

Raspodijeljeni sustavi (RS)

Nositelj: doc. dr. sc. Nikola Tanković

Asistent: Luka Blašković, mag. inf.

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



Fakultet informatike u Puli

#1

RS

(1) Programski jezik Python

Raspodijeljeni sustav je svaki računalni sustav koji se sastoji od više povezanih autonomnih računala koji 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, tehnologijama i alatima koji se koriste u njihovom razvoju te naučiti kako razvijati aplikacije s naglaskom na distribuiranu arhitekturu.

Posljednje ažurirano: 30.10.2024.

Sadržaj

- [Raspodijeljeni sustavi \(RS\)](#)
- [\(1\) Programski jezik Python](#)
 - [Sadržaj](#)
- [1. Uvod](#)
- [2. Priprema Python okruženja](#)
 - [2.1 Instalacija Pythona](#)
 - [2.2 Priprema virtualnog okruženja](#)
 - [2.2.1 Instalacija conda alata](#)
- [3. Python osnove](#)
 - [3.1 VS Code okruženje](#)
 - [3.2 Osnove Python sintakse](#)
 - [3.2.1 Varijable](#)
 - [3.2.2 Logički izrazi](#)

- [Aritmetički operatori \(eng. Arithmetic operators\)](#)
- [Operatori pridruživanja \(eng. Assignment operators\)](#)
- [Operatori usporedbe \(eng. Comparison operators\)](#)
- [Logički operatori \(eng. Logical operators\)](#)
- [Operatori identiteta \(eng. Identity operators\)](#)
- [Operatori pripadnosti \(eng. Membership operators\)](#)
- [**3.2.3 Upravljanje tokom izvođenja programa**](#)
 - [Selekcije](#)
 - [Doseg varijabli](#)
 - [Vježba 1: Jednostavni kalkulator](#)
 - [Vježba 2: Prijestupna godina](#)
 - [Iteracije \(Petlje\)](#)
 - [while petlja](#)
 - [Vježba 3: Pogađanje broja sve dok nije pogodjen](#)
 - [Vježba 4: Analiziraj sljedeće while petlje](#)
 - [for petlja](#)
 - [Vježba 5: Napišite program koji će izračunati faktorijel broja](#)
 - [Vježba 6: Analiziraj sljedeće for petlje](#)
- [**3.2.4 Ugrađene strukture podataka**](#)
 - [N-torce \(eng. Tuple\)](#)
 - [Lista \(eng. List\)](#)
 - [Rječnik \(eng. Dictionary\)](#)
 - [Skup \(eng. Set\)](#)
- [**3.2.5 Funkcije**](#)
 - [Vježba 7: Validacija i provjera jakosti lozinke](#)
 - [Vježba 8: Filtriranje parnih iz liste](#)
 - [Vježba 9: Uklanjanje duplikata iz liste](#)
 - [Vježba 10: Brojanje riječi u tekstu](#)
 - [Vježba 11: Grupiranje elemenata po paritetu](#)
 - [Vježba 12: Obrnите rječnik](#)
 - [Vježba 13: Napišite sljedeće funkcije:](#)
 - [Vježba 14: Prosti brojevi](#)
 - [Vježba 15: Pobroji samoglasnike i suglasnike](#)

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. Ovi 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 (Golang)**, koji je popularan izbor za razvoj mikroservisa zbog svoje brzine i ugrađene podrške za konkurentnost, ili Java koja nudi snažnu podršku za višedretvenost (*eng. Multithreading*)—mi smo za ovaj kolegij odabrali **Python** kao preferirani jezik.

Python omogućuje jednostavnu integraciju s postojećim bibliotekama i alatima koji nude unaprijed razvijene komponente prilagođene radu s distribuiranim sustavima. Takav pristup ubrzava razvoj aplikacija, omogućujući developerima da se usmjere na višu razinu apstrakcije bez potrebe za implementacijom osnovnih komponenti. Python jezik predstavlja osnovno znanje koje bi svaki developer trebao steći do kraja studija, a njegova popularnost i široka primjena, kako u industriji tako i u znanosti, čine ga neizostavnim alatom za rješavanje složenih problema i razvoj visokokvalitetnih aplikacija.

