Ruby programski jezik

Suhić, Domagoj

Undergraduate thesis / Završni rad

2016

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **Josip Juraj Strossmayer University of Osijek, Department of Mathematics / Sveučilište Josipa Jurja Strossmayera u Osijeku, Odjel za matematiku**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:126:628340

Sveučilište J.J.Strossmayera u Osijeku
Odjel za matematiku
Preddiplomski studij matematike

Domagoj Suhić

Ruby programski jezik
Završni rad

Osijek, 2016.

Sveučilište J.J.Strossmayera u Osijeku
Odjel za matematiku
Preddiplomski studij matematike

Domagoj Suhić

Ruby programski jezik
Završni rad

Voditelj: izv. prof. dr. sc. Domagoj Matijević

Osijek, 2016.

Sažetak. U ovom završnom radu proći ćemo kroz osnove programskog jezika Ruby. Predstavit ćemo leksičku strukturu, osnovne tipove podataka, izjave i izraze Rubyja. Na kraju ćemo proći kroz osnove objektno orijentiranog programiranja u Rubyju. Paralelno s tim radit ćemo usporedbu s programskim jezikom Python.

Ključne riječi: Ruby, Python, token, izraz, izjava, metoda, blok, proc, lambda, objektno orijentirano programiranje, objekt, klasa, modul.

Abstract. In this work, we'll cover the basics of the Ruby programming language. We'll present the lexical structure, basic data structures, expressions and statements in Ruby. At the same time, we'll be making a comparison with the Python programming language.

Keywords: Ruby, Python, token, expression, statement, method, block, proc, lambda, object oriented programming, object, class, module.

Sadržaj

1 Uvod **3**
- 1.1 Programski Jezik Ruby . . . 3
- 1.2 Objektno Orijentirano Programiranje . . . 3

2 Osnovne Karakteristike **5**
- 2.1 Leksička Struktura . . . 6
- 2.2 Sintaktička Struktura . . . 7

3 Tipovi Podataka **8**
- 3.1 Booleani . . . 8
- 3.2 Brojevi . . . 8
- 3.3 Tekst . . . 8
- 3.4 Polja . . . 11
- 3.5 Hashevi . . . 12
- 3.6 Rasponi . . . 12

4 Izrazi **14**

5 Kontrolne Strukture **16**
- 5.1 Uvjetne Strukture . . . 16
- 5.2 Petlje i Iteratori . . . 18
- 5.3 Blokovi . . . 21
- 5.4 Izjave . . . 21
- 5.5 Iznimke . . . 21
- 5.6 BEGIN i END . . . 24

6 Metode **25**
- 6.1 Lambda i Proc . . . 27

7 Objektno Orijentirano Programiranje u Rubyju **30**
- 7.1 Klase . . . 30
- 7.2 Moduli . . . 36
- 7.3 Eigenclass . . . 38

8 Zaključak **40**

9 Literatura **41**

1 Uvod

Cilj ovog završnog rada je proći kroz osnove Ruby programskog jezika. Prvo ćemo proći kroz osnovne karakteristike programskog jezika Ruby, uključujući leksičku i sintaktičku strukturu. U sljedećem poglavlju ćemo proći kroz tipove podataka u Rubyju, njihove literale te najčešće korištene metode na njima. U poglavlju poslije tog, proći ćemo kroz različite vrste izraza u Rubyju, a u poglavlju poslije tog kroz kontrolne strukture Rubyja. U pretposljednjem poglavlju obradit ćemo metode i objekte koji reprezentiraju blokove. U posljednjem poglavlju ćemo se posvetiti objektno orijentiranom programiranju u Rubyju. Pri tome će se obratiti pozornost na razlike između Rubyja i Pythona te Rubyjeve nedostatke i prednosti općenito. Međutim, iako je Ruby dizajniran s jednostavnošću kao ciljem, on je vrlo moćan jezik s vrlo širokim rasponom mogućnosti. Shodno tome, ovaj rad neće ulaziti u neke od kompliciranijih tema u Rubyju poput paralelnog programiranja i metaprogramiranja. Kako se ovaj rad bavi programskim jezikom, neophodno je pojavljivanje nekog programskog koda u njemu. Svaki komad koda izgledat će ovako:

Listing 1: kod 1

```
1 a=[1, 2, 8]
2 a.each do |elem|
3   print elem
4 end
```

Ponekad će se kod napisan u Rubyju uspoređivati s ekvivalentnim kodom napisanim u Pythonu. Tada će kodovi izgledati ovako:

Listing 2: kod1

```
1 a=[1, 2, 8]
2 a.each do |elem|
3   print elem
4 end
```

Listing 3: kod1 Python

```
1 a=[1, 2, 8]
2 for elem in a:
3   print(elem)
```

1.1 Programski Jezik Ruby

Ruby je dinamičan programski jezik koji je u potpunosti objektno orijentiran. Tvorac Rubyja je Yukihiro "Matz" Matsumoto, čiji je cilj bio napraviti jezik "moćniji od Perla i više objektno orijentiran od Pythona." Prva verzija Rubyja (0.95) izdana je u prosincu 1995. godine, a verzija 1.0 u prosincu 1996. godine. Posljednja stabilna verzija 2.3.1 izdana je u travnju 2016. godine. Ovaj završni rad pisan je za verziju 1.9.1 Rubyja i uspoređivat će se s verzijom 3.4 Pythona.

1.2 Objektno Orijentirano Programiranje

Objektno orijentirano programiranje(skraćeno: OOP) je paradigma programiranja u kojoj je fokus stavljen na objekte - strukture koje sadržavaju podatke, u obliku atributa, te kodove, u obliku metoda. U OOP modeliramo stvarni život koristeći programske klase i objekte koji su određeni svojim stanjem i ponašanjem. Bitni pojam u OOP je klasa - nacrt objekta,

odnosno predložak po kojem je određeni objekt stvoren. Neke od karakteristika OOP su enkapsulacija i nasljeđivanje. Ruby i Python su oboje primjeri objektno orijentiranih jezika.

2 Osnovne Karakteristike

Ruby je dinamičan programski jezik jake tipizacije. To znači da se tipovi daju podacima za vrijeme izvršavanja, dok su promjene tipa moguće, ali se ne događaju implicitno. Sljedeći primjer to ilustrira:

Listing 4: kod2

```
1 def foo(a)
2   if a==1
3     puts "Sve je u redu!"
4   else
5     puts "string"+a
6   end
7 end
```

U gornjem primjeru će se

```
foo(1)
```

izvršiti bez problema, ali npr.

```
foo(2)
```

će izbaciti TypeError. Primjer nam je pokazao da je Ruby jezik jake tipizacije jer ne pretvara implicitno Fixnum u String, dok dinamičnost vidimo iz toga što se TypeError ne pojavljuje ako interpreter ne dođe do reda 5 u kodu. Python funkcionira na isti način.
Ruby je skriptni jezik. To znači da se izjave izvršavaju sekvencijalno, iako kontrolne strukture mogu utjecati na redoslijed izvršavanja. Za razliku od jezika poput C++, Ruby ne sadrži *main* metodu već se počinje izvršavati od prve linije koda (s iznimkom BEGIN izjave o kojoj će više biti riječi u poglavlju 5). Ruby je ekspresivan jezik i drži se principa najmanjeg iznenađenja, tj. dizajniran je za funkcioniranje na način na koji bi prosječan programer to i očekivao. Misao vodilja Matza u dizajniranju Rubyja je: "Ruby je dizajniran kako bi učinio programere sretnima." Tipičan primjer toga može se vidjeti u načinu na koji bismo pomoću Rubyja dva puta ispisali neku rečenicu:

Listing 5: kod3

```
1 2.times{puts "Hello! "}
```

Listing 6: kod3 Python

```
1 for i in range(2):
2   print("Hello! ")
```

Iako je i Python vrlo ekspresivan jezik, čak bi i potpuni programerski laici mogli razumjeti što će kod3 ispisati prilikom pokretanja.
Kao što je već napomenuto, Ruby je u potpunosti objektno orijentiran. Gotovo sve u Rubyju je nekakav objekt, iako postoje iznimke poput lista argumenata. Tako smo u prošlom primjeru mogli pozvati metodu times na **objekt** 2. Objektno orijentiranom programiranju u Rubyju ćemo se više posvetiti u poglavlju 7.
Ruby ima još jedno zanimljivo svojstvo. Naime, sve izjave u Rubyju su izrazi, tj. vraćaju neku vrijednost, makar to bilo *nil*. Čak su i neke kontrolne strukture poput *if* izrazi u Rubyju pa možemo imati kod poput ovoga:

Listing 7: kod4

```
1 y = if x>0 then 1 else 2 end
```

Više o tome ćemo reći u poglavljima 4 i 5.
Često navođena glavna razlika između Rubyja i Pythona je filozofska: Ruby se vodi filozofijom postojanja više načina za rješavanje nekog problema, dok se Python vodi filozofijom postojanja jedinstvenog načina rješavanja nekog problema.
Neke od karakteristika Rubyja koje bi mogle biti neobične onima koji su naviknuli programirati u drugim jezicima su mutabilnost stringova te činjenica da petlje i uvjetne strukture u Rubyju evaluiraju sve osim *false* i *nil* kao *true*, uključujući i nulu.

2.1 Leksička Struktura

Kod u Rubyju sastoji se od niza tokena, koji mogu biti komentari, literali, punktuacija, identifikatori ili ključne riječi, te od praznog prostora (tzv. *whitespace*).
Komentari počinju znakom # i sav tekst poslije njega do kraja reda se ignorira. Ako želimo napisati komentar koji se proteže kroz više redova, možemo to učiniti na dva načina. Prvi način je jednostavno na početku svakog reda staviti #. Drugi način je započeti red s =begin, ispisati komentar i na početku sljedećeg reda napisati =end.

