ODJEL ZA INFORMATIKU
SVEUČILIŠTE U RIJECI

Vedran Strčić

PYTHON 3.2

Skripta - Prvi dio

13. lipnja 2011.

Sadržaj

Python PyDev - Instalacija i početak rada

Opisana je instalacija PyDev for Eclipse. Python interpreter potrebno je najprije zasebno skinuti i instalirati, zatim se skine i raspakira jedan od standardnih Eclipse paketa, te se taj paket pokrene. Unutar standardnog Eclipse-a nalazi se izbornik koji aktivira instalaciju PyDev plugin-a. Slijedi opis cijelog postupka po koracima.

Instalacija "Python" interpretera

1. logirati se na računalo kao administrator sustava
2. učitati web stranicu na adresi: http://www.python.org/download/
3. odabrati željenu verziju Python-a (trenutno Python 3.2)
4. pri dnu slijedeće stranice nalazi se "Download" sekcija
5. odabrati i skinuti installer za vašu verziju operativnog sustava
6. pokrenuti installer
7. odabrati "Install for all users" ili "Install just for me" i kliknuti "Next"
8. odabrati folder za instalaciju (ili ostaviti default postavke) i kliknuti "Next"
9. odabrati željene "install features" (po default odabrano je sve) i kliknuti "Next"
10. pričekati da se instalacija završi (dozvoliti instalaciju ukoliko se pojavi upit)
11. kliknuti "Finish"

Instalacija "PyDev for Eclipse"

1. skinuti osnovni Eclipse paket (Eclipse Classic ili Eclipse IDE for Java Developers) sa web adrese: http://www.eclipse.org/downloads/
2. otpakirati i pokrenuti Eclipse (definirati workspace folder)
3. pokrenuti izbornik: Help - Install New Software...
4. kliknuti na dugme "Add..." smješteno desno od polja "Work with:"
5. za "Name:" upisati "Python Web Location" (ili neko drugo ime po izboru)
6. za "Location:" upisati "http://pydev.org/updates" (bez navodnih znakova)
7. kliknuti na dugme "OK"
8. nakon povratka u "Available Software" prozor nakon nekoliko trenutaka (ovisno o brzini internet konekcije) u središnjem dijelu pojaviti će se veze "PyDev" i "PyDev Mylyn Integration"
9. kvačicom označiti kvadratić pored "PyDev" opcije, ne označavati "PyDev Mylyn Integration"
10. kliknuti "Next"
11. u "Install Details" prozoru biti će naveden "PyDev for Eclipse"
12. kliknuti "Next"
13. otvoriti će se "Review Licenses" prozor
14. odabrati "I accept the terms of the license agreement" opciju smještenu dolje desno
15. kliknuti nadogume "Finish" i pričekati da se instalacija završi
16. ukoliko se pojavi prozor sa upitom za instalaciju plugina, odabrati "Install All"
17. ukoliko se pojavi prozor sa upitom "Do you trust these certificates?" kliknuti na dugme "Select All" pa zatim na dugme "OK"
18. kada se instalacija završi pojaviti će se zahtjev za ponovnim pokretanjem Eclipse-a
19. odabrati "Yes"

Podešavanje "PyDev for Eclipse"

1. nakon što je prethodna instalacija završena odabrati izbornik "Window - Preferences"
2. u popisu sa lijeve strane prozora proširiti "Pydev" (kliknuti na trokutić sa lijeve strane)
3. unutar "Pydev" liste odabrati "Interpreter - Python"
4. u "Python Interpreters" prozoru kliknuti na dugme "New"
5. za "Interpreter Name:" upisati "Python 3.2" (odnosno verziju koja je instalirana)
6. za "Interpreter Executable:" odabrati "C:\Python32\python.exe" (odnosno verziju koja je instalirana u folderu gdje je instalirana)
7. kliknuti na dugme "OK"
8. u "Selection Needed" prozoru odabrati sve osim "PySrc" i "python32.zip" (odabir je već takav po default) te kliknuti "OK" te još jednom "OK" čime ćete izići iz preferences prozora
9. Python IDE je sada podešen i može se početi sa radom

Definiranje prvog "Python" projekta

1. nakon pokretanja Eclipse-a odabrati "Window - Open Perspective - Other..." izbornik
2. u "Open Perspective" prozoru odabrati "Pydev" i kliknuti na "OK" dugme
3. odabrati u izborniku "File - New - Pydev Project"
4. odabrati i unjeti "Project name:"
5. za "Project type" odabrati "Python"
6. za "Grammar Version" odabrati "3.0"
7. kliknuti na "Finish"
8. u glavnom prozoru sa lijeve strane dvostruko kliknuti na novostvoreni folder (ime projekta koje ste prethodno podesili) da biste vidjeli njegov sadržaj
9. desnim klikom kliknuti na "src" folder unutar projekta i odabrati "New - Pydev Module"
10. pod "Name" upisati željeno ime datoteke i kliknuti "Finish"
11. u sredini prozora sada je otvoren prostor gdje možemo započeti sa pisanjem koda
12. da bi prilikom pisanja teksta u kodu mogli biti korišteni i hrvatski znakovi potrebno je u popisu projekata sa lijeve strane prozora desnim klikom kliknuti na folder koji predstavlja naš projekt (ime projekta) te odabrati opciju "Properties", zatim pod "Resource - Text file encoding" odabrati "Other: UTF-8" i kliknuti na "OK" dugme
13. test (pokretanje prvog programa):
    - upišite u prostor za pisanje koda: print('Moj prvi program!')
    - snimite promjene u datoteci klikanjem na ikonu diskete ili pritiskom "CTRL+S"
    - pokrenite program klikanjem na zeleno "Play" dugme ili pritiskom "CTRL+F11" tipki
    - u prozoru koji se tada (prvi puta) pokaže odabrati "Python Run"
    - u konzoli pri dnu ekrana ispisati će se tekst "Moj prvi program!"

Ispis na ekran

Za ispis standardnog teksta na ekran koristimo naredbu print:

```
print ('Ovo je početak programa!')
```

Sav tekst koji se nalazi unutar zagrada i apostrofa, nakon pokretanja programa, biti će ispisan u konzoli. Moguće je ispisivati i konkretan rezultat matematičkih izračuna, te ga kombinirati sa konkretnim tekstom, u tom slučaju konkretne brojčane vrijednosti i matematičke operacije nećemo pisati unutar apostrofa:

```
print (2+2)
```

Ili u kombinaciji sa konkretnim tekstom:

```
print ('Zbrojimo li 2 i 2 dobiti ćemo',2+2)
```

Sve djelove koji sadrže konkretan tekst odvajati ćemo od djelova koji sadrže brojčane vrijednosti ili matematičke operacije zarezima:

```
print ('2 i 2 je',2+2,'a malo slozeniji izračun iznosi',(2+2)*3/(4.5-1))
```

Primjetimo, prilikom ispisa, program će sam ubaciti razmake među djelovima sa tekstom i djelovima koji sadrže rezultate matematičkih izračuna. Praznu liniju ispisujemo ovako:

```
print ()
```

Operacije

Slijedi popis osnovnih matematičkih operacija kojima se možemo služiti u Pythonu.

Operacija	Simbol	Primjer
Potenciranje ...	**	3 ** 2 = 9
Množenje ...	*	4 * 3 = 12
Djeljenje ...	/	9 / 3 = 3.0
Cjelobrojno Djeljenje ...	//	14 // 3 = 4
Ostatak od Cjelobrojnog Djeljenja (Modulo) ...	%	14 % 3 = 2
Zbrajanje ...	+	5 + 2 = 7
Oduzimanje ...	-	7 - 3 = 4

Prioritet prilikom matematičkih izračuna slijedi matematička pravila:

- zagrade ( )
- potenciranje **
- množenje *, djeljenje /, cjelobrojno djeljenje //, i modulo %
- zbrajanje + i oduzimanje -

Komentari

Kada je potrebno umetnuti neku napomenu u programski kod koriste se komentari. Komentari su djelovi teksta unutar programskog koda koji se neće izvršavati. Jednu liniju testa označiti ćemo kao komentar tako da ispred te linije napišemo simbol '#':

```
# Ova linija je komentar, a u idućoj liniji je standardna naredba.

print ('Zbrojimo li 2 i 2 dobiti ćemo',2+2)
```

Kada je potrebno više linija označiti kao komentar, bilo to iz razloga što imamo malo veći opis, ili privremeno ne želimo da se dio našeg programa ne izvršava, umjesto da ispred svake linije upisujemo simbol '#' komentar možemo jednostavnije definirati tako da ispred prve i iza zadnje linije onog djela teksta koji želimo komentirati upišemo trostruke apostrofe:

```
'''
Sve linije teksta,
smještene između trostrukih apostrofa,
smatraju se komentarom.
'''
```

Korisnički ulaz i varijable

Da bi naši programi bili interaktivni, moramo na neki način omogućiti korisnicima unos podataka. Podatci koje korisnici unose pohranjuju se u varijable u programu, a unos je omogućen korištenjem funkcije input:

```
print ('Kako se zoveš?')
ime = input()
print ('Drago mi je',ime,'ja sam PyDev.')
```

Sadržaj koji korisnik, nakon prethodno postavljenog pitanja, unese pohraniti će se u varijablu ime. Na taj način korisnik će unos vršiti u konzoli u idućoj liniji nakon pitanja. Ukoliko želimo da korisnik svoj unos upiše u istoj liniji gdje je postavljeno pitanje napisati ćemo:

```
ime = input('Kako se zoveš? ')
print ('Drago mi je',ime,'ja sam PyDev.')
```

Treba naglasiti da će na ovaj način program sav korisnički unos registrirati kao tekst, odnosno string.

Pogledajmo slijedeći primjer:

```
a = input('Unesi prvi broj: ')
b = input('Unesi drugi broj: ')
print ('Ako zbrojimo',a,'i',b,'dobiti ćemo',a+b)
```

Ukoliko je korisnik ovdje unio brojeve 2 i 3, kao rezultat nebi dobio 5 već 23. To je zato što je program njegov unos prepoznao kao tekstualni unos, odnosno string, te nije zbrajao brojeve, nego je jednostavno spojio unesene znakove, odnosno riječi.

Da bi smo natjerali program da taj korisnički unos prepozna kao brojeve napisati ćemo:

```
a = int(input('Unesi prvi broj: '))
b = int(input('Unesi drugi broj: '))
print ('Ako zbrojimo',a,'i',b,'dobiti ćemo',a+b)
```

Na ovaj način striktno smo definirali da će ono što korisnik unese biti cjelobrojna vrijednost, odnosno int. Unos možemo definirati i kao bilo koju drugu vrijednost, na primjer decimalnu:

```
a = float(input('Unesi prvi broj: '))
b = float(input('Unesi drugi broj: '))
print ('Ako zbrojimo',a,'i',b,'dobiti ćemo',a+b)
```

Automatska dodjela vrijednosti

Varijable mogu biti inicijalizirane i automatski, odnosno bez korisničkog unosa, pa tako u programskom kodu možemo upisati slijedeće:

```
a = 2
b = 3.5
print (a+b)
```

Program će samostalno prepoznati kojeg su varijable tipa prilikom dodjele vrijednosti, te će ih na taj način i definirati.

Spomenuli smo da se stringovi mogu zbrajati, pri čemu ćemo kao rezultat dobiti string koji je sastavljen od tih dvoje ili više stringova, no to možemo činiti samo ako su sve varijable koje zbrajamo već definirane kao stringovi. Stringove, na taj način, nemožemo zbrajati sa brojevima, odnosno sa varijablama koje su definirane kao prava brojčana vrijednost:

```
a = 'Vedran'
b = 5
print (a+b)
```

Prethodni primjer javiti će pogrešku, broj ne možemo zbrajati sa stringom! No, za razliku od zbrajanja, operaciju množenja možemo koristiti:

```
a = 'Vedran'
b = 5
print (a*b)
```

Kao rezultat prethodnog primjera dobiti ćemo riječ Vedran ispisanu 5 puta.

