```
3  i = 123
4  j = 99
5  y = 123.45678912345
6
7  print "Vrijednosti su ", i,j,y
8  print u"########## Print naredba s razlicitim formatiranjem
       ##############"
9  print u"d - cijeli broj: %d" % (i)
10 print u"d - broj klizeceg zareza: %f" % (y)
11 print u"f - broj klizeceg zareza konacnog formata: %6.2f" % (y)
12 print u"f - broj klizeceg zareza konacnog formata: %6.1f" % (y)
13 print u"g - realni broj u eksponencijalnom zapisu: %g" % (y)
14 print u"g - realni broj u eksp. zapisu u konacnom formatu %6.2g
       " % (y)
15 print u"s - Vrijednost mog stringa je %s" % (x)
16 print u"int s 4 znaka razmaka: %4d %4d" % (i,j)
17 print u"int s 4 znaka razmaka: %4d %4d" % (j,i)
18
19 # print naredba automatski ukljucuje silazak u novi red
20 # dodatkom zareza (,)  ispustamo silazak u novi red
21
22 print u"d - an integer: %d " % (i) ,
23 print u"f - a floating point number: %f" % (y)
24
25 # razlicite primjene escape znakova
26
27 print r"Raw string s \n i \t u njemu"
28 print u"String s \n i \t u njemu"
```

što daje:

Zaslon 3.21

```
*** Remote Interpreter Reinitialized  ***
>>>
Vrijednosti su  123 99 123.456789123
########## Print naredba s razlicitim formatiranjem ##############
d - cijeli broj: 123
d - broj klizeceg zareza: 123.456789
f - broj klizeceg zareza konacnog formata: 123.46
f - broj klizeceg zareza konacnog formata:  123.5
g - realni broj u eksponencijalnom zapisu: 123.457
g - realni broj u eksp. zapisu u konacnom formatu 1.2e+02
s - Vrijednost mog stringa je Dobar dan
int s 4 znaka razmaka:  123   99
int s 4 znaka razmaka:   99  123
d - an integer: 123  f - a floating point number: 123.456789
Raw string s \n i \t u njemu
String s
```

```
 i   u njemu
>>>
```

S obzirom na veće mogućnosti koje pruža *format()* metoda, budući formatirani ispisi koristit će njenu snagu, te uključivati tumačenje parametara koji će se s njom koristiti. Dakako, oni se oslanjaju na već protumačene tipove podataka, posebice rječnika i liste. Ovdje će se dati samo neki primjeri koji su česti u radu sa stringovima:

Zaslon 3.22

```
>>> "{} {} {}".format("volim", "Python", "jezik")
'volim Python jezik'
>>> "{0} {1} {2}".format("volim", "Python", "jezik")
'volim Python jezik'
>>> "{0} {2} {1}".format("volim", "Python", "jezik")
'volim jezik Python'
>>> "{1} {2} {0}".format("volim", "Python", "jezik")
'Python jezik volim'
>>>
```

Primjetite kako se izostavljeni indeksi trojke zamjenjuje logičnim slijedom '{0}', '{1}', '{2}'.

U idućem primjeru vidi se primjena pozicijskog (n-troka) indeksa i imenovanog argumenta (rječnik).

Zaslon 3.23

```
>>> u"Gubec probesjedi {0} da se orilo preko seljačkih {cega}."
.format('glasno',cega='glava')
u'Gubec probesjedi glasno da se orilo preko seljačkih glava.'
>>>
```

ili za pozicioniranje teksta, lijepi ispis:

Zaslon 3.24

```
s = "Hrvatski narod"
print "{0}".format(s)       # pretpostavljeno formatiranje
print "{0:25}".format(s)    # minimalna širina 25
print "{0:>25}".format(s)   # desno poravnjanje na 25
print "{0:^25}".format(s)   # centralno poravnjanje na 25
print "{0:-^25}".format(s)  # - punjenje, centralno poravnjanje na 25
print "{0:.<25}".format(s)  # . punjenje, lijevo poravnjanje na 25
print "{0:.10}".format(s)   # maksimalna širina 10

>>>
Hrvatski narod
Hrvatski narod
           Hrvatski narod
     Hrvatski narod
-----Hrvatski narod------
Hrvatski narod.....
Hrvatski n
>>>
```

Ili malo složeniji ispis:

Zaslon 3.25

```
>>> print 'Ovo je {0:->10} \n bilo je {1:.>10}, i to je
{1:.<9}, ako {2:*^10}'.format('sef',[2,4],99)

Ovo je -------sef
 bilo je ....[2, 4], i to je [2, 4]..., ako ****99****
>>>
```

3.5 Iznimke

Pišući program, korisnik (programer) može načiniti tri vrste pogrešaka: *logičke pogreške*, *sintaktičke pogreške*, te *pogreške u radu* (eng. *run-time errors*) koje se pojavljuju tijekom izvođenja programa.

Logička pogreška poput neispravnog algoritma, uzrokuje netočne rezultate, ali ne spriječava izvođenje programa. Ovakve se pogreške teško mogu uočiti.

Sintaktička pogreška krši jedno od Pythonovih gramatičkih pravila i spriječava izvođenje programa. Ove je pogreške lako popraviti.

Pogreška u radu je pogreška kod izvršenja programa koja se događa dok je program u nekoj operaciji. Neki česti uzroci takvih pogrešaka u radu su neprimjereni ulazni podaci, arithmetičke pogreške, neispravne vrste objekata, nizovi indeksa izvan dometa, nepostojeći ključevi rječnika, nepravilno ispisana imena atributa, neinicijalizirane varijable, te problemi vezani uz operacijski sustav.

Iako to ne pomaže u slučaju logičkih pogrešaka, Python podiže *iznimku* (engl. *exception*) pri otkrivanju sintaktičkih pogrešaka ili pogrešaka tijekom izvođenja programa. Interpreter zaustavlja program te ispisuje dijagnostičku poruku o pogrešci, zvanu "*traceback*", koja ukazuje na vrstu iznimke te pokazuje približno mjesto pogreške. Sintaksne pogreške su najčešće pogreške u pisanju programa, a na njih upozorava Python prilikom izvođenja programa oznakom strijelice na mjestu na kojem se pogreška javlja. To mjesto ne mora doista biti ono na kojem se pogreška nalazi, nego je obično ono na kojem se prvi put pogreška manifestira. Točno lociranje pogrešaka nije jednostavno.

