

Programiranje u *Pythonu*

D460



Obrazovni
programi
Srca

priručnik za polaznike



srce

Sveučilište u Zagrebu
Sveučilišni računski centar

Ovu inačicu priručnika izradio je autorski tim Srca u sastavu:

Autor: Marko Hruška

Recenzentica: Vesna Zbodulja

Urednik: Dominik Kendel

Lektorica: Marina Negotić Kovač



Obrazovni
programi
Srca

Sveučilište u Zagrebu

Sveučilišni računski centar

Josipa Marohnića 5, 10000 Zagreb

edu@srce.hr

Verzija priručnika D460-20260930



Ovo djelo dano je na korištenje pod licencom Creative Commons Imenovanje-Dijeli pod istim uvjetima 4.0 međunarodna (CC BY-SA 4.0). Licenca je dostupna na stranici:

<https://creativecommons.org/licenses/by-sa/4.0/deed.hr>.

Sadržaj

Uvod	5
1. Visual Studio Code + Python dodatak	7
1.1. Instalacija razvojnog okruženja Visual Studio Code	7
1.2. Instalacija Python dodatka	7
1.3. Odabir Python interpretera	8
1.4. Pokretanje Python programškoga kôda	9
1.5. Otklanjanje grešaka	10
2. Specifičnosti programškoga jezika Python	11
2.1. Izrezivanje (engl. <i>slicing</i>)	11
2.2. Pakiranje i raspakiravanje vrijednosti	13
2.3. Ulančani operatori usporedbe	15
2.4. Konstrukcija: <code>for/else</code>	15
2.5. Ternarni operator	16
2.6. Naredba <code>pass</code>	18
2.7. Povratna vrijednost funkcije – vraćanje više vrijednosti	19
2.8. Vježba: Specifičnosti programškoga jezika Python	19
3. Napredno korištenje funkcijama	21
3.1. Rekurzivne funkcije	21
3.2. Vježba: Rekurzivne funkcije	24
3.3. Bezimene funkcije	25
3.4. Vježba: Bezimene funkcije	27
3.5. Ugniježdene funkcije	28
3.6. Vježba: Ugniježdene funkcije	30
3.7. Generatorske funkcije	30
3.8. Vježba: Generatorske funkcije	32
3.9. Pitanja za ponavljanje: Napredno korištenje funkcijama	33
4. Iznimke	35
4.1. Primjeri iznimaka	37
4.2. Obrada iznimaka	41
4.3. Vježba: Iznimke I	43
4.4. Podizanje iznimaka	44
4.5. Istovremena obrada više tipova iznimaka	45
4.6. Prosljeđivanje iznimaka	46
4.7. Korištenje vlastitih tipova iznimaka	47
4.8. Završne akcije: <code>else</code> i <code>finally</code>	48
4.9. Vježba: Iznimke II	52
4.10. Pitanja za ponavljanje: Iznimke	52
5. Algoritmi sortiranja	53
5.1. Sortiranje biranjem (engl. <i>selection sort</i>)	54
5.2. Vježba: Sortiranje biranjem	56

5.3.	Sortiranje zamjenom susjednih elemenata (engl. <i>bubble sort</i>)	56
5.4.	Poboljšani algoritam zamjene susjednih elemenata	59
5.5.	Usporedba funkcije <code>sorted()</code> i metode <code>sort()</code>	60
5.6.	Vježba: Sortiranje zamjenom susjednih elemenata	62
5.7.	Pitanja za ponavljanje: Algoritmi sortiranja.....	62
6.	Moduli i paketi	63
6.1.	Instalacija paketa trećih strana	63
6.2.	Kreiranje i uključivanje vlastitih modula	69
6.3.	Vježba: Moduli i paketi.....	73
7.	Rad s CSV datotekama	75
7.1.	Općenito o CSV datotekama	75
7.2.	Čitanje CSV datoteke	76
7.3.	Pisanje u CSV datoteku.....	78
7.4.	Kreiranje novoga dijalekta	79
7.5.	Vježba: Rad s CSV datotekama.....	82
7.6.	Pitanja za ponavljanje: Rad s CSV datotekama.....	82
8.	Objektno orijentirano programiranje	83
8.1.	Razredi i objekti	84
8.2.	Vrste metoda	86
8.3.	Specijalne metode (<code>__init__</code> , <code>__str__</code>).....	86
8.4.	Vježba: Objektno orijentirano programiranje I	89
8.5.	Metode objekta	89
8.6.	Statičke metode	92
8.7.	Vlasništvo varijabli	94
8.8.	Vježba: Objektno orijentirano programiranje II	96
8.9.	Vidljivost varijable – <code>public</code> , <code>protected</code> , <code>private</code>	97
8.10.	Svojstva	100
8.11.	Vježba: Objektno orijentirano programiranje III	103
8.12.	Nasljeđivanje (engl. <i>inheritance</i>)	104
8.13.	Preopterećenje (engl. <i>overloading</i>)	108
8.14.	Nadjačavanje (engl. <i>overriding</i>).....	109
8.15.	Vježba: Objektno orijentirano programiranje IV.....	111
8.16.	Apstraktni razredi (engl. <i>abstract base classes</i>).....	112
8.17.	Ulančavanje metoda	115
8.18.	Vježba: Objektno orijentirano programiranje V.....	118
8.19.	Pitanja za ponavljanje: Objektno orijentirano programiranje	118
Dodatak:	Rješenja vježbi	121
Literatura.....		154

Uvod

Svrha ovoga priručnika jest upoznati čitatelja s naprednijim konceptima i tehnikama konstrukcije programskoga kôda u programskome jeziku Python.

Sadržaj ovoga priručnika nadovezuje se na priručnik *Osnove programiranja (Python)*. Preduvjet za razumijevanje gradiva ovoga priručnika jest poznavanje rada na računalu i poznavanje osnova programiranja u programskome jeziku Python. Na ovome tečaju polaznici će naučiti naprednije koncepte programiranja u Pythonu, uključujući i objektno orijentirano programiranje. Usvojeni koncepti primjenjivi su i na većinu ostalih programskih jezika, čime se polaznicima omogućuje da nakon završenoga tečaja lakše svladavaju i druge programske jezike. Naučeni koncepti olakšavaju daljnje nadograđivanje znanja nakon tečaja uz pomoć pretraživanja interneta i korištenja stručnom literaturom. Nakon tečaja polaznici će posjedovati kompetencije za rješavanje složenijih programskih zadataka u programskome jeziku Python.

Priručnik se sastoji od osam poglavlja koja se obrađuju tijekom četiri dana, u trajanju od četiri školska sata dnevno. Uz teorijski dio gradiva, svako poglavlje sadrži vježbe koje će polaznici prolaziti s predavačem. Savjeti i zanimljivosti istaknuti su u okvirima sa strane.

Python podržava nekoliko različitih pristupa u pisanju programskoga kôda, a to su imperativno (uključujući strukturno), funkcijsko i objektno orijentirano programiranje [18].

Python je interpreterski programski jezik koji se najčešće izvršava putem interpretera, pri čemu se izvorni kôd prethodno prevodi u međukôd (engl. *bytecode*). Kako je Python interpreterski programski jezik, programi pisani u njemu izvršavaju se sporije u odnosu na programe pisane u programskim jezicima poput C i C++.

U ovome tečaju obrađuje se strukturno programiranje i objektno orijentirano programiranje. U strukturnom programiranju glavni je naglasak na kontroli toka programa uporabom struktura poput funkcija, grananja (odluka) i petlji. Ovakav način programiranja omogućuje programerima da se program podijeli na manje logičke dijelove, odnosno u funkcije te se tako povećava modularnost programskoga kôda koji je moguće višekratno upotrebljavati. Ovakav pristup pogodan je za manje i srednje složene programe, no u velikim programima može otežati organizaciju i održavanje kôda. Objektno orijentirano programiranje (kraće: OOP) jedan je od najpopularnijih pristupa razvoju programskoga kôda današnjice. Glavni je cilj ovoga pristupa razvoja programskoga kôda razvijanje razreda čije su funkcionalnosti implementirane metodama. Ovaj pristup često se primjenjuje u razvoju većih i složenijih programa.

U trenutku pisanja ovoga priručnika aktualna verzija programskoga jezika Python jest Python 3, stoga se i ovaj tečaj temelji na toj verziji. Također postoji Python 2, no ta verzija više nije podržana i ne razvija se

**Trajanje
poglavlja:**
10 min

Python – verzije

Verzija Pythona označava se formatom A.B.C, pri čemu svaki segment definira razinu i opseg promjena. Vodeći broj označava glavne nadogradnje koje mogu narušiti kompatibilnost, srednji donosi nove funkcionalnosti, dok posljednji služi za ispravke programskih grešaka i sigurnosne zakrpe.

Službena dokumentacija

Poveznica na službenu
Python 3 dokumentaciju:
<https://docs.python.org/3/>

aktivno. Sintaksa između Pythona 2 i Pythona 3 nije kompatibilna. Za razvoj i održavanje Pythona zadužena je neprofitna organizacija Python Software Foundation.

U priručniku su važni pojmovi pisani **podebljano**. Ključne riječi, imena varijabli, funkcija, metoda i ostale programske konstrukcije pisane su drugačijim fontom od uobičajenog, na primjer: `print("Hello World!")`. Nazivi na engleskome jeziku pisani su *kurzivom* i u zagradi, na primjer: "varijabla (engl. *variable*)".

Programski kôd pisan je na sljedeći način:

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

Izlaz:

```
Hello World!  
Hello World!
```

U prvom dijelu bit će
napisan programski kôd.

U drugom dijelu bit će prikazan
dobiveni izlaz programskoga kôda.

1. Visual Studio Code + Python dodatak

U ovome poglavlju prikazuje se korištenje razvojnim okruženjem Visual Studio Code (VS Code) za pisanje i pokretanje programa u programskome jeziku Python. Visual Studio Code predstavlja moderno i fleksibilno razvojno okruženje (engl. *Integrated Development Environment* – IDE) koje omogućuje učinkovito pisanje programskoga kôda, njegovo pokretanje te analizu i otklanjanje grešaka.

S obzirom na svoju jednostavnost uporabe i mogućnost proširivanja dodacima, Visual Studio Code jedan je od najpopularnijih alata za razvoj aplikacija te je pogodan kako za početnike, tako i za napredne korisnike. U nastavku će biti prikazan postupak instalacije i konfiguracije okruženja za pisanje Python programskoga kôda.

Za razliku od tradicionalnih IDE-ova, kao što je, na primjer, PyCharm, Visual Studio Code u osnovi je uređivač izvornoga kôda koji se proširuje instalacijom dodataka (engl. *extensions*). Time korisnik okruženje u kojemu radi može prilagoditi prema vlastitim željama i potrebama.

Trajanje
poglavlja:
10 min

1.1. Instalacija razvojnog okruženja Visual Studio Code

Visual Studio Code moguće je preuzeti sa službene mrežne stranice: <https://code.visualstudio.com>. Visual Studio Code dostupan je za sve vodeće operacijske sustave, kao što su Windows, Debian, Ubuntu, Red Hat, Fedora i macOS.

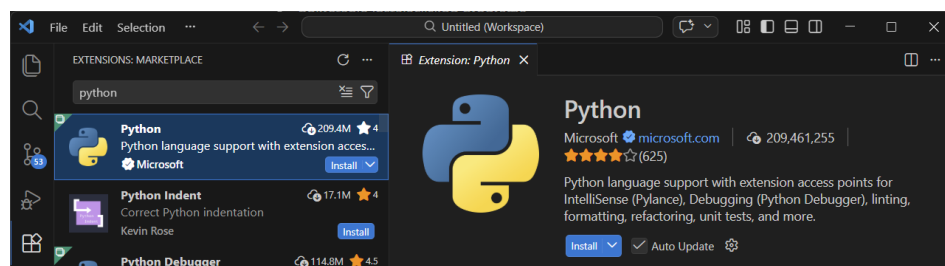
Postupak instalacije je jednostavan. Nakon preuzimanja instalacijske datoteke potrebno je pokrenuti instalacijski program te pratiti upute na ekranu. Nakon uspješne instalacije preporučuje se pokrenuti Visual Studio Code kako bi se izvršila početna konfiguracija.

1.2. Instalacija Python dodatka

Kako bi Visual Studio Code mogao pokretati programski kôd napisan u programskome jeziku Python, potrebno je instalirati Python dodatak.

Koraci za instalaciju Python dodatka:

1. Otvoriti Visual Studio Code.
2. Kliknuti na karticu *Extensions*.
3. U tražilicu upisati "Python".
4. Odabrati proširenje koje izdaje Microsoft.
5. Kliknuti na gumb *Install*.



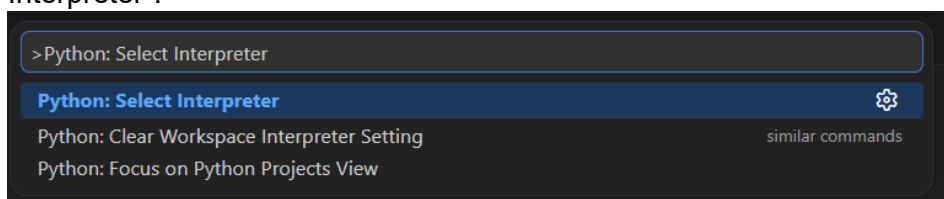
Python dodatak za Visual Studio Code omogućuje podršku za razvoj aplikacija u programskome jeziku Python. Instalacijom dodatka korisniku su dostupne brojne napredne funkcionalnosti koje olakšavaju pisanje, razumijevanje i održavanje programskoga kôda, smanjuju mogućnost grešaka i ubrzavaju razvojni proces. Jedna od osnovnih funkcionalnosti jest IntelliSense (Pylance), koji omogućuje automatsko dovršavanje kôda, prikaz informacija o funkcijama, varijablama i tipovima podataka te olakšava snalaženje u kôdu. Dodatak također uključuje alate za debugiranje (detaljna analiza izvođenja programa), lintanje (analiza kôda i upozoravanje na potencijalne greške), formatiranje, refaktoriranje, testiranje te niz ostalih mogućnosti koje unapređuju radno okruženje.

1.3. Odabir Python interpretera

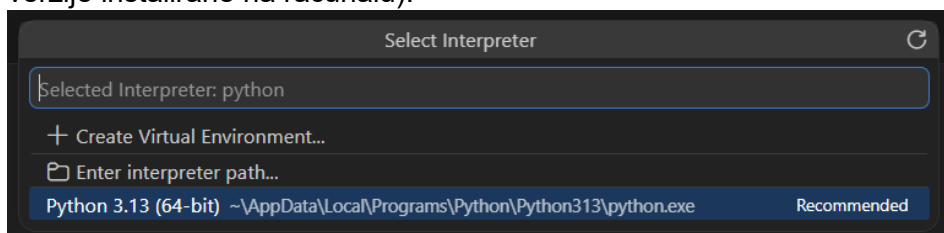
Nakon instalacije Python dodatka potrebno je odabrati Python interpreter koji će se upotrebljavati za izvršavanje programskoga kôda pisanoga u Pythonu. Interpreter predstavlja instaliranu verziju Pythona na računalo, stoga je prije njegova odabira potrebno osigurati da je Python prethodno instaliran na računalo.

Koraci za odabir interpretera:

1. Otvoriti Visual Studio Code.
2. Otvoriti *Command Palette*.
 - a. putem izbornika: *View > Command Palette*
 - b. uporabom tipkovničkoga prečaca: **Ctrl + Shift + P**
3. U otvoreni prozor potrebno je upisati: "Python: Select Interpreter".



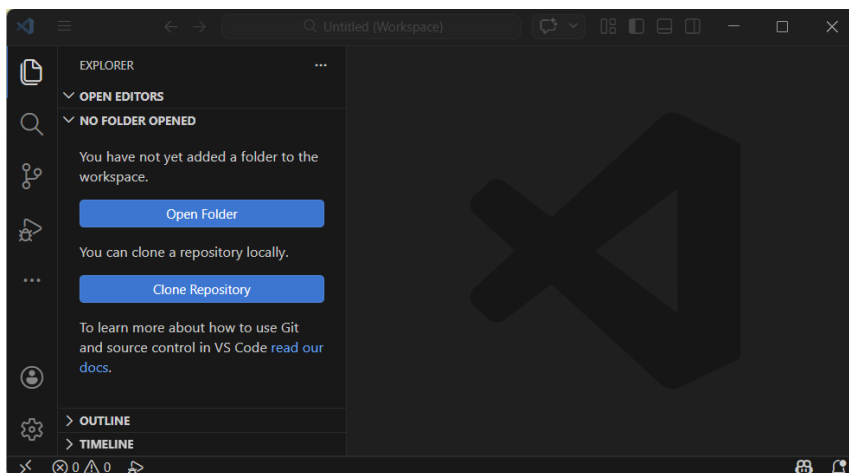
4. Odabrati željenu verziju Pythona s popisa (ovdje će se prikazati verzije instalirane na računalo).



1.4. Pokretanje Python programskoga kôda

Visual Studio Code omogućuje rad s pojedinačnim Python datotekama, ali i s projektima. Za efikasniji razvoj i lakše održavanje programskog kôda preporuka je organizirati projekt u zasebnu mapu i tu mapu potom otvoriti unutar Visual Studio Codea. Dodavanje novoga projekta (mape) u Visual Studio Code može se napraviti na više načina:

- klikom na gumb *Open Folder* (pogledati sliku niže)
- putem izbornika *File > Add Folder to Workspace*.

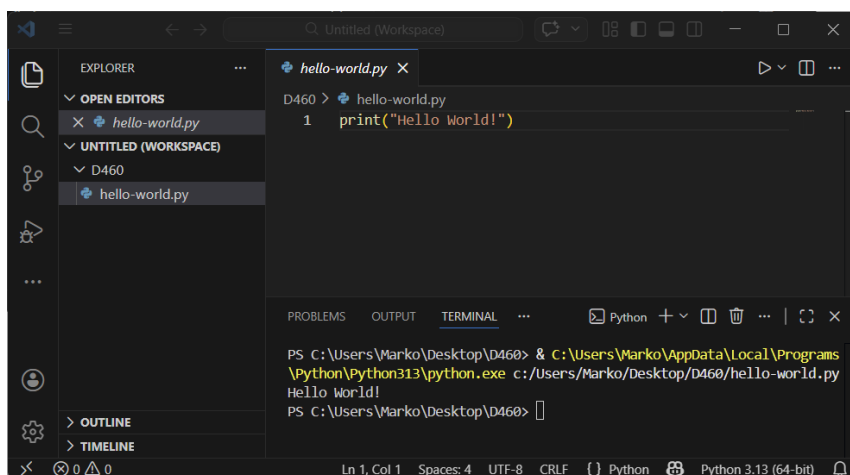


Nakon dodavanja novoga projekta (mape) unutar razvojnoga okruženja Visual Studio Code moguće je otvarati, uređivati, kreirati i brisati datoteke. Visual Studio Code sadrži ugrađeni terminal koji omogućuje izvršavanje programskoga kôda bez potrebe za napuštanjem razvojnoga okruženja. Terminal se može otvoriti putem izbornika *View > Terminal*.

Nakon što je programski kôd napisan i spremljen, program se može pokrenuti na više načina:

- putem izbornika *Run > Run Without Debugging*
- klikom na gumb *Run* u gornjem desnom kutu
- uporabom tipkovničkoga prečaca **Ctrl + F5**.

Nakon pokretanja programskoga kôda rezultat se ispisuje u ugrađenome terminalu unutar Visual Studio Codea.

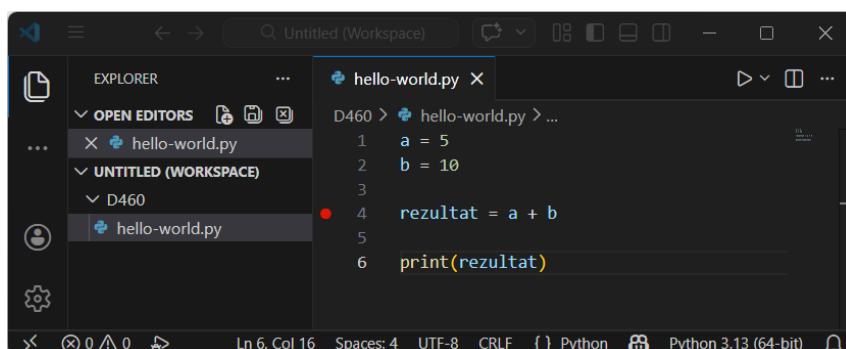


1.5. Otklanjanje grešaka

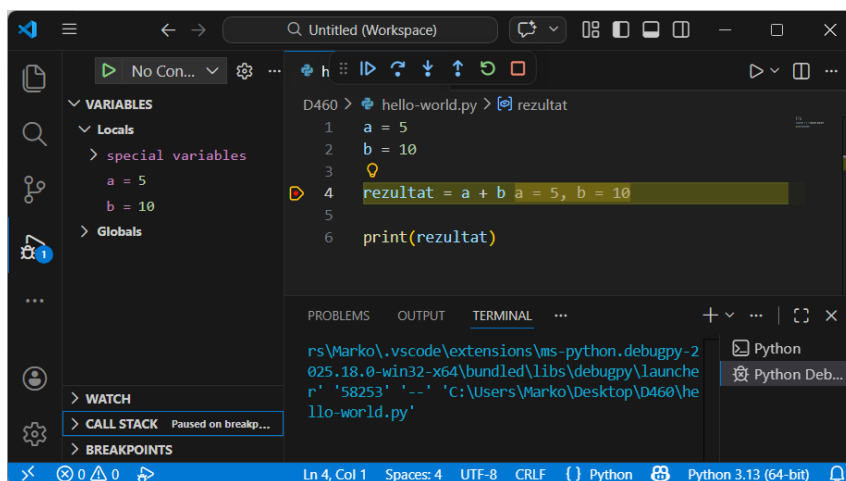
Visual Studio Code pruža mogućnosti otklanjanja grešaka (engl. *debugging*) programa Python. Ovaj proces omogućuje programerima da prate izvršavanje programskoga kôda, kontrolirano zaustavljaju izvođenje programa, promatraju vrijednosti varijabli u određenome trenutku i prate tijek izvršavanja, što znatno olakšava pronalaženje grešaka. Otklanjanje grešaka pokreće se klikom na karticu *Run and Debug* u lijevome bočnom izborniku ili pritiskom na tipku F5.

Osnovne funkcionalnosti otklanjanja grešaka u Visual Studio Codeu:

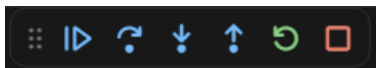
- Postavljanje točaka prekida (engl. *breakpoints*) omogućuje zaustavljanje i pregled stanja programa na određenoj liniji kôda. Točke prekida postavljaju se klikom lijevom tipkom miša uz broj linije (crvena točkica na slici u nastavku).



- Praćenje trenutanih vrijednosti varijabli tijekom izvođenja programa.



- Izvršavanje kôda korak po korak uporabom opcija za kontrolu: *Continue* nastavlja izvođenje programa do sljedeće točke prekida, *Step Over* izvršava trenutnu liniju bez ulaska u funkcije, *Step Into* ulazi unutar funkcija, a *Step Out* izlazi iz njih i nastavlja u roditeljskoj funkciji. Opcija *Restart* pokreće program iz početka, dok *Stop* odmah zaustavlja njegovo izvršavanje.



2. Specifičnosti programskoga jezika Python

Po završetku ovoga poglavlja polaznik će moći:

- *koristiti se nekim od mnogobrojnih specifičnosti programskoga jezika Python koje olakšavaju razvoj programskoga kôda.*

Trajanje poglavlja:
60 min

Programski jezik Python u sebi sadrži neke specifičnosti koje nisu implementirane u svim programskim jezicima. Takve specifičnosti programerima omogućuju brži razvoj programskoga kôda. U nastavku slijedi opis s primjerima nekoliko takvih specifičnosti.

2.1. Izrezivanje (engl. *slicing*)

Izrezivanje (engl. *slicing*) predstavlja mehanizam kojim se iz pobrojivih struktura podataka (na primjer, niz znakova, lista, n-torka) izdvajaju podskupovi elemenata.

Opći oblik izraza za izrezivanje:

```
sekvenca[start:stop:step]
```

Pojašnjenje parametara:

- `start` – indeks od kojega izrezivanje započinje (uključiv)
- `stop` – indeks na kojemu izrezivanje završava (isključiv)
- `step` – korak između elemenata.

Ako se neki od parametara izostavi:

- `start` – zadani početak jest 0
- `stop` – zadani kraj jest duljina pobrojivoga objekta
- `step` – zadani korak jest 1.

Detaljnije pojašnjenje:

- Dohvaćanje elemenata od indeksa `start` do indeksa `stop-1`:
`sekvenca[start:stop]`
- Dohvaćanje elemenata od indeksa `start` pa do kraja:
`sekvenca[start:]`
- Dohvaćanje elemenata od početka do indeksa `stop-1`:
`sekvenca[:stop]`

Vrijednosti parametara `start`, `stop` moguće je zadati kao negativne, tada se primjenjuju sljedeća pravila:

- Dohvaćanje zadnjega elementa:
`sekvenca[-1]`

Funkcija `reversed()`

Funkcija `reversed()` ugrađena je funkcija koja omogućuje iteriranje kroz elemente pobrojivoga tipa podataka u obrnutome redoslijedu. Za razliku od metode `reverse()`, koja mijenja listu (in-place), funkcija `reversed()` ne modificira izvorni objekt, već vraća iterator nad njegovim elementima u obrnutome poretaku.

Za one koji žele znati više

Pronađite i proučite kako funkcionira klasa:

```
slice(start, stop[, step])
```

- Dohvaćanje zadnjih dvaju elemenata:
`sekvenca[-2:]`
- Dohvaćanje svih elemenata osim zadnjih dvaju:
`sekvenca[:-2]`

Uz parametre `start` i `stop` moguće je zadati i treći parametar imena `step`. Parametar `step` označava korak. U nastavku slijedi nekoliko primjera.

- Dohvaćanje svakoga drugog elementa:
`sekvenca[::2]`
- Dohvaćanje svakoga drugog elementa, između indeksa 1 (uključivo) i 5 (isključivo):
`sekvenca[1:5:2]`
- Dohvaćanje svih elemenata u obrnutome poretku:
`sekvenca[::-1]`
- Dohvaćanje prvih dvaju elemenata u obrnutome poretku:
`sekvenca[1::-1]`
- Dohvaćanje zadnjih triju elemenata u obrnutome poretku:
`sekvenca[:-3:-1]`
- Dohvaćanje svih elemenata osim zadnjih dvaju u obrnutome poretku:
`sekvenca[-3::-1]`

Primjer 1: Primjeri izrezivanja liste.

```
>>> lista = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> lista[2:5]
[2, 3, 4]
>>> lista[2]
2
>>> lista[:5]
[0, 1, 2, 3, 4]
>>> lista[:]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> lista[-1]
9
>>> lista[-2:]
[8, 9]
>>> lista[:-2]
[0, 1, 2, 3, 4, 5, 6, 7]
>>> lista[::2]
[0, 2, 4, 6, 8]
>>> lista[1:5:2]
[1, 3]
>>> lista[::-2]
[9, 7, 5, 3, 1]
```

```
>>> lista[::-1]
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
>>> lista[1::-1]
[1, 0]
>>> lista[:-3:-1]
[9, 8]
>>> lista[-3::-1]
[7, 6, 5, 4, 3, 2, 1, 0]
```

Primjer 2: Dohvaćanje svih elemenata liste u obrnutome poretku.

```
lista = [1, 2, 5, 4, 8, 3]
obrnutaLista = lista[::-1]

print("Originalna lista:", lista)
print("Obrnuta lista:", obrnutaLista)
Izlaz:
Originalna lista: [1, 2, 5, 4, 8, 3]
Obrnuta lista: [3, 8, 4, 5, 2, 1]
```

Izrezivanje i nizovi znakova

Do sada obrađena sintaksa izrezivanja može se primjeniti i nad nizovima znakova.

Primjer 3: Dohvaćanje svih elemenata nizova znakova u obrnutome poretku.

```
nizZnakova = "abcdefghijkl"
obrnutiNizZnakova = nizZnakova[::-1]

print("Originalni niz znakova:", nizZnakova)
print("Obrnuti niz znakova:", obrnutiNizZnakova)
Izlaz:
Originalni niz znakova: abcdefghijkl
Obrnuti niz znakova: lkjihgfedcba
```

Funkcija `reversed()`

Funkcijom `reversed()` također se može koristiti za dohvat elemenata nizova znakova u obrnutome poretku. Ova funkcija vraća iterator koji prolazi kroz elemente u obrnutome poretku.

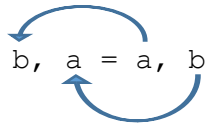
Primjer 4: Dohvaćanje svakoga drugog znaka.

```
nizZnakova = "abcdefghijkl"
noviNiz = nizZnakova[::2]

print("Originalni niz znakova:", nizZnakova)
print("Svaki drugi znak:", noviNiz)
Izlaz:
Originalni niz znakova: abcdefghijkl
Svaki drugi znak: acegik
```

2.2. Pakiranje i raspakiravanje vrijednosti

Python omogućuje jednostavan način zamjene vrijednosti dviju ili više varijabli u jednoj liniji. Sintaksa i grafički prikaz zamjene vrijednosti dviju varijabli:



Primjer 1: Zamjena vrijednosti dviju varijabli u jednoj liniji.

```
a = 10
b = 20

print(a, b)

b, a = a, b

print(a, b)

Izlaz:
    10 20
    20 10
```

Ovakav se način dodjeljivanja vrijednosti varijablama u pozadini koristi pakiranjem i raspakiravanjem n-terca (engl. *tuple packing / unpacking*). U nastavku slijedi detaljnije pojašnjenje.

1. Pakiranje n-terca (engl. *packing a tuple*) – sintaksa koja je prikazana u nastavku omogućuje kreiranje n-terca bez korištenja ustaljenom sintaksom.

```
x = 1, 2
```

Isto kao:

```
x = (1, 2)
```

2. Raspakiravanje n-terca (engl. *unpacking a tuple*) – sintaksa koja omogućuje da se dane vrijednosti zapakirane u n-torka raspakiraju i spremne u varijable.

```
x = 1, 2
```

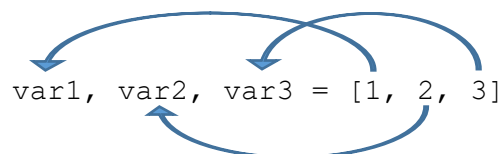
```
a, b = x # a, b = (1, 2)
```

Nakon izvršavanja ovoga kôda imamo: `a = 1, b = 2`.

Vrijednost n-terca `x` raspakirava se u 2 varijable `a, b`. Broj varijabli mora točno odgovarati broju elemenata u n-tercu ili će doći do greške.

Raspakiravanje vrijednosti iz liste

Programski jezik Python omogućuje da se vrijednosti iz liste raspakiraju u pojedinačne varijable, a to se postiže tako da Python iterira kroz listu i redom dodjeljuje vrijednosti varijablama. Sintaksa i grafički prikaz preslikavanja vrijednosti:



Primjer 2: Raspakiravanje triju vrijednosti iz liste u tri zasebne varijable u jednoj liniji.

```
lista = [1, 2, 3]
```

```
a, b, c = lista
```

```
print(a, b, c)
```

Izlaz:

```
1 2 3
```

2.3. Ulančani operatori usporedbe

Ulančavanje operatora usporedbe funkcionalnost je koja u mnogim drugim programskim jezicima nije dostupna. Ako želimo provjeriti je li vrijednost varijable `a` u intervalu `[10, 20]`, u programskome jeziku Python dovoljno je napisati kôd sljedeće sintakse:

```
10 <= a <= 20
```

U većini drugih jezika (npr. C, C++, Java) za istu je provjeru potrebno rabiti ovakav izraz:

```
10 <= a && a <= 20
```

Primjer 1: Ulančani operatori usporedbe.

```
a = 12
```

```
print(10 <= a <= 20)
```

```
a = 22
```

```
print(10 <= a <= 20)
```

Izlaz:

```
True
```

```
False
```

Prednost korištenja ulančanih operatora usporedbe jest pregledniji i razumljiviji kôd. Ista varijabla ili izraz pojavljuje se samo jednom, čime se smanjuje mogućnost greške. Također, zapis nalikuje matematičkomu zapisu, na primjer `10 <= a <= 20`, što kôd čini intuitivnijim i lakšim za čitanje, posebno pri izražavanju višestrukih usporedbi u jednome izrazu.

2.4. Konstrukcija: `for/else`

Petlje su ključni element svakoga programskog jezika. U programskome jeziku Python upotrebljavaju se petlje `for` i `while`. Svaki je programer upoznat s njihovim funkcioniranjem, no konstrukcija `for/else` često ostaje nepoznata.

Petlja `for` može sadržavati opcionalni blok `else`. Opcionalni blok `else` izvršava se samo ako petlja bez prekida odradi svoj posao, odnosno kada nije prekinuta pozivom naredbe `break`.

Kao primjer, upotrijebit ćemo petlju `for` koja prolazi kroz strukturu podataka i traži određeni element. Tijekom takvoga pretraživanja mogu se dogoditi dva slučaja:

- **Primjer 1:** Element postoji i petlja se prekida.

Ako petlja pronade traženi element te prekine njezino izvršavanje pozivom naredbe `break`, blok `else` se ne izvršava. U nastavku je prikaz programskoga kôda koji prikazuje ovaj slučaj.

```
lista = [1, 2, 3, 4, 5, 6]

for element in lista:
    if element == 5:
        break
else:
    print("Nije pronađeno!")

print("Kraj!")
```

Izlaz:

Kraj!

- **Primjer 2:** Element ne postoji u strukturi podataka.

Traženi element ne postoji i petlja `for` završava prolazak kroz sve elemente bez poziva naredbe `break`. U tome slučaju izvršava se programski kôd koji se nalazi u `else` bloku.

```
lista = [1, 2, 3, 4, 5, 6]

for element in lista:
    if element == 0:
        break
else:
    print("Nije pronađeno!")

print("Kraj!")
```

Izlaz:

Nije pronađeno!

Kraj!

Konstrukcija `for/else` posebno je korisna kada želimo razlikovati situaciju kada je petlja prekinuta pozivom naredbe `break` od situacije kada nije. Ovaj pristup omogućuje rukovanje obradom rezultata bez potrebe za dodatnim pomoćnim varijablama.

2.5. Ternarni operator

Ternarni operator omogućuje da se funkcionalnost `if-else` naredbe napiše u jednome retku.

Opći oblik ternarnoga operatora izgleda ovako:

```
[izraz_za_True] if [uvjet] else [izraz_za_False]
```

- uvjet – logički izraz čiji je rezultat `True` ili `False`, na temelju toga rezultata odlučuje koji će se izraz izvršiti: `izraz_za_True` ili `izraz_za_False`
- `izraz_za_True` – izvršava se ako je rezultat logičkoga izraza `True`
- `izraz_za_False` – izvršava se ako je rezultat logičkoga izraza `False`

Primjer 1: Korištenje ternarnim operatorom.

```
broj = 7

parnost = "Paran!" if broj % 2 == 0 else "Neparan!"

print(parnost)

Izlaz:
    Neparan!
```

Ternarni operator obično se upotrebljava za jednostavne uvjete i kratke izraze, za složeniju logiku bolje je rabiti klasičnu naredbu `if-else` radi bolje čitljivosti.

Poziv funkcije pomoću ternarnoga operatora

Ternarni operator može se rabiti i za odabir funkcije koja će se pozvati, ovisno o rezultatu logičkoga izraza.