Vrijednost se također može dodjeliti većem broju varijabli istovremeno. Ako dodjeljujemo istu vrijednost većem broju varijabli možemo napisati:

```
a = b = c = 10
```

Ukoliko želimo većem broju varijabli dodjeliti različite vrijednosti, bilo po veličini ili po tipu, to možemo učiniti na slijedeći način:

```
a, b, c = 10, 20.5, 'Subota'
```

Na ovaj način smo varijablama a, b, c, dodjelili različite vrijednosti koristeći samo jedan simbol jednakosti.

Da bi smo provjerili koji tip podataka je pohranjen u određenoj varijabli koju smo već prethodno definirali, možemo koristiti naredbu `type(imevarijable)`:

```
a = 100
b = 'Ovo je neki tekst!'
print ('U varijablu a pohranjena je vrijednost',a,'koja je tipa',type(a))
print ('U varijablu b pohranjena je vrijednost',b,'koja je tipa',type(b))
```

Na kraju, vodimo računa i o tome da ukoliko varijabli, koja je već prethodno definirana, dodjelimo novi tip vrijednosti, program će tu varijablu sam predefinirati na novi tip, tako da se naš novi podatak može u nju sigurno pohraniti.

Tipovi podataka

Tipove podataka, odnosno varijabli, u Pythonu najednostavnije je podjeliti na brojevne, tekstualne i složene tipove o kojima će riječi biti kasnije.

Brojevne tipove možemo podjeliti na cijele, decimalne i kompleksne brojeve, a cijeli brojevi, osim u dekadskom, mogu još biti zapisani i u binarnom, oktalnom ili heksadecimalnom obliku.

Evo nekih primjera različitih tipova brojeva:

int	bin	oct	hex	float	complex
100	0b101	0o10	0xF	55.5	3.14j

A ovako te vrijednosti dodjeljujemo varijablama u Pythonu:

```
a = 5              # dekadski broj 5
b = 0b101          # binarni broj 101 (iznosi 5 u dekadskom obliku)
c = 0o10           # oktalni broj 10 (iznosi 8 u dekadskom obliku)
d = 0xF            # heksadecimalni broj F (iznosi 15 u dekadskom obliku)
e = 55.5           # decimalni broj 55.5
f = 3.14j          # kompleksni broj 3.14j
```

Kada varijablama na ovaj način dodjelimo vrijednosti koje nisu dekadske, one se automatski konvertiraju u dekadsku vrijednost i na taj način zapišu u varijablu. Na taj način ove varijable možemo direktno koristiti u matematičkim izračunima. Kompleksni brojevi iznimno ostaju zapisani u svom originalnom obliku, no i njih se može direktno koristiti u izračunima.

Ukoliko želimo pretvoriti dekadsku vrijednost, koja je zapisana u određenoj varijabli, u neki drugi oblik zapisa, npr. binarni ili oktalni, da bi ju na taj način mogli ispisivati, moramo voditi računa da će se pri takvoj pretvorbi brojevni oblik zapisa pretvoriti u string, te ga nećemo moći direktno koristiti u izračunima prije nego li ga ponovno prethodno pretvorimo u dekadski oblik.

Evo primjera:

```
b = 0b101
```

U varijablu b zapisali smo binarni broj 101. Binarne brojeve zapisujemo tako da zapis započnemo sa brojem 0, iza kojeg slijedi slovo b, te zatim konkretan binarni broj. Prilikom dodjele vrijednosti binarni broj 101 se automatski pretvorio u dekadski broj 5, te na taj način zapisao u varijablu b.

```
print (b)          # ispisuje se broj 5 pošto je takav zapisan u varijabli b

b = bin(b)
```

Naredbom `bin(b)` konvertiramo dekadsku vrijednost koja je trenutno zapisana u varijabli b, u binarni oblik, te taj binarni zapis spremamo opet u istu varijablu `b=bin(b)`. Varijabla b je sada tipa string i sadrži tekstualni zapis 101.

```
print (b)          # ispisuje se tekst 101 koji je pohranjen u varijabli b
```

Da bi smo varijablu b mogli koristiti u izračunima, moramo je ponovno pretvoriti u dekadski oblik. To možemo učiniti na slijedeći način:

```
b = int(b,2)
```

Naredba `int(b,2)` pretvara vrijednost b, koja je trenutno zapisana u obliku sa bazom 2, u cijelobrojnu dekadsku vrijednost tipa int. Ta vrijednost se zapisuje u varijablu b, koja je sada automatski pretvorena u tip int, kako bi mogla prihvatiti broj.

```
print (b)          # ispisuje se broj 5 koji je zapisan u varijabli b
```

Naredba `rezultat=int(b,X)` pretvara tekstualni zapis pohranjen u varijabli b tipa string (taj zapis mora predstavljati broj zapisan u obliku baze X), u cjeli broj zapisan u dekadskom obliku, te ga pohranjuje u varijablu rezultat. Baza (X) može biti broj između 2 i 36.

Slijede primjeri naredbi za pretvorbu broja b sa dekadskim zapisom u oktalni i heksadecimalni, sa time da vodimo računa da b na početku konverzije bilo u oktalni ili u heksadecimalni oblik mora biti u dekadskom zapisu:

```
b = oct(b)
b = hex(b)
```

Liste

Liste su složeni tipovi podataka. Sastoje se od jednostavnih vrijednosti određenog tipa grupiranih u zajedničku listu. Vrijednosti grupirane unutar jedne liste ne moraju nužno biti istog tipa. Slijedi nekoliko jednostavnih primjera listi.

```
li1 = [2,4,6,8,10]

li2 = ['a','e','i','o','u']

li3 = ['a',5,'e',3,'i',1,'o',6,'u',8]
```

Definirane su 3 liste. Prva lista sadrži brojeve, druga lista znakove, a treća je mješana. Elementima liste pristupamo imenom liste, te indeksom elementa u listi. Indeks prvog elementa je 0, a indeks svakog slijedećeg je uvećan za 1. Indeks zadnjeg elementa u listi je n-1, gdje je n broj elemenata liste.

Elementi prve liste su tako `li1[0]`, `li1[1]`, `li1[2]`, `li1[3]` i `li1[4]`. Sa pojedinim elementima liste radimo na isti način kako bi smo radili i sa jednostavnim varijablama koje sadrže takve vrijednosti, dakle možemo ih ispisivati, učitavati, zbrajati, uspoređivati, itd. Jedini uvjet je, naravno, da je elemenat liste takav da željena operacija na njemu može biti primjenjena. Npr. ako zbrajamo 2 elementa jedne ili različitih listi, ti elementi moraju biti brojevi.

Evo nekoliko primjera za ispis elemenata liste, odnosno vršenja operacija nad njima:

```
print (li2[4])

print (li3[3]+li3[5])

print (li1[2]+li3[9])
```

Rezultat ispisa prethodnih nekoliko naredbi biti će znakovi (odnosno brojevi): u, 4 i 14.

Slijedi primjer dodjele vrijednosti, odnosno korisnički unos vrijednosti u elemente liste:

```
li1[0] = int(input('Unesi broj: '))

li1[0] = 5

li1[2] ='a'

li2[1] = li3[1]+5
```

Također, moguće je sastaviti ugnježđene liste, odnosno liste koje sadrže druge liste, npr.:

```
li4 = [li1,li2,li3]
```

Tada će nam npr. naredba `print(li4[0])` ispisati cjelu li1 listu, a naredba `print(li4[0][1])` drugi element li1 liste. Liste mogu biti spajane, zbrajane, te se nad njima mogu vršiti različite specifične funkcije. Naprednije korištenje listi biti će prikazano kasnije.

Tuple (n-torke)

Tuple, odnosno n-torke, su kao i liste, nizovi podataka određenog tipa, no za razliku od listi, one su nepromjenjive. Kako n-torku definiramo na početku, ona takva ostaje do kraja, jedini način da je modificiramo je da stvorimo novu.

Za razliku od listi, n-torke definiramo koristeći obične zagrade:

```
t1 = (1,2,'a','b')
t2 = (1,t1,'a')
t3 = (5,)
```

Vidimo da ostatak, ostaje isti, i u n-torke, kao i u liste, možemo spremati podatke različitog tipa, uključujući i druge n-torke. Kada stvaramo n-torku sa samo jednim članom, moramo nakon tog člana napisati zarez, bez obzira što je samo jedan član. Pristup se zatim vrši na slijedeći način:

```
print(t1[0])
print(t2[1][0])
```

Vidimo da je princip pristupa elementima n-torki, identičan kao i pristup elementima listi. U prethodnom primjeru, slijedeći principe koji vrijede i za liste, obje print naredbe pristupaju istom članu, prvom članu t1 n-torke, i ispisuju njegovu vrijednost.

Liste možemo pohranjivati u n-torke, i obratno. Pogledajmo primjer:

```
t1 = (1,2,'a','b')
l1 = [5,t1,'a']
t2 = ('c',l1)

print(t2[1][1][0])
```

Naredba print u prethodnom primjeru, slijedeći već navedene principe, pristupa prvom elementu t1 n-torke i ispisuje ga. Liste, čak i kada su pohranjene u n-torki, možemo mjenjati. To ne znači da se n-torka može mjenjati, ona ostaje nepromjenjena, te svi njeni članovi ostaju na svome mjestu, no pošto listu možemo samu po sebi mjenjati, posredno sadržaj jednog člana n-torke se na taj način može promjeniti. To ne vrijedi za ostale članove n-torke, npr. brojke i slova.

```
t1 = (1,2,'a','b')
l1 = [5,t1,'a']
t2 = ('c',l1)

print(t2[1][0])
l1.reverse()
print(t2[1][0])
```

U navedenom primjeru print naredba će u prvom slučaju ispisati brojku 5, a u drugom slovo a. Brisanje elemenata n-torke nije moguće. Obrišemo li sve elemente liste l1, u n-torci t2 na drugoj poziciji ostati će prazna lista. Različite n-torke možemo spajati i tako stvarati nove koje sadrže članove svih spojenih n-torki, te nad njima možemo primjenjivati različite funkcije koje smo koristili i kod listi, no samo one koje ne mjenjaju, na neki način, postojeću n-torku.

Uvjeti i logički izrazi

Osnovnu kontrolu toka postižemo korištenjem naredbe if koja nam uvjetuje izvođenje bloka programa, ispunjenjem određenog uvjeta. Za početak, ovako ćemo provjeriti da li varijabla postoji i je li različita od 0:

```
a = 1
b = 0

if a:
    print ('Varijabla a sadrži vrijednost različitu od 0!')
    print ('Ta vrijednost iznosi:',a)

if b:
    print ('Varijabla b sadrži vrijednost različitu od 0!')
    print ('Ta vrijednost iznosi:',b)

print ('Pozdrav!')
```

U predhodnom primjeru imamo dvije if naredbe sa svoja dva pripadna bloka, čije naredbe će se izvršiti samo ako je uvjet bloka ispunjen, u ovom slučaju ako su varijable a odnostno b brojevi različiti od 0. Konkretno "if a:" nas pita "Da li a postoji i jeli različit od 0?"

Blok koji pripada određenoj if naredbi moramo obavezno indentirati, odnosno odvojiti od lijevog ruba ekrana pomoću jednog tabulatora (tipka tab). Tako npr. vidimo da blok koji pripada "if a:" naredbi sadrži dvije print naredbe, blok koji pripada "if b:" naredbi sadrži svoje dvije print naredbe, a naredba print('Pozdrav!') je samostalna i ne pripada više niti jednom bloku, te će se izvršiti bezuvjetno.

U predhodnom primjeru dobili smo ispis poruke o varijabli a, pošto je bila različita od 0. Niti jednu poruku vezanu uz varijablu b nismo dobili, pošto je iznosila 0, i blok koji pripada "if b:" naredbi nije se izvršio. Na slijedeći način možemo ostvariti izvođenje različitog bloka naredbi ukoliko treženi uvjet nije ispunjen:

```
a = int(input('Unesi broj: '))

if a:
    print ('Varijabla a sadrži vrijednost različitu od 0!')
    print ('Ta vrijednost iznosi:',a)
else:
    print ('Varijabla a ne sadrži vrijednost različitu od 0!')

print ('Pozdrav!')
```

U ovom primjeru korisnik unosi neku brojčanu vrijednost, koja se zatim pohranjuje u varijablu a. Program zatim ispituje "if a:" da li varijabla a postoji i je li to broj različit od 0. Ukoliko jeste, izvršava prvi blok naredbi, a ukoliko nije, odnosno ukoliko je u varijabli a zapisana 0, izvršava blok pod "else:".