Zaslon 3.26 — Sintaksna pogreška.

```
>>> while True print Pozdrav svima'
File "<stdin>", line 1, in ?
while True print 'Pozdrav svima'
           ^
SyntaxError: invalid syntax
>>> ime="Mario
  File "<stdin>", line 1
    ime="Mario
             ^
SyntaxError: EOL while scanning string literal
>>>
```

Zaslon 3.27 — Logička pogreška.

```
>>> x,y=3,4
>>> prosjek=x+y/2
```

```
>>> print prosjek
5
```

U ovom primjeru se logička pogreška događa jer je potrebno izračunati srednju vrijednost po izrazu $\frac{x+y}{2}$, a u programski kôd se napiše izraz $x+y/2$ i tako se greškom umjesto 3.5 dobije se broj 5 kao rezultat.

Zaslon 3.28 — Pogreška u radu.

```
>>> broj=3+2/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
```

Python izvodi program u dva prolaska. U prvom prolasku provjerava ima li sintaksnih pogrešaka i ako ih nema onda ide u izvođenje kôda. Stoga, ako se sintaksna pogreška dogodi, nikakav se programski kôd neće izvršiti, dok ako se dogodi pogreška u radu, izvršit će se programski kôd do te pogreške.

Kada je program sintaksno dobro napisan, postoji mogućnost da se pogreške jave prilikom izvođenja programa. *Iznimka* je način na koji Python dohvaća i pogreške tog tipa.

3.5.1 Vrste iznimki

Python organizira iznimke po hijerarhijskom stablu. Na vrhu stabla iznimki je `Exception`, iz kojega su izvedene sve druge ugrađene iznimke, od kojih su prvi nasljednici: `SystemExit` i `StandardError`.

Ovakav hijerarhijski ustroj omogućuje otkrivanje i obradu skupnih, a ne samo individualnih iznimki. Na primjer, ako postoji skupina iznimki koje obrađuju matematičke izračune, onda se može prvo dohvatiti samo `ArithmeticError` iznimka, a ne i sva njezina djeca (`FloatingPointError`, `OverflowError` i `ZeroDivisionError`) pojedinačno, uz uvjet da se želi na isti način raditi sa svim iznimkama.

Evo nekih primjera čestih pogrešaka u programu koji podižu iznimke:

Zaslon 3.29

```
>>> 10 * (1/0)
Traceback (most recent call last):
File "<stdin>", line 1, in ?
ZeroDivisionError: integer division or modulo by zero
>>> 4 + miki*3
Traceback (most recent call last):
File "<stdin>", line 1, in ?
NameError: name 'miki' is not defined
>>> '2' + 2
Traceback (most recent call last):
File "<stdin>", line 1, in ?
TypeError: cannot concatenate 'str' and 'int' objects
```

Posljednja linija poruke o pogreški govori što se dogodilo. Budući da postoji više vrsta iznimaka, njihov tip se nalazi ispisan u poruci (npr. *ZeroDivisionError*, *NameError* i *TypeError*).

Svaka Python ugrađena iznimka podižu `true` pri pojavi pogreške njenog tipa. Ostatak ispisane linije poruke pruža detaljniji opis pogreške (`TypeError`), a dio prije imena iznimke govori o kontekstu u kojem se pogreška dogodila. Taj kontekst je u obliku slijeda (`Traceback`) obavijesti, od posljednjeg mjesta, rekurzivno prema početku moguće pogreške.

3.5.2 Rad s iznimkama

U Pythonu je moguće pisati programe koji će raditi (dohvaćati i obrađivati) promatrane iznimke, što se vidi na sljedećem primjeru:

Zaslon 3.30

```
>>> while True:
...     try:
...         x = int(raw_input("Unesite broj: "))
...         break
...     except ValueError:
...         print "Oops! To nije dobar broj. Probajte ponovo..."
```

Pri tomu, naporedba `try` radi po sljedećem načelu:

- ako se tokom izvršavanja ne pojavi iznimka, `except` linija se preskače i `try` završava
- ako se iznimka pojavi tijekom izvršavanja `try`-a, i ako odgovara iznimci navedenoj u `except` liniji, onda se ona izvršava
- ako se iznimka pojavi, a ne odgovara `except` liniji, provjeravaju se druge iznimke u `try` izrazu. Ako njih nema, program završava i pokazuje poruku koja je gore navedena.

Pri tomu `try` može imati više od jedne `except` linije (uvjeta), koji dohvaćaju neke druge iznimke. Pri tome se najviše izvršava jedna od njih, i to samo za iznimke navedene u tom djelu `except` linije, ali ne za druge. `Except` linija može imati više iznimki, definiranih unutar n-terca, npr:

```
... except (RuntimeError, TypeError, NameError):
... pass
```

`Try...except` izraz može sadržavati naredbu `else` koja se izvršava ako u `try` bloku nije dohvaćena niti jedna iznimka. Taj se način koristi umjesto pisanja dodatnih linija u `try`-u, jer se tako izbjegava mogućnost uporabe iznimke koja nije navedena u `try` bloku.

Ako iznimka ima argument, on se ispisuje kao zadnji dio poruke o pogrešci('detail'). Programi za rad s iznimkama mogu raditi i s funkcijama, koje se unutar njih pozivaju.

Zaslon 3.31

```
>>> def nemoguce_dijeljenje():
...     x = 1/0
...
>>> try:
...     nemoguce_dijeljenje()
... except ZeroDivisionError, detail:
...     print 'Run time pogreska u dijeljenju:', detail
...
Run time pogreska u dijeljenju: integer division or modulo by zero
```

Ovo je *Run time* pogreska u dijeljenju: `integer division or modulo by zero`.