2. Priprema Python okruženja

2.1 Instalacija Pythona

Python možete preuzeti i instalirati na više načina, a najjednostavniji način je za većinu korisnika preuzimanje i pokretanje instalacijskog programa sa [službene stranice Pythona](#). Preporuka je odabrati verziju **3.9** 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 environment varijable 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 punog puta do datoteke.

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

```
python --version
```

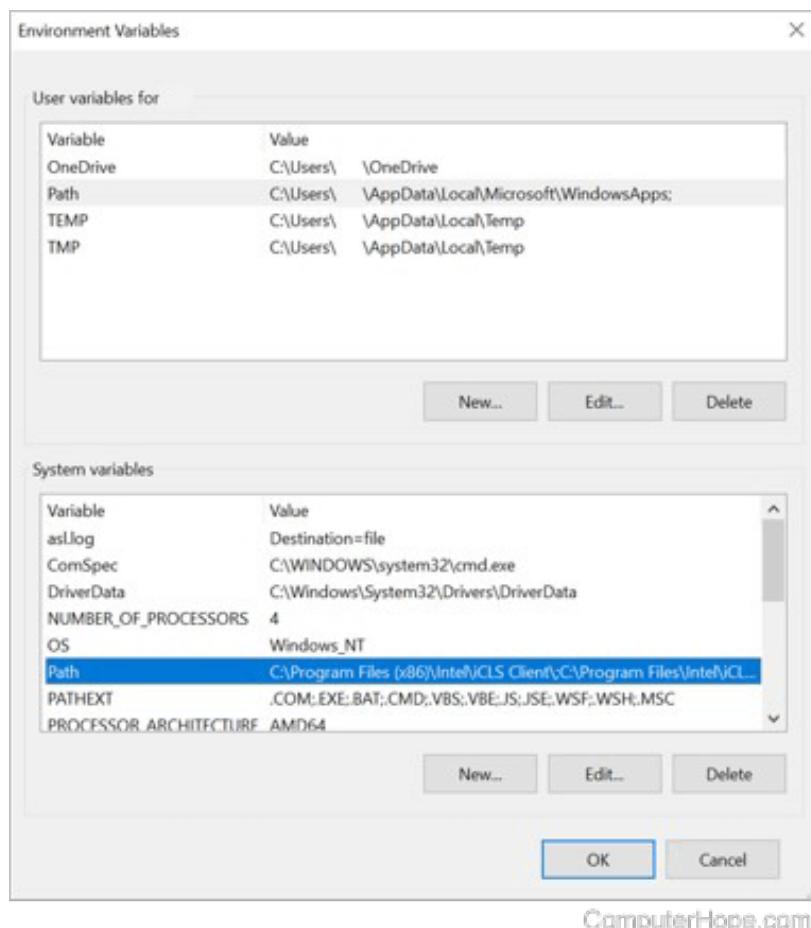
Ako dobijete grešku "Python is not recognized as an internal or external command" to znači da Python nije dodan u PATH. U tom slučaju, najčešće rješenje je ponovo pokrenuti Python installer i odabrati opciju **Add Python to PATH**.