Ukoliko želimo provjeriti, ne samo je li u nekoj varijabli smještena vrijednost različita od 0, već i kolika je ta vrijednost zbilja, je li veća ili manja od neke druge vrijednosti, i slično, koristiti ćemo neki od postojećih logičkih izraza. Logički izrazi koriste se za usporedbu vrijednosti.

Slijedi popis postojećih logičkih izraza:

Izraz	Opis	Primjer
==	Ako je veličina dviju uspoređenih vrijednosti jednaka, ovaj izraz je istinit.	if a==b:
!=	Ako je veličina dviju uspoređenih vrijednosti različita, ovaj izraz je istinit.	if a!=b:
>	Ako je prva vrijednost veća od druge, ovaj izraz je istinit.	if a>b:
<	Ako je prva vrijednost manja od druge, ovaj izraz je istinit.	if a<b:
>=	Ako je prva vrijednost veća ili jednaka od druge, ovaj izraz je istinit.	if a>=b:
<=	Ako je prva vrijednost manja ili jednaka od druge, ovaj izraz je istinit.	if a<=b:

Slijedeći primjer sadrži nekoliko mogućnosti usporedbe dviju vrijednosti:

```
a = int(input('Unesi broj: '))
b = int(input('Unesi broj: '))

if a==5:
    print ('a iznosi pet!')
if b<0:
    print ('b je negativan broj!')
if a!=b:
    print ('a i b su različite vrijednosti!')
if b>=10:
    print ('b je vrijednost veća ili jednaka od 10!')

print('Pozdrav!')
```

Primjer sadrži četiri različita if uvjeta sa pripadnim blokovima naredbi koje će se izvršiti ukoliko je neki od tih uvjeta ispunjen. Svi uvjeti se provjeravaju, jedan po jedan, te se za one uvjete koji su istiniti, pokreću i izvršavanju njihovi pripadni blokovi naredbi.

Kombiniranje logičkih izraza

Operatori za kombinaciju i negaciju logičkih izraza navedeni su u tabeli:

Izraz	Opis	Primjer
and	Oba izraza moraju biti istinita da bi ukupan izraz bio istinit.	if a>=b and a<=10:
or	Ako je barem jedan od izraza istinit, ukupan izraz je istinit.	if a==5 or b==5:
not	Postavlja vrijednost izraza na suprotnu vrijednost, tj. negira ju.	if not(a>b):

Ukoliko želimo postaviti složeniji logički uvjet, to ćemo učiniti kombiniranjem većeg broja jednostavnih izraza. Prioritet provjere izraza odrediti ćemo zagradama.

```
if (a>=5 and a<=10) or (b>=5 and b<=10):
    print ('Barem jedna od vrijednosti u a i b je broj između 5 i 10!')
```

U ovom primjeru najprije provjeravamo vrijede li svi zasebni logički izrazi, te vrijede li sklopovi izraza unutar zasebnih zagrada, te na kraju vrijedi li barem jedna od navedenih zagrada.

U prethodnom primjeru provjeravali smo da li je barem jedna od vrijednosti koje su smještene u varijablama a i b, broj između 5 i 10, uključujući 5 i 10. Ako bi na cjeli taj sklp uvjeta primjenili negaciju, tražili bi smo u biti suprotnu stvar, da niti jedna od vrijednosti smještenih u varijablama a i b, ne smijebiti broj između 5 i 10. Slijedi primjer takve negacije:

```
if not((a>=5 and a<=10) or (b>=5 and b<=10)):
    print ('Niti jedna od vrijednosti u a i b nije broj između 5 i 10!')
```

Ulančani i ugnježđeni IF

U prethodnoj cjelini naveden je primjer provjere složenijeg sklopa logičkih izraza. To smo činili tako da smo provjeravali više zasebnih izraza istovremeno, te vrijede li njihove kombinacije ostvarene preko nekih od operatora kombinacije. No ponekad, kombiniranje jednostavnih logičkih izraza nije dovoljno.

Ukoliko neku provjeru ima smisla izvršavati tek ukoliko neki prethodno provjereni uvjet nije zadovoljen, onda ćemo prvo postaviti provjeru prvog uvjeta, te ukoliko ona ne vrijedi, i samo ako ne vrijedi, izvršiti provjeru slijedećih mogućnosti. Takav sklop provjera zove se ulančani IF.

```
a = int(input('Unesi broj: '))

if a==1:
    print ('a je jedan!')
elif a==2:
    print ('a je dva!')
elif a==3:
    print ('a je tri!')
elif a==4:
    print ('a je četiri!')
elif a==5:
    print ('a je pet!')
else:
    print ('a nije niti jedan od navedenih brojeva!')
```

Ovakav sklop provjera specifičan je po tome što, ukoliko smo ustanovili da a iznosi jedan, nećemo više izvoditi niti jednu od slijedećih ulančanih provjera, što štedi procesorsko vrijeme. Da smo umjesto ovakve ulančane provjere imali 5 samostalnih IF provjera, tada bi se, čak i da se ustanovilo da je a jednak 1, ipak izvršavale i sve ustale samostalne IF provjere.

Sumirano, provjera gdje linija počinje sa IF uvjek se izvršava, dok se ELIF provjere izvršavaju tek ukoliko niti jedan od prethodnih ulančanih uvjeta nije zadovoljen. U trenutku kada se, u sklopu elif-ova dođe do uvjeta koji vrijedi, svi elif-ovi koji su ulančani iza toga se više ne provjeravaju. Ukoliko niti IF niti ijedan od elif-ova ne vrijedi, izvršava se ono što je navedeno pod ELSE, ako je taj else blok naveden.

IF možemo i ugnjezditi, što znači da ćemo tek ukoliko neka provjera vrijedi, i samo ako vrijedi, vršiti neke dodatne provjere koje se nalaze u njenom bloku.

```
a = int(input('Unesi broj: '))
b = int(input('Unesi broj: '))
c = int(input('Unesi broj: '))

if a==1:
    if b>=0:
        print ('a iznosi 1, a b je pozitivan!')
    else:
        print ('a iznosi 1, a b je negativan!')
elif a==2:
    if c>=0:
        print ('a iznosi 2, a c je pozitivan!')
    else:
        print ('a iznosi 2, a c je negativan!')
else:
    print ('a ne iznosi ni 1 ni 2!')
```

U prethodnom primjeru, najprije u glavnom ulančanom IF-u, provjeravamo iznosi li a 1 ili 2, ili nije niti jedna od te dvije vrijednosti. Ukoliko a iznosi 1, provjeravamo je li b pozitivan, a c nam nije bitan. Ukoliko a iznosi 2, provjeravamo je li c pozitivan, a b nam nije bitan. Ukoliko a ne iznosi ni 1 ni 2, ispisujemo poruku o tome, i ne vršimo nikakvu dodatnu provjeru.

Koji blok naredbi pripada kojem uvjetu vidimo preko uvlačenja linija. If, ELIF i ELSE, koji pripadaju istom lancu, uvučeni su u jednakoj mjeri od lijevog ruba. Ostalo što je uvučeno dalje, su blokovi koji pripadaju onom IF-u, odnosno ELIF-u koji se nalazi najbliže iznad tog bloka.

Petlje

Ukoliko želimo neki blok naredbi ponavljati određeni broj puta, taj blok smjestiti ćemo unutar petlje. Petlja ponavlja onaj blok naredbi koji joj pripada određeni specificirani broj puta, ili neprekidno, dok neki dodatni uvjet nije zadovoljen.

For petlja

Ovu petlju koristiti ćemo kada znamo koliko puta želimo nešto ponoviti. Krenimo sa najednostavnijim primjerom.

```
for a in range(1, 10):
    print (a)
```

Ova petlja ispisati će brojeve od 1 do 9. Dakle, varijabla a je brojač petlje, brojač je inicijalno broj 1, te se povećava za jedan dok ne dođe do 10, no u trenu kada postane 10, petlja se više ne izvršava nego se prekine, te zbog toga broj 10 nije ispisan. Nakon što je petlja prekinuta, program se nastavlja od iza zadnje linije bloka koji pripada toj petlji. Taj blok je naravno uvučen.

Ukoliko ne definiramo početnu vrijednost brojača, on inicijalno kreće od 0. Slijedeći primjer ispisati će brojeve od 0 do 4.

```
for a in range(5):
    print (a)
```

Evo sada malo složenijeg bloka:

```
for a in range(1, 10):
    b = int(input('Unesi broj: '))
    if b>=0:
        print ('b je veći ili jednak od nule!')
    else:
        print ('b je negativan broj!')
```

U ovom primjeru, program će tražiti korisnika da 9 puta (brojač petlje se mjenja toliko puta), upiše neki broj (b), te će ispitati je li taj broj koji je korisnik unio pozitivan ili negativan, te će ispisati poruku o tome. Cjeli ovaj blok naredbi pripada ovoj FOR petlji.

Brojač petlje može se, umjesto kroz raspon cjelih brojeva, kretati kroz raspon postojeće definirane liste. To ne mora biti lista brojeva. Slijedeći primjer ispisuje brojač, koji poprima vrijednosti svih znakova (jednog po jednog) u listi znakova:

```
li = ['a', 'b', 'c', 'd', 'e']

for a in li:
    print (a)
```

Brojač se može postaviti i na vrijednost duljine same liste. Uzmimo navedenu listu za primjer čija duljina je 5 (članova). Postavimo li brojač na vrijednost duljine ove liste, isto je kao da smo ga potavili i direktno na broj 5. Ispisati će se brojevi od 0 do 4.

```
li = ['a', 'b', 'c', 'd', 'e']

for a in range(len(li)):
    print (a)
```

Želimo li ispisati znakove koji su u listi, ali da nam pritom brojač poprima vrijednosti cjelih brojeva, a ne vrijednosti članova liste možemo to učiniti na slijedeći način:

```
li = ['a', 'b', 'c', 'd', 'e']

for a in range(len(li)):
    print (li[a])
```

Prethodni primjer možemo realizirati i na idući način:

```
for a, s in enumerate(li):
    print (a)
    print (s)
```

Varijabla a će nam poprimati vrijednosti od 0 do 4, a varijabla s vrijednosti članova liste.

Na ovaj način možemo iterirati i kroz višestruke liste u istoj petlji. Jednostavno definiramo više brojača, koje postavimo da poprimaju vrijednosti navedenih listi. Slijedi primjer:

```
li1 = ['a', 'b', 'c', 'd', 'e']
li2 = ['x', 'y', 'z', 'w', 'q']
li3 = [1, 2, 3, 4, 5]

for a, s, br in zip(li1, li2, li3):
    print (a)
    print (s)
    print (br)
```

Prethodni primjer možemo realizirati i na slijedeći način:

```
li1 = ['a', 'b', 'c', 'd', 'e']
li2 = ['x', 'y', 'z', 'w', 'q']
li3 = range(1,6)


for a, s, br in zip(li1, li2, li3):
    print (a)
    print (s)
    print (br)
```

Vodimo računa o tome da nam liste mogu biti različite duljine. U tom slučaju petlja će iterirati kroz sve liste, ali samo do duljine najkraće liste.

Želimo li da naš brojač prođe kroz određeni raspon brojeva, ali da se ne povećava samo za 1, pa da ne poprimi vrijednosti svih brojeva u rasponu, nego da se povećava za veći iznos pa da dio brojeva raspona preskoči, definirati ćemo ne samo raspon kroz koji brojač prolazi, nego i za koliko se povećava, na slijedeći način:

```
for a in range(1, 10, 2):
    print (a)
```

Treća vrijednost koju smo dodali pored raspona je vrijednost povećanja brojača u svakoj iteraciji petlje. Ovaj primjer iterira kroz brojeve od 1 do 9, no sa povećanjem brojača za 2 svaki krug petlje te će kao rezultat ove petlje ispisani biti brojevi 1, 3, 5, 7, 9.