Obradba iznimki

Kada Python podigne iznimku, ona se mora dohvatiti (uhvatiti) i obraditi, inače će Python nasilno završiti izvođenje programa. Obradba iznimki prisiljava programera na razmišljanje o tome što može poći krivo unutar programa i što se može učiniti u vezi toga. Nije praktično (i skoro je nemoguće) imati na umu sve što može poći krivo. Umjesto toga treba tražiti stanja koja se (govoreći općenito) mogu ponovno vratiti, koristeći korekcijski blok kôda, koji se zove 'obrada

iznimke' (engl. *exception handler*). Taj kôd može automatski popraviti stanje bez korisničke interakcije, upitati korisnika za dodatne ili prihvatljive podatke za popravak problema, ili u nekim okolnostima zatvoriti program sa `sys.exit()`. (Dakako, cijeli je smisao iznimki u tome da se takve fatalne situacije izbjegnu). Pythonova naredba `try` detektira i obrađuje iznimke. Ona je višeslojna naredba kao `if` ili `while`, te slijedi pravila uvlake. `Try` također mijenja uobičajeno upravljanje tijeka programa. (Upravljanje tokom programa je izvršni niz naredbi koje čine program).

Definicija 3.4 — try...except.

```
try:
   try_block
except ex:
   except_block
```

Python pokreće kôd u bloku `try` dok ga ne završi uspješno ili se pojavi iznimka. Ako završi uspješno, Python preskače blok `except`, te se izvršenje nastavlja na naredbi koja slijedi iza njega. Ako se pojavi iznimka, Python preskače ostatak naredbi bloka `try` i ispituje vrijednost `ex` u odlomku `except` kako bi se ustanovilo je li vrsta podignute iznimke odgovara vrsti koja je opisana sa `ex`.

Ako iznimka odgovara `ex`, Python pokreće *blok except*, te se upravljanje tokom programa prebacuje na naredbu koja slijedi blok `except`. Ako iznimka ne odgovara `ex`, iznimka se proširuje na bilo koje uključene `try` naredbe, koje bi mogle imati odlomak *except* u kojem bi se iznimka mogla obraditi. Ako ni jedna od obližnjih *except klauzula* ne obrađuje podignutu iznimku, onda Python zatvara program te ispisuje poruku o pogrešci.

Ignoriranje iznimke

Za ignoriranje iznimke koristi se naredba `pass` unutar odlomka `except`.

Definicija 3.5 — except pass.

```
try:
  try_block
except ex:
  pass
```

Pronalaženje argumenta iznimke

Kada se dogod iznimka, ona može imati i pridruženu vrijednost također poznatu i kao *argument iznimke*. Vrijednost i tip argumenta ovise o tipu iznimke. Kod većine iznimki, argument je n-terac sastavljen od jednog string člana koji upućuje na uzrok pogreške. Kod pogrešaka operacijskog sustava tipa `IOError`, argument sadrži i dodatne atribute, kao što su broj pogreške ili ime datoteke. U tom slučaju može se koristiti funkcija `dir()` da bi se ispisao popis atributnih argumenta. Za pronalaženje vrijednosti argumenta, treba se specificirati varijabla nakon imena iznimke u odlomku `except`.

Zaslon 3.32

```
try:
  try_block
except ex, target:
  except_block
```

pa ako se dogodi iznimka, Python će pridružiti argument iznimke varijabli `target`.

Obradba svih iznimki

Da bi se obuhvatilo sve iznimke, treba se specificirati odlomak `except` bez imena iznimke ili argumenta. Odlomak `except` koji na ovaj način obuhvaća sve iznimke je općenito slabo rješenje, jer obuhvaća sve iznimke, a ne samo one koje su bitne, a osim toga, može prikriti stvarne pogreške korisničkog programa.

Definicija 3.6 — except block.

```
try:
  try_block
except:
  except_block
```

gdje `except_block` prihvaća i obrađuje sve iznimke.

Moguće je koristiti jedan `except` odlomak da bi se obuhvatilo više tipova iznimki ili se može koristiti više except odlomaka kojima se obrađuju pojedine iznimke.

Definicija 3.7 — višestruki except.

```
try:
  try_block
except  (ex1, ex2, ...) [, target]:
  except_block
```

Odlomak `except` obuhvaća bilo koju od popisanih iznimki `(ex1, ex2, ...)` te pokreće isti `except_block` za sve njih. `Target` je dopustiva varijabla koja uzima argument iznimke.

3.5.3 Pokretanje obaveznog kôda

Naredba `try-finally` može se koristiti za pokretanje programskog kôda bez obzira je li Python podigao iznimku ili nije. Za razliku od odlomka `except` , odlomak `finally` ne dohvaća iznimke, nego definira akcije čišćenja koje se moraju obaviti u svakom slučaju, bez obzira na postojanje pogreške. Odlomci `finally` i `except` ne mogu se pojaviti zajedno u istom `try` odlomku. Programeri obično postave naredbu `try-finally` unutar naredbe `try-except` kako bi obradili iznimke koje su podignute u naredbi `try-finally`. Isto tako, odlomak `else` ne može se dodati uz naredbu `try-finally`.

Definicija 3.8 — finally block.

```
try:
  try_block
finally:
  finally_block
```

Python pokreće kôd unutar `finally_block-a` nakon što ga pokrene u `try_block-u`. Python će pokrenuti `finally_block` bez obzira kako se `try_block` izvrši: bilo normalno, bilo putem iznimke ili putem naredbe tipa `break` ili `return`. Ako se iznimka dogodi unutar `try_block-a`, Python će preskočiti ostatak `try_block-a`, te pokrenuti `finally_block`, a onda ponovno podignuti iznimku kako bi se dohvatila naredbom `try-except` više razine.

