Raspodijeljeni sustavi (RS)

Nositelj: doc. dr. sc. Nikola Tanković
Asistent: Luka Blašković, mag. inf.

Ustanova: Sveučilište Jurja Dobrile u Puli, Fakultet informatike u Puli

(1) Programski jezik Python

Raspodijeljeni sustav je svaki računalni sustav koji se sastoji od više povezanih autonomnih računala koja zajedno rade kao jedinstveni kohezivni sustav za postizanje zajedničkog cilja. Drugim riječima, raspodijeljeni sustavi su skupina nezavisnih računala (čvorova u mreži) koji međusobno komuniciraju i koordiniraju svoje radnje kako bi izvršili određeni zadatak.

Na ovom kolegiju studenti će se upoznati s osnovama raspodijeljenih sustava i njihovim karakteristikama te tehnologijama i alatima koji se koriste u njihovom razvoju te naučiti kako razvijati aplikacije s naglaskom na distribuiranu arhitekturu.

Ipak, prije nego krenemo, važno je naučiti programski jezik Python, koji ćemo intenzivno koristiti u nastavku kolegija.

🆙 **Posljednje ažurirano: 30.10.2025.**

Sadržaj

1. Uvod

Razvoj raspodijeljenih sustava postao je ključan za ostvarivanje **visoke dostupnosti**, **skalabilnosti** i **performansi** aplikacija u današnjem digitalnom svijetu. Raspodijeljeni sustavi omogućuju stvaranje složenih sustava sposobnih za obrade koje nadilaze mogućnosti pojedinačnih računala. Ti sustavi pružaju brojne prednosti, uključujući učinkovitiju obradu podataka, bolju prilagodbu velikim opterećenjima (*eng. High load*) te veću otpornost na kvarove (*eng. Fault tolerance*).

Razvoj raspodijeljenih sustava temelji se prvenstveno na **distribuiranoj arhitekturi (*eng. Distributed architecture*)** te razvoju manjih aplikacija koje često nazivamo i **mikroservisima (*eng. Microservices*)**. Mikroservis možemo zamisliti kao malu, nezavisnu aplikaciju, koja se izvršava u vlastitom procesu, obavlja jedan zadatak i komunicira s drugim mikroservisima putem mreže.

Budući da većina studenata koji slušaju ovaj kolegij već posjeduje temeljna znanja iz razvoja softvera, stečena kroz prethodne kolegije **Programsko inženjerstvo** i **Web aplikacije**, ovaj kolegij će se usredotočiti na proširivanje postojećih znanja i vještina (uz korištenje srodnih tehnologija) te njihovu primjenu u kontekstu razvoja raspodijeljenih sustava. Primjerice, na vježbama će se kao glavni protokol za komunikaciju koristiti i dalje **HTTP/HTTPS** te **NoSQL** baza podataka. Također, prisjetit ćemo se izrade jednostavnog sučelja kroz **Vue.js** razvojni okvir, ali i principa dobrog dizajna **REST API** sučelja.

Iako mnogi programski jezici pružaju izvrsne performanse i funkcionalnosti prikladne za razvoj distribuiranih sustava, poput jezika **Go**, koji je popularan izbor za razvoj mikroservisa zbog svoje brzine i ugrađene podrške za konkurentnost, ili pak Java i Rust koji nude snažnu podršku za višedretvenost (*eng. Multithreading*) i asinkrone modele razvoja; mi smo za ovaj kolegij odabrali **Python** kao preferirani jezik budući da je široko primjenjiv u znanstvenim i inženjerskim područjima te ga većina studenata diplomskog studija već dobro poznaje.

Python omogućuje jednostavnu integraciju s postojećim bibliotekama i alatima koji nude gotove komponente namijenjene radu s raspodijeljenim sustavima, poput modula *asyncio*. Takav pristup ubrzava razvoj aplikacija i omogućuje programerima da se usmjere na višu razinu apstrakcije, bez potrebe za implementacijom osnovnih mehanizama. Znanje Pythona danas je temeljna vještina koju bi svaki developer trebao savladati do kraja studija. Njegova popularnost i široka primjena u industriji i znanosti čine ga nezaobilaznim alatom za rješavanje složenih problema i razvoj kvalitetnih - pouzdanih softverskih rješenja.

2. Priprema Python okruženja

2.1 Instalacija Pythona (Win, Linux, macOS)

Python možete preuzeti i instalirati na više načina, a najjednostavniji za većinu korisnika je preuzimanje i pokretanje instalacijskog programa sa službene stranice Pythona. Preporuka je odabrati verziju **3.12** ili noviju.

Kada pokrenete *installer*, ključno je odabrati opciju **Add Python to PATH** kako bi Python bio dostupan iz naredbenog retka (*eng. Command Prompt*). Nakon što završite instalaciju, možete provjeriti je li Python uspješno instaliran pokretanjem naredbe `python --version` u naredbenom retku. Ako je Python uspješno instaliran, trebali biste vidjeti verziju Pythona koju ste instalirali.

> **PATH** je naziv jedne varijable okruženja (*eng. environment variable*) na operacijskim sustavima, a koja sadrži listu direktorija u kojima se nalaze skripte i izvršne datoteke koje možete pokrenuti iz naredbenog retka, bez potrebe za navođenjem pune putanje (*eng. absolute path*) do datoteke.

Jednom kada ste uspješno instalirali Python, možete provjeriti instaliranu verziju sljedećom naredbom u terminalu:

```
→ python --version

# Ili na Linuxu i MacOS-u:
→ python3 --version
```

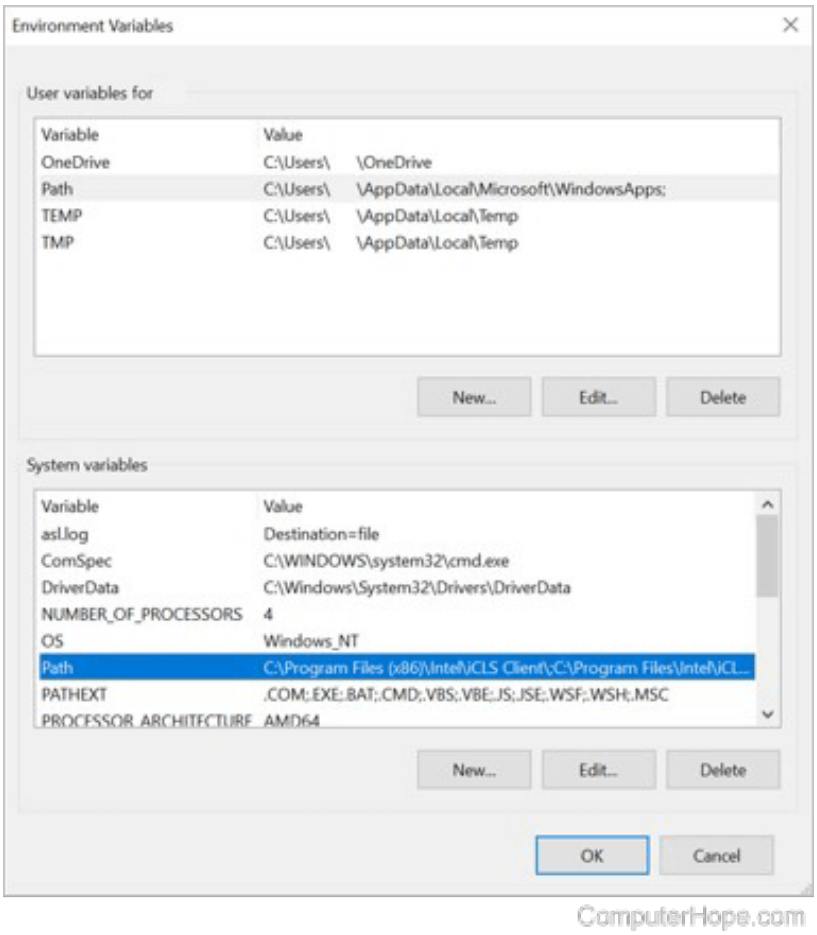
Ako dobijete grešku `"Python is not recognized as an internal or external command"` to znači da Python nije dodan u PATH. U tom slučaju, najčešće rješenje jest ponovo pokrenuti *Python installer* i odabrati opciju **Add Python to PATH**.

Ako imate problema s postavljanjem Pythona u PATH, kratki vodič ovdje.

Ako koristite Windows OS, možete provjeriti `PATH` varijablu pokretanjem naredbe `$Env:Path` u **PowerShell terminalu**. Na Windowsu se preporučuje koristiti **PowerShell** terminal ili novi **Terminal** koji dolazi s Win11, umjesto starog *Command Prompt*.

```
→ $Env:Path
```

Možete provjeriti i putem grafičkog sučelja na Windowsu, otvorite *Start* i ukucajte "environment" te odaberite **Edit the system environment variables**. U prozoru koji se otvori, kliknite na **Environment Variables** te u listi *System variables* pronađite **Path**. Kliknite na **Edit** i provjerite je li putanja do Python instalacijskog direktorija prisutna u listi.



Ako koristite **Linux** ili **MacOS**, Python je najvjerojatnije već instaliran na vašem sustavu. Možete provjeriti verziju Pythona pokretanjem naredbe:

```
→ python3 --version
```

Ako je Python instaliran, dobit ćete verziju Pythona koju koristite. Ako Python nije instaliran, možete ga instalirati putem apt ili homebrew **package managera** (ili drugog), ali i preuzimanjem instalacijskog paketa s Pythonove službene stranice.

> **Napomena**: Na Linuxu i MacOS-u, Python 3 se pokreće s naredbom `python3` kako bi se izbjegla konfuzija s Python 2 koji je još uvijek prisutan na nekim starijim sustavima.

Kako biste provjerili koji je Python interpreter postavljen kao zadani, možete pokrenuti naredbu:

```
→ which python3
```

Naredba `which` će vam izlistati apsolutnu putanju do Python interpretera koji se koristi kada pokrenete `python3` naredbu. Ako želite, možete dodati alias za vaš Python terminal tako da možete pokrenuti Python interpreter jednostavno pokretanjem naredbe `python` umjesto `python3`.

Za `bash` ili `zsh` korisnike, možete otvoriti `~/.bashrc` odnosno `~/.zshrc` konfiguracijsku datoteku kroz `nano` ili `vim` CLI editor:

```
→ nano ~/.bash_profile
→ vim ~/.bash_profile

# ili
→ nano ~/.zshrc
→ vim ~/.zshrc
```

i dodati sljedeću liniju na dno datoteke:

```
→ alias python=python3
```

Spremite izmjene naredbom `Ctrl + O` odnosno `:wq`, pritisnite `Enter` i izađite iz editora. Nakon toga pokrenite sljedeću naredbu kako bi se promjene primijenile:

```
→ source ~/.bashrc

# ili za zsh

→ source ~/.zshrc
```

odnosno za `zsh` korisnike:

```
→ source ~/.zshrc
```

Pokrenite novu sesiju terminala. Sada možete pokrenuti Python interpreter jednostavno pokretanjem naredbe `python`. Također, možete provjeriti koji je Python interpreter postavljen kao zadani pokretanjem naredbe:

```
→ which python
```

Trebali biste dobiti poruku: `python: aliased to python3`.

Kao i jednake rezultate za `python3` i `python`.

```
→ python --version # Python [instalirana_verzija]
→ python3 --version # Python [instalirana_verzija]
```

Ako samo unesete naredbu `python` ili `python3`, trebali biste ući u **Python REPL** (*eng. Read-Eval-Print Loop*) okruženje gdje možete unositi Python naredbe i odmah vidjeti rezultate.

```
→ python

Python 3.9.6 (default, Aug  8 2025, 19:06:38)
[Clang 17.0.0 (clang-1700.3.19.1)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

- izlaz iz ovog okruženja možete napraviti naredbom `exit()` ili kombinacijom tipki `Ctrl + D`.

Ili pak možete pokrenuti inline Python naredbe iz terminala koristeći `-c` opciju:

```
→ python -c "print('Hello, World!')"