Opći oblik poziva funkcije pomoću ternarnoga operatora:

```
([fun_za_True] if [uvjet] else [fun_za_False])([arg])
```

- `fun_za_True` – funkcija koja će se pozvati ako je rezultat uvjeta `True`
- `fun_za_False` – funkcija koja će se pozvati ako je rezultat uvjeta `False`

Primjer 2: Funkcije koje se pozivaju na ovaj način najčešće imaju isti broj i tip argumenata.

```
def pozitivno(x):
    print(f"Broj {x} je pozitivan!")

def negativno(x):
    print(f"Broj {x} je negativan ili nula!")

broj = -3

(pozitivno if broj > 0 else negativno)(broj)

Izlaz:
    Broj -3 je negativan ili nula!
```

Za one koji žele znati više

Ovaj način pozivanja funkcija funkcionira jer ternarni operator vraća funkcijski objekt – svaki razred može biti funkcija ako implementira metodu `__call__`.

2.6. Naredba `pass`

Ovom se naredbom koristi kada je potreban blok kôda (npr. tijelo unutar konstrukcija kao što su `if`, `for`, `while` itd.), ali iz nekoga razloga ne želi se unutar bloka napisati nijedna konkretna linija programskoga kôda. U takvim slučajevima nije dopušteno ostaviti prazan blok, već se potrebno koristiti naredbom `pass`.

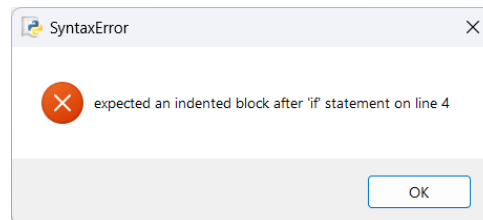
Primjer 1: U sljedećemu primjeru prilikom pokretanja programa dolazi do sintaksne greške jer tijelo uvjeta `if` ne sadrži nijednu naredbu, što nije dopušteno prema sintaksnim pravilima Pythona.

```
rijec = "Srxce"

for e in rijec:
    if e == 'x':

    else:
        print(e)
```

Izlaz:



Primjer 2: Ako želimo da se za određeni uvjet ne izvrši ništa, upotrebljava se naredba `pass`.

```
rijec = "SRxCE"

for e in rijec:
    if e == 'x':
        pass
    else:
        print(e)
```

Izlaz:

```
S
R
C
E
```

Primjer 3: Također, važno je napomenuti da naredba `pass` ne prekida izvršavanje programskoga kôda unutar bloka, već samo označava praznu naredbu. Sve naredbe koje se nalaze ispod nje u istome bloku i dalje će se izvršiti.

```
rijec = "SRxCE"

for e in rijec:
    if e == 'x':
        pass
        print("Ključna riječ pass")
```

```
else:
    print(e)
```

```
Izlaz:
S
R
Ključna riječ pass
C
E
```

2.7. Povratna vrijednost funkcije – vraćanje više vrijednosti

Za razliku od mnogih programskih jezika kao što su C, C++ i Java, Python omogućuje da funkcija na jednostavan način vrati više vrijednosti odjednom. Iako se čini da funkcija vraća više vrijednosti, ona zapravo vraća jedan n-torka (engl. *tuple*) koji sadrži više vrijednosti.

Primjer 1: U nastavku je prikazan primjer u kojemu funkcija `fun()` vraća u pozivajući dio programa 5 vrijednosti. Te vrijednosti vraćaju se zapakirane u n-torku.

```
def fun():
    return 1, 2, 3, 4, 5
```

```
rezultat = fun()
print(rezultat)
print(type(rezultat))
```

```
Izlaz:
(1, 2, 3, 4, 5)
<class 'tuple'>
```

Primjer 2: Dobivena n-torka može se raspakirati u više varijabli. Važno je napomenuti da broj varijabli mora odgovarati broju vrijednosti koje funkcija vraća (broju elemenata unutar n-torke).

```
def fun():
    return 1, 2, 3, 4, 5
```

```
a, b, c, d, e = fun()
print(a, b, c, d, e)
```

```
Izlaz:
1 2 3 4 5
```

2.8. Vježba: Specifičnosti programskoga jezika Python

1. Napišite program unutar kojega ćete u varijablu inicijalizirati listu proizvoljnoga sadržaja (lista od barem 3 vrijednosti). Elemente liste spremite u pojedinačne varijable u jednoj liniji. Kad elemente liste budete imali spremljene u zasebnim varijablama, ispišite vrijednosti tih varijabli.

2. U dvije varijable inicijalizirajte dva različita proizvoljna broja. Nakon inicijalizacije ispišite vrijednosti varijabli na ekran. Zamijenite vrijednosti tih dviju varijabli te ponovo ispišite njihove nove vrijednosti.
3. S tipkovnice učitajte broj proizvoljne vrijednosti. Za tako učitani broj provjerite zadovoljava li on uvjet: `10 < broj < 30`, ako je uvjet zadovoljen, ispišite na ekran: "Zadovoljava!", ako pak uvjet nije zadovoljen, ispišite na ekran: "Ne zadovoljava!". Ovaj zadatak potrebno je riješiti uporabom operatora usporedbe, bez korištenja logičkim operatorima i logičkih izraza.
4. Kreirajte listu te ju popunite s 5 proizvoljnih vrijednosti (brojeva). Nakon toga učitajte proizvoljnu vrijednost (broj) s tipkovnice. Unutar petlje `for` provjerite nalazi li se vrijednost koju ste učitali s tipkovnice unutar liste. Ako tražena vrijednost (broj) postoji u listi, ispišite na ekran "Postoji!", u suprotnome ispišite na ekran "Ne postoji!".
5. S tipkovnice učitajte broj proizvoljne vrijednosti. Pomoću ternarnoga operatora detektirajte je li učitani broj paran ili neparan. Ako je učitani broj paran, ispišite na ekran: "Paran!", ako je pak neparan, ispišite na ekran: "Neparan!".
6. Implementirajte dvije funkcije. Prva funkcija neka se zove `paran()` i ona neka ispisuje na ekran: "Paran!", druga funkcija neka se zove `neparan()` i ona neka ispisuje na ekran: "Neparan!". S tipkovnice učitajte broj proizvoljne vrijednosti te pomoću ternarnoga operatora detektirajte je li učitana vrijednost parna ili neparna te ovisno o parnosti pozovite jednu od dviju funkcija koje ste prethodno kreirali.
7. Implementirajte funkciju prototipa `rezultat(var1, var2)`, funkcija može primiti dvije vrijednosti. Tako kreirana funkcija neka u pozivajući dio programa vraća tri vrijednosti dobivene sljedećim aritmetičkim operacijama: `var1*var2`, `var1+var2`, `10*var1+var2`. U glavnome dijelu programa pozovite prethodno implementiranu funkciju te njezine povratne vrijednosti spremite u tri zasebne varijable proizvoljnoga imena i ispišite ih.
8. * Napišite program unutar kojega će biti inicijalizirane dvije varijable proizvoljnoga imena. U prvu varijablu spremite niz znakova: "Hello World!", a u drugu varijablu spremite listu sadržaja `[1, 2, 3, 4]`. Ispišite na ekran niz znakova i listu u obrnutome poretku na dva načina. Najprije vrijednosti ispišite pomoću petlje, a nakon toga vrijednosti ispišite pomoću izrezivanja.

Zadaci označeni zvjezdicom

Zadaci označeni zvjezdicom neće biti obrađeni na predavanjima. Polaznicima se preporučuje da ih pokušaju riješiti samostalno te tako provjere i unaprijede

3. Napredno korištenje funkcijama

Po završetku ovoga poglavlja polaznik će se moći koristiti složenijim tipovima funkcija, a to su:

- *rekurzivne funkcije*
- *bezimene funkcije*
- *ugniježđene funkcije*
- *generatorske funkcije*.

**Trajanje
poglavlja:**
100 min

3.1. Rekurzivne funkcije

Rekurzivne funkcije jesu funkcije koje pozivaju same sebe. Programski kôd koji je napisan pomoću rekurzivnih funkcija često je kraći i pregledniji, no rekurzivne funkcije imaju za posljedicu da njihovo izvođenje traje dulje, a također treba imati na umu veću potrošnju memorije zbog dodatnih poziva funkcije. U nastavku slijedi primjer programskoga kôda koji prikazuje rekurzivni poziv funkcije izračuna matematičke operacije faktoriijela ($n!$).

Primjer 1: Uzmimo za primjer da naša rekurzivna funkcija mora izračunati koliko iznosi 4 faktoriijela. Matematički gledano rezultat $4!$ dobiva se na sljedeći način $4*3*2*1 = 24$.

```
def faktoriijel(x):
    if x <= 1:
        return 1
    else:
        return x * faktoriijel(x - 1)

rez = faktoriijel(4)
print(rez)

Izlaz:
24
```

Napomena

Prototip je zaglavlje funkcije.

Implementacija je tijelo funkcije.

U gornjem primjeru vidimo da prototip naše funkcije izgleda `faktoriijel(x)`, dok u samoj implementaciji funkcije možemo raspoznati dva slučaja.

Dva slučaja:

- Osnovni (bazni) slučaj – na temelju ovoga uvjeta rekurzivno pozivanje funkcija završava.
- Rekurzivni poziv – na temelju ovoga uvjeta funkcija se ponovo poziva. Svakim novim rekurzivnim pozivom funkcija je sve bliže osnovnomu slučaju.

Gornja funkcija `faktoriijel(x)` pozvana je ukupno 4 puta, jednom je pozvana iz glavnoga programa, dok je 3 puta ta funkcija rekurzivno pozvala sama sebe.

Tijek izvođenja rekurzije

Rekurzivno izvođenje može se promatrati u dvjema fazama: fazi pozivanja i fazi vraćanja. U nastavku slijedi opis rekurzije prethodno prikazanoga programskog kôda.

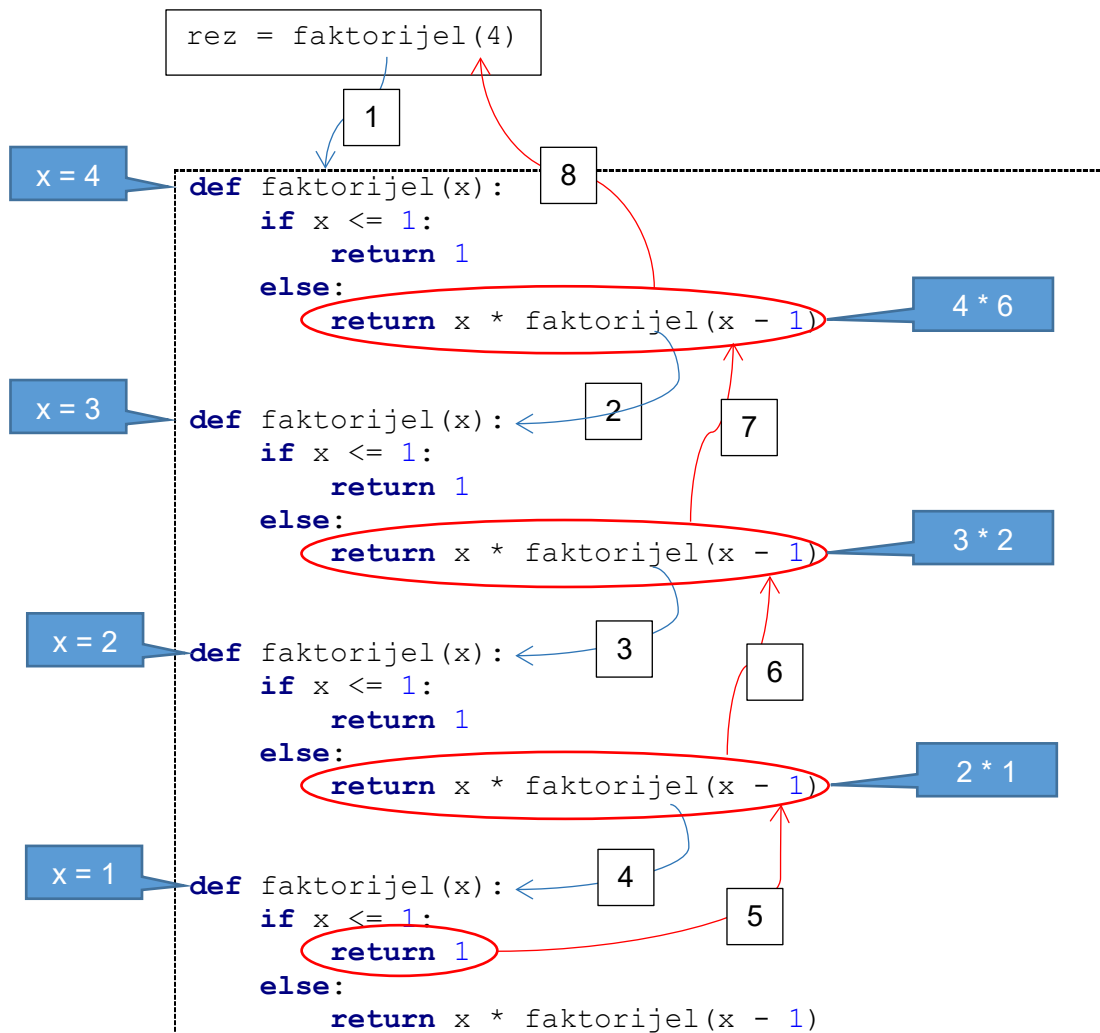
1. Faza pozivanja (spuštanje)

```
faktorijel(4)
= 4 * faktorijel(3)
= 4 * (3 * faktorijel(2))
= 4 * (3 * (2 * faktorijel(1)))
```

2. Faza vraćanja (penjanje)

```
= 4 * (3 * (2 * 1))
= 4 * (3 * 2)
= 4 * 6
= 24
```

Tijek izvođenja rekurzije podijeljen je u dva dijela. Faza pozivanja prikazana je koracima 1 – 4, a faza vraćanja koracima 5 – 8.



Opis koraka

1. Poziva se funkcija `faktorijel(x)`. Parametar `x` poprima vrijednost 4, pri provjeri uvjeta `if` utvrđuje se da uvjet nije zadovoljen, stoga se izvršava dio `else` te se funkcija rekurzivno poziva s argumentom `x - 1`, odnosno vrijednošću 3.
2. Ponavlja se postupak iz prethodnoga koraka, pri čemu parametar `x` sada poprima vrijednost 3.
3. Ponavlja se isti postupak, a parametar `x` poprima vrijednost 2.
4. U ovome koraku parametar `x` poprima vrijednost 1. Za razliku od prethodnih koraka, sada je uvjet `if` zadovoljen.
5. Ovaj korak predstavlja osnovni (bazni) slučaj. Funkcija vraća vrijednost 1 te započinje faza povratka iz rekurzivnih poziva.
6. Rekurzivno se vraća vrijednost parametra `x`, tj. 2 iz toga koraka rekurzivne funkcije pomnožen s povratnom vrijednošću osnovnoga (prethodnoga) koraka, tj. vrijednošću 1. Rezultat koji se vraća jest 2.
7. Rekurzivno se vraća vrijednost parametra `x`, tj. 3 iz toga koraka rekurzivne funkcije pomnožen s povratnom vrijednošću prethodnoga koraka, tj. vrijednošću 2. Rezultat koji se vraća jest 6.
8. Rekurzivno se vraća vrijednost parametra `x`, tj. 4 iz toga koraka rekurzivne funkcije pomnožen s povratnom vrijednošću prethodnoga koraka, tj. vrijednošću 6. Rezultat koji se vraća jest 24.

Rekurzivne funkcije ili iterativni način rješavanja problema

Rekurzivne funkcije mogu se primijeniti u rješavanju brojnih problema, no pritom je važno procijeniti je li njihova primjena opravdana (osim u slučajevima kada je rekurzija neophodan pristup, primjerice kod stabala ili rekurzivnih struktura).

Razlog za takvo promišljanje leži u činjenici da je rekurzivni poziv funkcije u pravilu puno "skuplji" od iterativnoga rješenja. Kako u pogledu brzine izvođenja, tako i u količini zauzete memorije, tim više ako će dani parametri uzrokovati preveliku dubinu gniježđenja. Svaki poziv funkcije uključuje operacije kao što su: adresiranje, postavljanje argumenata i povratnih vrijednosti na stog. Struktura podataka stog rabi se za pohranjivanje rezultata i povratak iz rekurzije.

Osnovni (bazni) slučaj

U slučaju da programer izostavi osnovni (bazni) slučaj u rekurzivnoj funkciji, funkcija bi sama sebe rekurzivno pozivala beskonačan broj puta, što bi dovelo do greške u izračunu vrijednosti i nikada se ne bi postigao uvjet za prekid rekurzije. U primjeru beskonačnoga rekurzivnog

Stog

Stog je apstraktna struktura podataka kod koje se svaki novi element dodaje na vrh te strukture. Također, kada se neki element uzima sa stoga, uzima se onaj koji se nalazi na vrhu stoga, tj. element koji je zadnji stavljen na stog. Iz toga razloga ova struktura podataka zove se još i struktura LIFO (engl. *Last In First Out*).

Stack overflow

Stack overflow dogodit će se ako rekurzivni poziv bude previše dubok ili pak beskonačan.

U tome slučaju prostor za spremanje varijabli i informacija koje su povezane sa svakim rekurzivnim pozivom više ne može stati na stog.

pozivanja koji je prikazan u nastavku vidljivo je da imamo fazu pozivanja, a faza vraćanja ne postoji.

```
rez = faktoriyel (4)
    = 4 * faktoriyel (3)
    = 3 * faktoriyel (2)
    = 2 * faktoriyel (1)
    = 1 * faktoriyel (0)
    = 0 * faktoriyel (-1)
    = -1 * faktoriyel (-2)
    = -2 * faktoriyel (-3)
    = -3 * faktoriyel (-4)
    = -4 * faktoriyel (-5)
    ...
```

Primjer 2: U sljedećemu primjeru vidimo implementaciju funkcije `faktoriyel(x)`, koja je slična prethodno prikazanom programskom kôdu, no u ovome primjeru izostavljen je osnovni (bazni) slučaj, koji je potreban kako bi se postigao uvjet za prekid rekurzije.

```
def faktoriyel(x):
    return x * faktoriyel(x - 1)
```

```
rez = faktoriyel(4)
print(rez)
```

Izlaz:

```
Traceback (most recent call last):
  File "rekurzija.py", line 4, in <module>
    rez = faktoriyel(4)
  File "rekurzija.py", line 2, in faktoriyel
    return x * faktoriyel(x-1)
  File "rekurzija.py", line 2, in faktoriyel
    return x * faktoriyel(x-1)
  File "rekurzija.py", line 2, in faktoriyel
    return x * faktoriyel(x-1)
  [Previous line repeated 990 more times]
RecursionError: maximum recursion depth
exceeded
```

3.2. Vježba: Rekurzivne funkcije

1. Napišite funkciju proizvoljnoga imena koja prima proizvoljnu pozitivnu cjelobrojnu vrijednost. Rezultat je zbroj svih brojeva od 1 do primljene vrijednosti. Na primjer, ako funkcija primi vrijednost 5, u pozivajući dio programa mora se vratiti vrijednost $1+2+3+4+5 = 15$. Ovo je potrebno implementirati pomoću rekurzivne funkcije.
2. Napišite program koji će s tipkovnice učitati jednu proizvoljnu cjelobrojnu vrijednost. Za tako učitani cijeli broj prebrojite i ispišite koliko se parnih znamenki nalazi u njemu. Prebrojavanje parnih znamenki napravite pomoću rekurzivne funkcije. Na primjer, za broj 12345 potrebno je na ekran ispisati vrijednost 2.

3. * Napišite program koji će od korisnika tražiti dvije vrijednosti. Prva vrijednost neka bude neki proizvoljan cijeli broj, a druga vrijednost neka bude znamenka od 0 do 9. Kreirajte rekursivnu funkciju koja će odrediti broj pojavljivanja zadane znamenke u zadanom broju. Prototip funkcije:

```
prebroji(broj, znamenka)
```

3.3. Bezimene funkcije

U programskome jeziku Python bezimena funkcija jest funkcija definirana bez imena. Bezimene funkcije (engl. *anonymous functions*) nazivaju se i lambda-funkcije (engl. *lambda functions*). Za razliku od klasičnih funkcija, koje se definiraju pomoću ključne riječi `def`, lambda-funkcije koriste ključnu riječ `lambda`. Bezimene funkcije jesu kratke, jednolinijske funkcije koje se mogu spremati u varijablu ili upotrebljavati izravno kao argumenti drugih funkcija.

Sintaksa definiranja bezimene funkcije:

```
lambda [argumenti] : [izraz]
```

Bezimene funkcije mogu primiti neograničen broj argumenata. Svi argumenti navode se ispred dvotočke i međusobno su odijeljeni zarezom. Ovaj tip funkcije sintaksno je ograničen na samo jedan izraz, što znači da ne može sadržavati petlje: `for`, `while` ni višelinijске `if` strukture. Ovo ograničenje može se izbjeći korištenjem ternarnog operatora.

Bezimena funkcija obrađuje svoje argumente, izračunava vrijednost izraza i tu izračunatu vrijednost automatski vraća u pozivajući dio programa (nije potrebno eksplicitno navoditi ključnu riječ `return` za vraćanje vrijednosti).

Primjer 1: Korištenje jednostavne bezimene funkcije.

```
uvecajZa10 = lambda x: x + 10
print(uvecajZa10(5))
Izlaz:
15
```

U gornjem primjeru bezimena funkcija je `lambda x : x + 10`. Argument je varijabla `x`, dok je izraz bezimene funkcije `x + 10`. Na temelju toga izraza izračunava se rezultat te se on vraća u pozivajući dio programa, tj. u gornjem primjeru ispisuje se na ekran. Bezimene funkcije nemaju ime, no mogu se spremati u varijablu, u ovome slučaju u varijablu `uvecajZa10`. Ovako kreirana i spremljena bezimena funkcija poziva se identično kao i klasične funkcije.

Primjer 2: U nastavku je prikazan primjer u kojemu je ista funkcionalnost, prethodno implementirana korištenjem lambda-funkcijom, ostvarena pomoću klasične funkcije.

```
def uvecaj(x):
    return x + 10
```

```
print(uvecaj(5))
```

```
Izlaz:
15
```

Primjer 3: Bezimena funkcija koja prima tri argumenta i vraća njihov zbroj.

```
zbroji = lambda a, b, c: a + b + c
```

```
print(zbroji(1, 2, 3))
```

```
Izlaz:
6
```

Složenije korištenje bezimenih funkcija

Bezimene funkcije često se upotrebljavaju u ugrađenim funkcijama, gdje se kao argument očekuje funkcija (bilo klasična, bilo bezimena, tj. lambda-funkcija). Primjeri takvih ugrađenih funkcija u Pythonu jesu: `filter()`, `map()`, `reduce()` i one kao prvi argument očekuju upravo funkciju. U nastavku slijedi primjer s ugrađenom funkcijom `filter()` i slanjem bezimene funkcije kao prvi argument. Prototip funkcije `filter()` jest:

```
filter(funkcija, iterable)
```

Pojašnjenje funkcije `filter()`:

- Prvi je argument funkcija (klasična ili bezimena) koja za svaki element drugoga argumenta vraća `True` ili `False`.
- Drugi je argument objekt po kojemu se vrši iteracija, npr. lista, torka, skup ili bilo koja druga struktura po kojoj je moguće napraviti iteraciju.
- Rezultat je taj da funkcija `filter()` vraća iterator koji sadrži samo elemente za koje funkcija uvjeta vraća `True`.

Primjer 4: Korištenje funkcije `filter()`.

```
lista = [1, 3, 4, 7, 11, 55, 100, 102, 104]
rez = list(filter(lambda e: (e % 2 == 0), lista))
print(rez)
```

```
Izlaz:
[4, 100, 102, 104]
```

Funkcija `filter()` iterira po primljenoj listi. Za svaku vrijednost pomoću bezimene tj. lambda-funkcije zaključuje je li ta vrijednost parna ili neparna. To se zaključuje na temelju izraza bezimene funkcije: `e % 2 == 0`, čiji je rezultat `True` ili `False`. Funkcija `filter()` vraća iterator koji sadrži samo elemente za koje je lambda-funkcija vratila `True`.

Primjer 5: U nastavku je prikazan programski kôd s istom funkcionalnošću kao i u prethodnome primjeru, no umjesto bezimene funkcije upotrijebljena je klasična funkcija. Unutar prikazanoga programskog kôda upotrijebljen je poziv funkcije `list()`, toj funkciji predaje se rezultat funkcije `filter()` iz razloga što funkcija `filter()` vraća iterator.

```
def parnost(x):
    return x % 2 == 0

lista = [1, 3, 4, 7, 11, 55, 100, 102, 104]
rezultat = list(filter(parnost, lista))
print(rezultat)

Izlaz:
[4, 100, 102, 104]
```

Klasična funkcija kao objekt može vratiti bezimenu funkciju. Takav primjer programskoga kôda nalazi se u nastavku. Funkcija prototipa `multipliciranje(n)` prima jedan argument `n`. Vrijednost parametra `n` nakon toga se ugrađuje u bezimenu, tj. lambda-funkciju. Pretpostavimo da je vrijednost parametra `n` u nekome pozivu 3, tada će se preko povratne vrijednosti klasične funkcije `multipliciranje(n)` vratiti bezimena funkcija:

```
lambda a : a * 3
```

Primjer 6: U donjemu programskom kôdu tako vraćena bezimena funkcija sprema se u objekt imena `triPut`. Pozivom bezimene funkcije na način:

```
triPut(3)
```

varijabla `a` u bezimenoj funkciji mijenja se s vrijednošću 3 te se izračunava rezultat izraza `a*3` i vraća se vrijednosti 9.

```
def multipliciranje(n):
    return lambda a: a * n

dvaPut = multipliciranje(2)
triPut = multipliciranje(3)

print(dvaPut(3))
print(triPut(3))

Izlaz:
6
9
```

3.4. Vježba: Bezimene funkcije

1. Napišite bezimenu funkciju koja prima 3 vrijednosti (pretpostavimo da su sve tri vrijednosti pozitivne i strogo veće od 0). Ta bezimena funkcija izračunava i vraća rezultat matematičke formule: $(a*b + c) / c$. U glavnome programu ispišite

rezultat matematičke formule za sljedeće vrijednosti: $a = 5$, $b = 10$; $c = 2$.

2. Nadogradite prethodni zadatak tako da prethodno implementirana bezimena funkcija može primiti 3 vrijednosti u rasponu od $<-\infty, \infty>$. Ako izračun nije moguće provesti, ispišite poruku: "Error!", u suprotnome ispišite rezultat matematičke formule. Napomena: obradite slučaj dijeljenja s 0.
3. * Pronađite na internetu u službenoj Python dokumentaciji funkcionalnost i način korištenja ugrađenom funkcijom `map()`.

Službenu Python 3 dokumentaciju možete pronaći na URL-u: <https://docs.python.org/3/>.

Kreirajte listu od 5 proizvoljnih vrijednosti koje će predstavljati temperaturu izraženu u stupnjevima Celzijevim. Napišite bezimenu funkciju koja će uz pomoć funkcije `map()` generirati vrijednost temperature izražene u farenhajtima. Formula: $(x \times \frac{9}{5}) + 32$, x je vrijednost izražena u stupnjevima Celzijevim. Tako dobivenu listu ispišite na ekran.

3.5. Ugniježdene funkcije

Ovo poglavlje bavi se ugniježđenim funkcijama (engl. *nested functions* ili *inner functions*). Ovaj koncept može se definirati kao funkcija unutar funkcije. Pravila za kreiranje ovakvoga tipa funkcija identična su kao i za klasične funkcije (globalno dohvatljive funkcije), s jedinom razlikom da se ugniježđena funkcija nalazi unutar vanjske funkcije, pa joj je dostupna samo unutar te vanjske funkcije.

Primjer 1: Primjer ugniježdene funkcije prikazan je u nastavku.

```
def vanjska():
    print("Vanjska funkcija.")

    def unutarnja():
        print("Unutarnja funkcija.")

vanjska()

Izlaz:
Vanjska funkcija.
```

U gornjem primjeru možemo vidjeti implementaciju dviju funkcija `vanjska()` i `unutarnja()`. U glavnome programu poziva se funkcija `vanjska()` te se unutar te funkcije izvršava poziv ugrađene funkcije `print()`, unutar funkcije `vanjska()` implementirana je ugniježđena funkcija `unutarnja()`. Ta funkcija nigdje nije pozvana te se iz toga razloga nije ni izvršila.

Primjer 2: Programski kôd u kojemu se poziva ugniježđena funkcija `unutarnja()`.

```
def vanjska():
    print("Vanjska funkcija.")

    def unutarnja():
        print("Unutarnja funkcija.")

    unutarnja()

vanjska()

Izlaz:
    Vanjska funkcija.
    Unutarnja funkcija.
```

Ugniježdjena funkcija može se pozvati samo iz svoje vanjske funkcije

Primijetimo da se u glavnome programu poziva funkcija `unutarnja()`. To je ugniježdjena funkcija globalne funkcije `vanjska()` te je programski jezik Python iz toga razloga javio grešku da ona nije definirana. Vrlo je bitno imati na umu da se ugniježdjena funkcija može dohvatiti samo unutar vanjske funkcije.

Primjer 3: Poziv unutarnje funkcije iz glavnoga dijela programa.

```
def vanjska():
    print("Vanjska funkcija.")

    def unutarnja():
        print("Unutarnja funkcija.")

unutarnja()

Izlaz:
    Traceback (most recent call last):
      File "test.py", line 7, in <module>
        unutarnja()
    NameError: name 'unutarnja' is not defined
```

Primjer 4: Slanje parametara ugniježdjenoj funkciji.

```
def vanjska(x):
    print("Vanjska funkcija, x =", x)

    def unutarnja(y):
        print("Unutarnja funkcija, y =", y)

    unutarnja(x * 2)

vanjska(10)

Izlaz:
    Vanjska funkcija, x = 10
    Unutarnja funkcija, y = 20
```

Ugniježdene funkcije često se upotrebljavaju kako bi se postigla enkapsulacija (engl. *encapsulation*), tj. skrivanje ponašanja nekoga dijela programskoga kôda od ostatka programa. Uzmimo za primjer gornji programski kôd. Implementacija funkcije `unutarnja()` može se pozvati samo iz funkcije `vanjska()`, što znači da je programski kôd napisan u funkciji `unutarnja()` skriven od globalnoga dosega.

3.6. Vježba: Ugniježdene funkcije

1. Isprobajte korištenje ugniježđenih funkcija. Vanjska funkcija imena `izracun()` neka prima dva parametra. Unutar vanjske funkcije kreirajte dvije unutarnje funkcije: `parniZbroj()` i `neparniZbroj()`. Funkcija `parniZbroj()` neka vraća rezultat matematičke formule $2*a+5*b$, a funkcija `neparniZbroj()` neka vraća rezultat matematičke formule $a*b-10$. Koja će se od dviju kreiranih unutarnjih funkcija pozvati, određuje se na temelju parnosti zbroja primljenih parametara $a+b$. U glavnome programu ispišite rezultat na ekran.

3.7. Generatorske funkcije

Generatorske funkcije (engl. *generator functions*) kao rezultat vraćaju generator, tj. objekt po kojemu je moguće vršiti iteraciju. U nastavku slijedi usporedba klasičnih i generatorskih funkcija.

Klasične funkcije	Generatorske funkcije
koriste se ključnom riječju <code>return</code>	koriste se ključnom riječju <code>yield</code>
vraćaju vrijednost	vraćaju generator

Generatorske funkcije u načelu su klasične funkcije, no za razliku od klasičnih funkcija koje se za povratnu vrijednost koriste ključnom riječju `return`, generatorske se funkcije koriste ključnom riječju `yield`. Klasične funkcije izračunatu vrijednost vraćaju samo jednom, dok generatorske funkcije vraćaju generator objekt (što je zapravo pojednostavljeni iterator). Tako dobivenim generatorom može se iterirati po njegovim elementima. Izvršavanje generatorske funkcije moguće je pauzirati i nastavljati iz glavnoga programa ovisno o potrebama (engl. *on the fly*). Generator (kao i iterator) predstavljaju pogled (engl. *view*) na kolekciju, tj. kolekcija se ne kopira, već se samo pristupa njezinim elementima.

Primjer 1: U nastavku slijedi jednostavan primjer na temelju kojega će se objasniti ponašanje generatorskih funkcija.

```
def generatorska(x):
    while x > 0:
        yield x
        x -= 1
```

```
g = generatorska(3)
```

```
print(next(g))
print(next(g))
print(next(g))
print(next(g))
```

Izlaz:

```
3
2
1
```

```
Traceback (most recent call last):
  File "test.py", line 11, in <module>
    print( next(g) )
StopIteration
```

U gore prikazanome programskom kôdu vidimo poziv funkcije `generatorska(3)`. Taj poziv ne izvršava generatorsku funkciju, već u varijablu `g` vraća generator. Elementima generatora pristupa se pozivom funkcije `next()`. Kada prvi put pozovemo funkciju `next(g)`, programski kôd generatorske funkcije počinje se izvršavati do prvoga pojavljivanja ključne riječi `yield`. Ključna riječ `yield` u gornjemu primjeru otpušta vrijednost varijable `x`. Nakon što vrijednost varijable `x` bude otpuštena, generatorska funkcija ne završava, već se pamti cjelokupno stanje iste do sljedećega poziva. Pri sljedećemu pozivu, tj. naredbom `next(g)` programski kôd u funkciji će se nastaviti izvršavati od posljednjega izvršavanja naredbe `yield`, sve dok se ponovo ne pojavi ključna riječ `yield` ili pak dok generatorska funkcija ne završi. Kada izvođenje generatorske funkcije dođe do kraja ili pak do naredbe `return`, generator završetak iteracije signalizira iznimkom `StopIteration`. Ovu iznimku možemo vidjeti kod ispisa greške četvrtoga poziva funkcije `next(g)`. Ova iznimka dogodila se zato što je generator došao do kraja, tj. vrijednost varijable `x` postala je 0 te je petlja `while` unutar koje se poziva naredba `yield` završila.

Primjer 2: U nastavku se nalazi primjer kôda u kojemu se u varijablu `g` vratio generator, a po elementima toga generatora iterira se pomoću petlje `for`.

```
def generatorska(vrijednost):
    i = 0
    while i < vrijednost:
        yield i
        i += 1
```

```
g = generatorska(4)
for e in g:
    print(e)
```

Izlaz:

```
0
1
2
3
```

Primjer 3: Sljedeći primjer prikazuje slučaj u kojemu se u varijablu `g` vraća generator. Pomoću funkcije `list()` elementi danoga generatora pretvaraju se u listu.

```
def generatorska(vrijednost):
    i = 0
    while i < vrijednost:
        yield i
        i += 1

g = generatorska(4)

print(list(g))
```

Izlaz:

```
[0, 1, 2, 3]
```

Napomena:

Povratna vrijednost generatorske funkcije, a to je generator, omogućuje samo jedan prolaz kroz svoje elemente.

Primjer 4: U nastavku je prikazan primjer u kojemu se najprije putem petlje `for` iterira po svim elementima generatora, a nakon toga se pozivom funkcije `list()` želi dohvatiti sadržaj generatora. Budući da je petlja `for` već iterirala kroz sve elemente generatora, generator je iscrpljen te naknadni pokušaj dohvaćanja elemenata daje praznu strukturu.

```
def generatorska(vrijednost):
    i = 0
    while i < vrijednost:
        yield i
        i += 1

g = generatorska(4)

for e in g:
    print(e)

print(list(g))
```

Izlaz:

```
0
1
2
3
[]
```

3.8. Vježba: Generatorske funkcije

1. Implementirajte generatorsku funkciju imena `danUTjednu()`, funkcija mora vratiti generator koji generira dane u tjednu. Nakon što svi dani u tjednu budu otpušteni, ne kreće se ponovo s

generiranjem vrijednosti od ponedjeljka. U glavnome programu pozivom generatorske funkcije ispišite sve dane u tjednu.