Zaslon 3.33 — Try-finally.

```
>>> import time   #ucitavanje modula koji sadrzi funkcije time.sleep()
>>> try:
...     p=7-15/0
...     print p
```

```
... finally:
...     print u"Bez obzira na sve 73*55 će biti izračunato..."
...     time.sleep(3)   # pauza 3 sekunde
...     print u"Gotovo! Rješenje {1} je:{0}".format(73*55,"73*55")
...
Bez obzira na sve 73*55 će biti izračunato...
Gotovo! Rješenje 73*55 je:4015
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ZeroDivisionError: division by zero
>>>
```

4 — Funkcije

Funkcije i metode služe za skupljanje naredbi u cjeline koje se često izvode, ali najčešće s drugim vrijednostima podataka. Razlikuje se *definicija* i *poziv* funkcije. U definiciji, kako samo ime govori, stvara se *ime funkcije*, *programski kôd* koji se izvodi i definiraju *tipovi ulaznih podataka*, te podaci koji će se iz funkcije *vratiti* (tzv. *izlazni podaci*). Kod poziva funkcije na mjesto *formalnih*, *ulaznih* podataka dolaze *stvarni* podaci, a iz funkcije se vraćaju *rezultati*, izlazni podaci. Ako se funkcija pridruži nekoj varijabli, onda se nakon izvođenja funkcije u toj varijabli nalaze vraćeni izlazni podaci.

4.1 Definicija funkcije

Funkcija se definira s pomoću ključne riječi `def` sa sljedećom sintaksom:

Definicija 4.1 — Funkcija.

```
def ime_funkcije(par1, par2, ...):
       """ komentar koji se koristi kod poziva help(ime_funkcije) """
       tijelo funkcije (naredbe)
       .....   # blokovi naredbi s uvlakama
       return  objekt   # funkcija može vratiti rezultat bilo kojeg tipa
```

`ime_funkcije` je identifikator, varijabla koja se povezuje (ili re-povezuje) s funkcijskim objektom prilikom izvršenja naredbe `def`. Unutar zagrada iza imena funkcije nalaze se `parameteri` kao dopuštena, slobodna lista identifikatora, ulaznih varijabli koji se još zovu *formalni parametri*, a koji se koriste kod poziva funkcija za pridruživanje stvarnim vrijednostima koje zovemo *argumenti*. U najjednostavnijem slučaju, funkcija nema nikakvih formalnih parametara, što znači da kod poziva funkcija ne uzima nikakve argumente. U ovom slučaju, definicija funkcije ima prazne zagrade koje slijede iza imena funkcije tj. `funkcijsko_ime()`.

Ako funkcija treba obraditi ulazne podatke, onda ona ima argumente, pa u definiciji funkcije postoje *parametri* koji sadrže jedan ili više identfikatora, varijabli, odvojenih zarezima. U ovom slučaju, svaki poziv funkcije pridružuje stvarne vrijednosti, *argumente*, s ovim parametrima (*formalnim argumentima*) specificiranima u definiciji funkcije. *Parametri* su *lokalne varijable* funkcije, pa svaki poziv funkcije povezuje te varijable s odgovarajućim stvarnim vrijednostima koje pozivatelj u funkciji navodi kao argumente.

Niz naredbi koji slijedi i nije prazan, poznatiji kao *tijelo funkcije* (engl. *function body*), kao ni naredba `def`, ne izvršava se prije poziva funkcije. Funkcija se može učitati u memoriju, npr. s *import* naredbom, ali se izvršava tek kad se funkcija pozove.

Tijelo funkcije može imati jednu ili više `return` naredbi koje vraćaju rezultat, ali može postojati i bez nje (u tom slučaju funkcija obično ispisuje rezultat, ali nikakav rezultat ne vraća u programski odsječak iz kojeg je pozvana).

Evo primjera jednostavne funkcije koja vraća obrnutu vrijednost stringa od one koju primi:

Funkcija 4.1 — Def i return.

```
def obrni(x):
    return x[::-1]
```

što kao rezultat za nekoliko poziva daje:

Zaslon 4.1

```
>>> obrni(u'svakidašnja jadikovka')
u'akvokidaj ajn\u0161adikavs'
>>> obrni(u'Tin')
u'niT'
>>> print obrni(u'svakidašnja jadikovka')
akvokidaj ajnšadikavs
>>> var=obrni(u'Ujević')
>>> print var
ćivejU
>>>
```

Parametar *x* (zvan još *formalni argument* ili *lokalna varijabla*) funkcije *obrni()*, povezuje se kod svakog poziva funkcije s argumentom (zvanim još *stvarni argument*). Dok je formalni argument samo jedan, definiran kad i funkcija, stvarni argument može biti (i obično jest!) različit kod svakog novog poziva. Tako je u gornjem primjeru formalni argument *x*, poprimao čak tri stvarne vrijednosti argumenata u različitim okolnostima (pozivima u interaktivnom radu, uz *print* naredbu i u pridružbi). Argumenti su bili: '*svakidašnja jadikovka*','*Tin*' i '*Ujević*' nad kojima je funkcija imala obradbu - okretala je string unatrag i preko return naredbe vraćala u naredbu poziva (interaktivno, print i pridružba). Ime argumenta je ime varijable, pa samim tim ne govori o tipu stvarnog argumenta koji se u funkciju prenosi. Unutar tijela funkcije mora se načiniti prikladna obradba koja će voditi računa i o tipu varijable.

Funkcija 4.2 — tipovi ulaznih podataka.

```
def spoji(a,b):
    if type(a) in [int,float] and type(b) in [int,float]:
        print 'Radim s brojevima'
    elif type(a)==str and type(b)==str:
        print 'Povezujem stringove!'
    else:
        print 'Moram vas pretvoriti u string!'
        a=str(a); b=str(b)
    return a+b
```

što daje, na primjer:

Zaslon 4.2

```
>>> spoji('Ivan',' Gundulić')
Povezujem stringove!
'Ivan Gunduli\xc4\x87'
>>> x=spoji('Suze sina ', 'razmetnoga')
Povezujem stringove!
>>> print x
Suze sina razmetnoga
>>> y=spoji('32',1984)
Moram vas pretvoriti u string!
>>> y=spoji(32,1984)
Radim s brojevima
>>> print spoji('Orwell ',1984)
Moram vas pretvoriti u string!
Orwell 1984
>>>
>>>
```

4.2 Poziv funkcije

Poziv funkciji je izraz sa sljedećom sintaksom:

```
funkcijsko_ime(argumenti)
```

`funkcijsko_ime` može biti bilo kakva referenca na objekt funkcije, što je najčešće samo ime dotične funkcije. Zagrade označuju operaciju poziva funkcije, dok su `argumenti` u najjednostavnijem slučaju niz od nula ili više izraza odvojenih zarezima, koji daju, prenose, vrijednosti odgovarajućim parametrima definirane funkcije. Kada se funkcija poziva, parametri poziva funkcije povezuju se s funkcijskim argumentima, tijelo funkcije se izvršava sa stvarnim vrijednostima argumenata, a funkcija nakon izvršenja na koncu vraća pozivatelju neku vrijednost izraza.