Hello, World!
```

> TLDR; Većina korisnika će koristiti `python3` za pokretanje Python interpretera na Linuxu i MacOS-u, dok će se na Windowsu koristi `python`. Međutim, ako hoćete, možete izraditi alias `python` za `python3` kako bi se izbjegla konfuzija.

2.1.1 WSL (Windows Subsystem for Linux)

Za korisnike operacijskog sustava Windows, preporuka je koristiti **Windows Subsystem for Linux (WSL)** tehnologije koja omogućuje pokretanje Linux okruženja unutar Windowsa. WSL pruža izvrsnu kompatibilnost s Linux alatima i bibliotekama, što može biti korisno za razvoj Python aplikacija.

Glavna prednost je korištenje *bash* ili *zsh shella*, čije je poznavanje ključno za rad na udaljenim poslužiteljima, ali i za lakši rad s alatima kao što su `git`, `ssh`, `docker` i drugi koje koristimo na ovim vježbama.

Za instalaciju WSL-a, otvorite **PowerShell** kao administrator i pokrenite sljedeću naredbu:

```
→ wsl --install
```

Nakon toga možete provjeriti instalirane Linux distribucije pokretanjem naredbe:

```
→ wsl --list --online
```

Specifičnu Linux distribuciju (npr. Ubuntu) možete instalirati pokretanjem naredbe:

```
→ wsl --install -d Ubuntu
```

Provjerite status WSL instalacije pokretanjem naredbe:

```
→ wsl --status
```

Ako je sve u redu, trebali biste vidjeti informacije o instaliranoj verziji WSL-a i zadanoj Linux distribuciji.

Nakon instalacije, možete pokrenuti WSL naredbom:

```
→ wsl
```

Kako biste provjerili koji *shell* koristite, pokrenite naredbu:

```
→ echo $SHELL

# ili

→ echo $0
```

Na sljedećim vježbama ćemo koristiti prvenstveno `bash`, međutim ako želite koristiti `zsh` (ili drugi), možete ga instalirati naredbama:

```
→ sudo apt update
→ sudo apt install zsh
```

> **Preporuka:** Osim poznavanja web tehnologija, ovaj kolegij kao preduvjet podrazumijeva osnovno razumijevanje rada u terminalu, poznavanje čestih CLI naredbi za navigaciju kroz datotečni sustav, osnovno upravljanje datotekama i direktorijima, poznavanje nekih alata poput `git`, `ssh`, `curl` te CLI uređivača teksta poput `nano` ili `vim`. Ako vam navedeni alati nisu poznati ili ih želite ponoviti, preporuka je proći kroz materijale iz kolegija Operacijski sustavi (OS) - posebice skripte OS1, OS2, OS3.

2.2 Priprema virtualnog okruženja

Virtualno okruženje (*eng. Virtual Environment*) je tehnologija koja omogućuje izradu izoliranog okruženja za naše Python projekte. Virtualno okruženje rješava mnogobrojne probleme koji se javljaju kada radimo na više projekata koji koriste različite verzije Pythona i/ili različite verzije paketa.

Postoji više alata za upravljanje virtualnim okruženjima, a najpoznatiji su `venv`, `virtualenv` i `conda`.

Slobodni ste koristiti bilo koji od navedenih alata, međutim mi ćemo u sklopu ovog kolegija koristiti `conda` alat.

2.2.1 Instalacija `conda` alata

`conda` je *open-source python package manager* i virtualno okruženje za upravljanje paketima i njihovim ovisnostima (*eng. dependency*). `conda` je dostupan za Windows, Linux i MacOS operacijske sustave.

`conda` je podskup `Anaconda` distribucije, koja dolazi s preinstaliranim paketima i alatima (npr. Jupyter Notebook). Međutim, za potrebe ovog kolegija, dovoljno je instalirati `conda` *package manager*.

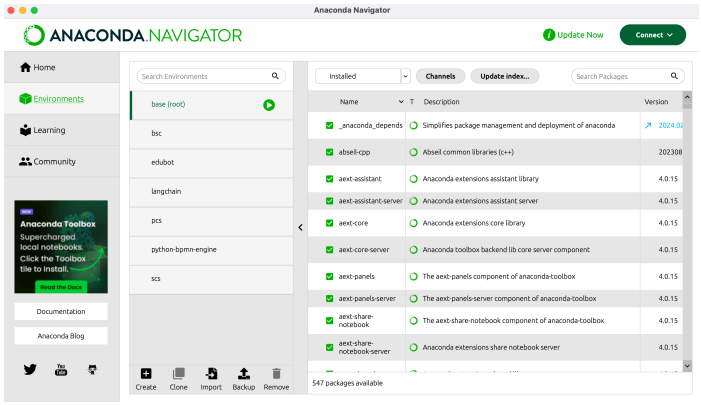
To možete učiniti kroz `Anaconda Navigator` aplikaciju ili preuzimanjem samo `conda` instalacijskog paketa sa službene stranice. Jednostavno odaberite verziju koja odgovara vašem operacijskom sustavu i slijedite upute za instalaciju. Isto tako, možete instalirati kroz *package manager*, npr. `brew` na MacOS-u.

Nakon što ste uspješno instalirali `conda` alat, možete provjeriti je li uspješno instaliran pokretanjem naredbe:

```
→ conda --version
```

Nije loše instalirati i sveobuhvatnu Anaconda distribuciju budući da dolazi s mnogim korisnim alatima, uključujući i grafičko sučelje `Anaconda Navigator` koje olakšava upravljanje okruženjima i paketima.

Anaconda distribuciju možete preuzeti sa službene stranice. Naravno, `conda` je uključena u ovoj distribuciji pa možete provjeriti na isti način prepoznaje li ju naredbeni redak.



> Prikaz aplikacije Anaconda Navigator i pregled izrađenih okruženja i Python paketa.

To je to! Spremni smo za rad s Pythonom! 🐍

---

3. Python osnove

Python je **visokorazinski** (*eng. high-level*) programski jezik **opće namjene** (*eng. general-purpose*) koji ističe jednostavnost sintakse i čitljivost koda, čime omogućuje brži i učinkovitiji razvoj softvera. Python je također **dinamički tipiziran** jezik (*eng. dynamically typed language*) što znači da se tipovi varijabli određuju za vrijeme izvođenja (*eng. runtime*), a ne za vrijeme kompilacije (*eng. compile-time*).

Popularan je i široko korišten u mnogim područjima, uključujući: web razvoj, data science, matematika, strojno učenje i umjetna inteligencija itd.

Ono što nam je još važno za zapamtiti, Python je tzv. **multi-paradigmatski** (*eng. multi-paradigm*) jezik, što znači da podržava više stilova programiranja, uključujući proceduralno, objektno orijentirano ili funkcijsko programiranje. Korisnik može odabrati stil programiranja koji najbolje odgovara problemu koji rješava pa i kombinirati različite stilove programiranja što čini ovaj jezik vrlo fleksibilnim.

3.1 VS Code okruženje

Za rad s Pythonom preporučujemo korištenje **Visual Studio Code** (**VS Code**) razvojno okruženje. VS Code je besplatan, open-source IDE (*eng. Integrated development environment*) kojeg održava Microsoft, a nudi bogat ekosustav ekstenzija i alata koji olakšavaju razvoj aplikacija u Pythonu. Naravno, možete koristiti IDE po želji, međutim mi ćemo na vježbama iz ovog kolegija koristiti VS Code.

VS Code možete preuzeti s službene stranice i instalirati na vaš operacijski sustav. Nakon instalacije, možete pokrenuti VS Code i instalirati ekstenziju koja će vam olakšati rad s Pythonom.

Python ekstenzija: nudi generalnu podršku za Python razvoj, uključujući IntelliSense, debugger (Python Debugger), formatiranje, linting, itd.

- ova ekstenzija instalirat će vam još i `Python Debugger` i `Pylance` ekstenzije koje upotpunjuju rad s Pythonom u VS Code-u.
- moguće je po želji instalirati i alternativni *linter* poput `Flake8`, `Pylint`, `Black Formatter` no `Pylance` je zadani *linter* koji dolazi s ovom ekstenzijom.

Provjerite jesu li sve ekstenzije uspješno instalirane i aktivirane. Možete ih pronaći u **Extensions** panelu na lijevoj strani VS Code sučelja.

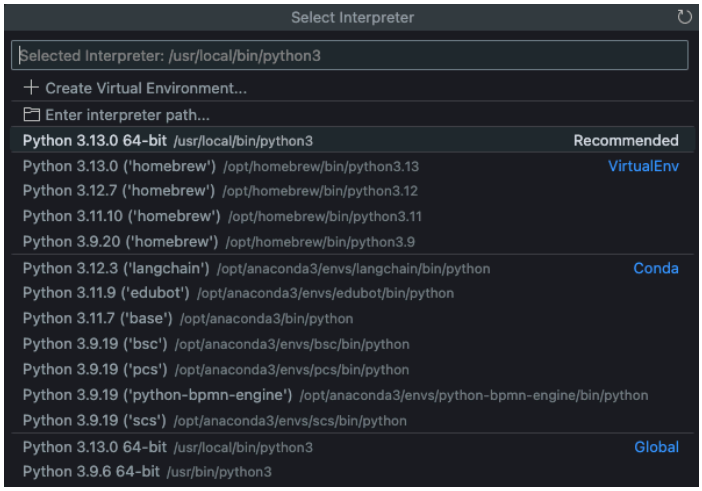
3.2 Osnove Python sintakse

Za početak nećemo raditi s bibliotekama i alatima, već ćemo se upoznati s osnovama Python sintakse, stoga nam za sada neće niti trebati virtualno okruženje.

Krenimo s izradom osnovne Python skripte. Kreirajte novu datoteku s ekstenzijom `.py`. Na primjer, nazovite datoteku `hello.py`.

U donjem desnom kutu VS Code sučelja uočite trenutni Python *interpreter* koji se koristi. Provjerite je li to Python *interpreter* koji ste instalirali i koji želite koristiti. Ako nije, možete ga promijeniti.

Ovaj prozor možete otvoriti i kraticom `CTRL/CMD + SHIFT + P` te upisivanjem naredbe `Python: Select Interpreter`.



> Odabran je Python interpreter (Python 3.13.0 /usr/local/bin/python3) koji će se koristiti za izvršavanje Python skripte `hello.py`.

U pythonu možemo ispisivati poruke u konzolu koristeći naredbu `print()`. Na primjer, možemo ispisati poruku "Hello, World!" koristeći sljedeći kod:

```
→ print("Hello, World!")
```

Spremite datoteku i pokrenite je klikom na gumb **Run** u gornjem desnom kutu datoteke ili pritiskom na `Ctrl + Alt + N` odnosno `Cmd + Alt + N` na MacOS-u.
Trebali biste vidjeti ispis "Hello, World!" u terminalu.

Drugi način je pokretanje skripte iz terminala. Otvorite terminal u VS Code-u klikom na **Terminal** > **New Terminal** i odaberite terminal po želji, preferabilno `bash` ili `zsh` terminal.

U terminalu se pozicionirajte u direktorij gdje se nalazi vaša Python skripta i pokrenite je naredbom:

```
→ python hello.py
```

Ili naredbom `python3` ako koristite Linux ili MacOS i niste dodali alias za `python`:

```
→ python3 hello.py
```

Kratki podsjetnik za bash navigaciju u terminalu:

- `cd [relativna/apsolutna putanja]` - promjena direktorija
- `cd ..` - povratak u prethodni direktorij
- `ls` - ispis sadržaja direktorija
- `pwd` - ispis trenutne putanje
- `clear` - "brisanje" sadržaja terminala

3.2.1 Varijable

Varijable u Pythonu se deklariraju na sljedeći način:

```
a = 5

b = "Hello, World!"

c = 3.14
```

Dakle, primijetite da se ne navodi tip (*eng. type*) varijable prilikom deklaracije, već se Python sam brine o tipu varijable. Varijabla `a` je tipa `int`, varijabla `b` je tipa `str`, a varijabla `c` je tipa `float`.

> Podsjetnik: Varijable u Pythonu su dinamički tipizirane, što znači da se tip varijable određuje za vrijeme izvođenja, a ne za vrijeme kompilacije.

U varijable je moguće pohraniti sve fiksne vrijednosti, često ih zovemo **literalima** (*eng. literals*). U Pythonu postoje sljedeći najvažniji literali:

- **Numerički literali**: `10` (`int`), `3.14` (`float`), 0b1010 (`binary`), `0o12` (`octal`), `0xA` (`hexadecimal`)
- **Tekstualni literali**: `"Hello, World!"`, `'Hello, World!'` (`str`)
- **Logički/boolean literali**: `True`, `False` (`bool`)
- **Nul literal**: `None` (predstavlja nul vrijednost)
- **Kolekcijski literali**: liste (`[]`), n-torke (`()`), rječnici (`{}`), skupovi (`set()`), itd.
- **Funkcijski literali**: `def` funkcije, *lambda* izrazi (`lambda`)

Moguće je pregaziti vrijednost varijable:

```
a = 5

a = 10

print(a) # 10
```

Varijablu možemo ispisati koristeći naredbu `print()`:

```
a = 5
b = 10

print(a + b) # 15
```

```
a = "Hello, "

b = "World!"

print(a + b) # Hello, World!
```

Osim što se mogu pregaziti vrijednostima, varijable se mogu i pregaziti tipom varijable:

```
a = 5

a = "Hello, World!" # može i s jednostrukim navodnicima

print(a) # Hello, World!
```

Varijable u Pythonu mogu sadržavati slova, brojeve i znak `_`, ali ne smiju započinjati brojem.

```

# Ovo je ispravno

my_variable = 5
myVariable = 10
myVariable2 = 15

# Ovo nije ispravno (SyntaxError)

2myVariable = 5 # ne može započinjati brojem
my-Variable = 10 # ne može sadržavati znak -
my Variable = 15 # ne može sadržavati razmak
```

Varijable u Pythonu su **case-sensitive**, što znači da razlikuju velika i mala slova.

```
my_variable = 5
My_Variable = 10
MY_VARIABLE = 15

print(my_variable) # 5
print(My_Variable) # 10
print(MY_VARIABLE) # 15
```

Jednolinijske komentare u Pythonu možemo pisati koristeći znak `#`:

```
# Ovo je komentar

a = 5 # Ovo je komentar
```

Dok višelinijske komentare možemo pisati koristeći znakove `"""` ili `'''`:

```
"""
Ovo
je
višelinijski
komentar
"""

# Ili

'''
Ovo
je
isto višelinijski
komentar
'''
```

Međutim, **moguće je** specificirati tip varijable (*eksplicitni type-casting*) koristeći tzv. *Type Casting*:

```
a = 5
# ili
a = int(5)
```

Rezultat je isti, ali se ovim pristupom istodobno **naglašava tip varijable** te, ako je potrebno, **provodi konverzija** tipa tijekom izvođenja programa.