2. Implementirajte generatorsku funkciju imena `jedanDvaTri()` koja vraća generator koji generira vrijednosti: "Jedan", "Dva", "Tri", "Jedan", "Dva", "Tri", "Jedan", "Dva", "Tri" i tako u beskonačnost. U glavnome programu ispišite sekvencu proizvoljan broj puta: "Jedan", "Dva", "Tri".
3. * Napišite generatorsku funkciju imena `bottles(brojBoca, nazivNapitka)`. Ova funkcija mora vratiti generator koji generira stihove pjesme: "99 Bottles of Beer". Svaki stih ove pjesme ima za jedan napitak manje od prethodnoga stiha, sve dok se ne potroše svi napitci. Zadana vrijednost za broj boca jest 99, dok je zadana vrijednost za naziv napitka "beer". U nastavku slijedi primjer ispisa za parametar `brojBoca = 4`.

4 bottles of beer on the wall, 4 bottles of beer.
Take one down and pass it around, 3 bottles of beer on the wall.

3 bottles of beer on the wall, 3 bottles of beer.
Take one down and pass it around, 2 bottles of beer on the wall.

2 bottles of beer on the wall, 2 bottles of beer.
Take one down and pass it around, 1 bottle of beer on the wall.

1 bottle of beer on the wall, 1 bottle of beer.
Take one down and pass it around, no more bottles of beer on the wall.

No more bottles of beer on the wall, no more bottles of beer.
Go to the store and buy some more, 99 bottles of beer on the wall.

3.9. Pitanja za ponavljanje: Napredno korištenje funkcijama

1. Što su to rekurzivne funkcije?
2. Je li rekurzivni poziv funkcije efikasniji od iterativnoga rješenja nekoga problema?
3. S obzirom na prethodno pitanje, zašto je tomu tako?
4. Što će se dogoditi ako je rekurzivna funkcija napisana tako da nema osnovnoga (baznoga) slučaja?
5. Što je to bezimena funkcija?
6. Koji je drugi naziv za bezimenu funkciju?
7. Kako izgleda sintaksa bezimenih funkcija?
8. Koliko argumenata može primiti bezimena funkcija?
9. Je li bezimenu funkciju moguće spremiti u objekt (varijablu)?

10. Što su to ugniježdene funkcije?
11. Kada se ugniježdene funkcije najčešće upotrebljavaju?
12. Koja je razlika između klasičnih funkcija i generatorskih funkcija?

4. Iznimke

Trajanje
poglavlja:
90 min

Po završetku ovoga poglavlja polaznik će moći:

- *obrađivati iznimke*
- *podizati iznimke*
- *istovremeno obrađivati više tipova iznimaka*
- *prosljeđivati iznimke*
- *koristiti se programskim blokovima `else` i `finally`.*

S povećanjem kompleksnosti programskoga kôda povećava se i vjerojatnost pojave iznimnih situacija, tj. grešaka. Takve situacije koje se mogu dogoditi tijekom izvršavanja programa nazivaju se iznimke (engl. *exceptions*). U modernim programskim jezicima, pa tako i u Pythonu, iznimke je moguće jednostavno obraditi pomoću ugrađenih mehanizama, čime se omogućuje kontrolirano i stabilno izvršavanje programskoga kôda. Greške mogu biti sintaksne, semantičke i logičke.

Primjer 1: Sintaksne greške – programski kôd ne može se pokrenuti jer nije u skladu s pravilima odabranoga programskog jezika (npr. nedostaje dvotočka, pogrešno indentiranje).

```
print("Sintaksna greška!")
for e in range(1,11)
    print(e)
```

Izlaz:

```
File "iznimke.py", line 2
    for e in range(1,11)
                        ^
SyntaxError: expected ':'
```

Primjer 2: Semantičke greške – programski kôd napisan je prema pravilima odabranoga programskog jezika te se uredno pokrene, ali dolazi do greške tijekom izvođenja programa (npr. dijeljenje s nulom, neprikladna operacija nad tipovima podataka). U ovome tipu greške odabrana je ispravna logika, no program ne daje ispravan rezultat.

```
print("Semantička greška!")
```

```
a = 10
b = "abcd"
rez = a + b
```

Izlaz:

```
Traceback (most recent call last):
  File "iznimke.py", line 5, in <module>
    rez = a + b
        ~~~~
TypeError: unsupported operand type(s) for +:
'int' and 'str'
Semantička greška!
```

Primjer 3: Logičke greške – programski kôd izvršava se bez greške, ali daje netočan rezultat zbog pogrešno implementirane logike.

```
print("Logička greška!")

a = 10
b = 20

prosijek = a + b / 2 # prvo dijeli b s 2, pa zbraja a

print("Prosijek:", prosijek)
```

Izlaz:

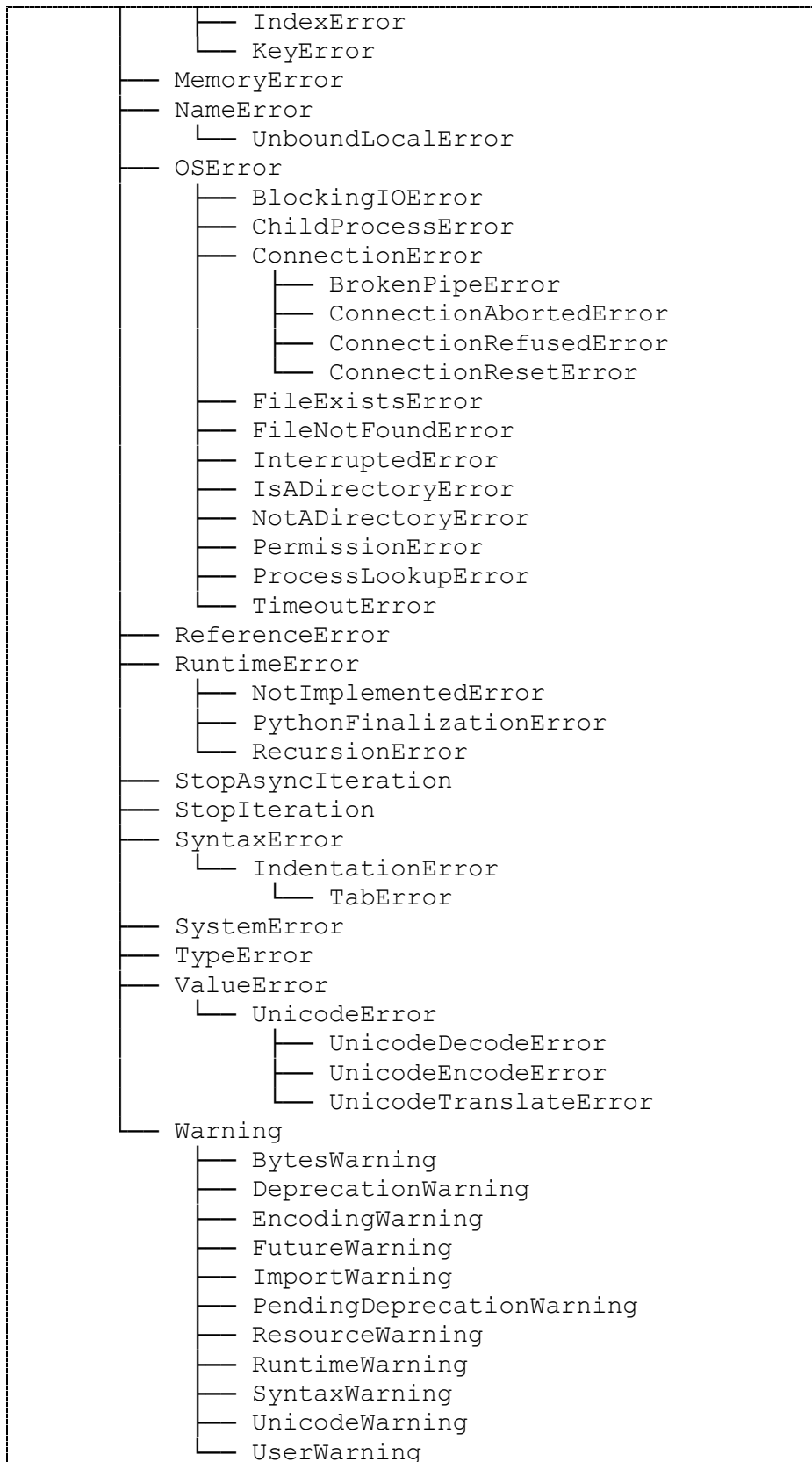
```
Logička greška!
Prosijek: 20.0
```

Greške otkrivene tijekom izvršavanja programa nazivaju se iznimkama (engl. *exceptions*). Takve greške ne moraju nužno dovesti do prekida izvođenja programa, već ih je moguće obraditi, o čemu će biti više riječi u ovome poglavlju.

U Pythonu sve iznimke nasljeđuju ugrađeni osnovni razred `BaseException`. `BaseException` je osnovni razred za sve iznimke, dok je `Exception` osnovni razred za iznimke koje ne uzrokuju prekid programa. Sve korisničke iznimke trebale bi nasljeđivati razred `Exception`.

U nastavku je prikazana hijerarhija iznimaka razreda `BaseException`. Kao što je iz hijerarhije moguće vidjeti, postoji mnogo različitih iznimaka koje su složene u smislene cjeline, čime se programerima omogućuje da iznimke obrađuju na različite načine. Na primjer, podrazredi `IndexError`, `KeyError` pripadaju razredu `LookupError`. Ako se u programskome kôdu hvata iznimka `IndexError`, onda se neće uhvatiti iznimka `KeyError`, no ako se hvata iznimka `LookupError`, onda će se uhvatiti bilo koja od dviju iznimaka koje pripadaju razredu `LookupError`.

```
BaseException
├── BaseExceptionGroup
├── GeneratorExit
├── KeyboardInterrupt
├── SystemExit
├── Exception
│   ├── ArithmeticError
│   │   ├── FloatingPointError
│   │   ├── OverflowError
│   │   └── ZeroDivisionError
│   ├── AssertionError
│   ├── AttributeError
│   ├── BufferError
│   ├── EOFError
│   ├── ExceptionGroup [BaseExceptionGroup]
│   ├── ImportError
│   │   └── ModuleNotFoundError
│   └── LookupError
```



4.1. Primjeri iznimaka

U ovome poglavlju obrađene su iznimke koje se najčešće pojavljuju. Detaljniji opis ostalih iznimaka moguće je pronaći u službenoj Python 3

dokumentaciji. Također, treba napomenuti da nije dobra praksa hvatati određene tipove iznimaka poput: `SyntaxError`, `NameError`.

SyntaxError

Ovaj tip greške javlja se kada programer napiše programski kôd koji nije u skladu s pravilima (sintaksom) Pythona. Zbog ovoga tipa greške interpreter ne zna kako interpretirati napisani programski kôd.

Primjer 1: `SyntaxError` iznimka

```
>>> return
SyntaxError: 'return' outside function
>>>
>>> None = 1
SyntaxError: can't assign to keyword
>>>
>>> 10 20
SyntaxError: invalid syntax
>>>
>>> iff x==0:
SyntaxError: invalid syntax
>>>
```

Python nije 100 % interpreterski programski jezik, već se svaka skripta prije izvršavanja kompajlira u bajt-kôd (engl. *bytecode*). Već prilikom toga procesa otkrivaju se sintaksne greške, što znači da je u trenutku izvršavanja programa skripta već prevedena i sintaksne greške nisu pronađene. To znači da se sintaksne greške ne mogu uhvatiti unutar skripte osim u nekoliko posebnih slučajeva. Tip greške `SyntaxError` moguće je uhvatiti ako se ona dogodi u programskome kôdu koji se izvršava pozivom funkcija: `exec()` ili `eval()` ili prilikom uključivanja modula u program pomoću naredbe `import`.

NameError

Ovaj tip greške javlja se kada korištena varijabla nije definirana, tj. kada joj nije pridružena nikakva vrijednost. Uz informaciju o tipu greške ispisuje se i ime varijable koja nije definirana.

Primjer 2: `NameError` iznimka

```
>>> novaVarijabla
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    novaVarijabla
NameError: name 'novaVarijabla' is not defined
>>>
>>>
>>> print(novaVarijabla)
Traceback (most recent call last):
  File "<pyshell#10>", line 1, in <module>
    print(novaVarijabla)
NameError: name 'novaVarijabla' is not defined
>>>
```

```
>>> 10 + novaVarijabla
Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    10 + novaVarijabla
NameError: name 'novaVarijabla' is not defined
>>>
```

`NameError` jedna je od iznimaka koje se u praksi rijetko hvataju. Programer bi trebao napisati kôd tako da se ova iznimka ne dogodi, tj. mora definirati sve varijable prije njihove uporabe.

TypeError

Ovaj tip greške javlja se kada operaciju ili funkciju primjenjujemo nad varijablom pogrešnog tipa podataka. Uz informaciju o tipu greške ispisuje se i njezin opis.

Primjer 3: `TypeError` iznimka

```
>>> 10 + "niz znakova"
Traceback (most recent call last):
  File "<pyshell#12>", line 1, in <module>
    10 + "niz znakova"
TypeError: unsupported operand type(s) for +: 'int'
and 'str'
>>>
>>> len(5)
Traceback (most recent call last):
  File "<pyshell#13>", line 1, in <module>
    len(5)
TypeError: object of type 'int' has no len()
>>>
```

IndexError

Ovaj tip greške javlja se kada operacija ili funkcija prima argument iz, na primjer, liste, niza znakova i slično, a vrijednost na željenome indeksu nije definirana (nepostojeći indeks). Uz informaciju o tipu greške ispisuje se i njezin opis.

Primjer 4: `IndexError` iznimka

```
>>> lista = [1, 2, 3, 4]
>>> lista[1]
2
>>> lista[4]
Traceback (most recent call last):
  File "<pyshell#23>", line 1, in <module>
    lista[4]
IndexError: list index out of range
>>>
>>> nizZnakova = "SRCE"
>>> nizZnakova[4]
Traceback (most recent call last):
  File "<pyshell#25>", line 1, in <module>
    nizZnakova[4]
IndexError: string index out of range
```

ValueError

Ovaj tip greške javlja se kada operacija ili funkcija prima argument ispravnoga tipa, ali s neispravnom vrijednošću. Uz informaciju o tipu greške ispisuje se i njezin opis.

Primjer 5: ValueError iznimka

```
>>> int("55")
55
>>> int("test")
Traceback (most recent call last):
  File "<pyshell#34>", line 1, in <module>
    int("test")
ValueError: invalid literal for int() with base 10:
'test'
>>>
>>> a, b = [1, 2, 3]
Traceback (most recent call last):
  File "<pyshell#37>", line 1, in <module>
    a, b = [1, 2, 3]
ValueError: too many values to unpack (expected 2)
>>>
```

KeyError

Ovaj tip greške javlja se kada se u rječniku ne nalazi traženi ključ. Uz informaciju o tipu greške ispisuje se naziv traženoga ključa koji ne postoji u rječniku.

Primjer 6: KeyError iznimka

```
>>> rjecnik = {"jedan" : "one", "dva" : "two"}
>>> rjecnik["jedan"]
'one'
>>> rjecnik["jedaan"]
Traceback (most recent call last):
  File "<pyshell#42>", line 1, in <module>
    rjecnik["jedaan"]
KeyError: 'jedaan'
>>>
```

AttributeError

Ovaj tip greške javlja se kada se nad nekim objektom poziva atribut koji ne postoji. U donjem primjeru nad objektom `nizZnakova` poziva se metoda `velika()`, no kako ta metoda nije implementirana i Python ju ne može pronaći, vraća se ovaj tip greške. Uz informaciju o tipu greške ispisuje se i njezin opis.

Primjer 7: AttributeError iznimka

```
>>> nizZnakova = "zagreb"
>>> nizZnakova.upper()
'ZAGREB'
>>> nizZnakova.velika()
Traceback (most recent call last):
```

```
File "<pyshell#48>", line 1, in <module>
    nizZnakova.velika()
AttributeError: 'str' object has no attribute
'velika'
>>>
```

4.2. Obrada iznimaka

U dosadašnjem dijelu ovoga poglavlja obrađene su najčešće iznimke koje se pojavljuju. No sve te iznimke blokiraju, tj. zaustavljaju daljnje izvršavanje programa te same po sebi, ako ih se ne obradi, nisu korisne. Iz toga razloga uvedena je naredba `try ... except` koja omogućuje hvatanje i obradu iznimaka (engl. *exception handling*).

Unutar tijela naredbe `try` izvodi se neki programski kôd. Ako se unutar tijela naredbe `try` pojavi iznimka (uključujući programski kôd koji se poziva iz ovoga tijela), daljnje se izvođenje tijela naredbe `try` prekida te se kreće izvoditi tijelo naredbe `except`. Sintaksa:

```
try:
    <tijeloTryNaredbe>
except:
    <tijeloExceptNaredbe>
```

Primjer 1: U nastavku slijedi pokazni primjer u kojemu je uzrokovana iznimka `IndexError`. Definirana je lista od 4 elementa, a unutar bloka `try` pokušava se ispisati vrijednost na indeksu 4 koja ne postoji. Zbog toga se izvršavanje tijela `try` prekida, a pokreće se tijelo `except`.

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

try:
    print(lista[4])
except:
    print("Podignuta je iznimka!")

Izlaz:
    Podignuta je iznimka!
```

Primjer 2: Ako se kôd unutar bloka `try` izvrši bez podignute iznimke, tijelo `except` se preskače.

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

try:
    print(lista[3])
except:
    print("Podignuta je iznimka!")

Izlaz:
    4
```

U ovakvome načinu obrade iznimaka naredba `except` hvata sve podignute iznimke te se sve one obrađuju jednako. To znači da se jednako obrađuje iznimka `IndexError` i iznimka `ValueError` i svaka druga podignuta iznimka. Za selektivno hvatanje iznimaka nakon ključne

riječi `except` navodi se tip iznimke. Takvim pristupom možemo uhvatiti i obraditi iznimnu situaciju na način specifičan za tu iznimku. Ako ne želimo selektivno hvatati sve potencijalne iznimke koje se mogu dogoditi, na kraju možemo napisati samo ključnu riječ `except`, koja će uhvatiti sve iznimke koje nisu prethodno selektivno navedene.

Sintaksa programskoga kôda koji selektivno obrađuje iznimke:

```
try:
    <tijeloTryNaredbe>
except <iznimka1>:
    <tijeloExceptNaredbe1>
except <iznimka2>:
    <tijeloExceptNaredbe2>
except:
    <tijeloExceptNaredbe>
```

Napomena: Dobra praksa obrade iznimaka jest selektivno hvatati iznimke, tj. sveobuhvatni blok `except` rabiti nakon što su iscrpljene sve selektivne iznimke. Sveobuhvatni blok `except` obično služi za daljnje prosljeđivanje ili pak bilježenje iznimaka, a ne za specifičnu obradu.

U nastavku slijede dva primjera u kojima se u bloku `try` podižu iznimke (žutom bojom označen je blok `except` koji je uhvatio iznimku).

Primjer 3:

```
try:
    lista = [1, 2, 3, 4]
    print(lista[4])
except ValueError:
    print("Podigla se ValueError iznimka!")
except IndexError:
    print("Podigla se IndexError iznimka!")
Izlaz:
    Podigla se IndexError iznimka!
```

U ovome primjeru podiže se iznimka `IndexError` jer se pokušava dohvatiti vrijednost na indeksu 4 koji ne postoji u definiranoj listi.

Primjer 4:

```
try:
    print("niz znakova" + 10)
except ValueError:
    print("Podigla se ValueError iznimka!")
except IndexError:
    print("Podigla se IndexError iznimka!")
except:
    print("Nešto je pošlo po zlu!")
Izlaz:
    Nešto je pošlo po zlu!
```

U drugome primjeru podiže se iznimka `TypeError`, no kako tu iznimku nismo selektivno obuhvatili, nju hvata sveobuhvatni blok `except`. Kao što je gore u tekstu navedeno, sveobuhvatni blok `except` obično se upotrebljava za prosljeđivanje iznimaka dalje. Ovo je primjer loše prakse. Ako se ovu iznimku želi obraditi prema pravilima struke, onda bi ovu iznimku trebalo selektivno obraditi. Takva obrada prikazana je u sljedećemu primjeru.

Primjer 5:

```
try:
    print("niz znakova" + 10)
except ValueError:
    print("Podigla se ValueError iznimka!")
except IndexError:
    print("Podigla se IndexError iznimka!")
except TypeError:
    print("Podigla se TypeError iznimka!")
```

Izlaz:

```
Podigla se TypeError iznimka!
```

Objekt iznimke

Putem objekta iznimke moguće je pristupiti detaljima (opisu).

Primjer sintakse s kojom se pristupa objektu iznimke:

```
try:
    <tijeloTryNaredbe>
except <iznimka1> as <objektIznimke>:
    <tijeloExceptNaredbe1>
except:
    <tijeloExceptNaredbe>
```

Primjer 6: Sljedeći primjer prikazuje ispis detalja iznimke putem objekta imena `e`. Ovaj je objekt opcionalan, no ako ga ne navedemo, onda opisu iznimke nije moguće pristupiti. Ovi nam podaci omogućuju da na ekran (ili negdje drugdje, na primjer u datoteku) zapišemo što se točno dogodilo i time omogućimo lakši pronalazak i ispravljanje greške.

```
try:
    print(1/0)
except ZeroDivisionError as e:
    print(e)
```

Izlaz:

```
division by zero
```

Objekt iznimke

Objekt iznimke priprada razredu iznimke koja se dogodila, u ovome primjeru to je iznimka `ZeroDivisionError`. Objekt u sebi sadrži informacije poput poruke o grešci, *stack trace*, itd. Pozivom naredbe `print(e)` objekt `e` implicitno se pretvara u `string`, čime se ispisuje samo poruka o grešci.

4.3. Vježba: Iznimke I

1. Napišite programski kôd unutar kojega postoji vjerojatnost da se podigne iznimka `ValueError`. Pokrenite program i namjestite ulazne vrijednosti koje će podići iznimku `ValueError`.

2. Programski kôd iz prethodnoga zadatka nadogradite tako da uhvatite iznimku koja se dogodila (postavite hvatanje svih iznimaka, a ne selektivno) te na ekran ispišite poruku "Iznimka je uhvaćena!".
3. Napišite proizvoljni programski kôd unutar kojega se mogu dogoditi barem dvije vrste iznimaka. Selektivno uhvatite iznimke i ispišite koji je tip iznimke uhvaćen.
4. Nadogradite prethodni zadatak tako da uz ispis tipa uhvaćene iznimke ispišete i detalje (opis) iznimke koje je generirao Python.

4.4. Podizanje iznimaka

Do ovoga je poglavlja sve iznimke podizao Python. Međutim, iznimke ne mora podizati isključivo Python, može ih podići (engl. *raise*) i sam programer. Programer podiže iznimke kada procijeni da bi za određene ulazne vrijednosti u nastavku izvođenja programa moglo doći do greške.

Za podizanje iznimaka rabi se ključna riječ `raise`. Sintaksom prikazanom u nastavku kreira se objekt koji je tipa `ImeIznimke`. Argument `<opcionalniArgument>` predstavlja poruku o grešci i tipa podataka je niz znakova, tj. `string`.

```
raise ImeIznimke(<opcionalniArgument>)
```

Primjer 1: U nastavku slijedi primjer programskoga kôda u kojemu se podiže iznimka `ValueError` bez opcionalnoga argumenta.

```
>>> raise ValueError #ili 'raise ValueError()'
Traceback (most recent call last):
  File "<pyshell#51>", line 1, in <module>
    raise ValueError
ValueError
>>>
```

Primjer 2: U sljedećemu primjeru programskoga kôda također se podiže iznimka imena `ValueError`, no ovdje se prenosi i opcionalni argument koji detaljnije opisuje uzrok greške. U obama slučajevima kreira se objekt tipa `ValueError`.

```
>>> raise ValueError("Ručno se podigla iznimka!")
Traceback (most recent call last):
  File "<pyshell#52>", line 1, in <module>
    raise ValueError("Ručno se podigla iznimka!")
ValueError: Ručno se podigla iznimka!
>>>
```

Primjer 3: U nastavku slijedi primjer u kojemu se podiže i hvata ručno podignuta iznimka.

```
try:
    raise ValueError("Iznimka je podignuta!")
```

```
except ValueError as e:
    print(e)
```

Izlaz:
Iznimka je podignuta!

4.5. Istovremena obrada više tipova iznimaka

U dosadašnjem dijelu ovoga priručnika iznimke su se u blokovima `except` hvatale pojedinačno. No ponekad je više različitih tipova iznimaka potrebno obraditi na isti način. U Pythonu je to omogućeno tako da se u zaglavlju bloka `except` navede n-torka koja sadrži sve iznimke koje se žele hvatati unutar toga bloka.

Sintaksa izgleda ovako:

```
except (ImeIznimke1, ImeIznimke2, ...):
```

Primjer 1: U nastavku je prikazan primjer programskoga kôda u kojemu se mogu izazvati dvije vrste iznimaka: `ValueError` i

`ZeroDivisionError`. U donjem primjeru iznimku `ValueError` moguće je prouzrokovati tako da se preko tipkovnice unese niz znakova koji nije moguće pretvoriti u broj (npr. unos sadrži slova). Iznimka `ZeroDivisionError` prouzrokuje se tako da se za vrijednost varijable nazivnik unese 0. U zaglavlju bloka `except` moguće je vidjeti n-torku unutar koje su navedene obje iznimke i kao takve obrađuju se istim programskim kôdom.

```
try:
    brojnik = int(input("Brojnik: "))
    nazivnik = int(input("Nazivnik: "))
    rezultat = brojnik / nazivnik
    print(rezultat)
except (ValueError, ZeroDivisionError) as e:
    print("Dogodila se iznimka, opis:", e)
except:
    print("Ostale iznimke!")
```

Izlaz 1:
Brojnik: **broj**
Dogodila se iznimka, opis: invalid literal
for int() with base 10: 'broj'

Izlaz 2:
Brojnik: **10**
Nazivnik: **0**
Dogodila se iznimka, opis: division by zero

Izlaz 3:
Brojnik: **^C #(Ctrl+C)**
Ostale iznimke!

4.6. Prosljeđivanje iznimaka

Ako se u pozivajući dio programa želi vratiti iznimka koja je uhvaćena unutar, na primjer, funkcije ili metode, to se može napraviti pomoću ključne riječi `raise`.

Primjer 1: U funkciji `kolicnik()`, koja je implementirana u programskome kôdu prikazanom u nastavku, prilikom dijeljenja s nulom Python podiže iznimku `ZeroDivisionError`. Iznimke koje će se dogoditi unutar funkcije `kolicnik()` hvataju se u sveobuhvatnome bloku `except` (blok koji hvata sve iznimke koje se dogode). Unutar bloka `except` u funkciji pomoću ključne riječi `raise` svaka iznimka koja se dogodi prosljeđuje se u pozivajući dio programa. Tako prosljeđena iznimka iz funkcije `kolicnik()` hvata se u glavnome programu. Poziv funkcije `kolicnik()` u glavnome programu nalazi se unutar bloka `try`, dok blok `except` hvata iznimku `ZeroDivisionError`. Sve ostale iznimke koje će se dogoditi bit će prosljeđene u pozivajući dio programa, no neće biti obrađene jer za njih ne postoji odgovarajući blok `except`.

```
def kolicnik(a, b):
    try:
        rez = a / b
        return rez
    except:
        raise

try:
    brojnik = int(input("Brojnik: "))
    nazivnik = int(input("Nazivnik: "))
    rezultat = kolicnik(brojnik, nazivnik)
    print(rezultat)
except ZeroDivisionError as e:
    print("Dogodila se iznimka, opis:", e)
```

```
Izlaz 1:
    Brojnik: 1
    Nazivnik: 0
    Dogodila se iznimka, opis: division by zero
```

```
Izlaz 2:
    Brojnik: 6
    Nazivnik: 2
    3.0
```

U slučaju da se dogodi iznimka, Python će tražiti najbliži blok `except`. Ako iznimka nije nigdje uhvaćena, Python će prekinuti izvođenje programa i ispisati poruku o grešci.

Uvijek se ulazi u prvi blok `except` u kojemu je uvjet zadovoljen. Na primjer, napisana su dva bloka `except`. Najprije je napisan blok: `except ZeroDivisionError` te je nakon njega napisan blok: `except ArithmeticError`. Prilikom dijeljenja s nulom oba su uvjeta zadovoljena (`ZeroDivisionError` je potklasa `ArithmeticError`), no kako je `ZeroDivisionError` napisan prvi, on će se i izvršiti.

4.7. Korištenje vlastitih tipova iznimaka

Većina uobičajenih iznimaka može se obraditi pomoću ugrađenih tipova iznimaka, kao što su `ValueError`, `ZeroDivisionError`, `IndexError`. Međutim, u kompleksnijim programima često se pojavljuje potreba za definiranjem vlastitih tipova iznimaka koje bolje opisuju specifične greške u aplikaciji. Takve iznimke zovu se vlastite ili korisnički definirane iznimke.

Kreiranje vlastite iznimke radi se tako da se definira nova klasa koja nasljeđuje neku od postojećih klasa iznimaka, najčešće klasu `Exception`.

```
class MojaIznimkaError(Exception):
    """Korisnički definirana iznimka."""
    pass
```

Vlastite iznimke omogućuju preciznije hvatanje specifičnih grešaka u programu. To je osobito korisno u velikim projektima ili modulima, gdje je važno razlikovati različite vrste grešaka. Na primjer, možemo unutar jednoga bloka `except` hvatati iznimke koje smo mi definirali, a unutar drugoga bloka `except` standardno definirane iznimke.

Primjer 1: U ovome primjeru funkcija `provjeriBroj()` podiže vlastitu iznimku imena `NegativanBrojError` kada je vrijednost parametra `broj` negativna. Za ručno podizanje iznimke rabi se ključna riječ `raise`.

```
class NegativanBrojError(Exception):
    """Iznimka koja se podiže za negativan broj."""
    pass

def provjeriBroj(broj):
    if broj < 0:
        raise NegativanBrojError(f"Greška: {broj}")
    else:
        print("Broj je ispravan:", broj)
```

```
provjeriBroj(5)
provjeriBroj(-3)
```

Izlaz:

```
Broj je ispravan: 5
Traceback (most recent call last):
  File "iznimke.py", line 14, in <module>
    provjeriBroj(-3)
    ~~~~~^
  File "iznimke.py", line 8, in provjeriBroj
    raise NegativanBrojError(f"Greška:
{broj}")
NegativanBrojError: Greška: -3
```

Primjer 2: U sljedećemu primjeru prvo se hvata korisnički definirana iznimka imena `NegativanBrojError`, dok se sve ostale iznimke hvataju u općem bloku `except Exception`. Takav pristup omogućuje da se specifične greške jasno identificiraju i obrade. Nepoznate se greške i dalje mogu detektirati i logirati za daljnju analizu.

```
class NegativanBrojError(Exception):
    """Iznimka koja se podiže za negativan broj."""
    pass

def provjeriBroj(broj):
    if broj < 0:
        raise NegativanBrojError(f"Greška: {broj}")
    else:
        print("Broj je ispravan:", broj)

try:
    broj = int(input("Unesi broj: "))
    provjeriBroj(broj)
except NegativanBrojError as e:
    print("Uhvaćena vlastita iznimka:", e)
except Exception as e:
    print("Uhvaćena neka druga iznimka:", e)
```

Izlaz 1:	Unesi broj: 10 Broj je ispravan: 10
Izlaz 2:	Unesi broj: -1 Uhvaćena vlastita iznimka: Greška: -1
Izlaz 3:	Unesi broj: a Uhvaćena neka druga iznimka: invalid literal for int() with base 10: 'a'

4.8. Završne akcije: `else` i `finally`

Blok `else`

Blok `else` opcionalan je i izvršava se kada se unutar bloka `try` ne dogodi iznimka. Drugim riječima, izvršava se samo ako je sav programski kôd unutar bloka `try` uspješno izvršen bez greške. Uporabom bloka `else` omogućuje se odvajanje programskoga kôda koji može izazvati iznimku (smještenoga u blok `try`) od kôda koji se treba izvršiti samo ako do iznimke nije došlo.

Blok `finally`

Blok `finally` opcionalan je i programski se kôd unutar njega uvijek izvršava prije završetka konstrukcije `try/except`, neovisno o tome je li do iznimke došlo ili nije. Blok `finally` najčešće se rabi za oslobađanje

zauzetih resursa (na primjer, zatvaranje datoteka ili drugih tokova podataka).

Primjer 1: Funkcija `kolicnik(a, b)` upotrebljava blokove `try` (za izračun količnika), `except` (za hvatanje iznimke pri dijeljenju s nulom), `else` (za ispis rezultata) i `finally` (za završnu obradu).

```
def kolicnik(a, b):
    try:
        rezultat = a / b
    except ZeroDivisionError:
        print("Dijeljenje s nulom!")
    else:
        print("Rezultat je:", rezultat)
    finally:
        print("Izvršavanje finally bloka!")

brojnik = int(input("Brojnik: "))
nazivnik = int(input("Nazivnik: "))
kolicnik(brojnik, nazivnik)
```

Izlaz 1:

```
Brojnik: 10
Nazivnik: 2
Rezultat je: 5.0
Izvršavanje finally bloka!
```

Izlaz 2:

```
Brojnik: 10
Nazivnik: 0
Dijeljenje s nulom!
Izvršavanje finally bloka!
```

Izlaz 3:

```
Brojnik: 10
Nazivnik: nazivnik
Traceback (most recent call last):
  File "iznimke.py", line 13, in <module>
    nazivnik = int(input("Nazivnik: "))
ValueError: invalid literal for int() with
base 10: 'nazivnik'
```

Izlaz 1 – iznimka se nije dogodila te su izvršeni blokovi: `try`, `else` i `finally`.

Izlaz 2 – zbog dijeljenja s nulom podigla se iznimka te su izvršeni blokovi: `try`, `except` i `finally`.

Izlaz 3 – prilikom učitavanja broja s tipkovnice, zbog unosa koji se ne može pretvoriti u broj podigla se iznimka `ValueError` u glavnome dijelu programa. Budući da taj dio nije unutar bloka `try` (tj. iznimka nije obrađena), program se prekida.

Primjer 2: Primjer zamjene ključne riječi `with` blokom `try/finally`.

```
with open("datoteka.txt", "w") as f:
    f.write("Hello Python!")
```

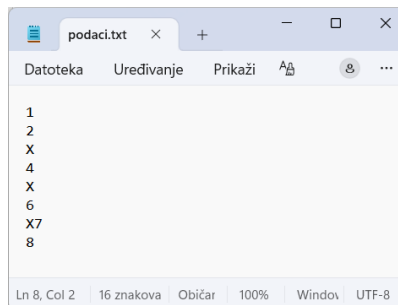
Naredba `with` osigurava da će se nakon izvršavanja pripadajućega programskog kôda automatski osloboditi zauzeti resurs (tok podataka spremljen u objektu imena `f`).

Ista funkcionalnost može se postići pomoću bloka `try/finally`. Unutar bloka `try` otvara se tok podataka te se izvršavaju željene akcije nad datotekom. Nakon što su sve akcije završene, uvijek se izvršava blok `finally` u kojemu se oslobađaju zauzeti resursi odnosno osigurava se zatvaranje toka podataka.

```
f = None
try:
    f = open("datoteka.txt", "w")
    f.write("Hello Python!")
except IOError:
    print("IOError!")
finally:
    if f is not None:
        f.close()
```

Izlaz:
IOError!