4.3 Funkcijski parametri

Funkcijski parametri mogu biti:

- - *pozicijski* (a,b,c,..), kad je važno njihovo mjesto u redoslijedu
- - *slijedni parametar* (*x), kad nije važan njihov broj
- - *imenovani parametri*(**r), kad se želi imati pretpostavljene vrijednosti

Funkcija može biti bez parametara, a ako ih ima, onda može imati bilo koji broj parametara bilo koje skupine. Pritom je važno zapamtiti samo da pozicijski parametri moraju biti *ispred* slobodnih i imenovanih.

4.3.1 Pozicijski parametri

Formalni parametri koji su jednostavni identifikatori navedeni u definiciji funkcije predstavljaju obvezatne parametre, što znači da svaki poziv funkcije mora pružiti i odgovarajuću vrijednost - (*argument*) za svaki parametar.

Funkcija 4.3 — Pozicijski parametri.

```
def mi(a,b,c):
    print u'ja sam a i značim:',a
    print u'ja sam b i značim:',b
```

```
    print u'ja sam c i značim:',c
```

što daje, na primjer:

Zaslon 4.3

```
>>> mi(3,u'praščića',[1,2,3])
ja sam a i značim: 3
ja sam b i značim: praščića
ja sam c i značim: [1, 2, 3]
>>>
>>> mi('k' in 'miki',5+5,3*'jupi ')
ja sam a i značim: True
ja sam b i značim: 10
ja sam c i značim: jupi jupi jupi
>>>
>>> mi('prvi parametar','drugi')
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
TypeError: mi() takes exactly 3 arguments (2 given)
>>>
```

Primjetite kako je moguće na mjestu parametra u pozivu funkcije da stvarni argument bude bilo koji izraz, složeni tip podatka (lista, rječnik i sl.), čak i funkcija (što će biti posebno pokazano).

4.3.2 Slijedni parametri

Nakon pozicijskih parametara može slijediti slijedni parametar, koji omogućuje slijed ili niz stvarnih argumenata, kojima unaprijed nije specificiran broj.

Funkcija 4.4 — Slijedni parametar.

```
def zbroji(*broj):
    rez = 0
    for x in broj: rez += x
    return rez

def povezi(*rijec):
    rez = ""
    for x in rijec: rez += x
    return rez
```

što daje, na primjer:

Zaslon 4.4

```
>>> zbroji(1.1,2.22,3.333,4.4444)
11.0974
>>> zbroji(5,4,1,1,2,0,-2,-3,5)
13
>>> povezi('more ','zlatno')
'more zlatno'
>>> print povezi('a','b','c',u'č',u'ć',u'đ')
```

```
abcčćđ
>>>
```

Za pozivanje funkcije koja zahtijeva niz članova može se koristiti i sintaksa 'otpakiravanja' argumenata:

Zaslon 4.5

```
>>> b=[1,2,8,8,8]
>>> print zbroji(*b)   # *b otpakirava listu b
27
>>> a=(1,2,7,7,8)
>>> print zbroji(*a)   # *a otpakirava n-terac a
25
>>> print zbroji(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in zbroji
TypeError: unsupported operand type(s) for +=: 'int' and 'tuple'
>>>
```

Primjetite kako su funkcije za povezivanje brojeva zbrajanjem i stringova nadovezivanjem identične. Razlika je samo u početnom uvjetu numeričke ništice i praznog stringa (*rez=0* i *rez=""*). Razmislite kako biste načinili program koji bi ovisno o tipu argumenata vraćao željeni rezultat.

4.3.3 Imenovani parametri

Nakon pozicijskih, pa slijednih parametara u definiciju funkcije mogu se ugraditi i imenovani (eng. keywords) parametri zadani parovima (*param=vrijednost*) ili generičkim (***parms*) slobodnim parovima, koji se kod poziva funkcije nadomještaju sa stvarnim parovima (*arg=vrijednost*).

Funkcija 4.5 — Imenovani parametri.

```
def pretpostavi(a, gr='glagol',sem='agens'):
    return a, gr, sem

def free(**x):
    print u'članovi:', x.items()
    print u'ključevi i vrijednosti', x.keys(), x.values()
```

što daje, na primjer:

Zaslon 4.6

```
>>> pretpostavi('biti',sem='stanje')
('biti', 'glagol', 'stanje')
>>> print pretpostavi('učiti')
('u\xc4\x8diti', 'glagol', 'agens')
>>> print pretpostavi(u'učiti')
(u'u\u010diti', 'glagol', 'agens')
>>> for i in pretpostavi(u'učiti'):
...     print i
...
učiti
```

```
glagol
agens
>>>
>>> free(glagol='orati', imenica='polje')
članovi: [('glagol', 'orati'), ('imenica', 'polje')]
ključevi i vrijednosti ['glagol', 'imenica'] ['orati', 'polje']
```

Rječnik se može otpakirati pri pozivu funkcije preko imenovanih argumenata:

Zaslon 4.7

```
>>> V={'glagol':'orati', 'imenica':'polje'}
>>> free(**V) # **V otpakirava rjecnik V,
ključevi i vrijednosti ['glagol', 'imenica'] ['orati', 'polje']
članovi: [('glagol', 'orati'), ('imenica', 'polje')]
```