Listing 8: komentar

```
1 # Jedan
2 # Nacin
3 =begin drugi
4    nacin
5 =end
```

Literali su vrijednosti koji su pojavljuju direktno u Rubyjevom izvornom kodu. Oni mogu biti brojevi, stringovi i regexpi, dok su literali polja i slično kompliciraniji izrazi koji nisu individualni tokeni. Primjeri:

Listing 9: literali

```
1 3
2 2.56
3 "rijec"
4 /regexp/
```

Punktuacija služi za delimitiranje stringova, regexpa itd., za pozivanje operatora te još niz raznih funkcija.
Identifikatori su imena varijabli, metoda, klasa itd. Mogu se sastojati od slova, brojeva i _ znaka, ali ne mogu počinjati brojem. Ako počinju velikim slovom, Ruby ih smatra konstantom i upozorava (ali ne zabranjuje) njihovo mijenjanje. Osim toga, identifikatori mogu započinjati ili završavati određenim punktuacijskim znakovima, svaki od kojih daje određeno značenje: stavljanjem znaka $ na početku identifikatora označavamo globalnu varijablu, stavljanjem @ varijablu instance, a stavljanjem @@ varijablu klase. Prema konvencijama, na kraju indentifikatora metoda koje vraćaju Boolean vrijednost stavljamo ?, a ! ako bi ih se trebalo oprezno koristiti. Postoji još i znak = koji stavljamo kao oznaku da metoda postavlja vrijednost neke varijable.
Ključne riječi su tokeni posebnog značaja u Rubyju. Njih ne možemo koristiti kao identifikatore, osim ako ne stavimo $, @ ili @@ kao prefiks. S druge strane, mogu se koristiti kao imena

metoda, ali se to ne preporuča jer rezultira potencijalno zbunjujućim kodom. Osim ključnih riječi, postoje neke metode koje su implementirane na *Kernel, Module, Class, Object* pa se u praksi ni njihova imena ne koriste. Ekstenzivna lista ključnih riječi i prethodno navedenih metoda može se naći u [2].

2.2 Sintaktička Struktura

Osnovna jedinica sintaktičke strukture u Rubyju je izraz. Primarni izrazi su oni koji direktno reprezentiraju vrijednosti poput literala stringova ili brojeva te referenci na varijable. Kompliciranije izraze i izjave dobivamo kombiniranjem izraza, operatora i izjava. Više jedinice sintaktičke strukture u Rubyju su metode, klase, moduli te blokovi. O njima ćemo reći više u nadolazećim poglavljima.

3 Tipovi Podataka

U ovom ćemo poglavlju proći kroz različite tipove podataka u Rubyju, od brojeva do hasheva.

3.1 Booleani

Za razliku od Ptyhona, u Rubyju ne postoji Boolean klasa, već tri posebne klase: TrueClass, FalseClass i NilClass čije su singleton instance *true, false i nil* vrijednosti. Kao što je već prethodno napomenuto, u Rubyju *true i false* nisu isto što i 1 i 0 pa se tako, u slučaju da Ruby zahtjeva Boolean vrijednost, i 1 i 0 ponašaju kao *true*, što je u suprotnosti s ponašanjem 1 i 0 u Pythonu.

3.2 Brojevi

U Rubyju je skoro pa sve objekt pa tako i brojevi. Točnije, brojevi u Rubyju pripadaju klasi Numeric koju nasljeđuju klase Integer, koja sadrži cijele brojeve, te klasa Float, koja sadrži brojeve s pomičnom točkom. Klasu Integer također nasljeđuju dvije klase: Fixnum, koja sadrži vrijednosti unutar 31 bita, te Bignum za ostale. Standard library Rubyja sadrži još klase Complex, BigDecimal i Rational za redom kompleksne brojeve, realne brojeve arbitrarne preciznosti i racionalne brojeve. Svi objekti koji pripadaju tim klasama su imutabilni. Na svim tim klasama su definirane standardne operacije +,-,* i /. Ako rezultat operacija na objektima klase Fixnum proizvede prevelik rezultat, Ruby automatski pretvara rezultat u Bignum. Dijeljenje funkcionira kao i u Ptyhonu, tj. ako su oba operanda cijeli brojevi, onda će i rezultat biti cjelobrojan.
U Pythonu pak brojevi nisu objekti. U Rubyju, za razliku od mnogih drugih programskih jezika, floating-point brojeve manje od 1 ne možemo zapisati bez početne nule, tj. naredba

Listing 10: kod5

```
1 x = .3
```

nije validna u Rubyju, ali je validna u Ptyhonu.

3.3 Tekst

U Rubyju postoje dvije bitne klase vezane uz tekst. Jedna je klasa String koja služi za reprezentiranje teksta, a druga je Regexp koja služi za reprezentiranje tekstualnih uzoraka. Ruby sadrži nekoliko načina za definiranje String objekata. Ovdje ćemo proći kroz neke od njih.
Vjerojatno najkorištenije metode za definiranje String objekata su korištenje jednostrukih ili dvostrukih navodnika. Tekst unutar navodnika tada predstavlja vrijednost stringa.

Listing 11: kod6

```
1 x = 'Ovo je primjer stringa unutar jednostrukih navodnika.'
2 y = "Ovo je primjer stringa unutar dvostrukih navodnika."
```

Problemi se mogu pojaviti kada želimo unutar stringa ubaciti navodnike bez da se string prekine. U Rubyju to radimo tako da stavimo backslash znak prije navodnika.

Listing 12: kod7

```
1 x = 'Pr\'mjer.'
2 y = "Pr\"mjer."
3 puts x,y
4 # Pr'mjer.
5 # Pr"mjer.
6 # => nil
```

Ako pak želimo prikazati backslash, onda koristimo dvostruki backslash:

Listing 13: kod8

```
1 x = 'Pr\\mjer.'
2 puts x
3 # Pr\mjer
4 # => nil
```

Međutim, kada koristimo jednostruke navodnike, udvostručavanje backslasha nije nužno. Naime, unutar jednostrukih navodnika backslash će imati posebnu funkciju isključivo ako se nalazi prije još jednog backslasha ili jednostrukog navodnika. S druge strane, unutar dvostrukih navodnika postoji puno veći broj posebnih nizova znakova, što s backslashom, što bez njega. Zbog toga u ovom radu nećemo proći ekstenzivno kroz njih, već samo predstaviti nekoliko primjera:

Listing 14: kod9

```
1 "\tTab nakon kojeg je novi red\npa jedan backslash\\"
2 x="\u0053" # \u predstavlja Unicode
3 puts x
4 # S
5 # => nil
```

Još jedna prednost korištenja dvostrukih navodnika za definiranje stringova je to što Ruby dozvoljava interpolaciju stringova: zamjenjivanje arbitrarnog izraza njegovom vrijednosti. To radimo tako da izraz stavimo unutar vitičastih zagrada ispred kojih stavimo ljestve.

Listing 15: kod10

```
1 "2+2=#{2+2}"
```

Ako pak želimo samo referencirati neku globalnu varijablu ili varijablu instance ili klase, onda možemo maknuti vitičaste zagrade.

Listing 16: kod11

```
1 $globalnavar = 'primjer'
2 "#$globalnavar"                    # "primjer"
```

Ruby također podržava arbitrarne delimitere stringova. Ako string započnemo s %q, odnosno %Q, tada znak koji stavimo poslije q, odnosno Q, postaje delimiter stringa, a string se ponaša kao da je delimitiran jednostrukim, odnosno dvostrukim navodnicima. U slučaju da koristimo %Q možemo izostaviti Q.

Listing 17: kod12

```
1 %q-Ovo je validan string-
```

Ako nam ne odgovara niti jedan delimiter, Ruby ima još jedan način za definiranje stringova, a to je korištenje here dokumenata. Here dokumenti počinju s << ili <<- nakon čega slijedi identifikator ili string koji definira završni delimiter. Tekst počinje u sljedećem redu i završava kada se delimiter pojavi sam u nekom redu.

Listing 18: kod13

```
1 document = <<KRAJ # Napomena: Jedino ovdje mozemo staviti komentar
2 Lorem ipsum dolor sit amet
3 ' "" \ bla # bla
4 KRAJ
```

Postoji još jedan način interpolacije stringova u Rubyju koji postoji i u Pythonu:

Listing 19: kod14

```
1 "2+2 je %d" % (2+2)
```

Postoje nekoliko korisnih operatora definiranih na String klasi. Neki od njih, poput +, ==, <, funkcioniraju isto kao i u Pythonu. S druge strane, konkatenacija stringova je puno jednostavnija i intuitivnija u Rubyju nego u Pythonu.

Listing 20: kod15

```
1 x="jedan"
2 y="dva"
3 x<<y     # x postaje "jedandva"
```

Listing 21: kod15 Python

```
1 x="jedan"
2 y="dva"
3 x = "".join([x,y])
```

Vjerojatno najveća razlika među stringovima u Rubyju i Pythonu je to što su u Rubyju stringovi mutabilni objekti. To znači da Ruby interpreter ne može koristiti jedan objekt za reprezentiranje identičnih string literala. Također, to u Rubyju omogućuje ovakav kod:

Listing 22: kod16

```
1 x='abcd'
2 x[1]='f'      # x postaje afcd
```

U Pythonu nam takav kod ne bi prošao.
Međutim, ako nam je potrebna klasa objekata koji sadrže niz znakova i koji su imutabilni, Ruby nam i tu može pomoći sa Symbol klasom. Simboli se definiraju dvotočkom nakon koje slijedi string koji, ako ne sadrži razmake, ne mora biti omeđen navodnicima. Ako pak je, možemo, kao i kod stringova, koristiti arbitrarne delimetere ako umjesto dvotočke stavimo %s. Tako su

Listing 23: kod17

```
1 :primjer
2 :'primjer'
3 :'pri mjer'
4 %s-primjer-
```

sve validni primjeri simbola.
Za razliku od stringova koji mogu imati isti sadržaj, ali biti dva različita objekta, kod simbola je tako nešto nemoguće. Operacije uspoređivanja simbola su brže nego iste operacije na

stringovima pa se preporuča korištenje simbola kada ne koristimo stringove zbog njihovog sadržaja već kao identifikatore. Ekvivalentna klasa ne postoji u Pythonu, ali nije ni potrebna jer se stringovi u Pythonu ponašaju kao simboli u Rubyju.

3.4 Polja

Polja su niz vrijednosti kojima pristupamo po njihovoj poziciji u nizu. Prva vrijednost u polju ima indeks nula. Literal polja je niz vrijednosti razdvojen zarezima i omeđen uglatim zagradama. Posebno, za izražavanje literala polja čiji su elementi stringovi bez zareza imamo posebnu sintaksu:

Listing 24: kod18

```
1 %w|( [ \t bla bla|                    # ["(","[","\t","bla","bla"]
```

Polja su mutabilna i ne moraju svi elementi biti istog tipa. Štoviše, element nekog polja može biti drugo polje. Nadalje, polja se mogu dinamički povećavati dodavanjem elementa izvan granica polja pri čemu se polje nadopuni *nil* elementima:

Listing 25: kod19

```
1 a=[1,[4,5],3]   # polje s 3 elementa
2 a[4]            # nil (Primjeti: Python bi ovdje izbacio IndexError)
3 a[4]=10         # a=[1,[4,5],3,nil,10]
```

Klasa Array također sadrži kontruktor *new* koji omogućuje još načina inicijaliziranja elemenata:

Listing 26: kod20

```
1 x = Array.new                    # []
2 y = Array.new(5)                 # [nil, nil, nil, nil, nil]
3 z = Array.new(2,x)               # [[],[]]
4 arr = Array.new(5) {|i| 2*i-1}   # [-1,1,3,5,7]
```

Prođimo sada kroz neke od metoda klase Array. Metoda *each* prolazi kroz elemente polja te za svaki izvrši određen kod. Identičan efekt se u Pythonu postiže korištenjem *for* petlje:

Listing 27: kod21

```
1 a=['a','b','c']
2 a.each{|x| puts x}
```

Listing 28: kod21 Python

```
1 a=['a','b','c']
2 for x in a:
3   print(x)
```

Jedna od Pythonovih prednosti nad Rubyjem je veći broj struktura podataka. U Pythonu tako osim lista, koje funkcioniraju gotovo identično kao polja u Rubyju, postoje još i *tuple* objekti (n-torke) koji su imutabilni te *deque* objekti koji su generalizacija stackova i queova. (Napomena: deque se nalazi u Collections modulu).

3.5 Hashevi

Hash je struktura podataka koja sadržava skup ključeva i svakom od njih pridružuje neku vrijednost. Drugi termini za hasheve su mape te asocijativna polja.
Hash literali u Rubyju se pišu vrlo slično diciontaryjima u Pythonu, s time da umjesto : pišemo =>

Listing 29: kod22

```
1 gradovi = {"Hrvatska"=>"Zagreb","Berlin"=>"Njemacka"}
2 gradovi["Hrvatska"]                    # "Zagreb"
```

Korištenje mutabilnih objekta kao ključeve u hashevima je problematično te se strogo preporuča da se takvo ponašanje izbjegava (Ruby, doduše, neće eksplicitno to zabraniti kao što će to Python napraviti s ključevima dictionaryja). Kako se stringovi često koriste kao ključevi, a mutabilni su, Ruby radi privatne kopije svih stringova koji se koriste kao ključevi. Međutim, svejedno je efikasnije umjesto njih kao ključeve koristiti simbole. U tom slučaju dvotočku možemo pomaknuti s početka na kraj simbola pa sintaksa postaje još sličnija onoj u Pythonu:

Listing 30: kod23

```
1 gradovi = {Hrvatska: "Zagreb", Berlin: "Njemacka"}
2 gradovi[:Hrvatska]                     # "Zagreb"
```

3.6 Rasponi

Rasponi u Rubyju su objekti klase Range. Literali raspona pišu se stavljanjem dvaju točaka, ako želimo uključiti krajnju vrijednost, ili tri točke između početne i krajnje vrijednosti. Za granične vrijednosti možemo koristiti samo one vrijednosti za koje je implementiran operator <=> koji uspoređuje dvije vrijednosti i vraća -1,0 ili 1. Postoje dvije vrste raspona: diskretni i neprekidni. Diskretni rasponi su oni čije granične vrijednosti imaju definiranu metodu *succ*, a neprekidni oni koji nemaju. Diskretne raspone možemo koristit kao iteratore pomoću metoda *each* i *step* te ih pretvoriti u polja pomoću metode *to_a*.

Listing 31: kod24

```
1 suma = 0
2 diskr = 1..5                           # diskretan raspon
3 diskr.each{|x| suma += x}              # suma postaje 15
4 diskr.step(2){|x| suma += x}           # suma postaje 24
5 diskr.to_a                             # [1,2,3,4,5]
6 nepr = 1.0...5.0                       # neprekidan raspon
7 nepr.each{|x| suma+= x}                # TypeError
```

U gornjem primjeru je došlo do TypeError jer klasa Float nema definiranu metodu *succ*. Isti bi ishod bio da smo *nepr* pokušali pretvoriti u polje metodom *to_a*.
U Pythonu rasponi mogu imati isključivo cijele brojeve za granične točke i kao *step* parametar te nikada ne uključuju krajnju točku. Međutim, proširenje range funkcije u Pythonu za

floating point brojeve je jednostavno (Jedan primjer može se pronaći na [7]).
Primarna uloga raspona je provjeravanje pripadnosti neke vrijednosti tom rasponu. Postoje dvije različite definicije za pripadnost rasponu: diskretna i neprekidna. Neprekidna definicija ima smisla za svaki raspon jer koristi metodu <=> koja je uvijek definirana za granične vrijednosti raspona. Prema toj definiciji neka vrijednost x pripada rasponu a..b, odnosno a...b ako vrijedi:

$$a <= x <= b, \text{ odnosno } a <= x < b$$

Diskretna definicija pripadnosti rasponu ovisi o *succ* metodi pa ima smisla isključivo na diskretnim rasponima. Prema njoj vrijednost x pripada rasponu a..b ako skup koji se sastoji od *a,a.succ,a.succ.succ,...* sadrži u sebi x. Ruby tako ima nekoliko metoda za provjeru pripadnosti rasponu. Metode *include? i member?* metode koriste neprekidnu definiciju na rasponima čije su granične točke brojevi, a diskretnu inače, dok metoda *cover?* uvijek koristi neprekidnu definiciju pripadnosti rasponu.

4 Izrazi

Izrazi su komadi Ruby koda za koje interpreter vraća neku vrijednost. Puno programskih jezika, poput Pythona, radi distinkciju između izjava i izraza poput petlji. U tim jezicima izjave kontroliraju tok programa, ali ne vraćaju neku vrijednost. U Rubyju ne postoji jasna distinkcija između izraza i izjava. Unatoč tome, izraze koji koriste sintaksu uobičajenu za izjave smatrat ćemo izjavama i njima ćemo se posvetiti u poglavlju 5.
Najjednostavniji izrazi su literali te određene ključne riječi poput *nil, true, false i self*. Među jednostavne izraze ubrajamo i reference varijabli te konstanti koje vraćaju vrijednost spremljenu u varijabli. Konstante su varijable za koje ne želimo da mijenjaju svoju vrijednost (iako Ruby to neće zabraniti). Imena konstanti izgledaju kao imena lokalnih varijabli s time da imaju veliko početno slovo. Konstante, neovisno o tome gdje su definirane, imaju vidljivost globalnih varijabli. Ako varijabla nije instancirana i pokušamo je referencirati, doći će do greške osim u sljedećim slučajevima:
Varijabla je globalna, varijabla instance ili je lokalna varijabla za koju se pojavio izraz pridruživanja, ali do samog pridruživanja nije došlo, tj. ako imamo ovakvu situaciju:

Listing 32: kod25

```
1 x = "primjer" if false
```

U sva tri slučaja će referenciranje tih varijabli vratiti *nil*. U Pythonu takve iznimke ne postoje. Pokušaj referenciranja varijable koja nije inicijalizirana uvijek izbacuje grešku.
Pozivi metoda su također izrazi i sastoje se od četiri dijela: Izraza čija je vrijednost objekt na kojem pozivamo metodu odvojen od imena metoda s . ili s :: nakon čega slijede zagrade unutar kojih se nalaze argumenti koje prosljeđujemo metodi. Konačno, možemo imati i blok koda delimitiran vitičastim zagradama ili s *do/end*. Ako izostavimo izraz na kojem pozivamo metodu, metoda se poziva na *self*. Ako metodi ne prosljeđujemo niti jedan argument, zagrade nam nisu potrebne.
Izrazi pridruživanja su izrazi koji specificiraju vrijednosti jednog ili više lvaluea (lvalue je termin za sve što se pojavljuje s lijeve strane operatora pridruživanja, a rvalue za sve što se pojavulje s desne). U Rubyju postoje tri vrste izraza pridruživanja: jednostavna, skraćena i paralelna.
Jednostavna pridruživanja se sastoje od jednog lvaluea, operatora = i jednog rvaluea. Skraćena pridruživanja mijenjaju vrijednost varijable djelovanjem nekim operatorom na nju. Paralelna pridruživanja su ona koji imaju više lvaluea ili rvaluea. Primjeri:

Listing 33: kod26

```
1 x = 1          # jednostavno pridruzivanje
2 x += 1         # skraceno pridruzivanje
3 x,y=1,2        # paralelno pridruzivanje
```

Kod paralelnog pridruživanja može ponekad doći do situacija u kojim nije u potpunosti jasno što će se dogoditi. Tu ćemo proći kroz te slučajeve.
Kada imamo samo jedan lvalue, a više rvaluea, onda Ruby stavlja rvalue u polje i pridružuje to polje lvalueu. Ako imamo više lvaluea i jedno polje (ili nešto što ima definiranu metodu to_ary) kao rvalue, tada Ruby prvom lvalueu pridružuje prvi element polja, drugom drugi itd. Ako imamo n više lvaluea od rvaluea, tada se posljednjim n lvalueima pridružuje *nil*, a

u obrnutom slučaju se posljednjih n rvaluea ne pridružuju nikome. Posebnu situaciju dobivamo kada jedan od lvaluea ili rvaluea ima ispred sebe *. Ako se * nalazi ispred rvaluea, to znači da se taj rvalue smatra poljem (ili objektom sličnim polju) te bi se u nizu rvaluea taj rvalue trebao zamijeniti svojim elementima. Ako je pak * ispred lvaluea, to znači da tom lvalueu pridružujemo polje koje sadrži sve viškove rvaluea. Takvih rvaluea možemo imati koliko hoćemo, a lvaluea samo jedan, ali na bilo kojoj poziciji. Posljednja posebna situacija je kada imamo dva ili više lvaluea unutar zagrada. To se smatra potpridruživanjem. Prvo se cijela zagrada smatra jednim lvalueom te nakon što joj se pridruži vrijednost se ta vrijednost pridružuje grupi lvaluea u zagradi. Ta vrijednost mora biti objekt ispred kojeg bi se mogla staviti *. Pogledajmo primjere:

Listing 34: kod27

```
1 x = 1,2,3                          # x=[1,2,3]
2 x,y=[1,2]                          # x,y=1,2
3 x,y,z=1,2                          # x=1;y=2;z=nil
4 x,y=1,2,3                          # x=1;y=2
5 *x,y=1,*[2,3],4                    # x=[1,2,3],y=4
6 x,(y,z),w=1,[2,3],4                # x=1;y,z=2;w=3
```

Posljednja vrsta izraza su operatori. Operatori su tokeni u Rubyju koji predstavljaju operacije na jednom ili više operanada. Pomoću njih možemo spajati izraze u veće izraze. Npr.

Listing 35: kod28

```
1 3+foo(10)>bar    # spajamo literale, poziv metode i operator >
```

Operatore dijelimo prema broju operanada na kojima djeluju, redoslijedu izvršavanja u odnosu na druge operatore te asocijativnosti. Komprehenzivan pregled operatora u Rubyju može se naći u [2].

5 Kontrolne Strukture

U većini slučajeva izrade koda ne želimo bezuvjetno izvršavanje svih linija koda te izvršavanje svake linije koda samo jednom i to redom kojim se pojavljuju u kodu. Zbog toga koristimo različite kontrolne strukture i izjave koje mijenjaju tok izvršavanja. Ruby ima bogat raspon struktura i izjava koje možemo koristiti za mijenjanje toka izvršavanja programa i kroz njih ćemo proći u ovom poglavlju.

5.1 Uvjetne Strukture

Uvjetne strukture su jedne od najčešćih kontrolnih struktura u programiranju. Funkcioniraju na način da određen dio koda izvrše isključivo ako je neki izraz istinit. U Rubyju se uvjetne strukture mogu koristiti koristeći više različitih sintaksi. Najjednostavniji od njih je *if*:

Listing 36: kod29

```
1 if izraz then                    # then nije nuzan
2    kod
3 end
4 # ekvivalentno:
5 if izraz then kod end
6 # ekvivalentno:
7 kod if izraz
```

U tom slučaju se *kod* izvrši isključivo ako *izraz* nije *nil* ili *false*. U većini programskih jezika, a između ostalih i Pythonu, je *if* izjava, no u Rubyju su sve izjave izrazi pa čak i *if*. Vrijednost koju *if* vraća je ili vrijednost zadnjeg izraza u kodu koji se izvršio ili *nil* ako se nijedan izraz nije izvršio. To omogućuje ovakav kod:

Listing 37: kod30

```
1 predznak = if x<0 then "-" else "+" end
```

Uz *if* možemo koristiti i *else* koji izvršava kod ako uvjet nije istinit:

Listing 38: kod31

```
1 if izraz
2   kod1
3 else
4   kod2
5 end
6 # ekvivalentni izrazi:
7 if izraz then kod1 else kod2 end
8 izraz ? kod1 : kod2
```

Ako želimo testirati više uvjeta možemo, umjesto korištenja *else if*, koristiti *elsif*:

Listing 39: kod32

```
if izraz1
   kod1
elsif izraz2
   kod2
else
   kod3
end
# ekvivalentno:
if izraz1 then kod1 elsif izraz2 then kod2 else kod3 end
```

Osim *if* postoje još dvije uvjetne strukture: *unless* i *case*. *unless* koristi istu sintaksu kao i *if* (bez dopuštanja *elsif*), ali ima obrnutu funkciju od *if*, tj. izvršava kod isključivo ako se uvjet evaluira kao *nil* ili *false*.

Listing 40: kod33

```
unless izraz
   kod
end
# ekvivalentno:
kod unless izraz
```

Posljednja uvjetna struktura u Rubyju je *case*. *case* možemo koristiti na dva načina u Rubyju. Prvi od njih je kao zamijena za *if*:

Listing 41: kod34

```
case                                    # ekvivalentno:
when izraz1                             # if izraz1
   kod1                                 #    kod1
when izraz2                             # elsif izraz2
   kod2                                 #    kod2
else                                    # else
   kod3                                 #    kod3
end                                     # end
```

Međutim, drugi način je puno moćniji od prvoga. Izrazi u kontrolnim petljama najčešće imaju jednake lijeve strane kao npr. u:

Listing 42: kod35

```
case
when x==1
   puts "jednina"
when x==2,x==3,x==4                  # zarezi su ekvivalentni ||
   puts "dvojina
else
   puts "mnozina"
end
```

Tada možemo koristiti ovakvu sintaksu:

Listing 43: kod36

```
puts case x
     when 1
        "jednina"
     when 2,3,4 then "dvojina"
     else "mnozina"
end
# ili skraceno:
puts case x when 1 "jednina" when 2,3,4 "dvojina" else "mnozina" end
```

Kada koristimo *case* na ovaj način, *case* uspoređuje izraz uz *when* s izrazom uz *case* koristeći operator ===. Operator === djeluje kao == na većini klasa. Međutim klasa Class definira === tako da vraća *true* samo ako je desni operand instanca klase imenovana lijevim operandom. Naprimjer:

Listing 44: kod37

```
puts case x
     when Fixnum then "cijeli broj"
     when Float then "realni broj"
end
```

5.2 Petlje i Iteratori

Ruby sadrži tri petlje: *while,until i for*, ali i mogućnost definiranja iteratora, tj. kustomiziranih petlji i one se koriste češće od Rubyjevih built-in petlji.
while i until su vrlo jednostavne petlje koje se izvršavaju dok god je uvjet istinit, odnosno lažan:

Listing 45: kod38

```
x=0
while x<2 do
        x += 1
        puts x
end
# program ispisuje 1,2
# ekvivalentno:
y=0
until y>=2 do
        y += 1
        puts y
end
```

Poput *if i unless*, *while i until* imaju skraćeni oblik:

Listing 46: kod39

```
1 x=0
2 puts x += 1 while x<2
```

Valja napomenuti da ako ovu sintaksu koristimo na izraz koji je omeđen s *begin i end*, petlja se uvijek izvršava barem jednom. Primjer:

Listing 47: kod40

```
1 x=0
2 begin
3     puts x
4 end while x>0              # ispisat ce se jedna nula
```

Posljednja built-in petlja u Rubyju je *for* petlja. Ona iterira kroz elemente prebrojivih objekata (poput polja), pridružuje vrijednost tog elementa varijabli petlje te onda izvrši kod unutar tijela petlje. Primjer:

Listing 48: kod41

```
1 for i in 1..4 do                          # do nije nuzan
2     puts i
3 end
4 # kod ce ispisati brojeve 1,2,3 i 4
```

Objekt se smatra prebrojivim ako je na njemu definirana metoda *each*. Tako umjesto koristeći *for* petlju, gornji primjer možemo napisati eksplicitno koristeći *each* iterator:

Listing 49: kod42

```
1 (1..4).each do |i|
2     puts i
3 end
```

Međutim, korištenje built-in petlji i iteratora nije u potpunosti ekvivalentno jer blok koda koji slijedi iterator definira novi prostor u kojem su definirane varijable.
Iterator u Rubyju je termin za sve metode koje koriste izjavu *yield*. To znači da iteratori ne moraju nužno iterirati ili služiti kao petlja. Uz iteratore se gotovo uvijek pojavljuju blokovi. Izjavom *yield* i blokovima ćemo se baviti u potpoglavlju 5.3
Na klasi Integer imamo definirana tri iteratora: *upto, downto i times*. Njihovo djelovanje je intuitivno i lako za razumjeti iz primjera:

Listing 50: kod43

```
1 3.upto(6){|x| puts x}          # ispisuje 3,4,5,6
2 6.downto(3){|x| puts x}        # ispisuje 6,5,4,3
3 4.times{|x| puts x}            # ispisuje 0,1,2,3
```

Iteriranje kroz floating-point brojeve radimo pomoću *step* metode koja se poziva na početnom broju i prima dva argumenta: prvi je krajnji broj, a drugi broj za koji se pomičemo u svakoj iteraciji pa tako na kraju sljedećeg koda:

Listing 51: kod44

```
1 suma = 0
2 0.step(2.5,0.5){|x| suma+=x}
```

će nam *suma* biti jednaka 7.5

Na puno različitih klasa poput Array i Range je definiran iterator *each* koji prosljeđuje elemente objekta nad kojim je pozvan asociranom bloku. Većina klasa s definiranom *each* metodom uključuju Enumerable modul u kojem se nalazi nekoliko specijaliziranih iteratora. Najčešće korišteni iteratori modula Enumerable su *collect, select, reject i inject.*

Metoda *collect* izvršava asocirani blok za svaki element te vraća polje čiji su elementi vraćene vrijednosti iz blokova. Metode *select i reject* rade dijametralno suprotne stvari: *select* vraća polje koje sadrži one elemente za koje asocirani blok vraća vrijednost različitu od *false i nil*, a *reject* polje s onim elementima za koje asocirani blok vraća *false* ili *nil.* Metoda *inject* je malo kompliciranija. Poziva asocirani blok s dva parametra od kojih je prvi nekakva suma vrijednosti iz prijašnjih iteracija, a drugi element objekta na kojem pozivamo *inject.* Vrijednost koju vraća blok postaje ili prvi parametar sljedeće iteracije ili vrijednost koju vraća iterator ako je u pitanju posljednja iteracija. Prvi parametar u prvoj iteraciji poprima ili vrijednost proslijeđenog argumenta, ako postoji, ili vrijednost prvog elementa objekta na kojem je metoda pozvana. Primjeri:

Listing 52: kod45

```
1 a=[1,2,3,4,5]
2 a.collect{|x| 2*x}                  # => [2,4,6,8,10]
3 a.select{|x| x<3}                   # => [1,2]
4 a.reject{|x| x<3}                   # => [3,4,5]
5 a.inject(10){|suma,x| suma+x}       # => 25 (=10+1+2+3+4+5)
```

Iako su gore navedeni iteratori vrlo korisni, ovisno o svojim potrebama možemo definirati svoje iteratore. Definirajuće svojstvo iteratora je pozivanje asociranog bloka što postižemo korištnjem metode *yield.* Ako bloku želimo proslijediti neke argumente, tada ih napišemo poslije *yield* odvojene zarezima. Pogledajmo jednostavan primjer iteratora:

Listing 53: kod46

```
1  # Metoda ocekuje blok i generira prvih n clanova niza a*b^i
2  # te ih yield metodom prosljeduje bloku
3  def gniz(a,b,n)
4      i = 0
5      while(i<n)
6          yield a*b**i
7          i+=1
8      end
9  end
10 suma = 0
11 gniz(1,2,3){|x| suma+=x}           # suma=7 (=1+2+4)
```

5.3 Blokovi

U prošlom potpoglavlju smo intenzivno koristili blokove kao dijelove iteratora. U ovom ćemo se potpoglavlju posvetiti samim blokovima.
Blokovi ne mogu biti samostalni, već se uvijek pojavljuju nakon poziva metode. Ako se u metodi nikad ne pozove blok pomoću *yield*, onda se blok ignorira. Blokove delimitiramo ili vitičastim zagradama ili s *do i end* ključnim riječima s time da se *do* mora nalaziti u istom redu u kojem je poziv metode. Blokovi se mogu parametrizirati i lista parametara je delimitirana okomitim crtama i odvojena zarezima.
Blokovi definiraju i novi prostor varijabli. Varijable stvorene unutar blokova postoje isključivo unutar tog bloka. Međutim, bloku su dostupne lokalne varijable metode pa treba pripaziti pri odabiru imena varijabli unutar bloka. Ako parametre bloka odvojimo točkom sa zarezom nakon kojih stavimo zarezima odvojen niz varijabli, onda će one sigurno biti blok-lokalne čak i ako postoji neka druga varijabla s istim imenom u istom prostoru varijabli. Parametri bloka su uvijek blok-lokalni. Primjer:

Listing 54: kod47

```
1 x = y = 100
2 3.times do |x;y|          # x je parametar, a y lokalna varijabla
3    y=x*x
4    puts y                          # ispisuju se 0,1,4
5 end
```

5.4 Izjave

Uz dosad navedene metode za kontroliranje programskog toka, Ruby podržava i niz izjava koje to rade: *return* koji izlazi iz metode i vraća vrijednost pozivaču metode (treba primjetiti da sve metode u Rubyju, ako u njima nije pozvan *return*, vraćaju posljednji evaluirani izraz, za razliku od Pythona koji eksplicitno traži pozivanje returna), *break* koji prekida petlju ili iterator, *next* koji tjera petlju ili iterator da preskoče trenutačnu iteraciju, *redo* koji ponavlja trenutačnu iteraciju, *retry* koji pokreće iterator ispočetka i *throw i catch* koji služe kao *break* koji izlazi iz više razina petlji i/ili metoda odjednom. Detaljan pregled ovih izjava može se naći na [8]

5.5 Iznimke

Iznimke su objekti koji predstavljaju iznimna stanja, tj. da je došlo do neke pogreške. U Rubyju oni pripadaju klasi Exception ili jednoj od njenih podklasa. Po defaultu se pri pojavi iznimke program prekida, ali moguće je definirati blok koda koji se izvrši pri pojavi iznimke. Klasa Exception definira dvije metode koje daju informacije o iznimci: *message*, koja vraća string s detaljima o tome što se dogodilo te *backtrace*, koja vraća polje stringova koji reprezentiraju stog za pozive u trenutku podizanja iznimke. Metodu *raise* možemo pozvati na nekoliko načina. Metoda ima tri opcionalna argumenta: prvi je neki objekt koji ima definiranu metodu *exception*, drugi je string koji će biti poruka pri ispisivanju iznimke, a treći je polje stringova koji će se koristiti kao backtrace iznimke. Ako prvi argument nije dan, iznimka će biti instanca RuntimeError klase, a ako nema trećeg koristit će se backtrace iznimke (koristeći metodu *caller*). Primjer podizanja iznimke:

Listing 55: kod48

```
1 def je_djeljiv?(n,m)
2     raise TypeError, "#{n} nije Integer" if not n.is_a? Integer
3     raise TypeError, "#{m} nije Integer" if not m.is_a? Integer
4     n%m==0 ? true : false
5 end
```

Međutim iznimke se mogu pojaviti bez da smo ih sami podizali pa nam je stoga potreban način za rukovanje iznimkama. U tu svrhu koristimo *rescue* klauzulu. Ona nije sama po sebi izjava, nego se pridružuje drugim izjavama. Najčešće se pridružuje *begin* klauzuli. To izgleda ovako:

Listing 56: kod49

```
1 begin
2     # neki kod
3 rescue
4     # ovaj se kod izvrsi ako dode do iznimke u gornjem kodu
5     # ili u jednoj od metoda koje poziva gornji kod
6 end
```

Unutar *rescuea* možemo referencirati iznimku pomoću globalne varijable $!, a ako želimo koristiti neko drugo ime to radimo ovako:

Listing 57: kod50

```
1 rescue => e      # sada mozemo referencirati iznimku pomocu e
```

Kako *rescue* ne definira novi prostor varijabli, onda varijabla *e* može nastavit postojati i nakon završetka *rescue* klauzule.
Ako želimo da *rescue* rukuje samo nekim tipovima iznimki, onda poslije *rescue* napišemo zarezima odvojen niz klasa iznimki kojima želimo da *rescue* rukuje. Naprimjer:

Listing 58: kod51

```
1 rescue ArgumentError, TypeError
```

Ako nam nije potrebno ime varijable i želimo da *rescue* rukuje svim iznimkama klase StandardError, onda *rescue* možemo koristiti i u ovoj sintaksi:

Listing 59: kod52

```
1 y = 1/x rescue 0
```

Kada se podigne iznimka, Ruby interpreter traži pripadajuću *rescue* klauzulu unutar sadržavajućeg bloka. Ako je tamo ne pronađe, traži u sadržavajućem bloku sadržavajućeg bloka te ako u cijeloj metodi ne nađe pripadajući *rescue*, izlazi iz metode i to bez vraćanja vrijednosti. Proces se nastavlja prema vrhu stoga za pozive. Ako se *rescue* ne nađe, interpreter ispisuje poruku iznimke, backtrace i završava s radom. Primjer:

Listing 60: kod53

```
1  def iznimka
2      raise TypeError
3  end
4
5  def unutarnja
6      begin
7          iznimka
8          return 100
9      rescue ArgumentError
10         puts "ArgumentError!"
11     end
12 end
13
14 def vanjska
15     begin
16         x = unutarnja
17     rescue TypeError
18         puts "TypeError!"
19     end
20 end
21
22 vanjska                              # metoda ce ispisati "TypeError!"
```

Ako unutar *rescue* klauzule iskoristimo *retry* izjavu, ponovno se izvrši blok koda uz koji je *rescue* pridružen. Koristiti *retry* često nema smisla, pogotovo ako je problem programerske prirode, ali ako je došlo do podizanja iznimke zbog problema s konektivnosti ima smisla koristiti *retry*.
Osim *rescue* klauzule, *begin* može sadržavati još i *else* i *ensure* klauzule. Kod unutar *else* klauzule se izvrši samo onda kada nije došlo ni do kakve iznimke i iznimkama pozvanim u njoj neće rukovati *rescue*. S druge strane, *ensure* klauzula sadrži kod koji se uvijek izvrši. Čak i ako pozovemo *return* unutar tijela *begin*, prije samog vraćanja vrijednosti će se izvršiti kod u *ensure* te ako *ensure* sadrži *return*, može se promijeniti vraćena vrijednost (ali zadnja izjava u *ensure* neće mijenjati tu vrijednost). Primjer:

Listing 61: kod54

```
1  begin
2    return 1
3  ensure
4    2               # metoda vraca 1
5   #return 2
6   #kada gornja linija koda ne bi bila zakomentirana
7   #metoda bi vracala 2
8  end
```

Klauzule *rescue, else i ensure* možemo, osim uz *begin* koristiti i uz *def, class* i *module*. Više o njima u sljedećim poglavljima.

5.6 BEGIN i END

BEGIN i END su ključne riječi u Rubyju koje označavaju kod koji se treba izvršiti na početku, odnosno na kraju programa. Ako postoji više BEGIN izjava, izvršavaju se redom kojim interpreter dođe do njih, dok za END izjave vrijedi obrnuti redoslijed izvršavanja. Kod koji slijedi poslije BEGIN ili END mora biti omeđen vitičastim zagradama koje se ne mogu zamijeniti s *do i end*. Ako se BEGIN ili END nalaze unutar petlje, izvršit će se samo jednom, a ako se BEGIN nalazi unutar neke uvjetne strukture, izvršit će se neovisno o istinitosti uvjeta.

6 Metode

Metode su imenovani blokovi parametriziranog koda asocirani s jednim ili više objekata. Poziv metode sastoji se od imena metode, objekta nad kojim se poziva i nula ili više argumenata. Posljednja evaluirana vrijednost izraza u metodi postaje vrijednost poziva metode. Neki jezici, poput Pythona, razlikuju metode i funkcije, no u Rubyju su sve metode prave metode koje su asocirane uz neki objekt pa u njemu prave funkcije ne postoje.
Za razliku od blokova, koji su također komadi parametriziranog koda, metode imaju imena i mogu se direktno pozivati, a ne nužno indirektno preko iteratora. Sintaksa pozivanja metoda je sljedeća:

Listing 62: kod55

```
1 objekt.imemetode(arg1,arg2,...,argn)
```

Ime metode je jedini nužni dio poziva metode i prema dogovoru počinje malim slovom. Ako izostavimo objekt, onda se metoda poziva na *self*. Zagrade oko argumenata su obično nepotrebne, a neke metode očekuju nula argumenata. Na kraju možemo imati i blok koda omeđen vitičastim zagradama ili s *do/end*.
Metode definiramo koristeći ključnu riječ *def* nakon koje slijedi ime metode te zagradama omeđen niz imena argumenata (zagrade možemo izostaviti ako metoda ne prima niti jedan argument). Nakon toga slijedi tijelo metode koje završava ključnom riječi *end*.
Ime metode može završavati s jednim od tri posebna znaka: upitnikom, uskličnikom ili znakom jednakosti. Metode koje završavaju upitnikom nazivamo predikatima i prema dogovoru odgovaraju na neko pitanje (poput npr. Fixnum metode *odd?* koja vraća *true* za neparne, a *false* za parne brojeve). Metodama stavljamo uskličnik na kraj ako želimo naglasiti da se upotrebljavaju s oprezom. To su često metode koje direktno mijenjaju objekt na kojem su pozvane (poput npr. Array metode sort!, za razliku od metode sort). Metode čije ime završava znakom jednakosti zovemo *setter* metodama i njima ćemo se baviti u sljedećem poglavlju.
Kao i u Pythonu, u Rubyju možemo definirati default vrijednosti nekih parametara tako da nakon njihovih imena stavimo znak jednakosti i vrijednost. Ruby zahtijeva da parametri s default vrijednostima budu svi jedni kraj drugih u nizu parametara.
Ruby nam također dozvoljava definiranje metoda s varijabilnim brojem parametara tako da ispred imena parametra stavimo znak *. Taj će parametar tada biti polje koje sadrži argumente proslijeđene na toj poziciji. Primjer:

Listing 63: kod56

```
1  def sumiraj(n,ispis=false,*sumandi)
2    suma=n
3    sumandi.each{|x| suma+=x}
4    if ispis
5      puts suma
6    end
7    suma
8  end
9  sumiraj(10)              # => 10, nema ispisa
10 sumiraj(10,true)         # => 10, s ispisom
11 sumiraj(10,true,4,2)     # => 16, s ispisom
```

Korištenjem *return* izjave možemo vratiti više vrijednosti odjednom (spremljenih u polje):

Listing 64: kod57

```
1 def foo(x,y)
2   if x==y
3     return 2*x,x+1
4   else
5     2*x
6   end
7 end
8
9 v,w = foo(3,3)        # => [6,4]
```

Možemo definirati tri vrste metoda: globalne metode, metode klase i singleton metode. Globalne metode definiramo izvan *class* izjava. One su metode klase Object i implicitno se pozivaju na *self*. Ako metodu definiramo unutar *class* izjave, onda ona postaje metoda te klase i definirana je na svim objektima koji pripadaju toj klasi. Naposlijetku, singleton metode su metode definirane na jednom objektu tako da nakon ključne riječi *def* stavimo izraz koji se evaluira kao neki objekt. Nakon toga stavimo točku i ime metode. Primjer:

Listing 65: kod58

```
1 x = [1,2,3]
2 def x.suma
3   self[0]+self[1]+self[2]
4 end
5 y=x.suma                    # => 6
```

Bitno je napomenuti kako Ruby ne dozvoljava definiranje singleton metoda za Numeric i Symbol objekte.
Jedno vrlo zanimljivo svojstvo metoda u Rubyju je to da jedna metoda može imati više imena. Točnije, u Rubyju postoji ključna riječ *alias* čija sintaksa glasi:

Listing 66: kod59

```
1 def foo
2   ...
3 end
4 alias bar foo
5 bar                         # identicno kao da smo pozvali foo
```

Jedna od primjena *alias* ključne riječi je dodavanje funkcionalnosti metodama. Primjer:

Listing 67: kod60

```
1 def foo
2   puts "foo v1"
3 end
4
```

```
5  alias bar foo
6
7  def foo
8    puts "foo v2"
9    bar                          # poziv originalne foo metode
10 end
```

6.1 Lambda i Proc

Kao što je napomenuto, metodama u Rubyju možemo proslijediti blok kao argument. Međutim, blokovi nisu objekti i zbog toga ne možemo s njima raditi određene stvari. No, možemo napraviti objekte koji reprezentiraju blokove: lambde i procove, koji su oboje instance klase Proc. Lambde se ponašaju poput metoda, dok se procovi ponašaju poput blokova.
Postoji nekoliko načina za stvaranje Proc objekata. Jedan od načina je definiranje metode s kojom asociramo blok. Dobiveni Proc objekt ovom metodom je uvijek proc. Svi Proc objekti imaju metodu *call* koja bloku prosljeđuje svoje argumente i onda pokreće kod unutar bloka:

Listing 68: kod61

```
1 def makeproc(&p)                 # & oznacava da je p blok
2   p
3 end
4
5 identiteta = makeproc{|x| x}
6 identiteta.call(2)               # => 2
```

U Rubyju postoje dva dodatna načina pozivanja Proc objekata koje možemo koristiti umjesto *call* metode:

Listing 69: kod62

```
1 identiteta[2]
2 identiteta.(2)
```

Kako su procovi objekti, možemo s njima raditi sve što možemo s drugim objektima:

Listing 70: kod63

```
1 suma = makeproc{|x,y| x+y}
2 razlika = makeproc{|x,y| x-y}
3
4 polje = [identiteta,suma,razlika]
5 polje.each{|f| puts f.call(10,5)}        # ispisuje se 10,15,5
```

Gore naveden način definiranja Proc objekata nije uobičajeno korištena, već se najčešće koriste tri već definirane metode za stvaranje Proc objekata (i procova i lambdi) te sintaksa definiranja lambdi kao literala. Prva od tih metoda je Proc.*new* koja ne očekuje niti jedan argument i vraća proc. Ako je pozovemo s asociranim blokom, tada vraća proc koji reprezentira taj blok. Ako je pozvana bez asociranog bloka unutar neke metode, tada vraćeni

proc reprezentira blok koji je asociran metodi. Metoda Kernel.*proc* je sinonim ove metode. Druga metoda je Kernel.*lambda* (koju zato što pripada Kernel modulu pozivamo samo s *lambda*). Metoda ne očekuje argumente, ali zahtijeva asociran blok i vraća lambdu koja reprezentira taj blok. Primjeri:

Listing 71: kod64

```
identiteta = Proc.new{|x| x}
suma = proc{|x,y| x+y}
razlika = lambda{|x,y| x-y}
```

Ruby podržava sintaksu definiranja lambdi kao literala koju možemo koristiti umjesto Kernel.*lambda*. Sintaksa je sljedeća: *lambda* zamijenimo s ->; listu argumenata stavimo unutar običnih zagrada koje stavimo između -> i {. Primjer:

Listing 72: kod65

```
razlika = ->(x,y){x-y}
```

Kao što je već prije napomenuto, procovi se ponašaju poput blokova, a lambde poput metoda. Jedna razlika između njih je ponašanje *return* izjave. Ako koristimo *return* unutar proca koji pozovemo unutar metode, *return* izlazi iz metode. Međutim, kada u istoj situaciji koristimo lambdu umjesto proca, *return* izlazi samo iz lambde:

Listing 73: kod66

```
def foo
  p = proc{|x| return x}
  p[0]
  return 1                # nece se izvrsiti
end
def bar
  p = lambda{|x| return x}
  p[0]
  return 1
end

foo                       # => 0
bar                       # => 1
```

Osim *return*a, razlike se pojavljuju u ponašanju *break* izjave: *break* se unutar proca ponaša kao unutar bloka, dok se unutar lambde ponaša kao unutar metode. Posljednja razlika između procova i lambdi se pojavljuje kod proslijeđivanja argumenata. Naime, proc koristi istu semantiku kao i *yield*, tj. koriste se pravila paralelnog pridruživanja vrijednosti, dok lambda koristi istu semantiku kao i invokacija metoda:

Listing 74: kod67

```
p = proc{|x,y| x}
q = lambda{|x,y| x}
```

```
3  p[1]          # x, y=1                                    => 1
4  q[1]          # ArgumentError, krivi broj argumenata
5  p[1,2]        # x, y=1,2        isto kao i q[1,2]         => 3
6  p[1,2,3]      # x, y=1,2,3                                => 3
7  q[1,2,3]      # ArgumentError, krivi broj argumenata
8  p[[1,2]]      # x, y=[1,2]                                => 3
9  q[[1,2]]      # ArgumentError, krivi broj argumenata
10 q[*[1,2]]     # kao i q[1,2]                              => 3
```

Konačno, valja proći i kroz sličnosti između procova i lambdi: *next, redo i raise* funkcioniraju identično u procovima, lambdama i blokovima, dok *retry* uvijek rezultira LocalJumpErrorom i u lambdama i u procovima.

7 Objektno Orijentirano Programiranje u Rubyju

Ruby je u potpunosti objektno orijentiran jezik. Zbog toga je ovo poglavlje najbitnije poglavlje ovoga rada. U njemu ćemo proći kroz pojmove klasa, modula i mixina te metoda i varijabli u smislu OOP.

7.1 Klase

Svaki objekt pripada nekoj klasi. Stoga ima smisla započeti ovo poglavlje klasama. Klase se u Rubyju definiraju pomoću ključne riječi *class* nakon koje slijedi ime klase (koje mora počinjati velikim slovom), tijelo klase te ključna riječ *end*:

Listing 75: kod68

```
1 class Pravokutnik
2   # ...
3 end
```

Kao što u Pythonu postoji *__init__*, u Rubyju postoji metoda kojom definiramo inicijalizaciju objekata klase: *initialize*. Objekte inicijaliziramo pozivanjem metode *new*:

Listing 76: kod69

```
1 class Pravokutnik
2   def initialize(a,b)
3     @a, @b = a,b
4   end
5 end
6
7 p = Pravokutnik.new(5,6)
```

Listing 77: kod69 Python

```
1 class Pravokutnik:
2   def __init__(self,a,b):
3     self.a,self.b=a,b
4
5 p = Pravokutnik(5,6)
```

Jedna od metoda koju bismo htjeli imati u klasi Pravokutnik je metoda koja vraća površinu pravokutnika sa duljinama stranica a i b:

Listing 78: kod70

```
1 def povrsina
2   @a*@b
3 end
```

Oni koji su upoznati s objektno orjentiranim programiranjem u Pythonu bi mogli očekivati kako je u Rubyju moguće pristupiti atributima *a i b* na sljedeći način:

Listing 79: kod71

```
1 p.a
```

Međutim, Ruby ne dozvoljava direktan pristup atributima objekta, već im isključivo možemo pristupiti preko metoda klase tog objekta. Ruby će gornji kod interpretirati kao poziv metode *a* (podsjetimo se da Ruby ne zahtijeva nužno zagrade pri pozivu metoda) tako da ako bismo htjeli pristupiti atributu *a* možemo definirati metodu *a* koja će vraćati vrijednost

atributa *a*. Ako pak želimo da klasa bude mutabilna, definiramo metodu *a=* koja prima jedan argument i postavlja *a* na tu vrijednost. Te metode nazivamo getter, odnosno setter metodama:

Listing 80: kod72

```
1  class Pravokutnik
2    def initialize(a,b)
3      @a, @b = a,b
4    end
5
6    def a
7      @a
8    end
9    def a=(value)
10     @a=value
11   end
12 end
```

Ruby definira dvije metode koje nam olakšavaju stvaranje getter i/ili setter metoda: *attr_accessor i attr_reader*, pogotovo ako ih želimo definirati za više atributa:

Listing 81: kod73

```
1  class Pravokutnik
2    # definiranje getter i setter metoda za a i b
3    attr_accessor :a, :b  # moze i attr_accessor "a", "b"
4    # ako zelimo samo getter metode, koristimo attr_reader
```

Kako su većina operatora u Rubyju metode, možemo ih definirati za našu klasu:

Listing 82: kod74

```
1  class Pravokutnik
2    attr_accessor :a, :b
3    def initialize(a,b)
4      @a,@b=a,b
5    end
6
7    def povrsina
8      @a*@b
9    end
10
11   def ==(p)
12     if p.is_a? Pravokutnik
13       @a==p.a && @b==p.b
14     else
15       false
16     end
17   end
```

```
18
19    # Mozemo definirati i eql? metodu za strozu usporedbu
20    def eql?(p)
21      if p.instance_of? Pravokutnik
22        @a.eql?(p.a) && @b.eql?(p.b)
23      else
24        false
25      end
26    end
27
28    def *(skalar)
29      Pravokutnik.new(@a*skalar,@b*skalar)
30    end
31
32    # Ruby nece implicitno znati sto napraviti ako zamijenimo
33    # redoslijed operanada za gornji operator
34    # Zato definiramo coerce metodu
35    def coerce(arg)
36      [self, arg]
37    end
38  end
```

Moguće je i definirati metodu [] ako želimo da objekti naše klase budu slični poljima ili hashevima, *each* metodu ako želimo da budu iterabilni, <=>metodu ako ih želimo moći uspoređivati itd. Primijetimo kako smo za redefiniranje operatora ==, * itd. direktno koristili te simbole za njihova imena. To je intuitivno, za razliku od pristupa u Pythonu koji bi za redefiniranje npr. operatora * tražio promjenu metode __mul__.
Osim gore definiranih metoda instance, koje pozivamo na objektima klase Pravokutnik, možemo definirati i metode klase:

Listing 83: kod75

```
1  # Definirajmo metodu koja ce za arbitraran broj Pravokutnika
2  # vracati sumu njihovih povrsina
3  class Pravokutnik
4    # ...
5    def Pravokutnik.povrsina(*pravokutnici)
6    # mogli smo umjesto Pravokutnik staviti self
7      suma = 0
8      pravokutnici.each {|p| suma+=p.povrsina}
9      suma
10   end
11 end
12
13 p=Pravokutnik.new(2,3)
14 q=Pravokutnik.new(4,4)
15
16 Pravokutnik.povrsina(p,q)                    # => 22
```

Osim metodi i varijabli instance, možemo još definirati i varijable klase, konstante i varijable instance klase.

Listing 84: kod76

```
class Pravokutnik
  # Konstantama mozemo pristupiti i definirati ih
  # izvan definicije klase na sljedeci nacin:
  # Pravokutnik::JEDINICNI_KVADRAT = Pravokutnik.new(1,1)
  JEDINICNI_KVADRAT = Pravokutnik.new(1,1)
  # Varijable klase su vidljive i dijele ih
  # metode klase, metode instance i sama definicija klase
  # Kao i varijablama instanci, ne mozemo im direktno
  # pristupati izvan klase
  @@n = 0                  # ukupan broj stvorenih pravokutnika
  def initialize(a,b)
    @a,@b=a,b
    @@n+=1
  end
  def n
    @@n
  end
end
```

Kako su klase objekti i one mogu imati varijable instance. Varijable instance definirane unutar definicije klase a izvan definicija metodi instance nazivamo varijablama instanci klase. Vrlo su slične varijablama klase, ali postoji nekoliko razlika. Naime, drugačije se ponašaju kada je u pitanju nasljeđivanje te se varijable instanci klase ne mogu koristiti unutar metoda instance pa bismo gornji kod morali preoblikovati na sljedeći način:

Listing 85: kod77

```
class Pravokutnik
  @n = 0
  # ne mozemo pristupiti n iz initialize
  def initialize(a,b)
    @a,@b=a,b
  end
  # ali mozemo pristupiti iz npr. Pravokutnik.new
  def self.new(a,b)
    @n += 1
    super        # poziva originalni new, vise o tome kasnije
  end
end
```

Za razliku od Pythona, u Rubyju, kao i u mnogim drugim objektno orjentiranim jezicima, metode instance mogu imati različitu vidljivost, tj. mogu biti *public, private ili protected*. Metode osim *initialize*, koja je implicitno *private*, su implicitno *public*. Takve metode se mogu pozvati bilo gdje. Za razliku od njih, *private* metode mogu pozivati samo druge metode instance te klase (ili njenih potklasa) te se implicitno pozivaju na *self*. Konačno,

protected metode su poput *private* metoda s tim da se mogu eksplicitno pozivati na instancama klase.
Postoje dva načina za definiranje vidljivosti metoda. Oba načina koriste *public, private i protected* metode koje se mogu koristiti poput ključnih riječi. Prvi način je koristiti ih bez argumenata. Tada sve metode čije definicije ih slijede imaju vidljivost koju oni specificiraju:

Listing 86: kod78

```
1  class Pravokutnik
2    # metode koje su ovdje su implicitno public
3
4    private
5    # metode koje su ovdje su private
6
7    protected
8    # metode koje su ovdje su protected
9
10   public
11   # metode koje su ovdje su eksplicitno public
12 end
```

Drugi način je koristiti imena metoda, kao simbole ili stringove, kao argumente. Tada te metode postavljamo na vidljivost koju specificiraju metode *public, private,* odnosno *protected.* U tom slučaju, metode moramo definirati prije nego im promijenimo vidljivost.