Primjer 3: U nastavku je prikazan primjer programa koji čita brojeve (liniju po liniju) iz tekstualne datoteke te izračunava zbroj tako pročitanih brojeva. Datoteka `podaci.txt` sljedećega je sadržaja:



Prva verzija programskoga kôda jest bez obrade iznimaka. Program čita zapise iz tekstualne datoteke te izračunava sumu pročitanih brojeva:

```
zbroj = 0

with open("podaci.txt", "r") as f:
    for linija in f:
        broj = int(linija)
        zbroj += broj
        print(linija, end="")

print("\n\nZbroj: ", zbroj)
```

Izlaz:

```
1
2
Traceback (most recent call last):
  File "C:/iznimke.py", line 5, in <module>
    broj = int(linija)
ValueError: invalid literal for int() with
base 10: 'X\n'
```

Kao što je moguće vidjeti, došlo je do iznimke. Iznimka se podigla zato što se u datoteci nalaze linije koje ne sadrže samo brojeve, već i druge znakove prilikom čijega se pretvaranja u broj pomoću funkcije `int()` podiže iznimka `ValueError`.

Primjer 4: Kako bi se izračunao zbroj svih ispravnih brojeva iz datoteke, neovisno sadržava li neka linija samo brojeve ili pak i nedopuštene znakove (linije s nedopuštenim znakovima ignoriraju se), potrebno je napraviti obradu iznimaka. U nastavku slijedi primjer programskoga kôda koji to omogućuje.

```
zbroj = 0
brojGresaka = 0

with open("podaci.txt", "r") as f:
    for linija in f:
        try:
            broj = int(linija)
        except ValueError:
            brojGresaka += 1
        else:
            zbroj += broj
        finally:
            print(linija, end="")

print("\n\nZbroj: ", zbroj)
print("Broj grešaka: ", brojGresaka)
```

Izlaz:

```
1
2
X
4
X
6
X7
8
```

Zbroj: 21

Broj grešaka: 3

4.9. Vježba: Iznimke II

1. Napišite funkciju imena `ucitajPaZbroji()`, koja učitava 3 broja s tipkovnice i ispisuje njihov zbroj. U toj funkciji uhvatite potencijalne iznimke te tako uhvaćene iznimke proslijedite u glavni dio programa (dio programa unutar kojega je funkcija pozvana) te tamo ispišite da se dogodila iznimka.
2. Prethodni zadatak nadogradite tako da rezultat zbroja ispisujete unutar bloka `else` funkcije imena `ucitajPaZbroji()`. Također, napišite i blok `finally` koji će ispisivati generičku poruku poput: "Finally blok!".
3. Napišite funkciju `provjeriOcjenu()` koja prima jedan cijeli broj koji predstavlja ocjenu. Funkcija treba provjeriti je li unesena vrijednost dopuštena ocjena, pri čemu su dopuštene ocjene 1, 2, 3, 4 i 5. Ako unesena ocjena nije jedna od ovih vrijednosti, funkcija treba podići vlastitu iznimku `NeispravnaOcjenaError`. U slučaju da je unesena ocjena ispravna, funkcija treba ispisati poruku da je ocjena ispravna. U glavnome programu omogućite unos ocjena sve dok korisnik ne unese vrijednost koja nije cijeli broj. Na kraju program treba ispisati ukupan broj ispravnih ocjena i ukupan broj neispravnih ocjena.

4.10. Pitanja za ponavljanje: Iznimke

1. U koja je tri skupa moguće podijeliti greške unutar programskoga kôda?
2. Koji razred nasljeđuju sve iznimke u Pythonu?
3. Koji se tip iznimke podiže u slučaju da korištena varijabla nije definirana?
4. Koje dvije ključne riječi tvore osnovni blok za obradu iznimaka?
5. Koje dvije dodatne ključne riječi tvore prošireni blok iznimaka?
6. Za što služi blok `except`?
7. Kada se izvodi blok `else`?
8. Kada se izvodi blok `finally`?

5. Algoritmi sortiranja

Trajanje
poglavlja:
60 min

Po završetku ovoga poglavlja polaznik će moći:

- objasniti što su to algoritmi sortiranja
- navesti najpopularnije algoritme sortiranja i njihovu složenost
- objasniti kada se rabi ugrađena funkcija `sort()`, a kada je potrebno implementirati algoritam za sortiranje podataka
- implementirati algoritme: sortiranje biranjem i sortiranje zamjenom susjednih elemenata.

Algoritmi sortiranja omogućuju sortiranje elemenata uzlaznim ili silaznim redoslijedom. Postoji mnogo različitih algoritama razvijenih u svrhu sortiranja podataka, a u ovome poglavlju opisana je implementacija sljedećih algoritama: sortiranje biranjem, sortiranje zamjenom susjednih elemenata te poboljšani algoritam sortiranja zamjenom susjednih elemenata. U nastavku su navedeni neki od najčešće korištenih algoritama sortiranja. S desne strane svakoga algoritma navedena je njegova vremenska složenost:

- sortiranje biranjem (engl. *selection sort*) – $O(n^2)$
- sortiranje zamjenom susjednih elemenata (engl. *bubble sort*) – $O(n^2)$
- sortiranje zamjenom elemenata (engl. *exchange sort*) – $O(n^2)$
- sortiranje umetanjem (engl. *insertion sort*) – $O(n^2)$
- sortiranje sjedinjavanjem (engl. *merge sort*) – $O(n \log n)$
- brzo sortiranje po Hoareu (engl. *quick sort*) – $O(n \log n)$
- sortiranje razvrstavanjem u pretince (engl. *bucket sort*) – $O(n)$

Svaki od navedenih algoritama temelji se na drugačijem principu rada (različitoj implementaciji) te ima različitu vremensku složenost, odnosno brzinu izvođenja. Razvijen je velik broj algoritama za sortiranje podataka, no teško je izdvojiti algoritam koji bi bio najbolji u svim situacijama. Svaki algoritam ima svoje prednosti i mane, pa se ovisno o problemu odabire najprikladniji.

U Pythonu postoji ugrađena metoda `sort()` koja služi za sortiranje podataka. Ova ugrađena metoda izrazito efikasno sortira elemente te je preporučljivo koristiti se njome kad god je to moguće. Algoritmi za sortiranje obično se pišu u situacijama kada se nije moguće koristiti ugrađenom metodom za sortiranje. Na primjer, ako postoje dvije liste, u jednoj listi nalaze se imena osoba, a u drugoj listi nalaze se rezultati natjecanja (podaci su povezani preko zajedničkoga indeksa). Ako je potrebno sortirati imena osoba uzlazno ovisno o ostvarenim bodovima, onda se nije moguće koristiti ugrađenom metodom `sort()`, već je

potrebno ručno implementirati algoritam koji će istovremeno sortirati obje liste.

Na poveznici: <https://www.toptal.com/developers/sorting-algorithms> moguće je vizualno usporediti brzinu sortiranja pojedinih algoritama.

Složenost algoritma

Složenost algoritma opisana je vremenom potrebnim za njegovo izvršavanje.

- $O(1)$ – vrijeme izvršavanja ne ovisi o broju elemenata u kolekciji
- $O(n)$ – vrijeme izvršavanja linearno ovisi o broju elemenata
- $O(n^2)$ – vrijeme izvršavanja proporcionalno je kvadratu broja elemenata
- neke od ostalih složenosti: $O(n \log n)$, $O(n!)$, $O(n^c)$...

Objašnjenje složenosti na konkretnim primjerima

Neka se za primjer uzme lista koju je potrebno sortirati algoritmima složenosti: $O(1)$, $O(n)$, $O(n \log n)$, $O(n^2)$. U listi se nalazi 10^6 elemenata (1.000.000), a uzeta je u obzir pretpostavka da računalo u sekundi može obraditi 10^9 instrukcija (1.000.000.000). U nastavku se nalazi matematički izračun vremena potrebnog za izvođenje operacije sortiranja liste, ovisno o složenosti odabranoga algoritma.

- $O(1) = \frac{1}{1.000.000.000} = 0,000000001 \text{ sec}$
- $O(n) = \frac{1.000.000}{1.000.000.000} = 0,001 \text{ sec}$
- $O(n * \log n) = \frac{1.000.000 \log 1.000.000}{1.000.000.000} = \frac{\log 1.000.000}{1.000} = 0,0138 \text{ sec}$
- $O(n^2) = \frac{1.000.000 \times 1.000.000}{1.000.000.000} = \frac{1.000}{1} = 1.000 \text{ sec}$

5.1. Sortiranje biranjem (engl. *selection sort*)

Ovaj algoritam temelji se na traženju vrijednosti, od najmanje prema najvećoj vrijednosti ili pak od najveće prema najmanjoj vrijednosti. U daljnjem tekstu svi algoritmi opisivat će sortiranje elemenata koji su u listi, uzlaznim redoslijedom, tj. od najmanje prema najvećoj vrijednosti.

Na početku ovoga algoritma pronalazi se najmanja vrijednost, koja se potom stavlja na prvo mjesto, nakon toga traži se sljedeća najmanja vrijednost koja se stavlja na drugo mjesto i tako dalje sve do trenutka kada niz bude sortiran.

Listu nad kojom ovaj algoritam radi moguće je podijeliti u dva dijela:

- Unutar prvoga dijela liste nalaze se vrijednosti koje su sortirane uzlaznim redoslijedom (ili silaznim). U vizualnome prikazu ovoga algoritma ovaj dio liste prikazan je zelenom bojom.

- Drugi dio liste sadrži elemente koji nisu sortirani. Unutar ovoga dijela liste traži se sljedeći element koji će se staviti na početak nesortiranoga dijela liste. U vizualnome prikazu ovoga algoritma ovaj dio liste prikazan je narančastom bojom.

U nastavku slijedi vizualni prikaz funkcioniranja ovoga algoritma.

7	4	2	8	6	<u>1</u>	3	5
1	4	<u>2</u>	8	6	7	3	5
1	2	4	8	6	7	<u>3</u>	5
1	2	3	8	6	7	<u>4</u>	5
1	2	3	4	6	7	8	<u>5</u>
1	2	3	4	5	7	8	<u>6</u>
1	2	3	4	5	6	8	<u>7</u>
1	2	3	4	5	6	7	<u>8</u>
1	2	3	4	5	6	7	8

Koraci algoritma:

- U nesortiranome dijelu liste (narančasti dio) traži se najmanja (ili najveća) vrijednost.
- Pronađena najmanja (ili najveća) vrijednost mijenja se prvim članom nesortiranoga dijela liste.
- Prethodna dva koraka ponavljaju se sve dok cijela lista ne bude sortirana.

Primjer 1: Algoritam "sortiranje biranjem" (uzlazno sortiranje).

```
niz = [7, 4, 2, 8, 6, 1, 3, 5]
print("Nesortiran niz:", niz)
brojElemenata = len(niz)

for i in range(brojElemenata):

    minIndex = i
    for j in range(i + 1, brojElemenata):
        if niz[j] < niz[minIndex]:
            minIndex = j

    niz[i], niz[minIndex] = niz[minIndex], niz[i]

    print("Iteracija:", i + 1, niz)

print("Sortiran niz:", niz)
```

Izlaz:

```
Nesortiran niz: [7, 4, 2, 8, 6, 1, 3, 5]
Iteracija: 1 [1, 4, 2, 8, 6, 7, 3, 5]
Iteracija: 2 [1, 2, 4, 8, 6, 7, 3, 5]
Iteracija: 3 [1, 2, 3, 8, 6, 7, 4, 5]
Iteracija: 4 [1, 2, 3, 4, 6, 7, 8, 5]
Iteracija: 5 [1, 2, 3, 4, 5, 7, 8, 6]
Iteracija: 6 [1, 2, 3, 4, 5, 6, 8, 7]
Iteracija: 7 [1, 2, 3, 4, 5, 6, 7, 8]
Iteracija: 8 [1, 2, 3, 4, 5, 6, 7, 8]
Sortiran niz: [1, 2, 3, 4, 5, 6, 7, 8]
```

5.2. Vježba: Sortiranje biranjem

```
sudionici = ["Ivica", "Marica", "Perica", "Nika",
"Nikica", "Tihana", "Stjepan", "Petra"]

rezultati = [10.11, 11.58, 10.08, 10.06, 10.77,
11.22, 11.55, 10.05]
```

Pogledajte gore navedene dvije liste utrke na 100 metara, `sudionici` i `rezultati`. Vaš je zadatak da ih sortirate i ispišete uzlazno prema rezultatima (od najbržega do najsporijega).

1. Sortiranje napravite pomoću algoritma "sortiranje biranjem" (engl. *selection sort*).

5.3. Sortiranje zamjenom susjednih elemenata (engl. *bubble sort*)

Još jedan od algoritama sortiranja jest "sortiranje zamjenom susjednih elemenata". Ovaj algoritam prolazi kroz listu te uspoređuje parove vrijednosti, odnosno susjedne elemente. U slučaju da su ti elementi (susjedi) u neispravnome poretku, njihove se vrijednosti mijenjaju. Algoritam kroz nesortirani dio liste prolazi sve dok cijela lista ne bude sortirana.

Listu nad kojom ovaj algoritam radi moguće je podijeliti u dva dijela:

- Unutar prvoga dijela liste nalaze se vrijednosti koje nisu sortirane. U vizualnome prikazu ovoga algoritma ovaj dio liste prikazan je narančastom bojom. Vrijednosti koje se uspoređuju i eventualno mijenjaju (u slučaju neispravnoga poretka) označene su sivom bojom.
- Drugi dio liste sadrži elemente koji su sortirani. U vizualnome prikazu ovoga algoritma ovaj dio liste prikazan je zelenom bojom.

Ovaj algoritam ima složenost $O(n^2)$.

Koraci algoritma:

- Algoritam prolazi kroz nesortirani dio liste (narančasti dio).

- Ako dvije susjedne vrijednosti koje se uspoređuju nisu u ispravnome poretku (ovisno o uzlaznome ili silaznome sortiranju), njihove se vrijednosti zamjenjuju. Nakon zamjene algoritam nastavlja sa sljedećim parom elemenata. Kada algoritam dođe do kraja liste, lista još nije u cijelosti sortirana, nego je na njezino zadnje mjesto dovedena najveća vrijednost (pri uzlaznome sortiranju) odnosno najmanja vrijednost (pri silaznome sortiranju).
- Nakon što algoritam završi jedan prolaz kroz nesortirani dio liste, u sljedećemu prolazu ponovo kreće od početka nesortiranoga dijela liste.
- Ovaj se postupak ponavlja sve dok lista ne bude sortirana.

1. prolaz (7 usporedbi)

7	4	2	8	6	1	3	5
4	7	2	8	6	1	3	5
4	2	7	8	6	1	3	5
4	2	7	8	6	1	3	5
4	2	7	6	8	1	3	5
4	2	7	6	1	8	3	5
4	2	7	6	1	3	8	5
4	2	7	6	1	3	5	8

2. prolaz (6 usporedbi)

4	7	2	6	1	3	5	8
4	7	2	6	1	3	5	8
4	2	7	6	1	3	5	8
4	2	6	7	1	3	5	8
4	2	6	1	7	3	5	8
4	2	6	1	3	7	5	8
4	2	6	1	3	5	7	8

3. prolaz (5 usporedbi)

4	2	6	1	3	5	7	8
2	4	6	1	3	5	7	8
2	4	6	1	3	5	7	8
2	4	1	6	3	5	7	8
2	4	1	3	6	5	7	8
2	4	1	3	5	6	7	8

4. prolaz (4 usporedbe)

2	4	1	3	5	6	7	8
2	4	1	3	5	6	7	8
2	1	4	3	5	6	7	8
2	1	3	4	5	6	7	8
2	1	3	4	5	6	7	8

5. prolaz (3 usporedbe)

2	1	3	4	5	6	7	8
1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8

6. prolaz (2 usporedbe)

1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8

7. prolaz (1 usporedba)

1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8

Primjer 1: Algoritam "sortiranje zamjenom susjednih elemenata" (uzlazno sortiranje).

```
niz = [7, 4, 2, 8, 6, 1, 3, 5]

print("Nesortiran niz:", niz)
brojElemenata = len(niz)

for i in range(brojElemenata - 1):
    for j in range(brojElemenata - 1 - i):
        if niz[j + 1] < niz[j]:
            niz[j], niz[j + 1] = niz[j + 1], niz[j]

    print("Iteracija:", i + 1, niz)

print("Sortiran niz:", niz)
```

Izlaz:

```
Nesortiran niz: [7, 4, 2, 8, 6, 1, 3, 5]
Iteracija: 1 [4, 2, 7, 6, 1, 3, 5, 8]
Iteracija: 2 [2, 4, 6, 1, 3, 5, 7, 8]
Iteracija: 3 [2, 4, 1, 3, 5, 6, 7, 8]
Iteracija: 4 [2, 1, 3, 4, 5, 6, 7, 8]
Iteracija: 5 [1, 2, 3, 4, 5, 6, 7, 8]
Iteracija: 6 [1, 2, 3, 4, 5, 6, 7, 8]
Iteracija: 7 [1, 2, 3, 4, 5, 6, 7, 8]
Sortiran niz: [1, 2, 3, 4, 5, 6, 7, 8]
```

5.4. Poboljšani algoritam zamjene susjednih elemenata

Prethodno objašnjeni algoritam moguće je poboljšati. Ako se pogleda vizualni prikaz algoritma "sortiranje zamjenom susjednih elemenata", vidljivo je da se u posljednjim dvama koracima nije dogodila nijedna zamjena (6. i 7. prolaz). Algoritam zamjene susjednih elemenata moguće je poboljšati tako da se uvede dodatna varijabla (u narednome primjeru programskoga kôda dodatna varijabla nazvana je imenom *zamjena*). U tu varijablu bit će zapisano je li se unutar nekoga prolaza dogodila zamjena ili ne. Ako unutar nekoga prolaza po nesortiranome dijelu niza nema zamjene, niz se proglašava sortiranim te se pomoću naredbe `break` prekida daljnje uspoređivanje susjednih elemenata, odnosno sljedeći prolazak po nizu.

Ako se pogleda vizualni prikaz funkcioniranja algoritma koji je prikazan u prethodnome poglavlju, u ovome poboljšanom algoritmu taj bi prikaz izgledao ovako:

- Od 1. do 5. prolaza sve ostaje identično.
- U 6. prolazu nema zamjene susjednih elemenata te se preddefinirana vrijednost varijable *zamjena* ne postavlja na `True`. Na kraju ovoga prolaza naredba `break` prekida izvođenje petlje, čime se prekida naredni prolaz kroz listu.

- 7. prolaz neće se dogoditi.

Primjer 1: Prikaz programskoga kôda poboljšanoga algoritma.

```
niz = [7, 4, 2, 8, 6, 1, 3, 5]
print("Nesortiran niz:", niz)
brojElemenata = len(niz)

for i in range(brojElemenata - 1):
    zamjena = False
    for j in range(brojElemenata - 1 - i):
        if niz[j + 1] < niz[j]:
            niz[j], niz[j + 1] = niz[j + 1], niz[j]
            zamjena = True

    print("Prolaz:", i + 1, niz)

    if zamjena == False:
        break

print("Sortiran niz:", niz)
```

Izlaz:

```
Nesortiran niz: [7, 4, 2, 8, 6, 1, 3, 5]
Prolaz: 1 [4, 2, 7, 6, 1, 3, 5, 8]
Prolaz: 2 [2, 4, 6, 1, 3, 5, 7, 8]
Prolaz: 3 [2, 4, 1, 3, 5, 6, 7, 8]
Prolaz: 4 [2, 1, 3, 4, 5, 6, 7, 8]
Prolaz: 5 [1, 2, 3, 4, 5, 6, 7, 8]
Prolaz: 6 [1, 2, 3, 4, 5, 6, 7, 8]
Sortiran niz: [1, 2, 3, 4, 5, 6, 7, 8]
```

5.5. Usporedba funkcije `sorted()` i metode `sort()`

Funkcija `sorted()`

- Ugrađena funkcija.
- Radi za bilo koji *iterable* (na primjer: niz znakova, lista, n-torka, rječnik (sortira se po ključevima), generator, iterator).
- Vraća novu sortiranu listu (nevezano za to koji je tip objekta predan).
- Originalni objekt ostaje netaknut.

Metoda `sort()`

- Ugrađena metoda liste.
- Radi isključivo za liste.
- Vraća `None`.
- Sortira listu na mjestu (engl. *in-place*), tj. lista se mijenja.
- Nakon poziva ove metode originalni se redoslijed elemenata liste gubi.

Ako se želi sortirati listu, metoda `sort()` brža je od funkcije `sorted()` jer sortira listu na mjestu, odnosno ne mora kreirati novu listu. Za bilo koji drugi *iterable* potrebno se koristiti funkcijom `sorted()`, koja će uvijek vratiti novu sortiranu listu, bez obzira na tip objekta koji joj je predan.

Primjer 1: Funkcija `sorted()`.

```
lista = [6, 4, 8, 5, 10, 7, 3, 1, 9]
print("lista:", lista, "\n")

sortiranaLista = sorted(lista)

print("Liste nakon korištenja funkcije sorted():")
print("lista:", lista)
print("sortiranaLista:", sortiranaLista)
```

Izlaz:

```
lista: [6, 4, 8, 5, 10, 7, 3, 1, 9]

Liste nakon korištenja funkcije sorted():
lista: [6, 4, 8, 5, 10, 7, 3, 1, 9]
sortiranaLista: [1, 3, 4, 5, 6, 7, 8, 9, 10]
```

Primjer 2: Metoda `sort()`.

```
lista = [6, 4, 8, 5, 10, 7, 3, 1, 9]
print("lista:", lista, "\n")

lista.sort()

print("Lista nakon korištenja metode sort():")
print("lista:", lista)
```

Izlaz:

```
lista: [6, 4, 8, 5, 10, 7, 3, 1, 9]

Lista nakon korištenja metode sort():
lista: [1, 3, 4, 5, 6, 7, 8, 9, 10]
```

Funkcija `sorted()` i metoda `sort()` za sortiranje rabe algoritam Timsort. Ovaj je algoritam kreirao Tim Peters 2002. godine, upravo za uporabu u programskome jeziku Python. Timsort je mješavina dvaju algoritama: sortiranja sjedinjavanjem (engl. *merge sort*) i sortiranja umetanjem (engl. *insertion sort*). Njegova je složenost $O(n \log n)$.

5.6. Vježba: Sortiranje zamjenom susjednih elemenata

```
sudionici = ["Ivica", "Marica", "Perica", "Nika",  
            "Nikica", "Tihana", "Stjepan", "Petra"]  
  
rezultati = [10.11, 11.58, 10.08, 10.06, 10.77,  
            11.22, 11.55, 10.05]
```

Pogledajte prethodne dvije liste utrke na 100 metara, `sudionici` i `rezultati`. Vaš je zadatak da ih sortirate i ispišete uzlazno prema rezultatima (od najbržega do najsporijega).

1. Sortiranje napravite pomoću poboljšanoga algoritma zamjene susjednih elemenata (engl. *bubble sort*).

5.7. Pitanja za ponavljanje: Algoritmi sortiranja

1. Navedite primjer kada se nije moguće koristiti ugrađenom metodom `sort()`, već je potrebno implementirati algoritam za sortiranje podataka.
2. Kako funkcionira algoritam za sortiranje imena "sortiranje biranjem"?
3. Kako funkcionira algoritam za sortiranje imena "sortiranje zamjenom susjednih elemenata"?
4. Kako se može poboljšati algoritam za sortiranje imena "sortiranje zamjenom susjednih elemenata"?

6. Moduli i paketi

Po završetku ovoga poglavlja polaznik će moći:

- *pretraživati i instalirati pakete trećih strana*
- *razvijati i uključivati u programski kôd vlastite module.*

**Trajanje
poglavlja:**
30 min

Moduli i paketi omogućuju lakši i brži razvoj programskoga kôda. Python pakete možemo podijeliti u dvije osnovne skupine, a to su paketi koji dolaze sa standardnom instalacijom Pythona (standardnom bibliotekom) te paketi koje razvijaju drugi programeri i oni se instaliraju prema potrebi.

Važno je razlikovati module i pakete. Moduli su `.py` datoteke unutar kojih je implementirana određena funkcionalnost, a takve datoteke sadrže funkcije, klase i varijable. Uzmimo za primjer nekoliko modula koji dolaze sa standardnom bibliotekom: `math`, `cmath`, `random`, `datetime`. Paketi su organizacijske cjeline (mape) koje sadrže više modula i omogućuju njihovo hijerarhijsko grupiranje. Primjer paketa koji dolazi sa standardnom bibliotekom jest `urllib`, koji se sastoji od više modula:

- `urllib.request`
- `urllib.error`
- `urllib.parse`
- `urllib.robotparser`

U ovome poglavlju obradit će se osnovni načini instalacije Python paketa trećih strana (onih koji nisu dio standardne biblioteke), kao i postupak izrade vlastitih modula te njihovo uključivanje u programski kôd.

6.1. Instalacija paketa trećih strana

Prije instalacije dodatnih paketa potrebno je provjeriti jesu li zadovoljeni svi preduvjeti za njihovu nesmetanu instalaciju. Naredbe `python` i `pip` moraju biti dodane u PATH varijable okoline (engl. *environment variables*). Time se omogućuje njihovo pokretanje iz naredbenoga retka.

```

Microsoft Windows [Version 10.0.26200.8037]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Marko>python --version
Python nije prona
C:\Users\Marko>pip --version
'pip' is not recognized as an internal or external command,
operable program or batch file.

C:\Users\Marko>

```

```
python --version
pip --version
```

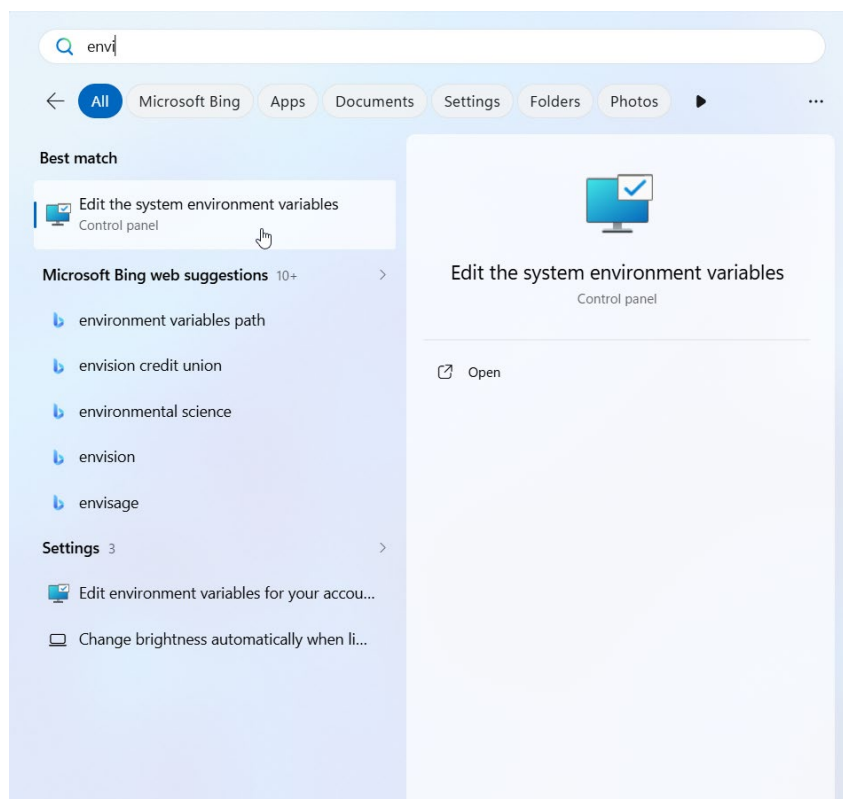
Pregledom prethodne slike moguće je utvrditi jesu li naredbe `python` i `pip` u PATH varijabli okoline. U gornjem slučaju nisu te ih je potrebno uključiti. U slučaju da jesu (ako se prikažu verzije instaliranih programa), potpoglavlje "6.1.1. Postavljanje PATH varijabli okoline" moguće je preskočiti.

PIP (engl. *Python Package Installer*) paketni je upravitelj za instalaciju i upravljanje Python paketima. Od verzije Python 3.4 i novijima, PIP dolazi uključen u standardnu instalaciju Pythona. Ako se upotrebljava starija verzija Pythona, onda je PIP potrebno instalirati. U tome slučaju upute za instalaciju dostupne su u službenoj dokumentaciji Pythona 3.

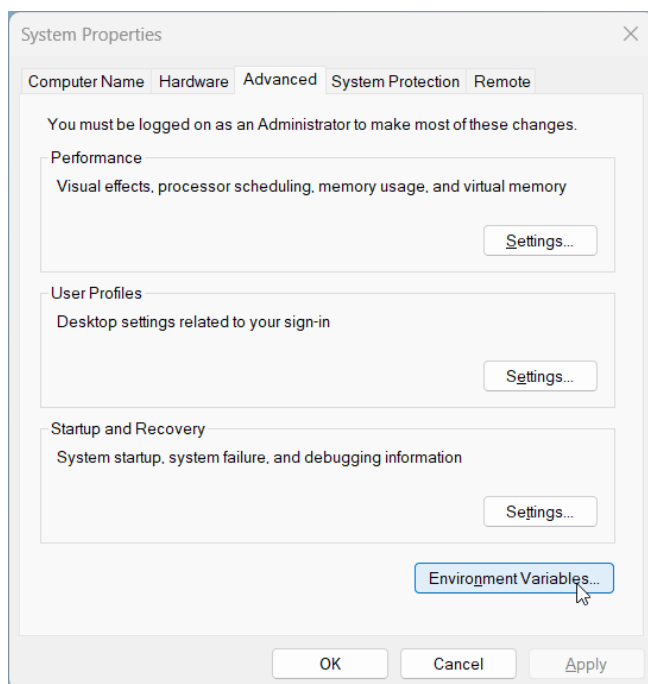
6.1.1. Postavljanje PATH varijabli okoline

Kako bi se omogućilo pokretanje ovih naredbi iz naredbenoga retka, potrebno ih je uključiti. Preduvjet za uključivanje ovih naredbi u PATH varijable okoline jest da je na računalu ili serveru instaliran Python. Nakon što je ovaj uvjet zadovoljen, moguće je uključiti `python` i `pip` u PATH varijablu okoline. U nastavku je opisano postavljanje PATH varijabli okoline na operativnome sustavu Microsoft Windows 11.

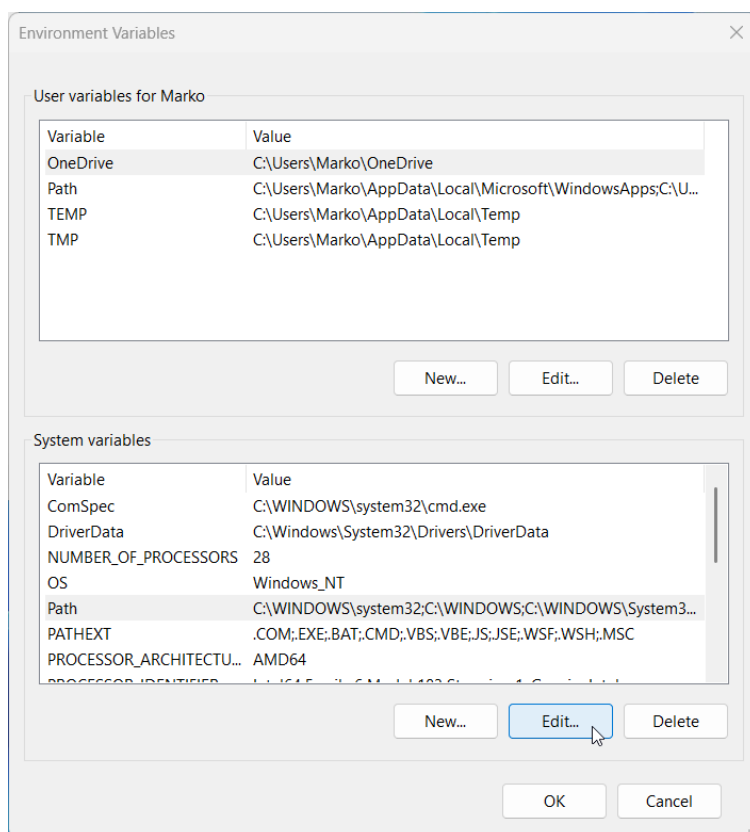
Slika 1: Na operativnome sustavu Windows 11 potrebno je otvoriti izbornik Start te u tražilicu upisati "Edit the system environment variables", nakon čega se odabire odgovarajuća opcija iz rezultata pretraživanja.



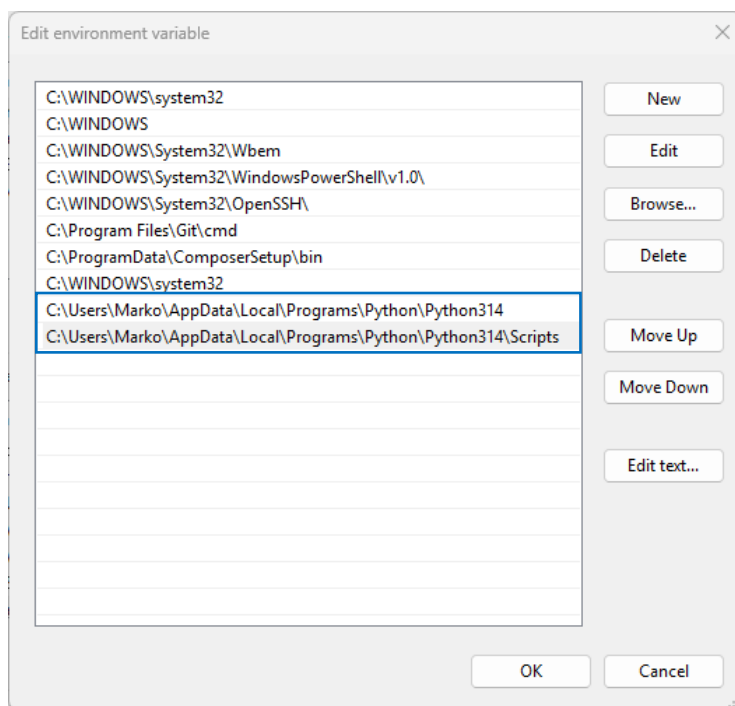
Slika 2: U kartici *Advanced* potrebno je odabrati opciju *Environment Variables*.



Slika 3: U sljedećemu prozoru prikazan je popis svih varijabli okoline. Unutar odjeljka *System variables* potrebno je pronaći stavku *PATH*, označiti ju te kliknuti na gumb *Edit*.



Slika 4: U otvorenom prozoru potrebno je kliknuti na *New*, nakon čega se dodaju željene putanje.