```
x = str(3)
y = int(3)
z = float(3)
```

Što će biti pohranjeno u varijable `x`, `y` i `z`?

▸ Spoiler alert! Odgovor na pitanje

Ako se želimo uvjeriti, možemo uvijek provjeriti tip varijable koristeći funkciju `type()`:

```
x = str(3)
y = int(3)
z = float(3)

print(type(x)) # <class 'str'>
print(type(y)) # <class 'int'>
print(type(z)) # <class 'float'>
```

Ipak, ako bismo željeli provjeriti tip varijable, bolje je koristiti `isinstance()` funkciju budući da vraća boolean vrijednost (`bool`), a ne `str`:

```
x = str(3)
y = int(3)

if isinstance(x, str):
    print("x je string") # x je string

if isinstance(y, int):
    print("y je integer") # y je integer
```

Primjer eksplicitne konverzije tijekom izvođenja programa:

```
a = "5"

b = int(a) # konvertiramo string u integer
print(b + 10, type(b)) # 15, <class 'int'>

c = 0.0

d = bool(c) # konvertiramo float u bool
print(d, type(d)) # False, <class 'bool'>
```

Najčešće korištene type-casting konverzije **literala** su:

- `int()` - konvertira u cijeli broj
- `float()` - konvertira u decimalni broj
- `str()` - konvertira u tekstualni niz
- `bool()` - konvertira u logičku vrijednost

Međutim, imajte na umu da neke konverzije mogu rezultirati greškom ako se pokušava konvertirati nekompatibilan tip:

Primjer: Konverzija stringa koji ne predstavlja broj u `int`:

```
a = "Hello"
b = int(a) # ValueError: invalid literal for int() with base 10: 'Hello'
```

> Napomena: Osim *type-castinga*, u Pythonu postoji i tzv. *type hinting*, koji programerima omogućava da naznače očekivani tip varijable bez potrebe za eksplicitnom konverzijom. Ova praksa poboljšava čitljivost koda i olakšava rad alatima za statičku analizu, a sve češće služi i kao temelj za razvoj programskih okvira i alata koji koriste te informacije za *runtime* optimizaciju. *Type hinting* ćemo raditi na budućim vježbama. Samim time, *type-casting* se gotovo i ne koristi za označavanje tipova varijabli, već isključivo za konverziju tipova.

Prilikom imenovanja varijabli s više riječi, može se koristiti konvencija imenovanja po izboru, međutim u Pythonu je uobičajeno koristiti **Snake Case** notaciju/konvenciju.

Camel Case

```
myVariable = 5
brojStudenata = 30
doneForToday = False
```

Pascal Case

```
MyVariable = 5
BrojStudenata = 30
DoneForToday = False
```

Snake Case

```
my_variable = 5 # preferirana konvencija za imenovanje varijablu u Pythonu
broj_studenata = 30
done_for_today = False
```

Python podržava tzv. **višestruko dodjeljivanje vrijednosti** (*eng. Multiple Variable Assignment*), što omogućava dodjeljivanje jedne ili više vrijednosti većem broju varijabli unutar jedne naredbe (jedne linije koda):

Primjerice imamo varijable `a`, `b` i `c` i hoćemo im dodijeliti vrijednosti `5`, `10` i `15`:

```
a, b, c = 5, 10, 15

print(a) # 5
print(b) # 10
print(c) # 15
```

Isto je moguće raditi i s drugim tipovima podataka:

```
a, b, c = "Hello", 5, 3.14

print(a) # Hello
print(b) # 5
print(c) # 3.14
```

> Napomena: Broj varijabli (lijevi operandi operatora `=`) mora odgovarati broju vrijednosti (desni operandi operatora `=`) koji se dodjeljuje, inače će Python baciti grešku.

Ipak, moguće je i dodjeljivanje iste vrijednosti više varijabli, ali moramo malo promijeniti sintaksu:

```
a = b = c = "same value" # koristimo više znakova jednakosti umjesto zareza

print(a) # same value
print(b) # same value
print(c) # same value
```

Moguće je i ispisati vrijednosti varijabli u jednom redu koristeći `print()` funkciju:

```
a = 5
b = 10
c = 15

print(a, b, c) # 5 10 15
```

Pa i izvršiti konkatenaciju varijabli:

```
a = "Moje "
b = "ime "
c = "je "
d = "Pero"

print(a + b + c + d) # Moje ime je Pero
```

Primijetite da smo nakon vrijednosti svake varijable dodali razmak kako bi rezultat bio čitljiv. Nećemo to raditi, već ćemo navoditi varijable odvojene zarezom unutar `print()` funkcije:

```
a = "Moje"
b = "ime"
c = "je"
d = "Pero"

print(a, b, c, d) # Moje ime je Pero
```

Na ovaj način Python će automatski dodati razmak (`" "`) između varijabli. Ako želimo promijeniti separator, možemo to učiniti koristeći `sep` argument:

```
a = "Moje"
b = "ime"
c = "je"
d = "Pero"

print(a, b, c, d, sep="-") # Moje-ime-je-Pero
```

Konačni znakovni niz moguće je dovršiti i nekim drugim znakom umjesto novog reda (`\n`), koristeći `end` argument:

```
a = "Hello"
b = "World!"
print(a, end=" ") # Hello (bez novog reda)

# ili

print(a, b, end="!!!\n") # Hello World!!!
```

> Napomena: argumenti `sep` i `end` su opcionalni argumenti funkcije `print()`, a mogu se koristiti zajedno ili odvojeno. Na početku funkcije prvu se navode **pozicijski** argumenti (*eng. positional arguments*) i može ih biti neograničen broj, nama su to varijable `a`, `b`. Nakon pozicijskih argumenata navode se **keyword** argumenti (*eng. keyword arguments*), a to su npr. `sep` i `end` (u slučaju naredbe `print()`). Pozicijski argumenti moraju biti navedeni prije keyword argumenata.

> Napomena: `print` naredba vrlo je korisna i često se koristi za ispis poruka u konzolu, najčešće tijekom brzog testiranja i otklanjanja pogrešaka. Ipak, u stvarnim projektima preporučuje se korištenje naprednijih alata za debuggiranje, kao što je `logging` biblioteka, koja omogućuje precizniju kontrolu i praćenje rada programa korištenjem različitih razina logiranja (npr. `DEBUG`, `INFO`, `WARNING`, `ERROR`), pa i usmjeravanje logova u datoteke ili udaljene sustave za praćenje (*eng. monitoring systems*).

3.2.2 Logički izrazi

Pri oblikovanju računskih postupaka često je potrebno usmjeriti tok izvođenja programa ovisno o nekom uvjetu. Uvjet može biti ispunjen ili ne, a ta dva ishoda se obično poistovjećuju s vrijednostima istinitosti iz matematičke logike odnosno logike sudova:

- istinito (*eng. True*)
- neistinito (*eng. False*)

Python za prikaz vrijednosti istinitosti definira poseban ugrađeni tip podatka `bool`, čije su moguće vrijednosti `True` (istinito) i `False` (neistinito). Obratite pažnju na **Velika Početna Slova** ovih ključnih riječi!

Logički izrazi se koriste za **usporedbu vrijednosti** i **provjeru određenog uvjeta**. Svaki logički izraz vraća vrijednost tipa `bool`.

Izraze možemo graditi koristeći operatore. U Pythonu postoji 7 osnovnih skupina operatora:

1. **Aritmetički operatori** (*eng. Arithmetic operators*)
2. **Operatori pridruživanja** (*eng. Assignment operators*)
3. **Operatori usporedbe** (*eng. Comparison operators*)
4. **Logički operatori** (*eng. Logical operators*)
5. **Operatori identiteta** (*eng. Identity operators*)
6. **Operatori pripadnosti** (*eng. Membership operators*)
7. **Operatori bitovnih operacija** (*eng. Bitwise operators*)

1. Aritmetički operatori (eng. Arithmetic operators)

Aritmetički operatori se koriste za izvođenje matematičkih operacija na brojevima. U Pythonu postoje sljedeći aritmetički operatori:

Operator	Opis	Primjer	Rezultat
`+`	Zbrajanje	`5 + 3`	`8`
`-`	Oduzimanje	`5 - 3`	`2`
`*`	Množenje	`5 * 3`	`15`
`/`	Dijeljenje (`float` - čak i ako je rezultat cjelobrojan)	`5 / 2`	`2.5`
`//`	Cjelobrojno dijeljenje	`5 // 2`	`2`
`%`	Ostatak pri dijeljenju (modulo)	`5 % 2`	`1`
`**`	Potenciranje	`5 ** 3`	`125`

> Napomena: Navedeni operatori su binarni (imaju dva operanda), međutim neki se mogu koristiti i kao unarni (npr. `-` za negaciju broja)

Pogledajmo nekoliko primjera aritmetičkih operacija:

```
a = 5
b = 3

print(a + b) # 8
print(a - b) # 2
print(a * b) # 15
print(a / b) # 1.6666666666666667 (float)
print(a // b) # 1 (int)
print(a % b) # 2
print(a ** b) # 125
```

U Pythonu se realni brojevi prikazuju ugrađenim tipom `float`, dok se cijeli brojevi prikazuju tipom `int`. Kao što je i uobičajeno, možemo ih stvarati konverzijom objekata drugih tipova primjenom konstruktora `float`:

Što će biti ispisano u sljedećem primjeru?

```
float(31), float(1 < 2) # konverzija brojeva
```

▸ Spoiler alert! Odgovor na pitanje

Pored toga, decimalni brojevi mogu nastati i kao rezultat dijeljenja cijelih brojeva:

```
print(1/11) # 0.09090909090909091
```

Za jako velike ili jako male brojeve često je praktičnije koristiti tzv. znanstveni zapis (*eng. scientific notation*) kod kojega se red veličine broja izražava prikladnom potencijom broja 10. Pritom se eksponent označava malim ili velikim slovom `e`, a može biti i negativan. Na primjer:

```
print(1.23e-4) # 0.000123
print(1.23e4) # 12300.0
```

Ako literal premaši najveću vrijednost koju može prikazati, Python će ga zapisati kao specijalnu vrijednost `inf` koja odgovara neizmjerno velikom broju (*eng. infinity*):

```
print(1e309) # inf
```

Prilikom dijeljenja s nulom, Python će baciti grešku `ZeroDivisionError`:

```
print(1/0) # ZeroDivisionError: division by zero
```

Što se tiče ugrađenih funkcija nad realnim brojevima, ima ih mnogo i možete ih pronaći vrlo lako na Internetu, za sada možemo spomenuti nekoliko njih koje se često koriste:

```
print(abs(-5)) # 5 (apsolutna vrijednost)
print(round(3.14159, 2)) # 3.14 (zaokruživanje na n decimala)
print(max(1, 2, 3, 4, 5)) # 5 (maksimalna vrijednost)
print(min(1, 2, 3, 4, 5)) # 1 (minimalna vrijednost)
```

> Napomena: Navedene funkcije su ugrađene u Python i dostupne su bez potrebe za uvozom biblioteke. Ako usporedimo sa JavaScriptom, sve ove funkcije moguće je pozivati s kolekcijama podataka, što u JavaScriptu nije slučaj.

Iz `math` biblioteke možemo koristiti veliki broj funkcija koje primaju realne brojeve. Uključene su važnije matematičke funkcije, korisne konverzije, uobičajene trigonometrijske i hiperbolne funkcije, te neke specijalne funkcije i konstante:

```
import math

print(math.sqrt(16)) # 4.0 (kvadratni korijen broja 16)
print(math.pow(2, 3)) # 8.0 (potenciranje - računa 2 na treću potenciju; 2^3))

print(math.exp(1)) # 2.718281828459045 (računa e^x, gdje je x=1)
print(math.log(10)) # 2.302585092994046 (ln(x)) - prirodni logaritam broja x (baza e)

print(math.trunc(3.14)) # 3 (uklanja decimalni dio bez zaokruživanja)
print(math.ceil(3.14)) # 4 (zaokružuje broj na najbliži veći cijeli broj)
print(math.floor(3.14)) # 3 (zaokružuje broj na najbliži manji cijeli broj)
```

Nekoliko praktičnih funkcija za testiranje konačnosti realnih brojeva koje su dostupne u `math` biblioteci:

```
import math

print(math.isfinite(1.0))  # True – provjerava je li vrijednost konačan broj (nije ±inf ili NaN)
print(math.isinf(1.0))     # False – provjerava je li vrijednost beskonačna (±inf)
print(math.isnan(1.0))     # False – provjerava je li vrijednost NaN (Not a Number)
```

2. Operatori pridruživanja (eng. Assignment operators)

Operatori pridruživanja se koriste za dodjeljivanje vrijednosti varijablama. U Pythonu postoje sljedeći operatori pridruživanja:

Operator	Opis	Primjer	Ekvivalent
`=`	Pridruživanje	`x = 5`	`x = 5`
`+=`	Dodaj i pridruži	`x += 3`	`x = x + 3`
`-=`	Oduzmi i pridruži	`x -= 3`	`x = x - 3`
`*=`	Pomnoži i pridruži	`x *= 3`	`x = x * 3`
`/=`	Podijeli i pridruži	`x /= 3`	`x = x / 3`
`//=`	Cjelobrojno podijeli i pridruži	`x //= 3`	`x = x // 3`
`%=`	Modulo i pridruži	`x %= 3`	`x = x % 3`
`**=`	Potenciraj i pridruži	`x **= 3`	`x = x ** 3`

3. Operatori usporedbe (eng. Comparison operators)

Operatori usporedbe se koriste za usporedbu dvije vrijednosti. U Pythonu postoje sljedeći operatori usporedbe:

Operator	Opis	Primjer	Rezultat
`==`	Jednako	`5 == 3`	`False`
`!=`	Nije jednako	`5 != 3`	`True`
`>`	Veće od	`5 > 3`	`True`
`<`	Manje od	`5 < 3`	`False`
`>=`	Veće ili jednako od	`5 >= 3`	`True`
`<=`	Manje ili jednako od	`5 <= 3`	`False`

Pogledajmo nekoliko usporedba cjelobrojnih podataka:

```
a = 5
b = 10

print(a == b) # False
print(a != b) # True
print(a > b) # False
print(a < b) # True
print(a >= b) # False
print(a <= b) # True
```

> Napomena: Treba biti oprezan prilikom uspoređivanja realnih brojeva zbog ograničenja u točnosti prikaza brojeva s pomičnim zarezom, odnosno zbog nepreciznosti njihova prikaza. Posebno se to odnosi na cjelobrojne razlomke i decimalne konstante jer nam njihov sažeti izvorni zapis može sugerirati jednaku sažetost njihovog internog prikaza u memoriji računala. Nikad ne smijemo smetnuti s uma da to gotovo nikada nije slučaj jer većina racionalnih brojeva u koje uvrštavamo i decimalne konstante nemaju konačan prikaz u binarnom brojevnom sustavu. Stoga, uvijek treba koristiti odgovarajuće funkcije za usporedbu realnih brojeva koje uzimaju u obzir određenu toleranciju.

Razmotrimo prvo dva razlomka čija bi razlika trebala biti točno 1, ali u praksi se to ne događa:

```
print(5/3 == 1+2/3) # False
```

Jednako tako moramo biti oprezni i s decimalnim brojevima:

```
print(0.1 + 0.2 == 0.3) # False
# ili
print(0.1 * 3 == 0.3) # False (ipak, ovo se može razlikovati ovisno o distribuciji
Pythona)
```

Računski rezultat možemo provjeriti i funkcijom `math.isclose()` koja omogućava usporedbu s određenom tolerancijom:

```
import math

print(math.isclose(5/3, 1 + 2/3)) # True
print(math.isclose(0.1 * 3, 0.3)) # True
```

U ovakvim slučajevima možemo koristiti i druge funkcije za usporedbu realnih brojeva koje uzimaju u obzir određenu toleranciju:

```
import fractions, decimal

print(fractions.Fraction(5, 3) == 1 + fractions.Fraction(2, 3)) # True

print(decimal.Decimal('0.1') * 3 == decimal.Decimal('0.3'))  # True
```

Operatore usporedbe moguće je primjenjivati i na većinu drugih ugrađenih tipova podataka u Pythonu, kao i na korisnički definirane tipove (*eng. user-defined types*) koji podržavaju odgovarajuće operatore, pri čemu značenje usporedbe ovisi o pojedinom tipu, odnosno o načinu na koji su operatori preopterećeni (*eng. operator overloading*).

Ono što je zanimljivo u Pythonu, i pomalo nekonvencionalno u odnosu na druge jezike, jest da se operatori usporedbe mogu ulančavati, kao matematički izrazi:

```
a = 5
b = 10
c = 15

print(a < b < c) # True (5 < 10 < 15)
```

Moguće je graditi "lance" proizvoljne duljine, npr.