Na sličan način možemo preskakati i kroz elemente liste, odnosno, možemo pristupati svakom n-tom članu liste. U slijedećem primjeru pristupamo svakom drugom članu liste i ispisujemo ga:

```
li = ['a','e','i','o','u']

for a in li[::2]:
    print (a)
```

I u rasponu brojeva, i u listi, brojač može svaku iteraciju petlje preskakati za n, sa time da n mora biti manji od veličine samog raspona, odnosno liste.

While petlja

Ukoliko želimo nešto ponavljati, do trenutka ispunjenja određenog uvjeta, ali neznamo unaprijed kada će taj uvjet biti ispunjen, odnosno neznamo koliko će se puta točno nešto morati ponoviti, koristiti ćemo while petlju.

While petlja, kao i for petlja, također može koristiti brojač, no taj se brojač mora mjenjati ručno unutar samog bloka naredbi koji pripada toj petlji. Najjednostavniji oblik while petlje može se koristiti za istu namjenu kao i for petlja, pa tako umjesto:

```
for a in range(1, 10):
    print (a)
```

možemo napisati:

```
br = 1
while (br < 10):
    print(br)
    br = br + 1
```

Dok nam, u prethodno vidljivim primjerima, for petlja odmah, unutar svog zaglavlja, definira sa kojom vrijednosti brojač kreće, i na kojoj završava, while petlja definira samo uvjet koji kaže dok god je brojač takav kako je navedeno (manji od 10) petlja će se izvršavati, a mi moramo sami prije zaglavlja petlje definirati početnu vrijednost brojača, te unutar bloka naredbi koji pripada petlji, definirati kako će se taj brojač mjenjati.

To je zato jer osnovna namjena while petlje nije izvršavanje neke radnje specifičan broj puta, to je namjena for petlje. Osnovna namjena while petlje je izvršavanje neke radnje sve dok neki uvjet, a čija promjena nama nije unaprijed poznata, ne prestane vrijediti.

```
br = 0
while (br<10) or (br>20):
    print('Unesi broj između 10 i 20: ')
    br = int(input('Unesi broj: '))

print('Broj je uspješno unesen!')
```

U prethodnom primjeru prikazana je jedna situacija gdje nam je potrebna while petlja. Primjer traži od korisnika programa unos broja čija vrijednost mora biti između 10 i 20, uključujući te dvije vrijednosti. Petlja ponavlja zahtjev za unos broja, sve dok je broj koji korisnik unosi manji od 10 ili veći od 20. U ovoj situaciji mi neznamo koliko će puta korisnik unjeti neodgovarajuću vrijednost, te se stoga while petlja ponavlja dok god je to potrebno, teoretski i beskonačno, ukoliko korisnik nikada ne unese ispravnu vrijednost.

Također, u primjeru vidimo i kako logičke uvjete možemo kombinirati pomoću operatora za kombinaciju logičkih izraza, koji su prethodno već bili spominjani. Bitno je samo pripaziti na prioritet zagrada da bi logika postavljenog uvjeta odgovarala našim željenim zahtjevima.

Uvjeti ne moraju nužno biti vezani uz konkretnu vrijednost broja, u uvjetu možemo tražiti ispunjenje bilo čega pod uvjetom da taj zahtjev može vratiti vrijednost true ili false.

Pogledajmo slijedeći primjer:

```
li = [11,30,24,58,42]

i=0
while i not in li:
    i = int(input('Pogodi broj u listi: '))

print('Pogodio si!')
```

Ovdje korisnik unosi neki broj, a petlja se ponavlja dok god uneseni broj ne odgovara vrijednosti ikojeg člana postojeće liste. Naravno, u ovoj situaciji, početnu vrijednost varijable u koju će se pohranjivati vrijednosti koje korisnik unosi, postaviti ćemo na početku na vrijednost koje nema u listi, inače se blok naredbi ove petlje nebi izvršio niti jednom.

```
ime = 'Python'

s='a'
while s not in ime:
    s = input('Pogodi slovo u imenu: ')

print('Pogodio si!')
```

U prethodnom primjeru uopće ne ispitujemo brojčane vrijednosti. Ovdje korisnik unosi slovo po slovo, dok god ne pogodi slovo koje se nalazi u prethodno definiranom imenu. Također, u ovom primjeru, korisnik može unjeti i ne samo slovo, već i dio imena, ili cijelo ime, te ukoliko pogodi, program prihvaća i to kao unos zadovoljavajući za ispunjenje uvjeta petlje. Pripazimo samo, program razlikuje velika i mala slova, te malo slovo ne izjednačava sa velikim. Početnu vrijednost varijable postavili smo, kao i u primjeru prije, na vrijednost koja ne paše uvjetu petlje, tako da se petlja izvrši barem jednom.

```
ime = 'Python'

s='a'
while s != ime:
    s = input('Pogodi ime: ')

print('Pogodio si!')
```

Ukoliko želimo da korisnik pogodi cijelo ime, uvjet možemo napisati kako je iznad navedeno.

Naredbe za kontrolu toka petlje

Može se dogoditi da, u slučaju ispunjenja nekog dodatnog vanjskog uvjeta, moramo kompletno prekinuti izvođenje petlje, ili možda preskočiti tu iteraciju petlje i nastaviti od slijedeće. Da bi smo u svom programu pokrili i takve mogućnosti koristiti ćemo naredbe break i continue.

Naredba break, u trenutku kada je pozvana, prekida izvođenje cijele petlje te tok programa nastavlja od prve slijedeće naredbe koja se nalazi izvan petlje.

Naredba continue, kada je pozvana, prekida izvođenje trenutne iteracije petlje, i vraća tok programa na početnu liniju petlje, ali u idućoj iteraciji.

Slijedi primjer korištenja naredbe break:

```
for s in 'Python':
    if s == 'h':
        break
    print(s)
```

Ova petlja ispisati će slova P, y, t, te u trenutku kada izvođenje dođe do slova h izvršiti će se naredba break će će se petlja kompletno prekinuti, a preostala slova zato neće biti ispisana.

Naredba continue, za razliku od naredbe break, prekida samo trenutnu iteraciju petlje:

```
for s in 'Python':
    if s == 'h':
        continue
    print(s)
```

Ova petlja ispisati će slova P, y, t, o, n, pošto naredba continue neće prekinuti izvršenje cjelokupne petlje, nego će u trenutku kada dođe do slova h, samo preskočiti na slijedeću iteraciju, te normalno nastaviti program od slova o na dalje.

Rad sa tekstom

String je tekstualni tip podataka. Varijable koje su tipa string, pohranjuju podatke u tekstualnom obliku. U takve varijable možemo pohraniti slova, riječi, rečenice, ili cjele odlomke teksta, a ono što je uneseno omeđujemo apostrofima ili navodnicima. Na taj način string možemo definirati na jedan od slijedeća 2 načina:

```
s1 = 'Python'
s2 = "Python"
```

U oba slučaja definirani tekst prihvaćen je kao string. Obveza je samo koristiti isti znak koji smo koristili za otvaranje stringa (npr. apostrof) i za zatvaranje stringa.

Ukoliko želimo unutar teksta, koristiti navodnike i apostrofe, kao dio samog teksta, moramo ispred svakog navodnika ili apostrofa koji je dio teksa, a ne otvarač ili zatvarač, koristiti znak backslash '\', inače bi nam prvi idući navodnik, odnosno apostrof (ovisno o tome sa čime smo otvorili string), zatvorio string, i sav tekst iza toga nebi se više računao kao dio to stringa.

```
nekitekst = 'Student\'s papers are terrible!'
```

Još jedna zanimljiva činjenica koja nam može olakšati posao je da backslash moramo koristiti samo za umetanje onog znaka sa kojim smo otvorili string. Npr. ako smo otvorili sa apostrofom, onda navodnike možemo koristiti unutar teksta nesmetano i bez korištenja backslash-a i obratno.

Najkorišteniji specijalni znakovi koji se koriste u stringu su \t za umetanje tabulatora i \n za novi red, a možemo koristiti i \\ da bismo umetnuli i sam simbol backslash-a u tekst.

Nakon što smo definirali string, i unjeli neki sadržaj u tu varijablu, mi možemo, osim do cjelokupnog teksta, doći i do svakog pojedinog znaka, ili određenog dijela teksta, unutar stringa, korištenjem brojača.

Uzmimo za primjer slijedeću rečenicu:

```
nekitekst = 'Ovo je neka rečenica.'

print('Ispis cijele rečenice:',nekitekst)
print('Ispis prvog slova:',nekitekst[0])
print('Ispis prvog slova:',nekitekst[7:10])
```

Varijabla nekitekst sadrži cjeli uneseni tekstualni sadržaj, dok nekitekst[n] označava n-to slovo u sadržaju, sa time da vodimo računa da prvi znak sadržaja ima indeks n=0.

U prethodnom primjeru, na taj način, nekitekst[0] predstavlja prvo slovo u navedenoj rečenici, tj. Slovo 'O', a ukoliko želimo pristupiti dijelu sadržaja od slova sa indeksom n, do slova sa indeksom m, možemo se poslužiti, prema navedenom primjeru, sa nekitekst[7:10] koji bi u ovom slučaju označavao dio teksta od slova sa indeksom 7 do slova sa indeksom 9 (jedan manje od navedenog). U navedenom slučaju taj dio teksta sadrži znakove 'nek'.

Stringove možemo i zbrajati, čime zadržaje dvaju ili više stringova lijepimo jedan na drugi, u onom poretku u kojem smo ih u operaciji naveli, a možemo ih i množiti sa nekim brojem x, gdje onda kao rezultat dobivamo sadržaj stringa kopiran x puta jedan za drugim.

```
s1 = 'Učim'
s2 = 'Python'

print (s1+s2)
print (s1+' '+s2)
print (s2*3)
print ((s2+' ')*3)
```

Pokretanjem programa iz prethodnog primjera, prilikom ispisa dobiti ćemo slijedeće linije:

```
UčimPython
Učim Python
PythonPythonPython
Python Python Python
```

Neka je s neki tekst. Slijedi popis nekih string metoda koje možemo nad tim tekstom koristiti:

Primjer	Opis
`print(s.lower())`	Pretvara sve znakove u tekstu u lowercase znakove.
`print(s.upper())`	Pretvara sve znakove u tekstu u uppercase znakove.
`print(s.swapcase())`	Svakom znaku u teksu mjenja kapitalizaciju.
`print(s.capitalize())`	Prvi znak u tekstu pretvara u uppercase znak.
`print(s.count('dio teksta'))`	Broji koliko puta se navedeni dio teksta pojavljuje u tekstu.
`print(s.find('dio teksta'))`	Nalazi i vraća indeks prvog znaka prvog pojavljivanja navedenog dijela teksta, inače vraća -1.
`print(s.isalnum())`	Vraća True ako su svi znakovi u tekstu alfanumerički.
`print(s.isdigit())`	Vraća True ako su svi znakovi u tekstu brojke.

Rad sa listama

Osnovna struktura liste i prikaz pristupa pojedinim elementima liste opisani su detaljnije već u jednom od prethodnih poglavlja. Ovdje će biti prikazano korištenje nekih funkcija za napredniji rad sa listama.

Podsjetimo se, lista može sadržavati elemente koji ne moraju biti istog tipa, te također može sadržavati i podliste, koje mogu biti sastavljene od dodatnih podlisti itd.

```
li01 = ['a',1]
li02 = ['b',2]
li03 = ['c',3]
li04 = ['d',4]

li11 = [li01,li02]
li12 = [li03,li04]

li21 = [li11,li12]

print(li21)
```

Pogledajmo primjer. Lista li21 sadrži dva člana, to su liste li11 i li12. Lista li11 sadrži liste li01 i li02, a lista li12 liste li03 i li04. Liste li01, li02, li03 i li04 sadrže slova i brojke. Ukoliko bi smo ispisali sadržaj liste li21 korištenjem navedene naredbe print(li21) dobili bi smo slijedeći ispis:

```
[[['a', 1], ['b', 2]], [['c', 3], ['d', 4]]]
```

Ovaj ispis najbolje prikazuje od čega se sastoji spomenuta lista li21. Svaki par uglatih zagrada predstavlja jednu listu, a temelj naništih podlisti su slova i brojevi. Da bi smo ispisali element 'a' koristili bi slijedeći poziv:

```
print(li21[0][0][0])
```

Element 'b' bi za usporedbu ispisali pomoću slijedećeg poziva:

```
print(li21[0][1][0])
```

Znači elementi liste li21 su li21[0] i li21[1] a predstavljaju liste li11 i li12. Članovi liste li11 su li11[0] i li11[1], a to su liste li01 i li02. Slijedeći to pravilo dolazimo do toga da je:

```
'a' = li01[0] = li11[0][0] = li21[0][0][0]
```

Idemo sada vidjeti što još sa listama možemo raditi, osim pristupa i korištenja njihovih elemenata. Prvo treba spomenuti korištenje negativnog indeksa. Znači ako je li[0] prvi element liste li, a li[1] drugi element, itd. krenemo li u suprotnom smjeru li[-1] je zadnji element, li[-2] predzadnji itd.