Za pokretanje naredbe `python` iz naredbenoga retka potrebno je dodati putanju:

`C:\Users\User\AppData\Local\Programs\Python\Python314`

Za pokretanje naredbe `pip` iz naredbenoga retka potrebno je dodati putanju:

`C:\Users\User\AppData\Local\Programs\Python\Python314\Scripts`

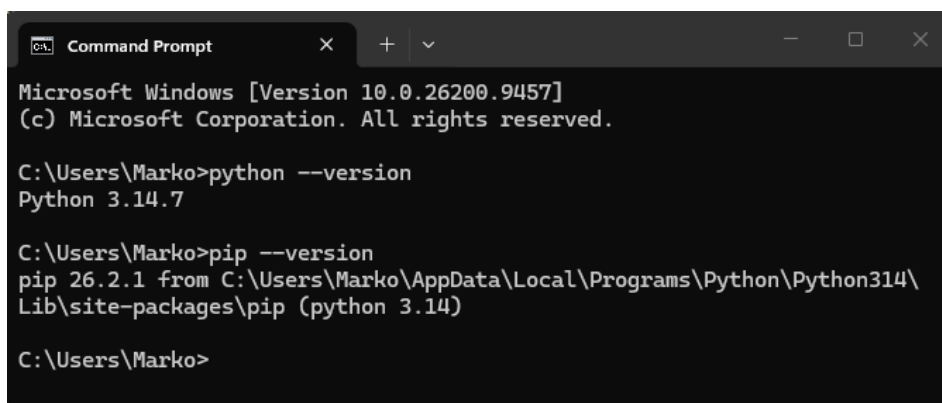
Napomena: Putanje do instalacije Pythona mogu se razlikovati ovisno o verziji Pythona (**Python314**) i korisničkome računu na operativnome sustavu Windows 11 (**User**).

Nakon što su gore opisane postavke postavljene, moguće je testirati ispravnost PATH varijabli okoline.

Nakon što su navedene postavke postavljene, potrebno je provjeriti ispravnost PATH varijable okoline pokretanjem naredbi sljedećih naredbi u naredbenome retku:

```
python --version
pip --version
```

Slika 5: Ako je sve ispravno postavljeno, prikazat će se verzije Pythona i PIP-a.



```

Microsoft Windows [Version 10.0.26200.9457]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Marko>python --version
Python 3.14.7

C:\Users\Marko>pip --version
pip 26.2.1 from C:\Users\Marko\AppData\Local\Programs\Python\Python314\
Lib\site-packages\pip (python 3.14)

C:\Users\Marko>

```

6.1.2. Instalacija paketa iz PyPI-a

PIP se najčešće upotrebljava za instalaciju paketa iz PyPI-a (engl. *Python Package Index*), a moguće ga je rabiti i za instalaciju paketa koji se nalaze na nekim drugim sjedištima. U ovome priručniku obrađuje se instalacija paketa dostupnih na PyPI-u. PyPI je repozitorij Python paketa koji omogućuje njihovo jednostavno pretraživanje, preuzimanje i instalaciju. Pakete dostupne na PyPI-u razvila je i objavila Python zajednica.

PyPI je dostupan na adresi: <https://pypi.org/>.

U nastavku slijede primjeri instalacije paketa pomoću alata `pip`. Radi jednostavnosti rabi se primjer naziva paketa "ImePaketa".

- Instalacija najnovije verzije paketa:

```
pip install "ImePaketa"
```

- Instalacija točno određene verzije paketa:

```
pip install "ImePaketa==1.4"
```

- Instalacija paketa verzije unutar određenoga raspona (u donjem primjeru veće ili jednake od 1, a manje od 2):

```
pip install "ImePaketa>=1,<2"
```

- Instalacija paketa kompatibilne verzije (u donjem primjeru instalacija bilo koje verzije ">=1.4.*", a da je ta verzija ">=1.4.2"):

```
pip install "ImePaketa~>=1.4.2"
```

6.1.3. Nadogradnja postojećega paketa

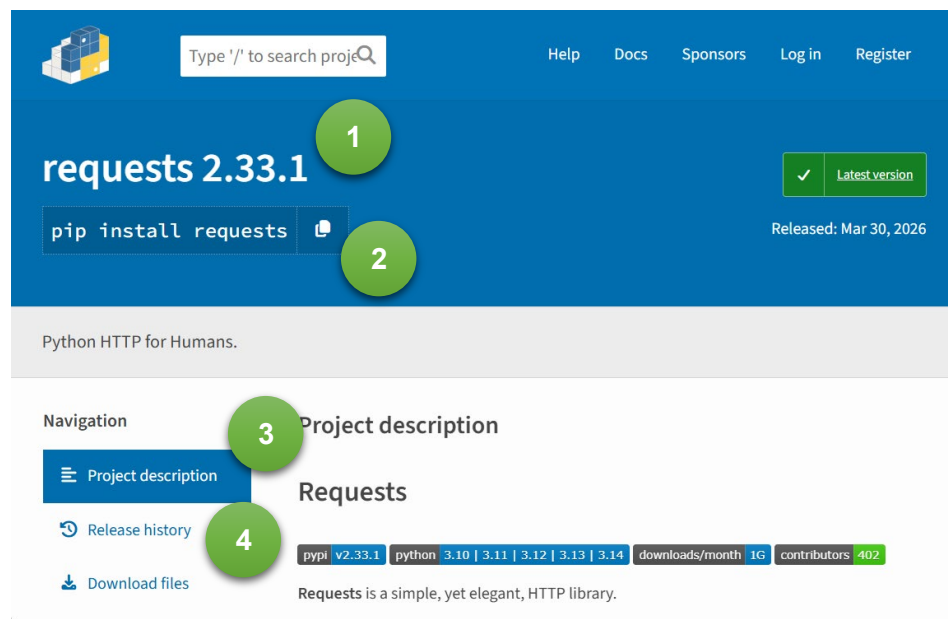
U slučaju kada je određeni paket već instaliran i kada se želi taj isti paket nadograditi na posljednju postojeću verziju, rabi se sljedeća naredba:

```
pip install --upgrade ImePaketa
```

1.1.4. Primjer instalacije paketa – requests

Slika 1: U nastavku je prikazana slika PyPI *web*-stranice na kojoj je otvoren paket `requests` (za edukativne potrebe odabran nasumično). Opis najvažnijih dijelova stranice:

1. naziv i verzija odabranoga paketa
2. naredba za instalaciju – omogućuje kopiranje naredbe za instalaciju odabranoga paketa
3. opis funkcionalnosti paketa – daje kratak pregled čemu paket služi i koje funkcionalnosti nudi
4. pregled svih dostupnih verzija – prikazuje sve verzije paketa koje je moguće instalirati.



Primjer 1: Primjer u nastavku sadrži samo jednu liniju kôda koja najavljuje korištenje funkcijom iz paketa `requests`. Ovaj programski kôd pokrenut je prije instalacije paketa i, kao što se može vidjeti, pojavila se greška da paket imena `requests` nije moguće pronaći.

```
import requests
Izlaz:
Traceback (most recent call last):
  File "moduli.py", line 1, in <module>
    import requests
ModuleNotFoundError: No module named
'requests'
```

Na PyPI-u se obično ne nalazi detaljna dokumentacija paketa, već se obično u opisu paketa nalazi poveznica na službeni *web*-stranicu gdje je moguće pronaći dokumentaciju i sve potrebne informacije. Za paket `requests` dokumentacija je dostupna na sljedećoj poveznici:

<https://requests.readthedocs.io/en/latest/>.

Instalacija paketa izvršava se u naredbenome retku pomoću sljedeće naredbe:

```
pip install requests
```

Primjer 2: Nakon što je paket `requests` instaliran, moguće je pokrenuti programski kôd koji se koristi implementacijama iz toga paketa bez pojave grešaka.

```
import requests

# Slanje GET zahtjeva
response = requests.get("https://www.srce.unizg.hr/")

# Provjera statusnog koda
print("Statusni kod:", response.status_code)

# Provjera sadržaja web-stranice
print("Sveučilišni računski centar" in response.text)

Izlaz:
    Statusni kod: 200
    True
```

6.2. Kreiranje i uključivanje vlastitih modula

Do sada su u ovome tečaju upotrebljavani moduli koji dolaze sa standardnom instalacijom Pythona ili moduli preuzeti s interneta i naknadno instalirani na računalo ili poslužitelj. U ovome poglavlju objašnjava se način izrade i uključivanja vlastitih modula u programski kôd.

Prije uključivanja vlastitoga modula u programski kôd potrebno je taj modul najprije napisati. Modul se u Pythonu definira kao `.py` datoteka koja sadrži funkcije, klase ili varijable.

Primjer 1: U nastavku se nalazi primjer programskoga kôda u kojemu su implementirane funkcije za izračun rezultata zbroja, razlike, umnoška i količnika dvaju broja. Ovaj programski kôd spremljen je u datoteku naziva `matematickeOperacije.py`, koja se nalazi na sljedećoj putanji: `D:\D460\Moduli`.

```
def zbroj(a, b):
    """Funkcija vraća zbroj dvaju brojeva."""
    return a + b

def razlika(a, b):
    """Funkcija vraća razliku dvaju brojeva."""
    return a - b

def umnozak(a, b):
    """Funkcija vraća umnožak dvaju brojeva."""
    return a * b

def kolicnik(a, b):
    """Funkcija vraća količnik dvaju brojeva."""
    return a / b
```

Primjer 2: Kako bi funkcije implementirane u modulu `matematickeOperacije` bile dostupne u drugome programskom kôdu, potrebno je najaviti njihovu uporabu ili uključiti modul u programski kôd. Prilikom najavljivanja ili uključivanja modula izostavlja se sufiks `.py`.

```
import matematickeOperacije
```

Izlaz:

```
Traceback (most recent call last):
  File "D:\D460\Skripte\skripta.py", line 1,
    in <module>
      import matematickeOperacije
ModuleNotFoundError: No module named
'matematickeOperacije'
```

Primjer programskoga kôda koji je prethodno prikazan spremljen je u datoteku naziva `skripta.py`, koja se nalazi na sljedećoj putanji: `D:\D460\Skripte`. Kao što je vidljivo iz prethodnoga primjera, prilikom pokretanja programa dogodila se greška iz koje je moguće iščitati da Python ne može pronaći modul imena `matematickeOperacije`. Razlog je tome što Python ne zna u kojemu se direktoriju traženi modul nalazi.

Primjer 3: Pozivom naredbe `sys.path` iz modula `sys` moguće je dobiti popis svih direktorija u kojima Python traži module. Iz donjega popisa vidimo da Python ima pristup različitim direktorijima, uključujući i direktorij u kojemu se nalazi pokrenuta skripta. Međutim, nema pristup direktoriju u kojemu se nalazi modul `matematickeOperacije` i zato Python ne može pronaći navedeni modul.

```
import sys
print(sys.path)
```

Izlaz:

```
['D:/D460/Skripte',
 'C:\\Users\\Marko\\AppData\\Local\\Programs\\
Python\\Python313\\Lib\\idlelib',
 'C:\\Users\\Marko\\AppData\\Local\\Programs\\
Python\\Python313\\python313.zip',
 'C:\\Users\\Marko\\AppData\\Local\\Programs\\
Python\\Python313\\DLLs',
 'C:\\Users\\Marko\\AppData\\Local\\Programs\\
Python\\Python313\\Lib',
 'C:\\Users\\Marko\\AppData\\Local\\Programs\\
Python\\Python313',
 'C:\\Users\\Marko\\AppData\\Local\\Programs\\
Python\\Python313\\Lib\\site-packages']
```

Ako želimo da Python ima pristup direktoriju u kojemu se nalazi naš modul (u slučaju kada se taj modul ne nalazi u istome direktoriju kao skripta koja se pokreće), potrebno je putanju do toga direktorija dodati u popis direktorija koje Python pretražuje prilikom učitavanja modula.

Sintaksa za dodavanje nove putanje:

```
sys.path.append(r"<putanja>")
```

Primjer 4: Dodavanje nove putanje.

```
import sys
sys.path.append(r"D:\D460\Moduli")
print(sys.path)
```

Izlaz:

```
['C:/D460/Skripte',
 'C:\\Users\\Marko\\AppData\\Local\\Programs\\
 Python\\Python313\\Lib\\idlelib',
 'C:\\Users\\Marko\\AppData\\Local\\Programs\\
 Python\\Python313\\python313.zip',
 'C:\\Users\\Marko\\AppData\\Local\\Programs\\
 Python\\Python313\\DLLs',
 'C:\\Users\\Marko\\AppData\\Local\\Programs\\
 Python\\Python313\\Lib',
 'C:\\Users\\Marko\\AppData\\Local\\Programs\\
 Python\\Python313',
 'C:\\Users\\Marko\\AppData\\Local\\Programs\\
 Python\\Python313\\Lib\\site-packages',
 'D:\\D460\\Moduli']
```

Primjer 5: Nakon što je dodana putanja do direktorija u kojemu se nalazi modul `matematickeOperacije`, moguće je najaviti korištenje funkcijama koje su implementirane unutar toga modula.

```
import sys

sys.path.append(r"D:\D460\Moduli")

import matematickeOperacije

print(matematickeOperacije.zbroj(1, 2))
```

Izlaz:

```
3
```

Primjer 6: Također je moguće uključiti u program i željene funkcije iz modula funkcije.

```
import sys

sys.path.append(r"D:\D460\Moduli")

from matematickeOperacije import zbroj

print(zbroj(1, 2))
```

Izlaz:

```
3
```

Primjer 7: Pozivom funkcije `help()` moguće je dobiti opis sadržaja modula. U nastavku je prikazan primjer korištenja ovom funkcijom.

```
import sys

sys.path.append(r"D:\D460\Moduli")

import matematickeOperacije

print(help("matematickeOperacije"))
```

```

Izlaz:
Help on module matematickeOperacije:

NAME
    matematickeOperacije

FUNCTIONS
    kolicnik(a, b)
        Funkcija vraća količnik dvaju
        brojeva.

    razlika(a, b)
        Funkcija vraća razliku dvaju brojeva.

    umnozak(a, b)
        Funkcija vraća umnožak dvaju brojeva.

    zbroj(a, b)
        Funkcija vraća zbroj dvaju brojeva.

FILE
    d:\d460\moduli\matematickeoperacije.py

None

```

Poziv funkcije `help()` za primjer modula funkcije daje nam sljedeće informacije:

- naziv modula
- popis svih funkcija definiranih u modulu (ako modul sadrži razrede i konstante, oni će također biti ovdje prikazani)
 - popis je sortiran abecednim redoslijedom
 - ako funkcija sadrži dokumentacijski string (docstring), njegov sadržaj bit će prikazan uz funkciju
- putanju na disku na kojoj se modul nalazi.

Napomena

Ako se u napisanome modulu osim funkcija nalazi i dodatni programski kôd izvan funkcija, nakon što uključimo modul u program, izazvat će se njegovo izvršavanje. U nekim slučajevima to nije poželjno jer se želimo koristiti samo funkcijama implementiranim unutar modula.

Primjer 8: U nastavku je primjer programskoga kôda u modulu `matematickeOperacije` koji se nalazi izvan funkcije `zbroj()`.

```

def zbroj(a, b):
    '''Funkcija vraća zbroj dvaju brojeva.'''
    return a + b

print("Pokretanje programa!")

```

Primjer 9: U nastavku slijedi primjer koji prikazuje da se uključivanjem modula `matematickeOperacije` program krenuo izvršavati.

```
import sys
sys.path.append(r"D:\D460\Moduli")

import matematickeOperacije

Izlaz:
    Pokretanje programa!
```

Kako bi se spriječilo automatsko izvršavanje programskoga kôda prilikom uključivanja modula, Python se koristi posebnom ugrađenom varijablom `__name__` koja može poprimiti sljedeće vrijednosti:

- `__main__` – kada se datoteka pokreće kao glavni program
- `imeModula` – kada se datoteka uključuje u drugi program.

Primjer 10: Unutar modula rabi se uvjet `if` kojim se provjerava vrijednost varijable `__name__`. Ako je vrijednost te varijable `__main__`, to znači da je datoteka pokrenuta kao samostalni program te se izvršava kôd unutar uvjeta `if`. Ako uvjet nije zadovoljen, to znači da je datoteka uključena u drugi program, pri čemu varijabla `__name__` poprima vrijednost naziva modula, u našem primjeru `matematickeOperacije`, te se kôd unutar uvjeta neće izvršiti.

```
def zbroj(a, b):
    '''Funkcija vraća zbroj dvaju brojeva.'''
    return a + b

if __name__ == "__main__":
    print("Pokretanje programa!")
```

6.3. Vježba: Moduli i paketi

1. Postavite PATH varijable okoline za `python` i `pip`. Provjerite unutar naredbenoga retka jesu li varijable ispravno postavljene pokretanjem naredbi `python --version` i `pip --version`.
2. U internetskome pregledniku otvorite PyPI te pronađite paket imena `requests`. Proučite koje su sve informacije dostupne u profilu traženoga paketa.
3. Instalirajte paket `requests`. Nakon instalacije napišite Python program koji dohvaća sadržaj *web*-stranice po vašem izboru.
4. Kreirajte modul imena `mojModul`, unutar njega implementirajte funkciju `funkcijaIspisuje()` koja na ekran ispisuje proizvoljni niz znakova. U nekome drugom Python programu najavite korištenje funkcije iz modula `mojModul` i pozovite ju.

7. Rad s CSV datotekama

Po završetku ovoga poglavlja polaznik će moći:

- *obrađivati CSV dokumente.*

U ovome poglavlju obrađuje se jedan od najčešće upotrebljavanih formata za pohranu podataka, a to je CSV.

**Trajanje
poglavlja:**
60 min

7.1. Općenito o CSV datotekama

CSV (engl. *Comma-Separated Values*) je datoteka tekstualnoga formata koja se koristi jednostavnom strukturom za pohranu podataka. Podaci su organizirani u retke i stupce, pri čemu se vrijednosti unutar retka odvajaju određenim znakom. Taj znak najčešće je zarez ili točka sa zarezom, a može biti i neki drugi proizvoljno definiran znak.

CSV datoteka može se kreirati iz raznih izvora, na primjer iz programa poput Excela, programskoga kôda, baze podataka i tako dalje. U nastavku je prikazan primjer izgleda podataka napisanih u Excelu te njihov izgled nakon izvoza u CSV format.

	A	B	C
1	Ime	Prezime	Grad
2	Ivan	Horvat	Zagreb
3	Pero	Novak	Split
4	Josip	Boban	Rijeka
5	Luka	Turk	Osijek

Nakon izvoza podaci za četiri osobe u CSV datoteci izgledaju ovako:

```
Ime;Prezime;Grad
Ivan;Horvat;Zagreb
Pero;Novak;Split
Josip;Boban;Rijeka
Luka;Turk;Osijek
```

Kao što se iz priloženoga vidi, vrijednosti unutar jednoga retka odvojene su znakom točke sa zarezom (;), dok su redci odvojeni novim retkom. Uobičajeno je da se datoteka formatira tako da prva linija opisuje elemente jednoga zapisa (retka), no nije obvezno. U gornjem primjeru to su: ime i prezime osobe te grad, a potom svaki sljedeći redak predstavlja jedan zapis podataka.

Znak koji se rabi za odvajanje vrijednosti naziva se razdjelnik (engl. *delimiter*). U gornjemu primjeru CSV datoteke kao razdjelnik se rabi točka sa zarezom (;), što je zadana vrijednost pri izvozu podataka iz Excela u nekim regionalnim postavkama. Moguće je rabiti i druge razdjelnike poput zareza (,), tabulatora (\t), dvotočke (:) i tako dalje. Za

ispravan rad s CSV datotekama potrebno je poznavati strukturu CSV datoteke, kao i koji je razdjelnik je upotrijebljen.

Rad s CSV datotekama vrlo je praktičan zbog njihove jednostavne strukture, a samim time i lake obrade. U programskome jeziku Python dostupan je ugrađeni modul `csv`, koji omogućuje jednostavno čitanje podataka iz CSV datoteka, kao i njihovo zapisivanje.

7.2. Čitanje CSV datoteke

Za potrebe demonstracije čitanja CSV datoteke koristit će se CSV datotekom sadržaja prikazanoga u nastavku, imena: `podaci.csv`, pri čemu se kao razdjelnikom koristi znakom: `(;)`.

```
Ime;Prezime;Grad
Ivan;Horvat;Zagreb
Pero;Novak;Split
Josip;Boban;Rijeka
Luka;Turk;Osijek
```

Primjer 1: U nastavku se nalazi programski kôd koji učitava i ispisuje sadržaj datoteke `podaci.csv`.

```
import csv

with open('podaci.csv') as f:
    csvCitac = csv.reader(f,
                           delimiter=';')

    header = next(csvCitac)
    print(f"{header[0]} {header[1]} {header[2]}\n")

    brojac = 0
    for zapis in csvCitac:
        print(f"{zapis[0]} {zapis[1]} {zapis[2]}")
        brojac += 1

    print(f"\nBroj zapisa: {brojac}.")
```

```
Izlaz:
    Ime Prezime Grad

    Ivan Horvat Zagreb
    Pero Novak Split
    Josip Boban Rijeka
    Luka Turk Osijek

    Broj zapisa: 4.
```

U gore prikazanome programskom kôdu može se vidjeti da se tok podataka prema CSV datoteci otvara identično kao i kod drugih tekstualnih datoteka. Novost je metoda `reader()` koja se nalazi u modulu `csv`.

Sintaksa metode `reader()` jest:

```
csv.reader(csvfile, dialect='excel', **fmtparams)
```

Povratna vrijednost metode `reader()` jest objekt koji omogućuje čitanje podataka iz CSV datoteke. Podaci se čitaju liniju po liniju, a unutar svake linije pristup pojedinom podatku moguć je putem indeksa. Indeksi započinju od vrijednosti 0. Upotrijebljena metoda `next()` dohvaća sljedeći red CSV datoteke, tj. zaglavlje.

Opis parametara metode `reader()`:

- `csvfile` – tok podataka iz kojega se podaci čitaju
- `dialect` – naziv upotrijebljenoga dijalekta, zadana vrijednost jest `'excel'` (Postoje tri preddefinirana dijalekta: `['excel', 'excel-tab', 'unix']`. Moguće je definirati i vlastite dijalekte s pripadajućim parametrima.)
- `fmtparams` – opcionalni parametri koji omogućuju definiranje specifičnih postavki na temelju kojih se CSV datoteka čita.

Opcionalni parametri metode `reader()` (`**fmtparams`):

- `delimiter` – znak koji razdvaja vrijednosti unutar retka
- `doublequote` – kontrolira ponašanje navodnika unutar podatka
 - `True` (zadano) – dva navodnika interpretiraju se kao jedan prilikom čitanja, pri pisanju jedan navodnik zapisuje se kao dva
- `escapechar` – parametar koji se rabi za preskakanje znaka koji se nalazi s njegove desne strane (taj se znak ne tretira kao poseban znak)
- `lineterminator` – znak koji označava kraj retka CSV datoteke
- `quotechar` – određuje znak koji omeđuje podatak s lijeve i desne strane
- `quoting` – kontrolira uporabu navodnika prilikom čitanja – metoda `reader()` ili pisanja – metoda `writer()`
 - `csv.QUOTE_MINIMAL` – podatak se omeđuje znakom `quotechar` samo ako sadrži razdjelnik, navodnik ili novi redak
 - `csv.QUOTE_ALL` – svi podaci uvijek se omeđuju znakom `quotechar`
 - `csv.QUOTE_NONNUMERIC` – parametar `quotechar` rabi se samo za tekstualne podatke, svi brojevi podaci automatski se konvertiraju u `float`

- o `csv.QUOTE_NONE` – parametar `quotechar` ne postavlja se
- `skipinitialspace` – ako je `True`, razmak nakon razdjelnika ignorira se
- `strict` – ako je `True`, podiže se iznimka kada CSV datoteka nije ispravnoga formata.

Preddefinirane vrijednosti parametara:

Parametar	Preddefinirana vrijednost
<code>delimiter</code>	<code>,</code>
<code>doublequote</code>	<code>True</code>
<code>escapechar</code>	<code>None</code>
<code>lineterminator</code>	<code>\r\n</code>
<code>quotechar</code>	<code>"</code>
<code>quoting</code>	<code>QUOTE_MINIMAL</code>
<code>skipinitialspace</code>	<code>False</code>
<code>strict</code>	<code>False</code>

7.3. Pisanje u CSV datoteku

Primjer 1: U nastavku se nalazi programski kôd koji zapisuje podatke u CSV formatu u datoteku `podaci.csv`.

```
import csv

with open('podaci.csv', mode='w') as f:
    csvPisac = csv.writer(f,
                           delimiter=';',
                           quotechar='"',
                           quoting=csv.QUOTE_ALL,
                           lineterminator='\n')

    csvPisac.writerow(['Ime', 'Prezime', 'Grad'])
    csvPisac.writerow(['Ivan', 'Horvat', 'Zagreb'])
    csvPisac.writerow(['Pero', 'Novak', 'Split'])
    csvPisac.writerow(['Josip', 'Boban', 'Rijeka'])
    csvPisac.writerow(['Luka', 'Turk', 'Osijek'])
```

Sadržaj datoteke `podaci.csv`:

```
"Ime";"Prezime";"Grad"
"Ivan";"Horvat";"Zagreb"
"Pero";"Novak";"Split"
"Josip";"Boban";"Rijeka"
"Luka";"Turk";"Osijek"
```

Za pisanje podataka u CSV formatu u datoteku najprije je potrebno pomoću metode `writer()` iz modula `csv` kreirati objekt koji omogućuje zapisivanje podataka u CSV datoteku.

Sintaksa metode `writer()`:

```
csv.writer(csvfile, dialect='excel', **fmtparams)
```

Povratna vrijednost metode `writer()` jest objekt koji omogućuje konvertiranje podataka u CSV format.

Opis parametara metode `writer()`:

- `csvfile` – tok podataka u koji će se podaci zapisivati
- `dialect` – naziv korištenoga dijalekta, preddefinirana vrijednost je `'excel'`
- `fmtparams` – opcionalni parametri koji omogućuju definiranje specifičnih postavki na temelju kojih će se kreirati CSV datoteka.

Napomena: ovi su parametri jednaki parametrima `fmtparams`, koji su opisani u prethodnome potpoglavlju 7.2. Čitanje CSV datoteke.

7.4. Kreiranje novoga dijalekta

U ovome poglavlju prikazano je kako kreirati vlastiti dijalekt koji definira specifične postavke za čitanje i pisanje CSV datoteka.

Unutar modula `csv` dostupna je metoda `register_dialect()` čija je sintaksa:

```
register_dialect(name, /, dialect='excel', **fmtparams)
```

Opis parametara metode `register_dialect()`:

- `name` – naziv novoga (vlastitoga) dijalekta
- `/` – označava da su svi parametri prije njega isključivo pozicijski tj. ne mogu se postaviti po imenu
- `dialect='excel'` – opcionalni parametar, ako se navede, novi dijalekt nasljeđuje sve postavke iz danoga dijalekta, zadani je dijalekt `excel`)
- `**fmtparams` – opcionalni parametri koji definiraju specifične postavke dijalekta (`delimiter`, `doublequote`, `escapechar`, `lineterminator`, `quotechar`, `quoting`, `skipinitialspace`, `strict`).

Primjer 1: Registracija i uporaba vlastitoga dijalekta (parametri `delimiter`, `lineterminator`).

```
import csv

csv.register_dialect('srce',
                    delimiter='|',
                    lineterminator='\n')

with open('podaci.csv', mode='w') as f:
    csvPisac = csv.writer(f, dialect='srce')

    csvPisac.writerow(['Ime', 'Prezime', 'Grad'])
    csvPisac.writerow(['Ivan', 'Horvat', 'Zagreb'])
    csvPisac.writerow(['Pero', 'Novak', 'Split'])
    csvPisac.writerow(['Josip', 'Boban', 'Rijeka'])
    csvPisac.writerow(['Luka', 'Turk', 'Osijek'])
```

Sadržaj datoteke - podaci.csv:

```
Ime|Prezime|Grad
Ivan|Horvat|Zagreb
Pero|Novak|Split
Josip|Boban|Rijeka
Luka|Turk|Osijek
```

Primjer 2: Registracija i uporaba vlastitoga dijalekta (parametri `delimiter`, `quotechar`, `quoting`, `lineterminator`).

```
import csv

csv.register_dialect('srce',
                    delimiter='|',
                    quotechar='#',
                    quoting=csv.QUOTE_ALL,
                    lineterminator='\n')

with open('podaci.csv', mode='w') as f:
    csvPisac = csv.writer(f, dialect='srce')

    csvPisac.writerow(['Ime', 'Prezime', 'Grad'])
    csvPisac.writerow(['Ivan', 'Horvat', 'Zagreb'])
    csvPisac.writerow(['Pero', 'Novak', 'Split'])
    csvPisac.writerow(['Josip', 'Boban', 'Rijeka'])
    csvPisac.writerow(['Luka', 'Turk', 'Osijek'])
```

Sadržaj datoteke - podaci.csv:

```
#Ime##Prezime##Grad#
#Ivan##Horvat##Zagreb#
#Pero##Novak##Split#
#Josip##Boban##Rijeka#
#Luka##Turk##Osijek#
```

Primjer 3: Registracija i uporaba vlastitoga dijalekta (parametri `doublequote`, `escapechar`).

```
import csv

# Registracija vlastitog dijalekta s parametrima
csv.register_dialect('srce',
    # Ponašanje navodnika unutar podataka
    doublequote=True,
    # Parametar za preskakanje znakova
    escapechar='\\')

# Pisanje podataka u CSV datoteku
with open('podaci.csv', mode='w', newline='') as f:
    csvPisac = csv.writer(f, dialect='srce')
    csvPisac.writerow(['Ime', 'Prezime', 'Grad'])

    # doublequote=True
    csvPisac.writerow(['Ivan', 'Horvat', 'Grad "ZG"'])

    # escapechar='\\' - u datoteku se zapisuje \\
    # ako ovog parametra nema, zapisuje se \
    csvPisac.writerow(['Pero', 'Novak', 'Split \\ Jug'])
```

Sadržaj datoteke - podaci.csv:

```
Ime,Prezime,Grad
Ivan,Horvat,"Grad "ZG"'"
Pero,Novak,Split \\ Jug
```

7.5. Vježba: Rad s CSV datotekama

1. U ovome se zadatku koristite priloženom datotekom imena `7.5.1.csv`. Učitajte CSV datoteku te ispišite sadržaj datoteke tako da korisničke podatke povežete sa značenjem tih podataka, koje se nalazi u prvome retku (zaglavlju). Izgled ispisa za svaki redak neka bude sljedećega oblika:

```
Ime: Ivan
Prezime: Horvat
Grad: Zagreb
Godine: 18
```

2. U ovome se zadatku koristite priloženom datotekom imena `7.5.2.csv`. Na temelju programskoga kôda napisanoga u prethodnome zadatku provjerite kako funkcionira opcionalni parametar `skipinitialspace`. Primijetite da se unutar CSV datoteke ispred elemenata nalazi nasumično odabrani broj razmaka. Navedeni parametar treba ukloniti tako dodane razmake.
3. Osmislite proizvoljan sadržaj kojim ćete, pozivom metode `writerow()` iz modula `csv`, popuniti CSV datoteku. Kao razdjelnik postavite znak (;). Također, isprobajte korištenje i ostalim opcionalnim parametrima: `doublequote`, `escapechar`, `lineterminator`, `quotechar`, `quoting`. Kako biste vidjeli utjecaj svakoga pojedinog parametra, dodajte ih jedan po jedan. Za svaki dodani parametar ponovo generirajte CSV datoteku te analizirajte njezin izgled.

7.6. Pitanja za ponavljanje: Rad s CSV datotekama

1. Navedite nekoliko najčešće korištenih znakova za odvajanje elemenata unutar CSV dokumenta.
2. Kako se zove modul u kojemu se nalaze metode za obradu CSV dokumenta?
3. Kako se zovu metode za čitanje i pisanje iz modula `csv`?
4. Za što u metodi `reader()` i `writer()` iz modula `csv` služi njihov opcionalni parametar `delimiter`?
5. Kako se zove metoda iz modula `csv` koja omogućuje kreiranje novoga dijalekta?

8. Objektno orijentirano programiranje

Trajanje
poglavlja:
300 min

Po završetku ovoga poglavlja polaznik će moći:

- *razumjeti osnovne pojmove objektno orijentiranoga programiranja (razred, objekt, atribut, metoda)*
- *kreirati razrede i objekte*
- *definirati i rabiti metode:*
 - *specijalne metode*
 - *metode objekata*
 - *statičke metode*
- *definirati vlasništvo i vidljivost varijable*
- *shvatiti kako funkcionira vidljivost varijable: `public`, `protected` i `private`*
- *koristiti se svojstvima*
- *objasniti i upotrijebiti:*
 - *nasljeđivanje*
 - *nadjačavanje*
 - *preopterećenje*
 - *apstraktne razrede*
- *ulančavati metode.*

Način razvoja programskoga kôda koji je prikazan u dosadašnjemu dijelu tečaja naziva se proceduralno programiranje. Riječ je o pristupu u kojemu se logika programa temelji na funkcijama, odnosno blokovima programskoga kôda koji izvršavaju određene zadatke.

Pristup koji će biti objašnjen u ovome poglavlju zove se objektno orijentirano programiranje, kraće OOP (engl. *Object-Oriented Programming*). Većinu programa moguće je uspješno napisati uporabom proceduralnoga pristupa, a u složenijim i velikim projektima preporučuje se rabiti objektno orijentirano programiranje. Objektno orijentirano programiranje omogućuje bolju organizaciju programskoga kôda te olakšava njegovo održavanje i nadogradnju. Objektno orijentirano programiranje samo po sebi ne sprječava pisanje lošega programskog kôda, već pruža jasna pravila koja programerima pomažu u razvoju kvalitetnijih rješenja.

U objektno orijentiranome programiranju ideja je da problem koji se rješava bude podijeljen na manje, međusobno povezane logičke cjeline. Na primjer, ako program treba obrađivati podatke o osobama, osnovni element takvoga programa može biti razred `Osoba`. Svakoj osobi možemo pridružiti određene vrijednosti, koje u duhu objektno orijentiranoga pristupa nazivamo atributima. Primjeri atributa jesu: ime, prezime, adresa, OIB i tako dalje. Osim atributa, u svakome razredu (instanca razreda naziva se objekt) mogu biti pridružene i metode. Na primjer, osoba može hodati, sjesti, trčati itd., pa tako imamo

odgovarajuće metode: `hodaј()`, `sjedni()`, `trci()`, koje objektu `Osoba` daju funkcionalnost i definiraju njegovo ponašanje. U nastavku se nalazi ilustrativni prikaz razreda `Osoba`.

Ime razreda:	Osoba
Atributi	ime
	prezime
	adresa
	oib
	...
Metode:	hodaј()
	sjedni()
	trci()
	...

Osnovni pojmovi objektno orijentiranoga programiranja:

- razred (engl. *class*)
- objekt (engl. *object*)
- nasljeđivanje (engl. *inheritance*)
- enkapsulacija ili ućahurivanje (engl. *encapsulation*)
- apstrakcija (engl. *abstraction*)
- polimorfizam ili višeobličje (engl. *polymorphism*).

8.1. Razredi i objekti

U nastavku slijedi tablica sinonima koji se često pojavljuju prilikom čitanja literature, a imaju isto značenje.

Hrvatski nazivi (sinonimi)	Engleski nazivi
razred/klasa	engl. <i>class</i>
atribut/svojstvo	engl. <i>attribute</i>
objekt/instanca/jedinka	engl. <i>instance</i>

Razred je osnovna programska cjelina u objektno orijentiranome načinu programiranja. Unutar razreda definiran je shematski plan (engl. *blueprint*) na temelju kojega se kreiraju objekti. Razred sadrži attribute i metode. Atributi opisuju objekt pomoću različitih podataka. Metode su funkcije definirane unutar razreda koje opisuju ponašanje objekta i pozivaju se nad objektom kojemu pripadaju. Pridruživanje metode instanci objekta ostvaruje se pomoću parametra `self`. Parametar `self` predstavlja referencu na trenutni objekt i omogućuje pristup njegovim atributima i metodama. Parametar `self` detaljnije je objašnjen u nastavku tečaja.