Listing 87: kod79

```
1  class Pravokutnik
2    # ...
3    def povrsina
4      @a*@b
5    end
6    private :povrsina                # povrsina postaje private metoda
```

Osim metoda instance, možemo definirati vidljivost i metoda klase koristeći *private_class_method i public_class_method* metode koje koristimo kao *public i private* metode kod metoda instance, ali bez mogućnosti korištenja bez argumenata. Za razliku od nekih jezika, poput C++, u Rubyju nije moguće postavljati vidljivost varijabli: Varijable instance i klase su efektivno *private*, a konstante *public.*
U većini objektno orijentiranih jezika, uključujući Python i Ruby, postoji mehanizam nasljeđivanja, odnosno definiranja potklasa neke klase. Kada definiramo neku klasu, možemo naznačiti da nasljeđuje neku drugu klasu koju tada zovemo natklasa. Sve klase u Rubyju imaju točno jednu natklasu, osim klase BasicObject koja nema nijednu. Ako ne specificiramo natklasu kada definiramo klasu, implicitno joj kao natklasu stavljamo Object, čija je natklasa BasicObject. U Pythonu je pak moguće da jedna klasa ima više natklasi. Sintaksa nasljeđivanja je jednostavna:

Listing 88: kod80

```
1  # Klasa Kvadrat koja je potklasa klase Pravokutnik
```

```
2 class Kvadrat < Pravokutnik
3 end
```

Kada smo definirali gornju potklasu, ona je nasljedila većinu metoda iz klase Pravokutnik:

Listing 89: kod81

```
1 p=Pravokutnik.new(4,4)
2 q=Kvadrat.new(4,4)
3 p==q              # => true
4 p.eql?(q)         # => false
5 q.eql?(p)         # => true (jer koristi eql? metodu iz Pravokutnik)
6 q.povrsina        # => 16
7 Pravokutnik.povrsina(p,q)        # => 32
8 Kvadrat.povrsina(p,q)            # => 32
9 q.class           # => Kvadrat
```

Međutim, nekad ne želimo nasljediti određene metode natklase ili želimo drugačije ponašanje te metode u potklasi. Tada koristimo ili *undef* kako ne bismo nasljedili određene metode ili ih nanovo definiramo s *def* kako bismo dobili drugačije ponašanje. Ako pak želimo zadržati originalno ponašanje uz određene dodatke, koristimo ključnu riječ *super*: *super* funkcionira poput poziva metode; pozove metodu s istim imenom kao i metoda u kojoj se nalazi, ali u natklasi klase unutar koje se nalazi. Možemo proslijediti argumente kao i kod normalnog poziva metode:

Listing 90: kod82

```
1  class Kvadrat
2    def initialize(a)
3      super(a,a)            # poziva initialize(a,a) iz Pravokutnik
4    end
5
6    def eql?(k)
7      if k.instance_of? Kvadrat
8        @a.eql?(k.a)
9      else
10       false
11     end
12   end
13 end
14 p=Pravokutnik.new(4,4)
15 q=Kvadrat.new(4,4)
16 q.eql?(p)         # => false
```

Ako koristimo *super* bez argumenata i zagrada, onda *super* prosljeđuje argumente koji su bili prosljeđeni metodi u kojoj se *super* nalazi. Ako ne želimo prosljediti nijeda argument, moramo uz *super* staviti praznu zagradu.
Varijable se pak u Rubyju ne nasljeđuju. Varijable instance se definiraju kada im se pridruži vrijednost pa nam se može činiti suprotno ako koristimo *super* za pozivanje inicijalizatora natklase. Kako su varijable instanci klase u biti varijable instance, ista stvar vrijedi i za

njih tako da one nisu dijeljene između potklase i natklase. S druge strane, varijable klase su dijeljene između klase i svih njenih potklasa:

Listing 91: kod83

```
class Foo
  @@foo = 0
  def self.foo
    @@foo
  end
end
class Bar<Foo
  @@foo = 1
end
Bar.foo          # => 1
# ali takoder i:
Foo.foo          # => 1
```

S druge strane, konstante se nasljeđuju poput metoda instance, ali s par razlika. Klasa Kvadrat se može referirati na konstantu JEDINICNI_KVADRAT klase Pravokutnik ne samo kao Pravokutnik::JEDINICNI_KVADRAT, već i kao JEDINICNI_KVADRAT i Kvadrat::JEDINICNI_KVADRAT. Također, redefiniranjem konstante se zapravo stvara nova konstanta Kvadrat::JEDINICNI_KVADRAT te klasa Kvadrat može koristiti obje konstante. Konačno, ako Kvadrat naslijedi neku metodu koja koristi JEDINICNI_KVADRAT, koristit će se verzija konstante iz klase Pravokutnik.

7.2 Moduli

Osim klasa, Ruby sadrži još jednu strukturu koja je imenovani skup metoda, konstanti i varijabli - module. Oni su objekti klase Module koja je natklasa klase Class. Moduli se definiraju poput klasa, ali s ključnom riječi *module* umjesto *class*. Moduli se ne mogu instancirati i ne postoji mehanizam nasljeđivanja. Koriste se u dvije svrhe: kao prostori imena i kao mixini.
Dobar primjer korištenja modula kao prostora imena možemo pronaći u [2]: pretpostavimo da želimo napraviti metode za kodiranje i dekodiranje binarnih podataka u i iz teksta koristeći kodiranje u bazi 64. Nemamo potrebe raditi posebne objekte za kodiranje i dekodiranje pa nema potrebe za definiranjem klasa. Potrebne su nam samo dvije metode, jedna za kodiranje i jedna za dekodiranje:

Listing 92: kod84

```
def encode
  # ...
end
def decode
  # ...
end
```

No, imena ovih metoda bi mogla dovesti do problema s drugim metodama koje kodiraju ili dekodiraju. Jedan način za riješiti ovaj problem bio bi staviti prefiks *base64* na imena metoda. Međutim, uobičajena praksa je izbjegavati dodavanje metoda u globalni prostor imena. Stoga je bolje rješenje definirati te dvije metode unutar modula Base64:

Listing 93: kod85

```
1 module Base64
2   def self.encode
3     # ...
4   end
5   def self.decode
6     # ...
7   end
8 end
```

Sada te metode pozivamo na sljedeći način:

Listing 94: kod86

```
1 text = Base64.encode(data)
2 data = Base64.decode(text)
```

Moduli mogu sadržavati konstante i varijable klase kojima pristupamo isto kao i konstantama i varijablama klase u klasama. Također, imena modula, poput imena klasa, moraju počinjati velikim slovom.
Druga svrha u koju koristimo module je stvaranje mixina. Mixini su moduli s definiranim metodama instance koje onda "ubacimo" u druge klase. Ako imamo npr. klasu s definiranim operatorom <=>, ubacivanjem modula Comparable dobijamo <,<=,==,>,>= i *between?*. Kako bismo to napravili, koristimo metodu *include*. Ona je *private* metoda klase Module koja se implicitno poziva na *self*:

Listing 95: kod87

```
1 class MojaKlasa
2   def <=>(o)
3     # ...
4   end
5   include(Comparable)    # moze i bez zagrada
6 end
```

Metoda *include* isključivo prima "prave" module kao argumente, tj. ne prima klase iako je Class potklasa od Module. No, možemo ubaciti modul u neki drugi modul. Ubacivanje modula u klasu utječe na metodu *is_a?* i operator ===, dok metoda *instanceof?* isključivo provjerava klasu objekta na kojem je pozvana pa na nju ne utječu ubačeni moduli.
Drugi način ubacivanja modula je korištenje Object.*extend* metode. Ona pretvara metode instance modula u singleton metode objekta na kojem je pozvana, a ako je taj objekt instanca klase Class, te metode postanu metode klase te klase. Primjer:

Listing 96: kod88

```
p = Pravokutnik.new(6,4)
def p.each
  yield @a
  yield @b
end
p.extend(Enumerable)      # dobivamo, izmedu ostalog, sort metodu
print p.sort              # Ispisuje "[4, 6]"
```

Neke module, poput Math ili Kernel, možemo koristiti i kao prostor imena i kao mixine.

7.3 Eigenclass

U poglavlju 6 smo govorili o singleton metodama. Kako su klase objekti klase Class, možemo definirati singleton metode za njih:

Listing 97: kod89

```
def Pravokutnik.foo
  # ...
end
```

Dakle, vidimo kako su metoda klase ništa drugo nego singleton metode na instanci klase Class koja reprezentira tu klasu. Singleton metode objekta nisu definirane klasom tog objekta, ali kako su metode, moraju biti asocirane s nekom klasom. One su zapravo metode instance neimenovane svojstvene klase (eigenclass) asocirane s tim objektom. Drugi termini za tu klasu su singleton klasa i metaklasa. U Rubyju je moguće dodavati metode u svojstvenu klasu objekta te tako odjednom definirati više singleton metoda nekog objekta. Sintaksa je sljedeća:

Listing 98: kod90

```
p = Pravokutnik.new(5,6)
class << p
  def singl_met1
    # ...
  end
  def singl_met2
    # ...
  end
end

# identicno mozemo dodavati metode klase nekoj klasi
class << Pravokutnik
  def met_klase1
    # ...
  end
  def met_klase2
```

```
17      # ...
18    end
19  end
20
21  # mogli smo to napraviti i unutar definicije klase
22  class Pravokutnik
23    # ...
24    class << self          # mogli smo Pravokutnik umjesto self
25      # ...
26    end
27  end
```

8 Zaključak

Tijekom ovog rada, prošli smo kroz osnove programskog jezika Ruby te ga uspoređivali s Pythonom. Oba jezika imaju svoje prednosti, poput implicitnog vraćanja vrijednosti u metodama u Rubyju ili bogatiji skup podatkovnih struktura Pythona. Također, postoji puno razlika za koje ne možemo reći jesu li prednosti ili mane, poput razlika u evaluiranju vrijednosti u smislu *true* ili *false*. Međutim, postoji i mnogo sličnosti. Oba jezika su dinamični jezici jake tipizacije i oba jezika su objektno orijentirana. Zbog toga je nemoguće dati odgovor na pitanja tipa "Je li bolji Ruby ili Python?" jer se odgovor na to pitanje mijenja ovisno o tome koji problem želimo riješiti.

9 Literatura

[1] O. Frieder, G. Frieder, D. Grossman, Computer Science Programming Basics in Ruby, O'Reilly Media, SAD, 2013.
[2] D. Flanagan, Y. Matsumoto, The Ruby Programming Language, O'Reilly Media, SAD, 2008.
[3] http://blade.nagaokaut.ac.jp/cgi-bin/scat.rb/ruby/ruby-talk/382
[4] https://www.ruby-lang.org/en/
[5] http://www.etfos.unios.hr/ lukic/oop/Auditorne_vježbe_1.pdf
[6] https://docs.python.org
[7] http://pythoncentral.io/pythons-range-function-explained/
[8] https://ruby-doc.org/core-1.9.3/Kernel.html