```
print(0 < 3 < 5 < 100) # True
```

To naravno mogu biti bilo kakvi izrazi, ne samo "veće" i "manje" usporedbe:

```
print(4 == 2*2 == 2**2) # True
```

Slično kao i u drugim jezicima, u Pythonu se određeni *non-bool* izrazi tumače kao `True` ili `False` odnosno tzv. "truthy" i "falsy" vrijednosti.

Na isti način kao što koristimo *Casting* za promjenu ili definiranje tipa varijable (npr. `int()`, `str()`, `float()`), možemo koristiti i `bool()` konstruktor za pretvorbu vrijednosti u `bool` tip.

Vrijede "uobičajena" pravila:

```
print(bool(0)) # False (0 se tumači kao False)
print(bool(1)) # True (svi brojevi osim 0 se tumače kao True)
print(bool(-1)) # True (pa i negativni brojevi)

print(bool("")) # False (prazan string se tumači kao False)
print(bool("cvrčak")) # True (svi stringovi osim praznog se tumače kao True)
print(bool(" ")) # True (čak i prazan string s razmakom se tumači kao True)

print(bool([])) # False (prazna lista se tumači kao False)
print(bool([1, 2, 3])) # True (sve liste osim prazne se tumače kao True)

print(bool(())) # False (prazna n-torka se tumači kao False)
print(bool((1, 2, 3))) # True (sve n-torke osim prazne se tumače kao True)

print(bool({})) # False (prazan rječnik se tumači kao False)
print(bool({"key": "value"})) # True (svi rječnici osim praznog se tumače kao True)

print(bool(None)) # False (None se tumači kao False)
```

4. Logički operatori (eng. Logical operators)

Logički operatori se koriste za kombiniranje logičkih izraza. Nad objektima logičkog tipa `bool` moguće je primjenjivati uobičajene operatore `and`, `or` i `not`.

Operator	Opis	Primjer	Rezultat
`and`	Konjunkcija ili logičko "I" - `True` ako su oba izraza `True`	`True and False`	`False`
`or`	Disjunkcija ili logičko "ILI - `True` ako je barem jedan izraz `True`	`True or False`	`True`
`not`	Negacija ili logičko "NE". Negiramo operande, ne operatore.	`not True`	`False`

Izračunavanje logičkih operatora prestaje **čim konačna vrijednost izraza postane "jasna"**.

Primjer:

```
False and x # ?

True or x # ?

x or y or z or True # ?
```

Je li nam bitna vrijednost `x` u ovim izrazima?

▸ Spoiler alert! Odgovor na pitanje

Sad kad smo uveli logičke, usporedne i aritmetičke operatore, možemo reći da se ulančani operatori usporedbe interpretiraju kao **konjunkcija pojedinačnih binarnih usporedbi**.

- Primjerice, izraz `1 < x < 6` se interpretira poput: `1 < x and x < 6`. Pritom svaki od ugniježđenih operanada ovakvih izraza **izračunava samo jednom** , a vrijednost cijelog izraza postaje `False` čim neka od usporedbi ne bude zadovoljena - **naknadne usporedbe se u tom slučaju više neće provoditi**.

Primjer:

```
1 < 2+3 < 6 # koliko će se usporedbi izvršiti?
```

▸ Spoiler alert! Odgovor na pitanje

Izraz se interpretira kao `1 < 2+3 and 2+3 < 6`, dakle izvršit će se dvije usporedbe.

Međutim, zbrajanje će se izvršiti samo jednom, budući da Python izračunava izraz (2+3) samo jednom, a onda primjenjuje dobivenu vrijednost na obe usporedbe.

```
1 < 4 < 3 < 6 # koliko će se usporedbi izvršiti?
```

▸ Spoiler alert! Odgovor na pitanje

Izraz se interpretira kao `1 < 4 and 4 < 3 and 3 < 6`.

Prva usporedba je zadovoljena, ali druga nije, pa se izračunavanje prekida i cijeli izraz se tumači kao `False`.

Drugim riječima, treća usporedba se neće uopće izvesti.

Logičke operatore možemo primijeniti i na podatke ostalih tipova (npr. integere). Operator `not` jednostavno vraća negiranu logičku vrijednost operanda (literala koji slijedi).

- Operator `and` vraća lijevi argument ako je njegova logička vrijednost `False`, inače vraća desni argument.
- Operator `or` vraća lijevi argument ako je njegova logička vrijednost `True`, a u protivnom vraća desni argument.

```
print(not True) # output: False

print(5 and 3) # output: 3 - jer je 5 True, a 3 je zadnji argument

print(0 and 3) # output: 0 - jer je 0 False, a 3 se neće ni provjeravati

print(5 or 3) # output: 5 - jer je 5 True, a 3 se neće ni provjeravati

print(0 or 3) # output: 3 - jer je 0 False, a 3 je zadnji argument
```

OK, što će vratiti izraz `5 and 'cvrčak'`?

▸ Spoiler alert! Odgovor na pitanje

A što će vratiti izraz `'' and 42`?

▸ Spoiler alert! Odgovor na pitanje

Iako je `bool` samostalan podatkovni tip, on je ujedno i podtip tipa `int`. Zbog toga se vrijednosti `True` i `False` mogu koristiti i u aritmetičkim izrazima, pri čemu se ponašaju kao brojevi `1` i `0`. To pokazuju sljedeći primjeri:

```
print(True + True) # 2

print(False + False) # 0

print (True + 1) # 2

print (False * 3) # 0
```

5. Operatori identiteta (eng. Identity operators)

Postoje dva operatora identiteta: `is` i `is not`. Ovi operatori koriste se za **usporedbu identiteta objekata**, ne nužno njihovih vrijednosti. Što to znači?

Operator `is` u Pythonu vraća `True` ako dvije varijable referenciraju isti **objekt u memoriji** (tj. imaju isti identifikator objekta), a `False` ako referenciraju različite objekte, čak i ako su im vrijednosti jednake. Dakle, `is` provjerava identitet objekta, a ne jednakost vrijednosti koje objekt (npr. neka kolekcija) definira.

```
a = [1, 2, 3] # lista
b = [1, 2, 3] # lista

print(a is b) # False (memorijske lokacije su različite i liste su promjenjive)

print (a == b) # True (vrijednosti su jednake)
```

Što se događa u sljedećem primjeru?

```
a = [1, 2, 3]
b = a

print(a is b) # ?
print(a == b) # ?
```

▸ Spoiler alert! Odgovor na pitanje

A ovdje, što će biti?

```
a = 10
b = 10

print(a is b) # ?
print(a == b) # ?
```

▸ Spoiler alert! Odgovor na pitanje

Simple answer: Brojevi su pohranjeni na istoj memorijskoj lokaciji (*cached*) i nepromijenjivi su (*eng. immutable*).

Operator `is not` vraća `True` ako objekti nisu jednaki, odnosno ako se objekti ne nalaze na istoj memorijskoj lokaciji.

```
a = 10
b = 20
print(a is not b) # True

str1 = "hello"
str2 = "hello"

print(str1 is not str2) # Nisu na istoj memorijskoj lokaciji, ali Python optimizira na
jednak način kao i manje brojeve, dakle False
```

6. Operatori pripadnosti (eng. Membership operators)

Sve Pythonove kolekcije omogućuju provjeru pripadnosti elementa pomoću operatora `in` i `not in`. Ovi operatori koriste se za provjeru pripadnosti elementa kolekciji. Neki ih svrstavaju u logičke operatore ili operatore usporedbe jer kao rezultat daju logičku vrijednost. Operator `in` vraća `True` ako se određeni element nalazi u kolekciji, a `False` ako se ne nalazi. Operator `not in` radi obrnuto.

Ovi operatori često se koriste u Pythonu kada radimo s nizovima znakova, listama, skupovima i rječnicima.

```
a = [1, 2, 3, 4, 5] # lista

print(1 in a) # True
print(6 in a) # False
print(1 not in a) # False
print(6 not in a) # True
```

```
iks = 'x'
print (iks in 'cvrčak') # False

samoglasnici = 'aeiou'

print('a' in samoglasnici) # True
print('b' in samoglasnici) # False
```

```
stabla = ['hrast', 'bukva', 'javor', 'bor'] # lista

print('bukva' in stabla) # True

print('jela' not in stabla) # True
```

> TLDR: operatori pripadnosti i identiteta u Pythonu
>
> - `in` vraća `True` ako se određeni element nalazi u kolekciji (npr. listi, stringu, setu, rječniku)
> - `is` vraća `True` ako se objekti nalaze na istoj memorijskoj lokaciji
> - `==` vraća `True` ako su objekti jednaki odnosno ako su im vrijednosti literala koje sadrže jednake

Što se tiče redoslijeda evaluacije logičkih operatora kada se kombiniraju, Python će:

1. prvo evaluirati `not`
2. zatim `and`
3. na kraju `or`

```
print(True or False and False) # evaluira se kao: "True or (False and False)" => "True or False" => True

print(not True and False or False) # evaluira se kao: "((not True) and False) or False" => "(False and False) or False" => "False or False" => False
```

3.3 Upravljanje tokom izvođenja programa

Kontrola toka (*eng. flow control*) odnosi se na programske konstrukte koji omogućuju izvršavanje određenih dijelova koda ovisno o zadanim uvjetima. U Pythonu se, kao i u većini programskih jezika, kontrola toka postiže prvenstveno korištenjem **Selekcija** (*eng. selection*) i **Iteracija** (*eng. iteration*).

3.3.1 Selekcije

Selekcija se definira korištenjem `if`, `elif` i `else` naredbi.

Logička pravila su ista kao i u većini programskih jezika, međutim treba obratiti pažnju na specifičnosti Python sintakse kao što su **indentacija koda** (*eng. code indentation*).

Indentacija koda je obavezna u Pythonu i koristi se za označavanje blokova koda.

Blok koda se označava uvlačenjem koda za **N praznih mjesta (razmaka)** (N je najčešće 2 ili 4) ili **jedan tabulator** (*eng. tab*). Python ne koristi vitičaste zagrade `{}` kao što je to slučaj u većini programskih jezika (C familija jezika, Java, JavaScript itd.), već koristi **indentaciju koda za definiranje blokova koda**.

`if` naredba u svojoj osnovnoj formi izgleda ovako:

```
if <logički_uvjet>: # zaglavlje
  <blok_naredbi> # tijelo
```

Na primjer, možemo provjeriti je li broj paran ili neparan:

```
a = 5

if a % 2 == 0:
  print("Broj je paran")
else:
  print("Broj je neparan")
```

Indentaciju želimo raditi koristeći **tabulator** - `Tab` (nikako ručni unosi razmaka).

Uočite još sljedeće:

- **nemamo zagrade oko uvjeta/logičkog izrada**, dakle ne pišemo `if (a % 2 == 0)`, već samo `if a % 2 == 0`
- **oznakom `:` označavamo kraj uvjeta/logičkog izrada** i početak bloka koda koji će se izvršiti ako je uvjet ispunjen/neispunjen

Primjer: Ekvivalentan kod u C++ bi izgledao ovako:

```
int a = 5;

if (a % 2 == 0) {
  cout << "Broj je paran" << endl;
} else {
  cout << "Broj je neparan" << endl;
}
```

ili u JavaScriptu...

```
let a = 5;

if (a % 2 == 0) {
  console.log("Broj je paran");
} else {
  console.log("Broj je neparan");
}
```

Ukoliko imamo više od dva uvjeta, koristimo `elif` naredbu:

Sintaksa:

```
if <logički_uvjet_1>:
  <blok_naredbi_1>
elif <logički_uvjet_2>:
  <blok_naredbi_2>
elif <logički_uvjet_3>:
  <blok_naredbi_3>
else:
  <blok_naredbi_else>
```

Primjer:

```
a = 5

if a % 2 == 0:
  print("Broj je paran")
elif a % 2 == 1:
  print("Broj je neparan")
else:
  print("Broj nije ni paran ni neparan")
```

Od korisnika možemo zatražiti unos koristeći `input()` funkciju:

```
a = input("Unesite broj: ")

if a % 2 == 0:
  print("Broj je paran")
elif a % 2 == 1:
  print("Broj je neparan")
else:
  print("Broj nije ni paran ni neparan")
```

Što se dešava ako korisnik unese "1"?

▸ Spoiler alert! Odgovor na pitanje

Uvjetne naredbe možemo gnijezditi, tj. pisati "jednu unutar druge":

```
tajni_broj = 42
broj = int(input("Pogodi broj! "))

if tajni_broj == broj:
  print("Bravo, pogodio si!")
else:
  if broj > tajni_broj:
    print("Manji je od tog broja!")
  else:
    print("Veći je od tog broja!")
print("Pokreni program ponovo za sljedeći pokušaj!")
```

Doseg varijabli

Kod većine popularnih programskih jezika (npr. C, C++, C#, Java, JavaScript) tijela stavka složenih naredbi nisu određena uvlačenjem, nego se grupiranje naredbi provodi vitičastim zagradama ili nekim drugim eksplicitnim oznakama. U tim programskim jezicima naredbe je moguće grupirati i i izvan složenih naredbi, a uvlačenje je proizvoljno i služi isključivo za bolju čitljivost koda.

Python ne dozvoljava "samostojeće" blokove naredbi, što znači da **naredbe ne smijemo uvlačiti izvan složenih naredbi**. Ako pokušamo, Python će baciti grešku `Unexpected indent`.

```
# indentacija bez složenih naredbi nije dozvoljena
  x = 5
  y = 10
  print(x + y) # Greška! Unexpected indent
```

```
# nedostaje indentacija unutar složene naredbe
if True:
x = 5
y = 10
print(x + y) # Greška! expected an indented block after 'if' statement
```

Glavna prednost takvih pravila jest da smo **prisiljeni pisati uredniji kod**, ali moramo biti svjesni da ova sintaksa odstupa od uobičajenih pravila u većini programskih jezika.

Python ima još jedno svojstvo koje ga čini različitim od većine ostalih popularnih jezika. Naime, imena definirana unutar složenih naredbi (npr. `if`, `for`) su u većini programskih jezika vidljiva samo unutar tih naredbi, odnosno lokalnog su dosega (*eng. local scope*) tog bloka koda.

Kod Pythona imena uvedena unutar složene naredbe ostaju dostupna i nakon njenog okončanja. Zato u sljedećem primjeru možemo ispisati vrijednost `x` koje je definirano unutar uvjetnog stavka naredbe `if` čak i ako to ime nije bilo definirano prije te naredbe. S druge strane, ne možemo ispisivati ime `y` jer mu se vrijednost dodjeljuje unutar alternativnog stavka koji se, zbog istinite vrijednosti logičkog izraza, neće izvršiti.

```
if True:
  x = 5
else:
  y = 6

print(x) # 5 (radi, ali u većini jezika bi bila "ReferenceError" greška)
print(y) # NameError: name 'y' is not defined
```

Što će ispisati sljedeći kod?

```
if False:
  x = 5
else:
  y = 6

print(x) # ?
```

▸ Spoiler alert! Odgovor na pitanje

Vježba 1: Jednostavni kalkulator

Napišite program koji traži od korisnika unos dva broja (`float`) te jedan od operatora (`+`, `-`, `*`, `/`). Program treba ispisati rezultat operacije nad unesenim brojevima u formatu:

```
Rezultat operacije 5.0 + 3.0 je 8.0
```

Ako korisnik pokuša dijeljenje s nulom, program treba ispisati poruku:

```
Dijeljenje s nulom nije dozvoljeno!
```

Ako korisnik unese nepodržani operator, program treba ispisati poruku:

```
Nepodržani operator!
```

Vježba 2: Prijestupna godina

Napišite program koji traži unos godine i provjerava je li godina prijestupna. Godina je prijestupna ako:

- je djeljiva s 4, ali ne sa 100 ili
- godina je djeljiva sa 400

Ako godina zadovoljava ove uvjete, program treba ispisati poruku:

```
Godina ____. je prijestupna.
```

Ako godina nije prijestupna, program treba ispisati poruku:

```
Godina ____. nije prijestupna.
```

3.3.2 Iteracije (Petlje)

Iteracije ponavljaju dijelove koda. U Pythonu postoje dvije osnovne vrste: `for` i `while` petlje, a tijelo petlje (blok koda) sadrži naredbe koje se ponavljaju. **Tijelo petlje definiramo *indentiranim* blokom koda**,a svako ponavljanje izvođenja tijela petlje odgovara **jednom prolazu kroz iterativni postupak**.

`while` petlja

Počet ćemo s `while` petljom budući da je jednostavnija. U **osnovnom** i **najčešćem** slučaju ta naredba se sastoji od samo jednog stavka.

Sintaksa:

```
while <uvjetni_izraz>: # zaglavlje osnovnog stavka
  <naredbe> # tijelo osnovnog stavka
```

Python provjerava uvjet iz zaglavlja osnovnog retka. Ako je ta vrijednost `False`, tijelo stavka se preskače i izvođenje se nastavlja prvom naredbom iza složene naredbe `while` petlje. Drugim riječima, može se dogoditi da se tijelo petlje uopće ne izvrši.

Za razliku od `if` naredbe gdje se uvjetni izraz izvodi najviše jednom, u `while` petlji se uvjetni izraz izvodi **svaki put prije izvršavanja tijela petlje**. Ako je uvjetni izraz `True`, tijelo petlje se izvršava, a zatim se ponovno provjerava uvjetni izraz. Ovaj postupak se ponavlja sve dok uvjetni izraz ne postane `False`.

Primjer: jednostavni program koji ispisuje kvadrate brojeva od 1 do 10 koristeći `while` petlju:

```
# inicijaliziramo vrijednost broja koji ćemo kvadrirati
brojač = 1

# petlja se nastavlja sve dok je brojač manji od 11
while brojač < 11:
  print(brojač ** 2) # ispisujemo kvadrat broja
  brojač += 1 # povećavamo brojač za 1
print("Gotovo!")
```

Koliko puta će se izvršiti sljedeća petlja?

```
brojač = 1

while brojač <= 10:
  print(brojač ** 2)
```

▸ Spoiler alert! Odgovor na pitanje

Najčešći use-case `while` petlje je kada **nije poznat broj ponavljanja unaprijed**. U takvim slučajevima petlja se obično koristi za čitanje podataka sve dok se ne zadovolji neki uvjet za prekid.

Primjer: (opća struktura `while` petlje):

```
while <uvjet_za_prekid_nije_zadovoljen>:
  <naredbe>
  <naredbe>
  <naredbe>
  <naredbe>
```

- Kako bi izašli iz petlje, možemo koristiti naredbu `break`, koja odmah prekida izvođenje petlje i nastavlja s prvom naredbom izvan petlje.
- Kako bi prekinuli trenutnu iteraciju petlje i započeli sljedeću, koristimo naredbu `continue`.

Primjer: Beskonačna petlja (sve dok uvjet nije zadovoljen) s prekidom:

```
while True:
  <naredbe>
  <naredbe>
  <naredbe>
  <naredbe>
  if <uvjet_za_prekid_zadovoljen>:
    break
```

Primjer: Preskakanje iteracije gdje je uvjet zadovoljen (izostavljanje parnih brojeva pri ispisu brojeva od 1 do 10):

```
brojač = 0

while brojač < 10:
  brojač += 1
  if brojač % 2 == 0:
    continue # preskače ispis parnih brojeva
  print(brojač) # ispisuje samo neparne brojeve
```

Petlje se, kao i selekcije mogu ugnijezditi, odnosno mogu se nalaziti unutar tijela drugih složenih naredbi.

Vježba 3: Pogađanje broja sve dok nije pogođen

Implementirajte igru pogađanja broja u rasponu od 1 do 100. Korisnik unosi svoj pokušaj, a program nakon svakog unosa ispisuje poruku koja označava je li uneseni broj veći, manji ili jednak traženom broju. Igra traje dok korisnik ne pogodi točan broj.

- Za izlazak iz igre koristite pomoćnu `bool` varijablu `broj_je_pogoden`.
- Na kraju ispišite korisniku poruku: "Bravo, pogodio si u __ pokušaja".

Vježba 4: Zbrajanje unesenih brojeva

Napišite program koji traži od korisnika unos cijelih brojeva sve dok korisnik ne unese broj `0`. Nakon unosa `0`, program treba ispisati zbroj svih prethodno unesenih brojeva.

`for` petlja

Ako je **broj ponavljanja unaprijed poznat**, tada je najpraktičnije definirati `for` petlju, koju ćemo najčešće upotrebljavati **(1) u sprezi s rasponom** `range` ili **(2) za iteraciju kroz pobrojive objekte** (npr. liste, n-torke, znakovne nizove, rječnike, skupove, itd.).

Raspon `range` je složeni podatkovni tip koji modelira slijed (*eng. sequence*) cijelih brojeva s konstantnim prirastom Tako će sljedeća naredba ispisati slijed brojeva od 0 do 9:

> Napomena: Drugi sekvencijalni podatkovni tipovi su liste (`list`) i n-torke (`tuple`); o njima više u poglavlju o kolekcijama.

```
for i in range(10):
  print(i) # ispisuje brojeve od 0 do 9

# range je pobrojivi objekt koji generira brojeve od 0 do 9 te moguće ga je pohraniti u varijablu
raspon = range(10)
for i in raspon:
  print(i) # ispisuje brojeve od 0 do 9
```

Ukoliko želimo ispisati raspon koji počinje od nekog drugog broja, možemo proslijediti dva argumenta funkciji `range(početna_vrijednost, krajnja_vrijednost)`:

```
for i in range(1, 11):
  print(i) # ispisuje brojeve od 1 do 10
```

`range` funkcija prima maksimalno tri argumenta: **početnu vrijednost**, **krajnju vrijednost** i **korak**.

- ako korak nije naveden, pretpostavlja se da je `1`
- ako je početna vrijednost izostavljena, pretpostavlja se da je `0`

Sintaksa:

```
range(početna_vrijednost, krajnja_vrijednost, korak) # krajnja vrijednost nije uključena
```

Početna vrijednost je uključena u raspon, a krajnja vrijednost nije. Dakle, `range(1, 11)` generira brojeve od 1 do 10.

Za objekt tipa `range` kažemo da je pobrojiv (ili iterabilan) jer je moguće uzastopno dohvaćati njegove elemente. U Pythonu, `for` petlja se jako često koristi za iteriranje kroz pobrojive objekte. Raspone ćemo najčešće koristiti upravo u tu svrhu.

Naredba `for` prilikom svakog prolaza kroz petlju uzastopno dohvaća po jedan element zadanog **pobrojivog objekta** i pridružuje ga **upravljačkom** (ili *obilazećem*) imenu.
Kao što vidimo, i pobrojivi objekt i upravljačko ime koje prima njegove elemente zadajemo u zaglavlju naredbe `for`.

Za razliku od spomenutih drugih jezika, Python nema varijantu klasične C-style `for` petlje (*inicijalizacija, uvjet, inkrementacija*). Umjesto toga, koristi koncept pobrojivih objekata i `for` petlju za iteraciju kroz njih.

Sintaksa:

```
for `upravljacko_ime` in `pobrojivi_objekt`:
  <tijelo>
```

Primjer: ispis tablice kvadrata brojeva od 1 do 10:

```
for x in range(1, 11):
  print(x ** 2) # 1 4 9 16 25 36 49 64 81 100
```

Primjer: ispis svakog slova u riječi "cvrčak":

```
for slovo in "cvrčak":
  print(slovo) # c v r č a k
```

Vidimo da je znakovni niz (`str`) također pobrojiv objekt pa se može koristiti u petlji na isti način.

Rekli smo da, ako konstruktoru objekta `range` proslijedimo treći argument, on određuje korak odnosno **veličinu promjene između uzastopnih vrijednosti**. Ako je broj pozitivan, range će se povećavati (inkrementirati), a ako je negativan, smanjivati (dekrementirati).

```
for i in range(1, 10, 2):
  print(i) # 1 3 5 7 9 (neparni brojevi od 1 do 10)

for i in range(10, 1, -2):
  print(i ** 2) # 100 64 36 16 4 (kvadrati parnih brojeva od 10 prema 2)
```

Petlje `while` i `for` se mogu gnijezditi, odnosno mogu se naći u tijelu drugih složenih naredbi.

Primjer: Želimo ispisati tablicu množenja za brojeve od 1 do 10, to možemo jednostavno napraviti dvjema ugniježđenim petljama:

```
for redak in range(1, 11):
  ispisRetka = ""
  for stupac in range(1, 11):
    umnozak = redak * stupac
    ispisRetka += f"{umnozak:4}" # f-string za formatirani ispis
  print(ispisRetka)
```

f-string

U ovom primjeru koristimo *f-stringove* za formatiranje ispisa. `f-string` je moderna sintaksa za formatiranje znakovnih nizova u Pythonu koja omogućava "ugrađivanje" varijabli u znakovni niz. Ugrađivanje se vrši pomoću vitičastih zagrada `{}` unutar znakovnog niza s ključnim slovom `f` ispred izraza.

Ukoliko želimo dodatno formatirati vrijednost, možemo koristiti dvotočku `:` i specifikator formata (*eng. format specifier*). U ovom primjeru koristimo specifikator formata `:4` kako bismo osigurali da svaki broj bude ispisivan u polju širine 4 znaka, što pomaže u poravnavanju ispisa u tabličnom obliku.

Sintaksa:

```
f"{varijabla:format_specifier}"
```

Primjer, kako ćemo ispisati brojeve od `1` do `10` (bez `10`) s prefiksom `"Broj: "`:

```
for i in range(1, 11):
  print(f"Broj: {i}")
```

Vježba 5: Analiziraj sljedeće `for` petlje

Pojasnite zašto sljedeća petlja nema (previše) smisla:

```
for i in range(1, 2):
  print(i)
```

Što će ispisati sljedeća petlja?

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

Što će ispisati sljedeća petlja?

```
for i in range(10, 1, -1):
  print(i)
```

Vježba 6: Krenimo "petljati"

1. Napišite program koji ispisuje sumu svih parnih brojeva od 1 do 100 (uključivo).
2. Napišite program koji ispisuje prvih 10 neparnih brojeva u obrnutom redoslijedu.
3. Napišite program koji ispisuje Fibonaccijev niz do 1000. Fibonaccijev niz počinje s brojevima 0 i 1, a svaki sljedeći broj je zbroj prethodna dva broja.

Svaki zadatak riješite `for` i `while` petljom.

3.3.3 Ugrađene strukture podataka

Python nudi nekoliko ugrađenih struktura podataka koje omogućuju pohranu više elemenata u jednoj varijabli. U ovom poglavlju upoznati ćemo se s osnovnim strukturama podataka koje su ugrađene u Python.

Strukture podataka (*eng. data structures*) u Pythonu se često u literaturi nazivaju i **kolekcijama** (*eng. collection*), a možemo ih podijeliti u dvije osnovne kategorije: **sekvencijalne** i **nesekvencijalne (neuređene)**. Ipak, treba naglasiti da je svaka kolekcija u Pythonu ujedno i struktura podataka, ali **svaka struktura podataka nije kolekcija** (npr. `memoryview`, `Enum`)

Sekvencijalne kolekcije nazivamo sekvencijalnim jer njihovim elementima možemo u konstantom vremenu `O(1)` pristupiti rednim brojem tj. indeksom (navedeno vrijedi za ugrađene sekvencijalne tipove).

Redoslijed obilaska elemenata slijednih kolekcija određen je **indeksima**: prvo se obilazi nulti element (`index = 0`), zatim prvi (`index = 1`), i tako dalje sve do kraja kolekcije (`index = N - 1`).

N-torke (eng. Tuple)

N-torke (*eng. tuple*) su jedna od dviju temeljnih **slijednih kolekcija** u Pythonu (druga je lista). N-torke su **nepromjenjive** (*eng. immutable*) kolekcije, što znači da se nakon što su kreirane ne mogu mijenjati. N-torke se u pravilu definiraju pomoću zagrada `()` i elemenata odvojenih zarezom, ali se mogu definirati i **bez zagrada** (npr. *type-casting*)

Primjer:

```
tuple = (1, 2, 3, 4, 5)
print(tuple) # (1, 2, 3, 4, 5)

tuple2 = 1, 2, 3, 4, 5
print(tuple2) # (1, 2, 3, 4, 5)

tuple3 = tuple([1, 2, 3, 4, 5]) # type-casting iz liste
print(tuple3) # (1, 2, 3, 4, 5)
```

N-torke mogu sadržavati elemente različitih tipova:

```
tuple = (1, "cvrčak", 3.14, True)
print(tuple) # (1, 'cvrčak', 3.14, True)
```

Nije moguće dodavati ili brisati elemente iz n-torki, mijenjati poredak elemenata itd. Iako se na prvu čini kao nedostatak, nepromjenjivost objekta može biti korisna kada želimo sačuvati integritet podataka predstavljenih n-torkom te spriječiti slučajne izmjene.

Indeksi u Pythonu počinju od `0`, stoga prvi element n-torke ima indeks `0`, drugi indeks `1`, i tako dalje. Posljednji element n-torke ima indeks `N - 1`, gdje je `N` veličina n-torke odnosno `len(n-torka)`.

Jedna od cool stvari kod Pythona je što podržava i negativno indeksiranje, gdje `-1` predstavlja posljednji element, `-2` pretposljednji, i tako dalje.

```
sastojci = ("jaja", "mlijeko", "brašno", "šećer", "sol")

print(sastojci[0]) # jaja
print(sastojci[1]) # mlijeko
print(sastojci[-1]) # sol
print(sastojci[-2]) # šećer

sastojci[0] = "kvasac" # TypeError: 'tuple' object does not support item assignment - n-torke su nepromjenjive
```

N-torke se mogu indeksirati i "rezati" (*eng. slicing*) na isti način kao što možemo i kod lista, znakovnih nizova, raspona i drugih slijednih kolekcija. Rezanje tj. *slicing* ustvari predstavlja dohvaćanje podniza (podskupa) n-torke na temelju zadanih početnog i završnog indeksa.

Sintaksa:

```
sequence[start:stop:step]
```

- `start` - početni indeks (uključivo), zadani je `0` ako nije naveden
- `stop` - završni indeks (ne uključivo), zadani je `len(sequence)` ako nije naveden.
- `step` - korak tj. inkrement između indeksa, zadani je `1` ako nije naveden

Zbunjujuće kod rezanja može biti što **niti jedan argument nije obavezan**.

Primjer:

```
sastojci = ("jaja", "mlijeko", "brašno", "šećer", "sol")

print(sastojci[1:3]) # ('mlijeko', 'brašno') - dohvati elemente od indeksa 1 do indeksa 3 (ne uključujući indeks 3)
print(sastojci[:3]) # ('jaja', 'mlijeko', 'brašno') - dohvati elemente od početka do indeksa 3 (ne uključujući indeks 3)
print(sastojci[3:]) # ('šećer', 'sol') - dohvati elemente od indeksa 3 do kraja

print(sastojci[::]) # dohvati sve (isto kao i sastojci[:] ili samo print(sastojci))
print(sastojci[::2]) # ('jaja', 'brašno', 'sol') - dohvati svaki drugi element od početka do kraja
print(sastojci[::-1]) # ('sol', 'šećer', 'brašno', 'mlijeko', 'jaja') - dohvati sve elemente u obrnutom redoslijedu
print(sastojci[4:1:-1]) # ('sol', 'šećer', 'brašno') - dohvati elemente od indeksa 4 do indeksa 1 u obrnutom redoslijedu
```

> Napomena: Rezanje n-torki (kao i drugih slijednih kolekcija) vraća novu n-torku koja sadrži odabrane elemente. **Originalna n-torka ostaje nepromijenjena**. Ista pravila vrijede i za liste, znakovne nizove, range objekte i druge slijedne kolekcije.
> Dobar slicing guide

Kako se radi o slijednoj kolekciji, n-torke se mogu iterirati pomoću petlje `for`:

```
sastojci = ("jaja", "mlijeko", "brašno", "šećer", "sol")

for sastojak in sastojci: # sastojak (upravljačko ime),
  print(sastojak)
```

Ukratko, sljedeća tablica prikazuje osnovne karakteristike n-torki (*eng. tuples*):

N-torka (*eng. tuple*)	***Primjer:*** `lokacija = (34.0522, -118.2437)` **ili** `lokacija = 34.0522, -118.2437`
Karakteristika	**Objašnjenje**
Nepromjenjivost (*eng. Immutable*)	N-torke nije moguće mijenjati nakon stvaranja (nema dodavanja, uklanjanja, mijenjanja redoslijeda)
**Uređenost (*eng.*	

***Ordered*)**	Elementi n-torke imaju definirani slijed koji se ne može promijeniti.
Indeksirani elementi (*eng. Indexed*)	Elementima se može pristupiti preko indeksa (npr, `tuple[0]`).
Hashable	N-torke se mogu koristiti kao ključevi rječnika (liste ne mogu!)
Fiksna veličina	Veličina n-torke je fiksna i definira se prilikom izrade
Heterogeni elementi	Može sadržavati različite *non-literals* elemente (npr. integere, stringove, liste, itd.)
Packing/Unpacking	Korisno za "pakiranje" više vrijednosti u jednu varijablu i njihovo "raspakiravanje" u pojedinačne varijable

N-torke možemo definirati na mnogo načina:

- `()` - prazna n-torka
- `(1,)` - n-torka s jednim elementom
- `(1, 2, 3)` - n-torka s tri elementa
- `1, 2, 3` - n-torka s tri elementa (bez zagrada)
- `tuple()` - prazna n-torka
- `tuple([1, 2, 3])` - n-torka iz liste
- `tuple("cvrčak")` - n-torka iz znakovnog niza
- `tuple(range(1, 10))` - n-torka iz raspona
- `tuple((1, 2, 3))` - n-torka iz n-torke
- itd.

Veličinu n-torke možemo dobiti pomoću funkcije `len()`:

```
sastojci = ("jaja", "mlijeko", "brašno", "šećer", "sol")

print(len(sastojci)) # 5
```

> Funkcija `len()` primjenjiva je i na većini drugih kolekcija (liste, znakovni nizovi, rječnici, skupovi...)

Lista (eng. List)

Lista je **promjenjiva** (*eng. mutable*) kolekcija koja omogućuje dodavanje, uklanjanje i mijenjanje elemenata. Liste se u pravilu definiraju pomoću uglatih zagrada `[]` i elemenata odvojenih zarezom. Za razliku od n-torki, liste se mogu mijenjati, npr. možemo naknadno dodati element, ukloniti element ili promijeniti vrijednost elementa na određenom indeksu.

Radi se o jednoj od najčešće korištenih struktura podataka u Pythonu, ali i u programiranju općenito.

Kao i n-torke, liste mogu sadržavati elemente različitih tipova:

```
lista = [1, 2, 3, 4, 5]
raznovrsna_lista = [1, "cvrčak", 3.14, True]
print(lista) # [1, 'cvrčak', 3.14, True]
```

Indeksiranje radimo na isti način kao i kod n-torki:

```
sastojci = ["jaja", "mlijeko", "brašno", "šećer", "sol"]

print(sastojci[0]) # jaja
print(sastojci[1]) # mlijeko
print(sastojci[-2]) # šećer
```

Međutim možemo i mijenjati "naše sastojke":

```
sastojci = ["jaja", "mlijeko", "brašno", "šećer", "sol"]
sastojci[0] = "kvasac"
print(sastojci) # ['kvasac', 'mlijeko', 'brašno', 'šećer', 'sol']

sastojci[-1] = "papar" # negativno indeksiranje kreće s kraja liste
print(sastojci) # ['kvasac', 'mlijeko', 'brašno', 'šećer', 'papar']
```

Liste mogu sadržavati i druge liste:

```
matrica = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

print(matrica[0]) # [1, 2, 3]
print(matrica[1][1]) # 5
```

Ali i n-torke:

```
sastojci = [("jaja", 2), ("mlijeko", 1), ("brašno", 3), ("šećer", 1), ("sol", 1)]

print(sastojci[0]) # ('jaja', 2)
print(sastojci[0][1]) # 2
```

Operacije nad listama najčešće uključuju **dodavanje** i **uklanjanje elemenata**.

Dodavanje elemenata na kraj liste vršimo pomoću metode `append()`, dok se novi elementi mogu dodavati i na određenu poziciju pomoću metode `insert()`:

```
sastojci = ["jaja", "mlijeko", "brašno", "šećer", "sol"]

sastojci.append("kvasac")

print(sastojci) # ['jaja', 'mlijeko', 'brašno', 'šećer', 'sol', 'kvasac']

# ili na određenu poziciju koristeći metodu insert()
sastojci.insert(2, "papar")

print(sastojci) # ['jaja', 'mlijeko', 'papar', 'brašno', 'šećer', 'sol', 'kvasac']
```

Uklanjanje elemenata iz liste vršimo pomoću metode `remove()` - uklanja prvi element s **danom vrijednošću**:

```
sastojci = ["jaja", "mlijeko", "brašno", "šećer", "sol"]

sastojci.remove("mlijeko")

print(sastojci) # ['jaja', 'brašno', 'šećer', 'sol']
```

ili pomoću metode `pop()` - uklanja element s određenim **indeksom** ili posljednji element ako indeks nije naveden:

```
sastojci = ["jaja", "mlijeko", "brašno", "šećer", "sol"]

sastojci.pop() # uklanja posljednji element iz liste; ekvivalentno: sastojci.pop(-1)

print(sastojci) # ['jaja', 'mlijeko', 'brašno', 'šećer']

sastojci.pop(1) # uklanja element na indeksu 1

print(sastojci) # ['jaja', 'brašno', 'šećer']
```

Liste možemo jednostavno iterirati:

```
sastojci = ["jaja", "mlijeko", "brašno", "šećer", "sol"]

for sastojak in sastojci:
  print(sastojak)

# ili koristeći enumerate() funkciju za ispisivanje indeksa
for indeks, sastojak in enumerate(sastojci):
  print(f"{indeks}: {sastojak}")
```

> Napomena: Funkcija `enumerate` dodaje *counter* iterabilnim elementima (korisno kada nam trebaju element, ali i indeks elementa za vrijeme iteracije)

Listama možemo promijeniti redoslijed elemenata koristeći metodu `reverse()` pa i sortirati ih koristeći metodu `sort()`:

```
sastojci = ["jaja", "mlijeko", "brašno", "šećer", "sol"]

sastojci.reverse()

print(sastojci) # ['sol', 'šećer', 'brašno', 'mlijeko', 'jaja']

sastojci.sort()

print(sastojci) # ['brašno', 'jaja', 'mlijeko', 'sol', 'šećer'] - sortira elemente u rastućem redoslijedu (abecedno)
```

Sve `list` metode možete pronaći ovdje.

Lista (*eng. List*)	***Primjer:*** `lista = [1, 2, 3, 4, 5]`
Karakteristika	**Opis**
Promjenjivost (*eng. Mutable*)	Liste je moguće mijenjati nakon izrade
Uređenost (*eng. Ordered*)	Elementi liste imaju definirani slijed koji se može mijenjati
Indeksirani elementi (*eng. Indexed*)	Elementima se može pristupiti preko indeksa (npr, `list[0]`)
Non-Hashable	Liste se ne mogu koristiti kao ključevi rječnika jer su promjenjiva struktura
Dinamička alokacija (*eng. Dynamic allocation*)	Liste se dinamički mijenjaju dodavanjem/oduzimanjem elemenata
Heterogeni elementi	Može sadržavati različite *non-literals* elemente (npr. integere, stringove, liste, itd.)
Fleksibilnost	Fleksibilne strukture koje mogu sadržavati duplikate, različite tipove, ugniježđene strukture itd.

Liste jednako kao i n-torke možemo stvarati na različite načine:

- `[]` - prazna lista
- `[1]` - lista s jednim elementom
- `[1, 2, 3]` - lista s tri elementa
- `list()` - prazna lista
- `list((1, 2, 3))` - lista iz n-torke
- `list("cvrčak")` - lista iz znakovnog niza
- `list(range(1, 10))` - lista iz raspona

- `list([1, 2, 3])` - lista iz liste
- itd.

Rječnik (eng. Dictionary)

Rječnik je **promjenjiva** (*eng. mutable*) kolekcija koja omogućuje pohranu parova ključ-vrijednost (*eng. key-value pairs*). Ključevi su jedinstveni, dok vrijednosti mogu biti bilo koji objekt. Rječnici se u pravilu definiraju pomoću vitičastih zagrada `{}` i parova ključ-vrijednost odvojenih zarezom.

Rječnici nisu uređeni, što znači da redoslijed elemenata nije definiran. To znači da se elementi rječnika ne mogu indeksirati, već se **njima pristupa pomoću ključeva**. Dakle ove strukture podataka **nisu slijedne/sekvencijalne**, **već su asocijativne**.

Asocijativne strukture podataka (eng. Associative data types) su one strukture koje spremaju svoje elemente u obliku parova ključ-vrijednost. Ključ je jedinstven i služi za identifikaciju vrijednosti.

Ključevi su obično znakovni nizovi, ali mogu biti i bilo koji drugi *hashable* objekt (često je to n-torka).

Rječnik najjednostavnije definiramo na sljedeći način:

```
rjecnik = {"ime": "Ivan", "prezime": "Ivić", "dob": 25}
print(rjecnik) # {'ime': 'Ivan', 'prezime': 'Ivić', 'dob': 25}
```

- ako su ključevi znakovni nizovi, moramo ih definirati dvostrukim `"kljuc"` ili jednostrukim navodnim znakovima `'kljuc'`

Pojedinim elementima rječnika pristupamo pomoću ključa:

```
rjecnik = {"ime": "Ivan", "prezime": "Ivić", "dob": 25}

print(rjecnik["ime"]) # Ivan
print(rjecnik["dob"]) # 25

print(rjecnik.dob) # NE! AttributeError: 'dict' object has no attribute 'dob' - ne možemo
pristupiti vrijednosti pomoću točkaste notacije

print(rijecnik[1]) # KeyError: 1 - ključ 1 ne postoji u rječniku
```

Ključevi rječnika moraju biti jedinstveni, ali vrijednosti ne moraju biti:

```
rjecnik = {"ime": "Ivan", "prezime": "Ivić", "dob": 25, "ime": "Marko"}

print(rjecnik) # {'ime': 'Marko', 'prezime': 'Ivić', 'dob': 25} - ključ "ime" s
vrijednošću "Ivan" je zamijenjen s "Marko"
```

U pravilu ne želimo mijenjati ključeve rječnika, ali možemo dodavati nove ključeve i mijenjati vrijednosti postojećih ključeva:

```
rjecnik = {"ime": "Ivan", "prezime": "Ivić", "dob": 25}

rjecnik["adresa"] = "Zagreb"

print(rjecnik) # {'ime': 'Ivan', 'prezime': 'Ivić', 'dob': 25, 'adresa': 'Zagreb'}

rjecnik["dob"] = 26

print(rjecnik) # {'ime': 'Ivan', 'prezime': 'Ivić', 'dob': 26, 'adresa': 'Zagreb'}
```

Rječnike možemo iterirati pomoću `for` petlje:

```
rjecnik = {"ime": "Ivan", "prezime": "Ivić", "dob": 25}

for kljuc in rjecnik: # automatski koristi metodu keys()
  print(kljuc, rjecnik[kljuc]) # ime Ivan, prezime Ivić, dob 25
```

Ključeve i vrijednosti rječnika možemo dohvatiti pomoću metoda `keys()` i `values()`, dok metodom `items()` možemo dohvatiti *ključ-vrijednost* parove:

```
rjecnik = {"ime": "Ivan", "prezime": "Ivić", "dob": 25}

print(rjecnik.keys()) # dict_keys(['ime', 'prezime', 'dob'])
print(rjecnik.values()) # dict_values(['Ivan', 'Ivić', 25])

# dohvaćanje ključeva i vrijednosti pomoću metode items()
for kljuc, vrijednost in rjecnik.items():
  print(kljuc, vrijednost) # ime Ivan, prezime Ivić, dob 25
```

Rječnik možemo definirati na mnogo načina:

- `{}` - prazan rječnik
- `{"ime": "Ivan", "prezime": "Ivić", "dob": 25}` - rječnik s tri ključ-vrijednost para
- `dict()` - prazan rječnik
- `dict(ime="Ivan", prezime="Ivić", dob=25)` - rječnik s tri ključ-vrijednost para
- `dict([("ime", "Ivan"), ("prezime", "Ivić"), ("dob", 25)])` - rječnik iz liste parova ključ-vrijednost

U pravilu, rječnike možemo, osim navođenjem izraza u vitičastim zagradama, stvarati i pozivom konstruktora `dict()` nad pobrojivim argumentom koji sadrži parove ključ-vrijednost:

```
tablica = dict([("rajčica", "povrće"), ("jabuka", "voće")])

print(tablica) # {'rajčica': 'povrće', 'jabuka': 'voće'}
```

Literale malih rječnika je praktično stvarati navođenjem imenovanih argumenata konstruktoru `dict()`:

```
cjenik = dict(ćevapi = 10, pivo = 15, kava = 7)

print(cjenik) # {'ćevapi': 10, 'pivo': 15, 'kava': 7}
```

Uobičajeno je da rječnici sadrže i druge rječnike, ali i liste kao **vrijednosti**:

```
namirnice = {
  "čokolada": ["smeđe", "ukusno", "zdravo"],
  "kelj": ["zeleno", "gorko", "zdravo"],
  "luk": ["bijelo", "smrdljivo", "zdravo"],
  "špek": ["crveno", "slano", "nezdravo"]
}

print(namirnice["čokolada"]) # ['smeđe', 'ukusno', 'zdravo']

print(type(namirnice)) # <class 'dict'>
# ali
print(type(namirnice["čokolada"])) # <class 'list'>
```

Rekli smo da sve ključeve rječnika možemo dohvatiti pomoću metode `keys()`:

```
namirnice = {
  "čokolada": ["smeđe", "ukusno", "zdravo"],
  "kelj": ["zeleno", "gorko", "zdravo"],
  "luk": ["bijelo", "smrdljivo", "zdravo"],
  "špek": ["crveno", "slano", "nezdravo"]
}

print(namirnice.keys()) # dict_keys(['čokolada', 'kelj', 'luk', 'špek'])

for kljuc in namirnice.keys():
  print(kljuc) # čokolada, kelj, luk, špek
```

Međutim, kako možemo dohvatiti samo zdrave namirnice ako nam je poznato da sadrže vrijednost `"zdravo"` unutar liste vrijednosti?

```
for kljuc, vrijednost in namirnice.items(): # koristimo metodu items() za dohvaćanje
ključeva i vrijednosti (parovi)
  if "zdravo" in vrijednost: # provjeravamo nalazi li se "zdravo" u listi vrijednosti
    print(kljuc) # čokolada, kelj, luk
```

Rekli smo da rječnici mogu sadržavati i n-torke kao ključeve:

```
coordinates = {
    (34.0522, -118.2437): "Los Angeles",
    (40.7128, -74.0060): "New York",
    (51.5074, -0.1278): "London"
}
```

Dok vrijednosti mogu biti bilo koji objekti, uključujući i liste ili druge rječnike:

```
coordinates = {
    (34.0522, -118.2437): ["Los Angeles", "California", "USA"], # n-torka : lista
    (40.7128, -74.0060): ["New York", "New York", "USA"], # n-torka : lista
    (51.5074, -0.1278): {"city": "London", "country": "UK"} # n-torka : rječnik
}
```

Rječnik (*eng. Dictionary*)	**Primjer:** `rjecnik = {"ime": "Pero", "prezime" : "Perić"}`
Karakteristika	**Opis**
Promjenjivost (eng. mutable)	Rječnike je moguće mijenjati nakon izrade
Neuređenost (*eng. unordered*) (Python < 3.7)	Prije Pythona 3.7, rječnici nisu održavali redoslijed umetanja.
Uređenost (*eng. ordered*) (Python ≥3.7)	Nakon Pythona 3.7, rječnici čuvaju redoslijed umetanja elemenata
Ključ-vrijednost parovi (eng. key-value pairs)	Asocijativna struktura - podaci se spremaju u obliku ključ-vrijednost parova
Ključevi moraju biti *Hashable*	Ključevi moraju biti *hashable* (npr. brojevi, znakovni nizovi, n-torke), dok vrijednosti mogu biti bilo što. Sam rječnik je *non-hashable*.
Jedinstveni ključevi	Svaki ključ je jedinstven, dupli ključevi se *overwritaju*
Učinkovito pretraživanje po ključu	Omogućuje brz pristup vrijednostima pomoću ključeva; prikladan za pretraživanje i dohvaćanje
Fleksibilnost i heterogenost	Fleksibilne strukture koje mogu sadržavati duple vrijednosti, različite tipove, ugniježđene strukture itd.

Skup (eng. Set)

Posljednja vrsta ugrađenih kolekcija koju ćemo spomenuti su skupovi (*eng. Set*). Skup je **asocijativna kolekcija u kojoj su vrijednosti ujedno i ključevi**. Skupovi su **neuređeni** (*eng. unordered*) skupovi **jedinstvenih elemenata** (*eng. distinct*) - matematički skupovi također ignoriraju duplikate.

Skupovi se ne indeksiraju; već koristimo skupovne operacije poput **ispitivanja pripadnosti** (`in`, `not in`), **unije** (`union`), **presjeka** (`intersection`), **razlike** (`difference`) i dr.

Python nudi dvije vrste skupova: **set** i **frozenset**. `Set` je promjenjiv skup, dok je `frozenset` nepromjenjivi skup. Drugih razlika između ova dva tipa skupova nema.

Skupovi se u pravilu definiraju pomoću **vitičastih zagrada** `{}` i elemenata odvojenih zarezom. **Skupovi nemaju ključ-vrijednost parove kao rječnici!**

```
skup = {1, 2, 3, 4, 5}

print(skup) # {1, 2, 3, 4, 5}

skup_2 = {"banana", "jabuka", "kruška"}

print(skup_2) # {'banana', 'jabuka', 'kruška'}
```

Nad promjenjivim skupovima možemo pozivati metode za ažuriranje slične onima kod lista:

```
skup = {1, 2, 3, 4, 5}
skup.add(6)
print(skup) # {1, 2, 3, 4, 5, 6}

skup.remove(3)
print(skup) # {1, 2, 4, 5, 6}
skup.add(1) # duplikat se neće dodati, skup ostaje nepromijenjen
```

Kao i kod ostalih kolekcija i pobrojivih tipova, tako i sve elemente željenog skupa možemo obići `for` petljom na sljedeći način:

```
skup = {1, 2, 3, 4, 5}

for element in skup:
  print(element)

# jednako tako možemo i koristiti operator `in` za ispitivanje pripadnosti

print(1 in skup) # True
print(6 in skup) # False
```

Metodama `add()` i `remove()` možemo dodavati i uklanjati elemente iz skupa. Metoda `discard()` također uklanja element iz skupa, ali neće baciti iznimku ako element ne postoji u skupu.

```
skup = {1, 2, 3, 4, 5}
skup.discard(3)
print(skup) # {1, 2, 4, 5}

skup.discard(6) # neće baciti iznimku
print(skup) # {1, 2, 4, 5}

skup.remove(6) # KeyError: 6 - element 6 ne postoji u skupu
```

Metoda `union()` vraća uniju dva skupa, metoda `intersection()` vraća presjek dva skupa, dok metoda `difference()` vraća razliku dva skupa:

```
voce = {"🍎", "🍌", "🍐", "🍊"}

povrce = {"🍅", "🥦", "🧅", "🥬"}

print(voce.union(povrce)) # {'🍎', '🍌', '🍐', '🍊', '🍅', '🥦', '🧅', '🥬'}

print(voce.intersection(povrce)) # set() - prazan skup, jer voće i povrće nemaju zajedničkih elemenata
```

Neki botaničari tvrde da rajčica 🍅 pripada voću, a ne povrću. For fun, idemo ju dodati u skup voća.

```
voce.add("🍅")

print(voce.intersection(povrce)) # {'🍅'} - rajčica je voće i povrće (presjek dvaju skupova)

print(voce.difference(povrce)) # {'🍎', '🍌', '🍐', '🍊'} - voće koje nije povrće

print(povrce.difference(voce)) # {'🥦', '🧅', '🥬'} - povrće koje nije voće
```

Skup (*eng. Set*)	***Primjer:*** `skup = {'5', '10', '15'}`
Karakteristika	**Opis**
Promjenjivost (eng. mutable)	Možemo dodavati i brisati elemente nakon izrade (kod *frozenset* ne možemo)
Neuređenost (*eng. unordered*)	Skupovi, poput matematičkih skupova, ne poznaju redoslijed elemenata
Jedinstveni elementi	Skupovi pohranjuju samo jedinstvene elemente, duplikati se brišu automatski
Neindeksirani elementi (eng. Unindexed)	Elementi se ne mogu dohvaćati putem indeksa, samim time ih ne možemo niti rezati
Dinamička alokacija (eng. Dynamic allocation)	Skupovi se dinamički mijenjaju dodavanjem/oduzimanjem elemenata
Non-Hashable	Skupovi su non-hashable (`frozenset` nije), ali elementi u skupu moraju biti hashable (npr. brojevi, znakovni nizovi, n-torke), ali ne i skupovi, liste, ni rječnici.
Podržava operacije nad skupovima	Skupovi podržavaju matematičke operacije kao što su unija, presjek, razlika itd.

Skupove možemo stvarati na različite načine:

- `{}` - prazan skup
- `{1, 2, 3}` - skup s tri elementa
- `set()` - prazan skup
- `set([1, 2, 3])` - skup iz liste
- `set("cvrčak")` - skup iz znakovnog niza - {'k', 'č', 'r', 'a', 'v', 'c'} - primijetite da elementi nisu uređeni
- `set(range(1, 10))` - skup iz raspona
- `set((1, 2, 3))` - skup iz n-torke
- itd.

3.4 Funkcije

Često je u programima niz naredbi potrebno ponoviti više puta. Kod naredbi za kontrolu toka vidjeli smo kako se isti niz operacija može ponoviti više puta unutar petlje. No što ako operacije treba obaviti na više različitih mjesta? U takvim situacijama koristimo **funkcije**.

Funkcije su blokovi koda koji se mogu izvršavati više puta. Funkcije se koriste za grupiranje sličnih operacija kako bi se kod učinio preglednijim i ponovno upotrebljivim. Funkcije se definiraju pomoću ključne riječi `def`, a blok koda koji pripada funkciji mora biti uvučen. Funkcije pozivamo pomoću imena funkcije i zagrada `()`.

Funkcije primaju tzv. **argumente** (ulazne parametre) i mogu vraćati **rezultat**. Argumenti su vrijednosti koje funkcija prima prilikom poziva, dok rezultat predstavlja vrijednost koju funkcija vraća nakon njenog uspješnog izvršavanja. Funkcije koje ne vraćaju nikakav rezultat zapravo vraćaju `None`.

Primjer jednostavne funkcije koja ispisuje poruku:

```
def pozdrav():
  print("Hello, world!")

pozdrav() # Hello, world!
```

Dakle osnovna sintaksa funkcije je:

```
def imeFunkcije(argument1, argument2, ..., argumentN):
  # blok koda
  return rezultat
```

Do sad smo već koristili mnogo funkcija koje dolaze ugrađene u Python, kao što su `print()`, `len()`, `type()`, input() i mnoge druge. Uobičajeno je funkcije koje se nalaze unutar klasa i koje manipuliraju instancama klase (objektima), nazivati **metodama** (*eng. methods*).

Do sad smo vidjeli i metode poput:

- `len()` - vraća duljinu kolekcije
- `append()` - dodaje element na kraj liste
- `remove()` - uklanja element iz liste
- `keys()` - vraća ključeve rječnika

- `values()` - vraća vrijednosti rječnika

Općenito, funkcije mogu primati nula, jedan ili više argumenata koji se navode nakon imena funkcije unutar oblih zagrada. Ako funkciji želimo poslati više argumenata, potrebno ih je međusobno razdvojiti zarezima. Funkcija može imati i **podrazumijevane/*defaultne* vrijednosti** (*eng. default values*) za argumente, što znači da se argumentima može pristupiti i bez navođenja vrijednosti.

Primjer funkcije koja prima dva argumenta:

```
def zbroj(a, b):
  return a + b

print(zbroj(3, 5)) # 8

print(zbroj(3)) # TypeError: zbroj() missing 1 required positional argument: 'b'
```

Primjer funkcije koja ima *defaultne* vrijednosti za argumente:

```
def zbroj(a=0, b=0):
  return a + b

print(zbroj()) # 0
print(zbroj(3)) # 3
print(zbroj(3, 5)) # 8
```

Primjer funkcije koja vraća više vrijednosti:

```
def zbroj_razlika(a, b):
  zbroj = a + b
  razlika = a - b
  return zbroj, razlika

z, r = zbroj_razlika(5, 3)
```

Koji tip podataka vraća funkcija `zbroj_razlika()`?

▸ Spoiler alert! Odgovor na pitanje

Funkcije mogu pozivati druge funkcije, a mogu pozivati i same sebe. Funkcije koje se pozivaju same sebe nazivaju se **rekurzivne funkcije** (*eng. recursive functions*). Rekurzivne funkcije koriste se za rješavanje problema koji se mogu podijeliti na manje probleme istog tipa.

Primjer rekurzivne funkcije koja računa faktorijel broja:

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

print(faktorijel(5)) # 120
```

Idemo definirati funkciju koja će nam izračunati točno vrijeme u lokalnoj vremenskoj zoni, za to ćemo koristiti modul `time`.

```
import time
def točnoVrijeme():
  vrijeme = time.localtime() # funkcija (metoda) koja vraća trenutno vrijeme
  sati = vrijeme.tm_hour # funkcija (metoda) koja vraća trenutni sat
  minute = vrijeme.tm_min # funkcija (metoda) koja vraća trenutnu minutu
  sekunde = vrijeme.tm_sec # funkcija (metoda) koja vraća trenutnu sekundu
  return f"{sati}:{minute}:{sekunde}"

print(točnoVrijeme())
```

Uočite što ćemo dobiti ako funkciju pozovemo bez običnih zagrada:

```
print(točnoVrijeme) # <function točnoVrijeme at <nekaAdresa>>
```

Prisjetimo se specifičnosti opsega varijabli unutar blokova koda u Pythonu. Lokalne varijable definirane unutar blokova koda, npr. kod `if` selekcija koja se izvrši, moguće je dohvatiti i izvan tog bloka koda.

```
a = 10

if a > 5:
  b = 5
print(b) # 5
```

Međutim, lokalne varijable definirane u funkcijskom bloku koda ne mogu se dohvatiti izvan tog bloka koda, čak i ako se funkcija uspješno izvrši (što nije slučaj kod selekcija):

```
def funkcija():
  c = 10
  return "Hello, world!"

funkcija()
print(c) # NameError: name 'c' is not defined
```

Python koristi tzv. **LEGB** pravilo za određivanje opsega varijabli. LEGB je akronim za: **Local**, **Enclosing**, **Global** i **Built-in**. Pretraživanje varijabli započinje u lokalnom opsegu, a zatim se kreće prema globalnom opsegu, ugniježđenim opsezima i na kraju ugrađenim opsezima. Više o tome možete pročitati ovdje.

```
x = "global x"

def vanjska_funkcija():
    x = "enclosing x"

    def unutarnja_funkcija():
        x = "local x"
        print("LINIJA 8: ", x)  # Ovo će ispisivati "local x"

    unutarnja_funkcija()
    print("LINIJA 11: ", x)  # Ovo će ispisivati "enclosing x"

vanjska_funkcija()
print("LINIJA 14: ", x)  # Ovo će ispisivati "global x"
```

Ipak, ovo pravilo možemo prekršiti korištenjem ključne riječi `global` za definiciju globalne varijable unutar funkcije:

```
x = "outsider x"

def funkcija():
    global x
    x = "insider x"
    print("LINIJA 6: ", x)  # Ovo će ispisivati "insider x"
funkcija()
print("LINIJA 9: ", x)  # Ovo će ispisivati "insider x"
```

Funkcije mogu primati **sve tipove podataka kao argumente**, uključujući i kolekcije. Idemo napisati funkciju koja će kao prvi argument primiti listu brojeva, a kao drugi argument broj koji će predstavljati faktor s kojim ćemo potencirati svaki broj iz liste.

```
def potenciranje_faktorom(lista, faktor):
  nova_lista = []
  for broj in lista:
    nova_lista.append(broj ** faktor)
  return nova_lista

print(potenciranje([1, 2, 3, 4, 5], 2)) # [1, 4, 9, 16, 25]
```

Funkcije mogu primati i druge funkcije kao argumente. Ovo je korisno kada želimo da funkcija izvrši neku operaciju nad drugom funkcijom. Primjer funkcije koja prima funkciju kao argument:

```python
def pomnozi_s_dva(x):
  return x * 2

def primjeni_na_listu(funkcija, lista):
  nova_lista = []
  for element in lista:
    nova_lista.append(funkcija(element))
  return nova_lista

print(primjeni_na_listu(pomnozi_s_dva, [1, 2, 3, 4, 5])) # [2, 4, 6, 8, 10]
```

Idemo napisati i jednu matematičku funkciju koja će računati vrijednosti trigonometrijskih funkcija za zadani kut izrađen u radijanima. Za to ćemo koristiti modul `math`.

```python
import math

def trigonometrija(kut):
  radijani = math.radians(kut) # pretvara kut u radijane
  sinus = math.sin(radijani)
  kosinus = math.cos(radijani)
  tangens = math.tan(radijani)
  return sinus, kosinus, tangens # vraća n-torku s vrijednostima trigonometrijskih
funkcija

# Poziv funkcije
kut = 45
sinus, kosinus, tangens = trigonometrija(kut)
print(f"Sinus: {sinus}, Kosinus: {kosinus}, Tangens: {tangens}")
```

To je to za sada...

Na sljedećim vježbama bavit ćemo se nekim naprednijim konceptima u Pythonu, kao što su **klase** i **objekti**, **moduli** i **paketi**, **greške i iznimke**, **rad s datotekama**, **lambda izrazi**, **dekoratori** te **comprehension** sintaksa.

Stay tuned! 🐍

U nastavku skripte slijede zadaci za vježbu.

Vježba 7: Validacija i provjera jakosti lozinke

Napišite program koji traži od korisnika da unese lozinku. Lozinka mora zadovoljavati sljedeće uvjete:

1. ako duljina lozinke nije između 8 i 15 znakova, ispišite poruku "Lozinka mora sadržavati između 8 i 15 znakova".
2. ako lozinka ne sadrži **barem jedno veliko slovo i jedan broj**, ispišite "Lozinka mora sadržavati barem jedno veliko slovo i jedan broj"
3. ako lozinka sadrži riječ `"password"` ili `"lozinka"` (bez obzira na velika i mala slova), ispišite: "Lozinka ne smije sadržavati riječi 'password' ili 'lozinka'"
4. ako lozinka zadovoljava sve uvjete, ispišite "Lozinka je jaka!"

Metode za normalizaciju stringova: `lower()`, `upper()`, `islower()`, `isupper()`.

Provjera je li znakovni niz broj: `isdigit()`

Kod za provjeru dodajte u funkciju `provjera_lozinke(lozinka)`.

Vježba 8: Filtriranje parnih iz liste

Napišite funkciju koja prima listu cijelih brojeva i vraća novu lista koja sadrži samo parne brojeve iz originalne liste.

Primjer:

```
lista = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

print(filtriraj_parne(lista)) # [2, 4, 6, 8, 10]
```

Vježba 9: Uklanjanje duplikata iz liste

Napišite funkciju koja prima listu i vraća novu listu koja ne sadrži duplikate. Implementaciju odradite pomoćnim skupom.

Primjer:

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

print(ukloni_duplikate(lista)) # [1, 2, 3, 4, 5]
```

Vježba 10: Brojanje riječi u tekstu

Napišite funkciju koja broji koliko se puta svaka riječ pojavljuje u tekstu (frekvencija riječi) i vraća rječnik s rezultatima.

Primjer:

```
tekst = "Python je programski jezik koji je jednostavan za učenje i korištenje. Python je vrlo popularan."

print(brojanje_riječi(tekst))

# {'Python': 2, 'je': 3, 'programski': 1, 'jezik': 1, 'koji': 1, 'jednostavan': 1, 'za': 1, 'učenje': 1, 'i': 1, 'korištenje.': 1, 'vrlo': 1, 'popularan.': 1}
```

Vježba 11: Grupiranje elemenata po paritetu

Napišite funkciju koja prima listu brojeva i vraća rječnik s dvije liste: jedna za parne brojeve, a druga za neparne brojeve.

Primjer:

```
lista = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

print(grupiraj_po_paritetu(lista))

# {'parni': [2, 4, 6, 8, 10], 'neparni': [1, 3, 5, 7, 9]}
```

Vježba 12: Obrnite rječnik

Napišite funkciju koja prima rječnik i vraća novi rječnik u kojem su ključevi i vrijednosti zamijenjeni.

Primjer:

```
rjecnik = {"ime": "Ivan", "prezime": "Ivić", "dob": 25}

print(obrni_rjecnik(rjecnik))

# {'Ivan': 'ime', 'Ivić': 'prezime', 25: 'dob'}
```

Vježba 13: Napišite sljedeće funkcije:

1. Funkcija koja vraća n-torku s prvim i zadnjim elementom liste u jednoj liniji koda.

```
lista = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

print(prvi_i_zadnji(lista)) # (1, 10)
```

2. Funkcija koja vraća n-torku s maksimalnim i minimalnim elementom liste, bez korištenja ugrađenih funkcija `max()` i `min()`.

```
lista = [5, 10, 20, 50, 100, 11, 250, 50, 80]

print(maks_i_min(lista)) # (250, 5)
```

3. Funkcija `presjek` koja prima dva skupa i vraća novi skup s elementima koji se nalaze u oba skupa.

```
skup_1 = {1, 2, 3, 4, 5}
skup_2 = {4, 5, 6, 7, 8}

print(presjek(skup_1, skup_2)) # {4, 5}
```

Vježba 14: Prosti brojevi

1. Napišite funkciju `isPrime()` koja prima cijeli broj i vraća `True` ako je broj prost, a `False` ako nije. Prost broj je prirodan broj veći od 1 koji je dijeljiv jedino s 1 i samim sobom.

Primjer:

```
print(isPrime(7)) # True
print(isPrime(10)) # False
```

2. Napišite funkciju `primes_in_range()` koja prima dva argumenta: `start` i `end` i vraća **listu** svih prostih brojeva **u tom rasponu**.

Primjer:

```
print(primes_in_range(1, 10)) # [2, 3, 5, 7]
```

Vježba 15: Pobroji samoglasnike i suglasnike

Napišite funkciju `count_vowels_consonants()` koja prima string i vraća rječnik s brojem samoglasnika i brojem suglasnika u tekstu.

```
vowels = "aeiouAEIOU"
consonants = "bcdfghjklmnpqrstvwxyzBCDFGHJKLMNPQRSTVWXYZ"
```

Primjer:

```
tekst = "Python je programski jezik koji je jednostavan za učenje i korištenje. Python je vrlo popularan."

print(count_vowels_consonants(tekst))

# {'vowels': 30, 'consonants': 48}
```

Vježba 16: Implementacija Dijsktra algoritma za pronalaženje najkraćeg puta

Napišite funkciju `dijkstra(graph, start)` koja prima graf predstavljen kao rječnik susjedstva i početni čvor te vraća rječnik s najkraćim udaljenostima od početnog čvora do svih ostalih čvorova u grafu koristeći Dijsktra algoritam.

Za rješavanje zadatka možete koristiti modul `heapq` za gotovu implementaciju prioritetnog reda.

Primjer ulaznih podataka

```
graph = {
    'A': [('B', 1), ('C', 4)],
    'B': [('A', 1), ('C', 2), ('D', 5)],
    'C': [('A', 4), ('B', 2), ('D', 1)],
    'D': [('B', 5), ('C', 1)]
}
```

Primjer poziva funkcije:

```
print(dijkstra(graph, 'A'))
# {'A': 0, 'B': 1, 'C': 3, D': 4}
```