Primjer 1: Osnovni primjer kreiranja razreda.

```
class MojRazred:
    # Varijable

    def imeMetode(self):
        # Implementacija metode
```

Svaki razred započinje ključnom riječju `class`, a nakon ključne riječi navodi se proizvoljno ime razreda. Prema Python preporuci riječi u nazivu razreda pišu se velikim početnim slovom, na primjer: `Osoba`, `KoordinatniSustav` i slično. Unutar tako kreiranoga razreda (klase) pišu se:

- atributi – svojstva razreda
- metode – funkcionalnosti razreda.

Primjer 2: U nastavku je prikazan programski kôd koji predstavlja neke radnje koje osobe mogu obavljati. Unutar razreda imena `Osoba` implementirane su tri metode `hoda()`, `sjedni()`, `trci()`. U glavnome programu kreiran je jedan objekt (instanca) razreda `Osoba` imena `o`. Razred je predložak (plan), dok je objekt konkretna instanca toga razreda. Drugim riječima, razred definira kakav će objekt biti, a objekt je stvarni primjer toga razreda u programu. Iz donjega primjera moguće je vidjeti da se objekt nekoga razreda može kreirati na sljedeći način:

```
imeObjekta = ImeRazreda()
```

Metode nekoga objekta pozivaju se pomoću sljedeće sintakse:

```
imeObjekta.imeMetode()
```

Napomena: unutar donjega primjera vidljivo je da se upotrebljava parametar `self`. Taj parametar bit će detaljnije objašnjen u nastavku tečaja.

```
class Osoba:
    def hoda(self):
        print("Hoda!")

    def sjedni(self):
        print("Sjedni!")

    def trci(self):
        print("Trci!")

o = Osoba()
o.hoda()
o.sjedni()
o.trci()

Izlaz:
Hoda!
Sjedni!
Trci!
```

U slučaju da je potrebno kreirati "beskoristan" razred koji ne sadrži programski kôd, to je moguće postići tako da se u tijelo kreiranoga razreda napiše naredba `pass`.

Primjer 3: U narednome primjeru prikazan je razred imena `Osoba` koji ne sadrži atribute ni metode. Ovaj je razred u našem primjeru definiran pomoću naredbe `pass`, koja označava prazno tijelo razreda. Unutar glavnoga programa kreirana su tri objekta (instance) razreda `Osoba`, a to su: `o1`, `o2`, `o3`. Iz ispisa je moguće zaključiti da se sva tri objekta nalaze na drugoj adresi u memoriji, što znači da su stvorena tri zasebna objekta iste klase `Osoba`. Svaki objekt ima vlastiti prostor u memoriji, iako dijeli istu strukturu definiranu razredom.

```
class Osoba:
    pass

o1 = Osoba()
o2 = Osoba()
o3 = Osoba()

print(o1)
print(o2)
print(o3)

Izlaz:
<__main__.Osoba object at 0x0000017735DC9400>
<__main__.Osoba object at 0x0000017737761450>
<__main__.Osoba object at 0x0000017737761590>
```

8.2. Vrste metoda

Unutar pojedinoga razreda moguće je definirati više vrsta metoda. Svaka od navedenih vrsta metoda ima posebnu ulogu i način uporabe, što će biti objašnjeno u nastavku.

Različite vrste metoda omogućuju fleksibilnije i organiziranije pisanje programskoga kôda te olakšavaju rad s objektima i njihovim podacima. Ovisno o načinu uporabe i svrsi metode, dijele se na sljedeće vrste:

- specijalne metode (engl. *special methods*)
- metode objekta (engl. *instance methods*)
- statičke metode (engl. *static methods*)
- metode razreda (engl. *class methods*) (*ne obrađuje se*).

8.3. Specijalne metode (`__init__`, `__str__`)

Specijalne metode jesu unaprijed definirane metode koje se mogu rabiti unutar razreda. Iako postoji mnoštvo unaprijed definiranih specijalnih metoda, kao što su `__init__`, `__str__`, `__repr__`, `__format__` i druge, u nastavku će se obraditi dvije specijalne metode, a to su: `__init__` i `__str__`.

8.3.1. Metoda `__init__`

Metoda `__init__` naziva se konstruktor (engl. *constructor*). Konstruktor se automatski poziva prilikom instanciranja objekta i njegova je osnovna namjena inicijalizacija atributa na željene vrijednosti prije početka korištenja kreiranim objektom. Parametar `self` predstavlja referencu na konkretan objekt. Bez toga parametra ne bi bilo moguće spremati podatke u objekt. `self` je detaljnije objašnjen u narednome poglavlju.

Sintaksa konstruktora bez parametara:

```
def __init__(self):
```

Primjer 1: Jednostavan primjer implementacije konstruktora koji ispisuje poruku i instanciranja dvaju objekata razreda `Osoba`.

```
class Osoba:
    def __init__(self):
        print("Kreiran je objekt!")

o1 = Osoba()
o2 = Osoba()

Izlaz:
    Kreiran je objekt!
    Kreiran je objekt!
```

Kao što je vidljivo iz gornjega primjera, u programu se kreiraju dva objekta: `o1` i `o2`. Iako metoda `__init__` nigdje nije eksplicitno pozvana, njezina se implementacija automatski izvršava prilikom instanciranja objekata razreda `Osoba`. Razlog izvršavanja jest to što se konstruktor uvijek poziva pri instanciranju objekta razreda u kojemu je definiran.

Sintaksa konstruktora s parametrima:

```
def __init__(self, arg1, arg2, ...):
```

Konstruktor osim inicijalizacije objekta kreira i attribute te ih postavlja na željene vrijednosti.

Primjer 2: Jednostavan primjer implementacije konstruktora koji postavlja dva atributa naziva `ime` i `prezime`, dok se u glavnome dijelu programa instanciraju dva objekta razreda `Osoba`.

```
class Osoba:
    def __init__(self, ime, prezime):
        self.ime = ime
        self.prezime = prezime

o1 = Osoba("Ivan", "Horvat")
o2 = Osoba("Pero", "Novak")

print(o1.ime, o1.prezime)
print(o2.ime, o2.prezime)

Izlaz:
    Ivan Horvat
    Pero Novak
```

U gornjemu primjeru programskoga kôda vidljivo je da se prilikom instanciranja objekta razredu `Osoba` prenosi vrijednost imena i prezimena. U prvome pozivu ime je "Ivan", dok je prezime "Horvat", a u drugome pozivu ime je "Pero", dok je prezime "Novak". Konstruktor će za svaki objekt (`o1` i `o2`) postaviti atribute `ime` i `prezime` na predane vrijednosti.

8.3.2. Metoda `__str__`

Metoda `__str__` rabi se za definiranje tekstualnoga prikaza objekta. Ova metoda automatski se poziva kada se objekt pokušava ispisati pomoću funkcije `print()`. Njezina je osnovna namjena omogućiti jasan i razumljiv prikaz podataka objekta korisniku.

Sintaksa metode `__str__`:

```
def __str__(self):
```

Primjer 3: Jednostavan primjer implementacije metode `__str__` koja vraća tekstualni prikaz objekta razreda `Osoba`.

```
class Osoba:
    def __init__(self, ime, prezime):
        self.ime = ime
        self.prezime = prezime

    def __str__(self):
        return self.ime + " " + self.prezime

o1 = Osoba("Ivan", "Horvat")
o2 = Osoba("Ana", "Marić")

print(o1)
print(o2)
```

Izlaz:

```
Ivan Horvat
Ana Marić
```

Kao što je vidljivo iz primjera, metoda `__str__` nije eksplicitno pozvana, ali automatski se izvršava prilikom ispisa objekta. Umjesto zadane reprezentacije objekta, koja uključuje naziv razreda i memorijsku adresu, metoda `__str__` omogućuje prilagođeni ispis.

Primjer 4: Ako metoda `__str__` nije definirana, ispis objekta izgleda ovako:

```
class Osoba:
    def __init__(self, ime, prezime):
        self.ime = ime
        self.prezime = prezime

o1 = Osoba("Ivan", "Horvat")
o2 = Osoba("Ana", "Marić")
```

```
print(o1)
print(o2)
```

Izlaz:

```
<__main__.Osoba object at 0x00000217CEC09400>
<__main__.Osoba object at 0x00000217CEFA1090>
```

Metoda `__str__` uvijek mora vraćati niz znakova, odnosno tip podataka `str`, inače će doći do greške prilikom izvođenja programa.

8.4. Vježba: Objektno orijentirano programiranje I

1. Napišite jednostavni razred imena `Vozilo`. Tako kreirani razred neka bude potpuno prazan, bez atributa i metoda. U glavnome programu kreirajte tri objekta razreda `Vozilo` i pozivom funkcije `print()` ispišite podatke o tim trima objektima.
2. Nadogradite razred `Vozilo` tako da implementirate konstruktor `__init__` koji prima sljedeće parametre: `marka`, `model` i `godina`. Unutar konstruktora definirajte odgovarajuće atribute i postavite ih na predane vrijednosti. U glavnome programu kreirajte najmanje dva objekta razreda `Vozilo` s različitim vrijednostima te ispišite njihove atribute pomoću metode `print()`.
3. Dodatno nadogradite razred `Vozilo` implementacijom metode `__str__` koja vraća tekstualni prikaz objekta u sljedećemu formatu: `Marka: Audi, Model: A5, Godina: 2020.`
U glavnome programu kreirajte dva ili više objekata razreda `Vozilo` te ih ispišite pomoću metode `print()`. Usporedite dobiveni ispis s ispisom iz prvoga zadatka i objasnite razliku.

8.5. Metode objekta

Ako se metoda nekoga razreda želi povezati s nekim objektom, to je moguće napraviti pomoću parametra `self`. Tako povezane metode zovu se metode objekata. Povezivanje metode objekta, uključujući i konstruktor, radi se tako da takva metoda kao prvi parametar prima referencu na objekt. Parametar `self` predstavlja referencu na trenutni objekt nad kojim se metoda poziva. Prema Python konvenciji za ime toga parametra uzima se riječ `self`, iako riječ `self` nije ključna riječ.

Sintaksa za definiranje takve metode jest:

```
def imeMetode(self, [arg1, ...])
```

Sintaksa za pristupanje atributima unutar takve metode jest:

```
self.imeAtributa
```

Parametar `self` ne prosljeđuje se prilikom poziva metode, već ga Python automatski dodaje. Ako se parametar `self` izostavi, Python neće moći povezati metodu s objektom te će se pojaviti greška tipa `TypeError` odmah prilikom poziva metode.

Primjer 1: Demonstracija korištenja metodom objekta.

```

class Osoba:

    def __init__(self, ime, prezime):
        self.ime = ime
        self.prezime = prezime

    def ispisPodataka(self):
        print("Podaci su: ", self.ime,
self.prezime)

    def hodaj(self):
        print(self.ime, "hoda!")

o1 = Osoba("Ivan", "Horvat")
o2 = Osoba("Ana", "Hrvat")

o1.ispisPodataka()
o2.ispisPodataka()

o1.hodaj()
o2.hodaj()

Izlaz:
    Podaci su:   Ivan Horvat
    Podaci su:   Ana Hrvat
    Ivan hoda!
    Ana hoda!

```

U gornjem primjeru kreiran je razred imena `Osoba`. Kreirani razred sastoji se od konstruktora, dvaju atributa i dviju metoda objekta. U glavnome dijelu programa kreirana su dva objekta razreda `Osoba`. Prilikom kreiranja objekata `o1` i `o2` automatski se poziva konstruktor koji atributima svakoga objekta postavlja predane vrijednosti, ime i prezime. Nakon kreiranja objekata razreda nad obama objektima pozvana je metoda `ispisPodataka()`. Ta metoda ispisuje vrijednosti atributa `ime` i `prezime` svakoga pojedinog objekta. Sintaksa poziva metode nad objektom s dodatnim argumentima jest:

```
imeObjekta.imeMetode([arg1, ...])
```

Metoda `hodaj()` ispisuje ime osobe koja hoda, odnosno vrijednost atributa `ime` s informacijom o hodanju.

8.5.1. Primjeri potencijalnih grešaka – parametar `self`

Primjer 2: Izostavljen je parametar `self` pri dohvaćanju vrijednosti atributa unutar metode objekta. Izostavljeni kôd precrtan je i označen žutom bojom.

```
class Osoba:

    def __init__(self, ime, prezime):
        self.ime = ime
        self.prezime = prezime

    def ispisPodataka(self):
        print("Podaci su: ", self.ime,
self.prezime)

o = Osoba("Ivan", "Horvat")
o.ispisPodataka()

Izlaz:
Traceback (most recent call last):
  File "D:\oop.py", line 12, in <module>
    o.ispisPodataka()
    ~~~~~^
  File "D:\oop.py", line 8, in ispisPodataka
    print("Podaci su: ", self.ime, prezime)
                                ^^^^^^
NameError: name 'prezime' is not defined. Did
you mean: 'self.prezime'?
```

Uvidom u ispis greške može se primijetiti da Python ne može pronaći atribut naziva `prezime`. To se dogodilo zato što Python nije znao iz kojega objekta uzeti vrijednost atributa `prezime`.

Primjer 3: Parametar `self` izostavljen je iz popisa parametara metode. Izostavljeni kôd precrtan je i označen žutom bojom.

```
class Osoba:

    def __init__(self, ime, prezime):
        self.ime = ime
        self.prezime = prezime

    def ispisPodataka(self):
        print("Podaci su: ", self.ime,
self.prezime)

o = Osoba("Ivan", "Horvat")
o.ispisPodataka()

Izlaz:
Traceback (most recent call last):
  File "D:\script.py", line 10, in <module>
    o.ispisPodataka()
    ~~~~~^
TypeError: Osoba.ispisPodataka() takes 0
positional arguments but 1 was given
```

Iz poruke o grešci može se iščitati da metoda imena `ispisPodataka()` u svojoj implementaciji prima 0 parametara, ali jedan je argument predan. No ako se pogleda poziv te metode koji se nalazi u glavnome programu, vidljivo je da u zagradama nema nikakve predane vrijednosti.

Kada se poziva metoda objekta na način:

```
imeObjekta.imeMetode([arg1, arg2, ...])
```

Python će takav poziv automatski konvertirati u poziv oblika:

```
ImeRazreda.imeMetode(imeObjekta, [arg1, arg2, ...])
```

Prototip metode objekta:

```
imeMetode(self, [arg1, arg2, ...])
```

Ako se pogleda konvertirani poziv metode objekta i prototip metode objekta, može se primijetiti poveznica između njih preko parametra `self`.

8.6. Statičke metode

Statičkim metodama omogućuje se pozivanje metode izravno preko razreda, bez potrebe za kreiranjem objekta. One su vezane za razred, a ne za njegove instance.

Za razliku od metoda objekta, statičke metode ne primaju parametar `self`, jer nisu vezane ni za jedan objekt. Zbog toga statičke metode ne mogu pristupati ni atributima razreda, ni atributima objekata, ni drugim metodama koje ovise o instancama. One su korisne kada želimo funkcionalnost koja se odnosi na razred općenito, a ne na pojedine objekte.

Sintaksa definiranja statičke metode:

```
def imeMetode([arg1, ...])
    pass
```

Sintaksa poziva statičke metode:

```
ImeRazreda.imeMetode([arg1, ...])
```

Primjer 1: U sljedećemu primjeru programskoga kôda vidljivo je da unutar glavnoga programa nigdje nije kreiran objekt razreda `Osoba`, no metoda imena `punoljetnaOsoba()` uspješno se poziva. Ono što je važno naglasiti jest da se kao prvi parametar statičke metode ne navodi parametar `self`, jer se u statičku metodu ne želi prenositi objekt razreda. Ako se navede parametar `self`, onda to više neće biti statička metoda, nego metoda objekta.

```
class Osoba:

    def punoljetnaOsoba(godina):
        return godina >= 18

print(Osoba.punoljetnaOsoba(17))
print(Osoba.punoljetnaOsoba(18))

Izlaz:
False
True
```

Primjer 2: U sljedećemu primjeru statička se metoda poziva nad objektom razreda.

Iz ispisane greške moguće je vidjeti da se prilikom poziva statičke metode preko objekta razreda uz broj godina prenosi i ime `self`. Kako bi se izbjegao ovaj problem, uveden je dekorator `@staticmethod`. Ovaj dekorator omogućuje da se statička metoda poziva preko objekta razreda.

```
class Osoba:

    def __init__(self, ime):
        self.ime = ime

    def punoljetnaOsoba(godina):
        return godina >= 18

o = Osoba("Ivan")
print(o.punoljetnaOsoba(17))

Izlaz:
Traceback (most recent call last):
  File "D:\oop.py", line 11, in <module>
    print(o.punoljetnaOsoba(17))
    ~~~~~^
TypeError: Osoba.punoljetnaOsoba() takes 1
positional argument but 2 were given
```

Sintaksa definiranja statičke metode uz dekorator `@staticmethod`:

```
@staticmethod
def imeMetode([arg1, ...])
    pass
```

Sintaksa poziva statičke metode uz dekorator `@staticmethod`:

1. način: `ImeRazreda.imeMetode([arg1, ...])`
2. način: `imeObjekta.imeMetode([arg1, ...])`

Primjer 3: Primjer korištenja statičkom metodom uz dekorator `@staticmethod` prikazan je u nastavku. Unutar glavnoga programa donjega primjera najprije se kreira objekt imena `o` razreda `Osoba`. Nakon što je objekt kreiran, prikazana su dva načina kojima se moguće koristiti za poziv statičke metode. U ovome slučaju zbog korištenja dekoratorom `@staticmethod` prilikom interpretiranja programskoga kôda ne izaziva se greška.

```
class Osoba:

    def __init__(self, ime):
        self.ime = ime

    @staticmethod
    def punoljetnaOsoba(godina):
        return godina >= 18
```

Dekorator

Dekoratori omogućuju dodatne funkcionalnosti nad postojećim objektom bez izmjene strukture.

```
o = Osoba("Perica")
print(o.punoljetnaOsoba(17))
print(Osoba.punoljetnaOsoba(18))
```

Izlaz:

```
False
True
```

8.7. Vlasništvo varijabli

Varijable kreirane u razredu pripadaju tomu razredu i nije im moguće pristupati bez navođenja naziva objekta ili razreda, ovisno o tipu varijable. Varijable unutar razreda možemo podijeliti na dvije vrste, ovisno o tome pripada li varijabla razredu ili pak objektu:

- varijable razreda
- varijable objekta.

Varijable razreda (engl. *class variables*) ili statičke varijable (engl. *static variables*) dijele se među svim objektima razreda. Postoji samo jedna vrijednost takve varijable, neovisno o broju kreiranih instanci, što znači da promjena vrijednosti u bilo kojoj instanci (objektu) znači da će sve ostale instance razreda vidjeti tu novu, odnosno promijenjenu vrijednost.

Sintaksa definiranja varijable razreda:

```
class ImeRazreda:
    imeVarijable = <vrijednost>
```

Postavljanje i dohvaćanje vrijednosti unutar razreda:

Sintaksa za postavljanje vrijednosti jest:

```
ImeRazreda.imeVarijable = <vrijednost>
```

Sintaksa za dohvaćanje vrijednosti jest:

```
ImeRazreda.imeVarijable
```

Postavljanje i dohvaćanje vrijednosti iz glavnoga programa:

Sintaksa za postavljanje vrijednosti jest:

```
ImeRazreda.imeVarijable = <vrijednost>
imeObjekta.imeVarijable = <vrijednost>
```

Sintaksa za dohvaćanje vrijednosti jest:

```
ImeRazreda.imeVarijable
imeObjekta.imeVarijable
```

Varijable objekta (engl. *instance variables*) ili nestatičke varijable (engl. *non-static variables*) varijable su koje su u vlasništvu objekta razreda (instanci). U ovome slučaju svaki objekt ima svoju varijablu, tj. ovakva varijabla nije dijeljena i ni na koji način nije povezana između različitih instanci jednoga te istoga razreda. Varijabla objekta vezana je za svaku pojedinu instancu. U programskome kôdu ta se veza ostvaruje

korištenjem imenom `self`, koje označava vezu instance (objekta) razreda i same varijable.

Postavljanje i dohvaćanje vrijednosti unutar razreda:

Sintaksa za postavljanje vrijednosti jest:

```
self.imeVarijable = <vrijednost>
```

Sintaksa za dohvaćanje vrijednosti jest:

```
self.imeVarijable
```

Postavljanje i dohvaćanje vrijednosti iz glavnoga programa:

Sintaksa za postavljanje vrijednosti jest:

```
imeObjekta.imeVarijable = <vrijednost>
```

Sintaksa za dohvaćanje vrijednosti jest:

```
imeObjekta.imeVarijable
```

Primjer 1: U nastavku slijedi primjer u kojemu je prikazano korištenje obama tipovima varijabli (varijablama razreda i varijablama objekta).

```
class Demo:
    # Varijabla razreda
    broj = 0

    def __init__(self, naziv):
        # Varijabla objekta
        self.naziv = naziv
        Demo.broj += 1

    def ispis(self):
        print("Kreirano instanci: ", Demo.broj)
        print("Naziv instance: ", self.naziv)

d1 = Demo("PRVA")
d2 = Demo("DRUGA")
d3 = Demo("TRECA ")

d1.ispis()
d2.ispis()
d3.ispis()

Izlaz:
    Kreirano instanci:  3
    Naziv instance:   PRVA
    Kreirano instanci:  3
    Naziv instance:   DRUGA
    Kreirano instanci:  3
    Naziv instance:   TRECA
```

U gornjemu programskom kôdu nalaze se dvije varijable: `broj` i `naziv`.

- Varijabla `broj` (varijabla razreda): definirana je izvan svih metoda i njezina početna vrijednost postavljena je na 0. Ova se

vrijednost inicijalizira samo jednom, u trenutku kada Python učitava definiciju razreda `Demo`. Ovu varijablu možemo smatrati "globalnom" unutar toga razreda jer je zajednička (dijeljena) svim instancama (objektima) toga razreda.

- Varijabla `naziv` (varijabla objekta): definirana je unutar konstruktora `__init__` uporabom ključne riječi `self`. Ona pripada isključivo objektu za koji je kreirana i svaki objekt ima vlastitu, neovisnu kopiju te varijable.

8.8. Vježba: Objektno orijentirano programiranje II

1. Napišite razred imena `Grad` tako da implementirate konstruktor `__init__` koji prima sljedeće parametre: naziv, država i broj stanovnika. Unutar konstruktora definirajte odgovarajuće atribute i postavite ih na predane vrijednosti. U glavnome programu kreirajte najmanje dva objekta razreda `Grad` s različitim vrijednostima te implementirajte metodu objekta imena `ispis()` koja ispisuje vrijednosti postavljenih atributa.
2. Napišite objektno orijentirani program u Pythonu koji implementira razred `Student` sa sljedećim zahtjevima:

Metode:

- `__init__(self, ime, prezime)` – prima ime i prezime studenta te inicijalizira praznu listu u koju će biti spremljene sve ocjene toga studenta
- `dodajOcjenu(self, ocjena)` – dodaje novu ocjenu u listu ocjena
- `prosjekOcjena(self)` – vraća aritmetičku sredinu svih ocjena pojedinoga studenta, ako student nema ocjena, metoda mora vratiti 0
- `medijanOcjena(self)` – vraća medijan ocjena pojedinoga studenta, ako student nema ocjena, metoda mora vratiti 0
- `brojOcjena(self, o)` – prima ocjenu kao argument i vraća koliko se puta ta ocjena pojavljuje u listi ocjena nekoga studenta
- `__str__(self)` – vraća tekstualni prikaz objekta u formatu:
 Student: <ime i prezime>
 Ocjene: [<ocjena1>, <ocjena2>, ...]
 Prosjek ocjena: <prosjekOcjena>
 Medijan ocjena: <medijanOcjena>
- `ukupanBrojOcjena()` – vraća ukupan broj danih ocjena svim studentima

Atributi:

- ime
- prezime
- lista ocjena
- ukupan broj dodijeljenih ocjena svim studentima

Glavni program:

- Kreirati najmanje dva objekta razreda `Student`.
- Pomoću metode `dodajOcjenу()` unijeti između 4 i 8 proizvoljnih ocjena po studentu.
- Ispitati ispravnost svih metoda.

8.9. Vidljivost varijable – `public`, `protected`, `private`

Za razliku od drugih programskih jezika kao što su C++ ili Java, Python nema efektivan mehanizam koji bi mogao ograničiti programeru pristup varijablama koje se nalaze unutar razreda. Postoji konvencija prema kojoj se varijable označavaju kao javne (`public`), zaštićene (`protected`) ili privatne (`private`). Način korištenja ovom konvencijom objašnjen je u nastavku.

Vidljivosti omogućuju ostvarivanje enkapsulacije ili učahurivanja (engl. *encapsulation*). Pomoću ovih triju svojstava (`public`, `protected`, `private`) moguće je ograničiti neovlašten pristup metodama i atributima. U narednome poglavlju imena "Svojstva" opisano je kako se pristupa varijablama koje su označene kao privatne varijable nekoga razreda (pomoću postavljajućih i dobavljajućih metoda). Enkapsulacija podrazumijeva skrivanje atributa i implementacije od drugih razreda. Učahurivanjem se postiže slaba povezanost između objekata, čime objekti postaju neovisniji i interne promjene jednoga objekta ne utječu na rad drugoga objekta.

8.9.1. Javne varijable (engl. *public*)

Sve varijable u Pythonu, ako se ne navede drugačije, automatski postaju javne, što znači da su vidljive i dohvatljive iz svih dijelova programskoga kôda.

Primjer 1: U nastavku slijedi primjer programskoga kôda u kojemu se prikazuje korištenje javnom varijablom.

```
class Demo:

    def __init__(self, naziv):
        self.naziv = naziv
```

```

def postavi(self, vrijednost):
    self.naziv = vrijednost

def ispis(self):
    print(self.naziv)

def __str__(self):
    return self.naziv

d = Demo("AAA")
print(d)

d.naziv = "BBB"
print(d)

d.postavi("CCC")
print(d)

Izlaz:
    AAA
    BBB
    CCC

```

Gore prikazani programski kôd sastoji se od jedne varijable imena `naziv`. Korištena varijabla javna je, tj. dohvatljiva je iz svih dijelova programskoga kôda. Osim što je moguće dohvatiti njezinu vrijednost, tu javnu varijablu moguće je i mijenjati iz svih dijelova programskoga kôda.

8.9.2. Zaštićena varijabla (engl. *protected*)

Varijabli ovoga tipa pristupa se unutar razreda u kojemu je definirana te unutar njegovih naslijeđenih razreda (nasljeđivanje je objašnjeno u nastavku tečaja). Međutim, u Pythonu ne postoji stvarni mehanizam koji bi spriječio neovlašteni pristup varijablama označenima kao `protected`.

Umjesto toga rabi se ustaljena konvencija, pa je odgovornost na programeru da organizira programski kôd tako da se takvim varijablama pristupa isključivo unutar razreda u kojemu su definirane ili unutar njegovih naslijeđenih razreda.

Varijable `protected` označavaju se tako da se ispred imena varijable postavi znak `'_'`.

Primjer 2: U nastavku slijedi primjer programskoga kôda u kojemu se prikazuje korištenje zaštićenom varijablom.

```

class Demo:

    def __init__(self, naziv):
        self._naziv = naziv

    def postavi(self, vrijednost):
        self._naziv = vrijednost

```

VAŽNO!

Vanjsko pristupanje privatnim i zaštićenim atributima loša je praksa koju valja izbjegavati!

```

def ispis(self):
    print(self._naziv)

def __str__(self):
    return self._naziv

d = Demo("AAA")
print(d)

d._naziv = "BBB"
print(d)

d.postavi("CCC")
print(d)

```

Izlaz:

```

AAA
BBB
CCC

```

Kao što se može primijetiti, gornja dva primjera kôda (primjeri 1 i 2), napisana za zaštićenu i javnu varijablu, ne razlikuju se ni po čemu osim po načinu imenovanja varijable: `_naziv`. U primjeru zaštićene varijable, iako se prema konvenciji ne bi smjelo pristupiti toj varijabli izvan razreda `Demo`, varijabli `_naziv` i dalje je moguće pristupiti kao da je javna.

To znači da programer sam mora voditi računa o tome da programski kôd bude napisan tako da se zaštićenim varijablama pristupa isključivo unutar razreda ili njegovih naslijeđenih razreda jer Python ne provodi stvarnu zaštitu takvih varijabli.

8.9.3. Privatna varijabla (engl. *private*)

Privatnim varijablama moguće je pristupiti isključivo unutar razreda u kojemu su definirane. Njihova je svrha dodatno ograničiti izravan pristup izvan razreda i time povećati razinu enkapsulacije.

Sintaksa označavanja privatne varijable jest:

```
__imeVarijable
```

Ispred imena varijable dva se puta navodi znak `'_'`.

Primjer 3: U nastavku slijedi primjer programskoga kôda u kojemu se iz glavnoga dijela programa pokušava pristupiti privatnoj varijabli.

```

class Demo:

    def __init__(self, naziv):
        self.__naziv = naziv

d = Demo("Test")

print(d.__naziv)

```

```
Izlaz:
Traceback (most recent call last):
  File "D:\oop.py", line 7, in <module>
    print(d.__naziv)
          ^^^^^^^^^
AttributeError: 'Demo' object has no
attribute '__naziv'
```

Kao što se može vidjeti iz prethodnoga primjera, privatnoj varijabli imena `__naziv` nije moguće izravno pristupiti iz glavnoga dijela programa.

Postoji način kako pristupiti privatnoj varijabli iz glavnoga dijela programa, no takav se pristup ne preporučuje jer nije u skladu s konvencijom korištenja privatnim varijablama.

Sintaksa je: `imeObjekta._ImeRazreda__imeVarijable`

8.10. Svojstva

U Pythonu je iz glavnoga dijela programa moguće pristupiti svim varijablama objekta, bilo da su one javne, zaštićene ili privatne. Za razliku od nekih drugih programskih jezika, Python ne provodi strogu zabranu pristupa. Smisao zaštićenih i privatnih varijabli jest signalizirati programerima da su one dio unutarnje logike razreda te da im se ne bi trebalo izravno pristupati niti ih mijenjati.

Primjer 1: U nastavku slijedi primjer koji prikazuje kako je moguće zaštićenoj varijabli pristupiti izravno iz glavnoga dijela programskoga kôda.

```
class Osoba:

    def __init__(self, ime):
        self._ime = ime

o = Osoba("Ivica")
print(o._ime)

Izlaz:
Ivica
```

VAŽNO!

Glavni razlog za postojanje dobavljajućih i postavljajućih metoda jest taj da se, u slučaju promjene tipa atributa ili logike kojom se dobavljaju/postavljaju vrijednosti, ta promjena treba napraviti samo na jednom mjestu (unutar dobavljajuće ili postavljajuće metode), a ne na svakom mjestu gdje se u programskom kôdu pristupa tomu nekom atributu.

Ovaj način omogućuje bolju kontrolu programskoga kôda i lakše upravljanje

Ovakav način pristupanja varijabli imena `_ime` funkcionira, no ta je varijabla zaštićena varijabla i njoj se ne bi smjelo pristupati iz glavnoga programa. Jedan od načina kako ispraviti ovakvo pristupanje zaštićenim i privatnim varijablama jest pomoću dobavljajućih metoda (engl. *getter method*) i postavljajućih metoda (engl. *setter method*).

Primjer 2: Korištenje dobavljajuće i postavljajuće metode.

```
class Osoba:

    def __init__(self, ime):
        self._ime = ime

    # Dobavljajuća metoda
    def getIme(self):
        return self._ime
```

```

# Postavljajuća metoda
def setIme(self, ime):
    self._ime = ime

o = Osoba("Ivica")
print(o.getIme())
o.setIme("Perica")
print(o.getIme())

Izlaz:
    Ivica
    Perica

```

Kao što je vidljivo iz gornjega primjera, metoda `getIme()` vraća vrijednost koja se nalazi u zaštićenoj varijabli `_ime`, a metoda `setIme()` postavlja tu varijablu na novu vrijednost primljenu preko parametra.

Ovo je klasičan pristup u mnogim objektno orijentiranim jezicima, no Python nudi svoj način putem takozvanih svojstava koja implementiraju funkcionalnost dobavljaćih i postavljajućih metoda.

8.10.1. Dobavljaća metoda – `@property`

Dekorator `@property` implementira dobavljaću (engl. *getter*) metodu, koja vraća vrijednost pohranjenu u nekoj varijabli.

Sintaksa definiranja:

```

@property
def imeMetode(self):
    return self._imeVarijable

```

Sintaksa poziva:

```
imeObjekta.imeMetode
```

Iako je riječ o metodi, prilikom poziva dobavljaće metode nije potrebno navoditi zagrade, odnosno poziva se kao da je riječ o običnome atributu.

Primjer 1: Dobavljaća metoda.

```

class Osoba:
    def __init__(self, ime):
        self._ime = ime

    @property
    def ime(self):
        return self._ime

o = Osoba("Ivica")
print(o.ime)

Izlaz:
    Ivica

```

8.10.2. Postavljajuća metoda – @imeMetode.setter

Dekorator @imeMetode.setter implementira postavljajuću (engl. *setter*) metodu, odnosno metodu koja mijenja vrijednost atributa.

Sintaksa definiranja:

```
@imeMetode.setter
def imeMetode(self, vrijednost):
    self._imeVariable = vrijednost
```

Sintaksa poziva (iako je riječ o metodi, prilikom poziva postavljajuće metode nije potrebno navoditi zagrade, odnosno poziva se kao da je riječ o običnom atributu):

```
imeObjekta.imeMetode = <vrijednost>
```

Primjer 2: Dobavljajuća i postavljajuća metoda.

```
class Osoba:
    def __init__(self, ime):
        self._ime = ime

    @property
    def ime(self):
        return self._ime

    @ime.setter
    def ime(self, value):
        self._ime = value
```

```
o = Osoba("Ivica")
print(o.ime)
o.ime = "Perica"
print(o.ime)
```

Izlaz:

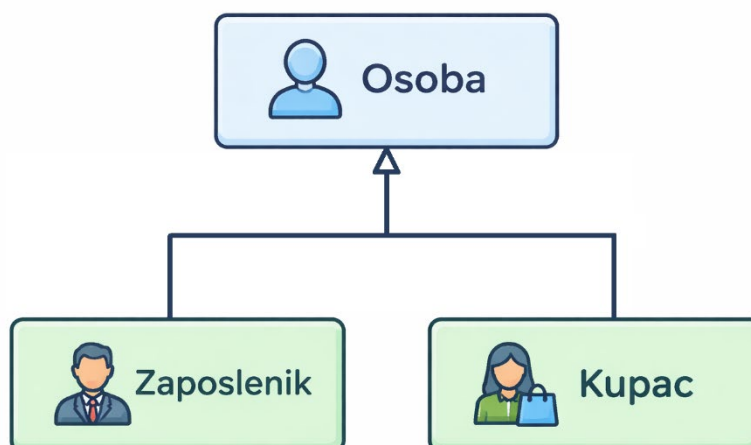
```
Ivica
Perica
```

8.11. Vježba: Objektno orijentirano programiranje III

1. Napišite razred imena `Osoba` te pomoću konstruktora tome razredu pridružite atribut `imeOsobe`. Tako kreirani atribut `imeOsobe` neka bude **javna** varijabla. U glavnome programu kreirajte jedan objekt razreda `Osoba` te iz njega ispišite vrijednost zapisanu u javnoj varijabli `imeOsobe` (bez korištenja dobavljajućom metodom). Nakon što ispišete vrijednost spremljenu u varijabli `imeOsobe`, unutar glavnoga programa promijenite tu vrijednost u neku drugu proizvoljnu vrijednost te ponovo ispišite vrijednost spremljenu u varijabli `imeOsobe`.
2. Prethodni zadatak prepravite tako da varijabla `imeOsobe` bude **zaštićena** varijabla. Unutar glavnoga programa ispišite njezinu vrijednost, nakon toga ju promijenite u neku drugu proizvoljnu vrijednost te ju ponovo ispišite. Postoji li razlika u ponašanju programa Python naspram 1. zadatka?
3. Prethodni zadatak prepravite tako da varijabla `imeOsobe` bude **privatna** varijabla. Unutar glavnoga programa ispišite njezinu vrijednost, nakon toga ju promijenite u neku drugu proizvoljnu vrijednost te ju ponovo ispišite. Postoji li razlika u ponašanju programa Python naspram 1. tj. 2. zadatka?
4. Prethodni zadatak prepravite tako da za **privatnu** varijablu `imeOsobe` napišete postavljajuću i dobavljajuću metodu bez korištenja dekoratorom. Unutar glavnoga programa kreirajte jedan objekt razreda `Osoba`. Pomoću postavljajuće i dobavljajuće metode pristupite varijabli `imeOsobe` te najprije ispišite njezinu vrijednost, zatim ju promijenite te ju opet ispišite.
5. Prethodni zadatak prepravite tako da ostvarite identičnu funkcionalnost, no ovoga se puta koristite dekoratorima za postavljajuću i dobavljajuću metodu.