Kao što se moglo primijetiti, Python za slijedne parametre koristi listu, a za imenovane parametre rječnik. To se još zornije može pokazati sljedećim primjerom u kojem su dane sve vrste parametara:

Funkcija 4.6 — Generički parametri.

```
def generic(x,y,*args, **kwargs):
    print x,y
    print args
    print kwargs

generic(1, 'izbor', 3, 4, 'sad', ime='izidor', \
        pjesma=u"Ja živim u kruzima koji se šire")
```

što daje, na primjer:

Zaslon 4.8

```
1 izbor
(3, 4, 'sad')
{'ime': 'izidor', 'pjesma': u'Ja živim u kruzima koji se šire'}
>>>
```

Na kraju formalnih pozicijskih parametara u definiciji funkcije, dopušteno je koristiti jedan ili oba posebna oblika: `*identifikator1` i `**identifikator2`. Ako su obadva na raspolaganju, onaj s dvije zvjezdice mora doći zadnji. Parametar `*identifikator1` omogućuje da poziv funkcije može osigurati dodatne pozicijske argumente, dok `**identifikator2` omogućuje da poziv funkcije može osigurati dodatne argumente s imenom. Na taj način rješavaju se pozivi funkcija s promjenljivim, varijabilnim brojem argumenata. Svaki poziv funkcije povezuje `identifikator1` s n-tercem čiji su članovi dodatni pozicijski argumenti (ili praznim n-tercem, ako ih nema). `Identifikator2` povezuje se s rječnikom čiji su članovi imena i vrijednosti dodatnih argumenata koji se imenuju (ili praznim rječnikom, ako ih nema).

Treba primijetiti da se isti objekt, pretpostavljena vrijednost, povezuje sa slobodnim parametrom kad god u pozivu funkcije nema pridruženog argumenta.

Ako je pretpostavljena vrijednost lista, onda se ona može mijenjati nakon svakog poziva funkcije:

Zaslon 4.9 — Promjenjiva pretpostavljena vrijednost.

```
>>> def f(x,y=[]):
...     y.append(x) # mijenja pretpostavljenu vrijednost od y
...                 # nakon svakog poziva f s jednim argumentom
...     return y
...
>>> print f(23)
[23]
>>> print f(42)
[23, 42]
>>> print f(3,[4]) # pretp. vrijednost od y ostaje i dalje [23, 42]!
[4, 3]
>>> print f(5)
[23, 42, 5]
>>>
```

4.4 Funkcija je objekt

Budući da su Python funkcije objekti, moguće je s njima pisati opće programe, u kojima se one pridružuju varijablama, vraćaju kao rezultat iz neke druge funkcije, ugrađuju u strukture podataka, prosljeđuju kao funkcijski argumenti i slično.

Zaslon 4.10 — Funkcija kao varijabla.

```
>>>def jutro(ime):  # ispiši pozdrav
...    print'Dobro jutro '+ ime + '!'
>>> jutro('Ivanka')
Dobro jutro Ivanka!
>>>
>>> x=jutro     # pridružba funkcije varijabli
>>> x('Ivana')  # poziv preko varijable
Dobro jutro Ivana!
>>>
```

Gornji primjer postaje zanimljiviji, kad se varijabla *x* prenese kao argument neke druge funkcije, čiji je formalni parametar funkcija. Uz već poznatu funkciju *jutro()* i varijablu *x*, novi primjer to pokazuje:

Zaslon 4.11 — Funkcija kao argument.

```
>>> def F_kao_arg(func, arg):
        func(arg)
>>>
>>> F_kao_arg(jutro, 'Matea')
Dobro jutro Matea!
>>> F_kao_arg(x, 'Matea')
Dobro jutro Matea!
>>>
```

Funkcije koje su samostalne i nemaju funkcijske argumente zovu se *funkcije prve vrste*, dok složene funkcije s funkcija kao argumentima zovemo *funkcije druge vrste*. U gornjem primjeru *jutro()* je funkcija *prve*, a *F_kao_arg()* funkcija *druge vrste*.

Gore opisano može se povezati u kratak program koji uključuje funkcije u strukturu podatka, npr. listu:

Program 4.1 Program ugradjuje imena dviju funkcija u listu.

```
1  def jutro(ime):  # ispiši pozdrav
2      print 'Dobro jutro '+ ime + '!'
3
4  def dan(ime):  # ispiši pozdrav
5      print 'Dobar dan '+ ime + '!'
6
7
8  pozdravi = [ (dan, 'Iva :)'), (jutro, 'Ivana :-)'),(jutro, '
       Ivanka ;-)'), (dan, 'Matea :D') ]
9  for (poz, koga) in pozdravi:
10     poz(koga)  # Poziv funkcije uključene u listu
```

što kao rezultat daje:

Zaslon 4.12 — Funkcija u strukturi podatka.

```
Dobar dan Iva :)!
Dobro jutro Ivana :-)!
Dobro jutro Ivanka ;-)!
Dobar dan Matea :D!
>>>
```

4.5 Funkcijski prostor imena

Formalni parametri funkcije i sve varijable u funkcijskom tijelu, tvore lokalni prostor *imena funkcije* (eng. *local namespace*), također poznat i kao *lokalni doseg* (eng. *local scope*). Svaka od ovih varijabli zove se *lokalna varijabla funkcije*. Lokalna varijabla ne može se dohvatiti (s njom se ne može raditi), izvan tijela funkcije.

