednici su ljuske *ksh* i *bash*. Druga grana standardnih Unix ljuski je izvedena iz ljuske *csh*, koju je 1979. predstavio Bill Joy sa sveučilišta Berkeley, a sintaksno je slična programskom jeziku C. Inženjerska udruga IEEE je razvila standard temeljen na Bournovoj ljusci *sh* i ostalim novijim izvedbama, *Shell and Utilities volume of IEEE Std 1003.1-2001*, poznatiji kao *POSIX* standard.

2.1 Ljuska bash

U okviru predmeta ćemo se baviti ljuskom *bash* (*Bourne Again Shell*). To je najraširenija ljuska, koja se standardno isporučuje s distribucijama Linuxa, kao i s Mac OS X. Možemo je koristiti i u okviru okoline Cygwin na MS Windowsima, koja je zapravo implementacija skupa Unix alata.

Kako bismo započeli s radom u ljusci, potrebno ju je najprije pokrenuti. Ako koristimo Linux ili Mac OS X, potrebno je pokrenuti program `Terminal`, a ako koristimo MS Windows, potrebno je instalirati okolinu Cygwin i pokrenuti je. U svakom slučaju, otvara se prozor s naredbenim retkom u koji možemo upisivati naredbe. Ukoliko u naredbenom retku nije pokrenuta ljuska *bash*, možemo je pokrenuti kao program (oznaka ”`$`” je odzivni znak (*engl.*

prompt): `$ bash` Kada poželimo napustiti ljusku, jednostavno upišemo naredbu: `$ exit`. Ljuska *bash* nudi mogućnost uređivanja naredbe, ponavljanja prethodno unesenih naredi (korištenjem strelica prema gore odnosno prema dolje prolazimo kroz listu prethodnih naredbi), automatskog nadopunjavanja (tipka `Tab`), te pretraživanja liste prethodnih naredbi (kombinacija tipki `ctrl-R`.

Na raspolaganju nam je i sustav pomoći. Dodatne informacije o pojedinoj Unix naredbi mogu se dobiti navođenjem parametra `--help`. Primjerice, informacije o naredbi `ls` dobit ćemo s: `$ ls --help`. Primijetit ćemo da je tekst dulji od jednog ekrana, pa će nam početak pobjeći. Listanje stranicu po stranicu možemo postići kombinacijom s naredbom `less`:
`$ ls --help | less`

Još jedan izvor informacija o Unix naredbama su *man* stranice (*engl. manpages*). Odgovarajuću stranicu dobit ćemo pokretanjem naredbe `man` kojoj se kao argument navodi naredba koja nas zanima:
`$ man less`
Kako bismo se lakše snašli unutar teksta koji je često vrlo opsežan, na raspolaganju nam stoji i mogućnost pretraživanja. To se postiže upisivanjem znaka "/" nakon kojega se upisuje tekst koji želimo pronaći. Taj tekst može biti i regularni izraz (o regularnim izrazima bit će riječi kasnije). Sljedeću pojavu traženog teksta možemo dobiti upisivanjem slova "n", a prethodnu slovom "N". Izlaz iz pregleda stranice dokumentacije postiže se upisom slova "q".

Osim navedenih, postoje i još neki alati za dobivanje dodatnih informacija kao što su `info`, `apropos` i `whatis`, no čitatelju ostavljamo da istraži na koji se način koriste. Uz to, danas je do korisnih informacija o pojedinim naredbama i načinu njihovog korištenja lako doći korištenjem web tražilica.

2.2 Datoteke i kazala

Unix koristi *hijerarhijski* datotečni sustav. Nakon pristupanja sustavu, korisnik se nalazi u svom osobnom kazalu datoteka (*engl. home directory*).Unutar svog kazala korisnik može stvarati nove datoteke, kao i oblikovati nova kazala. Rezultat je stablasta struktura, s korijenom (*engl. root directory*), čija je oznaka "/".

2.2.1 Operacije nad datotekama

Stvaranje novog kazala postiže se naredbom `mkdir`: `$ mkdir naziv_kazala`

Sadržaj tekućeg kazala može se prikazati naredbom `ls`, pri čemu se opcijama može definirati način prikaza liste datoteka: `$ ls`, `$ ls -l`, `$ ls -al`

Naredba `pwd` omogućuje prikaz oznake tekućeg kazala:

```
$ pwd
/home/korisnik/SkriptniJezici/ciklus1
```

Premještanje u drugo kazalo, odnosno promjena tekućeg kazala postiže se naredbom `cd`:
`$ cd naziv_kazala`
Premještanje u kazalo prve više razine hijerarhije postiže se korištenjem oznake "..":

```
$ cd ..
$ pwd
/home/korisnik/SkriptniJezici
```

Slična oznaka "." označava trenutno kazalo, dok se za premještanje u osobno kazalo korisnika koristi oznaka "~":

```
$ cd ~/SkriptniJezici/ciklus1
$ pwd
/home/korisnik/SkriptniJezici/ciklus1
```

Ponekad je potrebno obrisati neku datoteku, a to se postiže naredbom `rm`: `$ rm ime_datoteke`
Za brisanje kazala koristi se naredba `rm`, pri čemu se ne može obrisati kazalo koje nije prazno:
`$ rmdir ime_kazala`
Ako bismo htjeli obrisati sadržaj kazala i njihovih podkazala (rekurzivno) možemo primijeniti naredbu `rm` s opcijom `-r`. No, pritom treba biti vrlo oprezan, jer bismo nepažnjom mogli obrisati i datoteke koje ne želimo:
`$ rm -r ime_kazala`

Kopiranje datoteke postiže se naredbom `cp`, uz navođenje imena početne i odredišne datoteke: `$ cp datoteka1 datoteka2`, a preimenovanje (premještanje) datoteke naredbom `mv`:

```
$ mv ime_prije ime_poslije
$ mv ime_prije ~/SkriptniJezici/ciklus1/ime_poslije
```

2.2.2 Uređivanje datoteka

Kada budemo pisali skripte ili pripremali datoteke s ulaznim podacima, bit će nam potreban uređivač teksta (*engl. text editor*). Možemo odabrati neki od popularnih editora, već prema raspoloživosti na sustavu koji se koristi: `vi`, `emacs`, `joe`, `nedit`, `nano` i slično.

Alternativni način, prikladan jedino za vrlo kratke i jednostavne skripte ili probne datoteke je korištenjem naredbe `cat`. Tu je mogućnost popravljanja unosa ograničena samo na jedan redak, pa je na taj način vrlo nezgodno pisati složenije tekstove. Postupak započinjemo pokretanjem naredbe `cat > ime_datoteke`. Potom upisujemo redak po redak teksta, a završetak unosa označava se kombinacijom tipki `ctrl-D`. Istu naredbu možemo primijeniti i za prikaz sadržaja datoteke. Ako se pritom navede i opcija `-n`, svaki prikazani redak se numerira:

```
$ cat ime_datoteke
$ cat -n ime_datoteke
```

Za dulje datoteke, prikladno je imati mogućnost listanja sadržaja, pa se za to koristi naredba `more` ili naredba `less` koja omogućuje i listanje teksta prema početku. Obje naredbe se završavaju upisivanjem slova `q`, dok se listanje postiže tipkama `PgUP/PgDN`.

```
$ more ime_datoteke
$ less ime_datoteke
```

2.2.3 Preusmjeravanje

Unix omogućava preusmjeravanje (redirekciju) izlaza programa koji se uobičajeno ispisuje na standardni izlaz `stdout` (ekran) u datoteku. Slično se i standardni ulaz `stdin` umjesto s tipkovnice može preuzimati iz datoteke. Također se može postići i povezivanje dva programa na način da izlaz jednoga bude proslijeđen na ulaz drugoga. Za preusmjeravanje se brine ljuska, a zadaje se korištenjem posebne notacije.

Preusmjeravanje izlaza programa označava se operatorom ">":

```
$ ls > ime_datoteke
```

Svi ispisi koji bi se bez preusmjeravanja pojavili na ekranu, nakon preusmjeravanja završavaju u navedenoj datoteci. Dijagnostički ispisi (poruke o pogreškama) obično se ispisuju na odvojeni izlazni tok `stderr`. Ako nije preusmjeren, i on se kao i `stdout` pojavljuje se na ekranu. Da

bismo dijagnostičke ispise preusmjerili, potrebno je uz operator izlazne redirekcije navesti i oznaku 2. To je deskriptor izlaznog toka `stderr`. Izlazni tok `stdout` ima deskriptor 1, ali njega ne treba pisati.

```
$ program 2> ime_datoteke
```

U nekim slučajevima zgodno je `stderr` preusmjeriti na `stdout`, kako bismo omogućili ulančavanje s nekim dodatnim programom. Primjerice, ako želimo omogućiti listanje dijagnostičkih poruka prevoditelja:

```
$ 2>&1
$ gcc x.c 2>&1 | less
```

Primjena operatora izlazne redirekcije stvara novu datoteku, odnosno sadržaj postojeće datoteke istog imena se briše. Ukoliko želimo zadržati stari sadržaj, te u datoteku nadopisati izlaz programa, koristimo operator nadodavanja (*engl. append*) ”>>”:

```
$ ls >> ime_datoteke
```

Preusmjeravanjem standardnog ulaza (`stdin`, deskriptor 0, ne treba ga pisati) program umjesto s tipkovnice ulazne podatke čita iz navedene datoteke. Operator ulazne redirekcije je ”<”.

```
$ program < ime_datoteke
```

Već smo spomenuli da se izlaz jednog programa može proslijediti na ulaz drugoga. Takvu kombinaciju možemo nazvati ulančavanjem, iako je izvorni naziv cjevovod (*engl. pipeline*). Vidjeli smo već i neke takve primjere:

```
$ ls | less
```

Naredbe ili programi koji su pisani na način da podatke čitaju sa `stdin`, a rezultate zapisuju na `stdout` prikladni su za ulančavanje, a nazivaju se *filterima*. Većina Unix naredbi su upravo tog tipa. Kao primjer možemo napisati kombinaciju naredbi koja će ispisati dio sadržaja zadane (tekstne) datoteke i to od 21. do 35. retka, te rezultat zapisati u novu datoteku:

```
$ tail -n +21 < lista.txt | head -n 15 > kratka.txt
```

2.3 Varijable ljuske

Jedan od osnovnih programskih elemenata su varijable, koje omogućavaju pohranjivanje podataka i rezultata. Stoga i ljuska omogućava korištenje varijabli. Različite ljuske imaju različitu sintaksu dodjeljivanja vrijednosti varijablama i njihovog korištenja. U ljusci *bash*, vrijednost se dodjeljuje na sljedeći način:
`X="abcd"` ili `X=abcd`
U oba slučaja varijabli `X` pridružen je isti znakovni niz. Ovdje treba posebno uočiti da je *bash* osjetljiv na razmake prije i poslije znaka jednakosti. Točnije, ne smije ih biti. Ako se slučajno umetne razmak, *bash* će ime varijable tumačiti kao naredbu ili ime programa i pokušat će ga izvršiti.

```
$ y = 23
bash: y: command not found
```

U prethodnom uspješnom primjeru dodjele vrijednosti varijabli primijetili smo da navodnici oko znakovnog niza nisu nužni. No, u nekim ih situacijama moramo koristiti. Naime, ako se znakovni niz sastoji od više riječi, odnosno ako sadrži praznine, bez navodnika će niz biti razlomljen i pogrešno interpretiran. Stoga je ispravno napisati:

```
$ S="idemo na rucak"
```

dok će izraz bez navodnika izazvati pogrešku.

Kada se varijabla koristi, potrebno ju je predznačiti znakom "$". Primjerice, vrijednost varijable X iz ranijeg primjera možemo ispisati:

```
$ echo $X
abcd
```

Ako zaboravimo prefiks, X će se tumačiti kao znakovni niz:

```
$ echo X
X
```

Imena varijabli sastoje se od slova, brojeva i znaka "_", pri čemu se razlikuju velika i mala slova.

```
$ x=1234
$ echo $X : $x
abcd : 1234
```

2.3.1 Varijable okoline

Posebna vrsta varijabli ljuske su varijable okoline (*engl. environment variables*). Varijable okoline koje postavimo u ljusci nasljeđuju se u programima (procesima) pokrenutima iz ljuske, odnosno pod-ljuskama. Pod-ljuska nastaje pokretanjem nove ljusku kao programa, sjećamo se – ljuska je normalan program. I postavljanje varijabli okoline specifično je za svaku ljusku. U ljusci *bash*, varijabla postaje varijablom okoline tako da joj se postavi odgovarajući atribut naredbom export. To se može obaviti prilikom inicijalizacije varijable, ali i naknadno.

```
export varijabla=vrijednost
```

Popis varijabli okoline možemo dobiti naredbom env ili naredbom export -p. Varijable koje nisu eksportirane, program ili skripta pokrenuta iz ljuske neće vidjeti.

Još jedna specijalizacija su ugrađene (*engl. built-in*) varijable okoline, koje se nazivaju i varijablama sustava (*engl. system variables*). To su varijable koje imaju posebno značenje za operacijski sustav. Primjerice, varijabla $PATH sadrži listu kazala (odvojenih dvotočkom) u kojima se traže izvršne datoteke kada se ljusci zada naredba. Redoslijed kazala u listi određuje redoslijed pretraživanja.

```
$ echo $PATH
/usr/local/bin:/usr/bin:/bin
```

Još neke važnije varijable sustava su $HOME, koja određuje matično kazalo korisnika; $SHELL, koja sadrži ime pokrenute ljuske (uključujući put); $USER, koja sadrži korisničko ime; $TERM, koja sadrži tip terminala, itd. Kao jedan zgodan primjer možemo navesti i podešavanje varijable $PS1, koja pohranjuje oznaku koja se ispisuje na početku naredbenog retka. U tu varijablu se mogu upisati i neke specijalne sekvence koje će izmijeniti izgled ispisa, primjerice boju. Možemo isprobati sljedeći programski odsječak:

```
Green="\[\033[32m\]"
Brown="\[\033[33m\]"
White="\[\033[0m\]"
export PS1="$Green\u@\h:$Brown\w$White\n\$ "
```

Uočavamo da se na početku definiraju varijable koje sadrže specijalne sekvence za upravljanje terminalom tako da se dobije zeleni, smeđi ili bijeli ispis. Zatim se definira sam odzivni niz, na način da se najprije uključi zeleni ispis, zatim se ispiše ime korisnika (oznaka \u), zatim znak @, pa ime računala (oznaka \h) i dvotočka. Nakon toga se ispis prebacuje u smeđu boju i ispisuje se radno kazalo (oznaka \h). Konačno, uključuje se bijeli ispis, prelazi se u novi red i ispisuje oznaka $. Nakon izvršavanja tog programskog odsječka, naredbeni redak bi trebao izgledati otprilike ovako (boja je zanemarena):

```
zkalafatic@MAGELLAN:~/Work/SkriptniJezici
$
```

2.4 Koraci obrade naredbenog retka

Kada se naredba upiše u naredbeni redak, ljuska je analizira obavljajući pritom cijeli niz koraka. Te ćemo korake najprije pobrojati, a zatim postupno objasniti.

1. ljuska čita sadržaj ulaznog retka;
2. ulazna linija razlaže se u niz simbola (*engl. tokens*): riječi i operatora;
3. niz simbola se parsira – prepoznaju se jednostavne i složene naredbe;
4. obavlja se širenje (*engl. expansion*) pojedinih dijelova svake naredbe (riječi) – rezultat je lista imena putanja, te polja koja predstavljaju naredbe i argumente;
5. obavlja se preusmjeravanje, te uklanjaju operatori preusmjeravanja i njihovi operandi;
6. pokreće se funkcija, ugrađena naredba, izvršna datoteka ili skripta, pri čemu se prosljeđuje i lista argumenata;
7. ljuska čeka da naredba završi, te prikuplja njen izlazni status.

2.4.1 Navodnici i uloga u tumačenju naredbi

Prilikom tumačenja naredbenog retka, neki znakovi imaju posebna značenja. To su prije svega razmak, tabulator i prelazak u novi red, ali tu su i znakovi koji se koriste za redirekciju (< i >), te znakovi | & ; ( ) $ ` \ " ' .

Kako bi se specijalni znak (*metaznak*) tumačio doslovce (bez svog specijalnog značenja), potrebno ga je predznačiti (*engl. escape*) znakom "\".

Korištenje navodnika također mijenja ponašanje ljuske pri interpretaciji naredbenog retka. Tako se niz znakova uokviren jednostrukim navodnicima (' ') ne tumači u ljusci, već se samo prosljeđuje kao niz. To možemo ilustrirati primjerom u kojem se pojavljuje ime varijable. Jednostruki navodnici će spriječiti zamjenu imena varijable njenom vrijednošću:

```
$ echo 'Tekuce kazalo je $PWD'
Tekuce kazalo je $PWD
```

Nasuprot tome, dvostruki navodnici (" ")uokviruju niz kao cjelinu, ali pritom se obavlja širenje (*engl. expansion*) argumenata označenih posebnim znakovima ljuske. Primjerice, ime varijable se zamjenjuje njenom vrijednošću:

```
$ echo "Tekuce kazalo je $PWD"
Tekuce kazalo je /home/zkalafatic/Skriptni/ciklus1
```

Još se jedna vrsta navodnika koristi, ali njihovo je značenje bitno drugačije. Radi se o obrnutim jednostrukim navodnicia (*engl. backquotes*) (` `), koji označavaju zamjenu naredbi, o kojoj će kasnije biti malo više riječi.

2.4.2 Širenja u naredbenom retku

Nakon dijeljenja naredbenog retka u riječi, obavlja se niz transformacija koje se nazivaju širenjima (*engl. expansion*). Te transformacije obavljaju se u sljedećim koracima:

- širenje vitičastih zagrada;
- širenje znaka "~";
- širenje parametara i varijabli;
- zamjena naredbi;
- aritmetičko širenje;
- dijeljenje riječi;
- širenje imena datoteka (*engl. pathname expansion*).

Širenje vitičastih zagrada

Širenje vitičastih zagrada (*engl. brace expansion*) je mehanizam kojim se mogu generirati znakovni nizovi, čiji se uzorci mogu prikazati prefiksom, slijedom znakovnih nizova odvojenih zarezima ili izrazom za generiranje sekvence, koji se navode unutar vitičastih zagrada, te sufiksa. Primjerice, izraz `a{d,c,b}e` širi se u `'ade ace abe'`.

Slično:

```
$ echo ab{c,d,e,f}oooo
abcoooo abdoooo abeoooo abfoooo
```

Unutar vitičastih zagrada može se primijeniti i izraz za generiranje sekvenci. Takav izraz je oblika `{x..y}`, gdje su `x` i `y` brojevi (šire se numerički) ili znakovi (šire se leksikografski):

```
$ echo ab{1..4}ij
ab1ij ab2ij ab3ij ab4ij
```

Valja biti svjestan da se širenje vitičastih zagrada obavlja prije svih ostalih ekspanzija, a pritom znakovi za ostale ekspanzije ostaju očuvani.

Još jedan primjer primjene širenja vitičastih zagrada:

```
$ mkdir projekt/{old,new,dist,bugs}
```

Širenje oznake korisničkog kazala

Posebna vrsta širenja odnosi se na oznaku korisničkog kazala, a poznata je i pod imenom "tilda-ekspanzija". Naime, znak koji se koristi kao oznaka korisničkog kazala "~" naziva se tilda. Ovo se širenje obavlja po jednostavnim pravilima.

Ako riječ počinje znakom tilda, svi znakovi do prvog znaka "/" ili kraja riječi smatraju se tilda-prefiksom. Ako tildu slijedi riječ bez navodnika, ona se smatra mogućim korisničkim imenom. Ako korisnik tog imena ne postoji, ekspanzija se ne obavlja:

```
$ echo ~sgros
/home/zemris/sgros

$ echo ~francek     # ne postoji ovo korisničko ime
~francek
```

Ako je tilda-prefiks samo tilda, ona se zamjenjuje varijablom `$HOME`, odnosno matičnim kazalom korisnika koji je pokrenuo ljusku:

```
$ echo ~
/home/zemris/zkalafatic
```

Širenje varijabli i parametara

U nekoliko prethodnih primjera vidjeli smo kako se u ljusci `bash` koriste varijable. Podsjetimo se: ime varijable predznačava se znakom `$`. Zamjena varijable njenom vrijednošću naziva se širenjem varijable.

U nekim slučajevima pojavit će se potreba za umetanjem vrijednosti varijable neposredno prije sljedećeg znaka, što bi moglo dovesti do pogrešne interpretacije imena varijable. Primjerice, ako bismo htjeli konstruirati ime datoteke u kojem se pojavljuje broj pohranjen u varijabli, mogli bismo napisati sljedeći niz naredbi.

```
$ godina=2011
$ ime=ispit_$godina_c.tex
$ echo $ime
ispit_.tex
```

Kontrolni ispis pokazuje da ime datoteke nije ispravno konstruirano. Naime, ljuska je nakon znaka `$` pokupila niz znakova do prvog znaka koji se ne koristi u imenima varijabli, a to je u ovom slučaju točka, i taj niz je upotrijebila kao ime varijable. Kako varijabla `$godina_c` nije definirana, ona se zamjenjuje praznim nizom, pa je rezultat sasvim razumljiv. Kako bismo doskočili ovom problemu, ime varijable možemo okružiti vitičastim zagradama:

```
$ godina=2011
$ ime=ispit_${godina}_c.tex
$ echo $ime
ispit_2011_c.tex
```

U naslovu smo spomenuli i pojam širenja parametara. Parametri su zapravo specijalna vrsta varijabli, čija imena su brojevi koji označavaju njihov redoslijed. Pozicijski parametri pojavljuju se prilikom prijenosa argumenata iz naredbenog retka u skriptu, pa ćemo ih detaljnije obraditi kasnije.

Osim osnovne zamjene (širenja) varijabli, `bash` podržava i neke sofisticiranije inačice. Kao ilustraciju, ovdje ćemo navesti par primjera, a zainteresirani čitatelj može detaljnije proučiti dokumentaciju.

Oblik `${varijabla:-default}` označava da se u slučaju da varijabla nije inicijalizirana upotrijebi navedena pretpostavljena vrijednost.

```
$ echo ${nepostojeca:-podrazumijevana}
podrazumijevana
```

Varijanta `${varijabla:=default}` se ponaša slično, s tim da se u slučaju neinicijalizirane varijable ona i postavlja u zadanu pretpostavljenu vrijednost.

Zamjena naredbe

Zamjena naredbe (*engl. command substitution*) omogućuje da se navedeni tekst protumači i izvede kao naredba, te se zatim zamijeni generiranim izlazom izvršene naredbe. Sintaksno, koriste se dva oblika ove vrste širenja. Jedan smo spomenuli u odjeljku vezanom za navodnike, a tu se naredba zapisuje unutar obrnutih jednostrukih navodnika: `` `naredba` ``. Međutim, jednostruki navodnici su prilično neuočljivi u kôdu, pa je preporučljivije koristiti drugi oblik, u kojemu se naredba zapisuje unutar oblih zagrada kojima prethodi prefiks `$`.

Kao primjer, možemo iskoristiti ispis koji generira Unix naredba `date`. U primjeru se posebnim formatnim nizom definira oblik ispisa.

```
$ echo "Danas je $(date "+%A, %d.%m.%Y")"
Danas je Wednesday, 22.02.2012
```

Sličan primjer, samo uz korištenje alternativne sintakse:

```
$ Datum=`date`
$ echo $Datum
Wed Feb 22 15:38:24 CEST 2012
```

Aritmetičko širenje

Aritmetičko širenje omogućuje evaluaciju aritmetičkih izraza i njihovu zamjenu u naredbenom retku rezultatom izračuna. Ovo širenje je sintaksno dosta slično zamjeni naredbe, pa treba pripaziti da se ne pobrkaju. Ovdje se koriste dvostruke oble zagrade: `$((aritmetički izraz))`.

```
$ echo $((3*7+15/3))
26
```

Prije evaluacije aritmetičkog izraza obavljaju se sva ranije navedena širenja, pa u izrazima možemo koristiti varijable.

```
$ a=5
$ echo $(($a*3%4))
3
```

Aritmetički izrazi mogu se gnijezditi, a za neispravne izraze ispisuje se obavijest o pogrešci.

Dijeljenje riječi

Nakon što se obavi niz prethodno prikazanih zamjena i širenja u naredbenom retku, obavlja se podjela retka na riječi. Pritom valja napomenuti da se nad znakovnim nizovima koji su zapisani unutar navodnika (jednostrukih ili dvostrukih) dijeljenje na riječi ne obavlja.

Dijeljenje na riječi koristi sadržaj posebne sustavske varijable `IFS`, čiji sadržaj određuje listu znakova na kojima se redak dijeli na riječi. Uobičajeno su u toj varijabli znakovi razmak, tabulator i prelazak u novi red, pa se dijeljenje obavlja upravo na tim znakovima. Još je češće ta varijabla neinicijalizirana, što izaziva dijeljenje na upravo tim, podrazumijevanim, znakovima.

Širenje imena datoteka

Nakon dijeljenja ulaznog retka na riječi, pristupa se širenju imena datoteka, koje je poznato i pod imenom *globbing*. U svakoj riječi traže se znakovi `*`, `?` i `[`, te ako se pronađe neki od tih znakova, riječ se smatra uzorkom (*engl. pattern*) i zamjenjuje se abecedno poredanom listom imena datoteka koje odgovaraju uzorku. Uzorak je donekle sličan regularnim izrazima, o kojima će kasnije biti riječi, no radi se o nešto jednostavnijem mehanizmu. Svakako te dvije notacije za podudaranje uzoraka teksta treba razlikovati i zapamtiti da se kod širenja imena datoteka koristi jednostavnija.

U uzorku svaki znak (osim posebnih) predstavlja sam sebe, specijalni znakovi koji se žele doslovno podudarati moraju biti predznačeni znakom "\". Značenja posebnih znakova su:

- `*` predstavlja proizvoljan (pod)niz znakova, uključujući i prazan niz;
- ? predstavlja jedan proizvoljan znak;

- izraz `[...]` podudara se s jednim znakom iz skupa znakova navedenih unutar zagrada.

Izraz koji se navodi unutar uglatih zagrada predstavlja listu znakova koji se pri podudaranju mogu naći na specificiranom mjestu. Unutar zagrada može se zadati i raspon znakova, primjerice `[a-z]` predstavlja mala slova engleske abecede, a `[0-5]` se može podudariti sa znamenkom između 0 i 5. Lista znakova može se i negirati, tako da se kao prvi znak navede "^". Tako izraz `[^012]` označava da na zadanom mjestu može stajati bilo koji znak koji nije znamenka između 0 i 2.

Standard POSIX.2 definira i oznake za posebne *klase znakova*, koje se označavaju u formatu `[:klasa:]`, pri čemu su predviđene oznake za klase:

`alnum alpha ascii blank cntrl digit graph lower print punct space upper word xdigit`

Prilikom korištenja klasa znakova u uzorcima za širenje imena datoteka, one se moraju navesti unutar liste znakova, pa konačni izraz obično sadrži dvostruke uglate zagrade! To je ilustrirano sljedećim primjerom.

```
$ ls [[:alpha:]]*[[:digit:]]*}
skriptni_P2.pdf  skriptni_P2.ps  skriptni_P2.tex
```

Valja napomenuti da se ni ova vrsta širenja ne obavlja nad nizovima zapisanima unutar jednostrukih ili dvostrukih navodnika. Ilustrirajmo to primjerom izraza koji traži datoteke u tekućem kazalu čiji drugi znak imena je "P".

```
$ echo ?P*
MP3katalog MP3rename1 MP3rename2
```

Ako taj izraz napišemo unutar navodnika, širenje se neće obaviti i ispisuje se upravo taj izraz.

```
$ echo "?P*"
?P*
```

Ukoliko u tekućem kazalu nema niti jedne datoteke koja bi zadovoljila zadani izraz, širenje se ne obavlja i umjesto praznog znaka (što bismo vjerojatno očekivali) ispisat će se zadani izraz. U sljedećem primjeru izraz je primijenjen u kazalu unutar kojega ime niti jedne datoteke nije započinjalo znakom "Y".

```
$ echo Y*
Y*
```

2.5 Složeniji Unix alati

U Unix/Linux distribucijama standardno se isporučuje niz vrlo korisnih alata. Mi ćemo ih upoznati nekoliko koji se često koriste. To su alati za brojanje riječi, redaka i znakova (`wc`), pronalaženje redaka teksta u kojima se nalazi zadani uzorak (`grep`), manipulacija tekstnim tokom (`sed`), sortiranje (`sort`), manipulacija ponovljenim retcima (`uniq`), te traženje datoteka (`find`).

2.5.1 Brojanje riječi, znakova i redaka

Naredba `wc` (*engl. word count*) broji riječi, retke i znakove u zadanoj datoteci ili na standardnom ulazu. Naredbi se opcijama može zadati podatak koji nas zanima, a ako se ne navede ime datoteke, čita se standardni ulaz.

```
wc [opcije] [<datoteka>]
```

Moguće opcije su `w` (ispisuje se broj riječi), `l` (ispisuje se broj redaka) i `c` (ispisuje se broj znakova). U nastavku je nekoliko primjera.

```
$ wc /etc/passwd     # retci, riječi, znakovi
40   60 1873 /etc/passwd

$ wc -c /etc/passwd  # samo znakovi
1873 /etc/passwd

$ wc -wc /etc/passwd   # riječi i znakovi
60 1873 /etc/passwd

$ ls -l /usr/include | wc -l   # čita se stdin
```

2.5.2 Pretraživanje tekstnih datoteka

Za pretraživanje sadržaja tekstnih datoteka koristimo naredbu `grep`. Uzorak zadan *regularnim izrazom* traži se u zadanoj datoteci ili u podacima sa standardnog ulaza, redak po redak. Svaki redak u kojem se pronađe zadani uzorak ispisuje se na standardni izlaz.

Naredba podržava zadavanje mnoštva opcija kojima se određuje njeno ponašanje. Ovdje ćemo navesti samo neke važnije opcije. Oblik naredbe je:

```
grep [opcije] trazeni_uzorak [<datoteka>]
```

a neke češće opcije su:

- `-v` ispisuju se linije koje ne sadrže zadani uzorak
- `-i` pretraživanje bez razlikovanja malih i velikih slova
- `-c` ne ispisuju se retci teksta, već samo broj redaka u kojima je uzorak pronađen
- `-E` `trazeni_uzorak` predstavlja regularni izraz s proširenom sintaksom
- `-q` nema ispisa, izlazni status označava je li uzorak pronađen

Uzorak se zadaje regularnim izrazom, a to je posebna notacija koja omogućava definiranje skupova znakovnih nizova. Oni koji su slušali predmet ”Uvod u teoriju računarstva”, sjetit će se da je skup znakovnih nizova jezik, te da su regularni izrazi notacija koja omogućuje definiranje *regularnih jezika*. Kako se regularni izrazi koriste puno šire od same naredbe `grep`, posvetit ćemo im sljedeći odjeljak. Spomenimo da `grep` prihvaća dvije sintakse regularnih izraza (osnovnu i proširenu), a kako je proširena sintaksa nešto elegantnija i danas raširenija, koristit ćemo nju. Stoga ćemo obično zadavati opciju `-E`.

Regularni izraz može biti i običan znakovni niz, bez specijalnih operatora, pa najjednostavnija uporaba `grep`a za jednostavno pronalaženje nekih podnizova. Primjerice, mogli bismo u našim programima pisanima u C-u potražiti datoteke u kojima smo koristili biblioteku *OpenCV*, odnosno programe koji uključuju datoteku zaglavlja `cv.h`.

```
$ grep "<cv.h>" *.c
```

Uočite da će u ovom retku doći do širenja imena datoteka, pa će `grep` kao argumente dobili listu datoteka. Ispisat će se retci u kojima se pojavio zadani uzorak, a uz svaki takav redak bit će ispisano i ime datoteke u kojoj se taj redak nalazi.

Slično, možemo saznati podatke o korisniku koji su zapisani u datoteci `/etc/passwd`.

```
$ grep zekoslav /etc/passwd
```

Često je prikladno `grep` primijeniti nad izlazom neke druge naredbe. Primjerice, popis varijabli ljuske možemo dobiti naredbom `set`. Kako je obično taj popis dugačak, neprikladno je u njemu tražiti neku varijablu koja nas zanima. Ako bismo htjeli pogledati koji je sadržaj varijable `IFS`, mogli bismo to postići i na sljedeći način.

```
$ set | grep IFS
IFS=$' \t\n'
```

2.5.3 Regularni izrazi

Kao što smo već spomenuli, regularni izraz je uzorak zadan u posebnoj notaciji, koji *opisuje* skup znakovnih nizova. Regularni izrazi grade se od jednostavnijih regularnih izraza primjenom posebnih operatora., pri čemu je osnovni građevni blok je regularni izraz koji predstavlja pojedinačni znak. Većina znakova (npr. sva slova i znamenke) predstavljaju sami sebe, a posebni znakovi (metaznakovi) se mogu tumačiti doslovno tako da ih se predznači znakom "\".

Kao i kod izraza za širenje imena datoteka (globbing), i u regularnim izrazima mogu se navoditi liste znakova. Sintaksa je praktički jednaka. Ponovimo, unutar uglatih zagrada "`[ ]`" navodi se lista znakova s kojima se može podudarati (jedan) znak na odgovarajućem mjestu traženog uzorka. Ako je prvi znak unutar zagrada "`^`", lista se tumači kao negacija, tj. navedeni znakovi se na tom mjestu ne smiju nalaziti. Unutar zagrada može se definirati i raspon, određen s dva znaka između kojih je znak "`-`". Također, mogu se koristiti i POSIX klase znakova navedene u odjeljku 2.4.2 Mogu se koristiti i sinonimi za alfanumeričke znakove i njihovu negaciju. Oznaka "`\w`" je sinonim za `[[:alnum:]]`, a "`\W`" sinonim za `[^[:alnum:]]`.

Unutar liste većina metaznakova gubi svoje specijalno značenje, kako bismo u listu uključili i znakove koji je inače označavaju, treba ih posebno pozicionirati. Tako znak "`]`" mora biti naveden na početku liste, kako se "`^`" ne bi tumačio kao negacija, ne smije stajati na početku liste, a znak "`-`" mora biti naveden na kraju liste.

Točka "`.`" je oznaka da se na tom mjestu može naći bilo koji (jedan) znak. Ako povučemo paralelu sa notacijom širenja imena datoteka, sjetimo se da se tamo za istu namjenu koristio znak "?".

U regularnim izrazima koristi se i niz oznaka koje označavaju mjesto u retku na kojem se traženi podniz može nalaziti. Te se oznake podudaraju s praznim nizom, a nazivaju se sidrima. Znak "`^`" označava *početak retka*, a znak "`$`" *kraj retka*. Primjerice, naredba `grep '\.eps$' *.txt` pronaći će samo one retke koji završavaju podnizom "`.eps`". Uočimo i da je točka predznačena kako bi izgubila svoje specijalno značenje u regularnom izrazu.

Slično, oznake "\<" i "\>" označavaju početak odnosno kraj riječi , pa se izraz "`\<pra`" može podudariti s riječju "`pravednost`" ali ne i "`naprasno`", dok izraz "`ba\>`" pronalazi riječ "`grba`" ali ne i "`banalno`"

Sličnu namjenu ima i oznaka "`\b`", koja odgovara granici riječi. Stoga se izraz "`\bpra`" podudara s "`pravednost`" ali ne s "`naprasno`", a izraz "`ba\b`" pronalazi riječ "`grba`" ali ne riječ "`banalno`".

Suprotna je oznaka "`\B`", koja odgovara svemu *osim* granici riječi, pa izraz "`\Bpra`" pronalazi riječ "`naprasno`" ali ne i "`pravednost`", dok izraz "`ba\B`" pronalazi "`banalno`" ali ne "`grba`".

U regularnim izrazima koristi se i nekoliko operatora ponavljanja, odnosno *kvantifikatora*:

`?`	prethodni izraz se pojavljuje najviše jednom
`*`	prethodni izraz se pojavljuje 0 ili više puta
`+`	prethodni izraz se pojavljuje jednom ili više puta
`{n}`	prethodni izraz se pojavljuje točno `n` puta
`{n,}`	prethodni izraz se pojavljuje `n` ili više puta
`{n,m}`	prethodni izraz se pojavljuje barem `n` ali najviše `m` puta

Dva regularna izraza mogu se nadovezati, a tako složen izraz podudara se s nizovima koji se sastoje od dva podniza koji odgovaraju nadovezanim regularnim izrazima (ulančavanje). Također, dva regularna izraza mogu se povezati operatorom "`|`" (unija). Takav složeni izraz podudara se s nizovima koji se podudaraju s bar jednim od dva povezana regularna izraza.

Operacija ponavljanja obavlja se prva, zatim slijedi ulančavanje, a tek nakon toga se primjenjuje operacija unije. Redoslijed primjene ovih operator može se izmijeniti zagradama. Osim izmjene redoslijeda operacija, zagrade i grupiraju dijelove regularnog izraza.

Još jedna važna operacija koju omogućavaju regularni izrazi je povezivanje unazad (*engl. backreference*). Ona se koristi tako da se u izrazu može pozvati na već pronađena podudaranja. Referencirati se možemo samo na dijelove regularnog izraza koji su grupirani oblim zagradama. Sama referenca označava se dekadskom znamenkom predznačenom znakom "\". Vrijednost znamenke označava redni broj grupe na koju se referenciramo.

Da bi bilo jasnije o čemu se radi, ilustrirat ćemo to jednim jednostavnim primjerom. Recimo da u datoteci `rodjendani.txt` imamo zapisane datume rođenja naših prijatelja, u svakom retku po jedan. Željeli bismo pronaći one prijatelje čiji datum i mjesec rođenja se podudaraju. Naravno, možemo primijeniti naredbu `grep`, uz odgovarajući regularni izraz, pri čemu ćemo povratnom referencom zahtijevati da mjesec bude jednak danu rođenja.

```
$ grep '\([0-9]\{2\}\)\.\1' rodjendani.txt
pero 03.03.1981.
ivica 12.12.1977.
```

Ovdje možemo i komentirati razliku osnovne (*engl. basic*) i proširene (*engl. extended*) sintakse regularnih izraza. Razlika je zapravo malena, a radi se o tome da osnovna sintaksa zahtijeva da *metaznakovi* budu predznačeni: `\?`, `\+`, `\{`, `\|`, `\(`, `\)`, dok proširena sintaksa podrazumijeva da su nepredznačeni znakovi metaznakovi, a ako ih želimo koristiti u njihovom doslovnom značenju, onda ih trebamo predznačiti. Kako se posebni znakovi puno češće koriste upravo kao metaznakovi, proširena sintaksa je jasnija i preglednija. Za usporedbu možemo pogledati kako bi prethodni izraz bio zapisan u proširenoj sintaksi. Primijetimo da je točku u obje sintakse potrebno predznačiti kako se ne bi podudarala sa svakim znakom.

```
$ grep -E '([0-9]{2})\.\1' rodjendani.txt
```

Zato ćemo u pravilu koristiti proširenu sintaksu, odnosno opciju `grep -E`.

2.5.4 Sortiranje teksta

Za sortiranje teksta koristimo naredbu `sort`. U osnovnoj primjeni, zadani ulazni tekst se slaže (redak po redak) po (engleskoj) abecedi i zapisuje na standardni izlaz

```
$ sort imena.txt > imena_sortirano.txt
```

Naredba nudi mnoštvo opcija kojima se podešava ponašanje naredbe, a njen oblik je:

```
sort [opcije] [<datoteka>]}
```

Neke često korištene opcije su:

`-u`	jednaki retci se ne ispisuju dvaput
`-r`	sortiranje obrnutim redoslijedom
`-o`	navodi se ime izlazne datoteke
`-n`	numeričko sortiranje
`-k N,M`	sortiranje se obavlja na temelju teksta od N-tog do M-tog polja, uobičajeni graničnici su praznina i `tab`
`-t`	navodi se drugačiji graničnik

Kao primjer, možemo navesti numeričko sortiranje po trećem polju, pri čemu je graničnik znak ":"

```
$ sort -n -k 3 -t : /etc/passwd
```

2.5.5 Pronalaženje ponovljenih redaka

Naredba uniq služi za pronalaženje ponovljenih redaka teksta, pri čemu se ponavljanjem smatra *uzastopno* ponavljanje. Kako bismo mogli pronaći retke koji se u tekstu pojavljuju na proizvoljnim mjestima, tekst je nužno najprije sortirati.

```
$ sort imena.txt | uniq > imena_2.txt
```

Ako se prisjetimo da sort ima opciju -u, zaključit ćemo da smo isti rezultat mogli postići i bez naredbe uniq.

```
$ sort -u imena.txt > imena_2.txt
```

Međutim, uniq ima niz opcija koje omogućuju ostvarivanje još nekih funkcija. Oblik naredbe je:

```
uniq [opcije] [<in_file> [<out_file>]]
```

Neke često korištene opcije:

-d	ispisuju se samo ponovljeni retci
-c	ispisuje se i broj ponavljanja
-f N	iz usporedbe se isključuje prvih N polja
-s N	iz usporedbe se isključuje prvih N znakova
-i	ne uzima se u obzir razlika između velikih i malih slova

Korištenje nekih opcija možemo ilustrirati s par primjera. Sljedećom naredbom ćemo provjeriti postoje li duplikati u datoteci /etc/passwd:

```
$ sort /etc/passwd | uniq -d
```

Ako pak želimo ispisati broj ponavljanja višestrukih redaka teksta, možemo to postići ovako:

```
$ sort imena.txt | uniq -c
```

2.5.6 Manipulacija tekstom

Zasigurno ste se susreli s potrebom za izmjenama nekih dijelova teksta u programu za uređivanje teksta. U takvoj situaciji vrlo je korisna mogućnost pretraživanja i zamjene (*engl. Search/Replace*). Posebno je prikladno da takav alat podržava i regularne izraze. No, ako slične sekvence izmjena trebamo napraviti u različitim datotekama, ili ih trebamo obavljati u više navrata, bilo bi ih dobro moći pohraniti u obliku neke vrste programa.

Naredba sed (*engl. stream editor*) omogućava manipulaciju tekstom koji se čita iz datoteke ili sa standardnog ulaza. Osnovne operacije odgovaraju upravo pretraživanju i zamjeni dijelova teksta koji se zadaju regularnim izrazima. Više naredbi za sed moguće je pohraniti u posebnu datoteku i na taj način oblikovati program, no sed se može pozivati i iz skripte napisane za ljusku. Mi ćemo se zadržati na jednostavnijim primjerima korištenja.

Oblik naredbe je:

```
sed [opcije] [-e upute] [-f upute_dat][<datoteka>]...
```

Značenja opcija su sljedeća:

-n	ne ispisuje se izlazni rezultat
-r	koriste se regularni izrazi proširene sintakse
-e upute	navode se naredbe za sed, a ako je samo jedna naredba i ako nema opcije -f, onda se oznaka -e se može izostaviti
-f upute_dat	upute se čitaju iz datoteke upute_dat
<datoteka>	datoteka s ulaznim tekstom

Akcija zamjene dijelova teksta označava se kao "s/reg_ex/zamjenski_tekst/". Primjerice, ako bismo htjeli zamijeniti sve pojave niza "Unix" u datoteci intro.txt tekstom "UNIX", možemo to učiniti sljedećom naredbom:

```
$ sed 's/Unix/UNIX/' intro.txt
```

U ovom slučaju, tekst u kojemu su obavljene izmjene ispisuje se na standardni izlaz. Naravno, ako želimo pohraniti rezultat, možemo preusmjeriti standardni izlaz.

```
$ sed 's/Unix/UNIX/' intro.txt > intro2.txt
```

Ovdje valja napomenuti da se obavlja *samo jedna* pojava navedenog niza u pojedinom retku. Ako želimo zamijeniti sve podnizove koji odgovaraju navedenom regularnom izrazu, treba navesti opciju g koja označava globalnu primjenu zadane akcije.

```
$ sed 's/Unix/UNIX/g' intro.txt > intro2.txt
```

Ako želimo obrisati dio teksta, naprosto zamjenski niz ostavimo praznim. Primjerice, ako bismo htjeli obrisati zadnja 3 znaka u svakom retku teksta, možemo to odrediti regularnim izrazom i primijeniti sljedeću naredbu.

```
$ sed 's/...$//' datoteka.txt
```

Opcija -n isključuje ispis rezultata, a akcija pretraživanja zadaje se zapisivanjem samo regularnog izraza, bez zamjenskog niza: "/reg_ex/". Kako bismo ispisali redak u kojem je pronađen zadani uzorak, dodat ćemo opciju p:

```
$ sed -n '/UNIX/p' intro.txt
```

Retci u kojima je pronađen zadani uzorak mogu se i obrisati, i to navođenjem opcije d:

```
$ sed '/UNIX/d' intro.txt
```

2.5.7 Traženje datoteka

Za traženje datoteka koje odgovaraju nekim kriterijima, koristi se naredba find. To je složena naredba s mnoštvom opcija, kojima se može definirati različite kriterije pretraživanja. Oblik naredbe je:

```
find [opcije] [direktoriji] [uvjeti]
```

Značenja češće korištenih uvjeta su sljedeća:

-name <ime>	traži se datoteka zadanog imena
-type <tip>	traži se datoteka zadanog tipa d – kazalo, f – obična datoteka, l – simbolički link
-size <velicina>	traži se datoteka zadane veličine
-print	ispisuje se ime pronađene datoteke
-mtime n	traže se datoteke mijenjane prije n*24h
-regex izraz	traže se datoteke čije ime (uključujući stazu) odgovara zadanom regularnom izrazu

Najčešće se traži datoteka čije nam je ime poznato, ali ne znamo gdje se nalazi, pa možemo primijeniti naredbu poput sljedeće.

```
$ find ~ -name Skriptni_0_Uvod.tex
```

U ovom primjeru tražimo datoteku čije ime točno znamo, a pretragu započinjemo od našeg korisničkog kazala, kroz sva podkazala. No, često znamo samo fragment imena. Uvjet -name

omogućuje nam zadavanje uzorka po sintaksi koja se koristi kod širenja imena datoteka. Pritom treba paziti da se uzorak zapiše u navodnicima kako ga ne bi proširila ljuska. U tom bi slučaju naredba `find` dobila pogrešne argumente.

```
$ find ~ -name "Skrip*.tex"
```

Uz to, moguće je koristiti i regularni izraz za zadavanje imena. U tom se slučaju koristi uvjet `-regex`. Pritom se regularni izraz primjenjuje na cjelovito ime datoteke, koje uključuje i cijelu stazu do nje. I ovdje uzorak treba staviti pod navodnike da ljuska ne obavlja širenje.

```
$ find ~/ -regex ".*Skr.*P2.*\.pdf'"
```

Ako zaboravimo napisati dio izraza koji se podudara sa stazom, upit neće dati rezultat.

```
$ find ~/ -regex "Skr.*P2.*\.pdf'"   # ovo nije dobro!
```

Naredba `find` omogućuje pretraživanje i po drugim kriterijima. Primjerice, možemo tražiti datoteke neke veličine. To se postiže uvjetom `-size`, a može se i označiti da se traži veća ili manja veličina od zadane, što se postiže dodavanjem modifikatora `+` (veće) ili `-` (manje). Primjerice, možemo potražiti datoteke u kazalu `/etc` koje su veće (+) od 1 MB:

```
$ find /etc -size +1M
```

ili datoteke koje su manje (-) od 1 kB:

```
$ find /etc -size -1k
```

Upit možemo temeljiti i na vremenu izmjene datoteke. Primjerice, mogli bismo tražiti datoteke u svom matičnom kazalu koje su mijenjane danas, a uz to možemo ispisati i detaljnije podatke o nađenim datotekama (dodatak `-ls`):

```
$ find ~ -mtime 0 -ls
```

Interval se zadaje u danima, odnosno u višekratnicima od 24 sata, pa `0` znači da nije proteklo više od 24 sata od izmjene. Ako bismo htjeli pronaći datoteke izmijenjene u zadnja 3 dana, primijenili bismo sljedeću naredbu:

```
$ find ~ -mtime -3
```

Može se zadati i tip datoteke, pri čemu `-type d` označava da se radi o direktoriju, `f` označava običnu datoteku, a `l` simbolički link. Tako bismo sve direktorije unutar `/etc` pronašli naredbom:

```
$ find /etc -type d
```

Uvjeti se mogu i kombinirati, pri čemu se logički veznik I označava `-a`, a logički veznik i s `-o`. Grupiranje podizraza postiže se zagradama, koje međutim treba predznačiti. Primjerice, mogli bismo tražiti sva kazala i datoteke veće od 1 MB:

```
$ find /etc \( -type d -o -size +1M \) -ls
```

Ovdje još valja napomenuti da je često prikladnije umjesto naredbe `find` koristiti naredbu `locate`. Ona koristi unaprijed izgrađenu bazu podataka s imena datoteka, pa je pretraga značajno brža i manje opterećuje računalne resurse. Baza imena datoteka mora se periodično obnavljati, naredbom `updatedb`. Oblik naredbe `locate` je:

```
locate [<opcije>] <uzorak>
```

I ovdje se uzorak navodi po sintaksi koja se koriste pri širenju imena datoteka, a može biti i regularni izraz (opcija `-r`). Naravno, uzorke treba navesti unutar navodnika kako ljuska ne bi obavila širenje. Ukoliko želimo zanemariti razliku između malih i velikih slova, navodi se opcija `-i`.

2.6 Pisanje i pokretanje skripti

Skripte su obične tekstne datoteke, pa za njihovo upisivanje i uređivanje koristimo uređivač teksta. Skriptu možemo prikazati naredbom `cat` ili naredbom `less`. Jedna jednostavna skripta, prikazana naredbom `cat` može biti:

```
$ cat hello.sh
echo "Hello world!"
```

Skriptu možemo pokrenuti na nekoliko načina. Prvi je primjenom naredbe `source`. Navodi se ime skripte, odnosno datoteke s naredbama, te se naredbe izvršavaju u tekućoj okolini ljuske. Efekt je praktički kao da unosimo naredbu po naredbu u naredbenom retku.

```
$ source hello.sh
```

U ljusci `bash` postoji i alternativni način zapisivanja naredbe `source`, a to je točka. Dakle, isti efekt kao u gornjem primjeru dobivamo sljedećom naredbom (ovdje točka predstavlja naredbu `source`, a ne tekuće kazalo.

```
$ . hello.sh
```

Još jednom, na ovaj način skriptu izvršava ljuska iz koje smo je pokrenuli, pa i sve promjene u varijablama okoline ostaju sačuvane. Nasuprot tome, drugi način pokretanja skripte je moguć tek nakon što skriptu proglasimo izvršivom primjenom naredbe `chmod`.

```
$ chmod u+x hello.sh    # skripta postaje izvršiva
```

Nakon toga je skriptu moguće pokrenuti kao bilo koji program (ovdje točka označava tekuće kazalo i na taj način se ljusci navodi put do datoteke koju treba izvršiti.

```
./hello.sh
```

U ovom slučaju, pokreće se podljuska u kojoj se skripta izvršava. Po završetku izvršavanja skripte, napušta se i podljuska, pa se promjene varijabli okoline gube. Podljuska koja se pokreće je istog tipa kao i ljuska u kojoj je skripta pokrenuta, no moguće je navesti i drugu ljusku koja će izvršiti skriptu. Točnije, postoji mehanizam kojim se zadaje interpreter koji će izvršiti skriptu. Kasnije ćemo vidjeti da to može biti i Perl ili Python interpreter. Notacija se sastoji u tome da se u *prvom retku* skripte navede interpreter koji će izvršiti skriptu, s tim da su prva dva znaka "`#!`" (kolokvijalno se ta sekvenca naziva *sha-bang*).

```
#!/bin/sh

echo "Ljusku će pokrenuti ljuska sh"
echo "Hello world!"
```

Ulazni podaci mogu se skripti predati kao varijable ljuske ako se skripta pokreće u istoj ljusci:

```
$ a=3
$ cat > ispisi.sh  # ovako možemo upisati kratku skriptu
echo $a
$ source ispisi.sh
3
```

Međutim, ako se skripta pokreće u podljusci, na raspolaganju su joj samo varijable okoline (eksportirane varijable):

```
$ chmod u+x ispisi.sh
$ ./ispisi.sh   # skripta ne vidi varijablu

$ export a
$ ./ispisi.sh   # sada je već bolje
3
```

2.6.1 Argumenti naredbenog retka

Kao što smo vidjeli u prethodnom odjeljku, podaci se u skriptu mogu prenijeti kroz varijable okoline. No, uobičajen način prijenosa podataka je u obliku argumenata naredbenog retka koji se navode iza imena skripte koja se pokreće. Te smo *pozicijske* parametre spomenuli i u odjeljku o širenjima u naredbenom retku, a oni automatski dobijaju numerička imena koja odgovaraju njihovim rednim brojevima. Pritom parametar `$0` sadrži ime skripte ako je skripta pokrenuta u podljusci, odnosno ime same ljuske ako je skripta pokrenuta naredbom `source`.

```
$ cat parametri.sh
#!/bin/bash
# skripta koja ispisuje prva 3 parametra

echo $0
echo $1
echo $2

$ ./parametri.sh par1 par2
./parametri.sh
par1
par2

$ source parametri.sh par1 par2
bash
par1
par2
```

Parametri s rednim brojem većim od 9 označavaju se vitičastim zagradama:
`${10}`, `${11}` itd.

Naredbom `shift` obavlja se promjena imena pozicijskih parametara: indeksi parametara počevši od `$2` umanjuju se za 1. Ako se naredbi zada numerički argument (`shift n`), oznake argumenata umanjuju se za `n`, odnosno argument `${n+1}` postaje `$1`, `${n+2}` postaje postaje `$2` itd.

Broj prenesenih pozicijskih parametara navedenih u naredbenom retku može se pročitati iz varijable `$#`, dok se lista svih argumenata predanih skripti može se dobiti s `$@`.

```
$ cat parametri2.sh
echo "Broj parametara je: $#"
echo "Parametri su: $@"
echo $0; echo $1; echo $2; echo $3

$ ./parametri2.sh p1 p2 p3
Broj parametara je: 3
Parametri su: p1 p2 p3
./parametri2.sh
p1
```

```
p2
p3
```

2.6.2 Ispitivanje uvjeta

Svaka naredba (program, skripta) pri završetku izvođenja vraća *izlazni status*. To je broj koji obično označava je li se program uspješno izvršio. Uobičajeno izlazni status `0` označava uspješno izvršavanje, dok vrijednost različita od `0` označava neuspješno izvršavanje ili pogrešku. Primjerice, naredba `grep` će vratiti izlazni status `0` ako je traženi uzorak barem jednom pronađen, a neku drugu vrijednost ako uzorak nije pronađen ili se dogodi neka pogreška (neispravno navedeni argumenti, nečitljiva datoteka).

Izlazni status zadnje izvedene naredbe (ili skripte) pohranjuje se u varijablu ”`$?`”. Kada se izvršava ulančani niz naredbi, izlazni status vraća posljednja naredba u lancu.

Naredba if

Za ispitivanje uvjeta i odgovarajuće usmjeravanje programskog toka, koristi se naredba `if`. Njen najjednostavniji oblik je:

```
if naredba_t
then
    blok_naredbi
fi
```

Naredba `naredba_t` se izvršava, te se ispituje njen izlazni status. Ukoliko je status `0`, blok naredbi između `then` i `fi` se izvršava.

```
korisnik=$1   # ulazni argument je korisnicko ime
if grep -q $korisnik /etc/passwd
then
   echo ”Korisnik $korisnik postoji.”
fi
```

Složeniji oblici naredbe `if` mogu uključivati i blok `else`, koji se izvršava ukoliko uvjet nije ispunjen, odnosno blok `elif` kojim se uvodi ispitivanje sljedećeg uvjeta.

```
if naredba_1 ; then
    blok_naredbi1
elif naredba_2 ; then
    blok_naredbi2
else
    blok_naredbi3
fi
```

Naredba test

U naredbi `if` ispituje se izlazni status proizvoljne naredbe. No, kako bismo ispitali specifične uvjete, obično se koristi naredba `test`. Ona ispituje navedene uvjete i vraća odgovarajući izlazni status, koji se potom može ispitati u naredbi `if`. Opći oblik naredbe `test` je:

```
test izraz
```

pri čemu `izraz` predstavlja uvjete koji se ispituju. Ako je izraz istinit, naredba `test` vraća izlazni status `0`. Ako je izraz *neistinit*, izlazni status je različit od `0`. Primjerice, izraz za ispitivanje znakovnih nizova može biti:

```
test "$name" = Vedrana
```

Ovdje treba paziti da znak ispitivanja jednakosti bude *odvojen* od operanada, suprotno od operacije dodjeljivanja vrijednosti varijabli! Osim toga, dobro je varijable ljuske koje se prosljeđuju kao argumenti naredbi `test` uokviriti dvostrukim navodnicima, jer ako je varijabla neinicijalizirana (prazna), širenjem parametara će *nestati*. Navodnici će osigurati da i u tom slučaju naredba vidi argument.

Još jedan primjer:

```
dan=$(date +%A)
if test "$dan" = Sunday ; then
    echo "Danas je nedjelja."
fi
```

Naredba `test` podržava još nekoliko operatora za usporedbu znakovnih nizova:

`string1 = string2`	istinito ako su nizovi jednaki
`string1 != string2`	istinito ako su nizovi različiti
`string`	niz nije prazan
`-n string`	niz nije prazan (mora biti vidljiv naredbi `test`)
`-z string`	niz je prazan (mora biti vidljiv naredbi `test`)

Nekoliko primjera:

```
$ test "" ; echo $?
1

$ test "a" ; echo $?
0

$ test -n "a" ; echo $?
0

$ test -z "" ; echo $?
0
```

Naredba `test` ima i alternativni oblik, koji je čitljiviji, pogotovo kada se koristi kao uvjet u naredbi `if`. Umjesto naredbe `test izraz` može se napisati `[ izraz ]`. Pritom treba paziti da se izraz odvoji od zagrada, inače će `bash` prijaviti sintaksnu pogrešku. U toj notaciji možemo pisati:

```
dan=$(date +%A)
if [ "$dan" = Tuesday ] ; then
    echo "Danas je utorak."
fi
```

Osim znakovnih nizova, mogu se uspoređivati i brojevi. Za te usporedbe koristi se poseban skup operatora:

`int1 -eq int2`	istinito ako su brojevi jednaki
`int1 -ge int2`	istinito ako je `int1 >= int2`
`int1 -gt int2`	istinito ako je `int1 > int2`
`int1 -le int2`	istinito ako je `int1 <= int2`
`int1 -lt int2`	istinito ako je `int1 < int2`
`int1 -ne int2`	istinito ako je `int1 != int2`

Primjenu ovih operatora možemo ilustrirati na nekoliko primjera:

```
$ x1="005"
$ [ "$x1" = 5 ] ; echo $?   # usporedba nizova
1

$ [ "$x1" -eq 5 ] ; echo $?   # usporedba brojeva
0

$ [ "$x1" -lt 15 ] ; echo $?   # usporedba brojeva
0
```

Naredba `test` može se primijeniti i za ispitivanje svojstava datoteka. Primjenjuju se sljedeći operatori:

`-d file`	`file` je kazalo
`-e file`	`file` postoji
`-f file`	`file` je obična datoteka
`-r file`	čitanje datoteke dozvoljeno
`-s file`	duljina datoteke veća od 0
`-w file`	dozvoljeno pisanje u datoteku
`-x file`	datoteka je izvršna
`-L file`	`file` je simbolički link

Nekoliko primjera:

```
# postoji li (obična) datoteka navedenog imena?
[ -f /users/steve/phonebook ]

# je li nam dozvoljeno čitanje navedene datoteke?
[ -r /users/steve/phonebook ]

# ispitivanje je li datoteka u koju su zapisivane greške
# neprazna, ispis datoteke
if [ -s $ERRFILE ] ; then
    echo "Pronadjene greske: "
    cat $ERRFILE
fi
```

Rezultat evaluacije izraza može se negirati primjenom operatora logičke negacije (`!`).

```
[ ! -r /users/steve/phonebook ]
```

Izrazi se mogu i kombinirati primjenom logičkih operatora I (`-a`) i ILI (`-o`).

```
[ -f "$mailfile"   -a   -r "$mailfile" ]

[ -n "$mailopt" -o -r $HOME/mailfile ]
```

Operator I ima prednost pred operatorom ILI, a za promjenu redoslijeda evaluacije izraza mogu se koristiti *zagrade*. Zagrade je potrebno predznačiti: `"\("`, `"\)"`, a moraju i biti okružene prazninama.

```
[ \( "$count" -ge 0 \) -a \( "$count" -lt 10 \) ]
```

2.6.3 Izlazak iz skripte

Skripta završava kada se izvrše sve njene naredbe. No, skriptu možemo završiti i ranije, korištenjem naredbe `exit`. Ona omogućuje trenutan završetak izvođenja skripte. Naredbi se može

zadati i argument, koji se vraća se kao izlazni status : `exit n`. Ako se argument ne navede, vraća se status prethodno izvedene naredbe.

Primjenu možemo ilustrirati na jednostavnoj skripti koja na temelju proslijeđenog argumenta briše zapis u adresaru. Podaci u adresaru zapisani su u retcima tekstne datoteke, pri čemu se u retku pojavljuje i ime osobe na koju se podaci odnose. Skripta za brisanje zapisa kao argument očekuje ime osobe čiji se podaci trebaju obrisati. Kako ilustriramo primjenu naredbe `exit`, zanimljiv nam je dio u kojem se provjerava broj argumenta. Ako on nije 1, ispisuje se poruka i napušta skripta s izlaznim statusom 1.

```
# brisanje zapisa u adresaru

# provjera broja argumenata
if [ "$#" -ne 1 ]; then
    echo "Incorrect number of arguments."
    echo "Usage: rem name"
    exit 1
fi

grep -v "$1" phonebook > /tmp/phonebook
mv /tmp/phonebook phonebook
```

2.6.4 Naredba case

Naredba case omogućuje usporedbu jedne vrijednosti s nizom drugih vrijednosti i izvođenje bloka naredbi koji odgovara podudaranju vrijednosti. Oblik naredbe je:

```
case value in
   pat_1)   naredba_1
            naredba_2
               ...
            naredba_k;;  # PAZI! ;;
   pat_2)   blok_naredbi_2;;
     ...
   pat_n)   blok_naredbi_n;;
esac
```

Ispitna vrijednost obično se čita iz varijable, a vrijednosti s kojima se ona uspoređuje zadane su u obliku uzoraka kakvi se koriste u širenju imena datoteka. U sljedećem primjeru skripta preuzima jedan argument naredbenog retka, a zatim provjerava je li to znamenka, malo ili veliko slovo ili neki drugi pojedinačni znak (podsjetimo se da upitnik označava jedan znak). Ako niti jedno id ispitivanja nije uspješno, ispisuje se poruka (zvjezdica se podudara sa svakim nizom).

```
char=$1
case "$char" in
   [0-9] ) echo "znamenka";;
   [a-z] ) echo "malo slovo";;
   [A-Z] ) echo "veliko slovo";;
     ?   ) echo "specijalni znak";;
     *   ) echo "molim upisite jedan znak";;
esac
```

2.6.5 Programske petlje

Ljuska bash nudi nekoliko naredbi koje služe za ostvarivanje programskih petlji.

Naredba for

Tipična petlja kojom se prolazi kroz listu vrijednosti ostvaruje se naredbom for. Njen je oblik, a zadaje se varijabla petlje i lista vrijednosti koje varijabla preuzima u svakom prolasku kroz petlju:

```
for var in rijec_1 rijec_2 ... rijec_n
do
   naredba_1
     ...
   naredba_k
done
```

Primjer petlje kojom se prolazi listu brojeva:

```
for i in 1 2 3
do
   echo $i
done
```

Ako je niz brojeva dulji, nezgodno ga je pisati na ovaj način, pa se može primijeniti izraz za generiranje sekvence. Ovdje možemo ilustrirati i kako se naredba for zapisuje u jednom retku (potrebno je umetnuti znak ”;” na mjesta koja odgovaraju prelasku u novi red). Naredbi echo dodat ćemo opciju -n koja sprječava prelazak u novi red nakon ispisa.

```
$ for i in {1..9} ; do echo -n $i; done
123456789
```

Za generiranje liste kroz koju naredba prolazi možemo iskoristiti razne mehanizme ljuske. Jedna od čestih primjena je i iteriranje kroz listu dobivenu širenjem imena datoteka. Primjerice, možemo ispisati listu svih datoteka čiji nastavak imena je ”.sh”

```
for i in *.sh
do
   echo $i
done
```

Možemo prolaziti i kroz listu argumenata naredbenog retka odnosno pozicijskih parametara.

```
for arg in "$@"; do echo $arg; done
```

Ovdje je zgodno napomenuti i da se ova petlja može čak i kraće napisati. Naime, naredba for podrazumijeva upravo prolazak kroz listu argumenata, pa je čak ne treba niti napisati.

```
for arg; do echo $arg; done
```

Možemo iskoristiti i mehanizam zamjene naredbe, odnosno primijeniti naredbu koja će generirati listu. Ako je lista kroz koju želimo iterirati zapisana u datoteci, možemo je iščitati naredbom cat.

```
for file in $(cat lista.txt); do echo $file; done
```

Naredba while

Još jedna uobičajena programska petlja je `while`. Ona se ponavlja dok god je zadani uvjet ispunjen. Uvjet ima isto značenje kao i uvjet za naredbu `if`, odnosno ispituje se izlazni status neke naredbe, tipično naredbe `test`. Oblik naredbe je:

```
while test_uvjet
do
   naredba_1
     ...
   naredba_k
done
```

Ponašanje naredbe možemo ilustrirati s nekoliko primjera:

```
i=1
while [ "$i" -le 5 ]
do
   echo $i
   i=$((i + 1))
done
```

```
while [ -n "$1" ]
do
   echo $1
   shift
done
```

```
while [ "$#" -ne 0 ]
do
   echo $1
   shift
done
```

Naredba until

Sasvim slična je i naredba `until`. Razlika je samo u korištenju uvjeta. Petlja se ponavlja ako uvjet nije ispunjen, odnosno zaustavlja nakon što se uvjet ispuni. Oblik naredbe je:

```
until test_uvjet
do
   naredba_1
     ...
   naredba_k
done
```

Ponašanje naredbe ilustrirajmo s nekoliko primjera:

```
i=1
until [ "$i" -gt 5 ]
do
   echo $i
   i=$((i + 1))
done
```

```
$ cat mon
# Ceka da se zadani korisnik prijavi na sustav

if [ "$#" -ne 1 ] ; then
    echo "Usage: mon user"
    exit 1
fi

user="$1"

# Provjera svake minute: je li se korisnik prijavio?
until who | grep "^$user " > /dev/null ; do
    sleep 60
done

# Korisnik je docekan :-)
echo "$user has logged on"
```

Naredba break

Trenutni izlazak iz petlje može se inicirati naredbom `break`. Naredbi je moguće zadati i numerički argument `n` koji izaziva trenutan prekid izvršavanja `n` unutrašnjih petlji.

```
i=1
while true; do
   echo $i
   i=$((i + 1))
   if [ "$i" -gt 20 ]; then
      break
   fi
done
```

Naredba continue

Naredba `continue` izaziva preskakanje svih preostalih naredbi unutar petlje i nastavlja se sa sljedećim prolazom kroz petlju. Kao i kod naredbe `break`, moguće je zadati dodatni numerički argument. Naredba `continue n` izaziva preskakanje preostalih naredbi unutar `n` unutrašnjih petlji uz normalan nastavak izvršavanja petlji.

```
for file; do
    if [ ! -e "$file" ]; then
        echo "$file not found!"
        continue
    fi
    #
    # obrada datoteke file
    #
done
```

2.6.6 Preusmjeravanje u petlji

U odjeljku 2.2.3 upoznali smo se s preusmjeravanjem standardnog ulaza odnosno izlaza. Ljuska bash omogućava i preusmjeravanje ulaza odnosno izlaza cijele programske petlje. Ulaz preusmjeren u petlju primjenjuje se na sve naredbe koje podatke čitaju sa standardnog ulaza. Slično, izlaz preusmjeren iz petlje primjenjuje se na sve naredbe koje podatke zapisuju na standardni izlaz. To možemo ilustrirati s nekoliko primjera.

```
for i in 1 2 3 4; do
    echo $i
done > izlaz.txt
```

```
for i in 1 2 3 4; do
    head -n 3  # cita iz "lista.txt"
    echo "prolaz: $i "
done < lista.txt
```

2.6.7 Učitavanje i ispis

U ovom odjeljku upoznat ćemo se s naredbama koje omogućuju učitavanje sa standardnog ulaza, odnosno zapisivanje na standardni izlaz.

Učitavanje sa standardnog ulaza

Naredba read koristi se za učitavanje teksta sa standardnog ulaza. Kao argumenti naredbe navodi se lista varijabli.

```
read lista_varijabli
```

Ljuska čita redak sa standardnog ulaza, pohranjuje prvu riječ u prvu varijablu navedenu u listi, drugu riječ u drugu varijablu itd.

Ako je broj navedenih varijabli manji od broja riječi u retku, ostatak retka se pohranjuje u zadnjoj navedenoj varijabli. Primjerice,
`read x y`
učitava prvu riječ u x i ostatak retka u y. Često želimo naprosto učitati cijeli redak u jednu varijablu:

```
read mojRedak
```

Naredba read vraća izlazni status 0, osim kada stigne do kraja datoteke (EOF). Ako se podaci učitavaju s terminala, za označavanje kraja unosa treba utipkati Ctrl+D.

Ponašanje naredbe možemo ilustrirati primjerom u kojem se uzastopno učitavaju po dva broja, te se ispisuje njihov zbroj.

```
while read n1 n2
do
   echo $((n1 + n2))
done
```

Malo složeniji primjer kombinira podatke iz dvije tekstne datoteke. U jednoj se nalaze specifični podaci jedne podgrupe korisnika, primjerice bodovi studenata jedne laboratorijske grupe. Druga datoteka sadrži opće podatke svih korisnika, primjerice podatke o svim studentima koji su upisali kolegij. Zapis o korisniku je pohranjen kao redak tekstne datoteke, a u obje datoteke koristi se jedinstveni identifikator korisnika sastavljen od 10 znamenaka.

Skripta treba za svakog korisnika koji se pojavljuje u prvoj datoteci ispisati podatke iz druge datoteke.

```
while read Redak; do
  mbr=$(echo $Redak|sed -r 's/.*([0-9]{10}).*/\1/')
  grep $mbr svi.txt >> G3.txt || echo "-- $mbr">> G3.txt
done < "popis_G3.csv"
```

Skripta učitava redak po redak prve datoteke (`popis_G3.csv`), koristeći preusmjeravanje u petlju. Iz pročitanog retka isijeca se identifikator korisnika koristeći naredbu `sed` i regularni izraz koji nalazi 10-znamenkasti broj. Identifikator se pohranjuje u varijablu, a zatim se pomoću naredbe `grep` traže retci u drugoj datoteci u kojima se pojavljuje isti identifikator. Retci koje vraća `grep` nadodaju se u ciljnu datoteku.

U ovom primjeru susrećemo se i s *uvjetnim izvođenjem naredbi*, gdje se dvije naredbe kombiniraju logičkim izrazom i druga naredba se izvršava ovisno o rezultatu prve naredbe. U gornjem primjeru koristi se operator ILI (`||`), pa se druga naredba izvršava samo ako je prva završila s izlaznim statusom različitim od 0. Drugim riječima, ako `grep` ne pronađe niti jedan redak u kojem se pojavljuje identifikator korisnika, u ciljnu datoteku zapisuje se oznaka da taj korisnik nije pronađen.

U izrazu za uvjetno izvršavanje naredbi može se upotrijebiti i operator I (`&&`). U tom slučaju se druga naredba izvršava samo ako je prva naredba uspješno završila, odnosno njen izlazni status je 0.

Ispis na standardni izlaz

Za ispis na standardni izlaz do sada smo koristili naredbu `echo`, koja je u većini primjena dovoljna. Međutim, ponekad je potreban *formatirani* ispis, koji se može ostvariti uporabom naredbe `printf`. Ona je vrlo slična istoimenoj funkciji u C-u. Kao argument naredbe najprije se navodi formatni niz, a zatim slijedi niz argumenata koji se ispisuju u skladu sa zadanim formatom.

```
printf "format" arg1 arg2 ...
```

Znakovi formatnog niza `format`, ukoliko nisu predznačeni znakom `"%"`, prosljeđuju se na standardni izlaz. Znakovi predznačeni znakom `"%"` označavaju format u kojem se ispisuju navedeni argumenti. Značenja nekih oznaka su:

`%d`	cijeli broj
`%u`	cijeli broj bez predznaka
`%o`	oktalni broj
`%X`	heksadekadski broj
`%c`	znak
`%s`	niz znakova

Ponašanje naredbe ilustrirajmo s nekoliko primjera.

```
$ printf "Oktalna vrijednost broja \
%d je %o\n" 20 20
Oktalna vrijednost broja 20 je 24

$ printf "Heksadekadska vrijednost broja \
%d je %X\n" 30 30
Heksadekadska vrijednost broja 30 je 1E

$ printf "Nepredznacena vrijednost broja \
%d je %u\n" -1000 -1000
```

```
Nepredznacena vrijednost broja -1000 \
je 18446744073709550616
```

```
while read broj1 broj2; do
  printf "%12d %12d\n" $broj1 $broj2
done
```

3. Programski jezik Perl

Autor programskog jezika Perl je Larry Wall, a prva verzija jezika predstavljena je 1987. godine. Ime jezika je akronim za *Practical Extraction and Report Language*, a u šali autor ga je nazvao i *Pathologically Eclectic Rubbish Lister*. U tom nazivu vjerojatno je nejasna sam riječ *eclectic*, odnosno eklektičan, a ona se može prevesti kao raznorodan odnosno sačinjen od različitih dijelova. Naime, kao što ćete i sami primijetiti upoznajući se s jezikom, Larry Wall je u Perl ugradio elemente iz ljuske operacijskog sustava, raznih Unix alata i C-a. Naime, on je razvio Perl kao alat za vlastite potrebe, dok je nastojao generirati neka izvješća, a primjena skripti u Unix ljusci i alata poput `sed`-a, i `grep`-a mu nije bila dovoljna. Pri oblikovanju novog jezika, vodio se s nekoliko težnji. Želio je imati alat koji će mu omogućiti brzinu kodiranja poput programiranja u Unix ljusci, a istovremeno pružiti mogućnosti naprednijih alata kao što su `grep`, `sort` i `sed`. U tom trenutku nije ni slutio da će njegov alat prerasti u planetarno popularan programski jezik.

Perl nastoji ispuniti prazninu između programiranja niske razine (C, C++, asembler) i programiranja visoke razine (ljuska OS-a). Programiranje niske razine je relativno teško, ali bez ograničenja, pa se uz dobro programiranje može postići maksimalna brzina izvođenja na danom računalu. S druge strane, programiranje visoke razine omogućuje relativno lako i brzo pisanje koda, no programi su najčešće spori, a mogućnosti su ograničene na naredbe koje su nam na raspolaganju. U Perlu je relativno lako programirati, malo je ograničenja, a izvođenje je uglavnom brzo. Zahvaljujući svojoj snazi i fleksibilnosti, Perl se nametnuo kao jedan od najpopularnijih programskih jezika, posebno u domeni WWW programiranja, obrade teksta i administracije sustava.

Perl ima potpunu podršku regularnim izrazima, podržava objektno-orijentirano programiranje, ima podršku za mrežno programiranje, omogućava upravljanje procesima, proširiv je i podržava razvoj prenosivih programa. K tome je besplatan i otvorenog koda.

Na kraju ovog uvoda, spomenimo i moto Perla: ”Postoji više od jednog načina da se to učini” (*engl. ”There's More Than One Way To Do It”*).

3.1 Pisanje i pokretanje Perl programa

Perl program je obična tekst datoteka, pa je za njegovo pisanje dovoljan tekst editor. Program pohranimo u datoteku, a zatim pozovemo `perl` uz navođenje imena datoteke s programom kao argumenta.

```
#!/usr/bin/perl

# moj_program.pl
# ovo je jednostavan program u Perlu

print "Hello, world!\n";
```

```
$ perl moj_program.pl
```

Ako radimo pod operacijskim sustavom Unix, ne moramo eksplicitno pozivati `perl`, već možemo datoteku učiniti izvršivom i pozivati kao program (tako smo pozivali i skripte pisane u ljusci).

```
$ chmod a+x my_program.pl
$ ./moj_program.pl
```

Naravno, da bi ovakav način pozivanja funkcionirao, nužan je prvi redak skripte koji određuje interpreter koji će skriptu izvršiti (sekvenca `#!` na početku prvog retka, *sha-bang*):

```
#!/usr/bin/perl
```

To je ujedno i najmanje portabilan dio Perl programa, jer smještaj Perl interpretera može varirati. Osim toga, taj redak ne pomaže u pokretanju programa na ne-Unix operacijskim sustavima. Međutim, tradicionalno se koristi i na ne-Unix sustavima. Na taj način je pri čitanju programa odmah jasno da se radi o programu u Perlu. Osim toga, u tom retku se mogu navesti i neke opcije za interpreter, kao što je uključivanje dijagnostičkih upozorenja (`-w`), a te opcije interpreteri čitaju neovisno o operacijskom sustavu.

```
#!/usr/bin/perl -w
```

Perl je jezik slobodne forme, odnosno prazni znakovi mogu se slobodno koristiti. Komentari, kao i kod ljuske, označavaju se znakom `"#"` i završavaju krajem retka. S obzirom na slobodnu formu, program u Perlu može izgledati i ovako:

```
#!/usr/bin/perl
   print    # Ovo je komentar
   "Hello, world!\n"
   ;    # Nemojte ovako pisati svoje programe :-)
```

3.2 Skalarni podaci

Najjednostavnija vrsta podataka u Perlu naziva se `skalarima`. U skalarne podatke ubrajamo brojeve i znakovne nizove, a Perl ih koristi na vrlo sličan način – kao pojedinačne, skalarne vrijednosti. Na skalarne vrijednosti mogu se primijeniti operatori (zbrajanje, ulančavanje i sl.), pri čemu se obično dobija skalarna vrijednost, a takva se vrijednost može pohraniti u skalarnu varijablu.

3.2.1 Brojevi

Perl sve brojeve interno pohranjuje u istom formatu, neovisno radi li se o cijelim (`255, 2008`) ili realnim brojevima (`3.14159`). Interno, Perl računa s brojevima u zapisu s pomičnim zarezom dvostruke točnosti, pri čemu točna preciznost ovisi o prevoditelju kojim je Perl preveden, odnosno o računalu na kojem se izvodi.

Vrijednosti zapisane u izvornom kôdu, konstante, nazivamo još i *literalima*. Numerički literali u Perlu uglavnom odgovaraju onima u ostalim programskim jezicima. U nastavku navodimo nekoliko brojeva s pomičnom točkom, kao i cijelih brojeva.

```
1.25    -12e-24    255.000    -1.2E-23          7.25e45
0       2008       -40        61298040283768    61_298_040_283_768
```

Možemo primijetiti da cijeli brojevi u Perlu mogu poprimiti vrlo velike vrijednosti. Zadnji primjer ilustrira jednu neobičnost u zapisu: u broj je moguće umetati znak podvlaka ("_"), čime se omogućuje grupiranje znamenaka u cilju bolje preglednosti.

Uobičajeni zapis brojeva je u dekadskom brojevnom sustavu, no literali se mogu zapisati i kao nedekadski, što se označava prefiksima. Oktalni literali započinju vodećom nulom :

`0377` = 255_{10}, heksadekadski započinju prefiksom `0x` i mogu se koristiti i znamenke `A-F` ili `a-f`: `0xff` = 255_{10}, dok binarni literali započinju s `0b`: `0b11111111` = 255_{10}.

Perl koristi uobičajene aritmetičke operatore (`+, -, *, /,...`). Treba voditi računa o tome da se operator dijeljenja izvodi u aritmetici s pomičnom točkom, pa rezultat dijeljenja dva cijela broja može biti necjelobrojan (`10 / 3 = 3.33333333333333`). Operator `%` označava operaciju modulo, odnosno ostatak pri dijeljenju, pri čemu se operandi reduciraju na cjelobrojne (`10.5 % 3.2 = 1`). Operatorom `**` označava se potenciranje (`2**3 = 8`)

3.2.2 Znakovni nizovi

Znakovni niz u Perlu je niz *proizvoljnih* znakova (*engl. string*), što omogućuje i baratanje sa "sirovim" binarnim podacima. Za razliku od, primjerice, C-a, znakovni nizovi u Perlu *ne zaključuju se* nul-znakom ("`\0`"), već `perl` interno prati duljinu znakovnog niza.

Navodnici i posebni znakovi

U literalima, znakovni nizovi uokviruju se *jednostrukim* ili *dvostrukim* navodnicima. Slično kao i kod ljuske `bash`, različiti navodnici uzrokuju različito ponašanje pri interpretaciji niza. U znakovnom nizu unutar jednostrukih navodnika:

- svaki znak osim jednostrukog navodnika ("`'`") ili kose crte ("`\`"), uključujući prelaske u novi red, predstavlja sam sebe;
- kosa crta uključuje se u niz kao "`\\`", jednostruki navodnik može se uključiti kao "`\'`";
- *ne interpretiraju se* posebne sekvence poput prelaska u novi red ("`\n`");
- ne obavlja se zamjena (interpolacija) varijabli u niz.

Za znakovni niz u dvostrukim navodnicima vrijedi:

- obavlja se interpolacija varijabli (zamjena imena vrijednošću);
- kosa crta ("`\`") koristi se za označavanje posebnih znakova (*engl. backslash escapes*) poput "`\n`". Značenja nekih od posebnih znakova navedena su u sljedećoj tablici.

`\n`	novi red
`\t`	tabulator
`\007`	oktalna ASCII vrijednost
`\x7f`	heksadekadska ASCII vrijednost
`\l`	sljedeći znak prebaci u malo slovo
`\L`	prebaci u mala slova sve znakove do `\E`
`\u`	sljedeći znak prebaci u veliko slovo
`\U`	prebaci u velika slova sve znakove do `\E`
`\E`	završetak slijeda nakon `\L` i `\U`

Operatori nad znakovnim nizovima

Perl definira dva operatora nad znakovnim nizovima. To su nadovezivanje (operator "`.`") i ponavljanje (operator "`x`). U nastavku je nekoliko primjera izraza u kojima se koriste ovi operatori, pri čemu je rezultat izraza prikazan u obliku komentara.

```
"hello" . "world"       #  "helloworld"
"hello" . ' ' . "world" #  'hello world'
'hello world' . "\n"    #  "hello world\n"
```

```
"fred" x 3              # => "fredfredfred"
"barney" x (3+1)        # "barney" x 4
                        # => "barneybarneybarneybarney"
5 x 4.2                 # zapravo "5" x 4 => "5555"
```

Zadnji primjer ilustrira *implicitnu* pretvorbu tipova koju Perl obavlja pri evaluaciji izraza. Kako je ”x” operator koji kao lijevi operand očekuje znakovni niz, broj koji se našao (slučajno ili namjerno) na lijevoj strani automatski se pretvara u znakovni niz. Osim toga, ukoliko je zadani broj ponavljanja necijeli, on se reducira na cijeli.

3.2.3 Automatska pretvorba između brojeva i nizova

Ovisno o *operatoru* koji se koristi, Perl obavlja pretvorbu brojeva u nizove znakova ili obrnuto. Ako se nad skalarnom vrijednošću primjenuje numerički operator (+, -, *,...) vrijednost se koristi kao numerička, a za operatore koji su definirani za znakovne nizove (`.`, `x`), brojevi se koriste kao nizovi znakova.

Pretvorba brojeva u nizove je uvijek moguća i jednostavna. Na primjer, izraz: `"Z" . 35` daje vrijednost `"Z35"`. U obrnutom slučaju, znakovni nizovi se pretvaraju u brojeve. Primjerice, izraz: `"12" * "3"` daje vrijednost `36`. U nekim je slučajevima ta pretvorba primjerena, jer znakovni niz uistinu sadrži broj, no Perl će u broj pretvoriti svaki niz koji se nađe na mjestu koje zahtijeva pretvorbu. Pritom se koristi jednostavno pravilo: vodeće praznine se zanemaruju, a odbacuje se i dio niza od prvog znaka koji se ne može interpretirati kao numerički. Primjerice, izraz: `"   12fred34" * " 3"` evaluirat će se u `36`. Niz koji nakon ovog odsijecanja ostane prazan, pretvara se u vrijednost `0`. Primjerice, izraz: `"  abc"*3` daje rezultat `0`.

S obzirom da se ovakvi slučajevi zapravo ne bi trebali događati, oni najčešće nastaju zbog pogreške programera, pa ”potiha” pretvorba ne ide uvijek na ruku programeru. Perl nudi mogućnost prijave ovakvih sumnjivih pretvorbi u obliku upozorenja (*engl. warning*). Kako bismo uključili upozorenja, pri pozivu Perl interpretera trebamo navesti opciju `-w`.

```
$ perl -w  moj_program
```

Ako tu opciju želimo ugraditi u svoj program, možemo je dodati u prvom retku programa, pri navođenju interpretera skripte.

```
#!/usr/bin/perl -w
```

Više detalja o pojedinim vrstama upozorenja možete naći u Perl dokumentaciji, primjerice korištenjem naredbe `man`.

```
$ man perldiag
```

3.2.4 Skalarne varijable

Imena skalarnih varijabli započinju znakom ”`$`”, a slijedi *Perl identifikator*, koji se sastoji od niza slova, znamenaka i podvlaka (”_”), pri čemu prvi znak ne može biti znamenka. Velika i mala slova se razlikuju, a svi su znakovi značajni:

`$Count` i `$count` su različite varijable

`$a_very_long_variable_that_ends_in_1` i

`$a_very_long_variable_that_ends_in_2` su različite varijable.

Ako napravimo usporedbu s ljuskom `bash`, sjetit ćemo se da su i tamo imena varijabli bila predznačena znakom ”$”, ali samo kada se varijabla koristila, dok se pri dodjeli vrijednosti prefiks izostavlja. U Perlu se skalarne varijable u Perlu *uvijek* referenciraju predznačene s ”$”!

Skalarno dodjeljivanje vrijednosti

Dodjela vrijednosti u Perlu ne razlikuje se od dodjele u većini programskih jezika. Na lijevoj strani znaka jednakosti navodi se ime varijable (predznačeno znakom ”$”), a na desnoj strani se nalazi izraz čija se izračunata vrijednost pohranjuje u varijablu.

```
$fred = 17;
$barney = 'hello';
$barney = $fred + 3;
$barney = $barney * 2;
```

Izrazi u kojima se ista varijabla pojavljuje i na lijevoj i na desnoj strani su vrlo česti, pa Perl poput jezika C i Java uvodi kraći zapis:

```
$fred += 5;
$barney *= 2;
$str .= " ";  # vrijedi i za ulančavanje
$x **= 3;  # x=x^3
```

Ispis pomoću operatora print

Operator `print` ispisuje skalarnu vrijednost navedenu kao argument na standardni izlaz. Pritom nije nužno pisati zagrade oko liste argumenata.

```
print "hello world\n";
print 6 * 7;
print (".\n");
```

Operator `print` prihvaća listu argumenata, koje ispisuje jednog po jednog. Argumenti se odvajaju zarezima.

```
print "Rezultat je ", 6 * 7, ".\n";
```

Interpolacija skalarnih varijabli u nizovima

Kada je znakovni niz naveden u dvostrukim navodnicima, podložan je *interpolaciji varijabli*, pri čemu se ime varijable navedeno u znakovnom nizu zamjenjuje njenom vrijednošću.

```
$meal = "brontosaurus steak";}
$barney = "fred ate a $meal";
$barney = 'fred ate a ' . $meal; # isti rezultat
```

Ako skalarnoj varijabli nije dodijeljena vrijednost (odnosno vrijednost joj *nije definirana*), na njeno mjesto interpolira se *prazni niz.* Ako su uključena upozorenja, Perl će se na takvu situaciju ”požaliti”.

Da bismo u niz uključili znak ”$” u doslovnom značenju, možemo ga predznačiti silaznom kosom crtom kako se ne bi obavila interpolacija varijabli. Sjećamo se i da se interpolacija ne obavlja unutar jednostrukih navodnika.

```
$fred = 'hello';
print "Ime je \$fred.\n";    # ispisuje znak $
print 'Ime je $fred' . "\n"; # isto
```

Kao ime varijable upotrijebit će se *najdulji* mogući podniz koji ima smisla, a to u slučaju gniježđenja varijable u znakovnom nizu može dovesti i do problema zbog spajanja imena sa znakovima koji slijede. To možemo izbjeći zatvaranjem imena varijable u vitičaste zagrade ili dijeljenjem niza na poddijelove i njihovim nadovezivanjem.

```
$what = "brontosaurus steak";
$n = 3;
print "fred ate $n $whats.\n";   # varijabla je $whats
print "fred ate $n ${what}s.\n"; # sad je ime $what
print "fred ate $n $what" . "s.\n"; # može i ovako
```

3.2.5 Operatori i redoslijed primjene

Većina operatora u Perlu potječe iz C-a, pa za njih vrijede i jednaki redoslijedi primjene. U nastavku navodimo tablicu operatora, u kojoj prednost operatora opada prema dnu tablice.

asocijativnost	operatori
lijeva	zagrade i argumenti operatora listi
lijeva	`->`
	`++ --`
desna	`**`
desna	`\ ! ~ + -` (unarni operatori)
lijeva	`=~ !~`
lijeva	`* / % x`
lijeva	`+ - .` (binarni operatori)
lijeva	`<< >>`
	`< <= > >= lt le gt ge`
	`== != <=> eq ne cmp`
lijeva	`&`
lijeva	`| ^`
lijeva	`&&`
lijeva	`||`
lijeva	`.. ...`
desna	`?:` (ternarni operator)
desna	`= += -= .=` (i slični operatori pridruživanja)
lijeva	`, =>`
desna	`not`
lijeva	`and`
lijeva	`or xor`

Redoslijed primjene kod operatora iste razine razrješava se pravilima asocijativnosti:

```
4 ** 3 ** 2     # 4 ** (3 ** 2) = 4 ** 9 (desna asoc.)
72 / 12 / 3     # (72 / 12) / 3 = 6/3 = 2 (l. asoc.)}
36 / 6 * 3      # (36/6)*3 = 18
```

U slučaju nedoumice o redoslijedu primjene operatora dobro je koristiti zagrade

3.2.6 Operatori usporedbe

Za usporedbu brojeva koriste se operatori `< <= == >= > !=`, koji vraćaju vrijednost *true* ili *false*. Za usporedbu znakovnih nizova Perl ima poseban skup operatora: `lt le eq ge gt ne`. Njima

se ispituje jesu li dva znakovna niza jednaka, odnosno koji od njih je prije u standardnom redoslijedu sortiranja. U ASCII redoslijedu velika slova su ispred malih.

usporedba	brojevi	znakovni nizovi
jednako	==	eq
različito	!=	ne
manje od	<	lt
veće od	>	gt
manje ili jednako	<=	le
veće ili jednako	>=	ge

Podsjetimo se i da su operatori usporedbe kod ljuske bash bili (skoro) obrnuti. Tamo su se uobičajeni operatori koristili za znakovne nizove, a slovčani (i to s vodećim znakom ”-”) za usporedbu brojeva! Ponašanje operatora usporedbe možemo ilustrirati na nekoliko primjera.

```
35 != 30 + 5            # false
35 == 35.0              # true
'35' eq '35.0'          # false (usporedba nizova)
'fred' lt 'barney'      # false
'fred' lt 'free'        # true
'fred' eq "fred"        # true
'fred' eq 'Fred'        # false
' ' gt ''               # true
```

3.2.7 Ispitivanje uvjeta

Za ispitivanje uvjeta i usmjeravanje programskog toka koristi se naredba if.

```
if ($name gt 'fred') {
  print "'$name' se sortira nakon 'fred'.\n";
}
```

Ovdje valja napomenuti da se, za razliku od C-a, vitičaste zagrade zahtijevaju sintaksom čak i u slučaju da se if blok sastoji od samo jedne naredbe!

Naredba if može imati i blok else, koji se izvršava ako uvjet nije ispunjen.

```
    if ($name gt 'fred') {
       print "'$name' se sortira nakon 'fred' .\n";
     }
    else {
       print "'$name' se ne sortira nakon 'fred'.\n";
       print "a mozda je to isti niz.\n";
     }
```

Logičke vrijednosti

Pri ispitivanju uvjeta može se koristiti bilo koja skalarna vrijednost. To je prikladno ako želimo pohraniti vrijednost istinitosti u varijablu:

```
$is_bigger = $name gt 'fred';
if ($is_bigger) { ... }
```

Perl nema poseban logički tip podatka (*engl. boolean*), već se koristi nekoliko jednostavnih pravila:

- nedefinirana vrijednost odgovara logičkoj vrijednosti *false*;
- nula odgovara logičkoj vrijednosti *false*, ostali brojevi su *true*;
- prazan niz (`''`) je *false*, ostali nizovi su *true*. Iznimka je niz `'0'` čija vrijednost istinitosti je *false*;

Negacija logičkog izraza postiže se operatorom "`!`": `if (! $is_bigger) { ... }`

3.2.8 Učitavanje podataka

Za učitavanje sa standardnog ulaza koristi se operator `<STDIN>`. Kada se `<STDIN>` upotrijebi na mjestu gdje se očekuje skalarna vrijednost, Perl učitava cijeli redak teksta sa standardnog ulaza, sve do prve oznake novog reda. Tipično, vrijednost dobivena operatorom `<STDIN>` završava znakom "`\n`"

```
$line = <STDIN>;
if ($line eq "\n") {
  print "To je samo prazni redak!\n";
 }
else {
  print "Ucitani redak je: $line";
 }
```

U praksi ne želimo zadržati "`\n`" na kraju učitanog znakovnog niza, pa ga možemo ukloniti primjenom operatora `chomp`.

```
$text = "redak teksta\n"; # ili niz ucitan sa <STDIN>
chomp($text);             # skida  \n
```

Često se koristi kraći zapis, koji objedinjuje učitavanje retka i skidanje oznake prelaska u novi red:

```
chomp($text = <STDIN>); # ucitavanje teksta bez \n
```

3.2.9 Petlja while

Perl ima standardnu naredbu `while` kojom se ostvaruje programska petlja koja se ponavlja dok god je zadani uvjet ispunjen.

```
$count = 0;
while ($count < 10) {
  $count += 1;
  print "broj je sada $count\n";
                  # ispisuje brojeve 1 do 10
}
```

Uvjet se ponaša kao i u naredbi `if`, a i ovdje su vitičaste zagrade koje omeđuju blok naredbi *obavezne*. Uvjet se ispituje *prije* prvog ulaska u petlju, pa petlja može biti preskočena ako uvjet od početka nije ispunjen.

3.2.10 Nedefinirana vrijednost

Prije nego se varijabli prvi put dodijeli vrijednost, ona ima posebnu vrijednost `undef`. Ako se takva varijabla upotrijebi na mjestu gdje se očekuje znakovni niz, ona se ponaša kao prazni

niz. Ako se nedefinirana varijabla upotrijebi na mjestu broja, njena je vrijednost 0, što se može upotrijebiti umjesto inicijalizacije kao u sljedećem primjeru.

```
# zbrajanje neparnih brojeva
$n = 1;
while ($n < 10) {
  $sum += $n;
  $n += 2; # sljedeci neparni broj
}
print "Suma je $sum.\n";
```

Na sličan način mogu se koristiti i neinicijalizirani znakovni nizovi:

```
$string .= "more text\n";
```

No, iako ovakav način korištenja varijabli štedi pisanje dodatne naredbe za inicijalizaciju, njime se narušava čitljivost i razumljivost kôda. Stoga se ipak preporučuje eksplicitno inicijalizirati varijable.

Ako želimo ispitati je li neka varijabla definirana ili ne, možemo primijeniti funkciju `defined`. Primjerice, operator `<STDIN>` može vratiti vrijednost `undef` ako se upis retka prekine oznakom kraja datoteke (`EOF`, `ctrl-D`).

```
$redak = <STDIN>;
if ( defined($redak) ) {
  print "Ucitani redak je: $redak";
 }
else {
  print "Redak nije ucitan!\n";
 }
```

3.3 Liste i polja

Lista je uređeni niz skalarnih vrijednosti, a polje je varijabla koja sadrži listu. U Perlu se ta dva pojma koriste kao istoznačni, no ako želimo biti precizni – lista je skup podataka, a polje je varijabla. Lista ne mora biti spremljena u polje, ali polje uvijek sadrži listu (koja može biti i prazna). Svaki element polja ili liste je odvojena skalarna varijabla s nezavisnom vrijednošću, odnosno lista može sadržavati brojeve, znakovne nizove ili mješavinu različitih skalarnih vrijednosti. Elementi liste su uređeni (poredani) i indeksirani slijednim cijelim brojevima, s tim da prvi element liste ima indeks 0, a broj elemenata je ograničen samo raspoloživim memorijskim prostorom.

```
$fred[0] = "yabba";
$fred[1] = "dabba";
$fred[2] = "doo";
```

Ime polja nalazi se u potpuno odvojenom *prostoru imena* (*engl. namespace*) u odnosu na skalarne varijable, pa možemo imati skalarnu varijablu jednakog imena (u ovom slučaju `$fred`). Sintaksa je uvijek nedvosmislena, mada ponekad zbunjujuća. Svaki element polja može se koristiti gdje god se može koristiti skalarna varijabla, uz poneki izuzetak, primjerice upravljačka varijabla petlje `foreach` mora biti jednostavan skalar.

Indeksiranje polja

Indeks može biti bilo koji izraz koji daje numeričku vrijednost. Ako vrijednost nije cjelobrojna, reducira se na cjelobrojnu.

```
$broj = 2.71828;
print $fred[$broj - 1]; # isto kao print $fred[1]
```

Ako se pokuša čitati "element" iza kraja polja, njegova će vrijednost biti `undef`, a ako se pohrani vrijednost u element polja iza njegovog kraja, polje se automatski proširuje, i to bez ograničenja osim raspoloživog memorijskog prostora. Prazni elementi polja koji pritom mogu nastati imaju vrijednost `undef`.

```
$polje[0] = 'nula';         # prvi element
$polje[1] = 'jedan';        # drugi element
$polje[33] = 'tri_tri';     # nastaje 31 undef element
```

Ponekad je potrebno saznati indeks posljednjeg elementa u polju. U Perl-u se taj indeks može dobiti izrazom `$#polje`. Treba paziti na to da se radi o indeksu posljednjeg elementa a ne o broju elemenata. Manipulacijom ove vrijednosti mijenja se i veličina polja.

```
$end = $#polje;              # u ovom slučaju 33
$broj_elemenata = $end + 1;
$#polje = 5;                 # polje se smanjilo,
                             # ostali elementi su izgubljeni
```

Korištenje posljednjeg indeksa u polju je često, pa je u Perl uvedena pokrata. Korištenjem negativnih indeksa postiže se odbrojavanje elemenata od kraja polja. Tako indeks `-1` označava posljednji element, `-2` predposljednji itd. Prekoračenje veličine negativnim indeksom generira pogrešku.

```
$polje[ -1 ] = 'zadnji';     # posljednji element
$prvi = $polje[-6];          # 0-ti element u polju od 6 elemenata
$polje[ -50 ] = 'podbacaj';  # fatal error!
```

Liste kao literali

Liste se kao vrijednosti u programu (literali) navode u obliku niza vrijednosti odvojenih zarezima, unutar oblih zagrada.

```
(1, 2, 3)
(1, 2, 3,)          # ista lista, zarez na kraju se zanemaruje
("fred", 4.5)
( )                 # prazna lista - 0 elemenata
(1..5)              # isto kao (1, 2, 3, 4, 5) - range operator
(1.7..5.7)          # ista lista - int
(5..1)              # prazna lista - nema odbrojavanja unazad
(0, 2..6, 10, 12)   # lista (0, 2, 3, 4, 5, 6, 10, 12)
```

Elementi liste ne moraju nužno biti konstante, već mogu biti i izrazi koji se evaluiraju svaki put kada se literal koristi.

```
($a..$b)          # raspon odredjuju vrijednosti $a i $b
(0..$#polje)      # lista svih indeksa polja
($a, 17)          # dva elementa: vrijednost $a i 17
($b+$c, $d+$e)    # dva elementa
```

Kratica qw

Liste riječi su vrlo česte, pri čemu se svaka riječ navodi unutar navodnika. Kratica qw omogućuje zapisivanje liste riječi bez potrebe za brojnim navodnicima (*engl. quoted words*).

```
("fred", "barney", "betty", "wilma", "dino")  # lista riječi
qw/ fred barney betty wilma dino /            # ista lista
```

Riječi se koriste kao da su u *jednostrukim* navodnicima, pa nema tumačenja specijalnih sekvenci ni interpolacije varijabli. Praznine između riječi (nizovi razmaka, tabulatori i oznake novog reda) se zanemaruju.

```
qw/fred
  barney     betty
wilma dino/ # ista lista, neobičan stil
```

U prethodnim primjerima se kao graničnik koristi znak "/", no Perl dozvoljava izbor proizvoljnog znaka interpunkcije kao graničnika. U slučaju korištenja zagrada, listu zatvara druga istovrsna zagrada, a u slučaju samostalnih znakova interpunkcije, kraj liste se označava istim graničnikom. Graničnik se može koristiti unutar liste ako ga predznačimo znakom "\".

```
qw! fred barney betty wilma dino !
qw# fred barney betty wilma dino #
qw( fred barney betty wilma dino )
qw{ fred barney betty wilma dino }
qw[ fred barney betty wilma dino ]
qw< fred barney betty wilma dino >
qw! yahoo\! google excite lycos ! # \ escape
```

Ta mogućnost promjene graničnika korisna je primjerice ako želimo zapisati listu Unix datoteka.

```
qw{
  /usr/dict/words
  /home/rootbeer/.ispell_english
}
```

3.3.1 Pridruživanje vrijednosti listi

Pridruživanje vrijednosti listama slično je pridruživanju vrijednosti varijablama.

```
($fred, $barney, $dino) = ("jedan", "dva", undef);
```

Varijable u listi s lijeve strane znaka pridruživanja dobivaju nove vrijednosti kao u 3 odvojena pridruživanja. To nam omogućuje i jednostavnu zamjenu vrijednosti dviju varijabli, bez potrebe za korištenjem dodatne pomoćne varijable.

```
($fred, $barney) = ($barney, $fred);  # swap
```

Broj varijabli u listama s lijeve i desne strane znaka jednakosti ne mora uvijek biti jednal. Ako lista na desnoj strani ima višak članova, oni se zanemaruju. Ako je pak na lijevoj strani višak varijabli u listi, prekobrojne dobivaju vrijednost undef.

```
($fred, $barney) = qw< jedan dva tri cetiri >; # višak vrijednosti
($wilma, $dino) = qw[flintstone];              # $dino je undef
```

3.3.2 Referenciranje polja

Za referenciranje cijelog polja koristi se jednostavnija notacija. Ime polja predznači se znakom "@" i ne koriste se indeksne zagrade.

```
@polje = qw/ jedan dva tri /;
@tiny = ( );              # prazna lista
@giant = 1..1e5;          # lista od 100.000 elemenata
@stuff = (@giant, undef, @giant); # 200.001 element
```

Polje može sadržavati samo skalare, a ne i polja[1]. Ime polja *zamjenjuje se* listom koju sadrži.

```
@polje = (1,2,3);
@novoPolje = (0, @polje, 4);      # rezultat je (0, 1, 2, 3, 4)
```

Polje kojem još nisu dodijeljene vrijednosti sadrži praznu listu. Kada se polju pridruži vrijednost drugog polja, polje se zapravo kopira.

```
@kopija = @polje; # polje se kopira
```

Operatori za dodavanje i skidanje rubnih elemenata polja

Vidjeli smo da Perl svakim upisivanjem elementa s indeksom izvan polja proširuje polje. Novi elementi mogu se dodavati na kraj polja korištenjem sve većih indeksa, no Perl nudi i alternativne načine za rad s poljima – bez korištenja indeksa. Radi se o operatorima kojima se može dodati ili obrisati element na kraju ili početku polja. Takvim pristupom obično se dobiva brži rad programa, a ujedno se umanjuje mogućnost pogrešaka u pristupu elementima.

Često nam je potrebna stožna struktura podataka (LIFO), a ona se jednostavno ostvaruje operatorima push i pop. Operator pop uzima zadnji element iz polja.

```
@polje = 5..9;          # range operator
$fred = pop(@polje);    # $fred je 9, @polje je (5, 6, 7, 8)
$barney = pop @polje;   # $barney je 8, @polje je (5, 6, 7)
pop @polje;             # skida se 7, @polje je (5, 6)
```

Ako je polje prazno, pop ne utječe na njega, a vraća vrijednost undef.

Suprotna operacija je push, koja element ili listu elemenata koje dobije kao argumente dodaje na kraj polja.

```
push(@polje, 0); # @polje je (5, 6, 0)
push @polje, 8;  # @polje je (5, 6, 0, 8)
push @polje, 1..10; # dodaje se jos 10 elemenata
@others = qw/ 9 0 2 1 0 /;
push @polje, @others; # jos 5 -> ukupno 19
```

Prvi argument za push i argument za pop moraju biti varijable, što je i logično jer dodavanje ili skidanje elementa iz literala nema smisla.

Na sličan način rade i operatori shift i unshift, samo oni barataju s početkom polja. Operator shift skida prvi element polja, dok unshift upisuje listu elemenata na početak polja.

```
@polje = qw# dino fred barney #;
$a = shift(@polje);
       # $a je "dino", @polje je ("fred", "barney")
$b = shift @polje;
```

[1]Istina, u polju se mogu zapisati *reference* na polja i na taj način se može ostvariti logička struktura dvodimenzionalnog polja, no i reference su skalari. Referencama u Parlu nećemo se baviti u okviru ovog kolegija.

```
      # $b je "fred", @polje je ("barney")
shift @polje;       # @polje je prazno, ()
$c = shift @polje;  # $c je undef, @polje je ()
unshift(@polje, 5); # @polje je (5)
unshift @polje, 4;  # @polje je (4, 5);
@others = 1..3;
unshift @polje, @others; # @polje je (1, 2, 3, 4, 5)
```

Kao i pop, `shift` primijenjen na prazno polje vraća vrijednost `undef`.

3.3.3 Interpolacija polja

Kao i skalarne vrijednosti, vrijednosti polja mogu se *interpolirati* u znakovne nizove unutar dvostrukih navodnika. Pritom se elementi polja automatski razdvajaju prazninama. Zapravo je separator vrijednost specijalne varijable `$"`, čija je uobičajena vrijednost `'␣'`). Na početku i kraju polja ne umeću se praznine, za to se programer mora pobrinuti prema potrebi.

```
@rocks = qw{ flintstone slate rubble };
print "quartz @rocks limestone\n";

print "Three rocks are: @rocks.\n";
print "Zagrade (@empty) su prazne.\n";
```

Kako se u imenima polja pojavljuje prefiks `@`, potrebno je pripaziti na na e-mail adrese u znakovnim nizovima, jer bi ih Perl mogao protumačiti kao imena polja koja treba interpolirati.

```
$email = "fred@bedrock.edu";  #  "fred.edu"
     # interpolacija    --> @bedrock se zamjenjuje praznim nizom
$email = "fred\@bedrock.edu"; # OK - predznačavanje
$email = 'fred@bedrock.edu';  # OK - bez interpolacije
```

Pojedinačni element polja u znakovnom nizu zamjenjuje se svojom vrijednošću.

```
@fred = qw(hello dolly);
$y = 2;
$x = "This is $fred[1]'s place";          # "This is dolly's place"
$x = "This is $fred[$y-1]'s place";  # isto
```

3.3.4 Petlja foreach

Perl nudi upravljačku strukturu prikladnu za obradu cijelog polja ili liste. Naredbom `foreach` oblikuje se petlja koja prolazi kroz sve vrijednosti u listi i izvršava blok naredbi za svaku od vrijednosti.

```
foreach $rock (qw/ bedrock slate lava /) {
  print "One rock is $rock.\n";
        # Prints names of three rocks
}
```

Upravljačka varijabla (ovdje `$rock`) u svakoj iteraciji preuzima novu vrijednost iz liste. To *nije kopija* elementa liste, već se preko nje barata samim elementom. Ako se unutar petlje mijenja upravljačka varijabla, mijenja se element liste

```
@rocks = qw/ bedrock slate lava /;
foreach $rock (@rocks) {
  $rock = "\t$rock"; # tab prije svake rijeci
```

```
  $rock .= "\n"; # novi red iza rijeci
}
print "The rocks are:\n", @rocks;
```

Perl brine o tome da vrijednost upravljačke varijable po završetku petlje bude *ista* kao i prije ulaska u petlju.

Podrazumijevana varijabla

Ako se izostavi upravljačka varijabla na početku petlje, Perl koristi podrazumijevanu (*engl. default*) varijablu `$_`. To je mehanizam koji sada prvi put susrećemo, ali u Perlu se često koristi. Na taj se način može skratiti dijelove programa, a to je bila jedna od vodilja autoru jezika. Stoga ćemo je još susretati.

```
foreach (1..10) {      # koristi se $_ by default
   print "I can count to $_\n";
}
```

Još jedan jednostavan primjer korištenja podrazumijevane varijable je ponašanje operatora `print` kada mu se ne navede argument.

```
$_ = "Yabba dabba doo\n";
print;                # ispisuje se $_ by default
```

3.3.5 Izmjena redoslijeda elemenata liste

Operator reverse

Operator `reverse` preuzima listu vrijednosti i vraća listu poredanu obrnutim redom.

```
@fred = 6..10;
@barney = reverse(@fred); # 10, 9, 8, 7, 6
@wilma = reverse 6..10;   # isto, ali bez polja
@fred = reverse @fred;    # rezultat se vraca u isto polje
```

Operator `reverse` ne utječe na svoj argument (listu), pa ako se rezultat ne pohrani u neku varijablu, gubi se!

```
reverse @fred;  # pogresno - @fred se ne mijenja
```

Operator sort

Operator `sort` preuzima listu vrijednosti i vraća listu poredanu prema internom redoslijedu sortiranja. Obično se radi o ASCII redoslijedu, gdje velika slova dolaze prije malih, a znamenke prije slova. Ovaj se redoslijed može i izmijeniti, no time se sada nećemo baviti.

```
@rocks = qw/ bedrock slate granite /;
@sorted = sort(@rocks);         # bedrock, granite, slate
@back = reverse sort @rocks;            # slate, granite, bedrock

@rocks = sort @rocks;
@numbers = sort 97..102;        # 100, 101, 102, 97, 98, 99
```

Operator `sort` ne utječe na svoj argument (listu), pa se rezultat gubi ako se ne pohrani u neku varijablu.

```
sort @rocks; # pogresno - @rocks se ne mijenja
```

3.3.6 Skalari i liste ovisno o kontekstu

Perl tumači izraz ovisno o kontekstu u kojem se nalazi. Primjerice, operand u aritmetičkom izrazu mora biti skalar, a sortirati možemo samo listu.

Kada se ime polja nađe u kontekstu u kojem se očekuje lista, zamjenjuje se elementima polja, dok se u skalarnom kontekstu zamjenjuje *brojem elemenata.*

```
@people = qw( fred barney betty );
@sorted = sort @people; # lista: barney, betty, fred
$number = 42 + @people; # skalar: 42 + 3 = 45
```

Dodjeljivanje vrijednosti također stvara kontekst.

```
@list = @osobe;     # lista s 5 osoba
$n = @osobe;        # broj 5
```

Razni izrazi mogu se koristiti za generiranje listi, no pitanje je što ćemo dobiti ako te izraze upotrijebimo u skalarnom kontekstu. U nekim slučajevima rezultat nije sasvim očekivan.

Tako neki izrazi u skalarnom kontekstu nemaju vrijednost. Primjerice, operator `sort` u skalarnom kontekstu vraća vrijednost `undef`, iako bismo očekivali da vrati broj elemenata liste. S druge strane, zašto bismo sortirali listu da bismo dobili broj elemenata liste?

Operator `reverse` u kontekstu liste vraća elemente liste obrnutim redom, dok u skalarnom kontekstu vraća niz znakova koji se dobije promjenom redoslijeda znakova dobivenih ulančavanjem svih elemenata liste. Opet nešto što vjerojatno nikada ne bismo pogodili.

```
@backwards = reverse qw/ yabba dabba doo /;
             # --> doo, dabba, yabba
$backwards = reverse qw/ yabba dabba doo /;
             # --> oodabbadabbay
```

Još nekoliko primjera uobičajenih konteksta.

```
$fred = something;            # skalarni kontekst
@pebbles = something;         # contekst liste
($wilma, $betty) = something; # contekst liste
($dino) = something;          # contekst liste iako je 1 element
```

Još neke situacije u kojima se stvara skalarni kontekst.

```
$fred[3] = something;
123 + something
something + 654
if (something) { ... }
while (something) { ... }
$fred[something] = something;
```

Još neki primjeri konteksta liste.

```
push @fred, something;
foreach $fred (something) { ... }
sort something
reverse something
print something
```

Kada se skalar nađe u kontekstu liste, pretvorba je vrlo jednostavna. Njegova se vrijednost naprosto pretvara u listu se jednim elementom.

```
@fred = 6 * 7; # lista (42)
@barney = "hello" . ' ' . "world";
          # lista ("hello world")
@wilma = undef; # lista (undef), nije isto kao
@betty = ( );    # prazna lista - polje se brise
```

Forsiranje skalarnog konteksta

U nekim slučajevima želimo skalarni kontekst na mjestu gdje Perl očekuje listu. Tada možemo upotrijebiti funkciju `scalar`. To zapravo nije prava funkcija, već samo daje uputu Perlu da upotrijebi skalarni kontekst. Obrnute funkcije, za forsiranje konteksta liste, nema.

```
@rocks = qw( talc quartz jade obsidian );
print "How many rocks do you have?\n";
print "I have ", @rocks, " rocks.\n";   # ispisuje listu a ne broj
print "I have ", scalar @rocks, " rocks.\n";    # daje broj
```

Operator <STDIN> u kontekstu liste

Operator `<STDIN>` smo upoznali u skalarnom kontekstu, gdje vraća redak ulaznih znakova. U kontekstu liste, `<STDIN>` vraća sve retke do kraja datoteke (`EOF`), pri čemu svaki redak postaje poseban element liste.

```
@lines = <STDIN>;  # čitanje stdin u kontekstu liste
```

Ovdje nije loše napomenuti da Unix i Windows ne koriste istu oznaku za kraj datoteke. Za Unix je to kombinacija tipki `ctrl-D`, dok se na Windowsima koristi `ctrl-Z`.

Kako svaki učitani redak završava oznakom prelaska u novi red, često je potrebno skinuti te oznake iz svakog retka, odnosno u ovom slučaju iz svakog elementa dobivene liste. Zato je vrlo prikladna mogućnost primjene operatora `chomp` na cijelo polje učitanih redaka.

```
@lines = <STDIN>;   # učitaj sve retke
chomp(@lines);

# ili još elegantnije
chomp(@lines = <STDIN>);
```

3.4 Potprogrami

Perl omogućuje pisanje korisnički definiranih potprograma. Imena potprograma su, kao i imena varijabli, Perl identifikatori. I za njih Perl rezervira odvojeni prostor imena, koji se postiže prefiksiranjem imena znakom `&`, koji se ne navodi uvijek.

Definicija potprograma započinje ključnom rječju `sub`, slijedi ime potprograma (*bez* znaka `&`), te blok naredbi uokviren vitičastim zagradama.

```
sub marine {
  $n += 1;  # globalna varijabla $n
  print "Hello, sailor number $n!\n";
}
```

Definicija funkcije može se nalaziti *bilo gdje* u programu, definicije su *globalne*. Moguća je i situacija da se u programu nađu dvije definicije potprograma s istim imenom. Tada kasnije definicije pregazi prvu. Zadaća je programera da se takve situacije ne događaju.

Potprogram se može pozvati u bilo kojem izrazu tako da se navede ime funkcije *uključujući* znak `&`.

```
&marine;  # Hello, sailor number 1
&marine;  # Hello, sailor number 2
&marine;  # Hello, sailor number 3
&marine;  # Hello, sailor number 4
```

Svi Perl potprogrami imaju povratnu vrijednost, čak i ako se ne koristi operator `return`. Vraća se *rezultat zadnjeg izraza* u potprogramu.

```
sub sum_of_fred_and_barney {
  print "Potprogram je pozvan\n";
  $fred + $barney;  # rezultat je povratna vrijednost
}

$fred = 3; $barney = 4;
$wilma = &sum_of_fred_and_barney; # $wilma = 7
print "\$wilma je $wilma.\n";
```

To je nešto o čemu treba dobro voditi računa, jer lako se može dogoditi da dodamo još neku naredbu u potprogram, iza izraza koji je trebao vratiti vrijednost.

```
sub sum_of_fred_and_barney {
  print "Potprogram je pozvan\n";
  $fred + $barney; # rezultat NIJE povratna vrijednost
  print "Sada vracam vrijednost\n"; # Uh! Ali ne pravu!
}
```

Zadnji *evaluirani* izraz više nije ono što smo namjeravali vratiti, već poziv funkcije `print`. Zato se vraća njena povratna vrijednost! Ovdje nam uključivanje upozorenja može pomoći, jer sumnjiva je situacija da rezultat izraza nije nigdje pohranjen.

3.4.1 Argumenti

Iako se podaci koje koristi potprogram mogu prenijeti preko globalnih varijabli, kao u prethodnim primjerima, to nije dobra praksa. Uobičajeno je podatke prenostiti potprogramu u obliku argumenata. U Perlu se pri pozivu potprograma može navesti lista argumenata, koji su u potprogramu dohvatljivi preko posebne varijable `@_`. Ta je varijabla polje, preko nje se može saznati broj argumenata i pristupiti svakom od njih. Tako se prvom elementu pristupa izrazom `$_[0]`, drugom s `$_[1]`, itd. S obzirom na egzotične oznake posebnih varijabli u Perlu, valja upozoriti na to da varijabla `@_`, odnosno njeni elementi, nema nikakve veze s podrazumijevanom varijablom `$_`.

Kao primjer, napišimo potprogram koji prima dva argumenta i vraća veći od njih.

```
sub max {
  if ($_[0] > $_[1]) {
    $_[0];
  } else {
    $_[1];
  }
}
```

Program bismo pozvali navođenjem argumenata u oblim zagradama.

```
$n = &max(11, 17);     # poziv funkcije s 2 argumenta
```

Privatne varijable

U potprogramu `max` uočavamo da je, zbog kriptičnog načina pristupu argumentima, kôd prilično nečitljiv. Tome možemo doskočiti tako da na početku potprograma kopiramo argumente u varijable smislenijih imena. Međutim, te varijable bi bile globalne, vidljive i u ostatku programa, što bi moglo dovesti do neželjenih kolizija u imenima. Stoga se varijable predviđene isključivo za korištenje unutar potprograma obično proglašavaju (deklariraju) textitprivatnima. To se postiže operatorom `my`.

Sada bi naš potprogram za određivanje veće od dviju vrijednosti mogao biti:

```
sub max {
  my($m, $n);       # nove, privatne varijable
  ($m, $n) = @_;    # dajemo imena argumentima
  if ($m > $n) { $m } else { $n }
}
```

Privatne (*engl. private, scoped*) varijable nisu ograničene samo na potprograme, već za njih općenito vrijedi da im je doseg ograničen na *blok* naredbi u kojem se nalaze. Primjerice, privatnu varijablu možemo uvesti unutar petlje i ona neće kolidirati s eventualnom istoimenom varijablom u drugim dijelovima programa.

```
foreach (1..10) {
   my($square) = $_ * $_;  # privatna varijabla
   print "$_ squared is $square.\n";
}
```

Operator `my` ne mijenja kontekst pridruživanja.

```
my($num) = @_;    # kontekst liste, kao ($num) = @_
                  # $num poprima vijednost prvog elementa
my $num  = @_;    # skalarni kontekst, kao $num = @_
                  # u $num se pohranjuje broj elemenata
```

Bez zagrada `my` se odnosi samo na jednu varijablu.

```
my $fred, $barney;        # deklarira se samo $fred
my($fred, $barney); # deklariraju se obje varijable
```

Lista argumenata promjenjive duljine

Kao što smo vidjeli, u Perlu se pri definiciji potprograma, za razliku od "tradicionalnih" programskih jezika, ne deklarira tip niti broj argumenata. To nam omogućava pisanje programa koji koriste listu argumenata proizvoljne duljine. Ova fleksibilnost može donijeti i probleme, ukoliko je broj argumenata pri pozivu različit od onoga koji autor potprograma očekuje. Stoga je uputno u programu provjeriti broj argumenata ispitivanjem polja `@_`.

```
sub max {
  if (@_ != 2) {   # skalarni kontekst daje broj elemenata
    print "&max mora imati 2 argumenta\n";
    # ostatak kao i prije ...
}
```

Još je bolje, ako to funkcionalnost potprograma dopušta, napisati potprogram tako da se prilagođava broju argumenata s kojima je pozvan. Naš program za određivanje najveće vrijednosti može se napisati tako da traži najveću vrijednost u proizvoljno dugoj listi argumenata.

```
#       poopćenje potprograma max
sub max {
  my($max_so_far) = shift @_;   # prvi je zasad najveći
  foreach (@_) {                # za svaki od sljedećih
    if ($_ > $max_so_far) {     # je li ovaj veći ?
      $max_so_far = $_;         # $_ je varijabla petlje
    }
  }
  $max_so_far;    # povratna vrijednost
}

# poziv funkcije
$maximum = &max(3, 5, 10, 4, 6);
```

Možemo se još i zapitati kako bi se ponašao naš novi potprogram ako se pozove bez argumenata. Kratka analiza pokazuje da je ponašanje primjereno. Naime, primjena operatora `shift` nad praznom listom dat će vrijednost `undef`, a u petlja se neće niti jednom izvršiti. Stoga se vraća vrijednost `undef`, što je sasvim primjeren rezultat.

Operator return

Operator `return` omogućuje trenutan povratak iz potprograma uz generiranje povratne vrijednosti koja se navodi kao argument.

```
@names = qw/ fred barney betty dino wilma /;
$result = &which_element_is("dino", @names);

sub which_element_is {
  my($what, @array) = @_;
  foreach (0..$#array) {  # svi indeksi polja @array
    if ($what eq $array[$_]) {
      return $_;  # kada se pronadje - povratak
    }
  }
  -1;  # povratna vrijednost - return nije potreban
}
```

Izostavljanje znaka &

U nekim situacijama znak `&` *može* izostaviti u pozivu potprograma, a to je definirano s nekoliko pravila. Prvi slučaj je kada je iz sintakse jasno da se radi o pozivu potprograma, dakle kada nakon imena slijedi lista parametara navedena unutar zagrada.

```
@cards = shuffle(@deck_of_cards);    # &shuffle
```

Znak `&` nije obavezan ni ako je definicija potprograma navedena prije njegovog pozivanja. Tada se čak mogu i izostaviti zagrade pri pozivu, jer interpreter zna da postoji potprogram tog imena.

```
sub podijeli {
  $_[0] / $_[1];
```

```
}

$kvocijent = podijeli 355, 113;  # &podijeli
```

Međutim, ako potprogram ima isto ime kao ugrađena funkcija, *moramo* pri pozivu upotrijebiti znak `&`, inače će biti pozvana *ugrađena funkcija*.

Iako Perl nudi ove mogućnosti za skraćivanje programa, trebalo bi ih izbjegavati. Naime, ušteda od nekoliko znakova u pravilu nije vrijedna gubitka čitljivosti programa. Svakako je bolje biti dosljedan, što u ovom slučaju znači – koristiti znak `&` u pozivima korisničkih potprograma.

Lista kao povratna vrijednost

Povratna vrijednost potprograma može biti i lista. Za ilustraciju, napišimo potprogram koji će generirati niz brojeva između dvije zadane vrijednosti, pri čemu niz može biti i silazan. Sjetimo se, operator za generiranje nizova brojeva ne može generirati silazan niz.

```
sub raspon {
  if (@_ != 2) {   # očekujemo 2 argumenta
    print "&raspon se zadaje s 2 argumenta\n";
    return;
  }

  my ($pocetak, $kraj) = ($_[0], $_[1]);

  if ($pocetak < $kraj) {
    # brojimo uzlazno od $pocetak do $kraj
     $pocetak..$kraj;
  } else {
    # brojimo unazad
        reverse $kraj..$pocetak;
  }
}

@rezultat = &raspon(13,7);
print("@rezultat\n");     #  13 12 11 10 9 8 7
```

3.5 Učitavanje pomoću operatora <>

Operator <> upoznali smo kod učitavanja sa standardnog ulaza (`<STDIN>`). On se općenito koristi za učitavanje iz datoteke čiji se identifikator navodi unutar zagrada.

Međutim poseban i vrlo čest način korištenja je bez navođenja datoteke. U tom se slučaju podaci učitavaju (redak po redak) iz datoteka čija su imena pri pozivu programa navedena kao argumenti naredbenog retka. Ako se pak ne navede niti jedno ime datoteke, podaci se učitavaju sa standardnog ulaza. Zapravo se ovim operatorom vrlo jednostavno i sažeto ostvaruje ponašanje po uzoru na standardne Unix alate. Kako bi se omogućilo miješano učitavanje (dio iz datoteke, a dio sa standardnog ulaza), omogućeno je i zadavanje argumenta `"-"` čime se označava `stdin`.

Kao primjer, napišimo kratak program koji samo učitava redak i ispisuje ga na standardni izlaz.

```
while (defined($redak = <>)) {
```

```
  chomp($redak);
  print "Procitao sam: $redak \n";
}
```

Ako ovaj program pokrenemo bez argumenata, čitaju se retci s tipkovnice. Ako se navede više imena datoteka, čitaju se retci iz svake od njih, redom navođenja, kao da se radi o jednoj datoteci. Ime datoteke iz koje se u nekom trenutku čita, pohranjeno je u posebnoj varijabli `$ARGV`. Kada se dođe do kraja zadnje navedene datoteke, operator <> vraća vrijednost `undef`, što uzrokuje izlazak iz petlje.

U skladu s filozofijom Perla, i ovaj program se može još skratiti koristeći tipične pokrate.

```
while (<>) { # podrazumijevana varijabla $_
  chomp;  # podrazumijevana varijabla $_
  print "Procitao sam: $_ \n";
}
```

Korištenje argumenata naredbenog retka

Argumenti navedeni u naredbenom retku pri pozivu programa, u Perl programu dostupni su preko posebnog polja `@ARGV`. Pristup elementima polja `@ARGV` je kao i svakom drugom polju, pa se preneseni argumenti naredbenog retka mogu koristiti prema želji i potrebi programera.

Operator <> koristi upravo elemente polja `@ARGV` kao imena datoteka iz kojih čita podatke, što omogućuje i zanimljivu manipulaciju tim poljem kako bi <> čitao željene datoteke.

```
# natjerat ćemo <> da čita sljedeće datoteke
@ARGV = qw/ dat1 dat2 dat3 /; # imena datoteka
while (<>) {
  chomp;
  print "Ucitao sam redak: $_ \n";
}
```

3.6 Ispis na standardni izlaz

I u dosadašnjim primjerima susreli smo se s operatorom `print`, no sada ćemo o njemu reći još par riječi. Operator `print` kao argumente dobiva listu vrijednosti, koje jednu po jednu (kao znakovne nizove) ispisuje na standardni izlaz. Argumenti mogu biti i izrazi, a oni se prije poziva operatora izračunavaju.

```
$name = "Larry";
print "Hi, $name, did you know that 3+4 is ",
              3+4, "?\n";
```

Susreli smo se i s interpolacijom polja, no treba naglasiti da se ispis polja kao argumenta operatora `print` i polja interpoliranog u znakovnom nizu razlikuju. Naime, pri interpolaciji se između ispisanih elemenata liste umeću praznine (točnije, separatorski niz iz posebne varijable `$"`), dok se pri ispisu polja ne umeću separatori.

```
@array = qw/ fred barney betty /;
print @array;     # lista vrijednosti, bez razmaka
                  # fredbarneybetty

print "@array";  # interpolacija polja - razmaci izmedju elemenata
                 # fred barney betty
```

Zagrade kod operatora `print` su *neobavezne* i mogu se izostaviti ako to neće izazvati promjenu značenja naredbe

```
print("Hello, world!\n");
print "Hello, world!\n";
```

Međutim, ako izraz koji se ispisuje daje naredbi za ispis oblik poziva funkcije, mogući su pripremi pri interpretaciji naredbe.

```
print (2+3)*4;  # Ups!
```

U ovom primjeru izraz unutar zagrade shvaća se kao argument operatora `print`, pa se ispisuje se `"5"`, a povratna vrijednost (1 u slučaju uspješnog ispisa, 0 inače) množi se s `4` i odbacuje jer se izraz ne pohranjuje. Ponašanje je ekvivalentno sljedećem izrazu:

```
( print(2+3) ) * 4;  # Ups!
```

U ovoj situaciji uključena upozorenja mogu pomoći, jer će Perl primijetiti da se rezultat izraza nigdje ne koristi.

Formatirani ispis

I Perl, kao i ljuska, nudi mogućnost formatiranog ispisa. I tu se pojavljuje operator `printf`, koji je sličan istoimenoj funkciji u programskom jeziku C.

```
printf "Hi %s, your password expires in %d days!\n",
       $user, $days_to_die;
```

Kao prvi argument navodi se formatni niz, u kojem znakovi predznačeni s `%` određuju oblik ispisa, odnosno vrstu *pretvorbe* odgovarajućih argumenata navedenih u nastavku.

Oznake formata uglavnom odgovaraju onima u C-u. U primjeru ih je nekoliko ilustrirano.

```
# %g : općeniti format za broj -- automatski izbor formata
printf "%g %g %g\n", 5/2, 51/17, 51 ** 17;
                   # 2.5   3    1.0683e+29

# %d : dekadski cijeli broj
printf "in %d days\n", 17.85;
      # in 17 days

# %o : oktalni broj
# %x : heksadekadski broj
printf("oktalno: %o \t heksadekadski: %x \n", 254, 254);
      # oktalno: 376     heksadekadski: fe
```

Operator `printf` najčešće se koristi za tablični ispis, čemu pogoduje mogućnost definiranja širina polja koje se ispisuje. Ako podatak ne stane u zadanu veličinu polja, polje se proširuje.

```
printf "%6d\n", 42;
printf "%2d\n", 2e3 + 1.95;           # polje se prosiruje: 2001!
```

Negativna oznaka širine polja znači da se ispis lijevo poravnava.

```
printf "%-15s\n", "flintstone";
```

Format za brojeve s pomičnom točkom `%f` zaokružuje ispisani broj, pri čemo se može zadati i broj decimalnih mjesta.

```
printf "%10f\n", 6 * 7 + 2/3;   # 42.666667
printf "%10.3f\n", 6 * 7 + 2/3; #    42.667
printf "%10.0f\n", 6 * 7 + 2/3; #        43
```

Znak postotka može se uključiti u ispis operatorom `printf` kao `"%%"`.

3.7 Datoteke

Za pristup datotekama, Perl koristi poseban tip podataka i niz operatora, kojima se datoteke otvaraju, zatvaraju, odnosno kojima se podaci iz datoteka čitaju li u njih zapisuju. Datoteci se pri otvaranju dodjeljuje identifikator (*engl. filehandle*). To su Uobičajena Perl imena, s tim da se za razliku od ostalih imena, *ne predznačavaju.* Stoga postoji opasnost da se kao ime upotrijebi neka ključna riječ. Kako se to ne bi događalo, uobičajeno je identifikatore datoteka pisati velikim slovima. Perl koristi i 6 posebnih identifikatora datoteka: `STDIN`, `STDOUT`, `STDERR`, `DATA`, `ARGV` i `ARGVOUT`. Prva tri označavaju standardne podatkovne tokove za ulaz, izlaz i dijagnostički ispis. Preostala 3 imaju specifične namjene i rijetko će nam trebati, pa ih nećemo ovdje objašnjavati.

3.7.1 Otvaranje datoteke

Datoteka se otvara korištenjem operatora `open`, kojem se kao argumenti navode identifikator i ime datoteke. Kao prefiks imena datoteke navodi se i oznaka načina pristupa, pri čemu < označava čitanje (ta se oznaka ne mora niti navoditi jer se podrazumijeva), > označava datoteku koje se otvara za pisanje, dok >> označava nadodavanje sadržaja u datoteku koja se otvara.

```
open CONFIG, "dino";    # otvara se za citanje(default)
open CONFIG, "<dino";   # otvara se za citanje
open BEDROCK, ">fred";  # otvara se za pisanje
open LOG, ">>logfile";  # otvara se za nadodavanje
```

Kako se u znakovnom nizu koji se predaje kao argument mogu obavljati i interpolacije varijabli, čije vrijednosti u nekim slučajevima može upisivati i korisnik, ovakav način zadavanja načina pristupa može predstavljati sigurnosni rizik. Primjerice, moguć je scenarij u kojemu se u varijabli `$selected_output` preuzima od korisnika ime odredišne datoteke za neku operaciju. Ako korisnik podmetne odgovarajući znakovni niz, može izmijeniti predviđenu operaciju kao u sljedećem primjeru.

```
# recimo da je korisnik unio $selected_output = ">passwd"

# željeli smo zapisati podatke u obrisanu datoteku
# a sada se obavlja nadodavanje, i to u osjetljivu datoteku
open LOG, ">$selected_output";
```

Ovakvim se manipulacijama može doskočiti na razne načine, između ostaloga i provjerom upisanog niza. Međutim, zbog navedenih rizika, u novijim inačicama Perla postoji i oblik operatora `open` s tri argumenta, gdje je oznaka pristupa odvojena od imena datoteke.

```
open CONFIG, "<", "dino"; # citanje
open BEDROCK, ">", $file_name; # pisanje
open LOG, ">>", &logfile_name(  ); # nadodavanje
```

Nije uvijek moguće otvoriti željenu datoteku. Moguće je da datoteka zadanog imena ne postoji, ili korisnik nema odgovarajuću dozvolu pristupa ili nešto slično. Zato je uobičajeno ispitati uspješnost otvaranja datoteke temeljem povratne vrijednosti funkcije `open`.

```
my $success = open LOG, ">>logfile";
if ( ! $success) {
 # The open failed ...
}
```

Datoteku možemo zatvoriti operatorom `close` uz navođenje njenog identifikatora kao argumenta.

```
close BEDROCK;
```

Perl automatski zatvara datoteku ako njen identifikator koristimo za novo otvaranje, kao i kada program završi. Stoga Perl programi najčešće ne brinu o eksplicitnom zatvaranju datoteka.

Prijevremeni izlazak iz programa

U slučaju da naš program ustanovi da je došlo do problema s otvaranjem datoteke, vjerojatno ćemo htjeti prekinuti nastavak njegovog izvođenja. To se može učiniti funkcijom `die`. Naravno da njeno djelovanje nije ograničeno na rad s datotekama, već je možemo upotrijebiti bilo gdje u programu. Pozivom funkcije, na `stderr` se ispisuje poruka koja se zadaje kao argument, te se prekida izvođenje programa s izlaznim statusom različitim od `0`. Primjerice, pokušaj otvaranja datoteke i napuštanje programa u slučaju neuspjeha može izgledati ovako:

```
if ( ! open LOG, ">>logfile"){
   die "Ne mogu otvoriti datoteku: $!";
}
```

U ovom primjeru primjećujemo još jednu varijablu specijalnog imena, kojima Perl obiluje. Varijabla `$!` sadrži *poruku operacijskog sustava* o razlogu pogreške ("`permission denied`" ili "`file not found`"). Naravno, takva poruka ima smisla samo ako ispitujemo pogrešku koja nastaje pri pozivu OS-a. Funkcija `die` automatski nadodaje ime Perl programa u kojem je pozvana, te odgovarajući broj retka u programu. Primjerice, mogli bismo dobiti sljedeću poruku:

```
Ne mogu otvoriti datoteku: permission denied at program.pl line 123.
```

Ako *ne želimo* ispis imena i retka programa, niz koji se ispisuje treba zaključiti s `\n`.

```
if (@ARGV < 2){
   die "Not enough arguments\n";
}
```

3.7.2 Korištenje datoteke

Nakon što je datoteka uspješno otvorena, koristi se kao i `STDIN`/`STDOUT`. Dakle, učitavanje retka inicira se navođenjem identifikatora datoteke unutar trokutastih zagrada (<>), dok se zapisivanje podataka u datoteku postiže primjenom operatora `print` ili `printf`.

```
if ( ! open PASSWD, "/etc/passwd") {
  die "How did you get logged in? ($!)";
}

while (<PASSWD>) {
  chomp;
   . . .
}
```

Kako bi se podaci zapisali u datoteku, ona mora biti otvorena za pisanje ili nadodavanje, a operator `print` (ili `printf`) mora imati i identifikator datoteke. Identifikator se, malo neobično, navodi neposredno prije liste argumenata, bez odvajanja zarezom, kao u sljedećim primjerima.

```
print LOG "Captain's log, stardate 3.14159\n";

printf STDERR "%d percent complete.\n", $done/$total * 100;

printf (STDERR "%d percent complete.\n", $done/$total * 100);

printf STDERR ("%d percent complete.\n", $done/$total * 100);
```

Promjena podrazumijevane datoteke

Kada često zapisujemo podatke u istu datoteku, odnosno u svom programu imamo niz naredbi `print`, bilo bi zgodno izbjeći pisanje identifikatora datoteke. Perl omogućuje izmjenu podrazumijevanog identifikatora datoteke za ispis operatorima `print` i `printf`, koji je `STDOUT`. Tu promjenu možemo obaviti primjenom operatora `select`.

```
select BEDROCK;

# sada ispis ide u datoteku čiji je identifikator BEDROCK
print "I hope Barney doesn't find out about this.\n";
print "Wilma!\n";
```

Da ne bi bilo zabune kasnije u programu, poželjno je vratiti podrazumijevanu vrijednost na `STDOUT`.

```
select STDOUT;
```

3.8 Asocijativna polja

Kao i mnogi drugi skriptni jezici, Perlu skupu ugrađenih tipova podataka ima i tipove visoke razine. Jedan takav tip, liste, već smo upoznali. Drugi tip visoke razine apstrakcije su asocijativna polja (*engl. hash, associative array*). U Pythonu se takva struktura naziva rječnikom (*engl. dictionary*). Asocijativno polje je podatkovna struktura slična polju, uz razliku da indeksiranje elemenata nije slijedno, cijelim brojevima. Ovdje je indeks pojedinog elemenata proizvoljni ali *jedinstveni* niz znakova, koji nazivamo ključem (*engl. key*). Zapravo, kao ključ možemo koristiti proizvoljan skalar, koji se pretvara u string. Primjerice, kao ključ možemo koristiti izraz `50/20`, no on se pri indeksiranju asocijativnog polja automatski pretvara u znakovni niz `"2.5"`.

Veličina asocijativnog polja nije ograničena, osim veličinom raspoloživog memorijskog prostora. Asocijativno polje može se promatrati kao jednostavna baza podataka u kojoj se podacima pristupa preko ključa. Implementacija ove strukture podataka u Perlu je vrlo brza i koristi napredne tehnike raspršenog adresiranja (*engl. hashing*).

3.8.1 Pristup elementima asocijativnog polja

Indeksiranje elemenata asocijativnog polja slično je indeksiranju polja, ali se koriste vitičaste zagrade i znakovni niz (odnosno bilo koji skalar koji se pretvara u niz) kao indeks.

```
$procitani_element = $asocijativno_polje{$neki_kljuc}
```

Ime asocijativnog polja je standardni Perl identifikator, a i ovdje se koristi zaseban prostor imena. U slučaju pristupa elementu asocijativnog polja, to je jasno po vitičastim zagradama, dok se ime asocijativnog polja kao cjeline prefiksira znakom ”%”.

Pohranjivanje vrijednosti ilustrirano je s nekoliko primjera. Ukoliko već postoji element s navedenim ključem, on se prepisuje, a ako takav element ne postoji, upisuje se u asocijativno polje.

```
$family_name{"fred"} = "flintstone";
$family_name{"barney"} = "rubble";

# ključ može biti i izraz
$foo = "bar";
print $family_name{ $foo . "ney" }; # "rubble"
```

Ako pokušamo čitati element s nepostojećim ključem, vraća se vrijednost `undef`.

```
$proba = $family_name{"larry"};   # --> undef
```

3.8.2 Asocijativno polje kao cjelina

Asocijativno polje kao cjelina imenuje se predznačeno znakom postotka `"%"`, a u programima se ne pojavljuje kao literal. Umjesto toga, obavlja se inicijalizacija listom, koja mora biti u obliku parova ključ – vrijednost.

```
%some_hash = ("foo", 35, "bar", 12.4, 2.5, "hello",
              "wilma", 1.72e30, "betty", "bye\n");
```

U primjeru se ključu `"foo"` pridjeljuje vrijednost `35`, ključu `"bar"` vrijednost `12.4` itd. Uočite da se u nekim od parova kao ključ pojavljuje broj, a kao vrijednost znakovni niz. Sjetimo se da se ključevi prilikom pristupa elementu pretvaraju u znakovni niz iz kojega se izračunava vrijednost *hash* funkcije.

Kada se asocijativno polje nađe u kontekstu liste, automatski se pretvara u listu čiji elementi su poredani kao parovi ključ – vrijednost.

```
@any_array = %some_hash;
```

Pritom treba biti svjestan da parovi *nisu nužno* u istom redoslijedu u kojem je bila izvorna lista kojom smo inicijalizirali asocijativno polje, jer Perl pohranjuje parove redoslijedom koji mu odgovara zbog brzine pristupa.

3.8.3 Operacije s asocijativnim poljima

Asocijativno polje može se kopirati naredbom pridruživanja.

```
%new_hash = %old_hash;
```

Međutim, Perl kopiranje ne obavlja izravno, već najprije pretvara (”razmotava”) asocijativno polje u listu (*engl. unwind*), a zatim tom listom inicijalizira novo asocijativno polje.

Češća operacija je ”okretanje” asocijativnog polja operatorom `reverse`. U toj operaciji se zamjenjuje uloga ključa i vrijednosti, a operacija se obavlja tako da se početno asocijativno polje pretvara u listu, lista se okreće, a zatim se tom listom inicijalizira novo asocijativno polje.

```
%inverse_hash = reverse %any_hash;
```

Operacija je ispravna samo ako su vrijednosti jedinstvene, jer inače se duplicirani elementi prepisuju (ostaje zadnja zapisana vrijednost).

Još jedna notacija

Kada se obavlja inicijalizacija asocijativnog polja listom, nije uvijek očigledno koji element liste je ključ, a koji vrijednost. Perl nudi alternativnu notaciju, u kojoj se može jasno pokazati odnos ključeva i vrijednosti. Za povezivanje ključa i vrijednosti koristi se oznaka "=>".

```
%last_name = (
  "fred"   => "flintstone",
  "dino"   => undef,
  "barney" => "rubble",
  "betty"  => "rubble",
);
```

Zanimljivo je da se tu zapravo radi samo o drugom zapisu zareza, što znači da se ta oznaka može koristiti bilo gdje umjesto zareza.

Funkcije nad asocijativnim poljima

Perl ima nekoliko korisnih funkcija za rad s asocijativnim poljima. Funkcija `keys` vraća listu svih ključeva, a funkcija `values` listu svih vrijednosti u asocijativnom polju.

```
%hash = ("a" => 1, "b" => 2, "c" => 3);
@k = keys %hash;   # ("a", "b", "c")
@v = values %hash; # (1, 2, 3 )
```

Redoslijed elemenata može varirati, no redoslijed te dvije liste je *usklađen*, naravno, ako se između poziva funkcija `keys` i `values` asocijativno polje nije mijenjalo. Poziv ovih funkcija u skalarnom kontekstu daje broj elemenata asocijativnog polja.

```
$count = keys %hash;  # broj elemenata = 3
```

Funkcija `each` omogućava prolazak kroz sve elemente asocijativnog polja. Svakim pozivom funkcija vraća sljedeći par ključ – vrijednost, o obliku dvočlane liste. Kad se stigne do kraja asocijativnog polja, vraća se prazna lista, što je prikladno za korištenje u petlji.

```
while ( ($key, $value) = each %hash ) {
  print "$key => $value\n";
}
```

Lista koju vraća `each %hash` pridružuje se paru `($key, $value)`. Kako se pridruživanje obavlja u uvjetnom izrazu `while` petlje, radi se o skalarnom kontekstu, pa je uvjet ispunjen dok je *izvorna* lista neprazna. kad `each` vrati praznu listu, uvjet poprima vrijednost `false`, iako lista `($key, $value)` poprima vrijednost `(undef, undef)`, odnosno nije prazna.

Primjer primjene

Kao jedan ilustrativan primjer, oblikovat ćemo jednostavan program za evidenciju posudbi knjiga u knjižnici. Ime osobe može biti ključ u asocijativnom polju, a kao vrijednost pohranjujemo broj knjiga koje je osoba posudila.

```
# evidentiramo broj posuđenih knjiga
$books{"fred"} = 3;
$books{"wilma"} = 1;
```

Ako želimo saznati ima li osoba posuđenu knjigu, jednostavno ćemo ispitati sadržaj odgovarajućeg elementa asocijativnog polja.

```
if ($books{$osoba}) {
  print "$osoba ima bar jednu posudjenu knjigu.\n";
}
```

Ako osoba nema posuđenu knjigu, njen zapis je `0`, a ako nije nikada ni posuđivala knjige, vrijednost je `undef`, a između te dvije situacije bismo željeli vidjeti razliku. Da bismo ispitali postoji li neki ključ u asocijativnom polju, možemo primijeniti funkciju `exists`.

```
$osoba = "dino";

if (exists $books{$osoba}) {
  print "$osoba je clan knjiznice.\n";
}
```

Zapis iz asocijativnog polja možemo obrisati funkcijom `delete`.

```
delete $books{"betty"}; # betty se isčlanjuje
```

Interpolacija *elemenata* asocijativnog polja u znakovne nizove se obavlja kao i za bilo koju drugu skalarnu varijablu.

```
foreach $osoba (sort keys %books) {
  if ($books{$osoba}) {
    print "$osoba je posudila $books{$osoba} knjiga.\n";
  }
}
```

No, asocijativno polje kao cjelina *ne može se interpolirati.* Ako unutar znakovnog niza navedemo `"%books"`, ne dolazi do interpolacije, već se ispisuje doslovno taj niz.

3.9 Regularni izrazi

Podrška regularnim izrazima je jedna od najjačih strana Perla. Regularne izraze upoznali smo kod Unix alata (`grep, sed`). Značenje, a i sintaksa, u Perlu su praktički jednaki, pa ih nećemo ponavljati. Možete ih ponoviti po odjeljku 2.5.3. Sintaksa odgovara proširenoj sintaksi regularnih izraza. Ovdje ćemo komentirati neke specifičnosti regularnih izraza u Perlu, te dodatno pojasniti neke detalje koje nismo dovoljno obradili u prethodnom tekstu.

Perl kvantifikatori, kao uostalom i standardni kvantifikatori koje koriste `grep` i `sed`, su "pohlepni" (*engl. greedy*). To znači da se nastoji postići podudaranje najduljeg mogućeg podniza. Primjerice, regularni izraz `/[0-9]+/` podudara se s cijelim nizom `"1234567890"`.

To nam u nekim slučajevima neće odgovarati. Primjerice, ako u retku datoteke `passwd` koji izgleda otprilike ovako:

```
larry:x:100:10:Larry Wall:/home/larry:/bin/bash
```

želimo pronaći korisničko ime (u ovom slučaju "larry"), vjerojatno bismo razmišljali da treba pronaći proizvoljan niz znakova koji završava dvotočkom, pa bismo napisali regularni izraz `/.+:/`. Međutim, taj će se izraz podudariti s podnizom "`larry:x:100:10:Larry Wall:/home/larry:`". Radi se upravo o pohlepnom kvantifikatoru koji pronalazi najdulji niz koji zadovoljava iraz. Tu nam može pomoći negiranje klase znakova, pa bismo izraz mogli zapisati kao `/[^:]+:/`. Ovdje se u podnizu prije dvotočke dozvoljavaju samo znakovi različiti od dvotočke, pa je rezultat upravo podniz do *prve* dvotočke. Ovo rješenje jednako je primjenjivo i u `sed`-u.

Međutim, Perl nudi i elegantnije rješenje, koje nije podržano u POSIX regularnim izrazima. Radi se o "skromnim" (*engl. non-greedy, lazy*) kvantifikatorima, koji se uključuju tako da se

iza standardnog kvantifikatora doda upitnik. Regularni izraz iz našeg primjera mogao bi se napisati: `/.+?:/`, i doista će se podudarati s nizom proizvoljnih znakova do *prve* dvotočke.

Još jedno svojstvo na koje treba paziti je da regularni izrazi nastoje postići podudaranje *čim ranije*, pri čemu se pretraživanje obavlja s lijeva nadesno. Primjerice, ako želimo pronaći (i npr. izbrisati) niz znakova `"x"` u sredini niza `"fred xxxxxxx barney"`, regularni izraz `/x*/` neće pomoći. Pronaći će *prazni niz* na samom početku niza!

Klase znakova definiraju se kao i u standardnim izrazima, a za neke često korištene klase znakova Perl definira kratice:

`\w` : alfanumerički znakovi i podvlaka `[A-Za-z0-9_]`

`\d` : znamenke `[0-9]`

`\s` : prazni znakovi `[\f\t\n\r ]` Negacije ovih kratica imenovane su velikim slovima:

`\W` : alfanumerički znakovi i podvlaka `[^\w]`

`\D` : znamenke `[^\d]`

`\S` : prazni znakovi `[^\s]`

Kratice se mogu koristiti kao elementi klase znakova. Primjerice, izraz `/[\dA-Fa-f]+/` pronalazi heksadekadski broj. Ponekad je korisna kombinirana klasa znakova `[\d\D]`, jer se podudara se s proizvoljnim znakom, uključujući i `"\n"`, što nije slučaj s metaznakm `"."`

Sidra se također koriste kao i kod standardnih regularnih izraza, pri čemu omogućuju ograničavanje mogućih pozicija podudaranja (početak retka, kraj retka,...). Oznake odgovaraju onima koje smo već upoznali.

3.9.1 Operacije s regularnim izrazima

Uobičajene operacije koje rade s regularnim izrazima su podudaranje ili pretraživanje (*engl. pattern-matching*) i zamjena (*engl. substitution*). Među ove operacije uvjetno se može ubrojiti i zamjena znakova (*transliteracija*).

Ako se ne upotrijebi operator povezivanja (*engl. binding*), o kojemu će kasnije biti riječi, operacije se izvode nad podrazumijevanom varijablom (`$_`)

Podudaranje

Operacija podudaranja (*engl. matching*) ispituje podudara li se zadani regularni izraz s nekim podnizom zadanog znakovnog niza i vraća odgovarajuću vrijednost istinitosti (`true`, `false`), koja se može ispitati u uvjetnom izrazu.

Operator podudaranja je `m//`, pri čemu se unutar graničnika navodi regularni izraz koji se traži u znakovnom nizu. Kao i kod operatora `qw//`, kao graničnik se može koristiti i neki drugi znak interpunkcije: `m/abc/, m(abc), m<abc>, m!abc!, m^abc^,...`

Niz za podudaranje podrazumijeva se u varijabli `$_`.

```
$_ = "yabba dabba doo";
if (m/abba/)
  print "Pronasao!\n";
}
```

Ako se izostavi oznaka operatora, podrazumijeva se podudaranje.

```
if (/Windows 95/){
  print "Time to upgrade?\n"
}
```

Opcije podudaranja

Prilikom primjene operatora podudaranja mogu se specificirati neke opcije, a one se navode kao slovo iza operatora. Primjerice, opcija `/i` znači da se pri podudaranju zanemaruje razlika između velikih i malih slova (*engl. case-insensitive*).

```
print "Would you like to play a game? ";
chomp($_ = <STDIN>);
if (/yes/i) {  # case-insensitive match
  print "OK, I recommend bowling.\n";
}
```

Opcija `/s` označava da se cijeli znakovni niz koji se ispituje smatra jednim retkom (*engl. single-line*), što uzrokuje da se metaznak ".” podudara i sa znakom "`\n`". No, u tom načinu se oznake početka i kraja retka ne podudaraju.

Opcija `/m` je suprotna prethodnoj (*engl. multi-line*), čime se osigurava podudaranje metaznakova `^` i `$` s početkom i krajem retka određenima znakom `"\n"` unutar niza.

Opcija `/x` označava zanemarivanje praznih znakova i komentara unutar regularnih izraza, čime se ponekad može poboljšati preglednost. Primjerice, mogli bismo napisati sljedeći izraz uz komentiranje svake oznake.

```
/
  -?   # an optional minus sign
  \d+  # one or more digits before the decimal point
  \.?  # an optional decimal point
  \d*  # some optional digits after the decimal point
/x     # end of pattern
```

Opcija `/g` označava globalno podudaranje. U toj varijanti podudaranja vraća se lista svih pronađenih podnizova.

```
if (@perls = /perl/gi) {
    printf "Perl mentioned %d times.\n", scalar @perls;
}
```

Osim spomenutih, ima još nekoliko opcija. Više opcija može se kombinirati tako da se navedu jedna iza druge, kao u prethodnom primjeru.

Zamjena

Operator zamjene (*engl. substitution*) je oblika `s///`. Unutar graničnika navodi se najprije regularni izraz koji se traži, a zatim niz znakova kojim se pronađeni podniz zamjenjuje.

```
$_ = "He's out bowling with Barney tonight.";
s/Barney/Fred/;  # Replace Barney with Fred
print "$_\n";
```

Operator vraća logičku vrijednost `true` ako je zamjena uspješna (traženi uzorak je pronađen i zamijenjen), pa je tu vrijednost često uputno provjeriti.

```
$_ = "fred flintstone";
if (s/fred/wilma/) {
  print "Successfully replaced fred with wilma!\n";
}
```

Bez dodatnih opcija obavlja se samo zamjena prve pojave traženog uzorka. Ako želimo zamijeniti sve *nepreklapajuće* pojave uzorka, treba zadati opciju globalne zamjene `/g`.

Transliteracija

Operatorom `tr/lista_pretrazivanja/lista_zamjene/cds` obavlja se zamjena znakova liste pretraživanja znakovima liste zamjenskih znakova. Operator ima sinonim `y///` radi kompatibilnosti s naredbom (`sed`).

Zamjena se obavlja znak po znak, a povratna vrijednost je broj zamijenjenih (ili obrisanih) znakova. Operator prihvaća nekoliko opcija:

`/c` : komplement liste pretraživanja
`/d` : brisanje pronađenih znakova za koje nema zamjene
`/s` : slijed znakova koji su zamijenjeni istim znakom reducira se na jedan znak

Ako je niz zamjenskih znakova prazan, nema zamjene. Ponašanje operatora možemo ilustrirati s nekoliko primjera:

```
tr/A-Z/a-z/;      # $_ u mala slova
$cnt = tr/*/*/;  # broji zvijezde u $_
$cnt = tr/0-9//; # broji znamenke u $_
```

3.9.2 Operator povezivanja

U prethodnim primjerima vidjeli smo regularne izraze koji se primjenjuju na podrazumijevanu varijablu `$_`, u koju je prethodno potrebno upisati niz nad kojim se izraz primjenjuje. To je najčešće sasvim prikladno, jer se tekst nad kojim želimo obaviti učitava sa standardnog ulaza, korištenjem uobičajenih pokrata, koje pročitani redak smještaju upravo u podrazumijevanu varijablu.

Ako nam to zbog nekog razloga ne odgovara, i želimo regularni izraz primijeniti na neku drugu varijablu, možemo upotrijebiti operator povezivanja (*engl. binding*) `=~`, kao u primjeru.

```
my $some_other = "I dream of betty rubble.";
if ($some_other =~ /\brub/) {
  print "Aye, there's the rub.\n";
}
```

3.9.3 Interpolacija u regularnim izrazima

Regularni izrazi se interpoliraju kao znakovni nizovi u dvostrukim navodnicima, pa je moguće tijekom izvođenje skripte generirati izraze u varijablama.

```
#!/usr/bin/perl -w
my $uzorak = "larry";
while (<>) {
  if (/^($uzorak)/) { # uzorak na pocetku retka
    print "$uzorak je na pocetku niza $_";
  }
}
```

Ovdje treba napomenuti da ovo ne vrijedi za operaciju transliteracije, odnosno liste znakova koji se zamjenjuju ne mogu se ovako jednostavno dinamički generirati.

3.9.4 Varijable podudaranja

Zagrade služe za grupiranje dijelova regularnog izraza, no kada regularni (pod)izraz uokvirimo zagradama, aktivira se i pamćenje pronađenih uzoraka. Ukoliko je više podizraza uokvirenih

zagradama, za svaki se oblikuje varijabla koja pamti pronađene podnizove. Varijable podudaranja imenuju se numeričkim imenima (`$1`, `$2`, `$3` itd.) koja odgovaraju redoslijedu podizraza u zagradama.

```
$_ = "Hello there, neighbor";
if (/(\S+) (\S+), (\S+)/) { # zarez van zagrada
  print "words were $1 $2 $3\n";
}
```

Važno je biti svjestan da pamćenje varijabli traje do sljedećeg *uspješnog* podudaranja, kada se vrijednosti reinicijaliziraju. Zato je potrebno provjeriti uspješnost svakog podudaranja, a ne "na slijepo" koristiti rezultate iz varijabli. Naime,može nam se dogoditi da podudaranje bude beuspješno (uzorak se na pronađe) a mi pogrešno koristimo rezultate prethodnog.

3.9.5 Povezivanje unazad

Povezivanje unazad (*engl. backreference*) prikazali smo i u okviru regularnih izraza ljuske operacijskog sustava. I u Perlu taj mehanizam funkcionira na isti način, a usko je vezan sa varijablama podudaranja. Povezivanje unazad odnosi se na iste podnizove kao i varijable podudaranja, ali omogućuje korištenje *u samim izrazima* kako bi se pronađeni podniz može se koristiti za podudaranje u ostatku izraza. Reference se označavaju izrazima: `\1`, `\2`, `\3` itd.

Ilustrirajmo ovaj mehanizam još jednim jednostavnim primjerom. Napišimo regularni izraz koji će pronaći ponovljene troslovne riječi u tekstu (primjerice "the the" ili "ali ali").

```
/(\w\w\w)\s\1/;
```

Osim samog pronalaženja, bilo bi dobro i ukloniti duplikate, što možemo učiniti sljedećim izrazom:

```
s/(\w\w\w)\s\1/$1/g;
```

Primijetimo da se unutar uzorka za podudaranje koristi povezivanje unazad, dok se u zamjenskom tekstu koriste varijable podudaranja.

Podudaranje u kontekstu liste

Kada se operator podudaranja (`m//`) koristi u kontekstu liste, povratna vrijednost je lista varijabli podudaranja.

```
$_ = "Hello there, neighbor!";
my($first, $second, $third) = /(\S+) (\S+), (\S+)/;
print "$second is my $third\n";
```

Na ovaj način možemo "pospremiti" rezultat podudaranja za kasniju primjenu. Ako se koristi opcija `/g`, uzorak se može naći na više mjesta u nizu, jer svako podudaranje vraća varijable koje odgovaraju podizrazima u zagradama.

```
my $text = "Fred dropped a 5 ton granite block";
my @words = ($text =~ /([a-z]+)/ig);
print "Result: @words\n";
   # Result: Fred dropped a ton granite block!
```

Funkcija split

Osim do sada prikazanih operatora koji rade s regularnim izrazima, često se koristi funkcija `split`, koja dijeli znakovni niz prema navedenom uzorku. Ona se najčešće koristi s vrlo jednostavnim regularnim izrazima i vrlo je prikladna za dijeljenje podataka odvojenih jednostavnim separatorima kao što su tabulatori, dvotočke, razmaci i slično.... Pritom se separator zadaje regularnim izrazom. Tipičan oblik korištenja funkcije `split` je:

```
@fields = split /separator/, $string;
```

U nastavku je navedeno nekoliko primjera, čiji rezultati su navedeni u obliku komentara.

```
@fields = split /:/, "ab:c:de"; # ("ab", "c", "de")
@fields = split /:/, "ab::cd"; # ("ab", "", "cd")

$ulaz = "Ovo je  \t \t  test.\n";
@lista = split /\s+/, $ulaz; # ("Ovo", "je", "test.")
```

Podrazumijevani oblik, koji se dobiva kada se ne navedu argumenti, je dijeljenje podrazumijevane varijable `$_` na prazninama:

```
my @fields = split;  # kao split /\s+/, $_;
```

Funkcija join

Često je potrebno više nizova povezati u jedan. Primjerice kada smo ulazni niz rastavili na komponente pomoću funkcije `split`, od tih dijelova često želimo sastaviti nov, preoblikovani niz.

Oblik poziva funkcije je:

```
$result = join $glue, @pieces;
```

Prvi argument mora biti znakovni niz koji se umeće između dijelova, dok su ostali argumenti podnizovi koje želimo spojiti. Konačni niz dobiva se povezivanjem dijelova između kojih se umeće zadani niz (`glue`). Funkcija spojeni niz vraća kao povratnu vrijednost.

```
$x = join ":", 4, 6, 8, 10, 12;  # "4:6:8:10:12"
```

Još nekoliko primjera:

```
my $y = join "abc", "def"; # PAZI! samo "def"

my @lst = split /:/, $x;  # @lst = (4, 6, 8, 10, 12)
my $z = join "-", @lst;   # $z ="4-6-8-10-12"
```

3.10 Perl programi u naredbenom retku

Perl se može koristiti kao alat koji se poziva iz naredbenog retka, uz navođenje kratkog programa (*engl. oneliner*). Pri pozivu je moguće navesti cijeli niz opcija, a najjednostavnije je samo navođenje naredbe koju treba izvršiti.

```
perl -e'naredba'
```

Primjerice, kratki program koji ispisuje znakova s ASCII kodovima od 65 do 90 može se napisati izravno u naredbenom retku.

```
perl -e'for (65..90) { print chr($_) }'
```

Ova je mogućnost zanimljiva prije svega jer se na taj način `perl` može koristiti umjesto alata poput *sed*-a.

```
$ sed 's/Windows/Linux/g' OS.txt

$ perl -pe's/Windows/Linux/g' OS.txt
```

U gornjem primjeru opcija `-p` označava da se naredba primjenjuje na svaki redak učitan sa `STDIN` ili iz datoteka navedenih kao argumenti, te da se rezultat ispisuje. To je ekvivalent sljedećem pozivu.

```
perl -e'while (<>) {s/Windows/Linux/g;print}' OS.txt
```

4. Python

Python je jedan od danas najpopularnijih jezika. Iako ga u okviru ovog kolegija svrstavamo u skriptne jezike, odnosno jezike za pisanje skripti, radi se o bogatom objektno orijentiranom jeziku prikladnom za razvoj najzahtjevnijih aplikacija.

Važna odlika Pythona je njegova jednostavna i jasna sintaksa i dosljednost, koja ga čini lakim za učenje. Uz to, specifična sintaksa tjera programera na urednost, pa je kôd izuzetno čitljiv i pregledan.

Autor jezika je Guido van Rossum, a prv inačica jezika predstavljena 1991. godine. Ime jezika ne dolazi od zmije na koju asocira, već od popularne humorističke serije Leteći cirkus Montyja Pythona, pa su i mnogi primjeri u referentnim udžbenicima inspirirani njihovim skečevima.

Jezik se neprestano razvija i redovito izlaze nove inačice, a trenutne su Python 2.7.3 i Python 3.2.3, obje objavljene u travnju 2012. Neobičnost da su istovremeno aktualne dvije inačice jezika posljedica je uvođenja značajnih promjena u jezik s inačicom 3. Prva inačica Pythona 3 uvedena je 2009. godine, a njome su uvedene značajnije izmjene jezika, uz odricanje od kompatibilnosti starijeg kôda. Stoga je zadržana i varijanta jezika Python 2, koji se također dalje razvija, a zahvaljujući izuzetno bogatoj ponudi najrazličitijih biblioteka i dalje je vrlo raširen. S druge strane, mnoge od biblioteka postupno se prevode i usklađuju s Pythonom 3, pa se očekuje postupan prelazak korisnika na noviju inačicu.

U okviru predmeta koristit ćemo sintaksu Pythona 3, primjeri bi u većini slučajeva trebali raditi i na verziji 2, a u slučaju značajnijih razlika, na njih ćemo upozoriti.

4.1 Izvođenje programa u Pythonu

Kako biste mogli početi pisati programe u Pythonu, morate imati instaliranu odgovarajuću razvojnu okolinu. Najbolje je instalacijske datoteke preuzeti s web stranice `python.org`.

Python nudi nekoliko načina pokretanja naredbi odnosno programa, pri čemu je vrlo prikladno njegovo interaktivno sučelje koje omogućuje upis i trenutno izvršavanje naredbi. Interaktivni način rada je vrlo pogodan za učenje jezika i eksperimentiranje s raznim jezičnim konstruktima i bibliotekama, kao i za testiranje modula od kojih se grade složeniji programi.

Naravno, program (ili skriptu) moguće je unijeti u obliku tekstne datoteke, korištenjem uređivača teksta. Takva datoteka u terminologiji Pythona obično se naziva *modulom*, a može se pokrenuti iz razvojne okoline ili kao skripta iz naredbenog retka, na način koji smo već upoznali kod ljuske operacijskog sustava i Perla.

Interaktivno zadavanje naredbi

U interaktivnom načinu rada, nakon pokretanja interaktivnog sučelja Pythona, upisujemo naredbu po naredbu, naredba se odmah izvršava i rezultat se ispisuje.

```
$ python    # pokretanje interpretera iz naredbenog retka
>>> print('Hello world!')  # interaktivan unos naredbe
Hello world!
```

```
>>> print(2 ** 8)
256
```

U primjeru je prikazano pokretanje interpretera iz naredbenog retka, te unos dviju naredbi u interaktivnom sučelju. Znak >>> je odzivni znak koji označava da je Python spreman prihvatiti sljedeću naredbu. U sljedećem retku prikazan je ispis generiran izvršavanjem unesene naredbe.

Kada želimo napustiti Python, dovoljno je upisati naredbu `exit()`

Pisanje i pokretanje skripti

Interaktivni način rada je odličan za isprobavanje dijelova kôda, no obično svoje programe želimo pokretati više puta, pa ih u pravilu pohranjujemo u datoteke. Programske datoteke u Pythonu obično imenujemo s nastavkom `.py`, iako to nije obavezno. Međutim, ako našu programsku komponentu želimo moći učitati i pokrenuti iz drugog programa, onda je taj nastavak neophodan.

Primjerice, možemo napisati sljedeći program i pohraniti ga pod imenom `prvi.py`:

```
print(2 ** 8)  # 2^8
tekst = 'the bright side '
print(tekst + 'of life') # + je ulančavanje
```

Skriptu možemo pokrenuti iz naredbenog retka pozivanjem naredbe `python` kojoj se prosljeđuje ime programske datoteke.

```
$ python prvi.py
256
the bright side of life
```

Kao i skripte u Perlu ili u ljusci operacijskog sustava, i skripte u Pythonu mogu se deklarirati izvršnima i pokretati izravno. Naravno, to funkcionira na operacijskom sustavu Unix, i to ako u prvom retku skripte napišemo sekvencu za zadavanje interpretera.

```
#!/usr/bin/python
# ime skripte: drugi
print('leteći cirkus')
```

Skriptu treba označiti izvršivom i pokrenuti.

```
$ chmod +x drugi
$ ./drugi
leteći cirkus
```

Kako staza do interpretera Pythona može varirati ovisno o instalaciji, dobro je u prvom retku skripte upotrijebiti naredbu `env`. Time se postiže neovisnost o smještaju Python interpretera.

```
#!/usr/bin/env python
```

4.2 Moduli

Svaka datoteka s Python naredbama, čije ime sadrži nastavak `.py` je *modul*. Program može učitati (*engl. import*) neki modul, čime dobiva pristup njegovom sadržaju. Sadržaj modula se stavlja na raspolaganje okolini preko njegovih *atributa*.

Osnovna ideja programske arhitekture u Pythonu je da se veći programi se obično grade kao skup modula, koji učitavaju alate definirane u drugim modulima.

Operacija učitavanja modula inicira se naredbom `import`. Time se, između ostaloga, pokreće izvršavanje naredbi tog modula, pa je to ujedno i jedan od načina pokretanja programa.

```
$ python
>>> import prvi
the bright side of life
```

Pritom valja biti svjestan da se izvođenje naredbi modula obavlja samo pri prvom učitavanju. Naknadno ponovno pokretanje naredbe `import` neće imati efekta. Modul se neće ponovo učitati, čak ni ako se kôd modula u međuvremenu promijenio.

```
>>> import prvi   # novi pokušaj učitavanja nema efekta!
```

Razloge takvom ponašanju Pythona možemo tražiti u činjenici da je učitavanje modula skupa operacija. Potrebno je pronaći datoteku modula, prevesti u *byte-code*, te izvršiti naredbe. Zanimljivo je da Python iako se radi o interpretiranom jeziku, zapravo prilikom učitavanja prevodi modul u byte-code, koji se zatim izvodi na virtualnom stroju. To se radi zbog kasnijeg bržeg izvođenja dijelova učitanog modula. Prevedeni byte-code se pohranjuje u datoteci s nastavkom imena `.pyc`.

Ako ipak želimo ponovno učitati i izvršiti (eventualno izmijenjeni) modul, možemo to učiniti funkcijom `reload`. U Pythonu 2, `reload` je bila ugrađena funkcija, dok se u Pythonu 3 nalazi u modulu `imp`.

```
>>> import imp # reload više nije ugrađena funkcija
>>> imp.reload(prvi)
promijenili smo tekst :-)
<module 'prvi' from 'prvi.py'>
```

Samo se modul koji je već uspješno učitani može *ponovo* učitati. Treba napomenuti i to da je `reload` funkcija, dok je `import` naredba.

Namjena modula je organizacija biblioteka i alata, a modul zapravo predstavlja prostor imena (*engl. namespace*).

Učitavanjem modul postaje punopravni objekt programa u Pythonu, a atributi tog objekta odgovaraju imenima koje definira učitani modul. Program koji učitava modul dobiva pristup svim imenima pridijeljenima u najvišoj razini modula. Ta imena su najčešće pridijeljena *uslugama* koje modul nudi (*engl. eksportira*): funkcije, klase, varijable, namijenjene korištenju iz drugih programa. Pristup tim imenima postiže se naredbama `import` i `from`, ili pozivom funkcije `reload`. Ilustrirajmo to na primjeru modula `myfile.py`:

```
title = "The Meaning of Life"
```

Dva su načina pristupa atributima modula. Prvi je učitavanjem cijelog modula, nakon čega se ime atributa *kvalificira* imenom modula.

```
$ python
>>> import myfile
>>> print(myfile.title)  # kvalificiranje
The Meaning of Life
```

Sintaksa `objekt.atribut` nije specifična samo za module, već omogućuje pristup atributima bilo kojeg *objekta* i uobičajena je u Pythonu.

Drugi način pristupa imenima u modulu postiže se naredbom `from`.

```
$ python
```

```
>>> from myfile import title   # ime se kopira
>>> print(title)    # ne treba kvalifikacija
The Meaning of Life
```

Primijetite da se i ovim načinom učitavanja modula izvršavaju njegove naredbe, što se vidi iz generiranog ispisa. U ovom primjeru odabrali smo učitavanje samo jednog atributa modula (ovaj ih više niti nema), no općenito se može navesti lista atributa, a čest je i oblik u kojem se umjesto liste atributa navodi oznaka zvjezdica (*), koja označava da se učitavaju svi atributi modula. Zapravo se ovom naredbom stvaraju nova imena za objekte iz učitanog modula. Preko tih imena može se pristupiti atributima učitanog modula, a da pritom on kao cjelina ostaje nedohvatljiv.

```
>>> myfile.title   # greška
Traceback ...
```

Važno je uočiti da se ime modula u naredbama `import` i `from` navodi bez nastavka, a Python automatski nadodaje nastavak `.py` i traži datoteku. Zato je nužno programske datoteke koje namjeravamo učitavati kao module, primjereno imenovati.

Kao još jedan primjer, oblikujmo modul koji definira više imena (artibuta) i pogledajmo na koje načine modul možemo učitati.

```
# modul: tri.py

a = 'dead'    # definira 3 atributa
b = 'parrot'  # eksportiraju se
c = 'sketch'
print(a, b, c) # koriste se i u samom modulu
```

Modul možemo pokrenuti kao bilo koju skriptu u Pythonu. Pritom se naredbe modula izvršavaju.

```
$ python tri.py
dead parrot sketch
```

Međutim, modul možemo i učitati, primjerice iz interaktivne Pythonove ljuske. I u ovom slučaju naredbe se izvršavaju. Učitavanjem se stvara novi objekt, čijim atributima možemo pristupiti.

```
$ python
>>> import tri  # cijeli modul
dead parrot sketch
>>> tri.b, tri.c   # tuple
('parrot', 'sketch')
```

Drugi način učitavanja je, kao što smo već vidjeli, naredbom `from`. U tom slučaju se imena atributa modula kopiraju u prostor imena programa koji je učitao modul. U sljedećem primjeru uočite da se naredbe modula *ne izvršavaju*, jer je modul već ranije učitan.

```
>>> from tri import a, b, c     # kopiraju se imena
>>> b, c
('parrot', 'sketch')!
```

Listu imena koja su raspoloživa untar nekog modula možemo dobiti korištenjem ugrađene funkcije `dir`. Modul prije toga mora biti učitan, a funkciji se kao argument zadaje ime modula.

```
>>> dir(tri)    # mora prije biti učitan
['__builtins__', '__cached__', '__doc__', '__file__',
'__name__', '__package__', 'a', 'b', 'c']
```

Kao što vidimo u primjeru, funkcija `dir` vraća listu znakovnih nizova, od kojih neke prepoznajemo kao atribute modula kojega smo ranije definirali, dok su nam neka imena nepoznata. Primijetit ćemo i da ta imena započinju i završavaju dvostrukom podvlakom ("__"). To su imena *ugrađena*, Python ih unaprijed definira i imaju posebna značenja.

Ako modul učitamo naredbom `from`, objekt modul se stvara, ali se ne stvara ime koje bi ga referenciralo. U primjeru koji slijedi, `python` se ponovo pokreće.

```
$ python
>>> from tri import a, b      # naredbe se izvršavaju
dead parrot sketch
>>> a
'dead'
>>> tri.a        # ime koje referencira modul je nepoznato
...
NameError: name 'tri' is not defined
>>> import tri   # modul se neće izvršiti jer je već učitan
>>> tri.a        # ali sada mu možemo pristupiti
'dead'
>>> del tri     # brisanje reference na modul
>>> tri.a       # nema ga više
...
NameError: name 'tri' is not defined
>>> a # no ovaj je još tu
'dead'
```

4.3 Ugrađeni tipovi podataka

Kao i Perl, Python osim jednostavnih nudi i složene tipove objekata, liste i rječnike, kao sastavni dio jezika. Ugrađeni tipovi su učinkovito implementirani, jer se koriste optimirani algoritmi za baratanje strukturama podataka, koji su zbog brzine pisani u C-u.

Ugrađeni tipovi podataka s primjerima literala navedeni su u tablici.

Tip objekta	Primjer literala/stvaranja
Broj	`3.1415, 1234, 123e24, 3+4j`
Znakovni niz	`'spam', "guido's"`
Lista	`[1, [2, 'three'], 4]`
Rječnik	`{'food': 'spam', 'taste': 'yum'}`
n-torka	`(1,'spam', 4, 'U')`
datoteka	`text = open('eggs', 'r').read( )`

4.3.1 Brojevi

Python podržava više numeričkih tipova podataka: cijele brojeve, brojeve s pomičnom točkom i kompleksne brojeve. Neki primjeri su navedeni u tablici.

Literal	Interpretacija
`123, -24, 99999999999`	cijeli brojevi, neograničene duljine
`1.23, 3.14e-10, 4E210`	brojevi s pomičnom točkom
`0o177, 0x9ff, 0XFF`	oktalni i heksadekadski brojevi
`3+4j, 3.0+4.0j, 3J`	kompleksni brojevi

Cijeli brojevi

Cijeli brojevi u Pythonu pohranjuju se u dugom zapisu, pa je njihova veličina ograničena samo raspoloživom memorijom. To omogućuje računanje s vrlo velikim brojevima. Naravno, aritmetika s dugim brojevima je sporija od obične.

```
>>> 999999999999999999999999999999999999 + 1
1000000000000000000000000000000000000

>>> 2**175
47890485652059026823698344598447161988085597568237568
```

Osim u dekadskom brojevnom sustavu, cjelobrojni literali mogu se zapisivati i u heksadekadskom, oktalnom ili binarnom. Heksadekadski zapisani literali započinju prefiksom ”`0x`” ili ”`0X`”, a znamenke veće od 9 (A-F) mogu se zapisati malim ili velikim slovima. Slično, oktalni brojevi započinju prefiksom ”`0o`” ili ”`0O`”, a binarni prefiksom ”`0b`” ili ”`0B`”.

```
>>> 0x10, 0xFF
(16, 255)

>>> 0x10, 0XFF
(16, 255)

>>> 0o10, 0O14
(8, 12)

>>> 0b1010111
87
```

U gornjim primjerima, izrazi odvojeni zarezima čine n-torku (*engl. tuple*), a normalni zapis n-torke uključuje i oble zagrade, pa ih Python tako i ispisuje u interaktivnom načinu rada. N-torke su složeni tip podataka sličan listama.

Ispis cijelih brojeva je uobičajeno dekadski, ali postoje ugrađene funkcije za pretvorbu u heksadekadske, oktalne i binarne *znakovne nizove*.

```
>>> oct(64), hex(64), hex(255), bin(15)
('0o100', '0x40', '0xff', '0b1111')
```

Za obrnutu pretvorbu, iz znakovnog niza koji sadrži broj u broj, koristi se ugrađena funkcija `int`, čiji drugi argument određuje brojevnu bazu. Brojevna baza pritom može biti proizvoljna.

```
>>> int('0100'), int('0100', 8), int('0x40', 16), int('101',3)
(100, 64, 64, 10)
```

Vrijednosti istinitosti

Poseban cjelobrojni tip (`boolean`) koristi se za prikaz vrijednosti istinitosti u logičkim izrazima. Moguće su vrijednosti `True` i `False`.

```
>>> 2 == 1
False
>>> 3 == 3
True
```

Realni brojevi

Realni brojevi u Pythonu pohranjuju se i zapisuju kao i u drugim programskim jezicima, u obliku zapisa s pomičnom točkom. Python pri ispisu racionalne brojeve svodi na odgovarajući oblik. Dovoljno male brojeve prikazuje bez eksponenta:

```
>>> 4.444e11
444400000000.0
```

dok se veći brojevi prikazuje u zapisu s eksponentom:

```
>>> 9999999999999999.0
1e+16
```

Najveća vrijednost realnog broja s kojom Python može baratati je 1.7976931348623157e+308. Ako je premašimo, rezultat je oznaka numeričke pogreške, koju Python zapisuje kao `inf`.

```
>>> 1.798e308
inf
```

Kompleksni brojevi

Kompleksni brojevi su u Pythonu ugrađeni tip podataka. Izvedeni kao parovi brojeva s pomičnim zarezom, pri čemu se imaginarni dio označava dodavanjem sufiksa `J` ili `j`. Npr. kompleksni broj čiji je realni dio `2` i imaginarni dio `-3` zapisuje se kao `2 - 3j`.

```
>>> 1j * 1J
(-1+0j)

>>> x = (2+1j)*3
>>> x
(6+3j)

>>> x.real, x.imag
(6.0, 3.0)
```

Nad kompleksnim brojevima podržani su uobičajeni matematički izrazi, te dodatni alati sadržani u standardnom modulu `cmath` .

Operatori

Nad brojevima je definiran cijeli niz operatora, koji su navedeni u sljedećoj tablici. Prednost operatora raste prema dnu tablice. Redoslijed primjene može se izmijeniti korištenjem zagrada.

<table>
<tr><th>Operatori</th><th>Opis</th></tr>
<tr><td>lambda args: izraz</td><td>neimenovana funkcija</td></tr>
<tr><td>x or y</td><td>y se evaluira ako je x false</td></tr>
<tr><td>x and y</td><td>y se evaluira ako je x true</td></tr>
<tr><td>not x</td><td>logička negacija</td></tr>
<tr><td>x < y, x <= y, x > y,
x >= y, x == y, x != y,
x is y, x is not y,
x in y, x not in y</td><td>usporedbe

provjera identiteta
pripadnost sekvenci</td></tr>
<tr><td>x | y</td><td>ILI</td></tr>
<tr><td>x ^ y</td><td>EX-ILI</td></tr>
<tr><td>x & y</td><td>I</td></tr>
<tr><td>x << y, x >> y</td><td>posmak lijevo, desno</td></tr>
<tr><td>x + y, x - y</td><td></td></tr>
<tr><td>x * y, x % y, x / y, x // y</td><td></td></tr>
<tr><td>-x, +x, ~x, x ** y</td><td></td></tr>
<tr><td>x[i], x[i:j], x.attr, x(...)</td><td></td></tr>
<tr><td>(...), [...], {...}</td><td>n-torka, lista, rječnik,...</td></tr>
</table>

Miješanje numeričkih tipova

Python dozvoljava miješanje numeričkih tipova u izrazima, u čemu se ponaša kao i drugi programski jezici. Operandi u izrazu se pretvaraju u najsloženiji tip koji se pojavljuje u izrazu. Pritom je najjednostavniji tip cjelobrojni, a zatim slijede realni i konačno kompleksni brojevi. Ovo vrijedi samo za numeričke tipove, dok općenito Python ne obavlja (implicitne) pretvorbe tipova. Primjerice, pokušaj pribrajanja znakovnog niza cijelom broju generira pogrešku, osim ako *eksplicitno* obavimo pretvorbu.

Varijable i imena

U Pythonu umjesto pojma varijable radije koristimo izraz *ime*. Ime se stvaraju kada mu se prvi puta dodjeljuje vrijednost.

```
$ python
>>> a = 3          # stvara se ime
>>> b = 4
```

Kada se ime pojavi u izrazu, zamjenjuje se vrijednošću koje mu je pridruženo. Da bismo ime mogli koristiti u izrazu, mora mu biti pridružena vrijednost. Imena se odnose na konkretne objekte i nikada se ne deklariraju unaprijed.

Numerički izrazi

Izraz se dobiva povezivanjem vrijednosti operatorima. Kao vrijednosti mogu se koristiti konstante (literali), imena kojima je prethodno pridružena vrijednost, te podizrazi.

Izraz se izračunava, čime nastaje novi objekt, kojemu se u naredbi pridruživanja može dodijeliti ime, čime on postaje dostupan ostatku programa.

```
>>> a = 3
>>> b = 4
>>>  b * 3, b / 2    # (4*3), (4/2)
(12, 2.0)
```

Izraz koji upišemo u interaktivnom retku, Python izračunava i ispisuje rezultat.

```
>>>  a % 2, b ** 2  # modulo, potencija
(1, 16)

>>>  2 + 4.0, 2.0 ** b   # pretvorba tipova
(6.0, 16.0)
```

Ako se u izrazu koristi ime kojemu nije pridružena vrijednost, Python prijavljuje grešku, a ne koristi neku pretpostavljenu vrijednost.

```
>>> c * 2
Traceback (most recent call last):
  File "<pyshell#49>", line 1, in <module>
    c * 2
NameError: name 'c' is not defined
```

Dijeljenje

Dok su ostale aritmetičke operacije u skladu s običajima, treba istaknuti specifičnosti dijeljenja. Naime, Python definira dvije operacije dijeljenja i dva odgovarajuća operatora. U "tradicionalnim" jezicima uobičajeno je da rezultat dijeljenja ovisi o tipu operanada, Ako su operandi cjelobrojni, onda se obavlja operacija cjelobrojnog dijeljenja i rezultat je cjelobrojan. No, u dinamički tipiziranim jezicima, dakle kada isto ime može biti vezano a različitim tipovima podataka, prosudba o tipu operanada može biti donesena tek pri izvođenju programa.

Stoga su dizajneri Pythona uveli dva operatora. Prvi je "pravo" dijeljenje (*engl. "true division"*), koje kao razultat daje realni broj čak i ako su operandi cijeli brojevi. Tu vlja napomenuti da je u ranijim verzijama Pythona rezultat ovisio o operandima (*engl. "classic division"*)!

```
>>> (5 / 2), (5 / 2.0), (5 / -2.0), (5 / -2), (6 / 3)
(2.5, 2.5, -2.5, -2.5, 2.0)
```

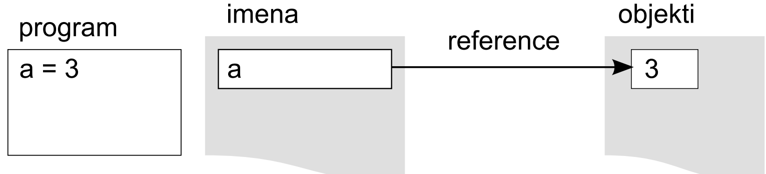
Druga operacija je cjelobrojno dijeljenje (*engl. "floor division"*), kod koje se necijeli dio rezultata odbacuje. Pritom treba biti svjestan da se ne radi zaokruživanja, već funkcija *najveće cijelo*. Razlika se uočava kod negativnih rezultata u primjerima.

```
>>> (5//2),(5 // 2.0),(5 // -2.0),(5 // -2),(6//3)
(2, 2.0, -3.0, -3, 2)
```

Operacije nad bitovima

Python podržava i operacije nad bitovima, kao što su posmak, te operacije I i ILI.

```
>>> x = 0b101  # binarni zapis broja 5
>>> x << 2     # posmak ulijevo za 2 mjesta
20
>>> x | 2      # ILI: 101 | 010 = 111
7
>>> x & 3      # I: 101 & 011 = 001
1
```



Slika 4.1: Stvaranje objekta naredbom dodjeljivanja.

Ostali numerički alati

Osim numeričkih operatora, Python sadrži i ugrađene funkcije, te ugrađene *module* za matematiku (`math`, `cmath`). U primjerima je ilustrirano nekoliko ugrađenih funkcija, te funkcija i konstanti definiranih u modulu `math`.

```
>>> import math
>>> math.pi, math.e
(3.141592653589793, 2.718281828459045)

>>> math.sin(2 * math.pi / 180)
0.03489949670250097

>>> abs(-42), 2**4, pow(2, 4) # ugradjene f-je
(42, 16, 16)

>>> int(2.567), round(2.567), round(2.567, 2)
(2, 3, 2.57)
```

4.3.2 Dinamičko upravljanje tipovima

Već smo u više navrata spominjali dinamičko upravljanje tipovima kao o jednoj od važnih značajki skriptnih jezika, a to vrijedi i za Python. Tipovi podataka određuju se automatski tijekom izvođenja programa, a ne na temelju deklaracija u programskom kodu. Ime se stvara kada mu se u programu prvi put dodijeli vrijednost, dok kasnije dodjele vrijednosti mijenjaju vrijednost pridruženu imenu. Ime ne sadrži informaciju o tipu podatka, već je tip pridružen *objektu*. Ime naprosto pokazuje (referencira) neki objekt. Kada se ime pojavljuje u izrazu, zamjenjuje se objektom na koji pokazuje. Da bi moglo biti upotrijebljeno u izrazu, imenu *mora biti dodijeljena vrijednost*. Na slici 4.1 je ilustrirano stvaranje objekta naredbom dodjeljivanja `>>> a = 3`.

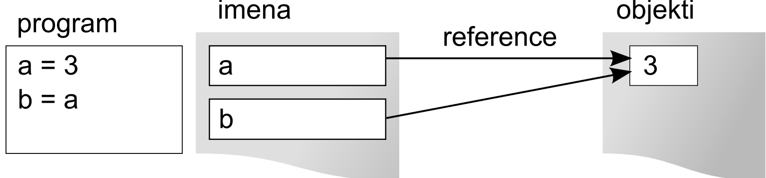
Izraz na desnoj strani naredbe dodjeljivanja stvara objekt, koji se smješta u memoriji. Ime koje se pojavljuje na lijevoj strani upisuje se u tablicu imena. Pritom se stvara i veza između imena i objekta.

Ako kasnije u programu pridružimo ovu varijablu novom imenu, nastaje *dijeljena referenca*, odnosno oba imena referenciraju *isti objekt*.

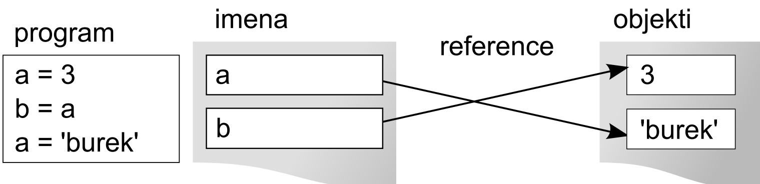
```
>>> a = 3
>>> b = a
```

Ako nakon toga imenu `a` dodijelimo novu vrijednost, reference se razdvajaju.

```
>>> a = 3
>>> b = a
>>> a = 'burek'
```



Slika 4.2: Dijeljene reference.



Slika 4.3: Tip je svojstvo objekta.

U ovom primjeru, stvoren je novi objekt (znakovni niz), s kojim se povezuje ime `a`. Ta dodjela ne utječe na referencu `b` – ona i dalje pokazuje na isti objekt. Uočimo i to da sada ime `a` pokazuje na objekt drugačijeg tipa. To je moguće, jer tip je svojstvo objekta, a ne imena (varijable).

Reference i izmjenjivi objekti

Uočimo da drugo dodjeljivanje vrijednosti imenu `a` ne utječe na objekt na koji ona pokazuje, a time ni na vrijednost varijable `b`. Brojevi, kao i znakovni nizovi s kojima ćemo se uskoro pobliže upoznati, su objekti koji se *ne mogu izmijeniti.* To svojstvo zovemo *nepromjenljivost* (*engl. immutability*).

No, nisu svi objekti u Pythonu takvi. Postoje vrste objekata i operacije nad njima koje uzrokuju izmjenu objekata ”na licu mjesta”. Za sada ćemo ih samo spomenuti, a kasnije ćemo se njima više baviti . Primjerice, neke operacije nad *listama* uzrokuju promjenu njihovih elemenata, a ne stvaranje nove liste. Kod takvih objekata treba paziti na dijeljene reference, jer promjena korištenjem jednog imena utječe i na vrijednost na koju pokazuje drugo ime!

```
>>> L1 = [2,3,4]
>>> L2 = L1
```

U primjeru imena `L1` i `L2` pokazuju na istu listu. Ako imenu `L1` dodijelimo kao novu vrijednost cijeli broj, reference se razdvajaju, a lista ostaje netaknuta.

```
>>> L1 = 24
>>> L2
[2, 3, 4]
```

No, ako preko `L1` dodijelimo novu vrijednost jednom elementu liste, lista se mijenja (i za `L2`).

```
>>> L1 = [2,3,4]
>>> L2 = L1
>>> L1[0] = 24
>>> L2
[24, 3, 4]
```

Reference i prikupljanje otpada

Python ima ugrađeni mehanizam za automatsko upravljanje memorijom, ili kako se to popularno naziva, sustav za zbrinjavanje otpada (*engl. garbage collection*). To znači da se automatski vodi evidencija o aktivnim referencama na pojedini objekt, te ako se pronađe objekt na koji ne pokazuje niti jedna referenca, sustav oslobađa memoriju koju je taj objekt zauzimao i stavlja je na raspolaganje za stvaranje novih objekata.

```
>>> x = 42
>>> x = 'limenka'     # recikliraj 42 (?)
>>> x = 3.1415        # recikliraj 'limenka' (?)
>>> x = [1,2,3]       # recikliraj 3.1415 (?)
```

4.3.3 Znakovni nizovi

Python nema tip podataka koji bi odgovarao pojedinačnom znaku, no za razliku od Perla, svakom znaku znakovnog niza (*engl. string*) može se pojedinačno pristupiti. Znakovni nizovi spadaju u *uređene kolekcije*, jer se sastoje od niza elemenata koji su poredani i mogu se identificirati indeksima. Sekvence su posebna kategorija podatkovnih tipova, a vidjet ćemo da su operacije nad svim tipovima te kategorije slične.

Znakovni nizovi su u Pythonu, kao i brojevi, nepromjenljivi (*engl. immutable*) tip podataka. To znači da, iako se svakom elementu može pristupiti, on se ne može izmijeniti. Možemo jedino na temelju starog niza formirati novi s izmijenjenim sadržajem.

U sljedećoj tablici pobrojane su neke od najčešćih operacija koje se mogu obavljati nad znakovnim nizovima.

Operacija	Opis
`s1 = ''`	prazni niz
`s2 = "spam's"`	dvostruki navodnici
`block = """..."""`	*blok* – može više redaka
`s3 = r'\temp\spam'`	“sirovi” znakovni niz
`s1 + s2 ; s2 * 3`	ulančavanje, ponavljanje
`s2[i] ; s2[i:j] ; len(s2)`	indeksiranje, izrezivanje, duljina
`"a %s parrot" % 'dead'`	formatiranje

`s2.find('pa')`
`s2.replace('pa', 'xx')`
`s1.split( )`
	pozivanje metoda

`for x in s2`	iteriranje
`'m' in s2`	pripadnost

Literali znakovnih nizova

Znakovni niz u Pythonu zapisuje se unutar navodnika. Pritom se mogu koristiti:

- jednostruki navodnici: `'spa"m'`;
- dvostruki navodnici: `"spa'm"`;
- trostruki navodnici: `'''... spam ...'''`, `"""... spam ..."""`.

U smislu interpretacije, jednostruki i dvostruki navodnici su istovjetni (za razliku od Perla ili ljuske `bash`). Naravno, niz se zaključuje istom vrstom navodnika kojom je i započet. Trostruki

navodnici (koji se mogu sastojati od tri jednostruka ili tri dvostruka navodnika) služe za zapisivanje znakovnih nizova koji se protežu kroz više redaka. To je vrlo praktično za zapisivanje dijelova dokumenata koje program treba generirati, a koristi se i za zapisivanje dokumentacijskih komentara (*engl. docstring*).

Kao i u Perlu i ljusci `bash`, moguće je korištenje specijalnih (*escape*) sekvenci za označavanje prelazaka u novi red, tabulatora i slično. Oni se prilikom ispisa znakovnog niza tumače. Jedan takav znakovni niz je `"s\tp\na\0m"`.

Ukoliko želimo spriječiti tumačenje specijalnih sekvenci u znakovnom nizu, niz treba predznačiti slovom ”r”, što označava ”sirovi” niz znakova.

```
r"C:\new\test.spm"    # sirovi znakovni niz
```

Znakovni nizovi u Pythonu 3 pohranjuju se kodirani po standardu Unicode. To znači da se u njima slobodno mogu koristiti i međunarodni znakovi.

```
niz = "Kvačice su u serijskoj opremi: šđžčćŠĐŽČĆ!\n"
print(niz)
```

Naravno, da bi se u datoteci s izvornim kôdom mogli zapisati međunarodni znakovi, i ona mora biti pohranjena u prikladnom kôdu. Podrazumijevano kodiranje datoteke s izvornim kôdom je `UTF-8`.

Već smo ranije spomenuli da su jednostruki i dvostruki navodnici ekvivalentni, odnosno oba oblika djeluju jednako i vraćaju isti tip objekta.

```
>>> 'student',"student"
('student', 'student')
```

Primjena različitih navodnika omogućuje gniježđenje navodnika (druge vrste) unutar znakovnog niza:

```
>>> 'knight"s', "knight's"
('knight"s', "knight's")
```

Python automatski nadovezuje susjedne znakovne nizove literale i bez eksplicitnog korištenja operatora nadovezivanja (+).

```
>>> title = "Meaning " 'of' " Life"
>>> title
'Meaning of Life'
```

Uključivanje navodnika u znakovni niz moguće je i predznačivanjem s ”\”

```
>>> 'knight\'s', "knight\"s"
("knight's", 'knight"s')
```

Posebne sekvence

Kao i u ljusci `bash` i u Perlu, i u Pythonu možemo u znakovne nizove umetati posebne znakove, korištenjem predznačavanja znakom \. Primjerice, u znakovni niz možemo umetnuti oznaku prelaska u novi red i tabulator.

```
>>> s = 'a\nb\tc'
```

Kada koristimo interaktivni ispis u Pythonovoj ljusci, posebni znakovi se ispisuju u obliku posebne (*engl. escape*) sekvence.

```
>>> s
'a\nb\tc'
```

No, ispis funkcijom `print` interpretira specijalne znakove, kao u ovom primjeru.

```
>>> print(s)
a
b       c
```

Duljinu znakovnog niza možemo doznati primjenom funkcije `len`. Primijetimo da, iako je u kôdu zapisan s dva znaka, svaki posebni znak zauzima jedno mjesto u znakovnom nizu.

```
>>> len(s)
5
```

Posebni znakovi i njihove oznake navedeni su u sljedećoj tablici.

Escape	Značenje
`\<novi_red>`	ignorira se – nastavljanje retka
`\\`	Backslash
`\'`	jednostruki navodnik
`\"`	dvostruki navodnik
`\a`	Bell
`\b`	Backspace
`\f`	Formfeed
`\n`	Newline (linefeed)
`\r`	Carriage return
`\t`	Horizontal tab
`\v`	Vertical tab
`\N{id}`	Unicode dbase id
`\uhhhh`	Unicode 16-bit hex
`\Uhhhh...`	Unicode 32-bit hex
`\xhh`	heksadekadska vrijednost hh
`\ooo`	oktalna vrijednost oo
`\0`	Null (ne označava kraj niza)
`\other`	nije escape (zadržava se i ”\”)

Uz uobičajene oznake, navedene u gornjem dijelu tablice, moguće je i zadavanje znakova iz Unicode tablice. Oznaka `\N` omogućuje zadavanje znaka njegovom simboličkom oznakom. Primjerice, ”`ASTERISK`” označava zvjezdicu. Moguće je zadavanje 16-bitnog ili 32-bitnog heksadekadskog Unicode zapisa, dok se 8-bitnom heksadekadskom ili oktalnom vrijednošću mogu zadati ASCII znakovi. Ukoliko se iza znaka ”\” nađe neki znak koji ne predstavlja početak posebne sekvence, on se naprosto zanemaruje.

```
>>> print("\x41 \u0041 \U00000041 \N{ASTERISK}")
A A A *

>>> print("\xF4 \u00A5 \N{YEN SIGN}")
ô ¥ ¥

>>> print("ovo se ne \zamjenjuje")
ovo se ne \zamjenjuje
```

Kao što smo već spomenuli, interpretacija posebnih sekvenci može se spriječiti korištenjem ”sirovih” znakovnih nizova, koji se predznačavaju slovom ”`r`” ili ”`R`”. To je korisno, primjerice, kod navođenja puta do datoteke pod Windowsima.

```
ne_valja = open('C:\new\text.dat', 'w')    # ovo nije dobro

bolje = open(r'C:\new\text.dat', 'w')      # sirovi string

moze = open('C:\\new\\text.dat', 'w')      # može i ovako
```

U interaktivnom ispisu, Python ispisuje svoju reprezentaciju znakovnog niza, bez interpretiranja posebnih sekvenci.

```
>>> path = r'C:\new\text.dat'
>>> path          # Show as Python code.
'C:\\new\\text.dat'

>>> print(path)   # "User-friendly" format
C:\new\text.dat
>>> len(path)
15
```

Trostruki navodnici

Python podržava zapisivanje znakovnih nizova koji se protežu kroz više redaka, a oni se navodeunutar trostrukih navodnika. Blok započinje s tri navodnika (jednostruka ili dvostruka), slijedi proizvoljan broj redaka teksta, a završetak bloka označava pojava tri navodnika (jednaka početnim). Unutar bloka mogu se pojaviti i navodnici i oni ne moraju biti predznačeni s "\".

```
>>> mantra = """Always look
...  on the bright
... side of life."""
>>>
>>> mantra
'Always look\n on the bright\nside of life.'!
```

Python prikuplja sve retke ovako definiranog bloka teksta i pakira ga u jedan niz, pri čemu se prelasci u novi red kodiraju kao "\n".

Operacije sa znakovnim nizovima

Nad znakovnim nizovima mogu se obavljati različite operacije. Primjerice, možemo saznati njegovu duljinu primjenom ugrađene funkcije `len`.

```
>>> len('abc')
3
```

Ulančavanje se označava operatorom "+", čiji smo polimorfizam već komentirali.

```
>>> 'abc' + 'def'
'abcdef'
```

Ponavljanje se zadaje operatorom "*" uz navođenje cjelobrojnog broja ponavljanja znakovnog niza.

```
>>> 'bla ' * 4
'bla bla bla bla '

>>> print('-'*80)  # primjer korisne uporabe :-)
```

Ulančavanjem dva znakovna niza stvara se novi niz, koji se može pridružiti nekom imenu kako bismo ga kasnije mogli koristiti. Za razliku od C-a, u Pythonu nije potrebno alocirati memoriju za smještaj novonastalog objekta i voditi računa o veličini polja u koje se niz pohranjuje.

Podsjetimo, Python je strogo tipizirani jezik i ne podržava automatske pretvorbe tipova, pa treba paziti na njihovo poštivanje.

Kroz znakovne nizove se može prolaziti (*engl. iterirati*) u petlji `for`, uz provjeru pripadnosti operatorom `in`. U slučaju petlje, upravljačka varijabla u svakom prolazu pokazuje na sljedeći znak niza. Operator za `in` ispituje je li lijevo navedeni niz podniz desno navedenoga.

```
>>> myjob = "hacker"
>>> for c in myjob: print(c, end=' ')  # iteriranje
...
h a c k e r
>>> "k" in myjob
True
>>> "ack" in myjob   # moze i podniz
True
>>> "z" in myjob     # False = (nije nadjen)
False
```

Indeksiranje i izrezivanje

Znakovni niz u Pythonu je definiran kao *uređena kolekcija* znakova, te se svakom znaku može pristupiti na temelju njegovog indeksa. Pritom se kao rezultat operacije indeksiranja dobiva *niz* duljine 1 znak. Sjetimo se, za razliku od C-a, Python nema poseban tip koji bi odgovarao pojedinačnom znaku. Kao što je to uobičajeno u programskim jezicima, indeksi kreću od `0`, a može se navesti i negativan indeks. Kao što je to bilo kod Perla, i u Pythonu negativan indeks označava brojanje elemenata od kraja niza. U tom slučaju pravi indeks elementa odgovara pribrajanju negativnog indeksa duljini niza.

```
>>> S = 'spam'
>>> S[0], S[-2]      # indeksiranje od početka ili kraja
('s', 'a')
```

Python nudi i operaciju izrezivanja (*engl. slicing*), kojom se stvara novi znakovni niz koji sadrži podniz izvornog niza. Navodi se indeks prvog elementa podniza, zatim indeks prvog elementa kojega ne želimo uključiti u rezultat, a može se navesti i treći parametar koji predstavlja korak izrezivanja. Parametri se odvajaju dvotočkom, a ako se neki ne navede, koriste se podrazumijevane vrijednosti. To su `0` kao prvi, odnosno duljina niza kao drugi indeks.

```
>>> S[1:3], S[1:], S[:-1]   # izrezivanje
('pa', 'pam', 'spa')

>>> kopija = S[:]           # jednostavan način kopiranja niza
>>> kopija
'spam'
```

Operacija izrezivanja i indeksi koji se navode ilustrirani su na slici 4.4.

Pretvorbe znakovnih nizova

Već smo spominjali da je Python strogo tipizirani jezik. Stoga se ne dozvoljava zbrajanje broja sa znakovnim nizom, čak ni ako niz sliči na broj (sjetimo se Perla i njegove fleksibilnosti u

rez = niz[3:8]

početak kraj

niz z n a k o v n i   n i z

0 1 2 3 4 5 6 7 8 9 10 11

rez k o v n i

Slika 4.4: Operacija izrezivanja.

takvim situacijama). Jasno je i zašto je tako: operator "+" je polomorfan, tj. može označavati zbrajanje ili ulančavanje, izbor pretvorbe bio bi *dvosmislen*. Nije jasno treba li broj pretvoriti u znakovni niz ili obrnuto. Kod Perla su korišteni različiti operatori, pa je operator određivao prikladnu pretvorbu.

Python programeru nudi mogućnost eksplicitne pretvorbe znakovnih nizova u brojeve i obrnuto, korištenjem ugrađenih funkcija `int`, `float` i `str`.

```
>>> int("42")     # pretvara znakovni niz u cijeli broj
42
>>> str(42)      # cijeli broj u znakovni niz
'42'
>>> int("42") + 1   # ovako ih možemo zbrojiti
43
>>> "432" + str(10)  # ili nadovezati
'43210'
>>> str(3.1415)   # vrijedi i za realne brojeve
'3.1415'
>>> float("1.234e-1")  # kao i obrnuto
0.1234
```

Neizmjenljivost znakovnih nizova

U Pythonu, znakovni nizovi su neizmjenljivi objekti. To znači da se jednom stvoren znakovni niz ne može izmijeniti. Jedino se može stvoriti novi objekt s izmijenjenim sadržajem izvornog znakovnog niza. Ukoliko pokušamo izmijeniti neki element znakovnog niza, dogodit će se iznimka.

```
>>> S = 'spam'
>>> S[0] = "x"  # probajmo izmijeniti
...
TypeError: 'str' object does not support item assignment
```

Kako bismo izmijenili znakovni niz, moramo kreirati novi niz s izmijenjenim sadržajem, uz eventualno pridjeljivanje imenu izvornog stringa

```
>>> S = S + 'SPAM!'  # kreira se novi string!
>>> S
'spamSPAM!'
>>> S = S[:4] + 'Burger' + S[-1]
>>> S
'spamBurger!'
```

Metode znakovnih nizova

Znakovni niz kao objekt ima skup metoda – funkcija koje su atributi objekta, a koje tipično implementiraju razne zadatke obrade teksta. Poziv metode objedinjuje dvije operacije: dohvat atributa i poziv funkcije. Izraz oblika `objekt.atribut` dohvaća vrijednost atributa u objektu, dok je izraz oblika `funkcija(argumenti)` poziv funkcije uz prosljeđivanje argumenata. Izraz oblika `objekt.funkcija(argumenti)` poziva metodu.

Metoda `replace` obavlja zamjenu podniza u nizu iz kojega se poziva drugim podnizom.

```
>>> S = 'spammy'
>>> S = S.replace('mm', 'xx')
>>> S
'spaxxy'
```

U primjeru je podniz `"mm"` zamijenjen nizom `"xx"`. Uočite da je pozivom metode stvoren *novi* znakovni niz, koji je zatim pridijeljen imenu koje je pokazivalo na izvorni niz. Ako ne bismo obavili dodjeljivanje, niz bi ostao neizmijenjen!

```
>>> S = 'spammy'
>>> S.replace('mm', 'xx')  # ovo nije dobro!
'spaxxy'
>>> S
'spammy'
```

Treba napomenuti da metoda, pozvana na ovaj način, obavlja *globalnu* zamjenu, tj. zamjenjuje sve pojave navedenog podniza.

```
>>> 'aa$bb$cc$dd'.replace('$', 'SPAM')
'aaSPAMbbSPAMccSPAMdd'
```

Broj zamjena može se i ograničiti, zadavanjem još jednog (trećeg) argumenta pri pozivu metode `replace`.

```
>>> S = 'xxxxSPAMyyyySPAMzzzzSPAMwwww'
>>> S.replace('SPAM', 'EGGS', 1)   # jedna zamjena
'xxxxEGGSyyyySPAMzzzzSPAMwwww'
>>> S.replace('SPAM', 'EGGS', 2)   # dvije zamjene
'xxxxEGGSyyyyEGGSzzzzSPAMwwww'
```

Metoda `find` vraća indeks na kojem započinje traženi podniz, ili `-1` ako podniz nije pronađen.

```
>>> S = 'xxxxSPAMyyyySPAMzzzzSPAMwwww'
>>> where = S.find('SPAM') # traži položaj (indeks)
>>> where
4
>>> S = S[:where] + 'EGGS' + S[(where+4):]
>>> S
'xxxxEGGSyyyySPAMzzzzSPAMwwww'
```

Zbog neizmjenljivosti znakovnih nizova, pri svakoj manipulaciji s njima nužno je stvaranje međukopija. U slučaju većeg broja uzastopnih operacija nad dugačkim znakovnim nizovima, dolazi ne samo do povećane potrošnje memorijskog prostora, već kopiranje dijelova troši i procesorsko vrijeme. Stoga je u nekom slučajevima povoljnije sadržaj znakovnog niza kopirati u izmjenljivi objekt, operacije obaviti na tom objektu i konačno rezultat pretvoriti u znakovni niz. Prikladan izmjenljivi objekt je lista, a stvaranje liste čiji su elementi znakovi iz znakovnog niza postiže se ugrađenom funkcijom `list` uz zadavanje znakovnog niza kao argumenta.

```
>>> S = 'spammy'
>>> L = list(S)
>>> L
['s', 'p', 'a', 'm', 'm', 'y']
>>>
>>> L[3] = 'x'  # radi s listama, ne sa stringovima
>>> L[4] = 'x'
>>> L
['s', 'p', 'a', 'x', 'x', 'y']
```

Nakon obavljenih izmjena, listu možemo ”spakirati” nazad u znakovni niz primjenom metode `join`.

```
>>> S = ''.join(L)   # "uvlači" listu u string
>>> S
'spaxxy'
```

Metoda znakovnih nizova `join` poziva se preko željenog graničnika, koji se umeće između pojedinih elemenata liste koji se povezuju u znakovni niz. Kao graničnik može se koristiti proizvoljan znakovni niz. Elementi liste, naravno, moraju biti znakovni nizovi, inače će doći do iznimke.

```
>>> 'SPAM'.join(['eggs', 'sausage', 'ham', 'toast'])
'eggsSPAMsausageSPAMhamSPAMtoast'
```

Osim spajanja znakovnog niza od dijelova, često je zanimljivo i razlomiti znakovni niz na dijelove, primjerice kada se radi o podacima učitanim iz datoteke ili sa standardnog ulaza. U nekim slučajevima, širine polja s podacima su unaprijed poznate, pa ih je lako izrezati.

```
>>> line = 'aaa bbb ccc'
>>> podatak1 = line[0:3]
>>> podatak3 = line[8:]
>>> podatak1, podatak3
('aaa', 'ccc')
```

Međutim, u drugim slučajevima polja nisu fiksne širine, već su odvojeni posebnim znakovima koji služe kao graničnici. Često se kao graničnik koristi zarez (”,”) ili neki drugi znak interpunkcije. Takav oblik zapisa koristi se i za zapisivanje podataka u tekstualnu datoteku, a naziva se `CSV` (*Comma Separated Values*). Za odvajanje podataka u tom je slučaju prikladna metoda znakovnih nizova `split`. Medoda se poziva iz niza kojega želimo razlomiti, a kao argument se navodi znakovni niz koji se koristi kao graničnik za dijeljenje. Podrazumijevani graničnik su praznine (jedan ili više znakova razmaka, tabulatora, prelazaka u novi red). Rezultat je lista podnizova.

```
>>> line = 'aaa bbb   ccc'
>>> cols = line.split(  )   # podrazumijeva se lomljenje na razmacima
>>> cols
['aaa', 'bbb', 'ccc']

>>> line = 'bob,hacker,40'
>>> line.split(',')         # može se zadati graničnik
['bob', 'hacker', '40']

>>> line = "i'mSPAMaSPAMlumberjack"
>>> line.split("SPAM")      # graničnik može biti i znakovni niz
```

```
["i'm", 'a', 'lumberjack']
```

Osnovne operacije nad tekstom mogu se obaviti, i to vrlo učinkovito, korištenjem navedenih metoda i operacije izrezivanja. Postoje i druge metode definirane nad znakovnim nizovima, ali ih ovdje nećemo obrađivati.

Valja napomenuti da metode znakovnih nizova ne podržavaju regularne izraze, već je za tu namjenu potrebno koristiti standardni modul `re`.

Formatiranje znakovnih nizova

U ranijim verzijama Pythona, formatiranje znakovnih nizova obavljalo se primjenom binarnog operatora "`%`". Taj način formatiranja moguće je koristiti i u Pythonu 3, no danas se preporučuje primjena metode znakovnih nizova `format`. Metoda se poziva iz formatnog niza, koji je drugačijeg oblika nego kod "klasičnog" operatora formatiranja, odnosno C funkcije `printf`.

Mjesta na koja se umeću podaci označavaju se vitičastim zagradama, a unutar zagrada može se navesti i redni broj podatka koji se na to mjesto umeće ili ime argumenta. To možemo ilustrirati sljedećim primjerom.

```
>>> a = 2
>>> b = 3
>>> print(  # promiješan redoslijed
           "Zbroj brojeva {1} i {2} je {0}".format(a + b, a, b))
Zbroj brojeva 2 i 3 je 5
```

Ako se argumenti umeću redom navođenja, indeksi nisu obavezni.

```
>>> print(  # bez miješanja
           "Zbroj brojeva {} i {} je {}".format(a, b, a + b))
Zbroj brojeva 2 i 3 je 5
```

Drugi način navođenja argumenata je pomoću imena. Pritom se imena koriste u formatnom nizu, ali se moraju odgovarajuće navesti i u listi argumenata.

```
>>> "{umanjenik} - {umanjitelj} = {razlika}".format(
                            razlika=a-b, umanjitelj=b, umanjenik=a)
'2 - 3 = -1'
```

Metoda `format` omogućuje upravljanje širinom polja i poravnanjem podataka. Pritom sintaksa samo donekle podsjeća na sintaksu funkcije `printf`. Oznaka se navodi unutar vitičastih zagrada, nakon oznake argumenta, odvojena dvotočkom.

```
>>> print("|{0:10s}|{1:10d}|{2:10.2f}|".format(
                       "Znakovi",123,543.1234))
|Znakovi   |       123|    543.12|
```

Prilikom formatiranja znakovnog niza, moguće je odrediti i poravnjanje pojedinih polja. Podrazumijevano poravnanje ovisi o vrsti podatka. Za znakovne nizove podrazumijeva se lijevo, a za brojeve desno poravnanje. Ako to ponašanje želimo izmijeniti, upotrijebit ćemo oznaku za lijevo poravnanje ("<"), desno poravnanje (">") ili za centriranje ("^").

```
>>> print("|{:>10s}|{:^10d}|{:<10.2f}|".format(
                       "Znakovi",123,543.1234))
|   Znakovi|   123    |543.12    |
```

Operator formatiranja

Kao što smo već spomenuli, za formatiranje znakovnih nizova može se upotrijebiti i operator "%". On je polimorfan, odnosno njegova uloga ovisi o operandima. Ako se primijeni s brojevima, daje ostatak pri dijeljenju, a ako je lijevi operand znakovni niz, podrazumijeva se operacija formatiranja. Lijevo od operatora "%" navodi se formatni niz u kojem se definiraju pretvorbe podataka, a na desnoj strani se navodi objekt koji želimo umetnuti u konačni znakovni niz. Ako formatiranje želimo primijeniti nad više objekata, nužno ih je "spakirati" u (*n-torku*), odnosno navesti unutar zagrada. Broj argumenata u n-torki mora odgovarati broju oznaka umetanja u formatnom nizu.

```
>>> ime = "Fiona"
>>> "The knight loves %s." % ime
'The knight loves Fiona.'

>>> "%d %s %d you" % (1, 'spam', 4)
'1 spam 4 you'
```

Tehnički gledano, isti rezultat možemo ostvariti ulančavanjima i pretvorbama, no formatiranje mogućava obavljanje većeg broja koraka kroz jednu operaciju.

Svaki tip objekta može se pretvoriti u znakovni niz (oznaka formata `%s`.

```
>>> "%s -- %s -- %s" % (42, 3.14159, [1, 2, 3])
'42 -- 3.14159 -- [1, 2, 3]'
```

Kao i kod C funkcije `printf`, na raspolaganju nam je cijeli niz formatnih oznaka, a navedene su u sljedećoj tablici.

Oznaka	Opis
`%s`	znakovni niz, ili bilo koji objekt
`%r`	kao `%s`, ali koristi `repr( )`, a ne `str( )`
`%c`	znak
`%d`	dekadski cijeli broj
`%i`	cijeli broj
`%u`	cijeli broj bez predznaka
`%o`	oktalni cijeli broj
`%x`	heksadekadski cijeli broj
`%X`	heksadekadski cijeli broj, zapis velikim slovima
`%e`	eksponencijalni zapis broja s pomičnom točkom
`%E`	kao `%e` ali veliki E
`%f`	floating-point decimal
`%g`	floating-point `%e` ili `%f`
`%G`	floating-point `%E` ili `%f`
`%%`	doslovno "%"

U nastavku je prikazano još nekoliko primjera formatiranja.

```
>>> x = 1234
>>> res = "integers: |%d|%6d|%-6d|%06d|" % (x,x,x,x)
>>> res
'integers: |1234|  1234|1234  |001234|'
```

Brojevi s pomičnom točkom mogu se formatirati na različite načine.

```
>>> x = 1.23456789
>>> '%e | %f | %g' % (x, x, x)
'1.234568e+00 | 1.234568 | 1.23457'

>>> '%-6.2f | %05.2f | %+06.1f' % (x, x, x)
'1.23   | 01.23 | +001.2'

>>> "%s" % x, str(x)
('1.23456789', '1.23456789')
```

4.3.4 Nizovi binarnih znakova

U Pythonu 3 znakovni nizovi su nizovi Unicode znakova, što znači da je pojedini znak kodiran s više bajtova. Zbog toga znakovni nizovi, za razliku od Pythona 2 ili Perla, nisu prikladni za pohranu binarnih podataka! Za prikaz binarnih podataka uveden je novi ugrađeni podatkovni tip `bytes`, u kojemu je svaki element niza jedan bajt. Niz bajtova kao literal prefiksira se oznakom ”b”

```
>>> bajtovi = b'abc\x00\x41\x42\x43'  # ASCII i hexa
>>> bajtovi
b'abc\x00ABC'

>>> bajtovi = b'čičak'  # ali ne i non-ASCII
SyntaxError: bytes can only contain ASCII literal
characters.
```

Da bismo znakovni niz pretvorili u niz bajtova, moramo ga kodirati, pozivom metode znakovnih nizova lstinline!encode!. Pritom je, naravno, nužno zadati shemu kodiranja. U sljedećim primjerima ilustrirano je što će se dogoditi prilikom kodiranja znakovnog niza koji sadrži znakove izvan ASCII raspona. Shema kodiranja UTF-8 prikazuje hrvatske dijakritičke znakove 16-bitnim kombinacijama (npr. ”ć” je kodiran kao `0xC487`), dok se u Windows-1250 taj isti znak kodira 8-bitno, kao `0xE6`. Naravno, ako ovakav znakovni niz pokušamo kodirati kao ASCII, dogodit će se iznimka jer taj kôd ne poznaje znakove ”č” i ”ć”.

```
>>> znakovi = "čekić"
>>> bajtoviUTF8 = znakovi.encode('utf-8')
>>> bajtoviUTF8  # č je 0xC48D
b'\xc4\x8deki\xc4\x87'

>>> bajtoviWin = znakovi.encode('windows-1250')
>>> bajtoviWin   # č je 0xE8, ć je 0xE6
b'\xe8eki\xe6'

>>> znakovi.encode('ascii')  # ovo neće ići
...
UnicodeEncodeError: 'ascii' codec can't encode
character '\u010d' in position 0: ordinal not
in range(128)

>>> "ASCII proba".encode('ascii')  # ovo je OK
b'ASCII proba'
```

Obratno, pri pretvorbi niza bajtova u znakovni niz, potrebno je obaviti dekodiranje, pozivom metode nizova bajtova decode.

```
>>> bajtoviUTF8.decode('utf-8')
'čekić'

>>> bajtoviWin.decode('windows-1250')
'čekić'

>>> bajtoviUTF8.decode('ascii')  # ovo neće ići
...
UnicodeDecodeError: 'ascii' codec can't decode
byte 0xc4 in position 0: ordinal not in range(128)

>>> b'Skriptni jezici'.decode('ascii')  # OK
'Skriptni jezici'
```

4.3.5 Liste

Liste (ugrađeni tip list) su fleksibilne uređene kolekcije. Za razliku od znakovnih nizova, liste mogu sadržavati bilo koju vrstu objekata, pa i druge liste. Liste mogu biti heterogene, odnosno pojedini elementi mogu biti različitih tipova. Kako se radi o uređenoj kolekciji, elementima se pristupa putem indeksa, a podržane su i operacije izrezivanja i nadovezivanja koje smo upoznali kod znakovnih nizova. Duljina liste se može mijenjati umetanjem i brisanjem elemenata.

Liste u Pythonu sadrže *reference* na objekte, slično polju pokazivača u C-u. Zapravo su liste i ostvarene kao C polja unutar, pa je dohvat elementa liste u Pythonu brz gotovo kao indeksiranje polja u C-u. Dakle, kada se umeće objekt u listu, pohranjuje se referenca, a ne kopija objekta.

Operacije nad listama

Nad listama je definiran niz operacija, a važnije su prikazane u sljedećoj tablici. Prikazani su i primjera literala listi.

Operacija	Opis
L1 = [ ]	prazna lista
L2 = [0, 1, 2, 3]	četiri elementa, indeksi 0..3
L3 = ['abc', ['def', 'ghi']]	ugniježdena podlista
L2[i] ; L3[i][j]	indeksiranje
L2[i:j] ; len(L2)	izrezivanje, određivanje duljine
L1 + L2 ; L2 * 3	ulančavanje, ponavljanje
for x in L2 ; 3 in L2	iteriranje, pripadnost
L2.append(4) L2.extend([5,6,7])	metode: dodavanje elemenata
L2.sort( ) ; L2.index(1) L2.reverse( )	sortiranje, pretraživanje,...
del L2[k] ; del L2[i:j] ; L2.pop( ) ; L2[i:j] = [ ]	brisanje
L2[i] = 1 ; L2[i:j] = [4,5,6]	pridruživanje vrijednosti

Na liste se može primjeniti operacije nadovezivanja ("+") i ponavljanja ("*"), slično kao kod znakovnih nizova, a kao rezultat dobiva se nova lista.

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

>>> [1, 2, 3] + [4, 5, 6]
[1, 2, 3, 4, 5, 6]

>>> ['Ni!'] * 4
['Ni!', 'Ni!', 'Ni!', 'Ni!']

>>> 3 in [1, 2, 3] # pripadnost
True

>>> for x in [1, 2, 3]: print(x, end=' ')
...
1 2 3
```

Iako je ulančavanja definirano i za liste i za znakovne nizove, nije dozvoljeno miješanje tipova. Očekuje se isti tip sekvence na obje strane operatora, a u suprotnom se generira iznimka. Ulančavanje liste i znakovnog niza možemo postići tako da jedan od podataka pretvorimo o drugi tip, primjenom ugrađenih funkcija `str` ili `list`. Naravno, i rezultat će biti istoga tipa.

```
>>> L=[1, 2]
>>> L + "34"
...
TypeError: can only concatenate list
(not "str") to list

>>> str(L) + "34"   # kao "[1, 2]" + "34"
'[1, 2]34'

>>> L + list("34")  # kao [1, 2] + ["3", "4"]
[1, 2, '3', '4']
```

Indeksiranje i izrezivanje

Indeksiranje i izrezivanje su također primjenjivi i na liste. Pritom indeksiranje vraća (kopiju) objekta koji se nalazi u listi na mjestu na koje indeks pokazuje. Međutim, spomenuli smo da su u listi pohranjene reference, a ne sami objekti, pa treba biti svjestan da se i indeksiranjem zapravo vraća referenca na objekt! Izrezivanjem se stvara nova lista, koja sadrži kopije referenci iz izvorne liste.

```
>>> L = ['spam', 'Spam', 'SPAM!']

>>> L[2]      # indeksi kreću od 0
'SPAM!'

>>> L[-2]     # negativni indeks - odbrojavanje od kraja
'Spam'

>>> L[1:]    # izrezivanje
['Spam', 'SPAM!']
```

Matrice

Liste se mogu primijeniti za prikaz matrica, odnosno višedimenzionalnih polja. To se postiže gniježđenjem listi. Primjerice, matricu dimenzija 3×3 možemo zapisati kao u primjeru.

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

Kada pristupamo ovoj reprezentaciji matrice, primjenom jednog indeksa dobivamo cijeli redak matrice (ugniježđenu listu), a s dva indeksa dohvaćamo pojedini element.

```
>>> matrica[1]   # redak s indeksom 1
[4, 5, 6]

>>> matrica[1][1], matrica[2][0]   # pojedinačni elementi
(5, 7)
```

U programu ovakav literal možemo zapisati i preglednije, protezanjem liste kroz više redaka.

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

Izmjenljivost listi

Tipovi objekata koje smo do sada susretali (brojevi, znakovni nizovi i nizovi bajtova) bili su *neizmjenljivi*. Liste, s druge strane, pripadaju *izmjenljivim* objektima, što znači da se mogu mijenjati na licu mjesta, bez stvaranja kopije. Kako Python radi s referencama, izmjena objekta utječe i na druge reference na taj objekt, pa o tome uvijek treba voditi računa!

Sadržaj liste može se izmijeniti pridjeljivanjem vrijednosti pojedinom elementu ili cijelom isječku (*engl. slice*).

```
>>> L = ['spam', 'Spam', 'SPAM!']
>>> L[1] = 'eggs' # Index assignment
>>> L
['spam', 'eggs', 'SPAM!']

>>> L[0:2] = ['eat', 'more'] # Slice assignment
>>> L
['eat', 'more', 'SPAM!']
```

Pridjeljivanje vrijednosti isječku ponaša se *kao* brisanje i umetanje. Pritom broj elemenata koji se umeće ne mora odgovarati broju elemenata koji se brišu.

```
>>> L = [1, 2, 3]
>>> L[1:2]=[4,5]
>>> L
[1, 4, 5, 3]

>>> L[1:3]=[ ] # brisanje elemenata
>>> L
[1, 3]
```

Ova operacija funkcionira i u slučaju preklapanja isječka s vrijednostima koje se umeću s isječkom kojemu se dodjeljuju vrijednosti.

```
>>> L = [1, 2, 3, 4, 5, 6, 7, 8]
>>> L[2:5]=L[3:6]
>>> L
[1, 2, 4, 5, 6, 6, 7, 8]
```

Metode listi

Liste imaju niz specifičnih metoda, koje služe za proširivanje liste novim elementima, uzimanje elementa s kraja liste, mijenjanje redoslijeda elemenata i slično.

Za dodavanje *jednog* elementa na kraj liste koristi se metoda append.

```
>>> L = ['eat', 'more', 'SPAM!']
>>> L.append('please')  # dodavanje jednog elementa
>>> L
['eat', 'more', 'SPAM!', 'please']
```

Efekt izraza `L.append(X)` je sličan kao `L+[X]`, no drugi izraz stvara *novu* listu. Stoga je append obično brža operacija. Treba paziti na to da append mijenja listu na licu mjesta, a *ne vraća listu kao rezultat.* To znači da će naredba
`L=L.append(X)`
obrisati referencu na listu, odnosno toj listi više nećemo moći pristupiti (osim ako nekim slučajem nemamo još neku kopiju te reference).

```
>>> L = L.append('X')  # PAZI! vraća None
>>> L  # objekt None se ne ispisuje

>>> repr(L)  # eksplicitno tražim reprezentaciju
'None'
```

Metoda `extend` ima sličnu namjenu, uz razliku da ona kao argument prima listu elemenata koji se dodaju na kraj liste koja se proširuje. I ona obavlja izmjenu liste na licu mjesta, bez vraćanja reference.

```
>>> L = [1, 2]
>>> L.extend([3,4,5])    # dodavanje liste elemenata
>>> L
[1, 2, 3, 4, 5]
```

Metoda pop uzima zadnji element liste, briše ga i vraća kao povratnu vrijednost.

```
>>> L.pop( )  # briše i vraća zadnji element
5
>>> L
[1, 2, 3, 4]
```

Korištenjem metoda pop i append lako je implementirati brzu stogovnu strukturu. Pritom je kraj liste vrh stoga.

```
>>> L = [ ]
>>> L.append(1)   # Push
>>> L.append(2)
>>> L
[1, 2]
>>> L.pop( )     # Pop
2
```

```
>>> L
[1]
```

Metoda pop kao argument može primiti i indeks. Na taj se način može uzeti i obrisati proizvoljni element liste.

```
>>> L = [1, 2, 3, 4, 5]
>>> L.pop(3)     # ovo je indeks elementa
4
>>> L
[1, 2, 3, 5]
```

Sortiranje

Liste imaju i metode za promjenu redoslijeda elemenata. Prva je `reverse`, koja okreće redoslijed elemenata. I ova metoda mijenja listu na licu mjesta.

```
>>> L.reverse(  )
>>> L
[4, 3, 2, 1]
```

Druga metoda, `sort`, obavlja sortiranje liste. I sortiranje se obavlja na licu mjesta, a povratna vrijednost je objekt `None`!

```
>>> L = [21, 14, 77, 11, 7]
>>> L.sort()
>>> L
[7, 11, 14, 21, 77]


>>> L = ["Pero", "Marko", "Zdenka", "Matej"]
>>> L.sort()
>>> L
['Marko', 'Matej', 'Pero', 'Zdenka']
```

Da bi sortiranje bilo moguće, elementi liste moraju biti međusobno usporedivi, inače se generira iznimka.

```
>>> L = ['Pero', 77, 'Marko', 123, 'Matej']
>>> L.sort()
...
TypeError: unorderable types: int() < str()
```

Sortiranje se može postići i korištenjem ugrađene funkcije `sorted`, koja kao argument prima pobrojivi objekt (*engl. iterable*), na temelju kojega konstruira *novu* sortiranu listu. Pobrojivi objekt na koji se funkcija primjenjuje može biti lista, rječnik, n-torke, skup i slično. U sljedećem primjeru, argument je lista.

```
>>> # velika slova prije malih
>>> sorted(["Pero", "marko", "zdenka", "Matej"])
['Matej', 'Pero', 'marko', 'zdenka']
```

Funkcija `sorted` prima i dodatni, imenovani argument `key`, koji određuje funkciju koja se poziva za svaki element prije usporedbe. I metoda `sort` podržava istu mogućnost. Primjer ilustrira primjenu metode znakovnih nizova `str.lower` koja prebacuje znakove niza u mala slova. Na taj način sortiranje se obavlja uz zanemarivanje razlike između malih i velikih slova.

```
>>> # pretvorba u mala slova
>>> sorted(["Pero", "marko", "zdenka", "Matej"],
            key=str.lower)
['marko', 'Matej', 'Pero', 'zdenka']
```

Kodiranje znakovnih nizova po standardu Unicode omogućuje lokalizaciju sortiranja. Naravno, pritom je potrebno zadati jezik, čime je definiran i redoslijed sortiranja lokalnih znakova. Podrazumijevani redoslijed sortiranja smješta hrvatske znakove iza ostatka abecede.

```
>>> L = ['Marijana', 'Đurđa', 'Željka', 'Štefica']
>>> L.sort()    # ovo nije baš dobro?
>>> L
['Marijana', 'Đurđa', 'Štefica', 'Željka']
```

Za lokalizaciju koristimo modul `locale`. Uz pretpostavku da je operacijski sustav koji koristimo lokaliziran, odnosno da su postavke prilagođene hrvatskom, dovoljno je pozvati funkciju `locale.setlocale` kao u sljedećem primjeru. Nakon toga, pri pozivu funkcije `sorted` treba navesti funkciju usporedbe `locale.strxfrm`.

```
>>> import locale  # računamo da je sustav podešen
>>> locale.setlocale(locale.LC_ALL, '')
'Croatian_Croatia.1250'

>>> sorted(L,key=locale.strxfrm)  # sad je bolje
['Đurđa', 'Marijana', 'Štefica', 'Željka']
```

Brisanje elemenata liste

Za brisanje elementa ili dijela liste može se primijeniti naredba `del`. Ona se općenito koristi za brisanje objekta (bolje rečeno reference na objet) koji se navodi kao argument.

```
>>> L = ['SPAM!', 'eat', 'more', 'please']
>>> del L[0]   # brisanje jednog elementa
>>> L
['eat', 'more', 'please']

>>> del L[1:]   # brisanje isječka
>>> L
['eat']
```

Dijelovi liste mogu se obrisati i operacijom pridruživanja isječku.

```
>>> L = ['Already', 'got', 'one']
>>> L[1:] = [  ]
>>> L
['Already']
```

Ovdje valja napomenuti i to da pridjeljivanje prazne liste indeksiranjem ne briše element, već u njega pohranjuje referencu na praznu listu!

```
>>> L = ['još', 'jedna', 'lista']
>>> L[1] = []   # ovo ne briše element 1
>>> L
['još', [], 'lista']
```

Izrazi za stvaranje liste

Python ima mogućnost zadavanja izraza za stvaranje liste (*engl. list comprehension*). Obično se takvi izrazi koriste za stvaranje nove liste kao rezultata neke operacije nad članovima neke druge sekvence ili za pripremu podsekvence na temelju nekog uvjeta. U sljedećem primjeru stvara se lista kvadrata brojeva od 1 do 10 na klasičan način, petljom.

```
>>> listaKvadrata = []
>>> for x in range(1,11):
      listaKvadrata.append(x**2)

>>> listaKvadrata
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Isti rezultat možemo postići i izrazom za stvaranje liste, pri čemu je zapis kraći i elegantniji.

```
>>> listaKvadrata = [x**2 for x in range(1,11)]
```

Sintaksno, izraz za stvaranje liste sastoji se od uglate zagrade unutar kojih se nalazi izraz koji se primjenjuje nad pojedinim elementima sekvence, zatim slijedi naredba `for`, a može biti i još naredbi `for` i `if`, kao u sljedećem primjeru. U primjeru su formirane dvije petlje, vanjska po `x` i unutarnja po `y`, a k tome se ispituje i ispunjavanje uvjeta za dodavanje elementa u listu.

```
>>> [(x,y) for x in [1,2,3] for y in [3,1,4] if x < y]
[(1, 3), (1, 4), (2, 3), (2, 4), (3, 4)]
```

Izrazi za stvaranje liste mogu se i gnijezditi, čime se generira lista čiji su elementi liste. Primjerice, ako imamo matricu zapisanu u obliku liste redaka, njeno transponiranje možemo obaviti gniježdenjem izraza za generiranje liste.

```
>>> mat = [[1, 2, 3, 4],
           [5, 6, 7, 8],
           [9, 10, 11, 12]]
>>> matT = [[redak[i] for redak in mat]
               for i in range(4)]
>>> matT
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

4.3.6 Rječnici

Još jedan složeni tip podataka u Pythonu su rječnici (ugrađeni tip `dict`). Ovaj tip podataka odgovara asocijativnim poljima u Perlu. Radi se o neuređenoj kolekciji objekata, pa se elementima ne pristupa putem slijednih indeksa, već temeljem jedinstvenih *ključeva*. Indeksiranje elemenata u rječniku je vrlo brza operacija pretraživanja u kojoj se koristi raspršeno adresiranje (*engl. hash*). To znači da elementi nisu složeni u određenom redoslijedu, već ključevi određuju simboličko mjesto elemenata u rječniku. Python koristi optimirane algoritme raspršenog adresiranja, pa su dohvati vrlo brzi. Kao i kod listi, elementi rječnika mogu biti proizvoljnih tipova, uključujući i liste i rječnike, uz mogućnost gniježdenja proizvoljne dubine. Rječnici su, poput listi, izmjenljivi objekti, te sadrže *reference* na objekte.

Operacije nad rječnicima

U sljedećoj tablici prikazani su tipični literali rječnika i neke važnije operacije nad njima.

Operacija	Opis
`D1 = { }`	prazni rječnik
`D2 = {'spam': 2, 'eggs': 3}`	rječnik s dva elementa
`d3 = {'food': {'ham': 1,'egg': 2}}`	gniježdenje
`D2['eggs']; d3['food']['ham']`	indeksiranje ključem
`'eggs' in D2`	pripadnost
`D2.keys( ) ; D2.values( )` `D2.copy( ) ; D2.get('cake',0)`	ključevi, vrijednosti kopiranje ...
`len(D1)`	duljina
`D2['apple'] = 42 ; del D2['eggs']`	dodavanje; brisanje

Stvaranje rječnika i pristup elementima

Literali rječnika zapisuju se kao lista parova ključ-vrijednost unutar vitičastih zagrada, pri čemu su ključ i vrijednost odvojeni dvotočkom, a pojedini elementi zarezima. Adresiranje pojedinog elementa sintaksno je slično pristupu elementima liste, osim što se umjesto cjelobrojnog indeksa navodi ključ. Ključ može biti objekt proizvoljnog *neizmjenljivog* tipa.

```
>>> d2 = {'spam': 2, 'ham': 1, 'eggs': 3}
>>> d2['spam']  # dohvat vrijednosti na temelju ključa
2
>>> d2  # redoslijed je nepredvidljiv
{'eggs': 3, 'ham': 1, 'spam': 2}
```

Ako ispišemo vrijednost rječnika, redoslijed elemenata ne mora odgovarati onom u kojem su elementi uneseni u rječnik, što je logična posljedica raspršenog adresiranja. S obzirom na neuređenost rječnika, jasno je da se nad njima ne mogu primjenjivati uobičajene operacije nad sekvencama (izrezivanje, nadovezivanje).

Ugrađena funkcija `len` primjenjiva je na rječnike, a njena povratna vrijednost predstavlja broj pohranjenih elemenata.

```
>>> len(d2)
3
```

Metode rječnika

Metoda `keys` vraća "listu" svih ključeva rječnika. Zapravo se ne radi o listi nego o pobrojivom tipu (*engl. iterable*) – objektu kroz koji se može iterirati.

```
>>> D = { 'a': 2, 'b':3, 'c':4}
>>> L = D.keys()
>>> L
dict_keys(['a', 'c', 'b'])
>>> type(L)
<class 'dict_keys'>

>>> for i in L:
            print(i, end=' ')
a c b
```

I kroz sam rječnik može se iterirati, pri čemu se iterira kroz ključeve, kao što ilustrira sljedeći primjer.

```
>>> for i in D:
```

```
                print("{}:{} ;".format(i, D[i]), end = ' ')
a:2 ; c:4 ; b:3 ;
```

Slično, metoda `values` vraća vrijednosti pohranjene u rječniku kao pobrojivi objekt.

```
>>> for val in D.values():
                print(val, end=' ')

2 4 3
```

Metoda `items` vraća parove (ključ, vrijednost)

```
>>> for item in D.items():
                print(item, end=' ')

('a', 2) ('c', 4) ('b', 3)
```

Ispitivanje postojanja ključa u rječniku postiže se operatorom `in`.

```
>>> 'c' in D
True
```

Izmjenljivost rječnika

Rječnici su, kao i liste, izmjenljivi (*engl. mutable*) objekti, odnosno mogu se mijenjati, proširivati ili smanjivati bez stvaranja kopije. To možemo ilustrirati na nekoliko primjera.

```
>>> d2
{'eggs': 3, 'ham': 1, 'spam': 2}

>>> d2['ham'] = ['grill', 'bake', 'fry'] # izmjena
>>> d2
{'eggs': 3, 'ham': ['grill', 'bake', 'fry'],
 'spam': 2}

>>> del d2['eggs'] # brisanje
>>> d2
{'ham': ['grill', 'bake', 'fry'], 'spam': 2}

>>> d2['brunch'] = 'Bacon' # dodavanje elementa
>>> d2
{'brunch': 'Bacon', 'ham': ['grill', 'bake', 'fry'],
 'spam': 2}
```

Kao i kod listi, dodjela vrijednosti preko postojećeg ključa mijenja vrijednost elementa rječnika. Ako dodjeljujemo vrijednost za nepostojeći ključ, stvara se novi element rječnika. Podsjetimo se, kod liste pokušaj dodjele vrijednosti preko nepostojećeg indeksa uzrokuje iznimku, dok se dodavanje novog elementa *u listu* obavlja metodom `append` ili dodjelom vrijednosti isječku.

Pokušaj čitanja elementa preko nepostojećeg ključa rezultira greškom. Kako bismo izbjegli iznimke u slučajevima kada ne znamo postoji li element s traženim ključem, možemo to prethodno provjeriti primjenom već spomenutog operatora `in`. No, Python nudi i elegantnije rješenje, korištenjem metode `get`. Metoda u slučaju da rječnik ne sadrži element s traženim ključem vraća *pretpostavljenu vrijednost*, koja se navodi kao argument pri pozivu metode. Ukoliko se pretpostavljena vrijednost ne navede, koristi se objekt `None`.

```
>>> d2 = {'brunch': 'Bacon', 'ham': 7, 'spam': 2}

>>> d2['salt']
... KeyError: 'salt'

>>> d2.get('salt'),d2.get('salt',33),d2.get('spam',11)
(None, 33, 2)
```

Kao i liste, rječnici imaju i metodu pop, koja briše element iz rječnika i vraća njegovu vrijednost. Kao argument se navodi ključ traženog elementa. S obzirom se je rječnik neuređena kolekcija, pop bez navođenja ključa nema smisla te se generira iznimka.

```
>>> d2 = {'toast':4, 'eggs':3, 'ham':1, 'spam':2}
>>> d2.pop('toast')
4
>>> d2
{'eggs': 3, 'ham': 1, 'spam': 2}
```

Metoda update nudi funkcionalnost sličnu ulančavanju, pri čemu se ključevi i vrijednosti jednog rječnika spajaju se s drugim rječnikom. Ukoliko drugi rječnik sadrži element s ključem koji postoji i u rječniku koji se ažurira, nova vrijednost prepisuje staru.

```
>>> d2 = {'eggs':3, 'ham':1, 'spam':2}
>>> d3 = {'toast':4, 'muffin':5, 'eggs':7}

>>> d2 + d3  # rječnici ne poznaju ulančavanje
...TypeError: unsupported operand type(s)
   for +: 'dict' and 'dict'

>>> d2.update(d3)
>>> d2
{'toast': 4, 'ham': 1, 'spam': 2, 'muffin': 5,
 'eggs': 7}
```

Primjenu rječnika možemo ilustrirati jednim jednostavnim primjerom. U rječniku table navedeni su neki programski jezici i njihovi tvorci. Prikazan je pristup jednom elementu i njegovo pridruživanje imenu, te način iteriranja kroz ključeve rječnika i ispis pojedinih elemenata u obliku tablice.

```
>>> table = {'Python':  'Guido van Rossum',
                     'Perl':    'Larry Wall',
                     'Tcl':     'John Ousterhout' }

>>> language = 'Python'
>>> creator  = table[language]
>>> creator
'Guido van Rossum'


>>> for lang in table.keys(  ):
              print(lang, '\t', table[lang])

Python   Guido van Rossum
Tcl      John Ousterhout
Perl     Larry Wall
```

Rječnik kao fleksibilna "lista"

Ključ rječnika može biti objekt bilo kojeg nepromjenljivog tipa, pa tako i cijeli broj. To nam omogućuje da rječnikom oponašamo listu, pri čemu dobivamo fleksibilnost u smislu da možemo dodavati elemente s proizvoljnim indeksima. Kod liste bi pokušaj zapisivanja elementa s indeksom van trenutne duljine liste rezultiralo iznimkom.

```
>>> L = []
>>> L[99] = 'spam'
...
IndexError: list assignment index out of range
```

Naravno, mogli bismo prethodno alocirati veliku listu, npr. primjenom operatora ponavljanja: `L = [0]*100`, no umjesto toga možemo iskoristiti rječnik. Osim fleksibilnosti, to rješenje štedi i memorijski prostor u slučaju slabe popunjenosti ovakve liste. U gornjem primjeru zauzeli smo prostor potreban za smještaj 100 elemenata, a trebao nam je samo jedan. Rješenje pomoću rječnika bi zahtijevalo inicijalizaciju praznog rječnika, a nakon toga su svi pristupi sintaksno jednaki pristupima elementima liste. Jedini je preduvjet da koristimo cjelobrojne ključeve, koji odgovaraju indeksima liste.

```
>>> D = {}
>>> D[99] = 'spam'   # kao da je lista
>>> D[99]
'spam'

>>> D   # ispis rječnika kao cjeline pokazuje razliku
{99: 'spam'}
```

Pohrana rijetkih podatkovnih struktura

Kao nastavak prethodnog koncepta, rječnik je prikladan i za pohranu rijetkih matrica, odnosno općenito slabo popunjenih podatkovnih struktura. Tu mislimo na podatkovne strukture u kojima samo mali broj elemenata ima vrijednost različitu od 0. Primjer u nastavku ilustrira kako možemo koristiti rječnik za pohranu trodimenzijskog polja, u kojemu pojedini element adresiramo s 3 koordinate. Budući da se kao ključ može koristiti proizvoljan neizmjenljivi tip, možemo odabrati i n-torku. Taj tip podataka smo do sada samo spomenuli, ali radi se naprosto o više vrijednosti odvojenih zarezima i navedenih unutar oblih zagrada. Zagrade se mogu čak i izostavili, pa dobivamo elegantan zapis adrese pojedinog elementa.

```
>>> Matrix = {}
>>> Matrix[(2,3,4)] = 88  # adresiranje n-torkom
>>> Matrix[7,8,9] = 99  # može čak i ovako
>>> Matrix
{(2, 3, 4): 88, (7, 8, 9): 99}

>>> X = 2; Y = 3; Z = 4
>>> Matrix[X,Y,Z]
88
```

Ovakva struktura može biti rijetko popunjena, pri čemu bismo sve elemente čije vrijednosti nisu upisane htjeli interpretirati kao 0. Tu još moramo riješiti pitanje čitanja takvih (nedefiniranih) elemenata. Znamo da će pokušaj čitanja nepostojećeg elementa izazvati iznimku, a zapravo bismo htjeli u tom slučaju dobiti vrijednost 0. To možemo riješiti na više načina. Prvi je da uvijek prije pokušaja čitanja elementa provjerimo postojanje ključa.

```
>>> adr = (2, 3, 6)
>>> if adr in Matrix:
            print(Matrix[adr])
    else:
            print(0)
0
```

Drugi način bi bio *hvatanje iznimke.* Za to je potrebno koristiti naredbu `try` s odgovarajućim blokom `except`.

```
>>> try:
            print(Matrix[adr])
    except KeyError:   # iznimka nepostojećeg ključe
            print(0)
```

No, najjednostavnije rješenje je korištenjem metode rječnika `get` koju smo već ranije opisali, uz navođenje podrazumijevane povratne vrijednosti 0.

```
>>> Matrix.get((2,3,4), 0), Matrix.get((2,3,6), 0)
(88, 0)
```

4.3.7 n-torke

Python poznaje još jedan složeni ugrađeni tip podataka, n-torke (`tuple`), koji je vrlo sličan listama. I to je uređena kolekcija proizvoljnih objekata, ali *nije izmjenljiva.* Svojstvo neizmjenljivosti, između ostaloga, omogućuje korištenje n-torki kao ključeva rječnika, kao što smo to vidjeli u prethodnom primjeru. Još jedna razlika je da n-torke nemaju definiranih metoda.

Operacije nad n-torkama

U sljedećoj tablici navedeni su tipični literali n-torki, kao i neke uobičajene operacije.

Operacija	Opis
`( )`	prazna n-torka
`t1 = (0,)`	jednočlana n-torka (!)
`t2 = (0, 'Ni', 1.2, 3)`	četveročlana n-torka
`t2 = 0, 'Ni', 1.2, 3`	može i ovako
`t3 = ('abc', ('def', 'ghi'))`	ugniježdene n-torke
`t1[0] ; t3[1][1]`	indeksiranje
`t2[1:3] ; len(t2)`	izrezivanje, duljina
`t1 + t2 ; t2 * 3`	ulančavanje, ponavljanje
`for x in t2 ; 3 in t2`	iteracija, pripadnost

Za n-torke su podržane standardne operacije nad sekvencama, poput nadovezivanja, ponavljanja, indeksiranja i izrezivanja.

```
>>> (1, 2) + (3, 4)        # nadovezivanje
(1, 2, 3, 4)
>>> (1, 2) * 4  # ponavljanje
(1, 2, 1, 2, 1, 2, 1, 2)
>>> T = (1, 2, 3, 4)
>>> T[0], T[1:3]   # indeksiranje, izrezivanje
(1, (2, 3))
```

Ako želimo zapisati n-torku s jednim elementom, nije dovoljno taj element navesti unutar oblih zagrada, jer će to Python shvatiti kao izraz. Zato jednočlanu n-torku zapisujemo s dodanim zarezom.

```
>>> t = (1)
>>> t
1

>>> y = (40,)
>>> y
(40,)
```

Budući da n-torke ne podržavaju metode, često je prikladno n-torku pretvoriti u listu i koristiti metode listi. Po završetku obrade, listu možemo pretvoriti nazad u n-torku. Za pretvorbu se koriste ugrađene funkcije `list` i `tuple`.

```
>>> T = ('cc', 'aa', 'dd', 'bb')
>>> tmp = list(T)
>>> tmp.sort(  )
>>> tmp
['aa', 'bb', 'cc', 'dd']

>>> T = tuple(tmp)
>>> T
('aa', 'bb', 'cc', 'dd')
```

Za sortiranje n-torke možemo primijeniti i ugrađenu funkciju `sorted`, uz napomenu da je povratna vrijednost sortirana lista, koju također potom možemo pretvoriti u n-torku.

```
>>> sorted(('cc', 'aa', 'dd', 'bb'))
['aa', 'bb', 'cc', 'dd']
```

Neizmjenljivost n-torki

Pravilo o neizmjenljivosti n-torki odnosi se samo na najvišu razinu same n-torke. Naime, n-torka može sadržavati druge, izmjenljive objekte, primjerice liste. Element n-torke se ne može zamijeniti novim objektom, no ako je taj element izmjenljiv, njegov se sadržaj može promijeniti.

```
>>> T = (1, [2, 3], 4)
>>> T[1][0] = 'spam'   # izmjena!
>>> T
(1, ['spam', 3], 4)
>>> T[1] = 'spam'  # ne ide
...
TypeError: 'tuple' object does not support
item assignment
```

4.3.8 Skupovi

Python ima i podatkovni tip koji odgovara matematičkom pojmu skupa (ugrađeni tip `set`). Tu nas zanima samo pripadnost pojedinih elemenata skupu, te njihova jedinstvenost. Skupovi su neuređene kolekcije, i po tome su slični rječnicima. No, elementima se ne pristupa preko ključeva, već preko njihovih vrijednosti. Elementi skupa su jedinstveni, odnosno dodavanjem

elementa koji je već u skupu ne mijenja sadržaj skupa. Budući da se vrijednost koristi kao ključ, elementi skupa mogu biti isključivo nepromjenljivi objekti.

Novi skup se može stvoriti kao literal u kojem su elementi navedeni unutar vitičastih zagrada, ili korištenjem ugrađene funkcije `set`. Funkcija set kao argument prima pobrojivi objekt kojim se generiraju elementi novog skupa.

```
>>> skup = {'kruška', 'jabuka', 'šljiva'}
>>> skup
{'šljiva', 'kruška', 'jabuka'}
>>> type(skup)
<class 'set'>

>>> set(['kruška', 'jabuka', 'šljiva'])  # lista
{'šljiva', 'kruška', 'jabuka'}

>>> set(('kruška', 'jabuka', 'šljiva'))  # n-torka
{'šljiva', 'kruška', 'jabuka'}

>>> d2 = {'toast': 4, 'ham': 1, 'eggs': 7}
>>> set(d2.values())  # može i rječnik
{1, 4, 7}
```

Međutim, funkcija `set` ne može primiti više argumenata koji bi poslužili za inicijalizaciju skupa.

```
>>> skup = set('kruška', 'jabuka', 'šljiva')
...
TypeError: set expected at most 1 arguments, got 3
```

Operacije nad skupovima

Skupovi imaju niz metoda kojima se ostvaruju operacije dodavanja i brisanja elemenata, a definirano je i nekoliko operatora za uobičajene skupovne operacije.

Metoda `add` omogućuje dodavanje jednog elementa u skup, dok je za dodavanje više elemenata pogodna metoda `update`, koja nove elementa preuzima iz pobrojivog objekta.

```
>>> skup.add('jagoda')   # dodavanje jednog elementa
>>> skup
{'šljiva', 'jagoda', 'kruška', 'jabuka'}

>>> skup.update(['smokva','dinja'])  # dodavanje više elemenata
>>> skup
{'šljiva', 'smokva', 'kruška', 'jagoda', 'jabuka', 'dinja'}
```

Za ispitivanje pripadnosti elementa skupu, koristi se operator `in`. Za ispitivanje je li neki skup podskup, pravi podskup ili nadskup drugog skupa, koriste se operatori <=, < i >.

```
>>> 'smokva' in skup
True

>>> {'šljiva', 'smokva', 'dinja'} <= skup   # podskup
True
>>> {'šljiva', 'smokva', 'dinja'} < skup    # pravi podskup
True
>>> {'šljiva', 'smokva', 'dinja'} > skup    # nadskup
False
```

Brisanje elementa može se obaviti primjenom metode `remove`. Metoda `clear` namijenjena je brisanju svih elemenata iz skupa. Zanimljivo je da skupovi imaju i metodu `pop`, koja vadi i briše *slučajan* element iz skupa.

```
>>> skup.remove('jagoda')
>>> skup
{'šljiva', 'smokva', 'kruška', 'jabuka', 'dinja'}

>>> skup.pop()  # vadi proizvoljni element
'smokva'

>>> skup.clear()  # briše sve
>>> skup
set()
```

Podržane su i operacije algebre skupova: unija, presjek i razlika.

```
>>> voće = {'breskva', 'jabuka', 'rajčica'}
>>> povrće = {'rajčica', 'kupus', 'blitva'}
>>> voće | povrće   # unija
{'jabuka', 'blitva', 'breskva', 'rajčica', 'kupus'}

>>> voće & povrće   # presjek
{'rajčica'}

>>> voće - povrće   # razlika
{'breskva', 'jabuka'}
```

4.3.9 Datoteke

Datotekama, kao imenovanim spremnicima podataka, upravlja operacijski sustav. Ugrađeni tip podataka `file` u Pythonu omogućuje pristup datotekama operacijskog sustava. Za stvaranje objekta tipa `file`, koji služi kao veza prema fizičkoj datoteci, koristi se ugrađena funkcija `open`. Čitanje i pisanje u datoteku ostvaruje se pozivanjem metoda odgovarajućeg datotečnog objekta.

Operacije s datotekama

U tablici su ilustrirane uobičajene operacije s datotekama.

Operacija	Opis
`out = open('/tmp/spam', 'w')`	otvara datoteku za pisanje
`in = open('bla.txt', 'r')`	otvara datoteku za čitanje
`S = in.read( )`	učitava cijelu datoteku
`S = in.read(N)`	čita N bajtova
`S = in.readline( )`	učitava sljedeći redak
`L = in.readlines( )`	cijelu datoteku u listu L
`out.write(S)`	zapisuje znakovni niz
`out.writelines(L)`	zapisuje stringove iz liste
`out.close( )`	zatvaranje datoteke

Datoteka se otvara, uz kreiranje odgovarajućeg datotečnog objekta, ugrađenom funkcijom `open`. U osnovnoj primjeni dovoljno je kao argument navesti ime datoteke, koje može sadr-

žavati i apsolutni ili relativni put. Ako put nije naveden, datoteka se traži u tekućem kazalu. Može se zadati i cijeli niz dodatnih argumenata, koji imaju svoje podrazumijevane vrijednosti.

```
open(file, mode='r', buffering=-1, encoding=None,
     errors=None, newline=None, closefd=True)
```

Argument `mode` određuje način pristupa datoteci, a on može biti čitanje (`'r'`), pisanje (`'w'`), ili nadodavanje (`'a'`). U oznaku načina pristupa mogu se dodati i dodatne oznake za binarni način pristupa podacima (`'b'`), te za otvaranje datoteke za čitanje i pisanje (`'+'`). Argument `encoding` određuje način kodiranja znakova u datoteci (neke moguće vrijednosti su `'ascii'`, `'cp1250'` i `'utf8'`). Ako se način kodiranja ne navede, podrazumijevaju se postavke operacijskog sustava. Ostale argumente naveli smo samo radi potpunosti i nećemo ih objašnjavati.

Nakon otvaranja, datoteka je spremna za čitanje ili zapisivanje podataka. Kada nam datoteka nije više potrebna, možemo je zatvoriti metodom `close()`. Iako će u većini slučajeva Python sam zatvoriti datoteku kada se više ne koristi, dobra je navika voditi računa o zatvaranju datoteka.

Podaci se učitavaju odnosno zapisuju u datoteke u obliku znakovnih nizova. U nastavku je nekoliko primjera. Najprije se datoteka otvara za pisanje i u nju se zapisuje tekst primjenom metode `write`. Nakon toga se ista datoteka ponovo otvara i tekst se učitava metodom `readline`. U primjeru se vidi da čitanje iz datoteke kada se stigne do njenog kraja vraća prazan niz, za razliku od učitavanja praznog retka koje vraća `'\n'`.

```
>>> dat = open('tekst.txt', 'w', encoding='utf8')
>>> dat.write('Živjeli Skriptni jezici!\n\n')
26

>>> dat = open('tekst.txt', encoding='utf8')
>>> dat.readline()  # čitanje jednog retka
'Živjeli Skriptni jezici!\n'

>>> dat.readline()
'\n'

>>> dat.readline() # prazni string: EOF
''
```

Pri otvaranju datoteke uputno je definirati način kodiranja, jer je podrazumijevani način ovisan o platformi. Primjerice, ako na Windowsima otvorimo datoteku kodiranu po standardu UTF-8 bez navođenja načina kodiranja, dobit ćemo rezultat kao u primjeru. Vidimo da je podrazumijevani način kodiranja `cp1252`, pa se neki znakovi pogrešno interpretiraju i ispisuju.

```
>>> dat = open('tekst.txt')  # zapisana kao utf8
>>> dat.readline()
'Å½ivjeli Skriptni jezici!\n'
>>> dat.encoding  # možemo ga ispitati
'cp1252'
```

I sa zapisivanjem može biti problema jer primjerice `cp1252` ne podržava hrvatske znakove pa se oni ne mogu kodirati i generira se iznimka.

```
>>> dat = open('tekst2.txt', 'w')
>>> dat.write('Đurđice su lijepo cvijeće')
...
UnicodeEncodeError: 'charmap' codec can't encode
character '\u0110' in position 0: ...
```

Metoda `read` omogućava učitavanje cijele datoteke u znakovni niz.

```
>>> dat = open('tekst.txt', 'r', encoding='utf8')
>>> sadržaj = dat.read()   # cijela datoteka
>>> sadržaj, type(sadržaj)
('Živjeli Skriptni jezici!\n\n', <class 'str'>)
```

Datoteka se tipično učitava slijedno. Svaka operacija čitanja pomiče pokazivač datoteke. No, moguće je i izravno pomicanje pokazivača, čime se omogućava i izravan pristup željenom sadržaju u datoteci. Trenutni položaj pokazivača može se saznati korištenjem metode `tell`. Povratna vrijednost označava pomak pokazivača od početka datoteke, izražen u bajtovima. Za pomak pokazivača koristi se metoda `seek`, uz navođenje pomaka u bajtovima. Kao drugi argument može se navesti i oznaka odakle se pomak odbrojava. Podrazumijeva se odbrojavanje od početka datoteke, no može se odbrojavati i od trenutne pozicije ili od kraja. U primjeru je pokazan pomak pokazivača na početak datoteke.

```
>>> dat.seek(0)  # pozicija 0 od početka
0
```

Datoteka se može učitati odjednom i u *listu* znakovnih nizova, primjenom metode `readlines()`.

```
>>> retci = dat.readlines()
>>> retci
['Živjeli Skriptni jezici!\n', '\n']
```

Slično, zapisivanje *liste* znakovnih nizova u datoteku obavlja se metodom `writelines()`.

```
>>> dat = open('tekst2.txt', 'w', encoding='utf8')
>>> retci = ['prvi redak\n', '2. red\n', '3. retčić\n']
>>> dat.writelines(retci)
>>>
>>> dat = open('tekst2.txt', 'r', encoding='utf8')
>>> dat.read()
'prvi redak\n2. red\n3. retčić\n'
```

Datoteku možemo učitavati i redak po redak u petlji, korištenjem metode `readline`, ali i korištenjem datoteke kao pobrojivog tipa, izravnim iteriranjem.

```
>>> dat.seek(0)  # na početak
>>> for redak in dat:
        print(redak, end='')
prvi redak
2. red
3. retčić
```

Naredba with

U novijim programima u Pythonu uobičajeno je korištenje mogućnosti automatskog zauzimanja i otpuštanja resursa naredbom `with`. Objekt s kojim se radi mora biti tome prilagođen. Pri baratanju datotekom, na taj se način osigurava njeno automatsko zatvaranje i u slučaju iznimke. Koristi se idiom:

```
>>> with open('tekst2.txt', encoding='utf8') as dat:
              pročitano = dat.read()

>>> pročitano
'prvi redak\n2. red\n3. retčić\n'
```

Nakon završetka bloka `with`, datoteka se zatvara. U to se možemo uvjeriti ispitivanjem atributa `closed`.

```
>>> dat.closed  # datoteka je zatvorena automatski
True
```

Binarne datoteke

Ako u datoteku želimo pohranjivati podatke koji nisu tekstni, treba je otvoriti kao binarnu. To se postiže tako da u oznaci načina pristupa dodamo znak "b". Pri čitanju (odnosno zapisivanju) podataka iz binarno otvorene datoteke ne obavljaju se prilagodbe oznaka kraja retka (`'\r\n'` na Windowsima, `'\n'` na Unixu, `'\r'` na Macu). Osim toga, ne obavlja se kodiranje znakova kao u tekstnom načinu pristupa, za se ne zadaje ni odgovarajući argument. Pristup binarno otvorenoj datoteci obavlja se preko niza bajtova, a ne preko znakovnog niza.

```
>>> with open('tekst2.txt', 'rb') as dat:
              pročitano = dat.read()

>>> pročitano
b'prvi redak\r\n2. red\r\n3. ret\xc4\x8di\xc4\x87\r\n'
```

Pohranjivanje objekata u datoteku

Ponekad trebamo pohraniti sadržaj objekata koje koristimo u programu u datoteku. Jedan način je da ih pohranimo u obliku znakovnih nizova, kao u sljedećem primjeru. Nakon pohranjivanja, datoteka se ponovo otvara i ispisuje se sadržaj datoteke.

```
>>> X, Y, Z = 43, 44, 45  # razni tipovi objekata
>>> S = 'Spam'            # moraju se pretvoriti
>>> D = {'a': 1, 'b': 2}
>>> L = [1, 2, 3]

>>> with open('zapisi.txt', 'w') as F:
        F.write(S + '\n')
        F.write('{},{},{}\n'.format(X, Y, Z))
        F.write(str(L) + '$' + str(D) + '\n')

>>> with open('zapisi.txt', 'r') as dat:
        print(dat.read())

Spam
43,44,45
[1, 2, 3]${'a': 1, 'b': 2}
```

Kako bismo objekte pohranjene na ovaj način obnovili, potrebno ih je rekonstruirati iz znakovnih nizova pročitanih iz datoteke. Primjer ilustrira kako se znakovni nizovi dijele i pretvaraju u odgovarajuće objekte. Naravno, pritom je potrebno znati koje vrstu podatka očekujemo na kojem mjestu u datoteci, kako bismo mogli pozvati odgovarajuću funkciju za pretvorbu.

```
>>> F = open('zapisi.txt')
>>> line = F.readline( ).rstrip()
>>> line
'Spam'
```

```
>>> line = F.readline( ) # sljedeći redak
>>> line
'43,44,45\n'

>>> parts = line.split(',') # dijelimo na zarezima
>>> parts
['43', '44', '45\n']

>>> numbers = [int(P) for P in parts] # comprehension
>>> numbers  # int zanemaruje \n
[43, 44, 45]
```

Brojeve je jednostavno pretvoriti iz znakovnog zapisa u odgovarajuće objekte, no s listama i rječnicima to nije tako jednostavno. U takvim slučajevima možemo iskoristiti ugrađenu funkciju `eval`, koja koristi znakovni niz kao Pythonov *izraz*, dakle kao izvršivi kôd.

```
>>> line = F.readline()
>>> line
"[1, 2, 3]${'a': 1, 'b': 2}\n"

>>> parts = line.split('$') # dijeli na $
>>> parts
['[1, 2, 3]', "{'a': 1, 'b': 2}\n"]

>>> eval(parts[0])
[1, 2, 3]

>>> objects = [eval(P) for P in parts]
>>> objects
[[1, 2, 3], {'a': 1, 'b': 2}]
```

Za pohranu i učitavanje objekata u Pythonu vrlo je prikladan standardni modul `pickle`. Objekt (i objekti koje on sadrži) se pretvara u niz bajtova i pohranjuje u datoteku. Ovaj način zapisivanja podataka naziva se *serijalizacijom.* Pri otvaranju datoteke, potrebno je označiti binarni način pristupa. U nastavku je prikazan primjer pohranjivanja objekata u datoteku funkcijom `pickle.dump` (uočiti da se kao argument prenosi samo jedan objekt, koji može biti složen).

```
>>> objects
[[1, 2, 3], {'a': 1, 'b': 2}]

>>> import pickle
>>> with open('proba.pickle', 'wb') as f:
        pickle.dump(objects, f)
```

Učitavanje je slično, korištenjem funkcije `pickle.load`, kojoj se kao argument zadaje datoteka iz koje se objekt čita, a povratna vrijednost je učitani objekt.

```
>>> with open('proba.pickle', 'rb') as f:
              učitano = pickle.load(f)

>>> učitano
[[1, 2, 3], {'a': 1, 'b': 2}]
```

4.4 Usporedbe objekata

Objekti u Pythonu mogu se uspoređivati. Ako se radi o složenim objektima, uspoređuju se svi njihovi dijelovi, do trenutka kada se može odlučiti o odnosu između objekata koji se uspoređuju. Ugniježđeni objekti uspoređuju se rekurzivno, do potrebne dubine, a prva pronađena razlika određuje rezultat usporedbe.

```
>>> L1 = [1, ('a', 3)] # ista vrijednost
>>> L2 = [1, ('a', 3)] # različiti objekti

>>> L1 == L2, L1 is L2 # jednaki? isti?
(True, False)
```

Pri usporedbama koristi se operator == kojim se ispituje jednakost objekata. Postoji i operator `is`, kojim se uspoređuje njihov identitet, odnosno ispituje nalaze li se objekti na istoj adresi u memoriji.

Zanimljivo je pogledati što se događa, primjerice, s *kratkim* znakovnim nizovima (slično je i s brojevima). Primjer ilustrira da Python, iako je riječ o dvije nezavisne naredbe pridruživanja koje bi trebale stvoriti dva nezavisna objekta, koristi već stvoreni objekt koji ima istu vrijednost. Kako se radi o neizmjenljivim objektima, nije važno stvaranje "umjetne" dijeljene reference. Radi se o optimizaciji prostora.

```
>>> S1 = 'spam'
>>> S2 = 'spam'
>>> S1 == S2, S1 is S2
(True, True)
```

Međutim, ova optimizacija se ne koristi čim znakovni niz sadrži razmake. To se također može ilustrirati jednostavnim primjerom.

```
>>> S1 = 'a b'
>>> S2 = 'a b'
>>> S1 == S2, S1 is S2
(True, False)
```

Rekurzivnost usporedbi objekata s ugniježđenim objekata može se ilustrirati sljedećim primjerom.

```
>>> L1 = [1, ('a', 3)]
>>> L2 = [1, ('a', 2)]
>>> L1 < L2, L1 == L2, L1 > L2
(False, False, True)
```

4.5 Naredbe u Pythonu

Do sada smo susretali jednostavne *izraze*, te naredbe pridruživanja (*engl. assignment*) objekata imenima. Složene naredbe (*engl. compound statements*) u Pythonu imaju specifičnu strukturu. Za primjer promotrimo naredbu `if`. U jezicima čija sintaksa potječe iz C-a (C++, Java, JavaScript, Perl), možemo uočiti dio naredbe u kojem se ispituje uvjet, te blok naredbi koje se izvršavaju ako je uvjet ispunjen. Blok naredbi označava se vitičastim zagradama. U jezicima poput Pascala, za označavanje bloka naredbi koriste se ključne riječi poput `BEGIN` i `END`.

```
if (x > y) {
   x = 1;
```

```
    y = 2;
  }
```

U Pythonu, blokovi naredbi označavaju se uvlačenjem (indentacijom). Ta sintaksna specifičnost Pythona prisiljava programera na pisanje urednog kôda. Sintaksno ispravan program u Pythonu inherentno je čitljiv.

```
if x > y :
           x = 1
           y = 2
```

Složena naredba u Pythonu sastoji se od *zaglavlja*, koje završava dvotočkom, te *tijela* koje čine jednako poravnate naredbe. Nema posebnih oznaka početka i kraja bloka naredbi.

```
if x > y :    # zaglavlje
           x = 1    # ugniježdeni blok
           y = 2
```

Zagrade oko uvjeta koji se ispituje su *neobavezne*. Kako se svaki izraz može uokviriti zagradama, i izraz za ispitivanje uvjeta može se pisati u zagradama, no to nije "Pythonski stil".

Kraj retka označava kraj naredbe, pa nema potrebe za odvajanjem naredbi oznakom `";"`. Međutim, dozvoljeno je pisanje više *jednostavnih* naredbi u jednom retku.

```
a = 1; b = 2; print(a + b)
```

Postoji i mogućnost protezanja naredbe kroz više redaka kako bi se poboljšala čitljivost. To se može postići korištenjem zagrada. Naime, ako dio naredbe uključuje par zagrada (oblih, uglatih ili vitičastih), naredba ne može završiti dok se ne dođe do zatvarajuće zagrade.

```
mlist = [111,
         222,
         333]
```

Kako se bilo koji izraz može zatvoriti u zagrade, na taj način je lako ostvariti "dozvolu" protezanja naredbe u novi red. Ovaj pristup primjenjiv i na zaglavlje složene naredbe.

```
X = (A + B +
     C + D)

if (A == 1 and
    B == 2 and
    C == 3):
      print('spam' * 3)
```

Još jedan izuzetak predstavlja pisanje složenih naredbi u jednom retku, što može biti zgodno za sažeto pisanje naredbi `if` ili kratkih petlji.

```
if x > y: print(x)
```

Naredba if

Naredba `if` tipična je za većinu proceduralnih jezika. U Pythonu je njen općeniti oblik:

```
if <test1>:
    <naredbe1>   # blok naredbi
elif <test2>:       # opcionalni elif
    <naredbe2>
```

```
else:                # opcionalni else
    <naredbe3>
```

Python izvršava blok naredbi pridružen prvom uvjetu koji je ispunjen (`True`), ili blok naredbi pridružen grani `else` ukoliko niti jedan od uvjeta nije ispunjen. Dijelovi `elif` i `else` mogu se izostaviti. Vezanje dijelova iste naredbe: `if`, `elif` i `else` određuje se njihovim vertikalnim poravnavanjem!

```
>>> x = 'killer rabbit'

>>> if x == 'roger':
            print("how's jessica?")
    elif x == 'bugs':
            print("what's up doc?")
    else:
            print('Run away! Run away!')

Run away! Run away!
```

Višestruko grananje

Python nema naredbu oblika `switch` ili `case`, već se višestruko grananje izvodi nizom `if-elif` ispitivanja ili indeksiranjem rječnika. Kako se rječnici mogu graditi tijekom izvođenja programa, taj način upravljanja programskim slijedom može biti fleksibilniji od unaprijed kodiranih grananja. Primjerice, ispis ovisan o sadržaju neke varijable može se ostvariti kao u sljedećem primjeru.

```
>>> choice = 'ham'
>>> branch = {'spam': 1.25,
              'ham':  1.99,
              'eggs': 0.99}

>>> print(branch.get(choice, 'Bad choice'))
1.99

>>> print(branch.get('bacon', 'Bad choice'))
Bad choice
```

Rječnici su prikladni za ovakvu vrstu akcija, no mogu se primijeniti i za složenije situacije. Naime, rječnici mogu sadržavati i funkcije koje se onda pozivaju u ovisnosti o vrijednosti ključa.

Programska petlja while

Najopćenitija programska petlja ostvaruje se naredbom `while`, pri čemu se zadani blok naredbi izvršava dok god je uvjet ispunjen. Ako uvjet već na početku nije ispunjen, tijelo petlje se ne izvrši niti jednom. Općeniti oblik naredbe je:

```
while <test>:        # uvjet
    <naredbe1>       # tijelo
else:                # neobavezan dio, izvršava se
    <naredbe2>       # ako petlja nije završena s break
```

Dio `else` je opcionalan, a izvršava se ukoliko petlja nije prekinuta naredbom `break`. U nastavku je ponašanje ove naredbe ilustrirano na nekoliko primjera.

```
>>> while 1:    # beskonacna petlja
               print('Type Ctrl-C to stop me!')

>>> x = 'spam'
>>> while x:  # dok je niz neprazan
              print(x, end=' ')
              x = x[1:]  # odrezuje prvi znak

spam pam am m


>>> a=0; b=10
>>> while a < b: # petlja brojalica
              print(a, end=' ')
              a += 1   # isto kao a = a+1

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

Naredbe break i continue

Naredbe `break` i `continue` imaju smisla samo unutar programskih petlji. Naredba `break` uzrokuje izlazak iz (najbliže) petlje unutar koje se nalazi, dok `continue` omogućuje prijevremeni prelazak na sljedeću iteraciju petlje u kojoj se nalazi, odnosno premještanje izvođenja na zaglavlje petlje.

Ovdje ćemo spomenuti i naredbu `pass`, koja predstavlja *praznu* naredbu. Ona ne radi ništa, a koristi se na mjestima gdje sintaksa zahtijeva naredbu, a nemamo ništa "pametno" za raditi.

Oblik petlje `while`, uključujući `break` i `continue` prikazan je u nastavku, a zatim je ponašanje ilustrirano s nekoliko primjera.

```
while <test1>:
    <naredbe1>
    if <test2>: break  # izlaz
    if <test3>: continue # na početak
else:
    <naredbe2> # ako nije bio break
```

```
while 1: pass   # Type Ctrl-C to stop me!

>>> x = 10
>>> while x:
       x = x-1   # ili x -= 1
       if x % 2 != 0: continue  # preskoči neparne
       print (x, end=' ')
8 6 4 2 0

>>> while 1:
       broj = input("Upiši broj (ili 'stop'): ")
       if broj == 'stop': break
       print('kvadrat broja', broj, 'je', int(broj) ** 2)

Upiši broj (ili 'stop'): 12
kvadrat broja 12 je 144
Upiši broj (ili 'stop'): stop
```

Programska petlja for

Naredba `for` namijenjena je *iteriranju* kroz sekvence, odnosno može se primijeniti na svaki pobrojivi objekt – na znakovne nizove, liste, n-torke, rječnike i skupove, ali i za odgovarajuće oblikovane korisničke objekte. Oblik naredbe je:

```
for <varijabla_petlje> in <objekt>:   # elementi objekta
    <naredbe>   # tijelo petlje
else:
    <naredbe>   # ako nije bio break
```

Varijabli petlje pridjeljuje se vrijednost jednog po jednog elementa sekvence, i za svaki se izvršava blok naredbi. Vidimo da i petlja `for` podržava opcionalnu granu `else`, koja se izvršava pri normalnom (neprekinutom) završetku petlje. U petlji se mogu koristiti i naredbe `break` i `continue`. Ponašanje naredbe ilustrirano je primjerima.

```
>>> for x in ["spam", "eggs", "ham"]:
        print(x, end=' ')

spam eggs ham



>>> sum = 0
>>> for x in [1, 2, 3, 4]:
        sum = sum + x
>>> sum
10

>>> prod = 1
>>> for item in [1, 2, 3, 4]: prod *= item

>>> prod
24
```

Petlja `for` može se primijeniti i sa znakovnim nizovima i n-torkama.

```
>>> S, T = "lumberjack", ("and", "I'm", "okay")
>>> for x in S: print(x, end=' ')
l u m b e r j a c k

>>> for x in T: print (x, end=' ')
and I'm okay
```

Ako se prolazi kroz sekvencu n-torki, varijabla petlje može biti n-torka varijabli, kao u sljedećem primjeru.

```
>>> T = [(1, 2), (3, 4), (5, 6)]
>>> for (a, b) in T: # tuple assignment
        print(a, b)

1 2
3 4
5 6
```

Petlje se mogu gnijezditi. Za primjer napišimo skriptu koja će u listi objekata `items` potražiti objekte čije su vrijednosti jednake nekom od elemenata liste `tests`. Ovdje je ilustrirana i

primjena bloka `else`. Ako se traženi objekt pronađe, njegova se vrijednost ispisuje i unutrašnja petlja se prekida. Ako se petlja izvrti do kraja bez prekida, to znači da traženi objekt nije pronađen, pa se u bloku `else` ispisuje odgovarajuća poruka.

```
>>> items = ["aaa", 111, (4, 5), 2.01] # lista objekata
>>> tests = [(4, 5), 3.14]    # elementi koje tražimo

>>> for key in tests:  # za svaki ključ
            for item in items: # za svaki objekt
                    if item == key: # provjera podudaranja
                            print(key, "was found")
                            break
            else:
                    print(key, "not found")

(4, 5) was found
3.14 not found
```

Ovdje je dobro ilustrirati i elegantnije rješenje istog problema. Rješenje se oslanja na operator `in` kojim se provjerava pripadnost traženog objekta kolekciji. Jasno je da se, da bi se moglo odgovoriti na pitanje o pripadnosti, pretraživanje mora obaviti. No, u ovom rješenju to pretraživanje prepuštamo Pythonu. Kako su mehanizmi u Pythonu optimirani, možemo očekivati da implicitno pretraživanje bude brže nego naša realizacija petljom. Naravno, još jedna prednost ove realizacije je jezgrovitiji kôd.

```
>>> for key in tests:
            if key in items:  # prepustimo provjeru Pythonu
                    print(key, "was found")
            else:
                    print(key, "not found")

(4, 5) was found
3.14 not found
```

Funkcija range

Ugrađena funkcija `range` generira pobrojivi objekt (*engl. iterable*) s cijelim brojevima u zadanom rasponu. Funkcija se može pozvati na različite načine. Ako se pozove s jednim argumentom, generira niz cijelih brojeva od 0 do *ne uključujući* samog argumenta. Pri pozivu s dva argumenta, prvi određuje početak raspona, a drugi predstavlja prvi broj van raspona. Može se navesti i treći argument, koji određuje korak.

```
>>> range(2, 5), type(range(2,5))
(range(2, 5), <class 'range'>)

>>> list(range(5)), list(range(2,5)), list(range(0,10,2))
([0, 1, 2, 3, 4], [2, 3, 4], [0, 2, 4, 6, 8])
```

Raspon može uključivati i negativne brojeve, a može biti i silazan. Silazni raspon određen je negativnim korakom.

```
>>> list(range(-3, 3)), list(range(3, -3, -1))
([-3, -2, -1, 0, 1, 2], [3, 2, 1, 0, -1, -2])
```

Kroz generirani raspon može se iteirati petljom `for`.

```
>>> for i in range(3):
            print(i, 'Pythons')

0 Pythons
1 Pythons
2 Pythons
```

Generirani raspon može se koristiti i za indeksiranje elemenata liste.

```
>>> L = [1, 2, 3, 4, 5]
>>> for i in range(len(L)):
            L[i] += 1

>>> L
[2, 3, 4, 5, 6]
```

Funkcija zip

Ugrađena funkcija `zip` omogućava simultano iteriranje kroz više sekvenci.

Funkcija kao argumente prima jednu ili više sekvenci, a vraća iterator n-torki koje uparuju/združuju elemente tih sekvenci.

```
>>> L1 = [1,2,3]; L2 = [5,6,7]
>>> z = zip(L1,L2)
>>> type(z)
<class 'zip'>
>>> list(z)
[(1, 5), (2, 6), (3, 7)]
```

Rezultat može poslužiti za istovemeni prolazak kroz obje sekvence.

```
>>> for (x,y) in zip(L1, L2):
        print(x, y, '--', x+y)

1 5 -- 6
2 6 -- 8
3 7 -- 10
```

Funkcija može primiti i više od dvije sekvenca, pri čemu sekvence mogu biti različitih tipova, pa i različitih duljina. Pritom broj elemenata dobivenog iteratora odgovara kraćoj sekvenci.

Funkcija `zip` može se iskoristiti i za inicijalizaciju rječnika. Primjerice, ako imamo izgrađene liste ključeva i vrijednosti koje želimo upariti u rječniku, možemo to učiniti na sljedeći način.

```
>>> keys = ['spam', 'eggs', 'toast']
>>> vals = [1, 3, 5]
>>> D2 = {}
>>> for (k, v) in zip(keys, vals): D2[k] = v

>>> D2
{'toast': 5, 'eggs': 3, 'spam': 1}
```

Još elegantnije rješenje postiže se predavanjem kreiranog iteratora konstruktoru rječnika.

```
>>> D3 = dict(zip(keys, vals))

>>> D3
{'toast': 5, 'eggs': 3, 'spam': 1}
```

4.6 Funkcije

I u prethodnom tekstu susretali smo se s funkcijama, no to su bile ugrađene funkcije koje smo samo koristili. U ovom odjeljku vidjet ćemo kako se funkcije grade, te kakva su njihova svojstva i specifičnosti u Pythonu. Naime, funkcije u Pythonu ponašaju se drugačije nego funkcije u prevođenim jezicima. Prije svega, funkcije se definiraju *naredbom* `def`. Ta se naredba *izvršava* tijekom rada programa. Drugim riječima, funkcija nastaje tek prilikom izvršavanja programa, a ne kao u prevođenim jezicima prilikom prevođenja. Nadalje, stvorena funkcija je *objekt* i to punopravni (*engl. first class object*). To znači da se s funkcijama može obavljati sve što i s drugim vrstama objekata.

4.6.1 Definiranje i pozivanje funkcija

Naredbu `def` moguće je gnijezditi unutar petlji (i u svakom prolasku kroz petlju generirati novu funkciju), unutar `if` uvjeta, pa i unutar drugih naredbi `def`. Tipično se naredba `def` koristi u *modulima*, gdje se funkcije generiraju prilikom prvog učitavanja modula.

Naredba `def` ima sljedeći oblik.

```
def <name>(arg1, arg2,... argN):
    <statements>
    ...
    return <value> # nije obavezno
```

Vidimo da se radi o tipičnoj složenoj naredbi s retkom zaglavlja. Blok naredbi postaje tijelo funkcije, odnosno kôd koji se izvršava svaki put kada se funkcija pozove. Imena argumenata navedena u zaglavlju definicije funkcije pridružuju se objektima koji se pri pozivu prosljeđuju funkciji. Naredba `return` može se pojaviti bilo gdje u tijelu funkcije, a njenim izvršavanjem završava se izvršavanje funkcije i vraća rezultat pozivatelju. Naredba `return` nije obavezna, a ako je nema, funkcija vraća objekt `None`.

Već smo spomenuli da se naredba za kreiranje funkcije može naći unutar uvjetnog grananja, pa možemo ovisno o ispunjavanju nekog uvjeta stvoriti, i kasnije koristiti, različite inačice funkcije.

```
if test:
    def func(  ): # jedna definicija
        ...
else:
    def func(  ): # alternativna definicija
        ...
...
func(  ) # poziv odabrane verzije
```

Funkcija je objekt, a njeno ime je referenca. To znači da istu funkciju možemo pridružiti i nekom drugom imenu i preko njega je koristiti.

```
othername = func # objekt se pridruzuje drugom imenu
othername() # poziv iste funkcije drugim imenom
```

Primjerice, možemo definirati jednostavnu funkciju koja obavlja množenje proslijeđenih argumenata.

```
>>> def times(x, y):
            return x * y  # tijelo se izvodi pri pozivu
```

S pozivima funkcija smo se već susretali, i oni se ne razlikuju od poziva funkcija u drugim programskim jezicima. Navodi se ime funkcije, te argumenti nabrojani unutar oblih zagrada.

```
>>> times(2, 4) # argumenti u zagradama
8
```

Povratna vrijednost može se i pohraniti, odnosno pridružiti nekom imenu.

```
>>> x = times(3.14, 4)
>>> x
12.56
```

Valja uočiti da se pri definiciji funkcije ne zadaju tipovi argumenata. Stoga se funkcija može pozvati s argumentima proizvoljnih tipova. U našem slučaju funkcije zamišljene za množenje argumenata, možemo pokušati proslijediti znakovni niz i broj.

```
>>> times('Ni', 4)
'NiNiNiNi'
```

Vidimo da se funkcija uspješno izvršila, no rezultat je ponovljeni znakovni niz. Operacije u funkciji su obavljene u skladu s tipovima operanada, čime je manifestiran inherentni *polimorfizam* u Pythonu. Jedini preduvjet da se funkcija uspješno izvrši za neki skup argumenata je da su operacije u funkciji definirane za proslijeđene argumente.

Kao još jednu ilustraciju inherentnog polimorfizma, definirajmo funkciju koja određuje presjek dviju sekvenci. Funkcija kao argumente očekuje dvije sekvence, kroz jednu se iterira, te se za svaki element provjerava postoji li jednak u drugoj sekvenci. Na kraju se vraća lista elemenata koji su zajednički za obje sekvence.

```
>>> def intersect(seq1, seq2):
        res = [  ]    # lokalna varijabla, prazna lista
        for x in seq1:         # scan seq1!
            if x in seq2:      # common item?
                res.append(x)  # add to end
        return res
```

Funkciju možemo pozvati s dva znakovna niza.

```
>>> s1 = "SPAM"
>>> s2 = "SCAM"
>>> intersect(s1, s2) # strings
['S', 'A', 'M']
```

Kako se u funkciji `intersect` pojavljuje samo iteriranje kroz prvi argument u petlji `for`, te ispitivanje pripadnosti drugom argumentu operatorom `in`, funkcija se može pozivati i s drugim tipovima objekata. Primjerice, možemo tražiti presjek liste i n-torke.

```
>>> intersect([1, 2, 3], (1, 4))  # mixed types
[1]
```

Doseg varijabli

Imena nastaju kad im se prvi put dodijeli vrijednost. Na temelju mjesta u programu gdje se dodjeljivanje vrijednosti obavlja, Python povezuje ime s odgovarajućim *prostorom imena*, odnosno određuje *doseg* (*engl. scope*) u kojem je ime vidljivo.

Imena definirana unutar funkcije su *lokalna*, odnosno vidljiva su samo unutar funkcije. Time je osigurano da ne može doći do sukoba imena s imenima van funkcije. Svaki poziv funkcije stvara *novi* lokalni doseg.

Osim lokalnog, Python poznaje i *globalni* doseg, a taj se pojam odnosi na doseg unutar modula. Atributi modula predstavljaju globalne varijable, te se globalni doseg proteže najviše kroz jednu datoteku (modul).

Razrješavanje imena u Pythonu može se sažeti u tri jednostavna pravila:

- referenca imena traži se u najviše četiri dosega (LEGB), i to sljedećim redom:
  - lokalni – L
  - doseg obuhvaćajuće (*engl. enclosing*) funkcije – E
  - globalni – G
  - ugrađeni (*engl. built-in*) – B
- dodjela vrijednosti stvara ili mijenja *lokalno* ime
- globalne deklaracije mapiraju imena u doseg obuhvaćajućeg *modula*

Funkcije mogu *koristiti* imena iz obuhvaćajućeg(E) ili globalnog (G) dosega, ali ih *ne mogu mijenjati* ako ih ne *deklariraju* globalnima. Valja napomenuti da se ovaj način razrješavanja primjenjuje za *jednostavna* imena varijabli, dok se za kvalificirana imena (oblika `objekt.atribut`) primjenjuju drugačija pravila.

Ilustrirajmo ove koncepte na nekoliko primjera. Prvi primjer pokazuje da se unutar funkcije vide globalna imena i mogu se koristiti.

```
>>> X = 99   # globalno ime

>>> def func(Y):
               Z = X + Y   # globalni X
               return Z

>>> func(1)
100
```

U primjeru su globalna imena `X` i `func`, a lokalna `Y` i `Z`. Lokalna imena postoje samo tijekom izvršavanja funkcije. Argumenti (ovdje `Y`) se uvijek prosljeđuju *pridruživanjem*.

Budući da se imena traže zadanim redoslijedom dosega (LEGB), uvođenje istog imena u lokalni doseg *zakriva* globalno ime.

```
>>> X = 88          # globalno ime

>>> def func(  ):
               X = 99    # dodjela --> lokalno ime

>>> func()
>>> X  # ispisuje se globalni
88
```

U ovom primjeru, dodjela vrijednosti imenu `X` unutar funkcije stvara *lokalnu* varijablu `X`. To ime nema nikakve veze s globalnom varijablom `X` u modulu izvan funkcije.

Ako ipak želimo obaviti izmjenu globalne varijable unutar funkcije, moramo to eksplicitno naglasiti. To se postiže korištenjem naredbe `global`, koja predstavlja deklaraciju koja se odnosi na programski blok u kojem se nalazi. Iza ključne riječi `global` navode se imena varijabli odvojena zarezima, koja se mapiraju u doseg obuhvaćajućeg *modula* (globalni doseg).

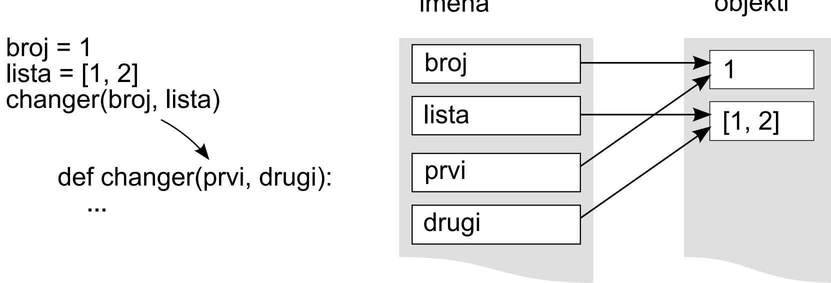
```
>>> X = 88          # globalno ime

>>> def func(  ):
            global X
            X = 99  # sada mijenja globalnu varijablu

>>> func()
>>> X   # globalna varijabla je izmijenjena
99
```

Ovdje valja istaknuti da naredba `global` mapira imena u globalni, a *ne u obuhvaćajući* doseg. To možemo ilustrirati sljedećim primjerom.

```
>>> def vanjska():
           x = 1
           def unutarnja():
                   global x  # doseg modula!
                   x = 2
                   print("unutarnja:", x)
           unutarnja()
           print("vanjska:", x)

>>> vanjska()
unutarnja: 2
vanjska: 1

>>> x  # ovo je globalni x
2
```

Da bi se omogućilo mapiranje imena u *obuhvaćajući* doseg, u Pythonu 3 uvedena je nova naredba `nonlocal`. Njeno djelovanje možemo ilustrirati sljedećim primjerom.

```
>>> def vanjska():
           x = 1
           def unutarnja():
                   nonlocal x  # obuhvaćajući doseg!
                   x = 2
                   print("unutarnja:", x)
           unutarnja()
           print("vanjska:", x)

>>> vanjska()
unutarnja: 2
vanjska: 2

>>> x  # nepoznato ime
...
NameError: name 'x' is not defined
```



Slika 4.5: Argumenti funkcije kao dijeljene reference.

4.6.2 Prenošenje argumenata u funkcije

Argumenti se u funkcije prenose *dodjeljivanjem* proslijeđenih objekata lokalnim imenima. Lokalna imena su reference na potencijalno dijeljene objekte koje referencira i pozivatelj. Dodjela novog objekta *imenu* argumenta unutar funkcije *ne utječe* na pozivatelja, jer je *doseg* imena je unutar funkcije. No, ako je argument izmjenljivi objekt, njegova izmjena u funkciji može utjecati na pozivatelja.

Iako se zapravo prosljeđuju reference, ponašanje argumenata ovisi o njihovoj izmjenljivosti. Tako se nepromjenljivi argumenti ponašaju *kao da su* proslijeđeni kao vrijednosti (iako se ne stvara kopija). S druge strane, izmjenljivi argumenti se ponašaju se kao da su proslijeđeni kao reference. Ovo ponašanje možemo ilustrirati primjerom.

```
>>> def changer(prvi, drugi):
            prvi = 2          # mijenja lokalnu vrijednost
            drugi[0] = 'burek'  # mijenja dijeljeni objekt

>>> broj = 1              # neizmjenljivi
>>> lista = [1, 2]        # izmjenljivi
>>> changer(broj, lista)

>>> broj, lista
(1, ['burek', 2])
```

U ovom primjeru, `prvi` je lokalno ime, kojem se unutar funkcije pridružuje novi objekt.Pritom veza globalnog imena `broj` i objekta `1` nije narušena. No, naredba `drugi[0] = 'burek'` mijenja element liste `lista`. Radi se naprosto o *dijeljenim referencama*, kao što ilustrira slika 4.5.

Ako želimo spriječiti izmjenu izmjenljivih objekata proslijeđenih u funkciju, možemo proslijediti eksplicitnu *kopiju*.

```
>>> lista = [1, 2]
>>> changer(broj, lista[:])  # prosljeđuje se kopija
```

Naravno, kopiju može napraviti i sama funkcija.

```
>>> def changer(prvi, drugi):
            drugi = drugi[:]        # kopija!
            prvi = 2
            drugi[0] = 'burek' # izmjena na kopiji
```

Još jedan način za sprječavanje neželjene izmjene objekta je njegovom pretvorbom u nepromjenljivi tip. Ako funkcija pokuša mijenjati proslijeđeni objekt, dogodit će se iznimka kao u sljedećem primjeru.

```
>>> changer(broj, tuple(lista))  # n-torka
... TypeError:
'tuple' object does not support item assignment
```

Ipak, treba biti svjestan da izvorna lista može sadržavati i promjenljive objekte – niti kopija niti pretvorba u n-torku ne može spriječiti mijenjanje tih ugniježdenih objekata!

Oponašanje izlaznih argumenata

Python nema mehanizam poziva s referencom (*engl. call-by-reference*), no možemo ga oponašati tako da se izlazni parametri vrate kao n-torka, koja se zatim dodjeljuje izvornim argumentima u pozivatelju. Rezultat je izmjena izvornih argumenata.

```
>>> def multiple(x, y):
        x = 2   # mijenjaju se lokalna imena
        y = [3, 4]
        return x, y

>>> X = 1
>>> L = [1, 2]
>>> X, L = multiple(X, L) # povratna dodjela rezultata

>>> X, L
(2, [3, 4])
```

Imenovanje argumenata

Uobičajeni način dodjele imena argumentima funkcije je *pozicijski.* Pri pozivu se argumenti navode istim redoslijedom kao u definiciji funkcije i mora ih biti jednak broj. Međutim, Python nudi i druge mogućnosti, koje omogućuju dodatnu fleksibilnost. Radi se o tri koncepta:

- povezivanje argumenata imenom (*keywords*) pri pozivu funkcije, čime se omogućuje navođenje argumenata proizvoljnim redoslijedom;
- zadavanje pretpostavljenih vrijednosti (*default*) za argumenate pri definiciji funkcije, što omogućuje pozivanje funkcije s manjkom argumenata;
- preuzimanje promjenjivog broja argumenata (*varargs*), što omogućuje pozivanje funkcije s različitim brojem argumenata.

Sljedeći primjeri ilustriraju pozicijsko i imenovano povezivanje argumenata pri pozivu funkcije `f`. Vidimo da se način pozivanja ne odražava na definiciju funkcije, već se manifestira samo u njenom pozivu.

```
>>> def f(a, b, c): print(a, b, c)

>>> f(1, 2, 3)  # pozicijsko povezivanje
1 2 3
>>> f(c=1, b=3, a=2)  # povezivanje imenom
2 3 1
>>> f(3, c=2, b=1)  # miješano
3 1 2
```

Broj argumenata u oba načina pozivanja mora odgovarati broju argumenata u definiciji funkcije. U slučaju miješanog povezivanja imena, najprije se dodjeljuju pozicijski argumenti, s lijeva udesno. Zato sljedeći poziv nije ispravan. Python najprije povezuje vrijednost 3 s argumentom a, a nakon toga se pokušava imenovanim povezivanjem istom argumentu dodijeliti druga vrijednost.

```
>>> f(3, c=2, a=1)  # ne valja!
TypeError:
f() got multiple values for keyword argument 'a'
```

Pretpostavljene vrijednosti argumenata

Pretpostavljene vrijednosti argumenata omogućavaju da neki argumenti pri pozivu budu opcionalni. Ako se argument ne proslijedi, dodjeljuje mu se pretpostavljena vrijednost. Pretpostavljene vrijednosti zadaju se u definiciji funkcije. Mogućnosti pozivanja funkcije s pretpostavljenim vrijednostima ilustrirane su u sljedećim primjerima.

```
>>> def f(a, b=2, c=3): print(a, b, c)

>>> f(1)  # moramo proslijediti vrijednost za a
1 2 3

>>> f(a=1)   # može i po imenu
1 2 3

>>> f(1, 4)
1 4 3

>>> f(1, 4, 5)
1 4 5

>>> f(1, c=6)  # imenovanje pomaže
1 2 6
```

Proizvoljna lista argumenata

Pyton nudi i mogućnost definiranja funkcija koje se mogu pozivati s proizvoljnim brojem argumenata. Tu se razlikuju dva oblika. U prvom obliku prekobrojni argumenti prikupljaju se u n-torku. Takva n-torka se označava zvjezdicom prije imena argumenta u definiciji funkcije, kao u sljedećem primjeru.

```
>>> def f(*args): print(args)

>>> f()
()

>>> f(1)
(1,)

>>> f(1,2,3,4)
(1, 2, 3, 4)
```

Drugi oblik predviđen je za pridruživanje argumenata imenom, a proslijeđeni argumenti se prikupljaju u rječnik. Rječnik se označava s dvije zvjezdice prije imena argumenta.

```
>>> def f(**args): print(args)

>>> f()
{}
>>> f(a=1, b=2)
{'a': 1, 'b': 2}
```

Moguće je i kombiniranje ova dva načina prikupljanja argumenata, uz napomenu da se najprije moraju navesti pozicijski, a zatim imenovani argumenti.

```
>>> def f(a, *pargs, **kargs): print(a, pargs, kargs)

>>> f(1, 2, 3, x=1, y=2)
1 (2, 3) {'y': 2, 'x': 1}
>>>
>>> f(1, 2, 3, x=1, y=2, 4, 5)  # ovo ipak ne ide
SyntaxError: non-keyword arg after keyword arg
```

4.6.3 Bezimene funkcije

Već smo spomenuli da su funkcije u Pythonu objekti prvog reda., koji se stvaraju naredbom `def`. No, u nekim situacijama trebamo definirati jednostavnu funkciju koja će se pojaviti samo na jednom mjestu. Python za takve slučajeve nudi mehanizam stvaranja bezimenih (anonimnih) funkcija, korištenjem *lambda izraza.* Kako se radi o *izrazu* a ne naredba, takva definicija funkcije može se pojaviti na mjestima gdje sintaksa ne dozvoljava naredbu. Tipičan primjer je unutar liste ili u pozivu funkcije. Lambda izraz vraća vrijednost (novu funkciju), kojoj se može, ali i ne mora, dodijeliti ime. Opći oblik lambda izraza je:

```
lambda argument1, argument2,... argumentN : izraz
```

Nakon ključne riječi `lambda` navode se argumenti, koji se koriste u izrazu. Generirana funkcija radi jednako kao i funkcija stvorena naredbom `def`, s time da je tijelo lambda funkcije ograničeno na *jedan izraz.* Iz toga je jasno da se na ovaj način mogu graditi samo vrlo jednostavne funkcije. Kao primjer, kreirajmo lambda funkciju koja vraća zbroj svoja tri argumenta. Novom funkcijskom objektu možemo dodijeliti ime i nakon toga se funkcija koristi jednako kao da je definirana na uobičajen način.

```
>>> f = lambda x, y, z: x + y + z
>>> f(2, 3, 4)  # normalan poziv
9
```

I u lambda funkcijama se mogu koristiti i pretpostavljene vrijednosti argumenata.

```
>>> x = (lambda a="fee", b="fie", c="foe": a + b + c)
>>> x("wee")
'weefiefoe'
```

Također, vrijede i uobičajena pravila za razrješavanje dosega imena, što možemo ilustrirati sljedećim primjerom.

```
>>> def knights( ):
        title = 'Sir'
        action = (lambda x: title + ' ' + x)
        return action
```

```
>>> act = knights(  )
>>> act('robin')
'Sir robin'
```

Lambda izrazi omogućuju nam da ugradimo definiciju funkcije unutar kôda koji ju koristi. Na taj način možemo definirati, primjerice, povratne funkcije (*engl. callback handlers*) u samom pozivu za registriranje. Također, možemo izgraditi tablice grananja pohranjene u listi ili rječniku, kao u sljedećem primjeru.

```
>>> L = [(lambda x: x**2), (lambda x: x**3),
                           (lambda x: x**4)]

>>> for f in L: print(f(2), end=' ')
4 8 16
>>> L[0](3)
9
```

Primjerice, korištenjem rječnika čiji su elementi funkcije, možemo ostvariti uvjetno grananje.

```
>>> key = 'got'
>>> {'already': (lambda: 2 + 2),
     'got':     (lambda: 2 * 4),
     'one':     (lambda: 2 ** 6)}[key]()
8
```

4.7 Još malo o modulima

Koncept modula smo upoznali u odjeljku 4.2. Svaka datoteka s nastavkom imena `".py"` je modul i može se učitati naredbama `import` ili `from`. Prilikom *prvog* učitavanja, izvršavaju se naredbe modula i stvara objekt modul. Učitani modul je, kao i funkcija, objekt kao i svaki drugi. Globalna imena modula (njegovi atributi) postaju dostupna modulu koji ga je učitao. Ako je modul učitan s `import modul`, njegovim atributima se pristupa *kvalifikacijom*: `modul.atribut`. Ako je modul učitan s `from modul import atribut`, imena atributa *kopiraju se* u prostor imena učitavatelja. Naredbom `from modul import *`, kopiraju se imena svih atributa učitanog modula. Ukoliko je potrebno ponovo učitati modul (npr. izmijenili smo ga), koristi se funkcija `imp.reload` iz modula `imp`.

4.7.1 Moduli i prostori imena

Kao i `def`, `import` i `from` su *naredbe*. Modul i njegova imena nisu raspoloživi dok se naredba učitavanja ne izvrši. Kao i `def`, `import` i `from` predstavljaju implicitna dodjeljivanja. Kada se koristi `import`, cijeli modul kao objekt se dodjeljuje jednom imenu (koje odgovara imenu modula). S druge strane, ako se koristi naredba `from`, također se stvara objekt modul, ali se imena navedenih atributa kopiraju u prostor imena učitavajućeg modula. Ta imena postaju reference na atribute modula, a dodjela novog objekta nekom od tih imena ne utječe na atribut učitanog modula. Ako se radi o izmjenljivom atributu, preko imena u učitavatelju može se utjecati na odgovarajući atribut učitanog modula. To ćemo ilustrirati na nekoliko primjera.

Definirajmo jednostavni modul s dva atributa. Prvi je neizmjenljiv, a drugi izmjenljiv.

```
# modul small.py
x = 1
y = [1, 2]
```

Modul učitavamo naredbom `from`. Imena navedenih atributa se kopiraju, preko njih možemo pristupiti objektima učitanog modula. Međutim, naredba `from` ne stvara ime koje bi bilo referenca na učitani modul. Ako nakon toga izvršimo naredbu `import`, naredbe modula se ne izvršavaju ponovo (objekt modul je već ranije stvoren), ali se stvara ime kao referenca na učitani modul. Nakon toga je moguće pristupiti atributima učitanog objekta i vidjeti da je obavljena izmjena liste. Naravno, dodjela nove vrijednosti imenu `x` nije utjecala na odgovarajući atribut učitanog modula.

```
>>> from small import x, y      # kopiramo imena
>>> x = 42             # mijenja se lokalna kopija
>>> y[0] = 42          # mijenjamo izvorni objekt

>>> small.x      # učitanom modulu nije pridruženo ime
...
NameError: name 'small' is not defined

>>> import small   # modul se ne učitava ponovo, samo se stvara ime
>>> small.x    # to je drugi x
1
>>> small.y    # ali listu y dijelimo
[42, 2]
```

Korištenjem funkcije `imp.reload` iz modula `imp` modul se *ponovo* učitava, izvršavaju se njegove naredbe i nastaje novi objekt, s obnovljenim vrijednostima atributa.

```
>>> import imp
>>> imp.reload(small)  # nanovo stvaramo objekt modul
>>> small.y  # obnovljen!
[1, 2]

>>> y  # ovaj pokazuje na stari objekt!
[42, 2]
```

Za učitani modul globalni prostor imena je datoteka u kojoj je definiran i on ne vidi prostor imena modula koji ga je učitao. Za ilustraciju, definirajmo dva modula, od kojih jedan učitava drugi, a oba imaju istoimene globalne varijable.

```
# modul moda.py
X = 88   # globalna varijabla za ovaj modul

def f( ):
  global X  # mijenja globalnu var OVOG MODULA
  X = 99    # ne vidi imena u drugim modulima
```

```
# modul modb.py
X = 11    # i on ima svoj globalni X

import moda
moda.f( ) # postavlja moda.X, a ne X iz ovog modula
print(X, moda.X)
```

```
$ python3 modb.py
11 99
```

Moduli se mogu i gnijezditi, odnosno međusobno se učitavati u više razina. Definirajmo tri modula, od kojih `mod2` učitava `mod3`, a `mod1` učitava `mod2` i `mod3`. U sva tri modula pojavljuju se istoimene globalne varijable. U primjeru možemo vidjeti kako se kvalifikacijom može pristupiti atributima učitanih modula.

```
# modul mod3.py
X = 3
```

```
# modul mod2.py
X = 2
import mod3
print(X, mod3.X, end = " - ")
```

```
# modul mod1.py
X = 1
import mod2
print(X, mod2.X, end = " - ")
print(mod2.mod3.X)   # iz ugniježdenog mod3
```

```
$ python3 mod1.py
2 3 - 1 2 - 3
```

4.7.2 Skrivanje podataka u modulu

Kao što smo vidjeli, modul u Pythonu eksportira sva imena dodijeljena u najvišoj razini datoteke u kojoj se nalazi i nema načina za sprječavanje klijenta da promijeni ime u modulu ako to baš želi. U Pythonu postoji mehanizam skrivanje podataka u modulu, no to je *konvencija* a ne sintaktičko ograničenje. Mehanizam skrivanja podataka temelji se na nekoliko pravila. Prije svega, naredba `from *` zapravo *ne kopira sva* imena učitanog modula. Imena koja započinju znakom `"_"` (podvlaka) ne kopiraju. Namjera ovog pravila je smanjivanje "onečišćenja" prostora imena (ne kopiraju se imena koja nisu predviđena za korištenje od strane klijenta), a ne *zabrana* izmjene. No, ako klijent to želi, izmjena je i za takva imena moguća. Preduvjet je da dobijemo referencu na učitani modul, a to se lako postiže naredbom `import`, kao što smo pokazali u prethodnom odjeljku. Demonstrirajmo skrivanje podataka primjerom. Ime atributa koje započinje podvlakom ne kopira se u prostor imena učitavajućeg modula, no ako dobijemo referencu na učitani modul naredbom `import`, i takvim atributima možemo pristupiti.

```
# modul skrivanje.py
mene_vidiš = 3
_mene_ne_vidiš = 11
```

```
>>> from skrivanje import *

>>> mene_vidiš
3
>>> _mene_ne_vidiš
...
NameError: name '_mene_ne_vidiš' is not defined

>>> import skrivanje     # ovako ga vidimo
>>> skrivanje._mene_ne_vidiš
11
```

Još jedan način za ostvarenje skrivanja podataka temelji se na definiranju liste znakovnih nizova s imenima koja želimo eksportirati (učiniti vidljivima učitavatelju). Ta lista mora biti imenovana `__all__` u najvišoj razini modula. I ovaj oblik skrivanja odnosi se *samo* na `from *` oblik učitavanja modula. Pri izvršavanju te naredbe, Python najprije traži listu `__all__` unutar modula, a ako takva lista nije definirana, kopiraju se sva imena bez vodeće podvlake.

```
# modul skrivanje2.py
ja_sam_vidljiv = 7
ja_bih_bio_nevidljiv = 13
_vidiš_li_me = 99

__all__ = ["ja_sam_vidljiv", "_vidiš_li_me"]
```

```
>>> from skrivanje2 import *
>>> ja_sam_vidljiv, _vidiš_li_me
(7, 99)
>>> ja_bih_bio_nevidljiv
...
NameError: name 'ja_bih_bio_nevidljiv' is not defined

>>> import skrivanje2  # opet ima lijeka
>>> skrivanje2.ja_bih_bio_nevidljiv
13
```

4.8 Razredi

I do sada smo koristili pojam *objekt*, a upoznali smo i objekte koji imaju *metode*. Python je dosljedan objektno orijentirani jezik koji podržava i oblikovanje novih (korisničkih) tipova objekata – *razreda*, te *nasljeđivanje*. Stvaranje novog tipa objekta ostvaruje se naredbom `class`, a razredi su poput funkcija i modula punopravni objekti. Novi tipovi objekata mogu se oblikovati tako da se ponašaju slično ugrađenim tipovim objektima. Nasljeđivanje je mehanizam koji podupire prilagođavanje i višestruko korištenje koda. Korištenje objektno orijentiranog programiranja u Pythonu je neobavezno, ali kako ga koriste mnogi alati, dobro je steći barem grubu sliku o ovim konceptima.

4.8.1 Stvaranje razreda

Naredba `class` generira novi *razred*. Unutar razreda mogu se nalaziti naredbe koje stvaraju atribute razreda. Atributi mogu biti i funkcije.

```
>>> class C:
      x = 1   # stvaraju se atributi
      def f():  # to mogu biti i funkcije
                print("razred C")
```

Razred je objekt prvog reda, a njegovo ime je referenca preko koje možemo pristupati njegovim atributima. Možemo koristiti ili mijenjati podatkovne atribute, ali i pozivati funkcije koje su definirane unutar razreda.

```
>>> C.x  # možemo pristupati atributima
1
>>> C.f() # pozivati funkcije
razred C
```

Razred možemo pridružiti drugom imenu i preko njega pristupati atributima.

```
>>> ime = C  # pridružiti novom imenu
>>> ime.f()  # pristupati preko novog imena
razred C
```

4.8.2 Stvaranje primjerka

Primjerak se stvara pozivanjem razreda, koje je sintaksno slično pozivanju funkcije. Pritom primjerak nasljeđuje sve atribute razreda.

```
>>> primjerak = C()

>>> type(primjerak)
<class '__main__.C'>

>>> primjerak.x  # atributi se nasljeđuju
1

>>> primjerak.f()  # ime je poznato, ali funkciju
                   # ne možemo samo tako pozvati
...
TypeError: f() takes no arguments (1 given)
```

U gornjem primjeru vidimo da je ime funkcije koja je definirana unutar razreda poznato i unutar primjerka. No, pozivanje funkcije na ovaj način ne uspijeva. U čemu je problem saznat ćemo uskoro, kad se upoznamo načinom definiranja metoda.

Nakon stvaranja primjerka, i njemu se mogu definirati atributi. Oni su vidljivi samo tom primjerku i nemaju veze s atributima drugih primjeraka istog razreda.

```
>>> primjerak.dodatak = 123
>>> primjerak.dodatak
123

>>> C.dodatak
...
AttributeError: type object 'C' has no attribute 'dodatak'
```

4.8.3 Metode

Funkcije vezane za primjerke razreda nazivaju se *metodama*. Metode smo već koristili kod nekih ugrađenih tipova, poput znakovnih nizova, listi i rječnika. Metoda je funkcija koja se definira unutar razreda, ali ima specifičnost da se kao prvi argument pojavljuje referenca na objekt – primjerak razreda iz kojega se poziva. Uobičajeno je i preporučljivo (iako ne i obavezno) taj argument nazvati `self`. Njegova je uloga kao `this` u C++-u ili Javi, ali, no u Pythonu je `self` uvijek eksplicitno naveden u definiciji metode.

Pri inicijalizaciji novog primjerka razreda poziva se posebna, inicijalizacijska metoda `__init__()`. Ona se može eksplicitno definirati, ili Python generira podrazumijevani oblik. Pri pozivu metode, argument `self` se na navodi, već ga Python samostalno prosljeđuje kao referencu na primjerak iz kojega je pozvana. To objašnjava i neuspješno pozivanje funkcije `f` koja je u razredu `C` definirana bez argumenata. Kada je pozvana iz primjerka, Python je automatski generirao argument `self`, pa se broj argumenata nije podudarao s definicijom funkcije. Sljedeći

primjer ilustrira definiciju metoda, te njihovo pozivanje iz primjerka. Vidimo da je definirana i inicijalizacijska metoda, koja pri stvaranju primjerka inicijalizira njegove atribute.

```
>>> class DrugiRazred:
        def __init__(self, naziv):
          self.ime = naziv
        def jedna_metoda(self):
          print(self.ime)

>>> drugi_primjerak = DrugiRazred("Dvojka")
>>> drugi_primjerak.jedna_metoda() # self se ne navodi
Dvojka
```

4.8.4 Nasljeđivanje

Pri definiciji razreda, može se zadati nasljeđivanje atributa nekog drugog, već definiranog razreda. Sintaksno, nasljeđivanje se specificira navođenjem razreda koje novi razred nasljeđuje unutar zagrada koje slijede njegovo ime.

```
class NoviRazred(Mama, Tata):
  ...
```

Primjerice, možemo definirati dva razreda `C2` i `C3`, a zatim ih iskoristiti u kreiranju novog razreda `C1`. Radi lakšeg praćenja zbivanja prilikom stvaranja primjeraka, u razredima ćemo definirati nekoliko atributa s prikladnim porukama. Nakon toga, stvaraju se dva primjerka razreda `C1`. Kako bismo ih razlikovali, inicijalizacijska metoda za svaki primjerak postavlja atribut s imenom.

```
>>> class C2:
      x = "x u razredu C2"
      z = "z u razredu C2"
```

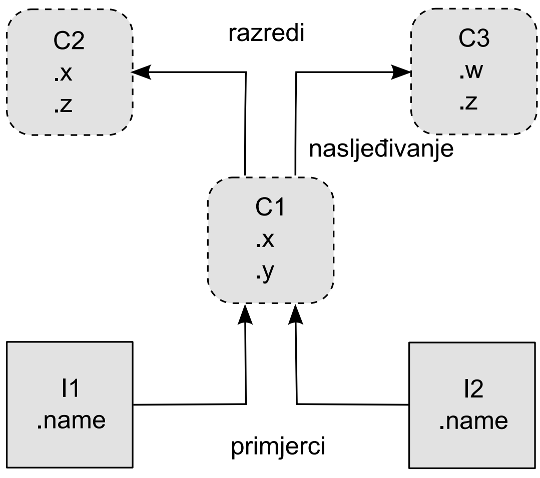
```
>>> class C3:
      w = "w u razredu C3"
      z = "z u razredu C3"
```

```
>>> class C1(C2, C3):  # nasljeđuje C2 i C3
      x = "x u razredu C1"  # nadomješta naslijeđeni x
      y = "y u razredu C1"
      def __init__(self, ime):
         self.name = ime
```

```
>>> I1 = C1("prvi")
>>> I2 = C1("drugi")
```

Nasljeđivanje kao pretraživanje

Objekt ima *atribute*, a pristup atributu započinje izrazom poput `objekt.atribut`. Kad se ovakav izraz primijeni na objekt izveden na temelju naredbe `class`, pokreće se operacija *pretraživanja* stabla povezanih objekata, pri čemu se traži *prvo pojavljivanje* navedenog atributa. Stablo se pretražuje od dna prema vrhu, krenuvši od samog objekta, pa zatim kroz klase iz kojih je izveden. Objekti niže u stablu *nasljeđuju* atribute objekata smještenih više u stablu. U Pythonu



Slika 4.6: Nasljeđivanje se ostvaruje pretraživanjem.

je ovaj koncept doista i izveden kao pretraživanje. Kôd koji definira razrede i primjerke određuje stablastu strukturu povezanih objekata, a Python tijekom izvođenja programa pokreće pretraživanje svaki put kada naiđe na izraz tipa `objekt.atribut`.

Na slici 4.6 je prikazano stablo pretraživanja koje odgovara prethodnom primjeru s razredom `C1`koji nasljeđuje razrede `C2` i `C3`.

Razred je objekt koji svojim atributima(podaci i funkcije) definira ponašanje koje nasljeđuju svi *primjerci* koji se iz tog razreda generiraju. *Primjerak* predstavlja konkretni objekt u domeni programa, a njegovi atributi sadrže podatke koji variraju od primjerka do primjerka. Primjerak nasljeđuje atribute svojeg razreda, a razred atribute svojih *nad-razreda.* Ovi odnosi odgovaraju stablu pretraživanja. S obzirom na to da pretraživanje napreduje od dna prema vrhu, podrazred može izmijeniti (*engl. override*) ponašanje definirano u nadrazredu.

Kada se pokuša pristupiti atributu `w` primjerka `I2`, Python započinje pretraživanje od primjerka `I2`, prema vrhu stabla, pretraživanjem povezanih objekata `I2, C1, C2, C3`. Pretraživanje se zaustavlja prvim nailaskom na atribut odgovarajućeg imena. U ovom slučaju to je `C3.w`. Drugim riječima, primjerak `I2` nasljeđuje atribut `w` od razreda `C3`.

```
>>> I2.w   # pronalazi ga u C3
'w u razredu C3'
```

Referenciranje drugih atributa završava pretraživanjem drugih grana stabla, kao što je ilustrirano u sljedećim primjerima.

```
>>> I1.x, I2.x   # overriding
('x u razredu C1', 'x u razredu C1')

>>> I1.y
'y u razredu C1'

>>> I1.z  # pretraživanje s lijeva udesno
'z u razredu C2'

>>> I1.name, I2.name  # svaki ima svoje ime
('prvi', 'drugi')
```

Redoslijed navođenja roditeljskih razreda određuje redoslijed pretraživanja u stablu.

```
class C1(C2, C3): ...  # nasljedjuje C2 i C3
```

Zbog načina na koji se izvodi pretraživanje/nasljeđivanje, izbor objekta za koji se veže atribut je ključan, on određuje doseg imena. Atributi vezani za primjerke, odnose se samo na konkretan primjerak, dok atribute vezane za razred *dijele* svi podrazredi i svi primjerci izvedeni iz tog razreda. Atributi koji su vezani uz razred obično se stvaraju dodjelom u naredbama unutar definicije razreda. Atributi vezani za primjerke obično se stvaraju dodjelama posebnom argumentu `self` koji se prosljeđuje metodama.

```
class C1(C2, C3):
    ...
    def setname(self, who): # još jedna metoda
       self.name = who    # self je I1 ili I2
```

Atributi se primjerku mogu dodati/mijenjati i izravno.

```
>>> I2.novi_atribut = "nešto novo"
>>> I2.novi_atribut
'nešto novo'
```

Definiranje i korištenje razreda ilustrirat ćemo na još jednom primjeru, u kojem ćemo oblikovati jednostavnu bazu podataka o zaposlenicima. Osnovni razred definira pretpostavljeno ponašanje zajedničko svim vrstama zaposlenika u poduzeću:

```
class Employee:  # temeljna nadklasa
    def computeSalary(self):   # zajedničko
        ...
    def giveRaise(self):
        ...
    def promote(self):
        ...
    def retire(self):
        ...
```

Prilagodba ponašanja za neku vrstu zaposlenika naziva se specijalizacijom, a postiže se tako da izvedeni podrazred zamjenjuje (*engl. override*) neku metodu, u ovom primjeru metodu za izračun plaće.

```
class Engineer(Employee):     # izvedeni razred
    def computeSalary(self):  # prilagodba
        ...
```

Primjerci mogu biti generirani iz osnovnog ili iz izvedenog razreda.

```
bob = Employee(  )  # osnovno ponašanje
mel = Engineer(  )  # izmijenjeno ponašanje
```

Bibliografija

[1] Stephen Kochan and Patrick Wood, *Unix Shell Programming*, Sams, 3rd edition, 2003.

[2] Randal L. Schwartz, Tom Phoenix, and brian d foy, *Learning Perl*, O'Reilly, 5th edition, 2008.

[3] Mark Lutz, *Learning Python*, O'Reilly, 4th edition, 2009.