8.12. Nasljeđivanje (engl. *inheritance*)

Jedno od ključnih svojstava objektno orijentiranoga programiranja jest ponovna iskoristivost već napisanoga programskog kôda. Jedan od najučinkovitijih načina iskorištavanja već napisanoga programskog kôda jest nasljeđivanje. Nasljeđivanje je osnovna funkcionalnost u objektno orijentiranim programskim jezicima. Nasljeđivanje omogućuje da razred specifičnih funkcionalnosti preuzme svojstva osnovnoga razreda. Nasljeđivanje je najlakše moguće zamisliti kao vezu između glavnoga razreda i njemu podređenoga razreda, pri čemu podređeni razred nasljeđuje osobine osnovnog. Radi lakšega razumijevanja u nastavku se nalazi grafički prikaz nasljeđivanja.



Neke od prednosti nasljeđivanja jesu:

- Ako je potrebno promijeniti ili dodati neki novi atribut ili metodu kojom su zahvaćeni svi izvedeni razredi, umjesto da se taj atribut, tj. metoda dodaje na nekoliko mjesta u programskome kôdu (po svim izvedenim razredima), potrebno ga je dodati samo u osnovni razred.
- Promjene u jednome izvedenom razredu ne utječu ni na jedan drugi izvedeni razred ili pak osnovni razred.
- Zbog mogućnosti korištenja programskim kôdom iz osnovnoga razreda nije potrebno taj isti kôd pisati (ili pak kopirati) još jednom, kao što bi bilo potrebno kad bismo imali neovisne razrede.

Primjer 1: U primjeru koji slijedi *Osoba* je **osnovni razred** (engl. *base class*, *superclass*), dok su *Zaposlenik* i *Kupac* **izvedeni razredi** (engl. *derived classes*, *subclasses*).

Razredu *Osoba* pridruženi su atributi `ime` i `prezime` te metode `ispis()` i `hodaaj()`. Razredi koji opisuju zaposlenika i kupca imaju dodatne attribute i metode specifične za njih, no istovremeno mogu pristupati i metodama osnovnoga razreda.

Razred `Zaposlenik` nasljeđuje razred `Osoba`. Uz atribute i metode iz razreda `Osoba` ovom razredu dodijeljen je dodatni atribut `placa` i dodatne metode `infoZaposlenik()` i `naplati()`. Ovaj razred može pristupati metodama iz osnovnoga razreda.

Razred `Kupac` također nasljeđuje razred `Osoba`. Uz atribute i metode iz razreda `Osoba` ovom razredu dodijeljen je dodatni atribut `tipKupca` i dodatna metoda `infoKupac()` i `kupi()`. Ovaj razred može pristupati metodama iz osnovnoga razreda.

Sintaksa kojom izvedeni razred nasljeđuje osnovni razred je:

```
class IzvedeniRazred(OsnovniRazred):
```

Konstruktor izvedenoga razreda nadjačava konstruktor osnovnog razreda, pa ga je potrebno eksplicitno pozvati. Sintaksa za eksplicitno pozivanje konstruktora osnovnog razreda jest:

```
OsnovniRazred.__init__(self, [param1], ...)
```

```
class Osoba:
    def __init__(self, ime, prezime):
        self.ime = ime
        self.prezime = prezime

    def ispis(self):
        print("Ime:", self.ime)
        print("Prezime:", self.prezime)
        print("-" * 30)

    def hodaj(self):
        print("Osoba hoda!")

class Zaposlenik(Osoba):
    def __init__(self, ime, prezime, placa):
        Osoba.__init__(self, ime, prezime)
        self.placa = placa

    def infoZaposlenik(self):
        print("Ime:", self.ime)
        print("Prezime:", self.prezime)
        print("Plaća:", self.placa)
        print("-" * 30)

    def naplati(self):
        print("Zaposelnik naplaćuje!")

class Kupac(Osoba):
    def __init__(self, ime, prezime, tipKupca):
        Osoba.__init__(self, ime, prezime)
        self.tipKupca = tipKupca

    def kupi(self):
        print("Kupac kupuje!")
```

```

def infoKupac(self):
    print("Ime:", self.ime)
    print("Prezime:", self.prezime)
    print("Tip kupca:", self.tipKupca)
    print("-" * 30)

o = Osoba("Pero", "Perić")
z = Zaposlenik("Ivo", "Ivić", 10000)
k = Kupac("Lovro", "Lovrić", "Fizički kupac")

print(">>> Klasa Osoba:")
o.ispis()
o.hodaj()

print("\n>>> Klasa Zaposlenik:")
z.infoZaposlenik()
z.ispis()
z.hodaj()
z.naplati()

print("\n>>> Klasa Kupac:")
k.infoKupac()
k.ispis()
k.hodaj()
k.kupi()

```

Izlaz:

```

>>> Klasa Osoba:
Ime: Pero
Prezime: Perić
-----
Osoba hoda!

>>> Klasa Zaposlenik:
Ime: Ivo
Prezime: Ivić
Plaća: 10000
-----
Ime: Ivo
Prezime: Ivić
-----
Osoba hoda!
Zaposelnik naplaćuje!

>>> Klasa Kupac:
Ime: Lovro
Prezime: Lovrić
Tip kupca: Fizički kupac
-----
Ime: Lovro
Prezime: Lovrić
-----
Osoba hoda!
Kupac kupuje!

```

Funkcija `super()`

Funkcija `super()` vraća objekt koji omogućuje pristup metodama sljedeće klase u hijerarhiji nasljeđivanja (osnovni razred). Također omogućuje ispravan rad s višestrukim nasljeđivanjem (višestruko nasljeđivanje nije obrađeno u ovom priručniku). U gore opisanome primjeru prilikom nasljeđivanja osnovnoga razreda sintaksa kojom se unutar konstruktora izvedenoga razreda poziva konstruktor osnovnoga razreda je:

```
def __init__(self, [param1], ...):
    OsnovniRazred.__init__(self, [param1], ...)
```

Elegantniji način za postavljanje atributa jest korištenjem sljedećom sintaksom:

```
def __init__(self, [param1], ...):
    super().__init__([param1], ...)
```

Primjer 2: Poziv konstruktora iz osnovnoga razreda korištenjem funkcije `super()`.

```
class Osoba:

    def __init__(self, ime, prezime):
        self.ime = ime
        self.prezime = prezime

class Zaposlenik(Osoba):

    def __init__(self, ime, prezime, placa):
        super().__init__(ime, prezime)
        self.placa = placa

    def infoZaposlenik(self):
        print("Ime:", self.ime)
        print("Prezime:", self.prezime)
        print("Plaća:", self.placa)

z = Zaposlenik("Pero", "Perić", 10000)
z.infoZaposlenik()

Izlaz:
    Ime: Pero
    Prezime: Perić
    Plaća: 10000
```

Prednost ovakvoga načina pisanja kôda jest ta da prilikom prosljeđivanja argumenata konstruktoru osnovnoga razreda nije potrebno eksplicitno navoditi parametar `self`. Također, kada se rabi `super()`, nije potrebno navoditi ime osnovnoga razreda, čime se olakšava njegova potencijalna promjena. U tome slučaju promjenu osnovnoga razreda potrebno je navesti samo na jednome mjestu, a ne na dvama mjestima.

8.13. Preopterećenje (engl. *overloading*)

Napomena

Za razliku od nekih drugih programskih jezika, preopterećenje u Pythonu nije preopterećenje u punom smislu toga termina.

Preopterećenje u Pythonu nije sasvim fleksibilno jer upotrebljava opcionalne parametre koji definiraju tijek izvršavanja programa.

Ako bi se preopterećenje u Pythonu napisalo kao, na primjer, u Javi, tako da se eksplicitno definiraju dvije metode s istim imenom i različitim potpisima, to bi rezultiralo korištenjem samo posljednjim definiranim metodama.

Napomena - None

None je objekt razreda NoneType.

- None podržava operatore `==` i `is`.
- None nije jednako kao i False.
- None nije 0.
- None nije prazni niz znakova.
- Uspoređujući None s bilo čime, vratit će se False, osim u slučaju ako je u objektu koji se uspoređuje s None pohranjeno None.

Preopterećenje metoda znači da dvije ili više metoda imaju isto ime, ali različit broj ili tip parametara. Međutim, u Pythonu ne postoji pravo preopterećenje metoda jer ako se definira više metoda s istim imenom, upotrebljavat će se samo posljednja definirana metoda. Efekt preopterećenja u Pythonu može se postići korištenjem zadanim (preddefiniranim) parametrima.

Ovdje se može primijetiti jedan od osnovnih pojmova OOP-a, a to je polimorfizam u kojemu je ista metoda implementirana i djeluje na različite načine.

Primjer 1: Razred s nekoliko preopterećenih metoda.

```
class Demo:

    def metoda(self, param1=None, param2=None):

        if param1 is None and param2 is None:
            print("Metoda bez parametara!")

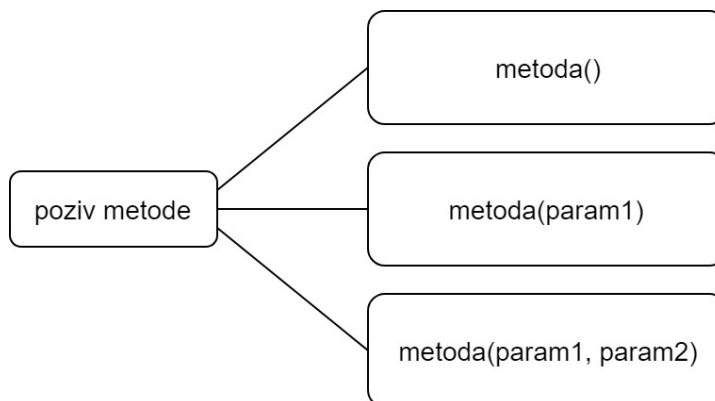
        elif param2 is None:
            print("Parametar:", param1)

        else:
            print("Parametri:", param1, "i",
                  param2)

d = Demo()
d.metoda()
d.metoda("A")
d.metoda("A", "B")

Izlaz:
Metoda bez parametara!
Parametar: A
Parametri: A i B
```

Preopterećenje se rabi kada se želi rabiti metoda jednoga imena, no s mogućnošću prenošenja različitoga broja argumenata.



Gore navedeni programski kôd ilustriran je gornjom slikom iz koje se vidi da imamo jednu metodu naziva `metoda()` i nju možemo pozvati na više različitih načina (u gornjem slučaju na 3 različita načina).

- Poziv metode bez argumenata:
Ako tu metodu pozovemo a da joj ne prenesemo nijedan argument, onda će varijable `param1` i `param2` poprimiti preddefiniranu vrijednost, odnosno `None`.
- Poziv metode s jednim argumentom:
U ovome slučaju varijabla `param1` poprimit će vrijednost prenesenoga argumenta, dok će varijabla `param2` poprimiti preddefiniranu vrijednost, odnosno `None`.
- Poziv metode s dvama argumentima:
U ovome slučaju obje varijable `param1` i `param2` poprimit će vrijednosti prenesenih argumenata.

8.14. Nadjačavanje (engl. *overriding*)

Metoda istoga imena u izvedenome razredu nadjačava metodu iz osnovnoga razreda ako je definirana u izvedenome razredu. Razlog nadjačavanja metode osnovnoga razreda jest promjena njezine funkcionalnosti. Također, i ovdje se primjenjuje jedan od osnovnih pojmova OOP-a, a to je polimorfizam, pri čemu ista metoda može imati različitu implementaciju i ponašanje u dvama različitim razredima.

Primjer 1: U donjem programu kreiran je objekt imena `z`, koji je instanca razreda `Zaposlenik`. Metoda `ispis()` implementirana je unutar razreda `Osoba`. Pozivom metode `ispis()` nad objektom `z`, koji pripada razredu `Zaposlenik`, poziva se metoda iz razreda `Zaposlenik` ako je nadjačana, u suprotnome se rabi metoda naslijeđena iz osnovnoga razreda `Osoba`.

```
class Osoba:
    def __init__(self, ime, prezime):
        self.ime = ime
        self.prezime = prezime

    def ispis(self):
        print("Ja sam:", self.ime, self.prezime)

class Zaposlenik(Osoba):
    def __init__(self, ime, prezime, placa):
        super().__init__(ime, prezime)
        self.placa = placa

z = Zaposlenik("Pero", "Perić", 10000)
z.ispis()

Izlaz:
Ja sam: Pero Perić
```

Primjer 2: Ako se nad objektom `z` želi pozvati metoda istoga imena kao `ispis()`, ali s drugačijom funkcionalnošću, onda se primjenjuje nadjačavanje metode (engl. *overriding*).

Ovaj princip primjenjuje se tako da se unutar izvedenoga razreda, u ovome slučaju razreda `Zaposlenik`, implementira metoda istoga imena kao ona koju se želi nadjačati, ali s drugačijom funkcionalnošću.

```
class Osoba:

    def __init__(self, ime, prezime):
        self.ime = ime
        self.prezime = prezime

    def ispis(self):
        print("Ja sam:", self.ime, self.prezime)

class Zaposlenik(Osoba):

    def __init__(self, ime, prezime, placa):
        super().__init__(ime, prezime)
        self.placa = placa

    def ispis(self):
        print("Ime:", self.ime)
        print("Prezime:", self.prezime)
        print("Plaća:", self.placa)

z = Zaposlenik("Pero", "Perić", 10000)
z.ispis()

Izlaz:
    Ime: Pero
    Prezime: Perić
    Plaća: 10000
```

8.15. Vježba: Objektno orijentirano programiranje IV

1. U ovome zadatku potrebno je simulirati nasljeđivanje. Kreirajte osnovni razred imena `Stavka` i dva razreda koji nasljeđuju osnovni razred, imena `Sok` i `Hrana`.

Osnovni razred `Stavka` mora sadržavati konstruktor i metodu `cijenaPdv()`. Konstruktor prima dva parametra, naziv i cijenu bez PDV-a. Metoda `cijenaPdv()` mora vratiti cijenu s PDV-om (25 %).

Razred `Sok` mora naslijediti osnovni razred te mora sadržavati konstruktor i metodu `ispis()`. Konstruktor prima tri parametara, naziv, cijenu i volumen pića. Primijetite da se naziv i cijena nalaze u osnovnome konstruktoru. Metoda `ispis()` mora ispisati naziv, cijenu (bez PDV-a) i volumen pića.

Razred `Hrana` mora naslijediti osnovni razred te mora sadržavati konstruktor i metodu `ispis()`. Konstruktor prima tri parametara, naziv, cijenu i broj kalorija hrane. Primijetite da se naziv i cijena nalaze u osnovnome konstruktoru. Metoda `ispis()` mora ispisati naziv, cijenu (bez PDV-a) i broj kalorija.

U glavnome programu kreirajte dva objekta, jedan razreda `Sok`, drugi razreda `Hrana`. Nad svakim objektom pozovite njegovu pripadajuću metodu `ispis()` te pomoću metode `cijenaPdv()` iz osnovnoga razreda ispišite cijenu pića ili hrane s PDV-om.

2. U ovome zadatku potrebno je simulirati nadjačavanje. Nadogradite prethodni zadatak ove vježbe tako da u izvedenim razredima implementirate metodu `cijenaPdv()` koja će nadjačati tu istu metodu iz osnovnoga razreda. Tako implementirana metoda za piće mora vratiti cijenu s porezom od 15 %, dok za hranu mora vratiti cijenu s porezom od 5 %.
3. U ovome zadatku potrebno je simulirati preopterećenje metode. Kreirajte razred imena `GeometrijskiLik`, on neka implementira jednu metodu razreda imena `opseg()` koja može primiti 0 parametara ili pak maksimalno 3 parametra. Ovisno o broju primljenih parametara ova metoda ispisuje pripadajući rezultat.

0 parametara – Ispišite poruku o grešci (Dogodila se greška!)

1 parametar – ispišite opseg kruga ($2 * r * \pi$)

2 parametra – ispišite opseg pravokutnika ($2*a + 2*b$)

3 parametra – ispišite opseg raznostraničnoga trokuta ($a+b+c$)

U glavnome programu pozovite tako implementiranu metodu razreda imena `opseg()` na sva 4 moguća načina.

8.16. Apstraktni razredi (engl. *abstract base classes*)

Apstraktni razredi upotrebljavaju se kada želimo definirati "pravila" koja svi izvedeni razredi moraju slijediti, bez definiranja konkretne implementacije. Apstraktni razred definira što neki razred mora raditi, ali ne i kako.

Apstraktni razred rabi se za:

- sprječavanje instanciranja osnovnog razreda (osnovni razred služi kao predložak i nema smisla stvarati njezine instance)
- osiguravanje da izvedeni razredi implementiraju sve apstraktne metode osnovnoga razreda
- definiranje zajedničkoga sučelja.

Kako bi se neki razred mogao proglasiti apstraktnim razredom, potrebno se koristiti modulom imena `abc` (engl. *abstract base classes*). Za proglašenje nekoga razreda apstraktnim u obliku zagradama uz ime razreda navodi se `ABC`, dok se metoda proglašava apstraktnom navođenjem dekoratora `@abstractmethod` ispred njezina prototipa.

```
from abc import ABC, abstractmethod

class ApstraktniRazred(ABC):

    @abstractmethod
    def apstraktnaMetoda(self):
        pass
```

Primjer 1: Razred `Oblik` nije moguće instancirati jer sadrži apstraktnu metodu.

```
from abc import ABC, abstractmethod

class Oblik(ABC):

    @abstractmethod
    def površina(self):
        pass

    def opis(self):
        return "Ja sam neki oblik"

o = Oblik()

Izlaz:
Traceback (most recent call last):
  File "D:\oop.py", line 13, in <module>
    o = Oblik()
TypeError: Can't instantiate abstract class
Oblik without an implementation for abstract
method 'brojStranica'
```

Primjer 2: U ovome primjeru definiramo apstraktni razred `Oblik` koji sadrži apstraktnu metodu `brojStranica()`. Svaki razred koji nasljeđuje osnovni razred `Oblik` mora implementirati tu metodu.

```
from abc import ABC, abstractmethod

class Oblik(ABC):

    @abstractmethod
    def brojStranica(self):
        pass

class Pravokutnik(Oblik):

    def brojStranica(self):
        print("Pravokutnik ima 4 stranice!")

class Trokut(Oblik):

    def brojStranica(self):
        print("Trokut ima 3 stranice!")

# Kreiranje objekata izvedenih razreda
p = Pravokutnik()
t = Trokut()

# Pozivi metoda
p.brojStranica()
t.brojStranica()

Izlaz:
    Pravokutnik ima 4 stranice!
    Trokut ima 3 stranice!
```

Izvedeni razred mora implementirati sve apstraktne metode definirane u osnovnome (apstraktnome) razredu. U slučaju da unutar izvedenih razreda (u gornjem primjeru unutar razreda `Trokut` ili `Pravokutnik`) nije implementirana metoda `brojStranica()`, Python će vratiti sljedeću grešku prilikom pokušaja stvaranja objekta:

```
TypeError: Can't instantiate abstract class Trokut
with abstract methods brojStranica.
```

Python provjerava jesu li sve apstraktne metode implementirane tek u trenutku instanciranja objekta, a ne prilikom definiranja razreda. Ako neka metoda nedostaje, instanciranje nije dozvoljeno, a korisniku se prikazuje iznimka `TypeError`. Takvim načinom rada osigurava se da svi izvedeni razredi imaju potpunu i ispravnu implementaciju koja je definirana u apstraktnome osnovnom razredu.

Primjer 3: Dinamički polimorfizam pri nasljeđivanju.

```

from abc import ABC, abstractmethod
import math

class GeometrijskiLik(ABC):

    @abstractmethod
    def ime(self):
        pass

    @abstractmethod
    def opseg(self):
        pass

    @abstractmethod
    def povrsina(self):
        pass

class Kvadrat(GeometrijskiLik):

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

    def ime(self):
        return "Kvadrat"

    def opseg(self):
        return 4 * self.a

    def povrsina(self):
        return self.a ** 2

class Krug(GeometrijskiLik):

    def __init__(self, r):
        self.r = r

    def ime(self):
        return "Krug"

    def opseg(self):
        return 2 * math.pi * self.r

    def povrsina(self):
        return math.pi * self.r ** 2

def ispisi(lik):
    print(f"Lik: {lik.ime()}")
    print(f"O = {lik.opseg():.2f}")
    print(f"P = {lik.povrsina():.2f}")
    print("-" * 20)

```

```
# Lista različitih objekata (polimorfizam)
likovi = [
    Kvadrat(5),
    Krug(10)
]

for lik in likovi:
    ispisi(lik)
```

```
Izlaz:
    Lik: Kvadrat
    O = 20.00
    P = 25.00
    -----
    Lik: Krug
    O = 62.83
    P = 314.16
    -----
```

U ovome primjeru prikazan je dinamički polimorfizam objekata pri nasljeđivanju. Razred `GeometrijskiLik` definiran je kao apstraktni razred koji sadrži tri apstraktne metode: `ime()`, `opseg()` i `povrsina()`. Te metode predstavljaju zajedničko sučelje koje svi izvedeni razredi moraju implementirati. Razredi `Kvadrat` i `Krug` nasljeđuju razred `GeometrijskiLik` te implementiraju sve njegove apstraktne metode na način specifičan za svaki pojedini lik.

Funkcija `ispisi(lik)` prima objekt (instancu) koji može biti bilo koji izvedeni razred (`Kvadrat`, `Krug` ili neki drugi razred koji nasljeđuje osnovni razred `GeometrijskiLik`). Unutar funkcije pozivaju se metode `ime()`, `opseg()` i `povrsina()` bez potrebe da se unaprijed zna o kojemu je točno tipu objekta riječ.

Dinamički polimorfizam omogućuje da se ista funkcija rabi za različite tipove objekata, pri čemu se odgovarajuća metoda poziva ovisno o stvarnome tipu objekta u vrijeme izvođenja programa.

8.17. Ulančavanje metoda

Ulančavanje metoda (engl. *method chaining*) tehnika je koja se rabi za uzastopno pozivanje više metoda nad istim objektom u jednoj naredbi, koristeći se referencom objekta samo jednom.

Ulančavanje metoda postiže se tako da sve metode koje sudjeluju u ulančavanju kao povratnu vrijednost vraćaju referencu na trenutačni objekt, odnosno `self`.

U nastavku slijede dva primjera programskoga kôda. U prvome nije implementirano ulančavanje metoda, dok je u drugome ono implementirano.

Primjer 1: Programski kôd bez ulančavanja metoda.

```

class Gradovi:
    def __init__(self, n=None):
        self.naziv = n

    def ispis(self):
        print(self.naziv)

    def zagreb(self):
        self.naziv = "Zagreb"

    def split(self):
        self.naziv = "Split"

g = Gradovi()

g.zagreb()
g.ispis()
g.split()
g.ispis()
Izlaz:
    Zagreb
    Split

```

U ovome primjeru svaka od metoda – `zagreb()`, `split()`, `ispis()` poziva se zasebno nad objektom `g` tako da se povratne vrijednosti tih metoda ne rabe.

Primjer 2: Programski kôd s ulančavanjem metoda.

```

class Gradovi:
    def __init__(self, n=None):
        self.naziv = n

    def ispis(self):
        print(self.naziv)
        return self

    def zagreb(self):
        self.naziv = "Zagreb"
        return self

    def split(self):
        self.naziv = "Split"
        return self

g = Gradovi()
g.zagreb().ispis().split().ispis()
Izlaz:
    Zagreb
    Split

```

Razred `Gradovi` sadrži konstruktor, metodu `ispis()` koja ispisuje vrijednost atributa `naziv` te dvije metode `zagreb()` i `split()` koje

postavljaju novu vrijednost atributa `naziv`. Ove tri metode (bez konstruktora) kao povratnu vrijednost vraćaju referencu na isti objekt tj. `self`, čime se omogućuje pozivanje sljedeće metode nad rezultatom prethodne.

Napomena: Ako neka metoda u lancu ne vraća `self`, ulančavanje će biti prekinuto.

Opis izvršavanja po koracima:

1. korak:

```
g.zagreb().ispis().split().ispis()
```

Izvršava se `g.zagreb()`, unutar metode `zagreb()` najprije se atribut `naziv` postavlja na vrijednost "Zagreb" te se vraća `self`. Svaka metoda vraća `self`, što omogućuje nastavak ulančavanja. U objašnjenju po koracima povratni objekt `self` označit ćemo kao `obj`.

2. korak:

```
obj.ispis().split().ispis()
```

Izvršava se `obj.ispis()`, unutar metode `ispis()` ispisuje se atribut `naziv` te se vraća `self`.

3. korak:

```
obj.split().ispis()
```

Izvršava se `obj.split()`, unutar metode `split()` najprije se atribut `naziv` postavlja na vrijednost "Split" te se vraća `self`.

4. korak:

```
obj.ispis()
```

Izvršava se `obj.ispis()`, unutar metode `ispis()` ispisuje se atribut `naziv` te se vraća `self`.

Sažetak:

- Ulančavanje metoda omogućuje višestruke pozive nad istim objektom u jednoj naredbi.
- Sve metode koje sudjeluju u lancu moraju vraćati `self`.
- Ako neka metoda ne vraća `self`, lanac se prekida.
- Ova tehnika poboljšava čitljivost i smanjuje ponavljanje reference na objekt.

8.18. Vježba: Objektno orijentirano programiranje V

1. Napišite apstraktni razred imena `TransportnoSredstvo` koji definira dvije apstraktne metode: `brojPutnika()` koja treba vratiti broj putnika za koje je transportno sredstvo predviđeno i `vrstaGoriva()` koja vraća vrstu goriva ili energije kojom se sredstvo koristi. Nakon toga implementirajte dva izvedena razreda: `Automobil` i `Bicikl`, oni nasljeđuju osnovni razred `TransportnoSredstvo`. U glavnome programu kreirajte po jedan objekt svakoga razreda i ispišite informacije o broju putnika i vrsti goriva ili energije za oba transportna sredstva.
2. Nadogradite prethodni zadatak tako da se za ispis informacija o broju putnika i vrsti goriva ili energije za oba transportna sredstva rabi dinamički polimorfizam pri nasljeđivanju. Napišite funkciju koja prima objekt tipa `TransportnoSredstvo` i unutar sebe poziva metode `brojPutnika()` i `vrstaGoriva()` kako bi ispisala informacije o transportnome sredstvu. Kreirajte listu objekata i dodajte u nju instance razreda `Automobil` i `Bicikl` iz prethodnoga zadatka te pozovite funkciju za svaki objekt u listi.
3. Napišite razred `Racun` koji predstavlja bankovni račun. Razred treba sadržavati atribut `stanje` koji predstavlja trenutno stanje računa. Implementirajte metode `dodajIznos()`, `oduzmiIznos()` i `ispisiStanje()`. Metode `dodajIznos()` i `oduzmiIznos()` trebaju primiti jedan argument (`iznos`) i odgovarajuće mijenjati stanje računa, dok metoda `ispisiStanje()` ispisuje trenutno stanje računa. Sve metode, uključujući `ispisiStanje()`, trebaju vraćati `self` kako bi se omogućilo ulančavanje metoda.

8.19. Pitanja za ponavljanje: Objektno orijentirano programiranje

1. Je li za kompleksnije programe, bolji pristup uporaba proceduralnoga programiranja ili objektno orijentiranoga programiranja?
2. Kako se zove osnovna cjelina u objektno orijentiranome programiranju?
3. Kako se naziva specijalna metoda koja se automatski pokreće prilikom instanciranja objekta?
4. Mora li obvezno u svakoj klasi postojati konstruktor?
5. Čemu služi parametar `self`?
6. Koja je razlika između varijable objekta i varijable razreda?
7. Kada je riječ o vidljivosti varijabli, koja tri tipa vidljivosti postoje?

8. Koja je razlika između javne varijable i privatne varijable?
9. Što je nasljeđivanje?
10. Što je preopterećenje?
11. Što je nadjačavanje?

Dodatak: Rješenja vježbi

2.8.1.

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

a, b, c, d = lista

print(a, b, c, d)
```

Izlaz:

```
1 2 3 4
```

2.8.2.

```
a = 5
b = 10

print(a, b)

a, b = b, a

print(a, b)
```

Izlaz:

```
5 10
10 5
```

2.8.3.

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

broj = int(broj)

if 10 < broj < 30:
    print("Zadovoljava!")
else:
    print("Ne zadovoljava!")
```

Izlaz 1:

```
Unesite broj: 55
Ne zadovoljava!
```

Izlaz 2:

```
Unesite broj: 22
Zadovoljava!
```

2.8.4.

```

lista = [1, 2, 3, 4, 5, 6, 7]

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

for e in lista:
    if e == broj:
        print("Postoji!")
        break
else:
    print("Ne postoji!")

```

```

Izlaz 1:
    Unesite broj: 2
    Postoji!

```

```

Izlaz 2:
    Unesite broj: 11
    Ne postoji!

```

2.8.5.

```

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

print("Paran!" if broj % 2 == 0 else "Neparan!")

```

```

Izlaz 1:
    Unesite broj: 1
    Neparan!

```

```

Izlaz 2:
    Unesite broj: 2
    Paran!

```

2.8.6.

```

def paran():
    print("Paran!")

def neparan():
    print("Neparan!")

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

(paran if broj % 2 == 0 else neparan)()

```

```

Izlaz 1:
    Unesite broj: 1
    Neparan!

```

```

Izlaz 2:
    Unesite broj: 2
    Paran!

```

2.8.7.

```
def rezultat(var1, var2):
    a = var1 * var2
    b = var1 + var2
    c = 10 * var1 + var2

    return a, b, c

var1, var2, var3 = rezultat(5, 10)

print(var1, var2, var3)

Izlaz:
    50 15 60
```

2.8.8.

```
nizZnakova = "Hello World!"
lista = [1, 2, 3, 4]

print("Niz znakova (petlja): ", end="")
i = len(nizZnakova) - 1
while i >= 0:
    print(nizZnakova[i], sep="", end="")
    i -= 1

print("\nLista (petlja): ", end="")
i = len(lista) - 1
while i >= 0:
    print(lista[i], sep="", end="")
    i -= 1

print("Niz znakova:", nizZnakova[::-1])
print("Lista:", lista[::-1])

Izlaz:
    Niz znakova (petlja): !dlroW olleH
    Lista (petlja): 4321
    Niz znakova: !dlroW olleH
    Lista: [4, 3, 2, 1]
```

3.2.1.

```
def zbrojiBrojeveDo(x):
    if x == 1:
        return 1
    else:
        return x + zbrojiBrojeveDo(x - 1)

zbroj = zbrojiBrojeveDo(5)
print(zbroj)
```

Izlaz:

15

3.2.2.

```
def brojParnihZnamenki(x):
    if x == 0:
        return 0
    elif x % 2 == 0:
        return 1 + brojParnihZnamenki(x // 10)
    else:
        return brojParnihZnamenki(x // 10)

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

broj = brojParnihZnamenki(x)
print(broj)
```

Izlaz:

Unesite broj: **123456**

3

3.2.3.

```
def prebroji(broj, znamenka):
    if broj == 0:
        return 0
    elif (broj % 10) == znamenka:
        return 1 + prebroji(broj // 10, znamenka)
    else:
        return prebroji(broj // 10, znamenka)

rezultat = prebroji(1223455, 2)
print(rezultat)
```

Izlaz:

2

3.4.1.

```
fun = lambda a, b, c: (a * b + c) / c

rezultat = fun(5, 10, 2)

print(rezultat)
Izlaz:
    26.0
```

3.4.2.

```
fun = lambda a, b, c: (a*b+c) / c if (c != 0) else
"Err!"

rezultat = fun(5, 10, 0)

print(rezultat)
Izlaz:
    Greška u predanim podacima!
```

3.4.3.

```
celsius = [0, 5, 10, 15.8]

fahrenheit = map(lambda x: (float(9) / 5) * x + 32,
celsius)

print(list(fahrenheit))
Izlaz:
    [32.0, 41.0, 50.0, 60.44]
```

3.6.1.

```
def izracun(a, b):
    def parniZbroj(a, b):
        return 2 * a + 5 * b

    def neparniZbroj(a, b):
        return a * b - 10

    if (a + b) % 2 == 0:
        return parniZbroj(a, b)
    else:
        return neparniZbroj(a, b)

rezultat = izracun(5, 15)
print(rezultat)
Izlaz:
    85
```

3.8.1.

```
def danUTjednu():
    dani = ["Ponedjeljak",
            "Utorak",
            "Srijeda",
            "Četvrtak",
            "Petak",
            "Subota",
            "Nedjelja"]

    for e in dani:
        yield e
```

```
g = danUTjednu()

for e in g:
    print(e)
```

Izlaz:

```
Ponedjeljak
Utorak
Srijeda
Četvrtak
Petak
Subota
Nedjelja
```

3.8.2.

```
def jedanDvaTri():
    brojac = ["Jedan", "Dva", "Tri"]

    while True:
        for e in brojac:
            yield e
```

```
g = jedanDvaTri()
print(next(g))
print(next(g))
print(next(g))
print(next(g))
print(next(g))
print(next(g))
print(next(g))
print(next(g))
print(next(g))
```

Izlaz:

```
Jedan
Dva
Tri
Jedan
Dva
Tri
Jedan
Dva
```

3.8.3.

```

def bottles(brojBoca=99, nazivNapitka="beer"):
    brojBocaNaPocetku = brojBoca

    while brojBoca >= 0:

        if brojBoca >= 3:
            yield (str(brojBoca) + " bottles of " +
                    nazivNapitka + " on the wall, " +
                    str(brojBoca) + " bottles of " +
                    nazivNapitka + ".\n" +
                    "Take one down and pass it around, " +
                    str(brojBoca - 1) + " bottles of " +
                    nazivNapitka + " on the wall.\n")

        elif brojBoca == 2:
            yield (str(brojBoca) + " bottles of " +
                    nazivNapitka + " on the wall, " +
                    str(brojBoca) + " bottles of " +
                    nazivNapitka + ".\n" +
                    "Take one down and pass it around, " +
                    str(brojBoca - 1) + " bottle of " +
                    nazivNapitka + " on the wall.\n")

        elif brojBoca == 1:
            yield (str(brojBoca) + " bottle of " +
                    nazivNapitka + " on the wall, " +
                    str(brojBoca) + " bottle of " +
                    nazivNapitka + ".\n" +
                    "Take one down and pass it around, " +
                    "no more bottles of " +
                    nazivNapitka + " on the wall.\n")

        else:
            yield ("No more bottles of " + nazivNapitka +
                    " on the wall, no more bottles of " +
                    nazivNapitka + ".\n" +
                    "Go to the store and buy some more, "
                    + str(brojBocaNaPocetku) +
                    " bottles of " +
                    nazivNapitka + " on the wall.")

        brojBoca -= 1

g = bottles(4)

for e in g:
    print(e)

```

Izlaz:

```

4 bottles of beer on the wall, 4 bottles of
beer.
Take one down and pass it around, 3 bottles of
beer on the wall.

3 bottles of beer on the wall, 3 bottles of
beer.
Take one down and pass it around, 2 bottles of
beer on the wall.