To možemo zamisliti na način da zamislimo da dvije identične liste stoje jedna pored druge, recimo da naša lista li ima 3 elementa:

[prazno] ... li[-3]='prvi' li[-2]='drugi' li[-1]='treći' ... li[0]='prvi' li[1]='drugi' li[2]='treći' ... [prazno]

Uzmimo da imamo listu:

```
li = ['a',1,'b',2,'c',3]
```

Ovu listu možemo, za početak, rezati. Pa napišimo:

```
print(li[:3])
print(li[3:])
print(li[2:4])
```

Prva naredba će pristupiti elementima od početnog do elementa sa indeksom do 3 (ali bez 3) znači elementima sa indeksom 0, 1, 2. Druga naredba će pristupiti elementima počevši sa elementom sa indeksom 3, pa do zadnjeg, znači elementima 3, 4, 5. Treća naredba pristupa elementima počevši sa indeksom 2 pa do 4 (ali bez 4), znači elementima sa indeksom 2, 3. U prethodnom primjeru mi te elemente samo ispisujemo, ali izraz li[x:y] možemo koristiti u sklopu bilo koje druge naredbe koja radi sa elementima liste, pa teko recimo možemo stvoriti novu listu od samo dijela elemenata postojeće liste, na jednostavan način:

```
li2 = li[2:4]
```

Postojećoj listi možemo dodatati elemente. Uzmimo opet našu listu li iz prethodnog primjera.

```
li = ['a',1,'b',2,'c',3]

li.append(4)
```

Naredba append dodaje jedan element (upisan u zagradama) na kraj liste te sada imamo:

```
['a', 1, 'b', 2, 'c', 3, 4]

li.insert(6,'d')
```

Naredba insert na poziciji sa indeksom N (6) umeće element E ('d'), a ostali elementi koji su iza pozicije sa tim indeksom pomiču se jedno mjesto udesno, te sada imamo:

```
['a', 1, 'b', 2, 'c', 3, 'd', 4]

li.extend(['e',5])
```

Naredba extend dodaje elemente liste, navedene u zagradama, postojećoj listi li, te dobijamo:

```
['a', 1, 'b', 2, 'c', 3, 'd', 4, 'e', 5]
```

Treba napomenuti, da smo ovdje na kraju umjesto naredbe extend, koristili naredbu append, i naveli u zagradama `['e',5]` listi bi se na zadnje mjesto umetnuo samo jedan član, kojeg bi u biti predstavljala ova podlista, te bi to izgledalo ovako:

```
['a', 1, 'b', 2, 'c', 3, 'd', 4, ['e', 5]]
```

Prilikom korištena metode extend, u zagradama možemo navesti i ime neke druge, postojeće liste čije elemente želimo pridodati prvotno navedenoj listi.

Elemente liste možemo i pretreživati. Za to možemo koristiti metodu index:

```
print(li.index('b'))
```

Metoda index će nam u ovom slučaju vratiti vrijednost 2, pošto je to indeks prve pozicije na kojoj se pojavljuje element 'b'. Ukoliko traženog elementa u listi nema, ova funkcija vratiti će nam iznimku. O iznimkama će više riječi biti kasnije. Za sada, da bi smo izbjegli tu pogrešku najjednostavnije je koristiti provjeru, da li element postoji u listi, te ako postoji, tek onda provjeravati njegovu poziciju:

```
if 'd' in li:
    print('Index trazenog elementa je: ',li.index('d'))
else:
    print('U listi nema tog elementa!')
```

Određeni element iz liste možemo obrisati korištenjem metode remove:

```
li.remove('b')
```

Poziv ove metode briše prvo pojavljivanje navedenog elementa iz liste. Ukoliko se navedeni element pojavljuje više puta, ova metoda briše samo prvi po redu. Ukoliko elementa nema u listi, javlja se iznimka kao i u primjeru pretraživanja, te bi trebalo iskoristiti istu provjeru postojanja nekog elementa, prije nego zatražimo njegovo brisanje.

Ukoliko želimo obrisati samo zadnji element postojeće liste, dovoljno je pozvati metodu pop:

```
li.pop()
```

Poziv ove briše zadnji element u listi, te vraća njegovu vrijednost, što znači ukoliko bismo ovu naredbu koristili u kombinaciji sa naredbom print, `print(li.pop())` uz brisanje zadnjeg elementa, dobili bismo i ispis vrijednosti tog elementa u konzoli.

```
li.pop(2)
```

Ukoliko želimo pobrisati element liste koji se nalazi na određenoj poziciji, metodi pop dodati ćemo parametar. Taj parametar biti će indeks pozicije elementa koji će biti obrisan.

Liste se također mogu spajati operatorom zbrajanja, to ima isti rezultat kao korištenje prikazane metode extend. Također, operator zbrajanja možemo koristiti da postojećoj listi dodamo još jedan element:

```
li = li1 + li2
li = li + ['x']
```

Množimo li listu sa nekim cjelim brojem x, kao rezultat dobiti ćemo tu listu sa svojim osnovnim elementima ponovljenim x puta.

Sa listama možemo koristiti i razne dodatne metode poput `li.sort()` da sortiramo njene elemente ukoliko su istog tipa, `li.reverse()` da im okrenemo poredak, itd.

Funkcije

Funkcija je blok organiziranog koda koji ima svrhu da izvrši neki zadatak. Drugim riječima, funkcija je skup naredbi koje su organizirane u jednu cjelinu, i kada god se funkcija pozove, sve naredbe koje pripadaju toj funkciji se izvrše. Slijedi primjer definicije jedne funkcije:

```
def ispis():
    print('Ovo je ispis u funkciji!')
    return
```

Ime navedene funkcije je ispis, ključna riječ def, definira da se ovdje radi o funkciji, a zagrade nakon njenog imena služe da bi se unutra naveli vrijednosti koje joj, eventualno, iz programa prosljeđujemo. Nakon dvotočke, u idućem redu slijede naredbe koje pripadaju bloku funkcije, sve do naredbe return koja označava kraj, te eventualno vraćanje neke vrijednosti programu ukoliko je ta vrijednost navedena. Funkciju možemo definirati u bilo kojem dijelu programa, no naravno moramo ju definirati prije njenog prvog poziva. Funkciju, nakon što je definirana, pozivamo njenim imenom i zagradama, na slijedeći način:

```
ispis()
```

Navedena je jednostavna funkcija koja ne redi ništa drugo osim ispisa jedne linije teksta, no postojanje složenijih funkcija omogućuje nam visok stupanj ponovne iskoristivosti koda. Drugim riječima, sve naredbe koje smo u funkciji jednom definirali više ne moramo nikada ponovno pisati, dovoljno je svaki put kada trebamo obaviti taj zadatak, koji taj sklop naredbi rješava, samo pozvati funkciju njenim imenom, i cijeli taj sklop naredbi će se izvršiti.

Ponovna iskoristivost koda, još je bolje uočljiva kada funkciji prosljeđujemo neke vrijednosti. Slijedi primjer takve funkcije:

```
def ispis2(ime):
    print('Dobar dan! Ja se zovem',ime,'!')
    return
```

Ovoj funkciji prosljeđujemo jedan parametar, varijablu ime. Funkcija će, koji god tekst joj prosljedimo prilikom poziva njegog imena, pridodati na pozdrav, i ispisati cijelu rečenicu. Pozivi te funkcije mogu izgledati na slijedeći način:

```
ispis2('Python')
ispis2('Vedran')

ime = input('Unesi ime: ')
ispis2(ime)
```

Funkciji možemo prosljediti konkretnu vrijednost, u ovom slučaju konkretan tekst, ili neku varijablu prikladnog tipa, čija vrijednost je bila definirana negdje drugdje u programu. Funkcijama možemo prosljeđivati varijable svih tipova, uključujući i cjele liste.

U slijedećem primjeru definirati ćemo listu imena. Funkciju ćemo koristiti za dodavanje imena u listu. Svaki poziv funkcije pokrenuti će upit korisniku za dodavanje jednog dodatnog imena u listu:

```
def dodajlisti(mojalista):
    ime = input('Napiši ime koje želiš dodati u listu: ')
    mojalista.append(ime)
    return

mojalista = ['Vanda','Antonio','Lili']

dodajlisti(mojalista)
dodajlisti(mojalista)
```

Početna lista sadrži tri imena a funkcija je pozvana dva puta, što znači da će se blok naredbi funkcije izvršiti dva puta. Znači, dva puta će se korisnika tražiti da unese neko ime, i to ime će se dodati u listu. Lista je kao parametar prosljeđena funkciji iz glavnog programa, te jeto lista koja se modificira. Nakon što je korisnik unio dva imena, npr. Bono i Silvia, lista će izgledati ovako:

```
['Vanda', 'Antonio', 'Lili', 'Bono', 'Silvia']
```

U Pythonu, svi se parametri automatski prenose preko reference, što znači da će svaka promjena, koja se nad prosljeđenim varijablama primjeni unutar funkcije, vrijediti i u ostatku programa. Tako ova dva imena koja smo listi dodali u funkciji, u listi ostaju trajno, dok ih mi sami ne obrišemo.

Vrijednosti parametara možemo postaviti i na pretpostavljenu vrijednost, koja će se zatim primjeniti ukoliko taj parametar ne prosljedimo prilikom poziva funkcije:

```
def info(ime,prezime,jezik='Python'):
    print('Zovem se',ime,prezime,'i učim',jezik)
    return

info('Vedran','Strčić','C++')
info('Vedran','Strčić')
```

U prethodnom primjeru definirana je funkcija sa tri parametra, sa time da treći parametar ima pretpostavljenu vrijednost. To znači da ukoliko vrijednost tog parametra ne prosljedimo funkciji prilikom poziva, primjeniti će se ta default vrijednost. Dva poziva ovoj funkciji u navedenom primjeru dati će slijedeći ispis:

```
Zovem se Vedran Strčić i učim C++
Zovem se Vedran Strčić i učim Python
```

Svim postojećim parametrima funkcije može se dodjeliti pretpostavljena vrijednost.

Funkcije također mogu primiti i više parametara nego što smo na početku definirali. Ukoliko postoji mogućnost da će funkcija, u određenim situacijama, morati primiti više parametara nego što smo planirali, dovoljno je pored poznati parametara, koje smo već definirali, definirati i Tuple (n-torku), koju ćemo označiti sa simbolom * prije imena.

Navedena n-torka može biti prazna, ukoliko se ne unese niti jedan dodatni parametar prilikom poziva funkcije, ili se može napuniti sa n dodatnih elemenata prilikom poziva. Ta n-torka funkcionira kao lista, no nepromjenjiva. Elementi se iz nje ne mogu brisati, dodavati novi, niti im se mogu mjenjati pozicije jednom kada je ta n-torka stvorena. Ta n-torka se stvara, u ovom slučaju, prilikom poziva funkcije, ukoliko joj prosljedimo više parametara nego što je to definirano. Slijedi primjer:

```
def ispis(var1,var2,*ostalo):
    print(var1,var2)
    for i in ostalo:
        print(i)
    return

ispis(10,20)

ispis('Python','C++')

ispis(10,20,'Ana',30.5,40,'Vedran',['a',10],[20,'b'])
```

U prethodnom primjeru vidimo, kako definiranoj funkciji uvjek moramo proslјediti barem dva elementa, a ukoliko ih prosljedimo više, dodatni elementi, koji mogu biti bilo kojeg tipa, čak i liste, umeću se u n-torku, te im preko nje možemo pristupati i koristiti ih.

Funkcije koje vraćaju neku vrijednost moraju pored return imati navedenu varijablu koja sadrži vrijednost koju vraćaju, to može biti konkretna vrijednost, jedna varijabla, ili matematički izraz. Funkciju koja vraća vrijednost ne možemo pozvati samu za sebe, već njen poziv mora biti dio nekog matematičkog izraza, dio ispisa, ili dio neke druge naredbe. Slijedi primjer:

```
def zbroj(var1,var2):
    return var1+var2

print(zbroj(10,20))

x=zbroj(10,20)*2
print(x)
```

Sve varijable koje su definirane izvan funkcija, globalne su varijable i prepoznatljive su u svakom dijelu programa, dok su varijable definirane unutar funkcija lokalne, i prepoznatljive su samo unutar matične funkcije.

Preopterećivanje funkcija u Pythonu nije potrebno, pošto je Python dovoljno dinamičan da svaka funkcija, već sama po sebi, ima sposobnost primanja varijabilnog broja parametara. Pogledajmo:

```
def operacija(*parametri):
    if len(parametri)==1:
        return parametri[0]/2
    elif len(parametri)==2:
        return parametri[0]*parametri[1]
    elif len(parametri)==3:
        return parametri[0]+parametri[1]+parametri[2]
    else:
        return 0

print(operacija(10))
print(operacija(2,4))
print(operacija(5,10,15))
```

U prethodnom primjeru vidimo kako funkcija, koju smo nazvali operacija, prima nedefiniran broj parametara i pohranjuje ih u n-torku imena parametri. Unutar funkcije zatim ispitujemo duljinu n-torke, te ovisno o broju prosljeđenih parametara izvršavamo različit blok naredbi, odnosno u navedenom primjeru, izvršavamo različitu matematičku operaciju sa primljenim parametrima i vraćamo rezultat te operacije. Izvršavajući pozive ove funkcije prema primjeru, rezultati koje će print naredba ispisati biti će redom 5.0, 8, 30.

Objektno orijentirani pristup

Metode objektno orijentiranog pristupa omogućuju nam stvaranje prototipova koji opisuju svojstva objekata sa kojima želimo raditi. Iz jednog definiranog prototipa mi zatim možemo stvoriti neodređen broj instanci, odnosno kopija tog objekta.

Uzmimo za primjer da stvaramo prototip automobila, te definiramo sve atribute koje taj automobil mora imati (naziv modela, snaga motora, maksimalna brzina, ukupna masa, itd.). Kada smo opisali prototip, mi možemo definirati neodređen broj varijabli koje su tog tipa, odnosno predstavljaju automobile, te imaju konkretizirane vrijednosti svih navedenih svojstava automobila. Drugim riječima sada naše varijable ne predstavljaju samo jednostavne brojeve, ili tekst, već predstavljaju složene objekte sa svim njihovim svojstvima.

Navedimo neke osnovne termine vezane uz objektno orijentirani pristup:

Klasa - je definirani prototip koji opisuje neki objekt, odnosno sadrži popis i definiciju svih atributa i funkcija tog objekta. Npr. klasa Automobil.

Atributi klase - su konkretna svojstva objekta predstavljena kroz varijable određenog tipa. Npr. maksimalna brzina automobila. Svaki pojedini stvoreni automobil može kasnije imati svoju vrijednost pridodanu ovom atributu, npr. 180, 200, 250.

Metode klase - su funkcije koje opisuju kako se pripadni objekt ponaša u određenim situacijama. Npr. funkcije za ubrzavanje i kočenje, mogu ovisno o ulaznim parametrima količina gasa i intenzitet kočenja, mjenjati vrijednost atributa koji predstavlja trenutnu brzinu automobila.

Instance klase - su sve varijable koje su tipa te klase, odnosno pretstavljaju kopije tog prototipa. Npr. automobili a, b i c. Svaki od tih automobila ima svoje vrijednosti atributa klase, npr. svoju vrijednost snage motora, no svi se ponašaju slijedeći ista generalna pravila, odnosno prate pravila definirana u metodama klase.

Nasljeđivanje - je stvaranje strukture klasa, gdje klase koje su niže u stablu, nasljeđuju sve atribute direktno nadređenih klasa. Ukoliko dio različitih klasa koje želimo definirati dijele dio svojstava, da nebismo morali sva ta svojstva nanovo opisivati u svakoj od tih klasa, taj dio svojstava ćemo onda dodjeliti klasi roditelj, a ostale klase definirati ćemo kao djecu te klase, te će time one automatski dobiti sva ta svojstva, a baz da su ona opisana u svakoj od tih klasa. Npr. klasa prijevozno sretstvo ima atribute naziv modela, maksimalna brzina i masa. Klase automobil i motocikl naslijeđuju klasu prijevozno sredstvo, time smo definirali da su i automobil i motocikl prijevozna sredstva, te da oboje imaju i naziv modela, neku maksimalnu brzinu i masu, no imati će različite neke druge karakteristike koje će onda biti opisane u tim klasama. Klasu automobil dalje mogu nasljediti terenac i limuzina, koje će onda imati sve karakteristike automobila kao takvog, no one će se opet međusobno razlikovati nekim svojim atributima.

Rad sa objektima

U slijedećem primjeru definirati ćemo klasu Automobil koja je opisana nazivom modela i maksimalnom brzinom automobila. Nakon toga stvoriti ćemo dva automobila a i b, koji će biti različitih modela i imati će različitu maksimalu brzinu.

```
class Automobil:
    model = ''
    maxbrzina = 0

a = Automobil()
b = Automobil()

a.model = 'Seat Ibiza 1.9 TDI'
a.maxbrzina = 190

b.model = 'VW Polo 1.4'
b.maxbrzina = 160

print (a.model)
print (a.maxbrzina)
print (b.model)
print (b.maxbrzina)
```

Prilikom same definicije klase, te njoj pripadnih atributa, definirali smo i pretpostavljene vrijednosti koje će ti atributi imati, u ovom slučaju prazan string, te brzinu 0. Nakon toga definiramo varijable a i b koje predstavljaju taj automobil, te im u nastavku pridodajemo vrijednosti naziv modela i maksimalne brzine. Ukoliko to nebismo konkretno naveli ovdje, preuzete bi bile pretpostavljene vrijednosti definirane u samoj klasi. Atributima automobila a i b, pristupamo navođenjem imena varijable koja predstavlja taj automobil, zatim točke, te u nastavku imena atributa kojem želimo pristupiti. U takvom obliku, sa njihovim vrijednostima možemo raditi sve što možemo raditi i sa vrijednostima upisanim u klasične varijable.

Stvorimo sada listu automobila:

```
class Automobil:
    model = ''
    maxbrzina = 0

l1 = [Automobil(),Automobil()]

l1[0].model = 'Seat Ibiza 1.9 TDI'
l1[0].maxbrzina = 190

l1[1].model = 'VW Polo 1.4'
l1[1].maxbrzina = 160

print(l1[0].model)
print(l1[0].maxbrzina)
print(l1[1].model)
print(l1[1].maxbrzina)
```

U primjeru je definirana lista koja sadrži dva automobila. Automobilima ćemo pristupati preko imena liste, te indeksa pozicije na kojoj je taj automobil u listi smješten. Sa listom instanci objekata možemo raditi sve što i sa bilo kojom drugom listom, čiji članovi su nekog drugog tipa.

Naravno kada bi smo ovako definirali liste, bilo bi nezgodno definirati listu od 500 automobila. Slijedi primjer gdje korisnik unosi podatke za 5 automobila koji se pohranjuju u listu:

```
class Automobil:
    model = ''
    maxbrzina = 0

l1 = []

for i in range(0, 5):
    l1.append(Automobil())
    l1[i].model = input('Unesi naziv modela: ')
    l1[i].maxbrzina = int(input('Unesi maksimalnu brzinu: '))

for i in range(0, 5):
    print (l1[i].model)
    print (l1[i].maxbrzina)
```

Prethodni primjer možemo modificirati tako da prije petlje korisnika upitamo koliko automobila želi, te da za broj ponavljanja petlje upišemo varijablu koja sadrži taj korisnikov unos, ili možemo for petlju zamjeniti while petljom, te nakon upisa svakog pojedinog automobila pitati korisnika želi li unositi još ili prekinuti unos.

Slijedeći primjer prikazuje kako se definiraju metode unutar klase, te kako se pozivaju i koriste:

```
class Automobil:
    model = ''
    maxbrzina = 0
    trenutnabrzina = 0
    def ubrzaj (self,gas):
        self.trenutnabrzina = self.trenutnabrzina + gas
    def uspori (self,kocnica):
        self.trenutnabrzina = self.trenutnabrzina - kocnica

a = Automobil()
a.model = 'Testni Automobil'
a.maxbrzina = 250
a.trenutnabrzina = 0

while a.trenutnabrzina>=0 and a.trenutnabrzina<=a.maxbrzina:
    print('Vozite se trenutnom brzinom koja iznosi',a.trenutnabrzina)
    print('Spusti li se ispod 0 auto ce se ugasiti!')
    print('Predje li preko',a.maxbrzina,'motor ce eksplodirati!')
    odgovor = int(input('Unesite 1 za UBRZANJE ili 2 za USPORENJE:'))
    if odgovor==1:
        gas = int(input('Koliko ubrzati?:'))
        a.ubrzaj(gas)
    elif odgovor==2:
        kocnica = int(input('Koliko usporiti?:'))
        a.uspori(kocnica)
    else:
        print('Unesite 1 ili 2. Za kraj voznje ugasite ili raznesite auto!')

if a.trenutnabrzina<0:
    print('Auto se ugasio!')
elif a.trenutnabrzina>a.maxbrzina:
    print('Boom!')
```

U prethodnom primjeru klasi automobil dodali smo dvije metode, ubrzaj i uspori. Kao obavezni parametar sve metode klase imaju ključnu riječ self, koja označava sam objekat kojem te metode pripadaju. Self se koristi unutar metode u kombinaciji sa imenima atributa objekta, npr. self.imeatributa, kako bi se naglasilo da se radi sa atributom samog objekta, a ne sa lokalnom varijablom istog imena koja pripada metodi. Uz taj obvezni, metode mogu imati i dodatni neodređeni broj parametara.

U programu, metode nekog objekta pozivamo tako da navedemo ime instance tog objekta, zatim točku, te ime same metode gdje u zagradama ne navodimo self, već samo dodatne parametre, ako metodi nešto prosljeđujemo.

U navedenom primjeru, nakon definicije klase sa pripadnim atributima i metodama, stvaramo automobil a, te definiramo konkretne vrijednosti njegovih atributa, model i maksimalnu brzinu. Zatim se pokreće while petlja koja predstavlja vožnju, korisnik bira hoće li u svakom pojedinom krugu petlje ubrzati ili usporiti automobil, te koliko, nakon čega se pozivaju metode za ubrzanje ili usporenje automobila a, kojima se prosljeđuje količina ubrzanja, odnosno usporenja. Te metode zatim modificiraju trenutnu brzinu automobila za prosljeđeni iznos. Ukoliko se trenutna brzina automobila spusti ispod 0, ili poveća iznad maksimalno moguće, vožnja se prekida.

U slijedećem primjeru klasi automobil dodati ćemo i konstruktor. Konstruktor je metoda koja inicijalizira objekt, odnosno, ta se metoda pokreće čim stvorimo novi objekat, u ovom slučaju novi automobil, bez da smo išta drugo učinili. Kao što će u primjeru biti vidljivo, ukoliko smo na taj način definirali konstruktor, već prilikom stvaranja novih objekata možemo odmah u sklopu inicijalizacije već definirati vrijednosti atributa tog objekta:

```
class Automobil:
    ukupno_automobila = 0
    model = ''
    maxbrzina = 0
    def __init__(self,model,maxbrzina):
        self.model = model
        self.maxbrzina = maxbrzina
        Automobil.ukupno_automobila += 1

a = Automobil('Seat Ibiza 1.9 TDI',190)
b = Automobil('VW Polo 1.4',160)

print('Ukupno smo stvorili',Automobil.ukupno_automobila,'automobila!')
```

Metoda __init__(self) je konstruktor. Njoj smo dodali još parametre model i maxbrzina. Prilikom stvaranja novog automobila u zagradama smo odmah naveli vrijednosti atributa tog automobila, koje su se prosljedile konstruktoru, koji se pokreće odmah prilikom tog stvaranja. Na taj način u automobil su se odmah pohranile prosljeđene vrijednosti. Ovdje također vidimo razliku između self.varijable i varijable, gdje se varijabla koja je prikačena na self odnosi na atribut objekta, a varijabla bez self, odnosi na lokalnu varijablu.

Varijable koje pozivamo ne pomoću imena instance klase, već pomoću imena same klase, točke, te naziva varijable, u našem slučaju Automobil.ukupno_automobila, promjeniti će vrijednost u svim instancama klase, ne samo za jednu instancu. Na ovaj način možemo npr. pratiti koliko instanci neke klase smo već stvorili, kao u primjeru. Naredba del imeinstance, briše instancu.

Funkcijama hasattr (klasa,'atribut'), setattr (klasa,'atribut',vrijednost) i delattr (klasa,'atribut'), ispitujemo postojanje atributa u klasi, stvaramo nove, ili brišemo postojeće atribute.

Iznimke

Procesiranje iznimaka je programski mehanizam osmišljen da upravlja pojavama posebnih okolnosti koje se mogu desiti prilikom izvođenja programa, i prekinuti njegov normalni tok.

Generalno, to se čini tako da se pohrani trenutno stanje izvođenja na neko preddefinirano mjesto u memoriji, te se izvršenje tog dijela programa prebaci u potproceduru, koju još nazivamo rukovoditeljem iznimki. Izvođenjem, tog dijela programa u podproceduri, sprječava se rušenje samog programa u situacijama kada bi taj dio koda to inače prouzročio, te se umjesto toga korisniku javlja poruka o pogrešci, uz mogućnost ponavljanja izvođenja tog djela sa drugim parametrima ili nastavak programa sa zaobilaženjem tog dijela.

Do pogrešaka ne mora nužno doći radi pogrešno napisanog koda, već postoji mnogo uvjeta koji mogu prouzročiti rušenje programa, npr. neodgovarajući korisnički unos prilikom izvođenja programa. Takve se problematične situacije mogu preduhitriti i spriječiti korištenjem procesiranja iznimaka.

Sa programskog stajališta, dio koda za koji vjerujemo da bi prilikom nekorektne upotrebe, od strane korisnika ili sustava, mogao prouzročiti rušenje programa ili njegov nepravilan rad, obuhvatiti ćemo iznimkom, to znači da će se prilikom izvođenja programa, taj dio koda najprije izvesti u posebnom mjestu u memoriji, te će se ukoliko je sve prošlo uredu program normalno nastaviti, a ukoliko se ustanovi da je došlo do pogreške, izvest će se ono što smo mi u programskom kodu napisali da će se u tom slučaju desiti.

Dakle mi željeni dio koda obuhvaćamo tjelom iznimke, nakon čega dopisujemo dodatni kod koji definira što se dešava u posebnim situacijama. Uzmimo za primjer situaciju gdje radimo sa listom podataka, u toku izvođenja programa korisnik briše članove liste i lista ostane prazna. Nakon toga korisnik pokuša ispisati članove liste. Naravno pošto je lista prazna, program bi javio pogrešku. Za primjer definirajmo praznu listu, i pokušajmo ispisati njenog prvog člana:

```
li = []
print (li[0])
```

Ovo će javiti pogrešku:

```
IndexError: list index out of range
```

Sada ovu print naredbu, za koji smo ustanovili da u određenim situacijama može uzrokovati pogrešku, obuhvatimo u iznimku, te definirajmo poruku koja ce se u takvoj situaciji ispisati korisniku umjesto rušenja programa:

```
li = []

try:
    print(li[0])
except:
    print('Nemogu izvrsiti ispis!')
```

U navedenom primjeru vidimo da dio koda koji može uzrokovati pogreške ubacujemo u try blok, dok u except bloku definiramo sto se desava u slucaju pogreske.

Modificirajmo sada malo ovaj kod da dobijemo detaljniji opis pogreške. Vidjeli smo ranije iz originalne poruke o pogrešci da je riječ o grešci koju je Python nazvao IndexError. U slijedećem primjeru ulovit ćemo informaciju o toj konkretnoj pogrešci u varijablu e, te ispisati korisniku točan opis pogreške:

```
li = []

try:
    print(li[0])
except IndexError as e:
    print('Opis greske:',e)
```

Pogreške se mogu javiti zbog mnogo razloga, hvatanjem samo određenog tipa iznimke lovimo pogrešku samo ako je te navedene vrste, dok druge pogreške nećemo uloviti. Ukoliko želimo uloviti više vrsti pogrešaka, i u slučaju različite vrste pogreške učiniti različitu stvar, možemo dodati više except blokova:

```
try:
    # neki kod
except SomeError as e1:
    print('Opis greske:',e1)
    # napravi nesto
except SomeOtherError as e2:
    print('Opis greske:',e2)
    # napravi nesto drugo
```

Ukoliko nismo sigurni o kojem tipu pogreške se radi, možemo uloviti bilo koju iznimku pomoću ključne riječi Exception koja obuhvaća sve vrste pogrešaka:

```
li = []

try:
    print(li[0])
except Exception as e:
    print('Opis greske:',e)
```

Ponekad, uloviti samo jedan dio koda u iznimku nije dovoljno:

```
try:
    a = int(input('Unesi umanjenik: '))
    b = int(input('Unesi umanjitelj: '))
except Exception as e1:
        print ('Opis pogreske:',e1)

print('Rezultat iznosi:',a-b)
```

U prethodnom primjeru provjerili smo da li je korisnik zbilja unio brojeve u varijable a i b, i ako nije javili smo poruku o pogrešci, no naredba print(a-b) koja slijedi nakon toga se izvrši uvjek, čak i ako a ili b nisu dobru uneseni, pa bi u tom slučaju ta naredba uzrokovala pogrešku. Tome ćemo doskočiti tako da definiramo da se ta naredba izvršava samo ukoliko su i a i b ispravno uneseni kao brojevi. To ćemo učiniti na slijedeći način:

```
try:
    a = int(input('Unesi umanjenik: '))
    b = int(input('Unesi umanjitelj: '))
except Exception as e1:
        print ('Opis pogreske:',e1)
else:
    print('Rezultat iznosi:',a-b)
```

U prethodnom primjeru kažemo, pokušaj unjeti a i b, ukoliko nisu uneseni kako treba javi informaciju o pogrešci, a inače (ako jesu) oduzmi ih i ispiši rezultat.

Rad sa datotekama

Najčešće će rezutate biti dovoljno prikazati samo na ekranu, no ukoliko se ukaže potreba za trajnijom pohranom tih podataka, kako bi oni mogli biti korišteni i naknadno, možemo ih upisati i u datoteku.

Otvaranje i zatvaranje datoteka

Da bi smo mogli koristiti datoteku, bilo za pisanje ili za čitanje, moramo ju najprije otvoriti. To ćemo učiniti na slijedeći način:

```
moja_datoteka = open('naziv_datoteke' , 'nacin_otvaranja')
```

Varijablu moja_datoteka koristiti ćemo u programu kako bi smo imenovali sa kojom datotekom nešto radimo, pošto je moguće da imamo više datoteka odjednom otvoreno. String naziv_datoteke odnosi se na fizički naziv datoteke na disku, a string nacin_otvaranja na način na koji je datoteka otvorena, npr. za čitanje ili za upisivanje podataka (mogući načini otvaranja navedeni su u tebeli koja slijedi).

Nakon što smo varijabli u programu doznačili neku datoteku na disku, bilo već od ranije postojeću ili tek stvorenu, podatke o toj datoteci možemo saznati na slijedeći način:

```
print ('Fizičko ime datoteke na disku je: ', moja_datoteka.name)
print ('Da li je datoteka zatvorena: ', moja_datoteka.closed)
print ('Način u kojem je datoteka otvorena: ', moja_datoteka.mode)
```

Pomoću prethodnih naredbi možemo saznati koja je konkretno datoteka na disku pridodana određenoj varijabli, da li je ta datoteka zatvorena, te ukoliko nije, na koji način je ona otvorena.

Ukoliko u programu radimo sa više fizičkih datoteka odjednom, morati ćemo ih sve otvoriti, te svaku fizičku datoteku doznačiti drugoj varijabli u programu.

Naredba `moja_datoteka.close()` zatvara datoteku nakon čega se u nju ne može više pisati, dok se postupak otvaranja datoteke ne izvrši ponovno.

Način otvaranja datoteke može biti:

Način	Opis
r	Otvara datoteku samo za čitanje. Pokazivač je inicijalno smješten na početku datoteke. U datoteku se ne može pisati.
rb	Otvara datoteku za binarno čitanje. Pokazivač je inicijalno smješten na početku datoteke. U datoteku se ne može pisati.
r+	Otvara datoteku za čitanje i pisanje. Pokazivač je inicijalno smješten na početku datoteke. Ako se počne pisati bez pomicanja pokazivača, postojeći sadržaj se prepisuje.
rb+	Otvara datoteku za binarno čitanje i pisanje. Pokazivač je inicijalno smješten na početku datoteke. Ako se počne pisati bez pomicanja pokazivača, postojeći sadržaj se prepisuje.
w	Otvara datoteku samo za pisanje. Prepisuje datoteku ako datoteka sa tim nazivom već postoji. Ako datoteka ne postoji kreira novu.
wb	Otvara datoteku za binarno pisanje. Prepisuje datoteku ako datoteka sa tim nazivom već postoji. Ako datoteka ne postoji kreira novu.
w+	Otvara datoteku za pisanje i čitanje. Prepisuje datoteku ako datoteka sa tim nazivom već postoji. Ako datoteka ne postoji kreira novu.
wb+	Otvara datoteku za binarno pisanje i čitanje. Prepisuje datoteku ako datoteka sa tim nazivom već postoji. Ako datoteka ne postoji kreira novu.
a	Otvara datoteku samo za dodavanje sadržaja. Ako datoteka postoji pokazivač je inicijalno smješten na kraju datoteke. Ako datoteka ne postoji otvara se nova datoteka samo za pisanje.
ab	Otvara datoteku za binarno dodavanje sadržaja. Ako datoteka postoji pokazivač je inicijalno smješten na kraju datoteke. Ako datoteka ne postoji otvara se nova datoteka za binarno pisanje.
a+	Otvara datoteku za dodavanje i čitanje. Ako datoteka postoji pokazivač je inicijalno smješten na kraju datoteke. Ako datoteka ne postoji otvara se nova datoteka za pisanje i čitanje.
ab+	Otvara datoteku za binarno dodavanje i čitanje. Ako datoteka postoji pokazivač je inicijalno smješten na kraju datoteke. Ako datoteka ne postoji otvara se nova datoteka za pisanje i čitanje.

Pohrana podataka u tekstualne datoteke

Otvorimo datoteku za pisanje i zapišimo nešto u nju:

```
moja_datoteka = open('datoteka.txt' , 'w')

moja_datoteka.write('Ovo je prva linija teksta u datoteci!')
moja_datoteka.write('Ovo je nastavak prve linije teksta, bez razmaka!')
moja_datoteka.write('\n')
moja_datoteka.write('Ovo je druga linija teksta u datoteci!')

moja_datoteka.close()
```

U prethodnom primjeru koristili smo 'w' način da otvorim datoteku za zapisivanje podataka u nju, nakon čega smo koristili naredbu write kako bi smo u nju zapisali navedeni tekst. Tekst se zapisuje u prvu liniju u datoteci sve dok konkretno ne zapišemo kraj linije '\n'.

U navedenom primjeru u datoteku smo zapisali dvije linije teksa, nakon čega smo datoteku zatvorili.

Datoteka kao standardan ulaz prepoznaje tekst, dakle string, a ne brojeve. Što bi se desilo kada bismo u datoteku na ovaj način pokušali zapisati brojevne vrijednosti?

```
moja_datoteka = open('datoteka.txt' , 'w')

a=int(input('Broj 1:'))
b=int(input('Broj 2:'))

moja_datoteka.write(a)
moja_datoteka.write(b)

moja_datoteka.close()
```

Kao rezultat ovakvog pokušaja upisa, program bi nam javio:

```
TypeError: must be str, not int
```

Dakle, da bi funkcionirao ispravno, navedeni program moramo modificirati na način da prije upisa u datoteku, brojevne vrijednosti pretvorimo u string. Također, dodati ćemo i oznaku za kraj linije iza svakog broja, kako nam znamenke brojeva nebi bili zapisane spojeno:

```
moja_datoteka = open('datoteka.txt' , 'w')

a=int(input('Broj 1:'))
b=int(input('Broj 2:'))

a=str(a)
b=str(b)

moja_datoteka.write(a)
moja_datoteka.write('\n')
moja_datoteka.write(b)
moja_datoteka.write('\n')

moja_datoteka.close()
```

U prethodnom primjeru, prije zapisa brojevnih vrijednosti u datoteku, te smo vrijednosti konvertirali u string naredbom `a=str(a)` za varijablu a, te ekvivalentnom naredbom za varijablu b. Nakon svakog broja, zapisali smo i oznaku za kraj linije.

Pogledajmo primjer kombiniranog zapisa. Na kraju će u datoteci sve biti u tekstualnom obliku:

```
moja_datoteka = open('datoteka.txt' , 'w')

ime=input('Kako se zoveš?:')
godine=int(input('Koliko imaš godina?:'))

godine=str(godine)

moja_datoteka.write('Ti se zoveš ')
moja_datoteka.write(ime)
moja_datoteka.write(' i imaš ')
moja_datoteka.write(godine)
moja_datoteka.write(' godina!\n')

moja_datoteka.close()
```

U prethodnim primjerima otvarali smo datoteku u 'w' načinu, čime smo uvjek početkom zapisivanja stvarali novu datoteku, te smo moguću već postojeću datoteku sa tim nazivom prepisivali. Želimo li u datoteku sa postojećim sadržajem, nadodati novi sadržaj, bez brisanja starog sadržaja, datoteku moramo otvoriti u 'a' načinu:

```
moja_datoteka = open('datoteka.txt' , 'a')

novi_tekst=input('Unesi tekst koji će se nadodati u datoteku:')

moja_datoteka.write(novi_tekst)
moja_datoteka.write('\n')

moja_datoteka.close()
```

Pošto je u ovakvom otvaranju datoteke pokazivač za unos inicijalno smješten na kraju datoteke, uneseni će se sadržaj zapisati na kraj datoteke, iza već postojećeg sadržaja.

Čitanje podataka iz tekstualnih datoteka

Prilikom čitanja podataka iz tekstualnih datoteka moramo voditi računa o tome da će sve što pročitamo biti u tekstualnom obliku, pa čak i moguće brojevne vrijednosti koje se tamo nalaze. Kao što smo, prilikom zapisa u tekstualnu datoteku, brojevne vrijednosti morali prethodno pretvoriti u string, tako ćemo sada pročitane vrijednosti koje su brojevi zapisani u tekstualnom obliku, morati iz stringa pretvoriti u brojevnu vrijednost.

Za početak pogledajmo jednostavan primjer čitanja sadržaja iz tekstualne datoteke:

```
moja_datoteka = open('datoteka.txt' , 'r')

sadrzaj_datoteke = moja_datoteka.read()
print(sadrzaj_datoteke)

moja_datoteka.close()
```

U prethodnom primjeru najprije smo otvorili datoteku u 'r' načinu koji dozvoljava samo čitanje iz datoteke, zatim smo definirali varijablu sadrzaj_datoteke te joj pridodali rezultat funkcije `read()` koja učita cjeli sadržaj datoteke. Ovim postupkom dobili smo string varijablu koja sadrži cijeli sadržaj tekstualne datoteke. Krajevi linija označeni su oznakama za kraj linije '\n'.

Kako bi smo učitali samo jednu liniju teksta umjesto cijelog sadržaja datoteke?

```
moja_datoteka = open('datoteka.txt' , 'r')

linija_teksta = moja_datoteka.readline()
print(linija_teksta)

moja_datoteka.close()
```

Funkcija `readline()` učitava samo onaj sadržaj koji se nalazi između trenutne pozicije pokazivača u datoteci i prve slijedeće oznake za kraj linije.

Želimo li učitati cijeli sadržaj od trenutne pozicije pokazivača do kraja datoteke, no ne želimo da se sve pohrani u jednu string varijablu, možemo koristiti funkciju `readlines()` koja kreira listu, te svaki element proizvedene liste bude jedna učitana linija iz datoteke:

```
moja_datoteka = open('datoteka.txt' , 'r')

lista_linija = moja_datoteka.readlines()
print(lista_linija)

moja_datoteka.close()
```

Ispišimo sada liniju po liniju, koje su pohranjene u listi, pristupajući svakom članu liste zasebno i ispisujući ga:

```
moja_datoteka = open('datoteka.txt' , 'r')

lista_linija = moja_datoteka.readlines()
broj_linija = lista_linija.__len__()

print('Broj linija u datoteci je:',broj_linija,'\n')

for i in range(0,broj_linija):
    print('Linija',i+1,':',lista_linija[i])

moja_datoteka.close()
```

U navedenom primjeru vidimo kako, nakon otvaranja datoteke, te korištenja funkcije `readlines()` kako bismo pohranili linije teksta iz datoteke u listu, prvo provjeravamo koliko ta lista ima članova, što je u stvari broj linija teksta u datoteci, te pomoću for petlje čiji indeks iterira od 0 do broja linija, odnosno broja članova liste (podsjetnik na for petlju, iteracija u stvari ide do navedenog maksimuma – 1), ispisujemo svaki član liste, odnosno svaku liniju teksta zasebno. Na ovaj način možemo pristupiti i samo određenim linijama u datoteci, te vršiti određene funkcije nad njima, ignorirajući ostatak sadržaja datoteke.

Kako bismo pristupili svakom znaku u datoteci posebno? Tako da najprije pristupimo svakoj liniji kao članu liste, odnosno zasebnom stringu, te zatim iteriramo kroz taj string. Pogledajmo primjer:

```
moja_datoteka = open('datoteka.txt' , 'r')

lista_linija = moja_datoteka.readlines()
broj_linija = lista_linija.__len__()

for i in range(0,broj_linija):
    trenutna_linija = lista_linija[i]
    for j in range(0,trenutna_linija.__len__()):
        print(trenutna_linija[j])

moja_datoteka.close()
```

U prethodnom primjeru najprije smo pročitali sadržaj datoteke, zatim smo njen sadržaj, liniju po liniju, pohranili u listu lista_linija. Nakon toga, iterirali smo kroz tu listu, i čitali svaki njen član, odnosno liniju teksta, zasebno, te svaku liniju pohranili u privremenu varijablu trenutna_linija. U unutarnjoj petlji, iteriramo kroz svaku liniju zasebno, znak po znak, te ga u ovom primjeru samo ispisujemo, no ovdje možemo uključiti i razne druge postupke.

Iskoristimo sada mogućnost pristupa svakom znaku unutar linije, da uz provjere gdje se nalaze prazna mjesta i krajevi linija, stvorimo listu svih zasebnih riječi u datoteci, te ih ispisati:

```
moja_datoteka = open('datoteka.txt' , 'r')

lista_linija = moja_datoteka.readlines()
broj_linija = lista_linija.__len__()

lista_rijeci = []

for i in range(0,broj_linija):
    trenutna_linija = lista_linija[i]
    trenutna_rijec = ''
    for j in range(0,trenutna_linija.__len__()):
        if trenutna_linija[j] != ' ' and trenutna_linija[j] != '\n':
            trenutna_rijec = trenutna_rijec + trenutna_linija[j]
        elif trenutna_rijec.__len__() > 0:
            lista_rijeci.append(trenutna_rijec)
            trenutna_rijec = ''

for i in lista_rijeci:
    print(i)

moja_datoteka.close()
```

U navedenom primjeru, prilikom pristupa svakoj liniji teksta, u varijablu trenutna_rijec pohranjujemo sve slijedne znakove dok ne stignemo do prvog slijedećeg razmaka ili kraja linije, te zatim taj niz znakova (odnosno jednu riječ ili broj) pohranjujemo u listu riječi koju smo definirali na početku kao praznu listu. Na ovaj način u listu riječi pohranjeni su svi zasebni dijelovi teksta iz datoteke, koji su kompaktni i međusobno razdvojeni razmacima.

Dijelovi teksta koje smo u prethodnom primjeru razdvojili mogu biti riječi ili brojevi zapisani u tekstualnom obliku. Idemo sada razdvojiti tu listu, na liste riječi, cijelih i decimalnih brojeva:

```
lista_cijelih_brojeva = []
lista_decimalnih_brojeva = []
lista_obicnih_rijeci = []

for i in lista_rijeci:
    if i.isdigit():
        lista_cijelih_brojeva.append(i)
    else:
        try:
            float(i)
        except:
            lista_obicnih_rijeci.append(i)
        else:
            lista_decimalnih_brojeva.append(i)

for i in lista_cijelih_brojeva:
    print('Cijeli broj:',i)
for i in lista_decimalnih_brojeva:
    print('Decimalni broj:',i)
for i in lista_obicnih_rijeci:
    print('Obična riječ:',i)
```

Navedeni primjeri prikazuju kako pristupati svim zasebnim podatcima u tekstualnoj datoteci.

Pohrana objekata u datoteke

Ukoliko u datoteku želimo pohraniti strukturirane podatke, možemo se poslužiti postupkom prezervacije objekata, tzv. pickling-om. „Pickling“ je proces u kojem se struktura objekta pretvara u niz byte-ova. Taj niz byte-ova zatim možemo pohraniti u binarnu datoteku, te ga naknadno u bilo kojem trenutku pročitati. Na taj način možemo u datoteke pohranjivati objekte, te ih naknadno, kada se za time ukaže potreba, učitavati iz datoteka i ponovno koristiti.

Da bi smo se mogli služiti funkcijama picklinga, moramo u početku programa uključiti modul pickle. Da bi smo to učinili napisati ćemo slijedeću liniju:

```
import pickle
```

Što možemo pohraniti na ovaj način? Pohraniti možemo osnovne logičke vrijednosti true i false, cijele, decimalne i kompleksne brojeve, stringove, byte-ove, nizove byte-ova, n-torke, liste i riječnike koje sadrže prethodno navedene tipove podataka.

Iako u narednom primjeru u datoteku pohranjujemo samo jedan objekt, taj jedan objekt je lista koja sadrži cjeli niz podataka. Pogledajmo primjer kako pohraniti nešto u datoteku na takav način:

```
lista = ['Python',10,'Ima li pilota u avionu?',15.5]

moja_datoteka = open('datoteka.txt' , 'wb')

pickle.dump(lista, moja_datoteka)

moja_datoteka.close()
```

U navedenom primjeru u datoteku, koju smo otvorili za binarno zapisivanje, pohranili smo listu koja sadrži nekoliko članova, služeći se funkcijom `pickle.dump(objekt, naziv_datoteke)` dostupnom iz pickle modula. Idemo sada pogledati kako ćemo tu listu nekom drugom prilikom moći učitati iz datoteke, i ponovno koristiti:

```
moja_datoteka = open('datoteka.txt' , 'rb')

dohvacena_lista = pickle.load(moja_datoteka)

moja_datoteka.close()

print(dohvacena_lista)
```

U ovom primjeru, datoteku smo otvorili za binarno čitanje, te služeći se funkcijom `pickle.load(naziv_datoteke)` učitali objekt koji je u datoteci bio pohranjen i doznačili ga definiranoj varijabli u programu.

Kada se pickle postupak koristi u mrežnim aplikacijama, ukoliko se pickle-ani podatci čitaju iz datoteke sa udaljene lokacije koja nije sigurna, te se prevode na lokalnom računalu, postoji mogućnost da se iz datoteke isčita pickle-ani maliciozni kod, koji zatim prilikom prevođenja može učiniti štetu na sustavu računala. Prilikom udaljenog rada treba voditi računa o sigurnosti.