Program 4.2 Program ističe doseg varijabli.

```
1  def doseg(ime):
2      pjesma='NOSIM SVE TORBE A'
3      ime = ime + u" Tadijanović"
4      mjesto='Rastušje'
5      print 'ime iz funkcije =', ime
6      print 'pjesma iz funkcije =', pjesma
7      return ime
8
9  ime='Dragutin'
10 pjesma='NISAM MAGARAC'
11 doseg(ime)
12 print 'ime iz glavnog:',ime
13 print 'pjesma iz glavnog:',pjesma
14 print 'mjesto iz glavnog:',mjesto
```

što kao rezultat daje:

Zaslon 4.13 — Lokalne varijable.

```
ime iz funkcije =  Dragutin Tadijanović
pjesma iz funkcije =  NOSIM SVE TORBE A
ime iz glavnog: Dragutin
pjesma iz glavnog: NISAM MAGARAC
mjesto iz glavnog:Traceback (most recent call last):
  File "<module1>", line 14, in <module>
NameError: name 'mjesto' is not defined
>>>
```

Moguće je u tijelu funkcije imati varijablu koja se može dohvatiti izvan funkcije, samo u tom slučaju ispred njenog imena mora pisati oznaka `global`. Takve varijable zovu se *globalne varijable*. Ako lokalna varijabla unutar funkcije ima isto ime kao i globalna varijabla, kada god se to ime spomene u tijelu funkcije, koristi se lokalna, a ne globalna varijabla. Za ovakav pristup kaže se da lokalna varijabla *sakriva* globalnu varijablu istog imena kroz cijelo tijelo funkcije.

Program 4.3 Program ističe razliku globalnih i lokalnih varijabli.

```
1  def doseg(ime):
2      global pjesma
3      pjesma='NOSIM SVE TORBE A'
4      ime = ime + u" Tadijanović"
5      global mjesto
6      mjesto='Rastušje'
7      print 'ime iz funkcije =', ime
8      print 'pjesma iz funkcije =', pjesma
9      return ime
10
11 ime='Dragutin'
12 pjesma='NISAM MAGARAC'
13 doseg(ime)
14 print 'ime iz glavnog:',ime
15 print 'pjesma iz glavnog:',pjesma
16 print 'mjesto iz glavnog:',mjesto
```

što kao rezultat daje:

Zaslon 4.14 — Lokalne i globalne varijable.

```
ime iz funkcije =  Dragutin Tadijanović
pjesma iz funkcije =  NOSIM SVE TORBE A
ime iz glavnog: Dragutin
pjesma iz glavnog: NOSIM SVE TORBE A
mjesto iz glavnog: Rastušje
>>>
```

Uz pomoć globalne varijable moguće je prenositi podatke u funkciju i bez ulaznih parametara, ali takav pristup nije dobar, jer programer mora pamtiti sve definirane globalne varijable i ne dopustiti da ih neka lokalna (s istim imenom) sakrije. Zato je preporuka dobrog programiranja sve podatke iz vanjskog programa prenositi u funkciju preko njenih ulaznih parametara.

Varijable definirane izvan funkcije mogu se koristiti i unutar nje:

Zaslon 4.15 — Pristupačnost varijabli izvan funkcije.

```
>>> a=1
>>> def F(b):
...     return a+b
...
>>> print F(3)
4
>>> a=2
>>> print F(3) # koristi se nova vrijednost od a
5
>>> a=1        # a je vracen na 1
>>> def M(b):
...     a=2    # stvaranje nove lokalne varijable a
...     return a+b
...
>>> print M(2)
4
>>> print a    # globalni a nije promijenjen
1
>>>
```

4.6 Ugnježđene funkcije

Naredba `def` unutar tijela funkcije definira *ugnježđenu funkciju* (eng. *nested function*), a funkcija čije tijelo uključuje `def` je poznata kao *vanjska funkcija* (eng. *outer function*) s obzirom na ugnježđenu. Kôd unutar tijela ugnježđene funkcije može pristupiti (ali ne i re-povezati) lokalne varijable vanjske funkcije, također poznate i kao *slobodne varijable ugnježđene funkcije*.

Najjednostavniji način da se dopusti ugnježđenoj funkciji pristup vanjskim vrijednostima nije oslanjati se na ugnježđene dosege varijabli ili upotreba *global* naredbe, nego prijenos vanjskih vrijednosti preko funkcijskih parametara/argumenata ugnježđene funkcije. Vrijednost argumenta može se povezati kada se ugnježđena funkcija definira s pretpostavljenom vrijednosti za izabrani argument. Na primjer:

Funkcija 4.7 — Ugnježđena funkcija-prijenos preko argumenata.

```
def postotak_1(a, b, c):
    def pc(x, ukupno=a+b+c): return (x*100.0) / ukupno
    print "Postotci su: ", pc(a), pc(b), pc(c)
```

Isti program s upotrebom ugnježđenih dosega izgledao bi ovako:

Funkcija 4.8 — Ugnježđena funkcija-prijenos preko vanjskih lokalnih varijabli.

```
def postotak_2(a, b, c):
    def pc(x): return (x*100.0) / (a+b+c)
    print "Postotci su: ", pc(a), pc(b), pc(c)
```

U ovom slučaju, `postotak_1` ima malu prednost: izračun `a+b+c` događa se samo jednom, dok kod `postotak_2` nutarnja funkcija `pc` ponavlja izračun tri puta. Međutim, ako vanjska funkcija re-povezuje svoje lokalne varijable između poziva ugnježđene funkcije, ponavljanje

izračuna bi mogla biti i prednost. To znači da se preporučuje znanje oba pristupa, te izbor najprikladnijeg u danoj situaciji.

Ugnježđena funkcija koja pristupa vrijednostima s vanjskih lokalnih varijabla poznata kao je zatvorena funkcija ili *closure*. Sljedeći primjer pokazuje kako izgraditi *closure* bez ugnježđenih dosega (koristeći pretpostavljenu vrijednost):

Funkcija 4.9
```
def make_adder_1(augend):
    def add(addend, _augend=augend): return addend+_augend
    return add
```

Ista funkcija korištenjem ugnježđenih dosega izgleda ovako:

Funkcija 4.10
```
def make_adder_2(augend):
    def add(addend): return addend+augend
    return add
```

Closures su iznimke u općem pravilu, jer su objektno-orjentirani mehanizmi najbolji način povezivanja podataka i kôda zajedno. Kada se trebaju konstruirati pozivni objekti, s nekim parametrima u vremenu konstrukcije objekta, *closures* mogu biti efikasnije i jednostavnije od klasa. Na primjer, `rezultat make_adder_1(7)` je funkcija koja prihvaća jedan argument i dodaje 7 na taj argument (rezultat `make_adder_2(7)` se ponaša na isti način). *Closure* je "tvornica" za bilo kojeg člana obitelji funkcija koji se razlikuju po nekim parametrima, te može pomoći u izbjegavanju dvostrukog pisanja programskog kôda.

Primjer korištenja zatvorenih funkcija:

Zaslon 4.16 — Funkcija stvara funkciju.
```
>>> def F(x):
...     def G(y):
...         return x - y
...     return G
...
>>> D1 = F(11)     # D1 je nova funkcija, x=11
>>> D2 = F(12)     # D2 je nova funkcija, x=12
>>> D3 = F(13)     # D3 je nova funkcija, x=13
>>> print D1(7)
4
>>> print D2(7)
5
>>> print D3(7)
6
>>>
```

4.7 Rekurzivne funkcije

Funkcija može zvati i samu sebe. Pritom dakako, u njenom tijelu mora biti ugrađeno upravljanje koje će učiniti da funkcija završi. U sljedećem primjeru u funkciji je_li_Palindrom() ulazni string

se svakim pozivom smanjuje za prvo i zadnje slovo i pita jesu li oba jednaka. Ako nisu, proces se završava, te ne poziva ista funkcija još jednom. Ako pak jesu, onda se poziva ista funkcija, ali bez početnog i zadnjeg slova. Ako ostane samo jedno slovo ili niti jedno, iteracija završava i program vraća True - radi se o palindromu.

Funkcija 4.11 — Rekurzivna funkcija.

```
def je_li_Palindrom(s):
    ##print s
    if len(s)<=1:
        return True
    else:
        return s[0]==s[-1] and je_li_Palindrom(s[1:-1])
```

što kao rezultat daje:

Zaslon 4.17 — Poziv rekurzivne funkcije.

```
print je_li_Palindrom('ratar')
True
>>> print je_li_Palindrom('melem')
True
>>> print je_li_Palindrom('radost')
False
>>>
```

Palidrom je riječ (string) koji se jednako čita s lijeva i s desna. Priprava teksta ili rečenica za provjeru palindroma ide u smjeru eliminacije znakova interpunkcije, a potom pretvaranja velikih slova u mala.

Zaslon 4.18 — Priprava za provjeru palindroma.

```
niz='udovica, baci vodu'
z=''
for s in niz:
    if s not in [',',' ','.','!','?']:
        z+=s
>>>
>>> print je_li_Palindrom(niz)
False
>>> print je_li_Palindrom(z)
True
>>>
```

4.8 Anonimne funkcije

Anonimna jednostavna funkcija definirana je izrazom `lambda [arg]:[ret]` što znači da je to funkcija s argumentom `arg` koja vraća objekt `ret`.

Funkcija 4.12 — Anonimna funkcija.

```
>>> A = lambda B: B + 5
>>> A(7)
```

```
12
```

Kao što se vidi iz primjera, funkcija je anonimna sve dok se ne pridruži varijabli, kao u primjeru varijabli `A`. Onda je to isto kao sljedeća definicija funkcije:

Zaslon 4.19

```
>>> def A(B):
        return B+5
>>> A(7)
12
```

4.9 Funkcijsko dokumentiranje

Jedna od važnih pravila dobrog programiranja je dobro dokumentiranje. Python ima ugrađenu mogućnost da se svaka funkcija dokumentira i to tako da se ispod definicije funkcije, upiše željeni string kojim se funkcija opisuje. String obično ima trostruke navodnike na početku i koncu, kako bi se mogao protezati kroz više redaka, fizičkih linija. Taj dokumentacijski string (engl. *documentation string*), također je poznat kao *docstring* i može se koristiti kao `func_doc` ili `__doc__`. Naredbom

```
print ime_funkcije.__doc___
```

ispisat ćemo sadržaj tako definiranog dokumentracijskog stringa:

Funkcija 4.13 — Docstring.

```
def obrni(s):
    """obrće string: prvo slovo postaje zadnje, zadnje prvo,
       i tako rekurzivno. String čitan s lijeva nadesno
       postaje string čitan s desna nalijevo.

    Argumenti:
        s: zadani niz znakova koji se žele obrnuti.
    Povratak:
        vraća obrnuti string.
    Iznimke:
        TypeError: ako s nije string.

    """
    pass
```

što daje:

Zaslon 4.20

```
>>> print obrni.__doc__
obrće string: prvo slovo postaje zadnje, zadnje prvo,
       i tako rekurzivno. String čitan s lijeva nadesno
       postaje string čitan s desna nalijevo.

    Argumenti:
        s: zadani niz znakova koji se žele obrnuti.
```

```
    Povratak:
        vraća obrnuti string.
    Iznimke:
        TypeError: ako s nije string.

>>>
```

Dokumentacijski nizovi važan su sadržaj svakoga Python kôda. Njihova je uloga slična komentarima u bilo kojem programskom jeziku, ali im je iskoristivost šira, jer su dostupni pri procesu pokretanja programa (engl. *runtime*). Programerska okruženja i drugi alati mogu koristiti *docstring* da podsjete programera kako koristiti te objekte. To znači da djeluju kao *pomoć* (engl. *help*) programa. Da bi se *docstring* učinio što korisnijim, trebalo bi poštovati nekoliko jednostavnih dogovora. Prva linija *docstringa* trebala bi biti sažeti opis svrhe funkcije, a počinjala bi s velikim slovom i završavala točkom. Unutar nje se ne bi trebalo spominjati ime funkcije, osim ako ime nije riječ iz prirodnog, govornog jezika pa dolazi kao dio opisa operacije te funkcije. Ako je *docstring* protegnut preko više linija, druga linija trebala bi biti prazna, a sljedeće linije bi trebale formirati jedan ili više odlomaka, odvojenih praznim linijama, koji opisuju očekivane argumente funkcije, potrebne uvjete, izlazne (povratne) vrijednosti, te moguće nuspojave. Daljnja objašnjenja, književne reference, te primjeri korištenja (koji se mogu provjeriti sa `doctest`) mogu po slobodnom izboru slijediti iza, prema završetku *docstring*-a.