```

```
2 bottles of beer on the wall, 2 bottles of
beer.
Take one down and pass it around, 1 bottle of
beer on the wall.

1 bottle of beer on the wall, 1 bottle of
beer.
Take one down and pass it around, no more
bottles of beer on the wall.

No more bottles of beer on the wall, no more
bottles of beer.
Go to the store and buy some more, 4 bottles
of beer on the wall.
```

3.9.1.

Rekurzivne funkcije jesu funkcije koje pozivaju same sebe.

3.9.2.

Rekurzivni poziv funkcije nije efikasniji od iterativnoga rješenja.

3.9.3.

Rekurzivni poziv funkcije nije efikasniji od iterativnoga rješenja jer svaki poziv funkcije uključuje operacije kao što su: adresiranje te postavljanje argumenata i povratnih vrijednosti na stog.

3.9.4.

Funkcija će sama sebe rekurzivno pozivati beskonačno puta (u stvarnosti dubina rekurzije ograničena je veličinom memorije).

3.9.5.

Bezimena funkcija jest funkcija koja je definirana bez imena.

3.9.6.

Drugi naziv za bezimenu funkciju jest lambda-funkcija.

3.9.7.

Sintaksa bezimena funkcije jest:

```
lambda [argumenti] : [izraz]
```

3.9.8.

Bezimena funkcija može primiti neograničenu količinu argumenata.

3.9.9.

Da, bezimenu je funkciju moguće spremiti u objekt, tj. varijablu.

3.9.10.

Ugniježdene funkcije jesu funkcije koje su definirane unutar drugih funkcija.

3.9.11.

Ugniježdene funkcije najčešće se rabe kako bi se postiglo skrivanje ponašanja nekoga dijela programskoga kôda od ostatka programa.

3.9.12.

Klasične funkcije	Generatorske funkcije
koriste se ključnom riječju <code>return</code>	koriste se ključnom riječju <code>yield</code>
vraćaju vrijednost	vraćaju generator

4.3.1.

<pre>unos = input("Unesite broj: ") broj = int(unos) print(broj)</pre>
Izlaz 1: <pre>Unesite broj: 5 5</pre>
Izlaz 2: <pre>Unesite broj: broj Traceback (most recent call last): File "iznimke.py", line 2, in <module> broj = int(unos) ValueError: invalid literal for int() with base 10: 'broj'</pre>

4.3.2.

<pre>try: unos = input("Unesite broj: ") broj = int(unos) print(broj) except: print("Iznimka je uhvaćena!")</pre>
Izlaz 1: <pre>Unesite broj: 5 5</pre>
Izlaz 2: <pre>Unesite broj: broj Iznimka je uhvaćena!</pre>

4.3.3.

<pre>try: a = int(input("Unesite vrijednost a: ")) b = int(input("Unesite vrijednost b: ")) c = int(input("Unesite vrijednost c: ")) rezultat = (a + b) / c print(rezultat)</pre>

```

except ValueError:
    print("Iznimka - ValueError")
except ZeroDivisionError:
    print("Iznimka - ZeroDivisionError")
except:
    print("Ostale iznimke!")

```

```

Izlaz 1:
Unesite vrijednost a: 1
Unesite vrijednost b: broj
Iznimka - ValueError

```

```

Izlaz 2:
Unesite vrijednost a: 1
Unesite vrijednost b: 2
Unesite vrijednost c: 0
Iznimka - ZeroDivisionError

```

```

Izlaz 3:
Unesite vrijednost a: 1
Unesite vrijednost b: ^C #(Ctrl+C)
Ostale iznimke!

```

4.3.4.

```

try:
    a = int(input("Unesite vrijednost a: "))
    b = int(input("Unesite vrijednost b: "))
    c = int(input("Unesite vrijednost c: "))

    rezultat = (a + b) / c
    print(rezultat)
except ValueError as e:
    print("Iznimka - ValueError, opis:", e)
except ZeroDivisionError as e:
    print("Iznimka - ZeroDivisionError, opis:", e)
except:
    print("Ostale iznimke!")

```

```

Izlaz 1:
Unesite vrijednost a: 1
Unesite vrijednost b: broj
Iznimka - ValueError, opis: invalid literal
for int() with base 10: 'broj'

```

```

Izlaz 2:
Unesite vrijednost a: 1
Unesite vrijednost b: 2
Unesite vrijednost c: 0
Iznimka - ZeroDivisionError, opis: division by
zero

```

4.9.1.

```

def ucitajPaZbroji():
    try:
        a = int(input("Unesite vrijednost a: "))
        b = int(input("Unesite vrijednost b: "))
        c = int(input("Unesite vrijednost c: "))

```

```

    rezultat = a + b + c
    print("Zbroj je:", rezultat)
except:
    print("Iznimka je uhvaćena u funkciji!")
    raise

try:
    ucitajPaZbroji()
except:
    print("Iznimka je uhvaćena u glavnom programu!")

```

Izlaz 1:

```

    Unesite vrijednost a: 1
    Unesite vrijednost b: 2
    Unesite vrijednost c: 3
    Zbroj je: 6

```

Izlaz 2:

```

    Unesite vrijednost a: broj
    Iznimka je uhvaćena u funkciji!
    Iznimka je uhvaćena u glavnom programu!

```

4.9.2.

```

def ucitajPaZbroji():
    try:
        a = int(input("Unesite vrijednost a: "))
        b = int(input("Unesite vrijednost b: "))
        c = int(input("Unesite vrijednost c: "))

        rezultat = a + b + c
    except:
        print("Iznimka je uhvaćena u funkciji!")
        raise
    else:
        print(rezultat)
    finally:
        print("Finally blok!")

try:
    ucitajPaZbroji()
except:
    print("Iznimka je uhvaćena u glavnom programu!")

```

Izlaz 1:

```

    Unesite vrijednost a: 1
    Unesite vrijednost b: 2
    Unesite vrijednost c: 3
    6
    Finally blok!

```

Izlaz 2:

```

    Unesite vrijednost a: broj
    Iznimka je uhvaćena u funkciji!
    Finally blok!
    Iznimka je uhvaćena u glavnom programu!

```

4.9.3.

```

class NeispravnaOcjenaError(Exception):
    """Iznimka koja se podiže kada unesena ocjena
    nije ispravna."""
    pass

def provjeriOcjenu(ocjena):
    if ocjena not in [1, 2, 3, 4, 5]:
        raise NeispravnaOcjenaError(f"Neispravna
ocjena: {ocjena}")
    print(f"Ocjena je ispravna: {ocjena}")

brojIspravnih = 0
brojNeispravnih = 0

while True:
    unos = input("Unesi ocjenu (1-5): ")
    try:
        ocjena = int(unos)
        provjeriOcjenu(ocjena)
        brojIspravnih += 1
    except NeispravnaOcjenaError as e:
        print("Uhvaćena vlastita iznimka:", e)
        brojNeispravnih += 1
    except ValueError:
        print("Unos nije cijeli broj. Kraj unosa.")
        break

print("\nRezultat:")
print("Broj ispravnih ocjena:", brojIspravnih)
print("Broj neispravnih ocjena:", brojNeispravnih)

```

Izlaz 1:

```

Unesi ocjenu (1-5): 1
Ocjena je ispravna: 1
Unesi ocjenu (1-5): 2
Ocjena je ispravna: 2
Unesi ocjenu (1-5): 3
Ocjena je ispravna: 3
Unesi ocjenu (1-5): 4
Ocjena je ispravna: 4
Unesi ocjenu (1-5): 5
Ocjena je ispravna: 5
Unesi ocjenu (1-5): 6
Uhvaćena vlastita iznimka: Neispravna ocjena:
6
Unesi ocjenu (1-5): x
Unos nije cijeli broj. Kraj unosa.

```

```

Rezultat:
Broj ispravnih ocjena: 5
Broj neispravnih ocjena: 1

```

Izlaz 2:

```

Unesi ocjenu (1-5): 5
Ocjena je ispravna: 5

```

```
Unesi ocjenu (1-5): 4.9
Unos nije cijeli broj. Kraj unosa.

Rezultat:
Broj ispravnih ocjena: 1
Broj neispravnih ocjena: 0
```

4.10.1.

Sintaksne, semantičke i logičke greške.

4.10.2.

U Pythonu sve iznimke nasljeđuju ugrađeni osnovni razred `BaseException`.

4.10.3.

U slučaju da korištena varijabla nije definirana, podiže se iznimka `NameError`.

4.10.4.

Osnovni blok za obradu iznimaka tvore ključne riječi `try` i `except`.

4.10.5.

Prošireni blok za obradu iznimaka tvore ključne riječi `else` i/ili `finally`.

4.10.6.

Unutar `except` bloka iznimke se hvataju i obrađuju.

4.10.7.

Blok `else` izvodi se u slučaju kada se unutar bloka `try` ne dogodi iznimka ili ako se on ne završi pozivom naredbe `return`, `break` ili `continue`.

4.10.8.

Blok `finally` uvijek se izvršava prije završetka obrade iznimke (dogodila se iznimka ili ne).

5.2.1.

```

def ispis(sudionici, rezultati):
    i = 0
    for s in sudionici:
        print(s, ":", rezultati[i])
        i += 1

sud = ["Ivica", "Marica", "Perica", "Nika",
        "Nikica", "Tihana", "Stjepan", "Petra"]
rez = [10.11, 11.58, 10.08, 10.06, 10.77, 11.22,
        11.55, 10.05]

print("Prije sortiranja: ")
ispis(sud, rez)

brojSudionika = len(rez)
for i in range(brojSudionika):

    minIndex = i
    for j in range(i + 1, brojSudionika):
        if rez[j] < rez[minIndex]:
            minIndex = j

    rez[i], rez[minIndex] = rez[minIndex], rez[i]
    sud[i], sud[minIndex] = sud[minIndex], sud[i]

print("\nNakon sortiranja: ")
ispis(sud, rez)

```

Izlaz:

```

Prije sortiranja:
Ivica : 10.11
Marica : 11.58
Perica : 10.08
Nika : 10.06
Nikica : 10.77
Tihana : 11.22
Stjepan : 11.55
Petra : 10.05

```

```

Nakon sortiranja:
Petra : 10.05
Nika : 10.06
Perica : 10.08
Ivica : 10.11
Nikica : 10.77
Tihana : 11.22
Stjepan : 11.55
Marica : 11.58

```

5.5.1.

```

def ispisi(sud, rez):
    i = 0
    for s in sud:
        print(s, ":", rez[i])
        i += 1

sud = ["Ivica", "Marica", "Perica", "Nika",
        "Nikica", "Tihana", "Stjepan", "Petra"]
rez = [10.11, 11.58, 10.08, 10.06, 10.77, 11.22,
        11.55, 10.05]

print("Prije sortiranja: ")
ispisi(sud, rez)

brojElemenata = len(rez)
for i in range(brojElemenata - 1):

    zamjena = False
    for j in range(brojElemenata - 1 - i):
        if rez[j + 1] < rez[j]:
            rez[j], rez[j + 1] = rez[j + 1], rez[j]
            sud[j], sud[j + 1] = sud[j + 1], sud[j]
            zamjena = True

    if zamjena == False:
        break

print("\nNakon sortiranja: ")
ispisi(sud, rez)

```

Izlaz:

```

Prije sortiranja:
Ivica : 10.11
Marica : 11.58
Perica : 10.08
Nika : 10.06
Nikica : 10.77
Tihana : 11.22
Stjepan : 11.55
Petra : 10.05

```

```

Nakon sortiranja:
Petra : 10.05
Nika : 10.06
Perica : 10.08
Ivica : 10.11
Nikica : 10.77
Tihana : 11.22
Stjepan : 11.55
Marica : 11.58

```

5.7.1.

Ugrađenom metodom `sort()` nije se moguće koristiti u slučaju da postoje, na primjer, dvije liste, u jednoj listi nalaze se imena osoba, a u drugoj listi nalaze se rezultati natjecanja (podaci su povezani preko zajedničkoga indeksa). U tome slučaju potrebno je ručno implementirati algoritam koji će istovremeno sortirati obje liste.

5.7.2.

Na početku ovoga algoritma pronalazi se najmanja vrijednost, tako pronađena najmanja vrijednost stavlja se na prvo mjesto, nakon toga traži se sljedeća najmanja vrijednost koja se stavlja na drugo mjesto i tako dalje sve do trenutka kada niz bude sortiran.

5.7.3.

Ovaj algoritam prolazi kroz niz te uspoređuje parove vrijednosti, tj. susjedne elemente. U slučaju da su ti elementi (susjedi) u neispravnome poretku, njihove se vrijednosti mijenjaju. Algoritam kroz nesortirani dio niza prolazi sve dok niz ne postane sortiran.

5.7.4.

Algoritam zamjene susjednih elemenata moguće je poboljšati tako da se uvede dodatna varijabla u koju će biti zapisano je li se unutar nekoga prolaza dogodila zamjena ili ne. Ako unutar nekoga prolaza po nesortiranome dijelu niza nema zamjene, niz se proglašava sortiranim te se pomoću naredbe `break` prekida daljnje uspoređivanje susjednih elemenata.

6.3.1.

Obrađeno u poglavlju 6.1.1. *Postavljanje PATH varijabli okoline*

6.3.2.

Obrađeno u poglavlju 6.1.4. *Primjer instalacije paketa – requests*

6.3.3.

```
import requests

# Slanje GET zahtjeva
response = requests.get("https://www.srce.unizg.hr/")

# Provjera statusnog koda
print("Statusni kod:", response.status_code)

# Provjera sadržaja web-stranice
print("Sveučilišni računski centar" in response.text)
```

Izlaz:

True

6.3.4.

```
def funkcijaIspisuje():
    print("Ispis!")
```

moj.py

```
import moj

moj.funkcijaIspisuje()
```

Izlaz:

Ispis!

program.py

7.5.1.

CSV datoteka (7.5.1.csv):

```
Ime;Prezime;Grad;Godine
Ivan;Horvat;Zagreb;18
Pero;Novak;Split;19
Josip;Boban;Rijeka;20
Luka;Turk;Osijek;21
```

```
import csv

with open("7.5.1.csv", "r") as f:
    csvCitac = csv.reader(f, delimiter=';')

    header = next(csvCitac)

    for zapis in csvCitac:
        print(f"{header[0]}: {zapis[0]}")
        print(f"{header[1]}: {zapis[1]}")
        print(f"{header[2]}: {zapis[2]}")
        print(f"{header[3]}: {zapis[3]}\n")
```

Izlaz:

```
Ime: Ivan
Prezime: Horvat
Grad: Zagreb
Godine: 18
```

```
Ime: Pero
Prezime: Novak
Grad: Split
Godine: 19
```

```
Ime: Josip
Prezime: Boban
Grad: Rijeka
Godine: 20
```

```
Ime: Luka
Prezime: Turk
Grad: Osijek
Godine: 21
```

7.5.2.

CSV datoteka (7.5.2.csv):

```
Ime;Prezime;Grad;Godine
Ivan;Horvat;Zagreb;18
Pero; Novak;Split;19
Josip; Boban; Rijeka;20
Luka;Turk;Osijek;21
```

```
import csv

with open("7.5.2.csv", "r") as f:
    csvCitac = csv.reader(f,
                           delimiter=';',
                           skipinitialspace=True)

    header = next(csvCitac)

    for zapis in csvCitac:
        print(f"{header[0]}: {zapis[0]}")
        print(f"{header[1]}: {zapis[1]}")
        print(f"{header[2]}: {zapis[2]}")
        print(f"{header[3]}: {zapis[3]}\n")
```

```
Izlaz:
    Ime: Ivan
    Prezime: Horvat
    Grad: Zagreb
    Godine: 18

    Ime: Pero
    Prezime: Novak
    Grad: Split
    Godine: 19

    Ime: Josip
    Prezime: Boban
    Grad: Rijeka
    Godine: 20

    Ime: Luka
    Prezime: Turk
    Grad: Osijek
    Godine: 21
```

7.5.3.

```
import csv

with open('7.5.3.csv', mode='w') as f:
    csvPisac = csv.writer(f,
                           delimiter=';',
                           doublequote=False,
                           escapechar="\\",
                           lineterminator="KRAJ\n",
                           quotechar="&",
                           quoting=csv.QUOTE_ALL)

    csvPisac.writerow(
        ['Ime', 'Prezime', 'Grad', "PBR"])
    csvPisac.writerow(
        ['Ivan', 'Horvat', 'Zagreb', 10000])
    csvPisac.writerow(
        ['Pero', 'Novak', 'Split', 21000])
    csvPisac.writerow(
        ['Josip', 'Boban', 'Rijeka', 51000])
```

```
csvPisac.writerow(  
    ['Luka', 'Turk', 'Osijek, 31000'])
```

Izlaz:

```
&Ime&;&Prezime&;&Grad, "PBR"&KRAJ  
&Ivan&;&Horvat&;&Zagreb, 10000&KRAJ  
&Pero&;&Novak&;&Split, 21000&KRAJ  
&Josip&;&Boban&;&Rijeka, 51000&KRAJ  
&Luka&;&Turk&;&Osijek, 31000&KRAJ
```

7.6.1.

Pojedini elementi u CSV dokumentu mogu se odvojiti pomoću točke sa zarezom (;), zareza (,), tabulatora (\t), dvotočke (:) i tako dalje.

7.6.2.

Modul u kojemu se nalaze metode za obrađivanje CSV dokumenta zove se `csv`.

7.6.3.

Metoda pomoću koje se čita CSV dokument zove se `reader()`, a pomoću metode `writer()` omogućeno je pisanje u CSV dokument.

7.6.4.

Parametar `delimiter` označava korišteni razdjelnik između dvaju podataka.

7.6.5.

Metoda koja omogućuje stvaranje novoga dijalekta zove se `register_dialect()`.

8.4.1.

```
class Vozilo:
    pass

v1 = Vozilo()
v2 = Vozilo()
v3 = Vozilo()

print(v1)
print(v2)
print(v3)
```

Izlaz:

```
<__main__.Vozilo object at 0x0000022CDA289400>
<__main__.Vozilo object at 0x0000022CDBC41450>
<__main__.Vozilo object at 0x0000022CDBC41590>
```

8.4.2.

```
class Vozilo:
    def __init__(self, marka, model, godina):
        self.marka = marka
        self.model = model
        self.godina = godina

v1 = Vozilo("Audi", "A5", 2020)
v2 = Vozilo("Rimac", "Nevera", 2022)

print(v1.marka, v1.model, v1.godina)
print(v2.marka, v2.model, v2.godina)
```

Izlaz:

```
Audi A5 2020
Rimac Nevera 2022
```

8.4.3.

```
class Vozilo:
    def __init__(self, marka, model, godina):
        self.marka = marka
        self.model = model
        self.godina = godina

    def __str__(self):
        return f"Marka: {self.marka}, Model: {self.model}, Godina: {self.godina}"

v1 = Vozilo("Audi", "A5", 2020)
v2 = Vozilo("Rimac", "Nevera", 2022)

print(v1)
print(v2)
```

Izlaz:

Marka: Audi, Model: A5, Godina: 2020
 Marka: Rimac, Model: Nevera, Godina: 2022

8.8.1.

```
class Grad:
    def __init__(self, naziv, drzava, stanovnika):
        self.naziv = naziv
        self.drzava = drzava
        self.stanovnika = stanovnika

    def ispis(self):
        print(f"Naziv grada: {self.naziv}")
        print(f"Država: {self.drzava}")
        print(f"Broj stanovnika: {self.stanovnika}")
        print("-" * 30)

g1 = Grad("Zagreb", "Hrvatska", 800000)
g2 = Grad("Ljubljana", "Slovenija", 300000)

g1.ispis()
g2.ispis()

Izlaz:
Naziv grada: Zagreb
Država: Hrvatska
Broj stanovnika: 800000
-----
Naziv grada: Ljubljana
Država: Slovenija
Broj stanovnika: 300000
-----
```

8.8.2.

```
class Student:
    # Statička varijabla - ukupan broj svih ocjena
    ukupanBrojSvihOcjena = 0

    # Konstruktor
    def __init__(self, ime, prezime):
        self.ime = ime
        self.prezime = prezime
        self.ocjene = []

    # Dodavanje ocjene
    def dodajOcjenu(self, ocjena):
        self.ocjene.append(ocjena)
        Student.ukupanBrojSvihOcjena += 1

    # Prosjek ocjena
    def prosjekOcjena(self):
        if len(self.ocjene) == 0:
            return 0
        return sum(self.ocjene) / len(self.ocjene)
```

```

# Medijan ocjena
def medijanOcjena(self):
    if len(self.ocjene) == 0:
        return 0

    sortirane = sorted(self.ocjene)
    n = len(sortirane)
    sredina = n // 2

    if n % 2 == 0:
        o1 = sortirane[sredina - 1]
        o2 = sortirane[sredina]
        return (o1 + o2) / 2
    else:
        return sortirane[sredina]

# Broj određenih ocjena
def brojOcjena(self, o):
    return self.ocjene.count(o)

# Statička metoda - ukupan broj ocjena
def ukupanBrojOcjena():
    return Student.ukupanBrojSvihOcjena

# __str__ metoda
def __str__(self):
    prosjek = self.prosjekOcjena()
    medijan = self.medijanOcjena()
    return (
        f"Student: {self.ime} {self.prezime}\n"
        f"\tOcjene: {self.ocjene}\n"
        f"\tProsjek ocjena: {prosjek:.2f}\n"
        f"\tMedijan ocjena: {medijan}\n"
    )

s1 = Student("Ivan", "Horvat")
s2 = Student("Pero", "Novak")

# Dodavanje ocjena
for o in [5, 4, 3, 5, 2]:
    s1.dodajOcjenу(o)

for o in [4, 5, 5, 3, 4, 5]:
    s2.dodajOcjenу(o)

# Testiranje metoda
print(s1)
print(s2)

print("\nIvan - broj petica:", s1.brojOcjena(5))
print("Ana - broj četvorki:", s2.brojOcjena(4))

print("\nUkupan broj ocjena svih studenata:",
      Student.ukupanBrojOcjena())

```

```

Izlaz:
    Student: Ivan Horvat
    Ocjene: [5, 4, 3, 5, 2]
    Prosjek ocjena: 3.80
    Medijan ocjena: 4

    Student: Pero Novak
    Ocjene: [4, 5, 5, 3, 4, 5]
    Prosjek ocjena: 4.33
    Medijan ocjena: 4.5

    Ivan - broj petica: 2
    Ana - broj četvorki: 2

    Ukupan broj ocjena svih studenata: 11

```

8.11.1.

```

class Osoba:
    def __init__(self, imeOsobe):
        self.imeOsobe = imeOsobe

o = Osoba("Ivan")
print(o.imeOsobe)

o.imeOsobe = "Perica"
print(o.imeOsobe)

```

```

Izlaz:
    Ivan
    Perica

```

8.11.2.

```

class Osoba:
    def __init__(self, imeOsobe):
        self._imeOsobe = imeOsobe

o = Osoba("Ivan")
print(o._imeOsobe)

o._imeOsobe = "Perica"
print(o._imeOsobe)

```

```

Izlaz:
    Ivan
    Perica

```

8.11.3.

```
class Osoba:
    def __init__(self, imeOsobe):
        self.__imeOsobe = imeOsobe

o = Osoba("Ivan")
print(o.__imeOsobe)
o.__imeOsobe = "Perica"
print(o.__imeOsobe)
```

Izlaz:

```
Traceback (most recent call last):
  File "D:\oop.py", line 7, in <module>
    print(o.__imeOsobe)
          ^^^^^^^^^^^^^
AttributeError: 'Osoba' object has no
attribute '__imeOsobe'
```

8.11.4.

```
class Osoba:
    def __init__(self, imeOsobe):
        self.__imeOsobe = imeOsobe

    def getImeOsobe(self):
        return self.__imeOsobe

    def setImeOsobe(self, imeOsobe):
        self.__imeOsobe = imeOsobe

o = Osoba("Ivan")
print(o.getImeOsobe())

o.setImeOsobe("Perica")
print(o.getImeOsobe())
```

Izlaz:

```
Ivan
Perica
```

8.11.5.

```
class Osoba:
    def __init__(self, imeOsobe):
        self.__imeOsobe = imeOsobe

    @property
    def imeOsobe(self):
        return self.__imeOsobe

    @imeOsobe.setter
    def imeOsobe(self, imeOsobe):
        self.__imeOsobe = imeOsobe
```

```
o = Osoba("Ivan")
print(o.imeOsobe)

o.imeOsobe = "Perica"
print(o.imeOsobe)
```

Izlaz:

```
Ivan
Perica
```

8.15.1.

```

class Stavka:
    def __init__(self, naziv, cijena):
        self._naziv = naziv
        self._cijenaNeto = cijena

    def cijenaPdv(self):
        return self._cijenaNeto * 1.25

class Sok(Stavka):
    def __init__(self, naziv, cijena, volumen):
        super().__init__(naziv, cijena)
        self._volumen = volumen

    def ispis(self):
        print("Naziv:", self._naziv)
        print("Volumen:", self._volumen)
        print("Cijena (bez PDV-a):", self._cijenaNeto)

class Hrana(Stavka):
    def __init__(self, naziv, cijena, kalorije):
        super().__init__(naziv, cijena)
        self._kalorije = kalorije

    def ispis(self):
        print("Naziv:", self._naziv)
        print("Kalorije:", self._kalorije)
        print("Cijena (bez PDV-a):", self._cijenaNeto)

p = Sok("Naranča", 10, 0.5)
h = Hrana("Pizza", 40, 150)

p.ispis()
print("Cijena (s PDV-om):", p.cijenaPdv())
print("-----")
h.ispis()
print("Cijena (s PDV-om):", h.cijenaPdv())

```

Izlaz:

```

Naziv: Sok
Volumen: 0.5
Cijena (bez PDV-a): 10
Cijena (s PDV-om): 12.5
-----
Naziv: Pizza
Kalorije: 150
Cijena (bez PDV-a): 40
Cijena (s PDV-om): 50.0

```

8.15.2.

```

class Stavka:
    def __init__(self, naziv, cijena):
        self._naziv = naziv
        self._cijenaNeto = cijena

    def cijenaPdv(self):
        return self._cijenaNeto * 1.25

class Sok(Stavka):
    def __init__(self, naziv, cijena, volumen):
        super().__init__(naziv, cijena)
        self._volumen = volumen

    def ispis(self):
        print("Naziv:", self._naziv)
        print("Volumen:", self._volumen)
        print("Cijena (bez PDV-a):", self._cijenaNeto)

    def cijenaPdv(self):
        return self._cijenaNeto * 1.15

class Hrana(Stavka):
    def __init__(self, naziv, cijena, kalorije):
        super().__init__(naziv, cijena)
        self._kalorije = kalorije

    def ispis(self):
        print("Naziv:", self._naziv)
        print("Kalorije:", self._kalorije)
        print("Cijena (bez PDV-a):", self._cijenaNeto)

    def cijenaPdv(self):
        return self._cijenaNeto * 1.05

```

```

p = Sok("Naranča", 10, 0.5)
h = Hrana("Pizza", 40, 150)

p.ispis()
print("Cijena (s PDV-om):", p.cijenaPdv())
print("-----")
h.ispis()
print("Cijena (s PDV-om):", h.cijenaPdv())

```

Izlaz:

```

Naziv: Sok
Volumen: 0.5
Cijena (bez PDV-a): 10
Cijena (s PDV-om): 11.5
-----
Naziv: Pizza
Kalorije: 150
Cijena (bez PDV-a): 40
Cijena (s PDV-om): 42.0

```

8.15.3.

```
from math import pi

class GeometrijskiLikovi:

    def opseg(a=None, b=None, c=None):
        if a is None and b is None and c is None:
            print("Dogodila se greška!")

        elif a is not None and b is None and c is None:
            print(2 * a * pi)

        elif a is not None and b is not None and c is None:
            print(2 * a + 2 * b)

        else:
            print(a + b + c)

GeometrijskiLikovi.opseg()
GeometrijskiLikovi.opseg(1)
GeometrijskiLikovi.opseg(1, 2)
GeometrijskiLikovi.opseg(1, 2, 3)
```

```
Izlaz:
    Dogodila se greška!
    6.283185307179586
    6
    6
```

8.18.1.

```

from abc import ABC, abstractmethod

# Apstraktni razred
class TransportnoSredstvo(ABC):

    @abstractmethod
    def brojPutnika(self):
        pass

    @abstractmethod
    def vrstaGoriva(self):
        pass

# Izvedeni razred Automobil
class Automobil(TransportnoSredstvo):

    def brojPutnika(self):
        return 5

    def vrstaGoriva(self):
        return "Benzin"

# Izvedeni razred Bicikl
class Bicikl(TransportnoSredstvo):

    def brojPutnika(self):
        return 1

    def vrstaGoriva(self):
        return "Ljudska energija"

# Glavni program
auto = Automobil()
bicikl = Bicikl()

print("Automobil:")
print(f"Broj putnika: {auto.brojPutnika()}")
print(f"Vrsta goriva: {auto.vrstaGoriva()}")
print()
print("Bicikl:")
print(f"Broj putnika: {bicikl.brojPutnika()}")
print(f"Vrsta energije: {bicikl.vrstaGoriva()}")

```

```

Izlaz:

Automobil:
Broj putnika: 5
Vrsta goriva: Benzin

Bicikl:
Broj putnika: 1
Vrsta energije: Ljudska energija

```

8.18.2.

```

from abc import ABC, abstractmethod

# Apstraktni razred
class TransportnoSredstvo(ABC):
    @abstractmethod
    def brojPutnika(self):
        pass

    @abstractmethod
    def vrstaGoriva(self):
        pass

# Izvedeni razred Automobil
class Automobil(TransportnoSredstvo):
    def brojPutnika(self):
        return 5

    def vrstaGoriva(self):
        return "Benzin"

# Izvedeni razred Bicikl
class Bicikl(TransportnoSredstvo):
    def brojPutnika(self):
        return 1

    def vrstaGoriva(self):
        return "Ljudska energija"

# Funkcija se koristi dinamičkim polimorfizmom
def ispisiTransport(sredstvo):
    print(f"Broj putnika: {sredstvo.brojPutnika()}")
        f"\nGorivo: {sredstvo.vrstaGoriva()}"
        f"\n{"-" * 30}")

# Glavni program
auto = Automobil()
bicikl = Bicikl()

# Lista objekata različitih izvedenih razreda
sredstva = [auto, bicikl]
# Poziv funkcije za svaki objekt u listi
for s in sredstva:
    ispisiTransport(s)

Izlaz:
    Broj putnika: 5
    Gorivo: Benzin
    -----
    Broj putnika: 1
    Gorivo: Ljudska energija
    -----

```

8.18.3.

```
class Racun:
    def __init__(self, pocetnoStanje=0):
        self.stanje = pocetnoStanje

    def dodajIznos(self, iznos):
        self.stanje += iznos
        print(f"Dodano: {iznos} EUR")
        return self # vraća self za ulančavanje

    def oduzmiIznos(self, iznos):
        self.stanje -= iznos
        print(f"Oduzeto: {iznos} EUR")
        return self # vraća self za ulančavanje

    def ispisiStanje(self):
        print(f"Stanje: {self.stanje} EUR")
        return self # vraća self za ulančavanje

# Glavni program
racun = Racun(1000) # početno stanje 1000 EUR

# Demonstracija ulančavanja metoda
racun.dodajIznos(500).ispisiStanje().oduzmiIznos(200).ispisiStanje()

Izlaz:
    Dodano: 500 EUR
    Stanje: 1500 EUR
    Oduzeto: 200 EUR
    Stanje: 1300 EUR
```

8.18.1.

Za kompleksnije programe bolje se koristiti objektno orijentiranim programiranjem.

8.18.2.

Osnovna cjelina zove se razred. Razred sadržava kompletnu strukturu i funkcionalnosti objekata.

8.18.3.

Konstruktor.

8.18.4.

Ne.

8.18.5.

Parametar `self` povezuje metodu s objektom.

8.18.6.

Varijablu razreda moguće je dohvatiti iz svih objekata i ona će uvijek imati jednu te istu vrijednost, nevezano za to iz kojega ju se objekta dohvaća, dok je varijabla objekta strogo vezana za neki konkretan objekt.

8.18.7.

`public`, `protected`, `private`.

8.18.8.

Javna varijabla vidljiva je i dohvatljiva iz svih dijelova programskoga kôda, dok je privatna varijabla dohvatljiva samo iz razreda/objekta unutar kojega je definirana.

8.18.9.

Nasljeđivanje je princip kojim se definiranje nekoga razreda vrši korištenjem postojećim razredom.

8.18.10.

Kada dvije i više metoda (funkcija) imaju identično ime, ali različit broj parametara, to se zove preopterećenje (engl. *overloading*).

8.18.11.

Nadjačavanje je situacija u kojoj metoda u izvedenome razredu nadjačava (mijenja) metodu iz osnovnoga razreda.

Literatura

1. Allen Benjamin Downey, *Think Python*, O'Reilly Media, 2012.
2. Aleksandar Stojanović, *Elementi računalnih programa*, Element, 2012.
3. David Beazley, Brian Jones, *Python Cookbook, 3rd Edition*, O'Reilly Media, 2013.
4. Guido van Rossum, *Introduction to Python 3; Documentation for Python*, SoHoBooks, 2010.
5. John Paul Mueller, *Beginning Programming with Python For Dummies*, Wiley, 2014.
6. *Learn Python*, <https://www.learnpython.org/>, dohvaćeno 11. 2. 2019.
7. Leo Budin, Predrag Brođanac, Zlatka Markučić, Smiljana Perić, Dejan Škvorc, Magdalena Babić, *Računalno razmišljanje i programiranje u Pythonu*, Element, 2017.
8. Leo Budin, Predrag Brođanac, Zlatka Markučić, Smiljana Perić, *Napredno rješavanje problema programiranjem u Pythonu*, Element, 2013.
9. Leo Budin, Predrag Brođanac, Zlatka Markučić, Smiljana Perić, *Rješavanje problema programiranjem u Pythonu*, Element, 2012.
10. Mark Lutz, *Programming Python, 4th Edition*, O'Reilly Media, 2010.
11. Marko Hruška, *Osnove programiranja (Python)*, Srce, 2018.
12. Predrag Brođanac, Leo Budin, Zlatka Markučić, Smiljana Perić, *Izrada primjenskih programa u Pythonu*, Element, 2017.
13. *Python 3.x Documentation*, <https://docs.python.org/3/>, dohvaćeno 11. 2. 2019.
14. *Python Notes for Professionals*, GoalKicker, <https://goalkicker.com/PythonBook/>, dohvaćeno 11. 2. 2019.
15. *Python Tutorials*, <https://pythonspot.com/>, dohvaćeno 11. 2. 2019.
16. Zoran Kalafatić, Antonio Pošćić, Siniša Šegvić, Julijan Šribar, *Python za znatiželjne*, Element, 2016.

Bilješke: