Osnove distribuiranog programiranja

Praktikum

2023/2024

Profesor:

Imre Lendak

Asistenti:

Zorana Babić

Jelena Sekulić

Uvod

Bodovanje:

- Vežbe: 2 kolokvijuma po 30 bodova
- Test: 20 bodova
- Usmeni: 20 bodova

Plan rada:

26.02-01.03.	1. Uvod u programski jezik Python
04.03-08.03.	2. Kontrola toka i funkcije
11.03-15.03.	3. Klase i rad sa datotekama
18.03-22.03.	4. Klijent-server arhitektura
25.03-28.03. + 05.04.	5. Bezbednosni mehanizmi
20.04.	**6. Prvi kolokvijum**
22.04-26.04.	7. Replikacija
07.05-10.05.	8. Otpornost na otkaze
13.05-17.05.	9. Priprema za drugi kolokvijum
25.05.	**10. Drugi kolokvijum**
08.06.	**11. Popravni kolokvijum (prvi ILI drugi)**

Organizacija vežbi:

- Komunikacija isključivo putem mejla (zbabic@uns.ac.rs, jelena.sekulic@uns.ac.rs)
- Prisustvo vežbama je obavezno, dozvoljena su **maksimalno** 2 izostanka
- Neophodno je položiti oba kolokvijuma, minimum bodova na svakom je 15.5
- U terminu popravnog je moguće raditi samo jedan kolokvijum
- Konsultacije se održavaju u sledećim fiksnim terminima:
    - Zorana: četvrtak, 13-14h, kancelarija: NTP225
    - Jelena: četvrtak, 19.30-20.30h, kancelarija: NTP215

Sadržaj

Uvod u *Python*

Python je programski jezik visokog nivoa, opšte namene i velike popularnosti. Kreirao ga je holandski programer Gvido van Rosum (*Guido van Rossum*) 1991. godine. Od tada je našao primenu u razvoju veb aplikacija, kompjuterskih igara i drugog softvera, analizi i vizuelizaciji podataka, automatizaciji zadataka i mašinskom učenju. Može se koristiti na različitim platformama – *Windows*, *Linux*, *Mac*, ... Besplatan je i otvorenog koda. Ima jednostavnu sintaksu, nalik engleskom jeziku, koja omogućava pisanje programa u manje linija koda nego u drugim programskim jezicima (primer). Podržava proceduralnu, funkcionalnu i objektno-orijentisanu programsku paradigmu. Takođe je i interpretirani jezik, što znači da izvršava kod liniju po liniju. Time je znatno olakšano pronalaženje grešaka, ali je i izvršavanje programa usporeno. Ipak, čitljivost, jednostavnost i bogata standardna biblioteka (uz brojna proširenja), razlozi su zbog kojih je ovaj programski jezik omiljeni kod sve većeg broja programera, etičkih hakera i analitičara podataka.

Radno okruženje

Python kod može da se piše u tekstualnom editoru ili u nekom integrisanom razvojnom okruženju (*Integrated Development Environment* – IDE). *PyCharm* je jedno od popularnijih takvih okruženja, koje se može instalirati na *Windows*, *Linux* i *Mac* operativne sisteme, a postoje i brojna *online* okruženja, kao npr. *GeeksForGeeks*, *Programiz*, *OnlinePython*, itd. Često se koristi i veb-bazirano interaktivno okruženje – *Jupyter*.

Na ovom kursu će se koristiti *Visual Studio Code* – besplatni editor koda, koji kombinuje jednostavnost tekstualnog editora sa moćnim mehanizmima za razvoj programa. Uz podršku za mnogobrojne programske jezike, omogućava lakše pronalaženje i ispravljanje sintaksnih, ali i semantičkih grešaka, automatski vodi računa o poravnanju/uvlačenju koda, pruža mogućnost debagovanja i ima ugrađenu podršku za *Git*.

Podešavanje

1. **Instaliranje *Visual Studio Code*-a.**
   Nakon skidanja odgovarajuće verzije sa sajta: https://code.visualstudio.com/download, potrebno je ispratiti nekoliko jednostavnih instalacionih koraka (slike 1-8).
2. **Instaliranje *Python*-a**.
   Nakon skidanja odgovarajuće verzije sa sajta: https://www.python.org/downloads/windows/ dovoljno je instalirati *Python* sa podrazumevanim podešavanjima (slike 9-11).
3. **Instaliranje ekstenzije za *Python* u *Visual Studio Code*-u.**
   Do ekstenzije se može doći putem sledećeg linka: https://marketplace.visualstudio.com/items?itemName=ms-python.python ili direktno iz *Visual Studio Code*-a putem kartice za ekstenzije, pretragom po ključnoj reči '*Python*' (slika 12).

Napomena

*Većina Linux distribucija dolazi sa preinstaliranim Python-om. On se u terminalu pokreće komandom **python3** (**python** za ranije verzije).*

Vežba 1

Pokretanje programa

Za pisanje koda u programskom jeziku *Python*, potrebno je kreirati novi dokument i sačuvati ga sa proizvoljnim nazivom i ekstenzijom *.py*. Izvršavanje jedne naredbe ili niza selektovanih naredbi može se pokrenuti pritiskom tastera *Shift* i *Enter*. Rezultat odabranog bloka koda će biti vidljiv u *Terminalu*. Izvršavanje cele skripte pokreće se klikom na znak trougla u gornjem desnom uglu prozora. Vizuelni prikaz se može naći na slici 13.

Sintaksa jezika

Formatiranje koda

U *Python*-u se svaka naredba završava karakterom za prelazak u novi red. Više takvih naredbi čini blok koda. Za razliku od npr. C, C#, Java jezika, *Python* ne definiše blok vitičastim zagradama (**{ }**), nego isključivo uvlačenjem koda. Naredbe koje se nalaze u istom bloku moraju biti uvučene jednak broj belih karaktera (razmaka) od početka linije, dok se novi (ugnježdeni) blok dodatno uvlači na desno.

Primer 1

```
jezik = 'python'

if jezik == 'python':
    print('Radimo python!')
else:
    print('Nepoznat jezik.')
print('Gotovo!')
```

Kako *if* i *else* naredbama počinju novi blokovi, tako su prve dve naredbe za ispis (*print*) uvučene na desno. Poslednja naredba za ispis nije uvučena, što znači da ne pripada *else* bloku. (Vidi: Kontrola toka)

Primer 2

```
j = 1
while(j <= 5):
    print(j)
    j = j + 1
```

Ukoliko se navodi više naredbi u istom bloku, sve moraju biti uvučene do istog nivoa. U ovom primeru se i naredba za ispis i uvećavanje promenljive *j* izvršavaju u svakoj iteraciji *while* petlje. (Vidi: While petlja)

Komentari

Komentari predstavljaju korisne informacije koje programeri ostavljaju onima koji će čitati njihov kod, sa ciljem da povećaju čitljivost i nivo razumevanja koda. Komentari ne utiču na izvršavanje programa, pa se tako često tokom pisanja koda i njegovog testiranja neki delovi „zakomentarišu“, kako se ne bi izvršavali.

Kao i u većini programskih jezika, tako i u *Python*-u razlikujemo jednolinijske i višelinijske komentare. Jednolinijski komentari počinju znakom **#** i mogu se navesti kako na početku jedne linije, tako i na

kraju, nakon neke naredbe. Korisni su za kratko objašnjenje deklarisane promenljive, funkcije ili nekog izraza.

Primer 3

```
# Ovo je komentar
print("Komentar")
```

Primer 4

```
a, b = 1, 3  # Deklarisanje dve promenljive
zbir = a + b  # Sabiranje dve promenljive
print(zbir)  # Ispis rezultata
```

Primer 5

```
#print("Zdravo, svete!")
print("Zdravo, prijatelju!")
```

Prva naredba za ispis se neće izvršiti.

Višelinijski komentari se pišu jednostavnim ponavljanjem znaka **#** u svakom narednom redu komentara ili navođenjem teksta komentara između tri znaka navoda (**"""**) sa obe strane.

Primer 6

```
# Ovo je primer
# viselinijskog
# komentara

"""
Ovo je primer
viselinijskog
komentara
"""
```

Promenljive

Promenljiva (varijabla, engl. *variable*) služi za skladištenje vrednosti, koja se može menjati tokom izvršavanja programa. Promenljive se imenuju po pravilima koja variraju od jednog do drugog programskog jezika. Međutim, zajedničko pravilo jeste da bi promenljiva trebalo da nosi smislen naziv, koji će što jasnije ukazivati na njenu ulogu u kodu. Tako i u *Python*-u mogu postojati promenljive *x, y, a, b,* ali i promenljive tipa *boja, lista, suma* itd.

Ono što je važno jeste da ime promenljive počinje slovom ili donjom crtom (_), a nikako brojem. U ostatku imena dozvoljena su sva slova (A-z), kao i sve cifre (0-9), a od specijalnih karaktera samo donja crta. Dalje je važno zapamtiti da *Python***pravi** razliku između malih i velikih slova, pa bi tako 'a' i 'A' mogla biti imena dve različite promenljive. Ukoliko se naziv promenljive sastoji iz više reči, one će se pisati spojeno, pri čemu će svaka reč počinjati velikim slovim (pri čemu prva reč može počinjati i malim) ili će sve reči počinjati malim slovom, ali će biti razdvojene donjom crtom (*mojaPromenljiva*, *MojaPromenljiva* ili *moja_promenljiva*).

U *Python*-u nije potrebno posebno deklarisati promenljivu, kao ni specificirati joj tip. Promenljiva će se kreirati onda kada joj se prvi put dodeli neka vrednost i biće tipa upravo te dodeljene vrednosti. Više o naredbi dodele i tipovima podataka nalazi se u nastavku.

Operatori

Operatori su specijalni karakteri koju služe za vršenje operacija nad vrednostima i promenljivama.

Podela operatora u Python-u je sledeća:

- Aritmetički operatori
- Operatori dodele
- Operatori poređenja
- Logički operatori
- Operatori identiteta
- Operatori pripadnosti (engl. *membership operators*)
- Operatori bita (engl. *bitwise operators*)

Aritmetički operatori

Aritmetički operatori služe za obavljanje prostih matematičkih funkcija.

Operator	Operacija	Primer
+	Sabiranje	x + y
-	Oduzimanje	x - y
*	Množenje	x * y
/	Deljenje	x / y
%	Moduo (ostatak pri deljenju)	x % y
**	Stepenovanje	x ** y
//	Celobrojno deljenje (sa zaokruživanjem na dole)	x // y

Operatori dodele

Dodela vrednosti nekoj promenljivoj se vrši pomoću operatora dodele. Osnovni operator dodele jeste znak jednakosti (=), koji se može kombinovati sa aritmetičkim operacijama i operacijama nad bitima. Takođe, moguće je izvršiti višestruku dodelu vrednosti jednom naredbom dodele.

Operator	Primer	Isto kao
=	x = 5	x = 5
=	x, y, z = 5, 6, 7	x = 5 y = 6 z = 7

+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
//=	x //= 3	x = x // 3
**=	x **= 3	x = x ** 3
&=	x &= 3	x = x & 3
|=	x |= 3	x = x | 3
^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3

Operatori poređenja

Operatori poređenja služe za poređenje dve vrednosti.

Operator	Naziv	Primer
==	Jednakost	x == y
!=	Nejednakost	x != y
>	Veće od	x > y
<	Manje od	x < y
>=	Veće od ili jednako	x >= y
<=	Manje od ili jednako	x <= y

Logički operatori

Logičkim operatorima se kreiraju logički izrazi, koje se uglavnom koriste kao uslovne naredbe.

Operator	Opis	Primer
and	Vraća *True* ako su oba iskaza tačna	x < 5 and x < 10
or	Vraća *True* ako je barem jedan od iskaza tačan	x < 5 or x < 4
not	Negira rezultat, vraća *False* ako je iskaz tačan	not(x < 5 and x < 10)

Operatori identiteta

Operatori identiteta služe za poređenje objekata, ali ne po vrednosti, nego po tome da li su u pitanju zaista isti objekti, koji se nalaze na istoj memorijskoj lokaciji.

Operator	Opis	Primer
is	Vraća *True* ako su obe promenljive isti objekat	x is y
is not	Vraća *True* ako su promenljive različiti objekti	x is not y

Primer 7

```
x=5
y=5
x is y # Vraca True

list1 = []
list2 = []
list1 is list2 # Vraca False

list1 = list2
list1 is list2 # Vraca True
```

Operatori pripadnosti

Operatori pripadnosti proveravaju da li neka vrednost pripada određenom objektu. Najčešće se koriste za proveru postojanja karaktera unutar nekog teksta (*stringa*) ili za proveru pripadnosti objekata nekoj kolekciji.

Operator	Opis	Primer
in	Vraća *True* ako je vrednost promenljive sadržana u navedenom objektu	x in y
not in	Vraća *True* ako vrednost promenljive nije sadržana u navedenom objektu	x not in y

Primer 8

```
"a"in"Ana"# Vraca True
"b"in"Ana"# Vraca False
"b"notin"Ana"#Vraca True
1in [1, 2, 3] # Vraca True
4notin [1, 2, 3] # Vraca True
```

Operatori bita

Za operacije nad bitima, koriste se bitni operatori (engl. *bitwise operators*) ili operatori bita.

Operator	Naziv	Opis	Primer
&	AND	Ukoliko su oba bita 1, rezultujući bit je takođe 1	x & y
|	OR	Ukoliko je barem jedan bit 1, rezultujući bit	x | y

		je takođe 1	
^	XOR	Ukoliko je samo jedan bit 1, rezultujući bit je takođe 1	x ^ y
~	NOT	Negiranje (invertovanje) svih bita	~x
<<	Levo pomeranje (*left shift*)	Ubacivanje određenog broja nula (u ovom primeru 2) sa desne strane i pomeranje ostalih bita na levo, pri čemu poslednji biti (u ovom primeru opet 2) "otpadaju"	x << 2
>>	Desno pomeranje (*right shift*)	Ubacivanje određenog broja nula (u ovom primeru 2) sa leve strane i pomeranje ostalih bita na desno, pri čemu poslednji biti (u ovom primeru opet 2) "otpadaju"	x >> 2

Primer 9

```
a = 2   # 00000010
b = 3   # 00000011
a&b # 00000010 - Vraca 2
a|b # 00000011 - Vraca 3
a^b # 00000001 - Vraca 1
~a  # 11111101 - Vraca -3
a <<2  # 0001000 - Vraca 8
a >>2  # 0000000 - Vraca 0
```

Zadaci

1. Dodeliti vrednost sledećeg izraza promenljivoj x: $5^2 + \frac{3}{4}$
2. Neka je a = 17, a b = 3. Odrediti količnik i ostatak pri deljenju a sa b. Zapisati ih u promenljive *kolicnik i ostatak*.
3. Celobrojno podeliti *kolicnik* sa 2, a zatim rezultat uvećati tri puta.
4. Napisati logički izraz koji će vratiti tačno ukoliko se *a* nalazi između 5 i 10, uključujući ih.
5. Napisati logički izraz koji će vratiti tačno ukoliko je *a* strogo manje od –10 ili strogo veće od 10.
6. Negirati logičke izraze iz prethodna dva zadatka.

Tipovi podataka

Ugrađeni tipovi podataka

U programski jezik *Python* su ugrađeni sledeći tipovi podataka:

Tekstualni tip:	str
Brojevni tipovi:	int, float, complex
Logički tip:	bool
Binarni tipovi:	bytes, bytearray, memoryview
'None' tip:	NoneType
Kolekcije:	list, tuple, range; dict; set, frozenset

Primer 10 – provera tipa pozivom funkcije type*()*

```
x = 5
print(type(x)) # Ispisuje <class 'int'>
```

U ovom primeru je promenljivoj *x* implicitno dodeljen tip *int*, time što joj je dodeljena celobrojna vrednost 5. Dakle, dodelom određenih vrednosti i promenljiva dobija odgovarajući tip.

Primer 11 - dodela vrednosti različitih tipova

Naredba dodele	**Tip podatka**
x = "Zdravo"	str
x = 20	int
x = 20.5	float
x = 1j	complex
x = True, x = False	bool
x = b"Zdravo"	bytes
x = bytearray(5)	bytearray
x = None	NoneType
x = ["jabuka", "banana", "limun"]	list
x = ("jabuka", "banana", "limun")	tuple
x = range(6)	range
x = {"ime" : "Marko", "godine" : 23}	dict
x = {"jabuka", "banana", "limun"}	set

Primer 12 – eksplicitno navođenje tipa pri dodeli vrednosti

```
x = str("Zdravo")
x = int(20)
x = list(("jabuka", "banana", "limun"))
x = dict(ime="Marko", godine=23)
```

Logički tip

U sekciji sa operatorima dodele prikazano je kako se logički izrazi izračunavaju u tačno (*True*) ili netačno (*False*). Slično, sve vrednosti (konstante ili promenljive) drugih tipova imaju i svoju logičku vrednost. Određivanje te vrednosti vrši se pomoću funkcije *bool*().

Primer 13

```
bool("abc") # True
bool(123) # True
bool(["apple", "cherry", "banana"]) # True
```

Većina vrednosti se izračunava u *True*, dok se pojedine vrednosti različitih tipova tretiraju kao *False*.

Primer 14

```
bool(None) # False
bool(0) # False
bool("") # False
bool(()) # False
bool([]) # False
bool({}) # False
```

Posebna vrednost *None*, broj 0, prazan string i prazne kolekcije se tretiraju kao logičko *False*.

Konverzije tipova – *casting*

Dodela specifičnog tipa nekoj promenljivoj naziva se i prebacivanje (konverzija, *casting*) u određeni tip. Pri tome, vrednost koja se navodi u zagradama može već biti željenog tipa, ali uglavnom nije. Pa tako često prebacujemo npr. *float* vrednosti u *int* ili obrnuto, kao i bilo koji drugi tip u *string* iobrnuto.

Primer 15

```
x = int(1)   # x ce biti 1
y = int(2.8) # y ce biti 2
z = int("3") # z ce biti 3
w = int("3.2")# Greska!

x = float(1)     # x ce biti 1.0
y = float(2.8)   # y ce biti 2.8
z = float("3")   # z ce biti 3.0
w = float("4.2") # w ce biti 4.2
v = float("3,2")# Greska!

x = str("s1") # x ce biti 's1'
y = str(2)    # y ce biti '2'
z = str(3.0)  # z ce biti '3.0'
```

Pri prebacivanju u celobrojni tip (*int*), ukoliko je prosleđena decimalna vrednost, deo posle zareza se zanemaruje, a ukoliko je prosleđena tekstualna vrednost, ona mora predstavljati ceo broj.

Pri prebacivanju u decimalni tip (*float*), celobrojne vrednosti će se pisati u formatu sa zarezom (1.0), što važi i za prosleđene tekstualne zapise celih brojeva (3.0).

Pri prebacivanju u tekstualni tip (*string*) nema ograničenja.

Interakcija sa korisnikom

Pod interakcijom sa korisnikom podrazumevamo čitanje podataka koje je korisnik programa uneo putem tastature i ispis poruka ili rezultata nazad na ekran.

Čitanje korisničkog unosa – *input*()

Funkcija *input*() omogućava istovremeno ispisivanje poruke korisniku na ekran, kroz parametar, kao i preuzimanje dobijenog odgovora. Čitanje se prekida kada korisnik pritisne *enter*, a preuzeta vrednost će uvek biti *string* (tekstualni tip podataka).

Primer 16

```
tekst = input("Unesite neki tekst: ")
```

Nakon izvršenja prve linije, program će se zaustaviti i nastaviti sa radom tek kada se pritisne *enter*. Sve uneto pre *entera*, biće sačuvano u promenljivoj *tekst.*

Primer 17

```
broj = int(input("Unesite neki broj: "))
```

Ukoliko želimo da pročitamo broj, potrebno je da unos prebacimo u odgovarajući tip – u ovom primeru je to celobrojni tip podataka (*int*).

Primer 18

```
x, y = input("Unesite dve vrednosti: ").split()
print("Prva vrednost: ", x)
print("Druga vrednost: ", y)

x, y = input("Unesite dve vrednosti: ").split(',')
print("Prva vrednost: ", x)
print("Druga vrednost: ", y)
```

Ukoliko želimo da pročitamo više vrednosti istovremeno, na *input* funkciju može da se nadoveže funkcija *split*(). Ako se kao parametar te funkcije navede neki string, onda se unos deli po tom stringu, a ukoliko se parametar izostavi, deli se po razmacima.

Ispis na ekran – *print*()

Funkcija *print* ispisuje prosleđenu/e vrednost/i na ekran, pretvarajući ih u tekstualni tip (*string*). Moguće je, dakle, proslediti više vrednosti.

Primer 19

```
print("Neki tekst") # Ispisuje 'Neki tekst'
print(5) # Ispisuje '5'

x = 2.4
print(x) # Ispisuje '2.4'
print("x = ", x) # Ispisuje 'x = 2.4'

a = 2
b = 3
print("a + b =", a+b) # Ispisuje 'a + b = 5'
```

Primer 20

```
a=12
b=12
c=2022
print(a,b,c,sep="-") # Ispisuje '12-12-2022'
print('jabuke', 'narandze', 'banane', sep=', ') # Ispisuje jabuke, narandze,
banane
print('jabuke', 'narandze', 'banane', sep=' i ') # Ispisuje jabuke i narandze
i banane
print('A', 'B', 'C', sep='') # Ispisuje 'ABC' (bez sep: A B C)
```

Dodatni parametar *print* funkcije jeste ***sep*** parametar, kojim se zadaje tekst koji će se umetati između prosleđenih vrednosti prilikom ispisa.

Rad sa stringovima

Niz karaktera, odnosno tekst, u *Python*-u se predstavlja tipom *string*. Stringovi, dakle, imaju sve osobine nizova, te je moguće indeksirati ih, iterirati kroz njih, izračunati njihovu dužinu itd. Neke od ovih osobina će biti prikazane kasnije.

String se jednako kreira korišćenjem jednostrukih, kao i dvostrukih navodnika. Dakle, ispravno je i *“Zdravo”*i *‘Zdravo’*.

Primer 21

```
tekst = 'Neki tekst'
slovo = "A"
```

Višelinijski string

Tekst koji se prostire u više redova, navodi se među troduple znake navode – jednostruke ili dvostruke.

Primer 22

```
dugacakTekst = """Važno je, možda, i to da znamo
Čovek je željan tek ako želi
I ako sebe celoga damo
Tek tada i možemo biti celi.
Saznaćemo tek ako kažemo
Reči iskrene, istovatne
I samo onda kad i mi tražimo
Moći će neko i nas da sretne"""
```

Dužina stringa

Dužina stringa se dobija preko *len*() funkcije.

Primer 23

```
a = "Zdravo!"
print(len(a)) # Ispisuje 7
```

Provera pripadnosti stringu

Provera da li se određena fraza (ili karakter) nalazi u zadatom stringu vrši se preko operatora pripadnosti – *in* i *not in*.

Primer 24

```
tekst = "Najbolje stvari u životu su besplatne!"
print("besplatne"in tekst) # Ispisuje True
tekst = "Najbolje stvari u životu su besplatne!"
print("skupe"notin tekst) # Ispisuje True
```

Izvlačenje podniza karaktera iz stringa

Podniz karaktera se iz neke tekstualne vrednosti izvlači zadavanjem željenog opsega. Moguće je zadati početni i krajnji indeks ili samo jedan od ta dva, pri čemu se izdvajaju karakteri od zadatog indeksa do kraja stringa ili od početka stringa do zadatog indeksa.

Važno! Indeksi počinju od 0.

Primer 25

```
pozdrav = "Zdravo svima!"
print(pozdrav[0])# Ispisuje 'Z'
print(pozdrav[7:10]) # Ispisuje 'svi'
print(pozdrav[:6]) # Ispisuje 'Zdravo'
print(pozdrav[7:]) # Ispisuje 'svima!'
```

Modifikovanje stringova

Postoji ugrađeni skup funkcija za rad sa stringovima, od kojih se u nastavku navode neke od najčešće korišćenih.

Prebacivanje u velika slova – upper()

```
a = "Zdravo!"
print(a.upper()) # Ispisuje 'ZDRAVO!'
```

Prebacivanje u mala slova – lower()

```
a = "Zdravo, Marko!"
print(a.lower()) # Ispisuje 'zdravo, marko!'
```

Uklanjanje razmaka – strip()

```
a = " Zdravo!  "
print(a.strip()) # Ispisuje 'Zdravo!
```

Zamena – replace()

```
a = "Cao svima!"
print(a.replace("Cao", "Zdravo")) # Ispisuje 'Zdravo svima!'
```

Podela stringa na podnizove – split()

Funkcija *split*() deli string po zadatom karakteru i vraća niz dobijenih podnizova (engl. *substring*).

```
a = "Zdravo, Marko"
print(a.split(",")) # Ispisuje ['Zdravo', ' Marko']
```

Spajanje stringova

Spajanje ili konkatenacija stringova vrši se operatorom sabiranja (**+**).

```
a = "Zdravo"
b = "Marko"
c = a + b
print(c) # Ispisuje 'ZdravoMarko'

a = "Zdravo"
b = "Marko"
c = a + ", " + b
print(c) # Ispisuje 'Zdravo, Marko'
```

Formatiranje stringova

Kombinovanje teksta i brojevnih promenljivih u jedan ispis vrši se pomoću funkcije *format*().

```
godine = 23
tekst = "Moje ime je Marko i ja imam {} godine."
print(tekst.format(godine)) # Ispisuje 'Moje ime je Marko i ja imam 23
godine.'

kolicina = 3
idProizvoda = 567
cena = 50
porudzbina = "Želim {} komada proizvoda {} za {} dinara."
print(porudzbina.format(kolicina, idProizvoda, cena)) # Ispisuje 'Želim 3
komada proizvoda 567 za 50 dinara.'
```

Pisanje nedozvoljenih karaktera u stringu

Upotreba navodnika unutar stringa nije dozvoljena, tj. prekinuće tekući string i ostatak teksta će prouzrokovati grešku. Ovo je ilustrovano sledećim primerom:

Primer 26

```
tekst = "Ovo je "citat""
```

Ispravan način za pisanje navodnika (i jednostrukih i dvostrukih) unutar stringa jeste navođenjem obrnute kose crte ispred znaka navoda.

Primer 27

```
tekst = "Ovo je \"citat\""
print(tekst) # Ispisuje 'Ovo je "citat"'
```

Zbog ove funkcije obrnute kose crta, ona se takođe smatra specijalnim karakterom. To znači da joj prilikom ispisa mora prethoditi još jedna obrnuta kosa crta.

Primer 28

```
tekst = "Ispis obrnute kose crte: \\"
print(tekst) # Ispisuje 'Ispis obrnute kose crte: \'
```

Zadaci

1. Tražiti od korisnika da unese dva realna broja, sabrati ih i ispisati rezultat na ekran.

2. Tražiti od korisnika da unese tri cela broja, koji redom predstavljaju dan, mesec i godinu, a zatim ih ispisati na ekran u sledećem formatu: *dd/mm/yyyy*

3. Tražiti od korisnika da unese neki tekst, prebaciti ga u mala slova, obrisati sve bele karaktere sa oba kraja tog teksta i podeliti ga po razmaku.
4. Tražiti od korisnika da unese neki tekst i zameniti svaku pojavu reči 'cao' sa 'zdravo'.
5. Tražiti od korisnika da unese neki tekst i napisati sledeče logičke izraze:
    a. 'je' se nalazi u tekstu
    b. 'sam' se ne nalazi u tekstu
    c. Prvo slovo je 'A'
    d. Dužina teksta je različita od nule
6. Tražiti od korisnika da unese ime, prezime i godine, odvojene zarezom. Svaku vrednost sačuvati u posebnu promenljivu i zatim ih ispisati u sledećem formatu: „*Marko Marković* ima *20* godina."

Kolekcije podataka

Kolekcije podataka služe da bi čuvale više elemenata, tj. više vrednosti u jednoj promenljivoj.

U programskom jeziku *Python* postoje četiri ugrađena tipa kolekcija podataka:

- Lista (*list*) je uređena, promenljiva kolekcija, koja dozvoljava ponavljanje vrednosti.
- Torka (*tuple)* je uređena, nepromenljiva kolekcija, koja dozvoljava ponavljanje vrednosti.
- Skup (*set*) je neuređena, nepromenljiva kolekcija, koja ne dozvoljava ponavljanje vrednosti.
- Rečnik (*dictionary*) je uređena, promenljiva kolekcija, koja ne dozvoljava ponavljanje vrednosti.

Liste

Lista je uređena kolekcija, što znači da njeni elementi imaju određeni redosled, koji se vremenom ne menja. Prilikom dodavanja novih elemenata, oni se smeštaju na kraj kolekcije, što ne remeti poredak ostalih elemenata. Lista je promenljiva kolekcija, što znači da se elementi mogu dodavati, menjati i brisati nakon kreiranja liste. Lista je takođe i indeksirana, pa tako kažemo da se prvi element nalazi na indeksu 0, drugi na indeksu 1 itd. Pošto svakom elementu odgovara drugačiji indeks, moguće je skladištiti više istih vrednosti u ovoj kolekciji.

Primer 29

```
lista = ["jabuke", "banane", "kivi"]
print(lista) # Ispisuje ['jabuke', 'banane', 'kivi']
```

Dužina liste određuje se pomoću funkcije *len().*

Primer 30

```
duzina = len(lista)
print(duzina) # Ispisuje 3
```

Liste mogu sadržati vrednosti istog tipa ili različitih tipova. Tip same promenljive koja je lista će uvek biti *list.*

Primer 31

```
lista1 = ["jabuke", "banane", "kivi"]
lista2 = [1, 5, 7, 9, 3]
lista3 = [True, False, False]
lista4 = ["abc", 34, True, 40]
print(type(lista1)) # Ispisuje <class 'list'>
print(type(lista2)) # Ispisuje <class 'list'>
print(type(lista3)) # Ispisuje <class 'list'>
print(type(lista4)) # Ispisuje <class 'list'>
```

Listu je moguće kreirati i preko funkcije *list*(), koju nazivamo i konstruktorom, pri čemu se vrednosti prosleđuju kao parametar, obuhvaćene još jednim parom običnih zagrada.

Primer 32

```
lista = list((1, 2, 3))
print(lista) # Ispisuje [1, 2, 3]
```

Pristup elementima

Elementu liste se pristupa navođenjem njegovog indeksa u uglaste zagrade nakon imena liste. Više elemenata može da se pročita navođenjem opsega indeksa, gde se izostavljanjem početne vrednosti čitaju svi elementi od početka, a izostavljanjem krajnje vrednosti svi elementi do kraja liste. Dodatno, listu je moguće indeksirati negativnim vrednostima, pri čemu se elementi broje s kraja liste počevši od –1.

Primer 33

```
lista = ["jabuke", "banane", "kivi", "mandarine", "grozdje", "mango"]
print(lista[2]) # Ispisuje 'kivi'
print(lista[2:5]) # Ispisuje ['kivi', 'mandarine', 'grozdje']
print(lista[:3]) # Ispisuje ['jabuke', 'banane', 'kivi']
print(lista[3:]) # Ispisuje ['mandarine', 'grozdje', 'mango']

print(lista[-1]) # Ispisuje 'mango'
print(lista[-4:-1]) # Ispisuje ['kivi', 'mandarine', 'grozdje']
print(lista[:-2]) # Ispisuje ['jabuke', 'banane', 'kivi', 'mandarine']
print(lista[-4:]) # Ispisuje ['kivi', 'mandarine', 'grozdje', 'mango']
```

Važno je napomenuti da se prilikom zadavanja opsega uzima u obzir i element sa početnim indeksom, ali se NE uzima u obzir element na krajnjem indeksu. Zadavanjem opsega od 2 do 5 ([2:5]) izvlače se elementi sa indeksima 2, 3 i 4.

Iz tog razloga zadavanje negativnog opsega [-4:-1] ne vraća poslednja četiri elementa, nego samo četvrti, treći i drugi od nazad. Poslednja četiri elementa pročitana su zadavanjem opsega [-4:].

Dodavanje, izmena i brisanje elemenata

Dodavanje elemenata na kraj liste vrši se pomoću funkcije *append*(), ubacivanje elementa na određeni indeks pomoću funkcije *insert*(), a korišćenjem funkcije *extend*() moguće je proširiti trenutnu listu nekom drugom kolekcijom.

Primer 34

```
lista = ["jabuke", "banane", "kivi"]
lista.append("mango")
```

```
print(lista) # Ispisuje ['jabuke', 'banane', 'kivi', 'mango']

lista.insert(1, "kruska")
print(lista) # Ispisuje ['jabuke', 'kruska', 'banane', 'kivi', 'mango']
lista2 = ["maline", "kupine", "jagode"]
lista.extend(lista2)
print(lista) # Ispisuje ['jabuke', 'kruska', 'banane', 'kivi', 'mango',
'maline', 'kupine', 'jagode']
```

Izmena vrednosti na određenom mestu u listi vrši se pomoću indeksiranja i naredbe dodele. Pri tome je moguće izmeniti više vrednosti istovremeno, ubacujući na njihovo mesto manje, više ili jednako novih vrednosti.

Primer 35

```
lista = ["jabuke", "banane", "kivi", "mandarine", "grozdje", "mango"]

lista[0] = "kruske"
print(lista) # Ispisuje ['kruske', 'banane', 'kivi', 'mandarine', 'grozdje',
'mango']

lista[2:4] = ["jagode", "maline"]
print(lista) # Ispisuje ['kruske', 'banane', 'jagode', 'maline', 'grozdje',
'mango']

lista[2:4] = ["kupine"]
print(lista) # Ispisuje ['kruske', 'banane', 'kupine', 'grozdje', 'mango']

lista[2:4] = ["borovnice", "ribizle", "narandza"]
print(lista) # Ispisuje ['kruske', 'banane', 'borovnice', 'ribizle',
'narandza', 'mango']
```

Brisanje elemenata vrši se pomoću funkcija *remove*(), *pop*() i *clear*(), kao i pomoću naredbe *del.* Funkcija *remove* uklanja zadatu vrednost iz liste, dok funkcija *pop* uklanja vrednost sa zadatog indeksa ili sa kraja, ukoliko se parametar izostavi. Funkcija *clear* će obrisati sve elemente (lista će ostati prazna), dok pomoću naredbe *del* možemo u potpunosti obrisati i samu listu, ali i elemente na određenim indeksima.

Primer 36

```
lista = ["jabuke", "banane", "kivi", "mandarine", "grozdje", "mango"]

lista.remove("kivi")
print(lista) # Ispisuje ['jabuke', 'banane', 'mandarine', 'grozdje', 'mango']
lista.pop()
print(lista) # Ispisuje ['jabuke', 'banane', 'mandarine', 'grozdje']

lista.pop(0)
print(lista) # Ispisuje ['banane', 'mandarine', 'grozdje']
```

```
lista.clear()
print(lista) # Ispisuje []

lista = ["jabuke", "banane", "kivi", "mandarine", "grozdje", "mango"]
del lista[0]
print(lista) # Ispisuje ['banane', 'kivi', 'mandarine', 'grozdje', 'mango']

del lista[1:3]
print(lista) # Ispisuje ['banane', 'grozdje', 'mango']

del lista
print(lista) # Greska! NameError: name 'lista' is not defined.
```

Sortiranje elemenata

Sortiranje elemenata liste vrši se pomoću funkcije *sort*(), pri čemu se stringovi sortiraju alfabetski, a brojevi po vrednosti, u rastućem redosledu. Sortiranje u opadajućem redosledu zahteva da se funkciji prosledi sledeći parametar: *reverse = True.*

Primer 37

```
lista1 = ["jabuke", "banane", "kivi", "mandarine", "grozdje", "mango"]
lista2 = [1, 5, 7, 9, 3]
lista1.sort()
lista2.sort()
print(lista1) # Ispisuje ['banane', 'grozdje', 'jabuke', 'kivi', 'mandarine',
'mango']
print(lista2) # Ispisuje [1, 3, 5, 7, 9]

lista1.sort(reverse=True)
lista2.sort(reverse=True)
print(lista1) # Ispisuje ['mango', 'mandarine', 'kivi', 'jabuke', 'grozdje',
'banane']
print(lista2) # Ispisuje [9, 7, 5, 3, 1]
```

Za obrtanje redosleda elemenata u listi, tako da poslednji postane prvi, pretposlednji drugi itd, služi funkcije *reverse*().

Primer 38

```
lista1 = ["jabuke", "banane", "kivi", "mandarine", "grozdje", "mango"]
lista2 = [1, 5, 7, 9, 3]
lista1.reverse()
lista2.reverse()
print(lista1) # Ispisuje ['mango', 'grozdje', 'mandarine', 'kivi', 'banane',
'jabuke']
print(lista2) # Ispisuje [3, 9, 7, 5, 1]
```

Rečnici

Rečnik je uređena kolekcija, što znači da njeni elementi imaju određeni redosled, koji se vremenom ne menja. Rečnik je promenljiva kolekcija, jer se elementi mogu menjati, dodavati i brisati nakon kreiranja. Rečnik čuva elemente u parovima ključ-vrednost. To znači da se elementima može

pristupati preko ključa, koji mora biti jedinstven. Nije dozvoljeno postojanje više elemenata sa istim ključem.

Primer 39

```
recnik = {
    "marka": "Ford",
    "model": "Mustang",
    "godina": 1964
}
print(recnik) # Ispisuje {'marka': 'Ford', 'model': 'Mustang', 'godina': 1964}
```

Broj elemenata rečnika, odnosno broj parova ključ-vrednost, određuje se pomoću funkcije *len().*

Primer 40

```
duzina = len(recnik)
print(duzina) # Ispisuje 3
```

Rečnici mogu sadržati vrednosti (i ključeve) istog tipa ili različitih tipova. Tip same promenljive koja je rečnik će uvek biti *dict.*

Primer 41

```
recnik = {
    "marka": "Ford",
    "model": "Mustang",
    "godina": 1964,
    "boje": ["crvena", "crna", "zuta"]
}
print(type(recnik)) # Ispisuje <class 'dict'>
```

Rečnik je moguće kreirati i preko funkcije *dict*(), koju nazivamo i konstruktorom, pri čemu se vrednosti prosleđuju dodeljene ključevima, kao parametri.

Primer 42

```
recnik = dict(marka = "Ford", model = "Mustang", godina = 1964)
print(recnik) # Ispisuje {'marka': 'Ford', 'model': 'Mustang', 'godina': 1964}
```

Pristupanje elementima

Najčešće se vrednosti rečnika dobavljaju preko ključa. To je moguće uraditi na dva načina - navođenjem naziva ključa u uglastim zagradama ispred naziva rečnika ili pomoću *get*() funkcije kojoj se naziv ključa prosleđuje kao parametar.

Primer 43

```
recnik = {
    "marka": "Ford",
    "model": "Mustang",
    "godina": 1964,
    "boje": ["crvena", "crna", "zuta"]
}
print(recnik["godina"]) # Ispisuje 1964
print(recnik.get("model")) # Ispisuje 'Mustang'
```

Listu svih ključeva, svih vrednosti i svih elemenata moguće je dobiti pomoću funkcija *keys*(), *values*() i *items*(), respektivno.

Primer 44

```
recnik = {
    "marka": "Ford",
    "model": "Mustang",
    "godina": 1964,
    "boje": ["crvena", "crna", "zuta"]
}

kljucevi = recnik.keys()
print(kljucevi) # Ispisuje dict_keys(['marka', 'model', 'godina', 'boje'])

vrednosti = recnik.values()
print(vrednosti) # Ispisuje dict_values(['Ford', 'Mustang', 1964, ['crvena',
'crna', 'zuta']])

elementi = recnik.items()
print(elementi) # Ispisuje dict_items([('marka', 'Ford'), ('model',
'Mustang'), ('godina', 1964), ('boje', ['crvena', 'crna', 'zuta'])])
```

Može se primetiti da su povratne vrednosti ovih metoda redom: *dict_keys*, *dict_values* i *dict_items*, gde je poslednja zapravo lista torki od dva elementa, tačnije lista uređenih parova ključ-vrednost.

Dodavanje, izmena i brisanje elemenata

Dodavanje novog elementa ili izmena postojećeg rade se isto i to na dva načina. Prvi je pomoću pristupanja elementu preko ključa i naredbe dodele, a drugi pomoću funkcije *update*().

Primer 45

```
recnik = {
    "marka": "Ford",
    "model": "Mustang",
    "godina": 1964
}
recnik["godina"] = 2000
print(recnik) # Ispisuje {'marka': 'Ford', 'model': 'Mustang', 'godina': 2000}

recnik["boja"] = "crvena"
print(recnik) # Ispisuje {'marka': 'Ford', 'model': 'Mustang', 'godina': 2000,
'boja': 'crvena'}

recnik.update({"godina": 2020})
print(recnik) # Ispisuje {'marka': 'Ford', 'model': 'Mustang', 'godina': 2020,
'boja': 'crvena'}

recnik.update({"cena": 100000})
print(recnik) # Ispisuje {'marka': 'Ford', 'model': 'Mustang', 'godina': 2020,
'boja': 'crvena', 'cena': 100000}
```

U ovom primeru je redom: izmenjena vrednost pod ključem “godina” na 2000, dodat nov ključ “boja” sa vrednošću “crvena”, izmenjena vrednost pod ključem “godina” na 2020, dodat nov ključ “cena” sa vrednošću 100000.

Brisanje elemenata rečnika je moguće po ključu, pomoću funkcije *pop*() ili naredbe *del*, zatim je moguće obrisati element koji je poslednji dodat pomoću funkcije *popitem*(), obrisati sve elemente pomoću funkcije *clear*() i obrisati ceo rečnik pomoću naredbe *del*.

Primer 46

```
recnik = {
    "marka": "Ford",
    "model": "Mustang",
    "godina": 1964,
    "boje": ["crvena", "crna", "zuta"]
}

recnik.pop("marka")
print(recnik) # Ispisuje {'model': 'Mustang', 'godina': 1964, 'boje':
['crvena', 'crna', 'zuta']}

del recnik["model"]
print(recnik) # Ispisuje {'godina': 1964, 'boje': ['crvena', 'crna', 'zuta']}

recnik.popitem()
print(recnik) # Ispisuje {'godina': 1964}

recnik.clear()
print(recnik) # Ispisuje {}

del recnik
print(recnik) # Greska! NameError: name 'recnik' is not defined.
```

Zadaci

1. Data je lista:

```
lista = ["jabuke", "banane", "kivi", "mandarine", "grozdje", "mango"]
```

a) Ispiši drugi element.
b) Promeni treći element na “kupine”.
c) Dodaj vrednost “narandže” na kraj liste.
d) Ubaci vrednost “limun” na indeks 2.
e) Ukloni vrednost “mandarine” iz liste.
f) Ispiši elemente od drugog zaključno sa petim.
g) Ispiši poslednji element liste.
h) Ispiši dužinu liste.
i) Sortiraj listu.
j) Obriši promenljivu *lista*.

2. Dat je rečnik:

```
recnik = {
    "marka": "Ford",
    "model": "Mustang",
    "godina": 1964
}
```

a) Ispiši vrednost pod ključem “model” na oba načina.
b) Promeni vrednost pod ključem “godina” na 2003.
c) Dodaj novi par ključ-vrednost koji će definisati da je boja žuta.
d) Obrisati ključ “marka” iz rečnika.
e) Obrisati sve parove ključ-vrednost iz rečnika.

Torke

Torka je uređena kolekcija, što znači da njeni elementi imaju određeni redosled, koji se vremenom ne menja. Torka je **nepromenljiva** kolekcija, što znači da nije moguće dodavati, menjati i brisati elemente nakon kreiranja torke. Torka je takođe i indeksirana, pa tako kažemo da se prvi element nalazi na indeksu 0, drugi na indeksu 1 itd. Pošto svakom elementu odgovara drugačiji indeks, moguće je skladištiti više istih vrednosti u ovoj kolekciji.

Primer 47

```
torka = ("jabuke", "banane", "kivi")
print(torka) # Ispisuje ('jabuke', 'banane', 'kivi')
```

Dužina torke, odnosno broj njenih elemenata, određuje se pomoću funkcije *len().*

Primer 48

```
duzina = len(torka)
print(duzina) # Ispisuje 3
```

Torke mogu sadržati vrednosti istog tipa ili različitih tipova. Tip same promenljive koja je torka će uvek biti *tuple.*

Primer 49

```
torka1 = ("jabuke", "banane", "kivi")
torka2 = (1, 5, 7, 9, 3)
torka3 = ("abc", 34, True, 40)

print(type(torka1)) # Ispisuje <class 'tuple'>
print(type(torka2)) # Ispisuje <class 'tuple'>
print(type(torka3)) # Ispisuje <class 'tuple'>
```

Torku je moguće kreirati i preko funkcije *tuple*(), koju nazivamo i konstruktorom, pri čemu se vrednosti prosleđuju kao parametar, obuhvaćene još jednim parom običnih zagrada.

Primer 50

```
torka = tuple((1, 2, 3))
print(torka) # Ispisuje (1, 2, 3)
```

Pristup elementima

Elementu torke se pristupa navođenjem njegovog indeksa u uglaste zagrade nakon imena torke. Više elemenata može da se pročita navođenjem opsega indeksa, gde se izostavljanjem početne vrednosti čitaju svi elementi od početka, a izostavljanjem krajnje vrednosti svi elementi do kraja torke. Dodatno, torku je moguće indeksirati negativnim vrednostima, pri čemu se elementi broje s kraja torke počevši od –1.

Primer 51

```
torka = ("jabuke", "banane", "kivi", "mandarine", "grozdje", "mango")
print(torka[2]) # Ispisuje 'kivi'
print(torka[2:5]) # Ispisuje ('kivi', 'mandarine', 'grozdje')
print(torka[:3]) # Ispisuje ('jabuke', 'banane', 'kivi')
print(torka[3:]) # Ispisuje ('mandarine', 'grozdje', 'mango')
print(torka[-1]) # Ispisuje 'mango'
print(torka[-4:-1]) # Ispisuje ('kivi', 'mandarine', 'grozdje')
print(torka[:-2]) # Ispisuje ('jabuke', 'banane', 'kivi', 'mandarine')
print(torka[-4:]) # Ispisuje ('kivi', 'mandarine', 'grozdje', 'mango')
```

Važno je napomenuti da se prilikom zadavanja opsega uzima u obzir i element sa početnim indeksom, ali se NE uzima u obzir element na krajnjem indeksu. Zadavanjem opsega od 2 do 5 ([2:5]) izvlače se elementi sa indeksima 2, 3 i 4, odnosno treći, četvrti i peti element torke.
Iz tog razloga zadavanje negativnog opsega [-4:-1] ne vraća poslednja četiri elementa, nego samo četvrti, treći i drugi od nazad. Poslednja četiri elementa pročitana su zadavanjem opsega [-4:].

Dodavanje, izmena i brisanje elemenata

Pošto je torka nepromenljiva kolekcija, nakon njenog kreiranja nije dozvoljeno dodavanje, menjanje ni brisanje elemenata. Međutim, moguće je zaobići tu zabranu. To se radi tako što se torka prebaci u listu, zatim se nad listom izvrši željena promena elemenata i onda se lista vrati nazad u torku. U narednom primeru će biti prikazano dodavanje novog elementa, ali bi isti postupak bio i u slučaju izmene ili brisanja elemenata.

Primer 52

```
torka = ("jabuke", "banane", "kivi", "mandarine", "grozdje", "mango")
lista = list(torka)
lista.append("kruske")
torka = tuple(lista)
print(torka) # Ispisuje ('jabuke', 'banane', 'kivi', 'mandarine', 'grozdje',
'mango', 'kruske')
```

Jedina izmena dozvoljena nad ovom kolekcijom jeste spajanje sa još jednom torkom.

Primer 53

```
torka1 = ("jabuke", "banane", "kivi")
torka2 = ("mandarine", "grozdje")
torka3 = ("mango",)

torka1 += torka2
print(torka1) # Ispisuje ('jabuke', 'banane', 'kivi', 'mandarine', 'grozdje')
torka1 += torka3
```

```
print(torka1) # Ispisuje ('jabuke', 'banane', 'kivi', 'mandarine', 'grozdje',
'mango')
```

U ovom primeru vidi se i definisanje torke od jednog elementa (*torka3*), što se radi navođenjem zareza nakon te jedne vrednosti.

Otpakivanje torke

Proces dodeljivanja vrednosti torke pojedinačnim promenljivama naziva se otpakivanje (engl. *unpacking*). Ukoliko se pri tome navede manje promenljivih nego što je elemenata u torci, jedna od promenljivih se mora naznačiti zvezdicom (*) kako bi se svi viška elementi sačuvali u toj promenljivoj u vidu liste.

Primer 54

```
torka = ("jabuke", "banane", "kivi")

(crveno, zuto, zeleno) = torka
print(crveno) # Ispisuje 'jabuke'
print(zuto) # Ispisuje 'banane'
print(zeleno) # Ispisuje 'kivi'
```

Primer 55

```
torka = ("jabuke", "banane", "kivi", "mandarine", "grozdje", "mango")

(jabuke, *ostalo, mango) = torka
print(jabuke) # Ispisuje 'jabuke'
print(ostalo) # Ispisuje ['banane', 'kivi', 'mandarine', 'grozdje']
print(mango) # Ispisuje 'mango'
```

Zadaci

Data je torka:

```
torka = ("jabuke", "banane", "kivi", "mandarine", "grozdje", "mango")
```

1. Ispiši prva četiri elementa.
2. Ispiši poslednja dva elementa.
3. Kreiraj promenljivu za vrednost “mango” i promenljivu koja će sadržati sve ostale vrednosti torke.

Skupovi

Skup je neuređena kolekcija, što znači da njeni elementi nemaju određeni redosled. Ovo se naročito primeti prilikom ispisa skupa, gde će se elementi svaki put ispisati u drugačijem poretku. Samim tim je jasno da skup nije indeksirana kolekcija, te se elementima ne može pristupati ni pomoću indeksa ni pomoću ključa (vidi: Rečnici). Skup je nepromenljiva kolekcija, jer se elementi ne mogu menjati nakon kreiranja, ali ih je moguće brisati i dodavati nove. Skup ne dozvoljava pojavu istih vrednosti.

Primer 56

```
skup = {"jabuke", "banane", "kivi"}
print(skup) # Ispisuje {'kivi', 'banane', 'jabuke'} (ili nekim drugim
redosledom)
```

```
skup = {"jabuke", "banane", "kivi", "kivi"}
print(skup) # Ispisuje {'kivi', 'banane', 'jabuke'} (ili nekim drugim
redosledom)
```

Broj elemenata skupa određuje se pomoću funkcije *len().*

Primer 57

```
duzina = len(skup)
print(duzina) # Ispisuje 3
```

Skupovi mogu sadržati vrednosti istog tipa ili različitih tipova. Tip same promenljive koja je skup će uvek biti *set.*

Primer 58

```
skup1 = {"jabuke", "banane", "kivi"}
skup2 = {1, 5, 7, 9, 3}
skup3 = {True, False, False}
skup4 = {"abc", 34, True, 40}

print(type(skup1)) # Ispisuje <class 'set'>
print(type(skup2)) # Ispisuje <class 'set'>
print(type(skup3)) # Ispisuje <class 'set'>
print(type(skup4)) # Ispisuje <class 'set'>
```

Skup je moguće kreirati i preko funkcije *set*(), koju nazivamo i konstruktorom, pri čemu se vrednosti prosleđuju kao parametar, obuhvaćene još jednim parom običnih zagrada.

Primer 59

```
skup = set((1, 2, 3))
print(skup) # Ispisuje {1, 2, 3}
```

Pristupanje elementima

Elementima skupa ne može da se pristupa pomoću indeksa, kao što se to radi sa listama i torkama. Da bi se radilo sa pojedinačnim elementima, potrebno je iterirati kroz skup, što će biti objašnjeno u lekciji Petlje.

Dodavanje, izmena i brisanje elemenata

Elementi skupa ne mogu da se menjaju, ali je moguće dodati nove i ukloniti postojeće elemente.

Dodavanje jednog elementa radi se pomoću *add*() funkcije, a dodavanje neke druge postojeće kolekcije pomoću *update*() funkcije.

Primer 60

```
skup1 = {"jabuke", "banane", "kivi"}
skup1.add("kruske")
print(skup1) # Ispisuje {'kivi', 'banane', 'kruske', 'jabuke'}

skup2 = {"mandarine", "grozdje", "mango"}
```

```
skup1.update(skup2)
print(skup1) # Ispisuje {'mandarine', 'mango', 'jabuke', 'kivi', 'banane',
'kruske', 'grozdje'}
```

Brisanje određene vrednosti iz skupa može da se odradi pomoću funkcija *remove*() i *discard*(), pri čemu prva rezultuje greškom ukoliko prosleđeni element ne postoji u skupu, dok za drugu funkciju to nije slučaj.

Primer 61

```
skup1 = {"jabuke", "banane", "kivi"}
skup1.remove("kivi")
print(skup1) # Ispisuje {'banane', 'jabuke'}
skup1.remove("jabuka") # Greska!

skup2 = {"mandarine", "grozdje", "mango"}
skup2.discard("grozdje")
print(skup2) # Ispisuje {'mandarine', 'mango'}
skup2.discard("mandarina") # Nema efekta.
print(skup2) # Ispisuje {'mandarine', 'mango'}
```

Za uklanjanje elemenata može da se koristi i *pop*() funkcija, koja će nasumično obrisati neki element i vratiti njegovu vrednost.

Primer 62

```
skup1 = {"jabuke", "banane", "kivi", "mandarine", "grozdje"}
x = skup1.pop()
print("Obrisani element:", x) #Ispisuje 'Obrisani element: mandarine'
```

Brisanje svih elemenata radi se funkcijom *clear*(), a brisanje samog skupa naredbom *del*.

Primer 63

```
skup1 = {"jabuke", "banane", "kivi", "mandarine", "grozdje"}
skup1.clear()
print(skup1) # Ispisuje 'set()'

del skup1
print(skup1) # Greska! NameError: name 'skup1' is not defined.
```

Zadaci

Dat je skup:

```
skup = {"jabuke", "banane", "kivi", "mandarine", "grozdje", "mango"}
```

1. Dodaj vrednost “narandze”.
2. Dodaj skupu sledeću listu: [“visnje”, “jagode”, “maline”, “kupine”].
3. Tražiti od korisnika da unese vrednost koju želi da ukloni iz skupa. Brisanje izvršiti tako da program ne izbaci grešku ukoliko je korisnik uneo nepostojeću vrednost.

Vežba 2

Kontrola toka programa

Donošenje odluke koja će se sekvenca naredbi izvršiti sledeća naziva se kontrolom toka programa. Osnovni mehanizmi za ostvarivanje kontrole toka jesu grananja i petlje. Grananje podrazumeva odabir jednog od dva ili više mogućih blokova koda (sekvenci naredbi) u zavisnosti od nekih logičkih uslova.

S druge strane, petlja podrazumeva ponavljanje iste sekvence naredbi određeni broj puta (prebrojive petlje) ili dok god je neki uslov ispunjen (neprebrojive petlje).

Grananja

Grananje programa se definiše pomoću *if-else* strukture, pri čemu je *if* naredba osnova strukture.

Primer 64

```
if2<5:
    print("Dva je manje od pet.")
```

U sklopu *if* naredbe definiše se uslov, a zatim i blok naredbi koje će se izvršiti ukoliko je taj uslov ispunjen. Uslovi su zapravo logički izrazi, te se uslov smatra ispunjenim kada se evaluacijom logičkog izraza kojim je opisan dobije vrednost tačno (*True*). Ovom naredbom kreirana je posebna grana unutar koda, kojom tok izvršavanja programa može da krene, ali i ne mora. Ukoliko zadati uslov nije ispunjen, program će se nastaviti od prve sledeće naredbe nakon *if* naredbe.

Definisanje posebne grane, kojom bi program krenuo u slučaju da uslov u *if* naredbi nije ispunjen, omogućeno je upotrebom ključne reči *else*.

Primer 65

```
a = 33
b = 200
if b > a:
    print("b je veće od a")
else:
    print("b nije veće od a")
```

Na ovom mestu je zgodno podsetiti se da se novi blok koda definiše uvlačenjem naredbi, a ne vitičastim zagradama, kao u većini ostalih programskih jezika. To znači da bi kod iz prethodnog primera rezultovao greškom ukoliko bi sve naredbe bile poravnate. Dodatno se skreće pažnja na navođenje dvotačke na kraju *if* i *else* naredbi.

Ulančavanje više uslova, te i kreiranje više od dve grane koda, ostvaruje se pomoću ključne reči *elif*. Time se i kompletira *if-else* struktura, s tim što je potrebno naglasiti da je *if* naredba obavezna i navodi se samo jednom unutar jedne strukture. Naredbe *elif* i *else* su opcione, pri čemu *elif* blokova može biti više, dok je *else* blok jedan i ukoliko postoji navodi se na kraju strukture.

Primer 66

```
a = 33
b = 200
if b > a:
```

```
    print("b je veće od a")
elif a > b:
    print("a je veće od b")
else:
    print("a je jednako b")
```

Ukoliko se blok naredbi sastoji iz samo jedne naredbe, *if-else* strukture je moguće zapisati i skraćeno.

Primer 67

```
if a > b: print("a je veće od b")
print("A") if a > b else print("B")
```

Za pisanje uslova se mogu kombinovati operatori poređenja i logički operatori (*and*, *or* i *not*). Takođe, moguće je unutar *if*/*elif*/*else* blokova ugnezditi novu *if-else* strukturu.

Primer 68

```
x = 41

if x >10:
  print("x je veće od 10,")
  if x >20:
    print("a veće je i od 20")
  else:
    print("ali nije veće od 20")
```

Pass naredba

If/elif/else blokovi ne smeju biti prazni, tj. moraju sadržati makar jednu naredbu. Nekad je, međutim, za potrebe testiranja ili „čuvanja mesta“ neophodno da postoji prazan blok ili blok u kom je sav kod zakomentarisan. S obzirom na to da se komentari ne računaju u naredbe, takav blok bi se takođe smatrao praznim, te se zato u navedenim slučajevima piše *pass* naredba.

Primer 69

```
a = 33
b = 200
if b > a:
    pass
else:
    print("b nije veće od a")
```

Petlje

U *Python*-u postoje dva tipa programskih petlji – *for* i *while* petlja. Prva pretežno služi za iteriranje kroz kolekcije podataka, dok se druga koristi za ponavljanje određenog bloka dokle god je neki uslov ispunjen. Moguće je ugnezditi jednu ili više petlji unutar druge (neovisno od tipa petlje), kao i kombinovati ih sa *if-else* strukturama.

For petlja

Primer 70

```
lista = ["jabuka", "banana", "visnja"]
for x in lista:
  print(x)
```

```
rec = "jabuka"
for slovo in rec:
    print(slovo)
```

Ukoliko je potrebno iterirati kroz neki numerički opseg, ili jednostavno ponoviti neki blok koda određeni broj puta, to je moguće uraditi prolaskom *for* petljom kroz neki opseg (engl. *range*).

Primer 71

```
for x in range(6):
  print(x) # Ispisuje 0 1 2 3 4 5

for x in range(2, 6):
  print(x) # Ispisuje 2 3 4 5

for x in range(2, 30, 3):
  print(x) # Ispisuje 2 5 8 11 14 17 20 23 26 29
```

Zadavanjem jednog parametra *range*() funkciji, definisana je krajnja granica opsega, dok je startna granica 0. Obratiti pažnju na to da se zadata krajnja granica **ne uključuje** u opseg. Zadavanjem dva parametra definišu se i početak i kraj opsega, dok se dodavanjem trećeg parametra definiše i korak kojim će se kretati od početne do krajnje granice. U suprotnom je podrazumevani korak 1.

While petlja

S obzirom na to da se *while* petlja izvršava u zavisnosti od ispunjenosti nekog uslova, uglavnom se ne može unapred odrediti broj iteracija. U većini slučajeva će se u uslovu koristiti neka postojeća promenljiva, čija se vrednost tokom izvršavanja tela petlje menja, a zatim u uslovu proverava.

Primer 72

```
i = 1
while i <6:
  print(i)
  i += 1
```

Break, continue, pass i else naredbe

Uz oba tipa petlji moguće je koristiti sledeće naredbe:

- *break* – služi za prevremeni prekid petlje, nakon čega se kod nastavlja od sledeće naredbe nakon petlje
- *continue* – služi za prevremeni prekid trenutne interacije, nakon čega se vraća na proveru uslova u zavisnosti od kog se petlja nastavlja ili prekida
- *pass* – služi umesto praznog bloka
- *else* – služi za definisanje bloka koda koji će se izvršiti nakon prirodnog završetka petlje

Primer 73

```
for i in range(10):
    print(i)
    if i == 5:
        break

i = 1
while i <6:
```

```
  print(i)
  if i == 3:
    break
  i += 1
```

Primer 74

```
for i in range(1, 11):
    if i == 6:
        continue
    else:
        print(i, end=" ")

i = 0
while i <10:
    if i == 5:
        i += 1
        continue
    print(i)
    i += 1
```

Primer 75

```
n = 10
for i in range(n):
  pass

i = 0
while i <6:
    i += 1
    pass
```

Primer 76

```
for i in range(1, 4):
    print(i)
else:
    print("Ovo se ispisuje na kraju.")

i = 0
while i <4:
    i += 1
    print(i)
else:
    print("Ovo se ispisuje na kraju.")
```

Zadaci

1. Tražiti od korisnika da unese neki broj. Ukoliko je taj broj manji od 0, ispiši “Uneli ste negativni broj.”
2. Tražiti od korisnika da unese dva broja i sačuvati ih u promenljive a i b. Skraćenim zapisom *if-else* strukture ispisati onaj broj koji je veći, odnosno poruku da su jednaki ukoliko je to slučaj.
3. Kreirati listu povrća, a zatim proveriti da li se “krompir” nalazi u listi. Ukoliko se nalazi, ispisati odgovarajuću poruku, a ukoliko se ne nalazi, proveriti da li se “grasak” nalazi u listu. Opet, ukoliko se nalazi, ispisati odgovarajuću poruku, a ukoliko se ne nalazi ispisati “Vreme je za nabavku.” Zadatak uraditi bez upotrebe petlji.
4. Kreirati listu brojeva i pomoću *for* petlje ispisati svaki od njih.
5. Kreirati torku logičkih vrednosti i pomoću *for* petlje ispisati svaku od njih.
6. Kreirati skup stringova i pomoću *for* petlje ispisati svaki od njih.

7. Kreirati proizvoljni rečnik i pomoću *for* petlje ispisati sve ključeve.
8. Data je lista: lista = ["jabuke", "banane", "kivi", "mandarine", "grozdje", "mango"]. Pomoću *for* petlje ispisivati element po element. Ukoliko se „kivi“ nalazi u listi, preskočiti ga i nastaviti ispis od sledećeg elementa, a ukoliko se „grozdje“ nalazi u listi prekinuti izvršavanje petlje.
9. Data je promenljiva i = 5. Ispisivati njenu vrednost u *while* petlji dokle god je ona manja ili jednaka 10.
10. Data je promenljiva i = 1. Ispisivati njenu vrednost u *while* petlji dokle god je ona manja od 6, a kada prestane da bude manja, ispisati „Vrednost promenljive *i* više nije manja od 6“.

Obrada izuzetaka

U slučaju kada kod proizvede neku grešku, tj. kada se desi neki izuzetak, određene naredbe nam omogućavaju da taj izuzetak obradimo. Na taj način naš program može nastaviti sa radom, dok bi ga u suprotnom pojava greške prekinula. Deo koda za koji očekujemo da može da proizvede neku grešku ćemo enkapsulirati *try* blokom, a zatim ćemo u *except* bloku obraditi uhvaćeni izuzetak, ukoliko se on desi. U tom bloku ćemo najčešće ispisivati poruku greške.

Primer 77

```
try:
  print(x)
except:
  print("Greska!")

try:
  print(x)
except NameError as ex:
  print("Greska: ", ex)
except Exception as err:
   print(err)

try:
  print(x)
except (NameError, TypeError, SyntaxError):
  print("Greska!")
except:
   print("Neka druga greska!")
```

Za izvršavanje koda u situaciji kada se nije desila greška za koju smo testirali naš kod koristimo *else* blok, dok se za izvršavanje koda na kraju *try-except* bloka neovisno o tome da li je do greške došlo ili nije koristi *finally* blok. U tom bloku ćemo najčešće oslobađati resurse zauzete u *try* bloku.

Primer 78

```
try:
  print("Zdravo")
except:
  print("Greska!")
else:
  print("Nema greske!")

try:
  print(x)
```

```
except:
  print("Greska!")
finally:
  print("Kraj try-except bloka.")
```

S druge strane, bacanje/podizanje izuzetka, sa mesta na kom se desila greška, moguće je upotrebom ključne reči *raise*. Pri tome je moguće definisati tačan tip greške ili baciti izuzetak opšteg tipa.

Primer 79

```
x = int(input("Unesite nenegativan broj: "))

if type(x) isnot int:
   raise TypeError("Greska! Niste uneli ceo broj.")

if x <0:
  raise Exception("Greska! Broj je manji od 0.")
```

Funkcije

Funkcije su blokovi koda koji se izvršavaju po pozivu. Mogu da prihvataju neke podatke, koje nazivamo parametrima, kao i da vrate neke podatke, zvane i izlaznim/povratnim vrednostima ili rezultatom funkcije. Ideja iza upotrebe funkcija jeste izdvajanje zadataka koji se često ponavljaju nad različitim ulazima u posebne celine, čime se postiže veća čitljivost koda i omogućava ponovno korišćenje istog.

Osnovna podela funkcija jeste na ugrađene i korisnički definisane. Među ugrađene funkcije spadaju sve koje su do sada bile obrađene – *input*, *print*, *sort*, *upper*, *lower* itd.

Nova funkcija se definiše po sledećem šablonu:

```
defnaziv_funkcije(parametri):
    naredbe
```

Navodi se ključna reč *def*, posle koje se piše naziv funkcije. Zatim se u običnim zagradama navode parametri, ukoliko ih ima, nakon čega se dvotačkom otvara novi blok koda. Sledeći red se uvlači i piše se prva naredba. Ukoliko se definiše prazna funkcija, prva i jedina naredba će biti *pass*.

Primer 80

```
# Definicija
def funkcija():
   print("Ovo je funkcija.")
# Poziv
funkcija() # Ispisuje 'Ovo je funkcija.'
```

Parametri funkcije

Često se funkcije koriste kako bi iste naredbe izvršile nad drugačijim podacima. Prosleđivanje tih podataka funkciji se radi preko parametara, koji se navode unutar zagrada prilikom definisanja funkcije. Moguće je definisati funkcije sa određenim brojem parametara, kao i funkcije koje mogu da primaju različiti (neodređeni) broj parametara. Vrednosti koje se dodeljuju parametrima funkcije je moguće prosleđivati redom ili ih eksplicitno dodeljivati određenom parametru, a moguće je i

podesiti podrazumevanu vrednost parametrima. Naredni primeri ilustruju sve opisane varijante funkcija.

Primer 81

```
def predstavi_se(ime, prezime):
  print("Ja sam {} {}".format(ime, prezime))

predstavi_se("Petar", "Petrović") # Ispisuje 'Ja sam Petar Petrović'
predstavi_se(prezime = "Milošević", ime = "Miloš") # Ispisuje 'Ja sam Miloš
Milošević'
```

Primer 82

```
def funkcija_parametri(*args):
    for arg in args:
        print(arg)

funkcija_parametri("Prvi parametar", "Drugi parametar")
funkcija_parametri("Prvi parametar", "Drugi parametar", "Treći parametar")
```

Primer 83

```
def funkcija_imenovani_parametri(**kwargs):
    for key, value in kwargs.items():
        print("%s -> %s" % (key, value))

funkcija_imenovani_parametri(prvi='Prvi parametar', drugi='Drugi parametar',
treći='Treći parametar')
funkcija_imenovani_parametri(prvi='Prvi parametar', drugi='Drugi parametar')
funkcija_imenovani_parametri(prvi='Prvi parametar', drugi='Drugi parametar',
treći='Treći parametar', četvrti="Četvrti parametar")
```

Primer 84

```
def stepenovanje(x, y = 2):
   print(x**y)

stepenovanje(2, 3) # Ispisuje 8
stepenovanje(2) # Ispisuje 4
```

Sve prethodno definisane funkcije mogu da prihvate parametre bilo kog tipa. Tako, na primer, poslednja funkcija neće raditi ukoliko joj prosledimo string ili neku listu. Zbog toga je moguće eksplicitno zadati tip parametrima, kako bi pri pozivu bilo obavezno proslediti vrednosti tačno tog tipa. Ovde je zgodno napomenuti da se promenljive funkciji prosleđuju po referenci, te izmena parametra tokom izvršavanja funkcije ostaje vidljiva i nakon završetka funkcije. Međutim, ukoliko je parametru dodeljena nova vrednost, stvara se novi objekat i prekida veza sa promenljivom koja je prosleđena na mesto tog parametra. Takve izmene neće biti vidljive nakon završetka funkcije.

Primer 85

```
def stepenovanje(x: int, y: int):
    print(x**y)

stepenovanje(2, 3) # Ispisuje 8
stepenovanje("tekst", 4) # Greska!

def izmena(x):
   x[0] = 5

x = [1, 4, 3, 2, 1]
```

```
izmena(x)
print(x) # Ispisuje [5, 4, 3, 2, 1]

def dodela(x):
   x = [1, 2, 3, 4, 5]

x = [5, 4, 3, 2, 1]
dodela(x)
print(x) # Ispisuje [5, 4, 3, 2, 1]
```

Povratak iz funkcije

Funkcija se završava izvršenjem poslednje naredbe u definiciji/telu funkcije. Dalje se program nastavlja od prve sledeće naredbe nakon poziva funkcije. Eksplicitni izlazak iz funkcije omogućen je naredbom *return*, kojom je moguće vratiti i neke vrednosti na mesto u kodu sa kog je funkcija pozvana. Prilikom definisanja funkcije, moguće je zadati i tip povratne vrednosti.

Primer 86

```
def stepenovanje(x: int, y: int) -> int:
    return x**y

z = stepenovanje(2, 3)
print(z) # Ispisuje 8

def veci_broj(x, y):
    if x > y:
       return x
    elif y > x:
       return y
    returnNone
print(veci_broj(2, 5)) # Ispisuje 5
print(veci_broj(20, 5)) # Ispisuje 20
print(veci_broj(2, 2)) # Ispisuje None
```

Anonimne funkcije

Anonimne funkcije se kreiraju pomoću ključne reči *lambda* umesto ključne reči *def*. One mogu imati proizvoljan broj parametara, ali se obavezno sastoje iz samo jedne naredbe. Najčešće se anonimne funkcije koriste unutar običnih funkcija, npr. kao uslov za sortiranje, filtriranje i sl.

Primer 87

```
mnozenje = lambda a, b, c : a * b * c
print(mnozenje(5, 6, 2)) # Ispisuje 60

par_nepar = lambda broj: "Paran broj"if broj % 2 == 0else"Neparan broj"
print(par_nepar(20)) # Ispisuje 'Paran broj'

lista = ["1", "2", "9", "0", "-1", "-2"]
print("Pozitivni parni brojevi:",
      list(filter(lambda x: (int(x) % 2 == 0and int(x) >0), lista)))
# Ispisuje 'Pozitivni parni brojevi: ['2']'
```

Rekurzivne funkcije

Pojam rekurzije podrazumeva da se u definiciji funkcije nalazi poziv iste, odnosno da funkcija „poziva samu sebe“. Dobro implementirana, rekurzivna funkcija može biti efikasnija od obične (iterativne).

Primer 88

```
def faktorijel(n):
   if n>0:
      return n*faktorijel(n-1)
   else:
      return1

f = faktorijel(5)
print("5! = ", f) # Ispisuje '5! = 120'
```

Zadaci

1. Napisati funkciju koja prihvata listu kao parametar i menja je tako što na prvo mesto smešta najveći element, a na poslednje najmanji.
2. Napisati funkciju koja može da prima promenljiv broj parametara. Ukoliko je prosleđeno više od tri parametra, iz funkcije vratiti njihovu sumu, a u suprotnom vratiti njihov proizvod. Pozvati funkciju sa 2 i sa 4 parametra. Ukoliko je prosleđeno 0 parametara, baciti izuzetak.
3. Napisati funkciju koja prima dva celobrojna parametra i pokušava da ih podeli, te da vrati količnik. Ukoliko je za delilac prosleđena nula, odgovarajući izuzetak će biti uhvaćen i obrađen.
4. Datu listu sortirati numerički. Koristiti ugrađenu funkciju za sortiranje uz odgovarajuću lambda funkciju.
   lista = ["10", "2", "19", "0", "-1", "-20", "5"]
5. Napisati sopstvenu funkciju za filtriranje, koja kao parametre prihvata listu i anonimnu funkciju, kojom je određen uslov po kome se filtrira. Pomoću ove funkcije iz date liste izdvojiti sve parne brojeve, a zatim i sve negativne.
   lista = [2, 15, -5, 28, 9, -30, 4, -1]
6. Uvećati svaki element date liste za 10, ali tako da sve vrednosti ostanu stringovi. Koristiti ugrađenu funkciju *map* i odgovarajuću lambda funkciju.
   lista = ["10", "2", "19", "0", "-1", "-20", "5"]

Vežba 3

Klase i objekti

Klasa služi za grupisanje relevantnih podataka i ponašanja nekog entiteta iz realnog sveta. Objekat je jedan konkretni primerak (instanca) klase, te klasu nazivamo još i šablonom/nacrtom (engl. *blueprint*). Klase predstavljaju korisnički definisane tipove podataka. Sadrže atribute, koji su uvek javni i kojima se pristupa preko objekta koji ih sadrži, kao i funkcije kojima se definiše ponašanje entiteta koji se modeluje. Atributi se nazivaju još i svojstvima (engl. *property*) ili poljima klase, dok se funkcije uglavnom nazivaju metodama. Klase se definišu pomoću ključne reči *class*. Atributi i metode unutar njih se definišu po pravilima za kreiranje promenljivih i pisanje funkcija.

Primer 89

```
class Trougao:
   boja = "plava"

trougao = Trougao()
print(trougao.boja) # Ispisuje 'plava'
```

U ovom primeru je definisana klasa **Trougao** sa jednom klasnom promenljivom – *boja*. Zatim je instanciran objekat klase **Trougao** – ***trougao*** i ispisana vrednost njegovog atributa *boja*.

Ukoliko je potrebno da atributi različitih instanci iste klase imaju različite vrednosti, onda će se najčešće te vrednosti podešavati u metodi za inicijalizaciju, koja se još zove i konstruktorom. U toj metodi je moguće menjati vrednosti prethodno definisanim klasnim promenljivama, definisati nove atribute sa konstantim vrednostima ili im dodeliti vrednosti prosleđene kroz parametre ove metode.

Primer 90

```
class Trougao:
   boja = "plava"
   a = b = 1
   def __init__(self, a, b, c):
      self.a = a
      self.b = b
      self.c = c

trougao2 = Trougao(3, 4, 5)
print(trougao2.boja, trougao2.a, trougao2.b, trougao2.c) # Ispisuje 'plava 3 4
5'
```

U ovom primeru su definisana četiri atributa klase **Trougao**: *boja, a, b* i *c*, pri čemu se prva tri definišu na nivou klase, a zatim se atributima *a* i *b* vrednosti menjaju u konstruktoru, uz definiciju novog atributa (*c*).

Dodatni parametar *self* u metodi za inicijalizaciju predstavlja ništa drugo do tekući objekat koji se inicijalizuje. Ovaj parametar je dostupan u svim metodama klase i služi da bi se preko njega pristupalo atributima.

Naredni primer će nadograditi prethodni metodom koja definiše tekstualni prikaz objekta, odnosno način na koji će vrednosti njegovih atributa biti ispisane kada se on prosledi funkciji za ispis. Takođe, biće definisana još jedna metoda, metoda za računanje obima, koja radi sa atributima klase.

Primer 91

```
class Trougao:
    boja = "plava"
    a = b = 1

    def __init__(self, a, b, c):
      self.a = a
      self.b = b
      self.c = c

    def __str__(self):
       returnf"Trougao, boja: {self.boja}, stranice: {self.a}, {self.b},
{self.c}"
    def obim(self):
       returnself.a+self.b+self.c
trougao = Trougao(3, 4, 5)
print(trougao.boja, trougao.a, trougao.b, trougao.c) # Ispisuje 'plava 3 4 5'
print(trougao) # Ispisuje 'Trougao, boja: plava, stranice: 3, 4, 5'
print(trougao.obim()) # Ispisuje 12
```

Zadaci

1. Napisati klasu za studente FTNa. Neka atribut fakultet bude zajednički za sve instance ove klase, a neka se kroz konstruktor inicijalizuju ime, prezime i broj indeksa. Definisati metodu koja će prilikom ispisa objekta ispisivati vrednosti ova četiri atributa. Instancirati barem dva studenta i ispisati ih.
2. Napisati klasu koja opisuje merenja visine po gradovima u Srbiji. Potrebni atributi su: grad, drzava i kolekcija izmerenih vrednosti. Pored metoda za inicijalizaciju i tekstualni prikaz objekata, dodati i metodu koja računa prosečnu visinu za svaki grad (uz obradu izuzetaka). Instancirati barem tri grada i za svaki ispisati prosečnu visinu.
3. Napisati klasu Ucenik sa poljima: ime, prezime, ocene. Ocene su rečnik, pri čemu je ključ naziv predmeta, a vrednost lista ocena iz tog predmeta. Konstruktor kao parametre prihvata samo ime i prezime, dok se rečnik popunjava metodom za upis ocena, kojoj se prosleđuje predmet i ocena koja se upisuje. Pored ove metode, potrebno je definisati metodu koja spram prosleđenog naziva predmeta računa zaključnu ocenu za taj predmet i smešta je u novi recnik – zakljucene_ocene (uz obradu izuzetaka). Treća metoda računa prosek učenika spram svih zaključenih ocena (uz obradu izuzetaka).

Lokalni i globalni opseg

Svaka promenljiva je dostupna u okviru opsega (engl. *scope*) u kom je kreirana. Ukoliko se unutar neke funkcije kreira nova promenljiva, ona će biti dostupna samo unutar nje, ne i van. Takva promenljiva naziva se lokalnom, a telo funkcije - **lokalni opseg**. Suprotno tome jesu **globalni opseg** i globalne promenljive. Pod pretpostavkom da kod pišemo u dokumentu sa ekstenzijom *.py*, svaka promenljiva definisana u tom dokumentu jeste globalna, a ceo dokument predstavlja globalni opseg. Globalne promenljive su dostupne za korišćenje sa bilo kog mesta iz globalnog opsega, kao i iz svih lokalnih opsega.

Primer 92

```
deffunkcija():
x = 10# Lokalna promenljiva
print(x)

funkcija() # Ispisuje 10
print(x) # Greška! 'x' nije definisano - NameError: name 'x' is not defined
```

Primer 93

```
x = 10# Globalna promenljiva
defdupliraj():
print(x*2) # Upotreba globalne promenljive u lokalnom opsegu -> ok

dupliraj() # Ispisuje 20
```

Svaki opseg predstavlja zasebni prostor imena - sistem u kom svako ime mora biti jedinstveno, bilo da je to ime promenljive, funkcije ili klase. To znači da nije moguće u istom opsegu kreirati npr. promenljivu x, a zatim i funkciju x, ili još jednu promenljivu x. S druge strane, moguće je kreirati *globalnu* promenljivu x, a zatim i *lokalnu* promenljivu x unutar neke funkcije. Funkcija će u tom slučaju podrazumevano koristiti svoju lokalnu promenljivu.

Primer 94

```
x = 300# Globalno x
defmyfunc():
x = 200# Lokalno x
print(x) # Ispisuje lokalno x

myfunc() # Ispisuje 200
print(x) # Ispisuje 300 (globalno x)
```

Ukoliko je potrebno menjati vrednost globalne promenljive unutar lokalnog opsega, neophodno je iskoristiti ključnu reč *global* i time naznačiti da je u upotrebi globalna promenljiva.

Primer 95

```
x = 10# Globalna promenljiva
defdupliraj():
globalx
x = x*2

dupliraj()
print(x) # Ispisuje 20
```

Moduli

Svaki dokument koji sadrži *Python* funkcije, klase i promenljive predstavlja jedan modul. Pri tome bi svaki modul trebalo da sadrži logički povezane elemente, kako bi kod bio što bolje i smislenije organizovan. Moduli se dele na **ugrađene** i **korisnički definisane**. Bilo koji fajl sa *Python* kodom koji sačuvamo sa ekstenzijom *.py* predstavlja jedan korisnički definisani modul. Kao primer ugrađenog modula u ovom poglavlju će biti obrađen *datetime* modul, ali će u nastavku biti korišćeni još mnogi drugi, po potrebi. Kada nam zatrebaju funkcije definisane u nekom modulu, moguće je uvesti ih prostim uključivanjem modula u trenutni dokument (skriptu). Takođe, moguće je uključiti i samo jednu ili tek nekoliko funkcija/promenljivih iz određenog modula, ukoliko se on neće koristiti ceo. Sintaksa za uključivanje modula i korišćenje njegovih elemenata je sledeća:

```
import imeModula

imeModula.imeFunkcije()
```

Ukoliko se uključuju pojedinačni elementi, oni se koriste samo preko svog naziva. Nije potrebno navođenje i naziva modula, kao u primeru iznad. Kako ovo skraćuje pisanje imena, moguće je uključiti sve elemente iz nekog modula, navođenjem asteriksa (*).

```
from imeModula import imeFunkcije

imeFunkcije()

from imeModula import *
```

Napomena

Skidanje (i instaliranje) eksternih paketa/modula izvršava se naredbom ***pip install naziv_paketa*** u terminalu OS-a.

datetime

U *Python*-u je rad sa datumima i vremenom omogućen preko posebnog modula - *datetime*. Ovaj modul sadrži klase za rad sa vremenskim podacima, koji se, dakle, svodi na kreiranje i rukovanje objektima. Postoji ukupno šest klasa - *timedelta, tzinfo, timezone, date, time* i *datetime* - međutim, kako su poslednje tri najviše u upotrebi, u nastavku će biti obrađene samo one.

Klasa ***date*** služi za kreiranje objekata koji će opisivati neki datum, te sadrži polja za godinu, mesec i dan. Prilikom kreiranja novog datuma, ovo su upravo i obavezni parametri koje je potrebno proslediti konstruktoru kao cele brojeve (*int*). Datum će zatim biti dostupan u formatu YYYY-MM-DD. Pomoću metode *today*()moguće je dobiti trenutni datum i zatim po potrebi pristupati njegovim poljima: godina (*year*), mesec (*month*), dan (*day*).

Primer 96

```
fromdatetimeimportdate

datum = date(2000, 1, 1)
print(datum) # Ispisuje '2000-01-01'

danas = date.today()
print(danas) # Ispisuje '2023-08-23'

print("Trenutna godina: ", danas.year) # Ispisuje 'Trenutna godina:  2023'
print("Trenutni mesec: ", danas.month) # Ispisuje 'Trenutni mesec:  8'
print("Trenutni dan u mesecu: ", danas.day) # Ispisuje 'Trenutni dan u mesecu:
23'
```

Klasa ***time*** služi za kreiranje objekata koji će opisivati neki trenutak u vremenu, te sadrži polja za sate, minute, sekunde i mikrosekunde. Vreme će se ispisivati u formatu hh:mm:ss.ms. Prilikom kreiranja novog objekta ovog tipa, moguće je proslediti vrednosti (cele brojeve - *int*) za sve ove atribute, samo neke od njih ili nijedan. Ukoliko se za neki atribut ne zada vrednost, ona će podrazumevano biti nula. Vrednosti atributa će zatim biti dostupne preko kreiranog objekta i polja: sati (*hour*), minuti (*minute*), sekunde (*second*), mikrosekunde (*microsecond*).

Primer 97

```
fromdatetimeimporttime

vreme1 = time()
print(vreme1) # Ispisuje '00:00:00

vreme2 = time(hour=14)
print(vreme2) # Ispisuje '14:00:00'

vreme3 = time(second=30)
print(vreme3) # Ispisuje '00:00:30'

vreme4 = time(17, 42, 15, 500)
print(vreme4) # Ispisuje '17:42:15.000500'

print("Sati ima", vreme4.hour) # Ispisuje 'Sati ima 17'
print("Minuta ima", vreme4.minute) # Ispisuje 'Minuta ima 42'
print("Sekundi ima", vreme4.second) # Ispisuje 'Sekundi ima 15'
print("Mikrosekundi ima", vreme4.microsecond) # Ispisuje 'Mikrosekundi ima
500'
```

Klasa ***datetime*** objedinjuje informacije o datumu i vremenu u jedan objekat, te sadrži polja za: godinu, mesec, dan, sate, minute, sekunde i mikrosekunde. Sva polja moraju biti celobrojne vrednosti ukoliko se zadaju kao parametri prilikom kreiranja objekta, pri čemu su samo godina, mesec i dan obavezni. Vrednosti izostavljenih polja će podrazumevano biti nula. Pun format za dan i vreme je YYYY-MM-DD hh:mm:ss.ms. Nakon kreiranja objekta, moguće je pojedinačno pročitati vrednost svakog polja.

Primer 98

```
fromdatetimeimportdatetime

dt1=datetime(1999, 12, 12)
print(dt1) # Ispisuje '1999-12-12 00:00:00'

dt2=datetime(1999, 12, 12, 12, 12, 12, 342380)
print(dt2) # Ispisuje '1999-12-12 12:12:12.342380'

print("Godina: ", dt2.year) # Ispisuje 'Godina:  1999'
print("Mesec: ", dt2.month) # Ispisuje 'Mesec:  12'
print("Dan: ", dt2.day) # Ispisuje 'Dan:  12'
print("Sati: ", dt2.hour) # Ispisuje 'Sati:  12'
print("Minute: ", dt2.minute) # Ispisuje 'Minute:  12'
print("Sekunde: ", dt2.second) # Ispisuje 'Sekunde:  12'
print("Mikrosekunde: ", dt2.microsecond) # Ispisuje 'Mikrosekunde:  342380'
```

Slično kao što se preko klase *date* može dobiti današnji datum, tako se preko klase *datetime* može dobiti današnji datum i trenutno vreme i to pomoću metode *now*().

Primer 99

```
fromdatetimeimportdatetime

sadasnji_trenutak=datetime.now()
print(sadasnji_trenutak) # Ispisuje 2023-08-25 14:18:41.076007
```

Rad sa datotekama

Za rad sa datotekama u *Python*-u ključna je metoda *open*() koja se najčešće poziva sa dva parametra: naziv datoteke i način (engl. *mode*) otvaranja. Modovi otvaranja su:

- `"r"` – *Read*. Otvara datoteku za čitanje. Baca grešku ako datoteka ne postoji.
- `"a"` – *Append*. Otvara datoteku za dodavanje teskta. Kreira datoteku ako ne postoji.
- `"w"` – *Write*. Otvara datoteteku za pisanje. Kreira datoteku ako ne postoji.
- `"x"` – *Create*. Kreira datoteku sa zadatim nazivom. Baca grešku ako datoteka već postoji.

Dodatno, moguće je specificirati da li se sadržaj datoteke tretira kao binarni (`"b"`) ili tekstualni (`"t"`).

Podrazumevana vrednost za mod je `"rt"`, što znači da se datoteke otvaraju za čitanje sadržaja u tekstualnom formatu.

Sadržaj možemo čitati u celini, liniju po liniju ili samo određeni broj karaktera. Za sva tri slučaja koristimo *read*() ili *readline*() metode.

Na kraju rada sa datotekom, potrebno je zatvoriti je, odnosno pozvati *close*() metodu.

Primer 100

```
f = open("demo.txt", "r")
print(f.read())
# Ispisuje:
# Ovo je demo rada sa datotekama u Python-u.
# Ovo je novi red.
f.close()

f = open("demo.txt", "r")
print(f.readline()) # Ispisuje: Ovo je demo rada sa datotekama u Python-u.
f.close()
```

Kako bismo dodali još sadržaja, potrebno je datoteku otvoriti u *append* modu. Ukoliko želimo da prepišemo postojeći sadržaj i upišemo novi, otvorićemo je u *write* modu. Oba moda će kreirati datoteku sa zadatim nazivom ukoliko prethodno nije postojala.

Primer 101

```
f = open("demo.txt", "a")
f.write("\nDodali smo još jedan red!")
f.close()

f = open("demo.txt", "r")
print(f.read())
# Ispisuje:
# Ovo je demo rada sa datotekama u Python-u.
# Ovo je novi red.
# Dodali smo još jedan red!
f.close()
```

Primer 102

```
f = open("demo.txt", "w")
f.write("Ovo ce sada biti jedini red!")
f.close()
```

```
f = open("demo.txt", "r")
print(f.read()) # Ispisuje: Ovo ce sada biti jedini red!
f.close()
```

Za brisanje datoteke, neophodno je uvesti **os** modul i pozvati njegovu *remove*() metodu. S obzirom na to da ta metoda baca grešku ukoliko zadataka datoteka ne postoji, dobra je praksa to prvo proveriti.

Primer 103

```
import os
if os.path.exists("demo.txt"):
  os.remove("demo.txt")
else:
  print("Trazena datoteka ne postoji.")
```

Serijalizacija

Serijalizacija je proces prebacivanja objekata u niz bajtova. Taj niz se dalje može upisati u nekudatoteku ili poslati preko mreže. Obrnuti proces, odnosno rekonstrukcija objekta od niza bajtova, naziva se deserijalizacija. U ove svrhe koristi se *Python* modul ***pickle,*** sa metodama *dumps*() za serijalizaciju i *loads*() za deserijalizaciju.

Primer 104

```
classStudent:
def__init__(self, ime, prezime, broj_indeksa, prosek):
self.ime=ime
self.prezime=prezime
self.broj_indeksa=broj_indeksa
self.prosek=prosek

def__str__(self) ->str:
returnf"{self.broj_indeksa}{self.prezime}{self.ime}, prosek: {self.prosek}"

importpickle
student=Student("Jovan", "Jovanović", "PR23/2023", 9.0)
print(student) # Ispisuje 'PR23/2023 Jovanović Jovan, prosek: 9.0'

b_student=pickle.dumps(student) # Serijalizacija
print(type(b_student)) # Ispisuje '<class 'bytes'>'

student2=pickle.loads(b_student) # Deserijalizacija
print(student2) # Ispisuje 'PR23/2023 Jovanović Jovan, prosek: 9.0'
```

U ovom primeru je definisana klasa **Student**, a zatim i instanciran objekat te klase (*student*). Uvezen je modul *pickle*, te je objekat prvo serijalizovan (*dumps*), a zatim i deserijalizovan (*loads*). Može se primetiti kako je rezultat serijalizacije tipa *bytes*, a da je rezultat deserijalizacije isti objekat koji smo imali i pre serijalizacije.

Ukoliko želimo da serijalizovani objekat upišemo u datoteku, koristićemo *dump*() metodu, dok ćemo za iščitavanje objekta u binarnom formatu i njegovu deserijalizaciju koristiti metodu *load*(). Za to je potrebno otvoriti željenu datoteku u odgovarajućem modu (`"w"`/`"r"`) i naglasati da se radi sa podacima u binarnom formatu (`"b"`).

Primer 105

```
class Student:
    def __init__(self, ime, prezime, broj_indeksa, prosek):
        self.ime = ime
        self.prezime = prezime
        self.broj_indeksa = broj_indeksa
        self.prosek = prosek

    def __str__(self) -> str:
        returnf"{self.broj_indeksa}{self.prezime}{self.ime}, prosek:
{self.prosek}"

import pickle

student = Student("Jovan", "Jovanovic", "PR23/2023", 9.0)
print(student) # Ispisuje 'PR23/2023 Jovanović Jovan, prosek: 9.0'

f = open("student.txt", "wb")
pickle.dump(student, f)
f.close()

f = open("student.txt", "rb")
student2 = pickle.load(f) # Deserijalizacija
print(student2) # Ispisuje 'PR23/2023 Jovanović Jovan, prosek: 9.0'
f.close()
```

JSON

JSON je tekstualni format za skladištenje i razmenu podataka. Naziv je akronim od ***JavaScript Object Notation***, zato što se objekti opisuju po *JavaScript* notaciji. Ovaj format je i ljudski lako čitljiv, ali i mašinski, što znači da je jednostavno parsirati datoteke u ovom formatu i od pročitanogteksta kreirati objekte, kao i obrnuto – jednostavno je objekte predstaviti u JSON formatu i upisati ih u JSON datoteku. Tako *Python*, kao i svaki moderni programski jezik, ima ugrađenu podršku za rad sa JSON formatom (*json* modul), što ćemo i prikazati kroz naredni primer. Kao i u *pickle* modulu, metode *dumps*() i *loads*() služe za konvertovanje iz *Python* objekta u string i obrnuto, a metode *dump*() i *load*() za konvertovanje i upis u datoteku/čitanje iz datoteke.

Primer 106

```
import json

recnik = {
  "ime": "Jovan",
  "godine": 30,
  "student": False,
  "kola": [
    {"model": "BMW 230", "boja": "crna"},
    {"model": "Ford Edge", "boja": "crvena"}
  ]
}

print(recnik)
print(type(recnik)) # Ispisuje: <class 'dict'>

recnik_json = json.dumps(recnik)
print(recnik_json)
```

```
print(type(recnik_json)) # Ispisuje: <class 'str'>

nazad_u_recnik = json.loads(recnik_json)
print(nazad_u_recnik)
print(type(nazad_u_recnik)) # Ispisuje: <class 'dict'>

f = open("recnik.json", "w")
json.dump(recnik, f)
f.close()

f = open("recnik.json")
recnik2 = json.load(f)
print(recnik2)
f.close()
```

Kako bismo objekat korisnički definisanog tipa upisali u JSON datoteku, potrebno je pristupiti njegovom *__dict__* atributu, koji sadrži sva polja tog objekta u parovima ključ-vrednost (tj. naziv polja-vrednost polja). Povratak iz rečnika u objekat omogućen je upotrebom *setattr()* metode.

Primer 107

```
import json
import json

class Student:
    def __init__(self, ime, prezime, broj_indeksa, prosek):
        self.ime = ime
        self.prezime = prezime
        self.broj_indeksa = broj_indeksa
        self.prosek = prosek

    def __str__(self) -> str:
        returnf"{self.broj_indeksa}{self.prezime}{self.ime}, prosek:
{self.prosek}"

student = Student("Jovan", "Jovanovic", "PR23/2023", 9.0)
f = open("student.json", "w")
json.dump(student.__dict__, f)
f.close()

f = open("student.json")
student_json = json.load(f) # Ovo će biti rečnik.
student2 = Student("", "", "", 0)
for key, value in student_json.items(): setattr(student2, key, value)
print(student2) # Ispisuje: PR23/2023 Jovanovic Jovan, prosek: 9.0
```

Zadaci

1. Kreirati datoteku naziva „zadatak1.txt“ i u nju upisati svoj broj indeksa, ime i prezime.
2. Kreirati datoteku naziva „zadatak2.txt“ i upotrebom *while* petlje u nju upisivati korisnički unos sa terminala. Ručno izmeniti datoteku, zatim pročitati ceo sadržaj i ispisati ga.
3. Čitati sadržaj datoteke „zadatak3.txt“ liniju po liniju i za svaki pročitani red instancirati po jedan objekat klase **Student** iz datoteke *student.py* i dodati ga u listu studenata. Nakon završenog čitanja, ispisati sve studente iz liste.
4. Serijalizovati listu iz prethodnog zadatka i upisati je u datoteku „zadatak4.txt“.

5. Napisati klasu **Ispit**, koja sadrži sledeća polja: listu učionica u kojima se održava ispit, naziv predmeta i ime profesora. Kreirati objekat tipa Ispit za predmet Osnove distribuiranog programiranja, koji će se održati u učionicama 109, 109a i 108. Kreirani objekat upisati u datoteku „zadatak5.json“.
6. Učitati sadržaj „zadatak6.json“ datoteke. Pročitano ispisati u formatu *ključ* : *vrednost*.

Vežba 4

Klijent-server arhitektura

U *Python*-u je osnovna komunikacija između dve aplikacije omogućena *socket* modulom. Soketje jedan kraj dvosmernog komunikacionog kanala između dva procesa. Sadrži informacije o IP adresi i portu aplikacije. U kontekstu klijent-server arhitekture pričamo o serverskom i klijentskom soketu. Serverski soket će biti tačka pristupa serverskoj aplikaciji, koja služi za pružanje nekih usluga, tj. obrađivanje zahteva pristiglih od strane klijenata. Samim tim je klijentska aplikacija ona koja zahteve šalje, a klijentski soket tačka preko koje se uspostavlja komunikacija sa serverom.

Osnovni primer klijentskog i serverskog programa nalazi se u nastavku. Objašnjenje koda je navedeno nakon primera.

Primer serverske aplikacije

```
import socket

server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.bind(('localhost', 6000))
server.listen()
print("Server je pokrenut.")

kanal, adresa = server.accept()
print(f"Prihvacena je konekcija sa adrese: {adresa}")

whileTrue:
    poruka = kanal.recv(1024).decode()
    ifnot poruka : break
    print(f"Primljena poruka: {poruka}")
print("Server se gasi.")
server.close()
```

Prva naredba označava uvoz potrebnog modula – *socket* modula. Prvo sledeće jeste kreiranje soketa. Prvi parametar funkcije za kreiranje jeste AF_INET i on označava IPv4 familiju adresa. Drugi parametar, SOCK_STREAM, definiše da će se komunikacija odvijati po TCP protokolu. Sledeći korak jeste podešavanje adrese soketa. Adresa se sastoji iz IP adrese i porta. U ovom primeru soket se nalazi na lokalnoj adresi mašine i proizvoljno odabranom portu 6000. Važno je da se biraju portovi iznad 1024, jer su svi od 0 do 1023 rezervisani. Skreće se pažnja na to da funkcija *bind* prihvata torku (*tuple*) kao parametar ili još preciznije – uređeni par (adresa, port). Dalje, funkcijom *listen* server se otvara za klijentske konekcije; otvara se tzv. „slušajući“ soket. Funkcija *accept* blokira izvršavanje programa dok god ne stigne zahtev za povezivanje od strane nekog klijenta.

Povratne vrednosti su novi soket objekat koji predstavlja samu konekciju i adresa klijenta. Dalje se u beskonačnoj petlji prihvataju poruke poslate od strane klijenta, sve dok ne stigne prazna poruka. Tada se petlja prekida i konekcija zatvara sa serverske strane (funkcija *close*). Ukoliko je zaista primljen neki tekst, on se ispisuje na ekran.

Skreće se pažnja na *decode* funkciju, čiji je zadatak da prihvaćeni niz bajtova vrati nazad u tekstualni tip, tj. string.

Primer klijentske aplikacije

```
import socket

klijent = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
klijent.connect(('localhost', 6000))
print("Veza sa serverom je uspostavljena.")
whileTrue:
    poruka = input("Unesite poruku:")
    ifnot poruka : break
    klijent.send(poruka.encode())
print("Konekcija se zatvara.")
klijent.close()
```

Prva naredba označava uvoz potrebnog modula – *socket* modula. Prvo sledeće jeste kreiranje soketa. Prvi parametar funkcije za kreiranje jeste AF_INET i on označava IPv4 familiju adresa. Drugi parametar, SOCK_STREAM, definiše da će se komunikacija odvijati po TCP protokolu. Zatim klijent inicira uspostavljanje konekcije sa serverom. Pri tome zadaje adresu servera, te se i ovde skreće pažnja na to da funkcija *conncet* prihvata torku (*tuple*) kao parametar ili još preciznije – uređeni par (adresa, port). U beskonačnoj petlji se od korisnika traži da unosi poruke. Petlja se prekida kada korisnik ne unese ništa, tj. pritisne Enter. Uneta poruka se kodira u niz bajtova i šalje na server.Nakon izlaska iz petlje, konekcija se zatvara i sa klijentske strane.

Zadaci

1. Napisati klijentski i serverski program, tako da obe strane mogu i da šalju i da primaju poruke, tj. tako da mogu da se međusobno dopisuju.
2. Pomoću klijent-server arhitekture napraviti „*online*“ kalkulator. Klijent šalje matematičke izraze serveru, koji ih izračuna i vrati klijentu rezultat.
3. Napisati klasu **Student** koja ima sledeća polja: ime, prezime, broj indeksa (stringovi) i prosek (realni broj); konstruktor sa parametrima i metodu za tekstualni prikaz objekta. Napisati serverski program koji čuva podatke o studentima u kolekciji tipa rečnik (ključ: broj indeksa, vrednost: objekat tipa Student) i na zahteve klijenata vraća njihove tekstualne opise. Napisati klijentski program koji od korisnika traži da unese broj indeksa traženog studenta, a zatim ispisuje podatke o studentu dobijene od strane servera.
4. Proširiti prethodni zadatak tako da klijent ima mogućnost da doda novog studenta ili izmeni postojećeg.
5. Napisati klasu **Lek** koja ima sledeća polja: id (*int*), naziv (*str*), datum_proizvodnje (*date*), sastojci (*list*). Napisati serversku aplikaciju koja pruža sledeće opcije: 1) dodaj lek, 2) izmeni lek, 3) obriši lek, 4) pročitaj lek, 5) dodaj sastojke. Lekovi se čuvaju u kolekciji tipa rečnik, pri čemu je id leka ključ, a ceo objekat (tipa Lek) vrednost. Napisati i odgovarajuću klijentsku aplikaciju. Sa klijentske strane će se za kreiranje i izmenu lekova tražiti unos vrednosti za: id, naziv i datum, a zatim će se serveru slati kreirani objekat tipa Lek. Brisanje i čitanje se vrše po id-u, kao i dodavanje sastojaka (dodaju se sastojci za lek sa zadatim id-em). Sve akcije se loguju sa serverske strane u **log.txt** datoteku.

Zadatak za samostalni rad

Napisati klasu **Profesor**, koja sadrži sledeća polja: jmbg (*str*), ime (*str*), prezime (*str*), datum izbora u zvanje (*date*), lista predmeta (*list*).

Napisati serversku aplikaciju koja skladišti objekte tipa **Profesor** u rečniku na sledeći način: ključ je jbmg, vrednost je ceo objekat. Uz to, pruža sledeće opcije svojim kljentima: 1) dodaj profesora, 2) izmeni profesora, 3) obriši profesora (po jmbg-u), 4) pročitaj profesora (po jmbg-u), 5) dodaj predmete profesoru (po jmbg-u), 6) pročitaj predmete profesora (po jmbg-u) sortirane alfabetski i 7) pročitaj sve profesore zaposlene duže od 20 godina.

Klijentska aplikacija treba da prikupi sve potrebne parametre sa terminala i da ih pošalje serveru. Za opcije 1) i 2) šalje se serijalizovani objekat tipa Profesor pročitan iz odgovarajuće txt (ili json) datoteke; za 3), 4) i 6) se šalje jmbg, za 5) se šalje jmbg, pa zatim serijalizovana lista predmeta; dok se za 7) ne šalju nikakvi parametri. Rezultat opcije 7) treba da bude tekstualna datoteka sa informacijama o profesorima zaposlenim duže od 20 godina.

Sa datotekama raditi uz obradu izuzetaka.

Vežba 5

Mehanizmi bezbednosti

Osnovni mehanizam zaštite bilo kog sistema, pa tako i distribuiranog, jeste kontrola pristupa. Cilj ovog mehanizma jeste zaštita podataka i procesa od zlonamernih aktera, kako eksternih, tako i internih. Kontrola pristupa se ostvaruje kroz autentifikaciju i autorizaciju, za šta se produkcioni sistemi uglavnom oslanjaju na gotova rešenja trećih strana. U ovoj vežbi ćemo objasniti i samostalno implementariti jednostavni vid autentifikacije i autorizacije.

Autentifikacija

Autentifikacija podrazumeva jednoznačno utvrđivanje identiteta korisnika (klijenta). Najjednostavniji (i u praksi nedovoljan) vid autentifikacije jeste putem korisničkih imena i lozinki, odnosno jednofaktorska autentifikacija.Korisničkim imenom se korisnik predstavlja (identifikuje) sistemu, a lozinkom koju je odabrao prilikom registracije i od tada čuvao kao tajnu, dokazuje taj identitet. Na taj način sistem utvrđuje da je korisnik baš taj za kog se izdaje. Dakle, gledano sa strane sistema (servera), potrebno je 1) na odgovarajući način skladištiti podatke o korisnicima (parove korisničko ime-lozinka) i zatim prilikom prihvatanje nove konekcije 2) tražiti od korisnika da unese korisničko ime i lozinku i 3) proveriti da li se ti podaci poklapaju sa onima u bazi. Ukoliko je korisnik uspešno autentifikovan, odobrava mu se pristup sistemu i metodama servisa, u skladu sa pravima pristupa koja su mu dodeljena.

S obzirom na to da server u bazi skladišti tajne podatke (korisničke lozinke), oni ne smeju ostati u tzv. „čistom tekstu“ (engl. *plaintext*), već se moraju prebaciti u nečitljivi format. U praksi se za to koriste *hash* funkcije – jednosmerne funkcije koje ulaz proizvoljne veličine pretvaraju u izlaz fiksne veličine, tako da isti ulaz uvek rezultuje istim izlazom. Pri tome su ulazi u *hash* funkcije konkatenacije korisničkih lozinki sa nasumično generisanim stringovima, čime se izbegava da iste lozinke rezultuju istim *hash* vrednostima. Mi ćemo se, jednostavnosti radi, zadržati samo na upotrebi *hash* funkcije.

Uputstvo za implementaciju:

1. Kreirati klasu **Korisnik**, sa poljiima za korisničko ime, lozinku i oznaku da li je korisnik autentifikovan ili nije, čija će podrazumevana vrednost biti *False*.
2. U novom modulu (NE klasi) naziva **direktorijum_korisnika** inicijalizovati rečnik korisnika i definisati sledeće metode:
    a. **hesiranje** – metoda prihvata neki tekst (str) kao ulaz i vraća heš vrednost tog stringa (takodje str); koristiti modul *hashlib* i *sha256* funkciju
    b. **dodaj_korisnika –** metoda prihvata korisničko ime i lozinku kao parametre i u rečnik ubacuje novokreiranog korisnika, tako da korisničko ime bude ključ; prilikom kreiranja korisnika se za lozinku prosleđuje njena heš vrednost
    c. **autentifikacija –** metoda prihvata korisničko ime i lozinku kao parametre i proverava da li se korisničko ime nalazi u rečniku, kao i da li se lozinke poklapaju; ukoliko su provere uspešne, korisnik se obeležava kao autentifikovan i vraća se *True*, u suprotnom se vraća *False*
3. Na **klijentskoj strani** je nakon uspostavljanja konekcije potrebno od korisnika tražiti korisničko ime i lozinku i zatim ih prosleđivati serveru na proveru. Raditi ovo dok god server ne potvrdi da je autentifikacija uspešna.

4. Na **serverskoj strani** je pre svega potrebno dodati nekoliko korisnika u rečnik („bazu“). Zatim se, nakon prihvatanja konekcije, proveravaju kredencijali pristigli sa klijentske strane i klijentu vraća rezultat autentifikacije kao odgovor.

Autorizacija

Dodela prava pristupa korisnicima je tehnika koju nazivamo autorizacija. Provera tih prava prilikom pokušaja pristupa nekim podacima ili funkcionalnostima je upravo provera da li je korisnik autorizovan za taj pristup ili nije. Ograničavanjem pristupa validnim korisnicima sistema smanjuje se rizik od slučajnog curenja informacija, kao i od insajderskih pretnji, odnosno zlonamernih zaposlenih.

U našem sistemu će se za svaku akciju (metodu) vezivati jedno pravo, tj. dozvola (permisija) da se ta akcija izvrši. Tako će korisnici imati svako svoju listu prava u zavisnosti od toga koja je njihova uloga u sistemu.

Uputstvo za implementaciju:

1. Proširiti klasu **Korisnik** poljem za listu prava.
2. Proširiti metodu **dodaj_korisnika** parametrom za listu prava.
3. Na **serverskoj strani**:
    a. Prilikom dodavanja korisnika definisati koja prava imaju.
    b. Prilikom uspešne autentifikacije zapamtiti trenutno ulogovanog korisnika.
    c. Prilikom prihvatanja opcije od klijenta, proveriti da li trenutni korisnik ima odgovarajuće pravo i vratiti adekvatan odgovor. Dozvoliti izvršenje akcije pod tom opcijom isključivo ukoliko je korisnik uspešno autorizovan.

Zadaci za samostalni rad

1. Izmeniti metodu za autentifikaciju tako da se u slučaju uspešne autentifikacije generiše i token (str), koji se šalje nazad korisniku i pomoću kog se on dalje predstavlja, tj. šalje ga uz svaku poruku serveru. U skladu sa tim dopuniti klasu Korisnik poljem za token, kao i servera, koji će svaki put proveravati da li token pripada nekom od autentifikovanih korisnika.
2. Zameniti programsko dodavanje korisnika sa čitanjem iz „baze“ tj. tekstualne datoteke (probati i sa .txt i sa .json formatom).

Vežba 6

Replikacija

Replikacija se uvodi u distribuirane sisteme radi poboljšanja preformansi sistema i povećavanja dostupnosti istog. Moguća je (1) replikacija hardvera uvođenjem dodatnog hardvera koji stoji na raspolaganju ukoliko dođe do kvara opreme, (2) replikacija servisa kreiranjem više od jedne instance procesa koji pruža određenu uslugu u distribuiranom sistemu, odnosno (3) replikacija podataka u cilju izbegavanja gubitka podataka ukoliko dođe do kvara opreme ili softvera. Cilj ovih vežbi je da se prikažu sva tri tipa replikacije, sa naglaskom na replikaciji podataka.

Zadaci:

1. Proširiti datu klijentsku i serversku aplikaciju sa 2 dodatne opcije:
   (1) BACKUP – Ova opcija omogućava klijentu da u željenom trenutku trajno sačuva stanje baze. Dakle, potrebno je da pri odabiru ove opcije server sve podatke iz rečnika upiše u tekstualnu datoteku naziva „*backup.txt*“.
   (2) REPLICATE – Ova opcija podrazumeva pravljenje kopije backup fajla. Klijent zadaje putanju na koju želi da server kopira backup fajl (npr. „*kopija.txt*“).
2. Cilj ovog zadatka jeste uvođenje replike servisa; odnosno pokretanje sekundarnog servisa pored primarnog (koji je do sada bio i jedini). Potrebno je proširiti klijentsku aplikaciju, tako da pre svega može da se poveže na dva servisa, a zatim i da može da vrši replikaciju, tako što čita sve podatke iz baze primarnog servisa i upisuje ih na sekundarni. Dakle, i server će biti proširen READ_ALL i WRITE_ALL opcijama.
   Testirati rešenje na tri fizički odvojena računara (klijent, primarni server, sekundarni server).
3. Proširiti drugi zadatak, tako da se pročitano stanje sa primarnog servera upiše u jedan fajl, a zatim i upisano stanje na sekundarni server u drugi fajl. Fajlovi treba da budu na različitim lokacijama.

Zadaci za samostalni rad:

1. Prethodno rešenje proširiti tako da samo klijent sa pravom REPLICATE može da poziva opciju za replikaciju podataka.
2. Neka replikaciju sada vrši odvojeni klijent, na određeni vremenski period. Moguće nadogradnje: (1) replicirati samo izmene od poslednje replikacije (nove i promenjene podatke); (2) osmisliti način za repliciranje brisanja.

Vežba 7

Otpornost na otkaze

U distribuiranim sistemima se otpornost na otkaze povećava uvođenjem bezbednosnih kopija, tj. replika. Moguće je replicirati hardver, softver (tj. servise) i podatke. U prethodnim vežbama je pokazano jedno jednostavno rešenje za replikaciju podataka. Cilj ove vežbe je da pokaže da dodavanje rezervnog servisa (tj. dodatnog serverskog procesa) omogućava distribuiranom sistemu da se automatski oporavi od jednostrukog ispada servisa.

Redundantni servisi

Ukoliko u DS postoji veći broj servisa, onda oni mogu biti ravnopravni, tako da se zahtevi klijenata dele između njih. Takva implementacija je karakteristična za veb servere, koji pokreću veći broj paralelnih procesa za opsluživanje zahteva. Servisi u industrijskim sistemima su uglavnom složeniji i njihove redundantne konfiguracije se najčešće sastoje od primarnog servisa i jednog ili više rezervnih servisa. Klijenti u normalnom radu komuniciraju sa primarnim servisom. Ukoliko dođe do njegovog ispada, njegovu ulogu preuzima jedan (ili jedini) rezervni servis. U ovim vežbama ćemo prikazati upravo ovaj način replikacije servisa.

Praćenje stanja servisa

Serveri su potpuno nezavisni i u zavisnosti od njihove konfiguracije imaju različita, ali potencijalno i ista unutrašnja stanja, npr. dva primarna servera. Postojanje dva primarna servera je često neželjena pojava i na engleskom se naziva „*split brain*“, jer u tom slučaju imamo dva procesa koja mogu imati različite podatke (tj. imati nekonzistentne podatke) i opsluživati podskup klijenata u distribuiranom sistemu.

Za rešavanje gore navedenog problema, pa i njemu sličnih problema se često uvodi dodatni proces, koji nadzire status svih servisa i upravlja sa njima. U kontekstu operativnih sistema se takvi upravljački procesi nazivaju *„watchdog*“ procesima.

Zadatak 1.

Omogućiti uvođenje rezervnog servisa za obradu podataka o lekovima. Omogućiti sledeća stanja servisa: nepoznato, primarni i sekundarni. Dodati nove opcije za: 1) proveru statusa servisa (tj. da li je aktivan) i 2) ažuriranje stanja servisa (tj. postavljanje jednog od gore navedena tri stanja). Pokrenuti jedan primarni i jedan sekundarni servis.

Dodati proces koji nadzire i podešava stanja servisa. Nakon pokretanja sa povezuje na oba servisa, jedan podešava da bude primarni, drugi sekundarni. Periodično proverava stanja oba servisa. Detektuje ispad primarnog servisa ukoliko ne uspe da očita njegovo stanje i proglašava sekundarni servis primarnim.

Smernice za rešavanje zadatka:

1. Krenuti od datog klijent-server rešenja.
2. Na server dodati globalnu promenljivu *stanje* i podesiti je da bude prazan string.
3. Na server dodati dve nove opcije: GET_STATE i SET_STATE. Prva će kao odgovor slati vrednost globalne promenljive *stanje*, dok će druga prihvatati novu vrednost za stanje i dodeljivati je pomenutoj globalnoj promenljivoj.
4. Dodati novu komponentu - monitor.py.
5. Povezati je na oba servisa (uz obradu izuzetaka). Prvom ažurirati stanje na **primarni**, a drugom na **sekundarni**.
6. U beskonačnoj petlji proveravati stanje oba servisa, ispisati ih, a zatim ukoliko je prvi servis u stanju **nepoznato**, a sekundarni u stanju **sekundarni**, ažurirati stanje sekundarnog na **primarni**. ***Napomena: na kraju svake iteracije, zaustaviti program na 5 sekundi.***
7. Ukoliko je i sekundarni servis u stanju **nepoznato**, znači da smo izgubili oba servisa, te je potrebno zatvoriti konekciju i sa strane monitora i ugasiti taj proces.

Smernice za testiranje zadatka:

1. Pokrenuti oba servisa, monitor i pratiti ispis.
2. Zaustaviti primarni servis i pratiti ispis.
3. Zaustaviti sekundarni servis i pratiti ispis.

Zadatak 2.

Omogućiti (ponovno) povezivanje na servise unutar beskonačne petlje. Ovim se dodaje podrška za privremeni ispad i ponovno pokretanje jednog ili oba servisa.

Klijenti redundantnih servisa

Naglasak gore navedenih primera je bio na postojanju redundatnih servisa i na njihovom nadzoru i konfigurisanju. Korisnik tih servisa nije menjan i ne sadrži logiku za prevezivanje na rezervni servis u slučaju ispada primarnog, aktivnog servisa.

Zadatak 3.

Unaprediti korisnika usluga gore implementiranih servisa (tj. klijenta) dodavanjem koda za povezivanje na sekundarni servis, te i nastavka rada sa istim ukoliko dođe do otkaza primarnog. Za testiranje pokrenuti dve instance servisa i jednu instancu klijenta.

Zadaci za samostalni rad:

1. Omogućiti rad sa većim brojem ravnopravnih servisa od kojih se samo jedan proglašava primarnim i svi ostali sekundarnim, tj. rezervnim servisima. Adekvatno izmeniti i klijenta, tj. uopštiti rešenje trećeg zadatka na proizvoljan broj servisa.
2. Onemogućiti pristup servisima koji nisu u odgovarajućem stanju (*primarni*).

Vežba 8

Zadaci za vežbanje

1. Na dati templejt (*V8 template.zip*) dodati mogućnost prijave klijenata na sistem (autentifikaciju). Definisati validne korisnike proizvoljno. Zatim, klijentu omogućiti povezivanje sa dva servisa i replikaciju podataka sa prvog na drugi. Sve raditi uz obradu izuzetaka. Logovati sve akcije u sistemu u *log.txt* datoteku.
2. Na dati templejt (*V8 template.zip*) dodati mogućnost replikacije podataka sa primarnog na sekundarni servis, koju će inicirati klijent. Stanja servisa podešava monitor prilikom svog pokretanja (i nakon konekcije na oba servisa), a zatim portove primarnog i sekundarnog servera šalje klijentu, kako bi on znao sa kojim serverom primarno komunicira, a na koji replicira podatke. Sve raditi uz obradu izuzetaka. Logovati sve akcije u sistemu u *log.txt* datoteku.

   Napomena: Nakon što monitor iskomunicira sa servisima, zatvoriti te kanale. Nakon što server iskomunicira sa monitorom, zatvoriti kanal i sa te strane. Nakon što klijent iskomunicira sa monitorom, zatvoriti konekciju.
3. *(Za samostalni rad)* Implementirati distribuirani sistem koji skladišti informacije o fizičkim licima. Potrebno je:
   a. Kreirati modul za klasu **FizickoLice** sa poljima za **jmbg**(*str*), **ime**(*str*) i **prezime**(*str*), kao i metodom za ispis objekta.
   b. Kreirati modul **server.py**, u kom će biti implementirana funkcionalnost servera. Server skladišti objekte tipa **FizickoLice** u **rečniku**, pri čemu je **jmbg ključ** rečnika, a ceo **objekat vrednost**. Server pruža sledeće opcije: 1) Dodaj novo lice, 2) Izmeni postojeće lice, 3) Pročitaj postojeće lice, 4) Obriši postojeće lice, 5) Pročitaj sva lica iz rečnika. Opcije 1 i 2 prihvataju serijalizovane objekte tipa **FizickoLice**, dok opcije 3 i 4 prihvataju samo **jmbg** traženog lica. Opcija 5 ne prihvata parametre, a vraća serijalizovanu **listu** lica, sortiranu po prezimenima.
   c. Pristup metodama servisa dozvoliti samo validnim korisnicima sistema, koji imaju odgovarajuća prava: **ADD** za dodavanje, **EDIT** za izmenu i brisanje, **READ** za čitanje i **READALL** za čitanje svih lica. Bazu korisnika zadati u **korisnici.txt** datoteci.
   d. Implementirati klijentsku aplikaciju u modulu **klijent.py**, tako da korisnik može da se uloguje i odabere neku od opcija dostupnih na serveru.
   e. Dodatno, omogućiti replikaciju podataka na rezervnu instancu servera, dodavanjem opcije 6) Replikacija rečnika. Pozivanje ove opcije omogućiti samo korisnicima sa pravom **REPLICATE**.
   f. Klijent informaciju o tome koji servis je primarni, a koji sekundarni dobija od monitora, čiji je zadatak da ta stanja prvobitno i podesi.
   g. Sve raditi uz obradu zadataka. Logovati sve akcije u *log.txt* datoteku.

Primer

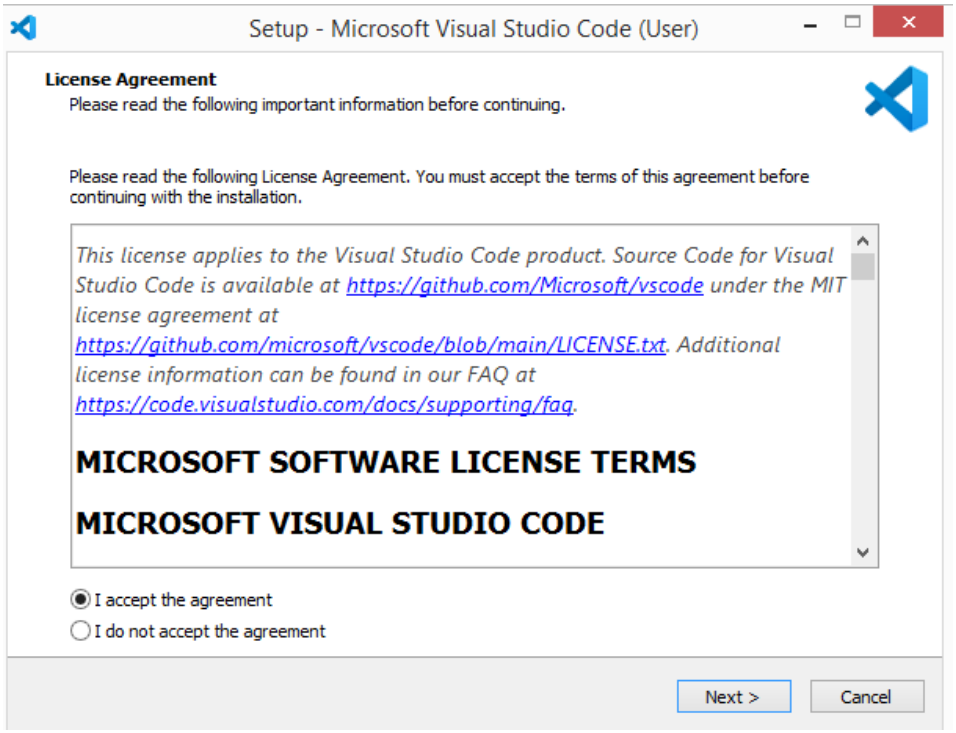
Java kod

```
publicclassHelloWorld
{
   publicstaticvoidmain (String[] args)
   {
      System.out.println("Hello, world!");
   }
}
```

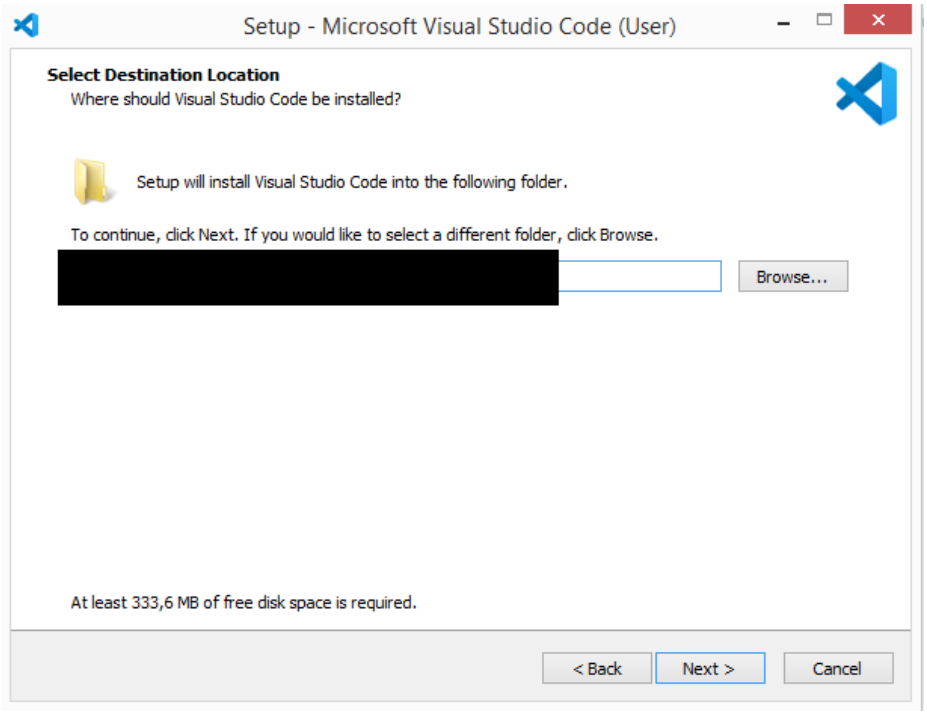
Python kod

```
print("Hello, world!")
```

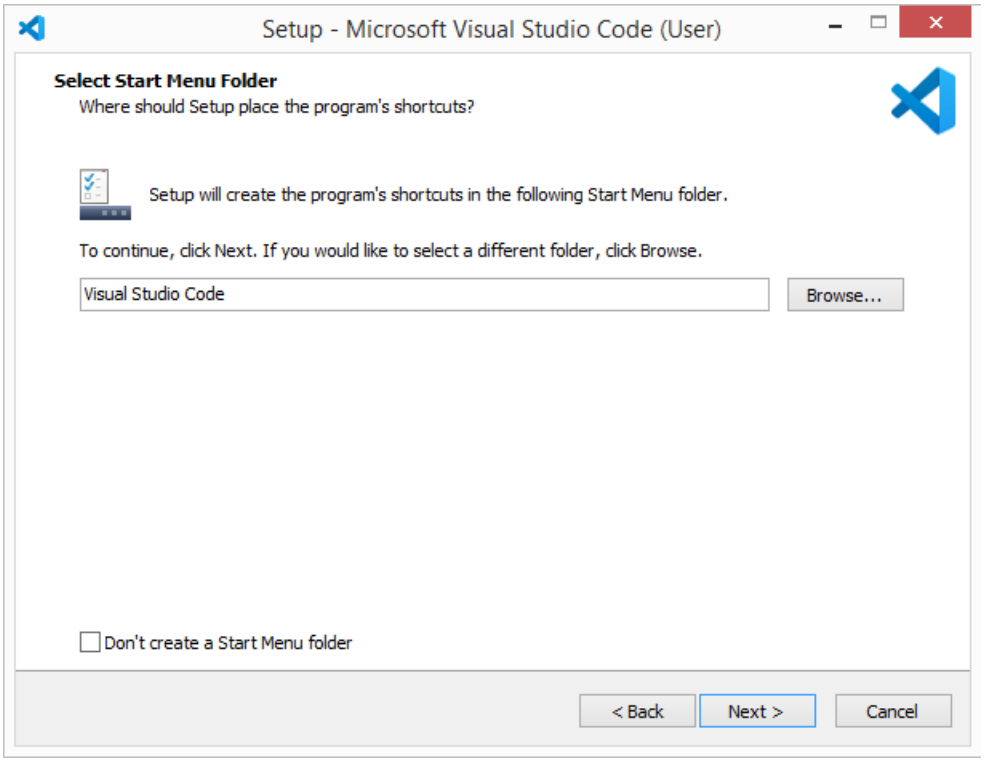
Slike



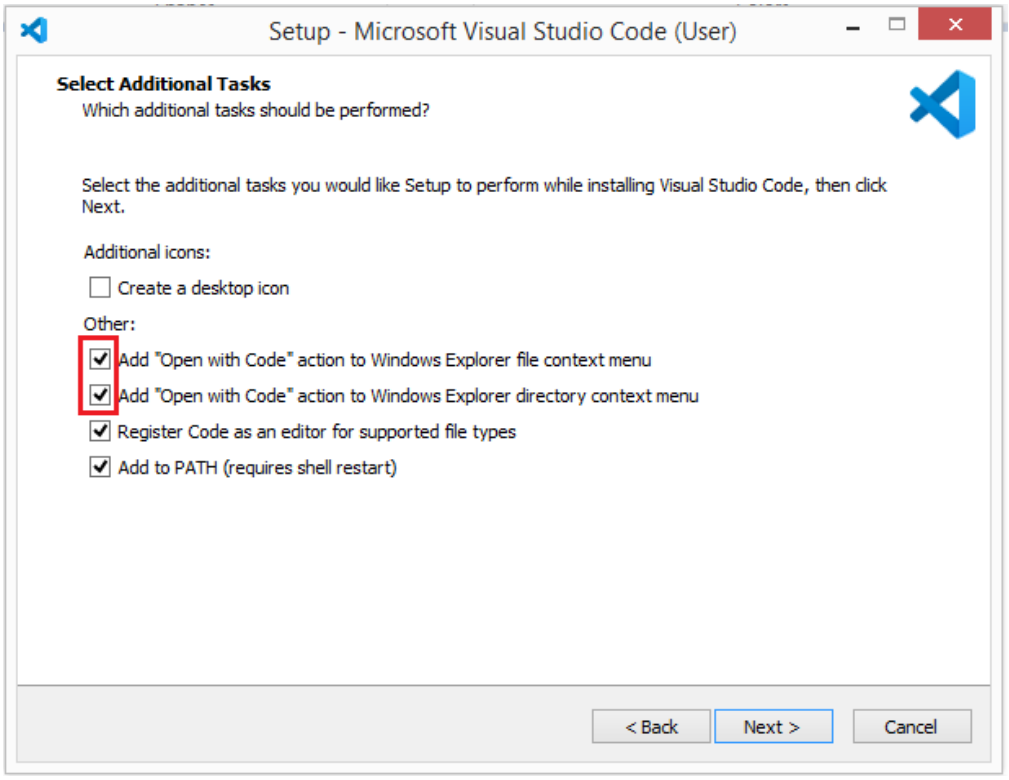
Slika 1. Prihvatanje uslova korišćenja



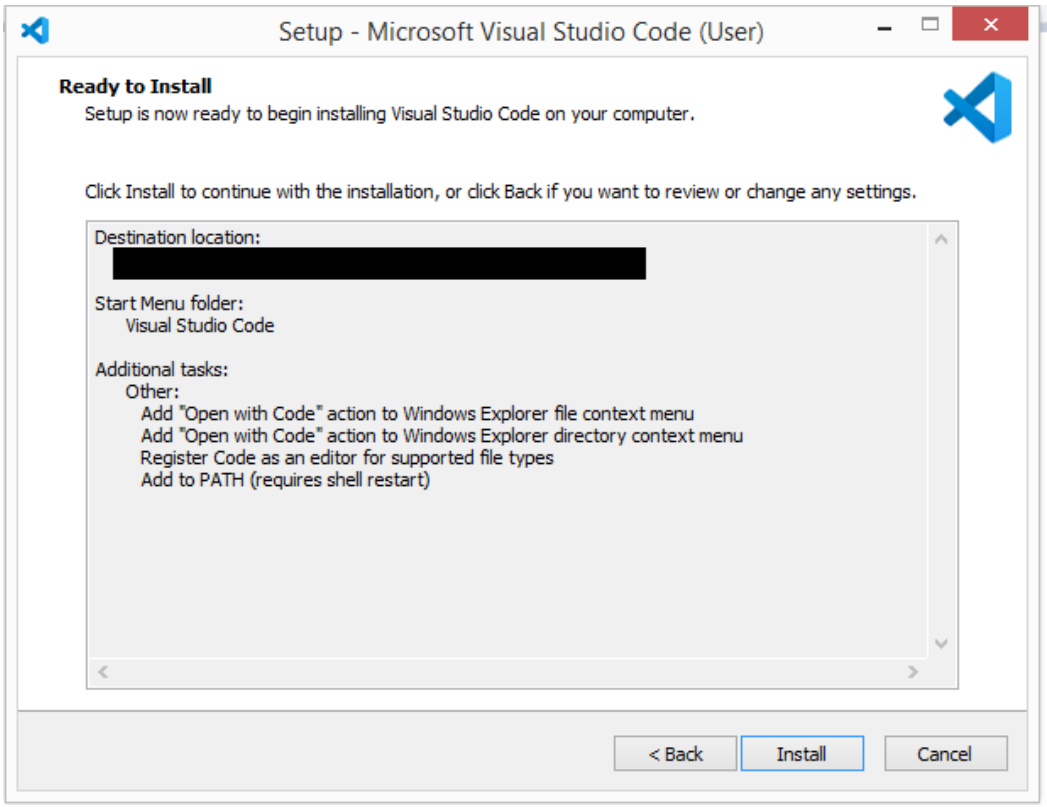
Slika 2. Odabir lokacije za instalaciju



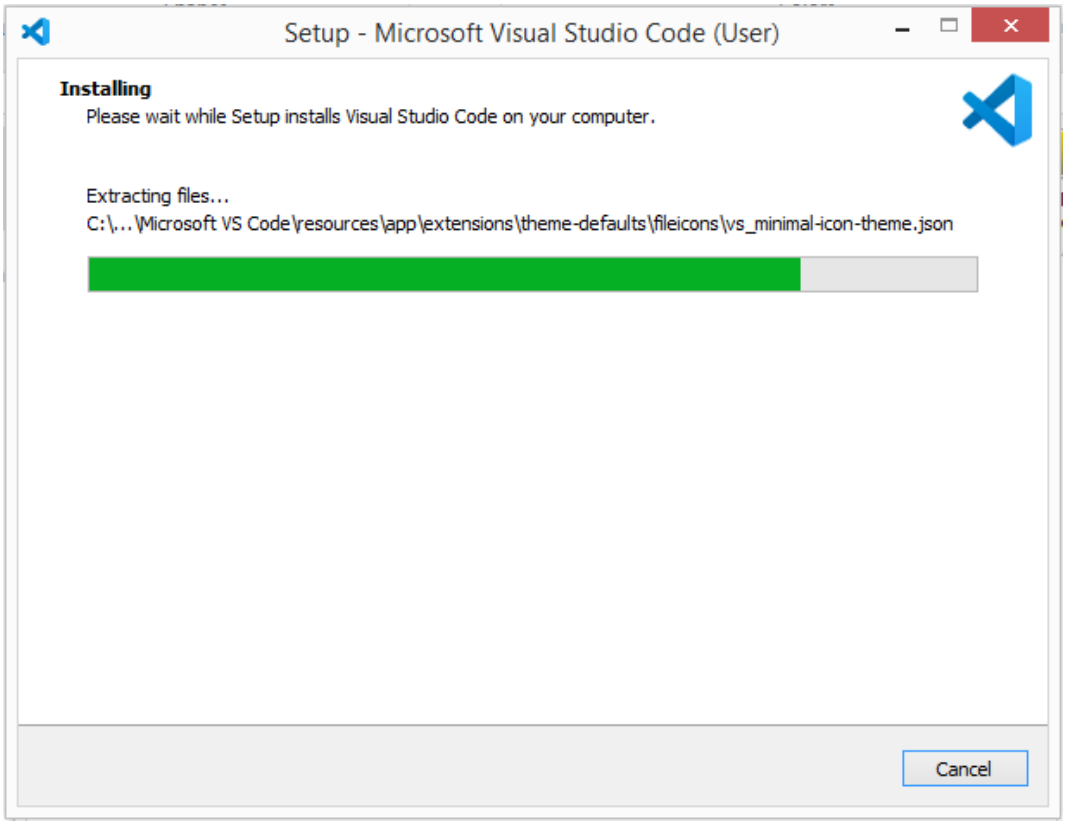
Slika 3. Podešavanje prečice



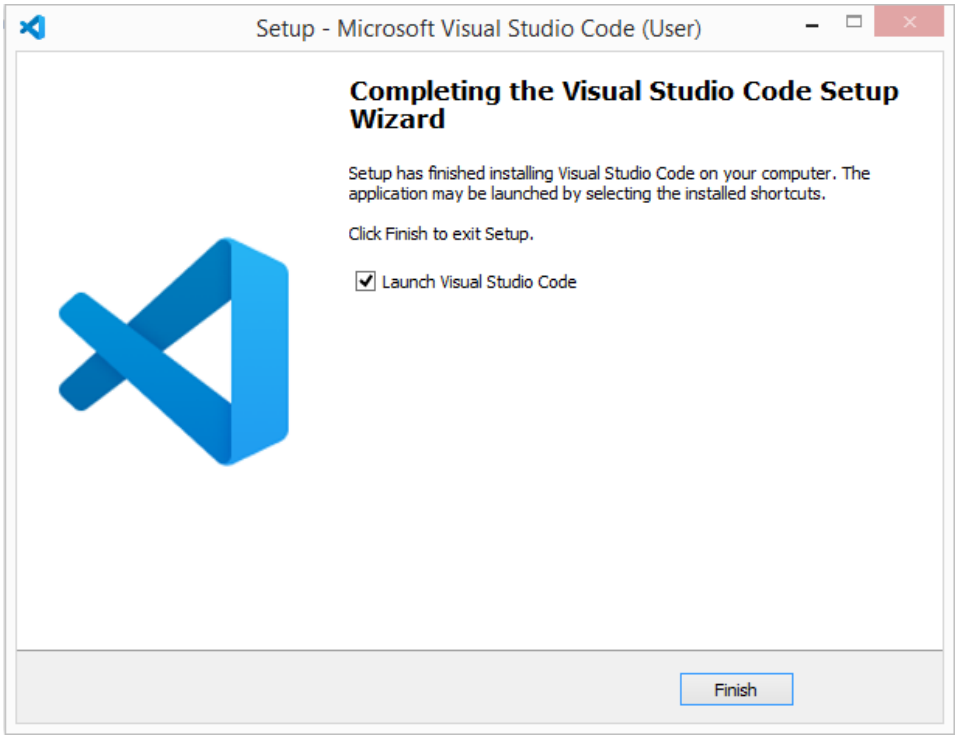
Slika 4. Podešavanje dodatnih opcija



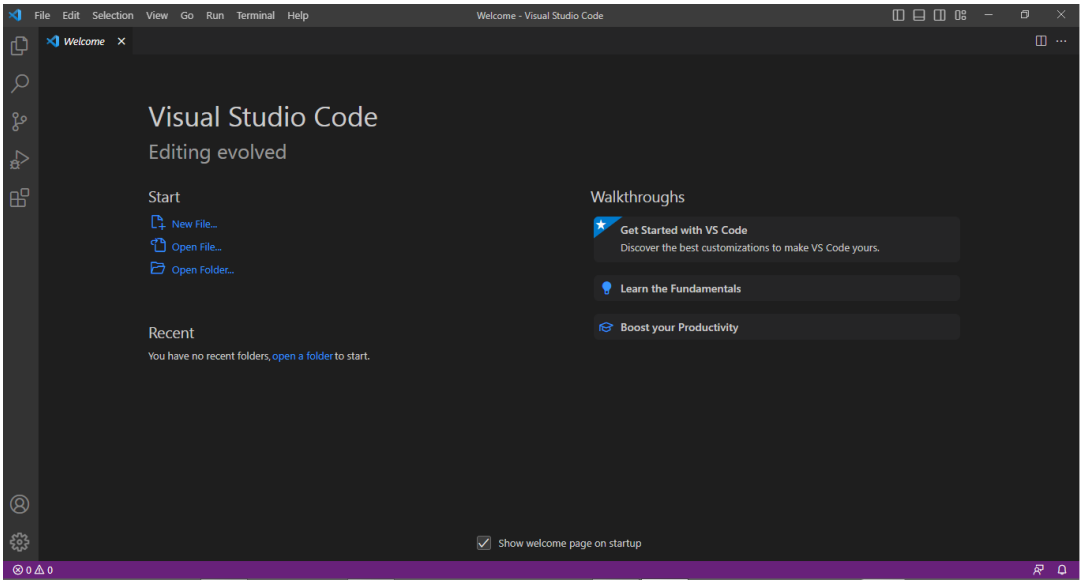
Slika 5. Pregled podešavanje pred instalaciju



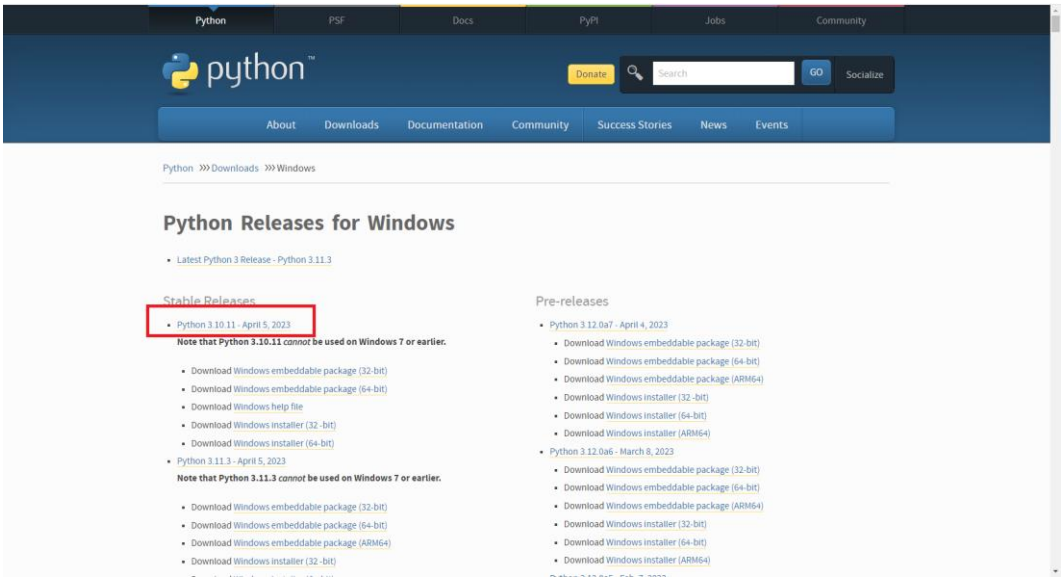
Slika 6. Instalacija



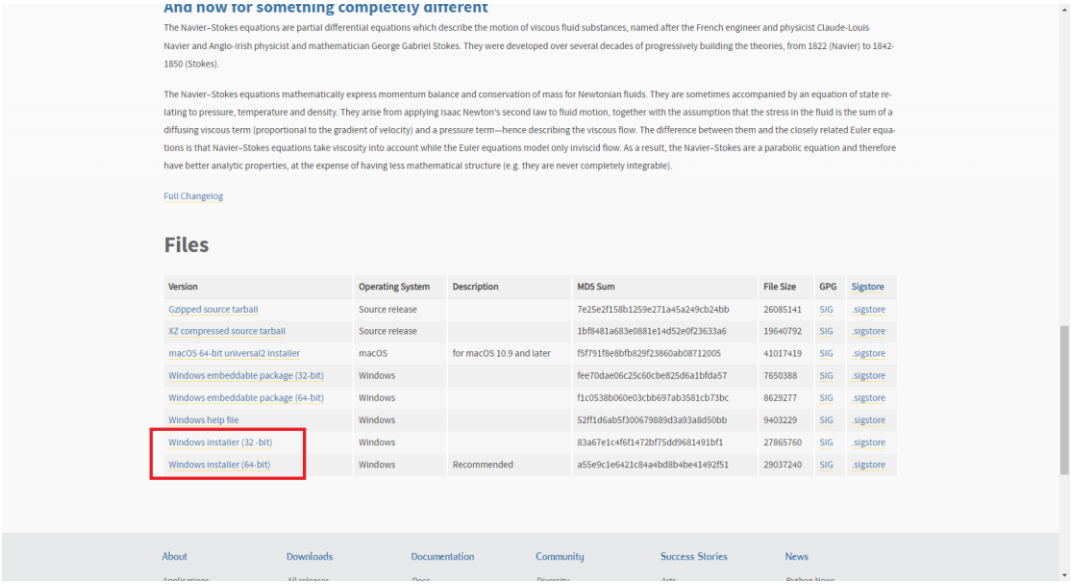
Slika 7. Gotova instalacija



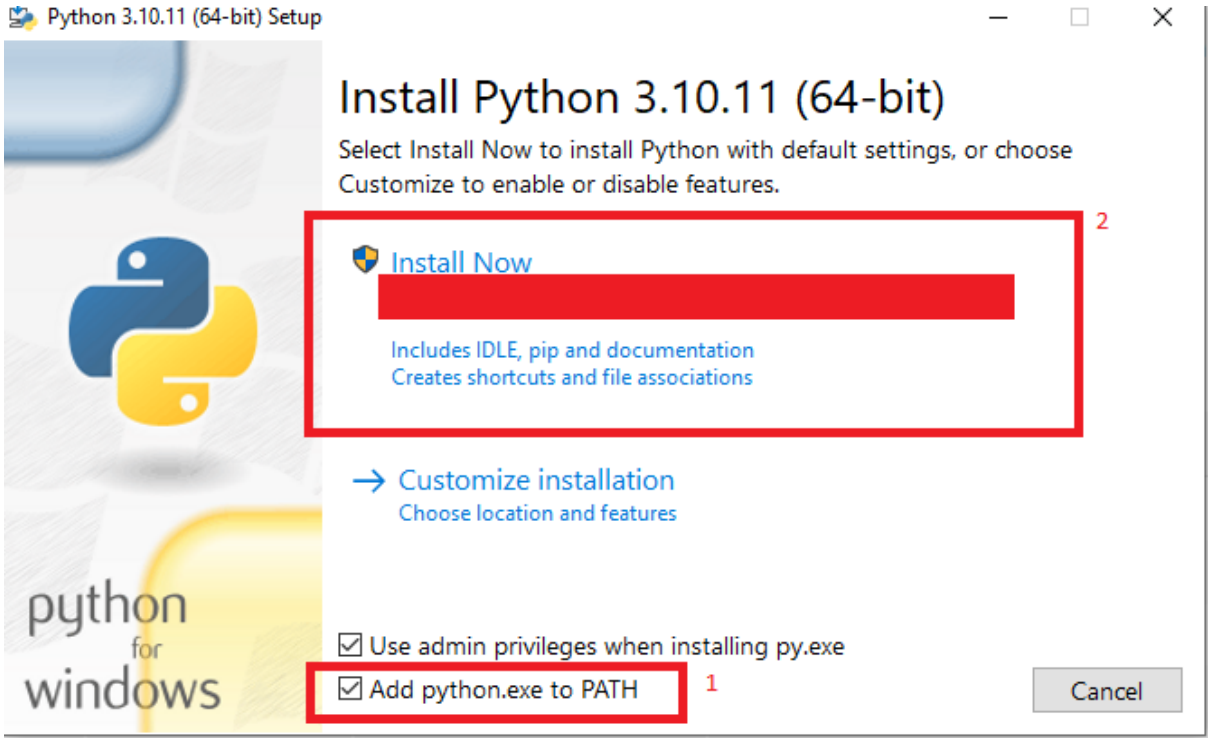
Slika 8. Početni ekran Visual Studio Code alata



Slika 9. Skidanje Python-a

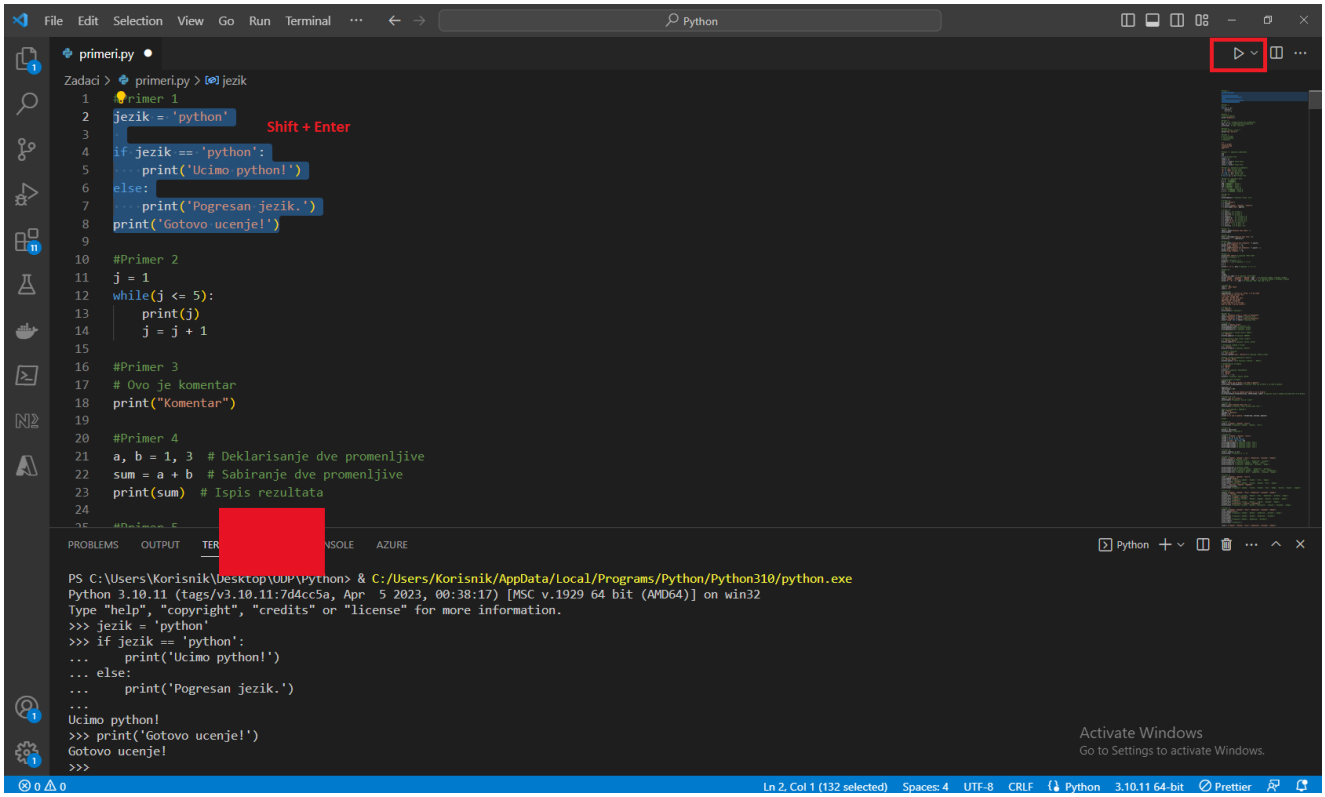


Slika 10. Odabir odgovarajuće verzije



Slika 11. Instalacija Python-a

Slika 12. Instaliranje ekstenzije za Python u VS Code-u



Slika 13. Selekcija koda, izvršavanje i prikaz rezultata u Terminalu. Označeno dugme za izvršavanje celog dokumenta (skripte).