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 je svakako preporuka koristiti **PowerShell terminal** umjesto Command Prompt terminala budući da je izlaskom Windowsa 10, PowerShell postao glavni terminal za Windows.

```
$Env:Path
```

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



ComputerHope.com

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 **brew** package managera, 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
```

Ova naredba će vam reći 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` korisnike, možete otvoriti `~/.bashrc` datoteku kroz `nano` editor:

```
nano ~/.bash_profile
```

i dodati sljedeću liniju na dno datoteke:

```
alias python=python3
```

Za `zsh` korisnike, možete otvoriti `~/.zshrc` datoteku kroz `nano` editor:

```
nano ~/.zshrc
```

i dodati sljedeću liniju na dno datoteke:

```
alias python=python3
```

Spremite izmjene naredbom `ctrl + o`, pritisnite `Enter` i izadjite iz editora naredbom `ctrl + x`. Nakon toga pokrenite sljedeću naredbu kako bi se promjene primijenile:

```
source ~/.bashrc
```

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]
```

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

2.2 Priprema virtualnog okruženja

Virtualno okruženje (*eng. Virtual Environment*) je tehnologija koja omogućuje kreiranje izoliranog okruženja za naše Python projekte. Virtualno okruženje rješava mnogobrojne probleme koji se javljaju kada radimo na više projekata koji koriste različite verzije Pythona 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* paketni menadžer i okruženje za upravljanje paketima i njihovim ovisnostima. `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` paketni menadžer.

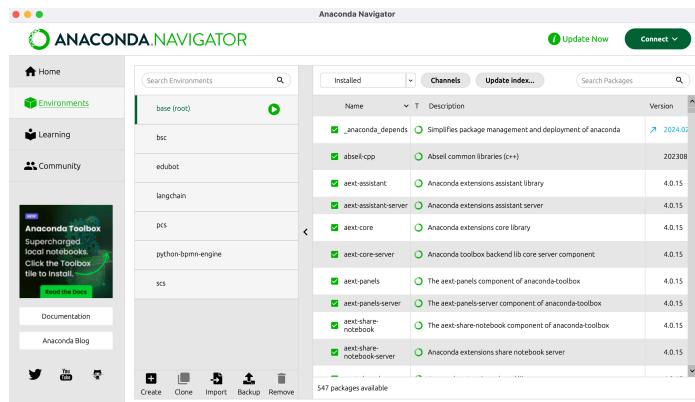
To možete učiniti kroz `Anaconda Navigator` aplikaciju ili preuzimanjem samo `conda` instalacijskog paketa sa [službene stranice](#). Jednostavno odaberite verziju koja odgovara vašem operacijskom sustavu i slijedite upute za instalaciju.

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

```
conda --version
```

Nije loše instalirati i ukupnu 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.



Izgled Anaconda Navigator aplikacije i pregled izrađenih okruženja i 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 projekata. Python je također dinamički tipiziran jezik (*eng. dynamically typed language*) što znači da se tipovi varijabli određuju za vrijeme izvođenja, a ne za vrijeme kompilacije.

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

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

3.1 VS Code okruženje

Za rad s Pythonom preporučujemo korištenje **Visual Studio Code** editora. VS Code je besplatan, open-source IDE (*eng. Integrated development environment*) kojeg razvija 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 upotpunjaju rad s Pythonom u VS Code-u.

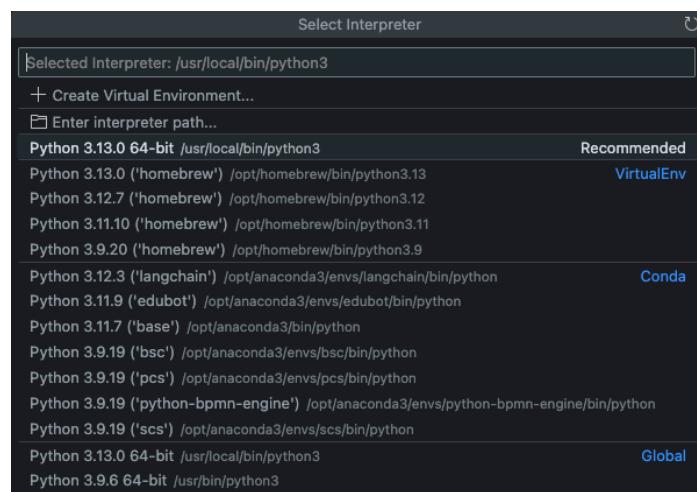
Provjerite jesu li sve ekstenzije uspješno instalirane i aktivirane. Možete ih pronaći u **Extensions** panelu na lijevoj strani VS Code sučelja.

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 promjeniti interpreter klikom na trenutni i odabirom željenog.



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

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 navigaciju u terminalu (Windows, Linux, macOS)

- `cd [ime direktorija]` - promjena direktorija
- `cd ..` - povratak u prethodni direktorij

- `ls` - ispis sadržaja direktorija
- `pwd` - ispis trenutne putanje
- `cls` ili `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 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`.

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

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)

myVariable = 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
```

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

```
# Ovo je komentar

a = 5 # Ovo je komentar
```

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

```
"""

Ovo
je
višelinjski
komentar
"""

# Ili

'''

Ovo
je
isto višelinjski
komentar
'''
```

Međutim, **moguće je** specificirati tip varijable koristeći tzv. [Casting](#):

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

Rezultat će biti isti, no ovime se naglašava tip varijable.

```
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'>
```

Prilikom imenovanja varijabli s više riječi, može se koristiti tehnika po izboru, međutim u Pythonu je uobičajeno koristiti **Camel Case** ili **Snake Case** notaciju.

Camel Case

```
myVariable = 5
```

Pascal Case

```
MyVariable = 5
```

Snake Case

```
my_variable = 5
```

Python dozvoljava i tzv. **Multiple Assignment**, odnosno dodjeljivanje više vrijednosti više varijabli u jednoj liniji 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
```

Može se koristiti i s drugim tipovima varijabli:

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

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

Napomena: Broj varijabli mora odgovarati broju vrijednosti koji se dodjeljuje, inače će Python baciti grešku.

Moguće je i dodjeljivanje iste vrijednosti više varijabli:

```
a = b = c = "same value"

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
```

Primjetite 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

```

`print` naredba vrlo je korisna i često se koristi za ispisivanje poruka u konzolu, ali njena upotreba je prvenstveno u svrhu debugiranja i testiranja. Međutim, na stvarnim projektima, koristit ćemo moćnije alate za debugiranje, poput `logging` biblioteke.

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 Pythona!

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 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*)

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)	5 / 2	2.5
//	Cjelobrojno dijeljenje	5 // 2	2
%	Ostatak pri dijeljenju (modulo)	5 % 2	1
**	Potenciranje	5 ** 3	125

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, realni brojevi mogu nastati i kao rezultat dijeljenja cijelih brojeva:

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

Za vrlo velike ili vrlo 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)
```

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)
print(math.pow(2, 3)) # 8.0 (potenciranje)

print(math.exp(1)) # 2.718281828459045 (e^x)
print(math.log(10)) # 2.302585092994046 (ln(x))

print(math.trunc(3.14)) # 3 (odbacuje decimalni dio)
print(math.ceil(3.14)) # 4 (zaokružuje prema gore)
print(math.floor(3.14)) # 3 (zaokružuje prema dolje)
```

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

```
import math

print(math.isfinite(1.0)) # True (je li broj konačan)
print(math.isinf(1.0)) # False (je li broj beskonačan tj. neizmjerno velik)

print(math.isnan(1.0)) # False (je li broj NaN, tj. Not a Number)
```

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

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 umu da to gotovo nikada nije slučaj jer većina racionalnih brojeva u koje uvrštavamo i decimalne konstante nemaju konačan prikaz u binarnom brojevnem 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
```

U ovakvim slučajevima koristimo funkcije za usporedbu realnih brojeva koje uzimaju u obzir određenu toleranciju:

```
import fractions

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

import decimal

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

Operatore usporedbe moguće je primjenjivati i na većinu ostalih ugrađenih tipova podataka u Pythonu, kao i na korisničke tipove koji podržavaju odgovarajuće operatore, pri čemu će smisao usporedbi ovisiti od tipa do tipa.

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-boolean" 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 pretvaranje 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)
```

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
<code>and</code>	Konjunkcija ili logičko "I" - <code>True</code> ako su oba izraza <code>True</code>	<code>True and False</code>	<code>False</code>
<code>or</code>	Disjunkcija ili logičko "ILI" - <code>True</code> ako je barem jedan izraz <code>True</code>	<code>True or False</code>	<code>True</code>
<code>not</code>	Negacija ili logičko "NE"	<code>not True</code>	<code>False</code>

Izračunavanje logičkih operatora prestaje **čim konačna vrijednost izraza postane jasna**. Uzmimo za primjer izraze:

`False and x`

`True or x`

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 ugniježdenih 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`

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. Operator `not` jednostavno vraća negiranu logičku vrijednost svog argumenta.

- 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) # False

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

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

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

print(0 or 3) # 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` zasebni podatkovni tip, on je ujedno i podtip cijelih brojeva. Stoga se vrijednosti `True` i `False` mogu pojaviti i u aritmetičkom kontekstu, pri čemu se ponašaju kao brojevi 1, odnosno 0, kao što ilustriraju sljedeći primjeri:

```
print(True + True) # 2  
  
print(False + False) # 0  
  
print (True + 1) # 2  
  
print (False * 3) # 0
```

Operatori identiteta (eng. Identity operators)

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

Objekti su pohranjeni u memoriji računala, a varijable su referenca na te objekte. Operator `is` vraća `True` ako su objekti jednaki odnosno ako se objekti nalaze na istoj memorijskoj lokaciji, odnosno `False` ako se objekti nalaze na različitim memorijskim lokacijama.

```
a = [1, 2, 3]  
b = [1, 2, 3]  
  
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 i nisu promjenjivi (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
```

Operatori pripadnosti (eng. Membership operators)

Sve kolekcije Pythona mogu ustanoviti pripadnost zadanog elementa operatorima `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.

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

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']

print('bukva' in stabla) # True

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

TLDR:

- `in` vraća `True` ako se određeni element nalazi u kolekciji (npr. listi, stringu, setu, rječniku)
- `is` vraća `True` ako su objekti jednaki odnosno ako se objekti nalaze na istoj memorijskoj lokaciji
- `==` vraća `True` ako su objekti jednaki odnosno ako su im vrijednosti jednake

3.2.3 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*).

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.

Indentacija koda je **obavezna** u Pythonu i koristi se za označavanje blokova koda. Blok koda se označava uvlačenjem koda za **4 prazna mjesta** (ili 2 ovisno o postavkama) ili **jedan tabulator**. 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 označavanje 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")
```

Primijetite da je blok koda nakon `if` i `else` naredbi uvučen za 4 prazna mjesta. Ovo je obavezno i Python će baciti grešku ako se ne pridržavate ovog pravila.

Indentaciju želimo raditi koristeći **tabulator** - `Tab`.

- **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

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. staviti 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++, Java, JavaScript ili C#) tijela stavka složenih naredbi nisu određena uvlačenjem, nego se grupiranje naredbi provodi vitičastim zagrada ili nekim drugim eksplisitnim 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. Ukoliko to pokušamo, Python će baciti grešku.

```

x = 5
y = 10
print(x + y) # Greška! Unexpected indent

```

```

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. scope*). Kod Pythona imena uvedena unutar složene naredbe ostaju dostupna i nakon njenog okončanja. Zato u sljedećem primjeru možemo ispisati ime `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 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 operator (`+`, `-`, `*`, `/`). 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 uneše 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 dijeljiva 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.
```

Iteracije (Petlje)

Iteracije se koriste za ponavljanje određenih dijelova koda. U Pythonu postoje dvije vrste petlji: `for` i `while`. Programske petlje su složene naredbe koje omogućavaju ponavljanje niza naredbi u tijelu petlje. 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`.

Ilustrirajmo jednostavnim programom koji ispisuje brojeve 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

Vježba 3: Pogađanje broja sve dok nije pogoden

Implementirajte igru pogađanja broja u rasponu od 1 do 100. Svaki pokušaj pogađanja sastoji se od unosa pretpostavljenog broja te ispisa odgovora je li uneseni broj veći, manji ili jednak broju koji treba pogoditi. Igra se nastavlja sve dok korisnik ne pogodi broj.

Za izlazak iz igre koristite pomoćnu `bool` varijablu `broj_je_pogoden`.

Na kraju ispišite korisniku poruku: "Bravo, pogodio si u __ pokušaja".

Vježba 4: Analiziraj sljedeće `while` petlje

Pokušajte pogoditi što će se ispisati tijekom izvođenja sljedeće petlje:

```
broj = 0
while broj < 5:
    broj +=2
    print(broj)
```

Objasnite zašto se prikazana petlja beskonačna:

```
broj = 0
while broj < 5:
    broj += 1
    print(broj)
    broj -= 1
```

Navedite što "ne valja" u sljedećoj petlji:

```
broj = 10
while broj > 0:
    broj -= 1
    print(broj)
    if broj < 5:
        broj += 2
```

for petlja

Ako je broj ponavljanja poznat unaprijed, tada je petlju najpraktičnije izraziti složenom naredbom `for`, koju ćemo najčešće upotrebljavati u sprezi s rasponom `range`.

Raspon `range` je složeni tip podataka koji modelira slijed cijelih brojeva s konstantnim prirastom. Tako će sljedeća naredba ispisati slijed brojeva od 0 do 9:

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

Ukoliko želimo ispisati brojeve od 1 do 10, možemo koristiti sljedeći kod:

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

Dakle, `range` funkcija prima 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`.

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 koristi za iteriranje kroz pobrojive objekte. Raspone ćemo zato najčešće koristiti u petljama.

Naredba `for` prilikom svakog prolaza kroz petlju uzastopno dohvaća po jedan element zadanog pobrojivog objekta i pridružuje ga upravljačkom (ili *obilazećem*) imenu `i`.

Kao što vidimo, i pobrojivi objekt i upravljačko ime koje prima njegove elemente zadajemo u zaglavlju naredbe `for`.

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

Primjer kako ćemo ispisati tablicu 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 kako ćemo ispisati svako slovo u riječi "cvrčak":

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

Vidimo da je znakovni niz također pobrojiv objekt, pa se može koristiti u petlji na ovaj način.

Proslijedimo li konstruktoru raspona tri argumenta, tada će treći argument biti interpretiran kao prirast. Stoga će sljedeća petlja ispisati kvadrate neparnih broja od 1 do 9:

```
for i in range(1, 10, 2):  
    print(i ** 2) # 1 9 25 49 81
```

Petlje `while` i `for` se mogu gnijezditi, odnosno mogu se naći u tijelu drugih složenih naredbi. Na primjer, ako želimo ispisati tablicu množenja brojeva 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}"  
    print(ispisRetka)
```

U ovom primjeru koristimo f-stringove za formatiranje ispisa. `f-string` je moderna sintaksa za formatiranje znakovnih nizova u Pythonu. Ugrađuje vrijednosti varijabli u znakovni niz. Ugrađivanje se vrši pomoću `{}` oznaka unutar znakovnog niza. Ukoliko želimo dodatno formatirati vrijednost, možemo koristiti dvotočku i specifikator formata. U ovom primjeru koristimo specifikator formata `:4` kako bismo osigurali da svaki broj bude ispisivan na 4 mjesta.

Sintaksa:

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

Primjer, kako ćemo ispisati brojeve od 1 do 10 s prefiksom "Broj":

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

Vježba 5: Napišite program koji će izračunati faktorijel broja

Program napišite na dva načina: koristeći `for` i `while` petlje.

Vježba 6: 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(1, 10, 2):
    print(i)
```

Što će ispisati sljedeća petlja?

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

3.2.4 Ugrađene strukture podataka

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

Strukture podataka u Pythonu se često u literaturi nazivaju i kolekcijama, a možemo ih podijeliti u dvije osnovne kategorije: **sekvencijalne i nesekvencijalne (neuređene)**.

Sekvencijalne kolekcija nazivamo sekvencijalnim jer njihovim elementima možemo u konstantom vremenu ($O(1)$) pristupiti **rednim brojem** ili **indeksom**. Redoslijed obilaska elemenata sljednih kolekcija određen je indeksima: prvo se obilazi nulti element, zatim prvi, i tako dalje sve do kraja kolekcije.

N-torce (eng. Tuple)

N-torce su jedna od dviju temeljnih sljednih kolekcija u Pythonu. N-torce su **nepromjenjive** (eng. *immutable*) kolekcije, što znači da se nakon što su kreirane ne mogu mijenjati. N-torce se u pravilu definiraju pomoću zagrada `()` i elemenata odvojenih zarezom, ali se mogu definirati i **bez zagrada**.

Primjer:

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

N-torce mogu sadržavati elemente različitih tipova:

```

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

```

N-torce su, poput znakovnih nizova, **nepromjenjive**, dakle nije moguće dodavati ili brisati elemente, mijenjati poredak elemenata itd. Iako se na prvu čini kao nedostatak, nepromjenjivost može biti korisna kada želimo sačuvati integritet podataka predstavljenih n-torkom.

Indeksi u Pythonu počinju od 0, stoga prvi element n-torce ima indeks 0, drugi indeks 1, i tako dalje.

```

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

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

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

```

N-torce se mogu indeksirati i rezati (*eng. slicing*) na isti način kao i znakovni nizovi.

```

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

```

Kako se radi o sljednoj kolekciji, n-torce se mogu iterirati pomoću petlje `for`:

```

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

for sastojak in sastojci:
    print(sastojak)

```

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

N-torka (eng. tuple)	Primjer: <code>lokacija = (34.0522, -118.2437)</code> ili <code>lokacija = 34.0522, -118.2437</code>
Karakteristika	Opis
Nepromjenjivost (eng. Immutable)	N-torce 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, <code>tuple[0]</code>).
Parcelable	N-torce se mogu koristiti kao ključevi rječnika (eng. Dictionary)
Fiksna veličina	Veličina n-torke je fiksna i definira se prilikom izrade
Heterogeni elementi	Može sadržavati različite elemente (npr, integers, strings, lists, itd.).
Packing/Unpacking	Korisno za pakiranje više vrijednosti u jednu varijablu i njihovo raspakiranje u pojedinačne varijable

N-torce 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
```

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-torce, 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 mijenjati naše sastojke:

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

sastojci[-1] = "papar"
print(sastojci) # ['kvasac', 'mlijeko', 'brašno', 'šećer', 'papar']
```

Naše 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-torce:

```
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()`:

```

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 određenom **vrijednošću**:

```

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

sastojci.remove("mlijeko")

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

```

Ili 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, jednako kao i sastojci.pop(-1)

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

sastojci.pop(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}")

```

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)

```

Lista (eng. List)	Primjer: <code>lista = [1, 2, 3, 4, 5]</code>
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, <code>list[0]</code>).
Hashable	N-torke se mogu koristiti kao ključevi rječnika (eng. Dictionary)
Dinamička alokacija (eng. Dynamic allocation)	Liste se dinamički mijenjaju dodavanjem/oduzimanjem elemenata
Heterogeni elementi	Može sadržavati različite elemente (npr, integers, strings, lists, tuple itd.).
Fleksibilnost	Fleksibilne strukture koje mogu sadržavati duplike, 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 pristupa elementima pomoću ključeva. Dakle ove strukture podataka **nisu sekvencijalne, već su asocijativne**.

Asocijativne strukture podataka 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 nepromjenjivi objekt (npr. n-torka).

Rječnik najjednostavnije definiramo na sljedeći način:

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

Pojedinim elementima rječnika pristupamo pomoću ključa:

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

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

print(rjecnik[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 petlje `for`:

```
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 parove ključ-vrijednost:

```
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

U pravilu, rječnike možemo, osim navođenjem izraza u vitičastim zagradama, stvarati i u 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

```

Rječnik (eng. Dictionary)	Primjer: <code>rjecnik = {"ime": "Pero", " prezime" : "Perić"}</code>
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 <i>hashable</i> (npr. strings, numbers, tuples), vrijednosti mogu biti bilo koja vrijednost.
Jedinstveni ključevi	Svaki ključ je jedinstven, dupli ključevi se <i>overwritaju</i>
Učinkovito pretraživanje po ključu	Omogućuje brz pristup vrijednostima pomoću ključeva, prikidan 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 (matematički skupovi također ne dozvoljavaju duplike).

Na skupove u pravilu ne primjenjujemo indeksiranje, već koristimo skupovne operacije poput **ispitivanja pripadnosti, unije, presjeka, razlike** i dr.

Python nudi dvije vrste skupova: **set** i **frozenset**. **Set** je promjenjiv skup, dok je **frozenset** nepromjenjiv 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!**

```
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 standardnom iteracijom 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("apple")

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

print(voce.difference(povrce)) # {'apple', 'banana', 'pear', 'orange'} - voće koje nije povrće

print(povrce.difference(voce)) # {'cucumber', 'onion', 'lettuce'} - povrće koje nije voće

```

Skup (eng. Set)	Primjer: <code>skup = {5, 10, 15}</code>
Karakteristika	Opis
Promjenjivost (eng. mutable)	Možemo dodavati i brisati elemente nakon izrade (kod <i>frozenset</i> ne možemo)
Neuređenost (eng. unordered)	Skupovi, poput matematičkih skupova, ne poznaju redoslijed elemenata
Jedinstveni elementi	Skupovi pohranjuju samo jedinstveni elementi, duplikati se brišu automatski
Neindeksirani elementi (eng. Unindexed)	Elementi se ne mogu dohvaćati putem indeksa, samim time niti <i>slicat</i>
Dinamička alokacija (eng. Dynamic allocation)	Skupovi se dinamički mijenjaju dodavanjem/oduzimanjem elemenata
Hashable	Elementi u skupu moraju biti hashable (npr. nizovi, brojevi, torke), ali skupovi su promjenjivi.
Podržava operacije nad skupovima	Skupovi podržavaju matematičke operacije kao što su unija, presjek, razlika i simetrična razlika.

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.

3.2.5 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 mesta? 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 pregleđnjim 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 zagrade `()`.

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 manipulirajuinstancama 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 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 podrazumijevane vrijednosti za argumete:

```

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

Primjer rekursivne 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 funkcijском bloku koda ne mogu se dohvatiti izvan tog bloka koda, čak i ako se funkcija uspješno izvrši.

```

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 [ovde](#).

```

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"

```

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:

```

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](#).

```

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 naprednjim konceptima u Pythonu, kao što su **klase i objekti, moduli i paketi, greške i iznimke, rad s datotekama, lambda izrazi, dekoratori te comprehension** sintaksa.

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 listu 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 duplike. Implementaciju odradite pomoćnim skupom.

Primjer:

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

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

Vježba 10: Brojanje riječi u tekstu

Napišite funkciju koja broji koliko se puta svaka riječ pojavljuje u tekstu 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, 'programske': 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(obići_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 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 do 1 koji je dijeljiv jedino sa 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}
```

Raspodijeljeni sustavi (RS)

Nositelj: doc. dr. sc. Nikola Tanković

Asistent: Luka Blašković, mag. inf.

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



(2) Napredniji Python koncepti

#2

RS

U ovoj skripti fokusirat ćemo se na naprednije aspekte programskog jezika Python, koji će vam biti korisni kako za jednostavniju implementaciju rješenja u okviru ovog kolegija, tako i za općenito učinkovitiji rad s Pythonom. Konkretno, naučit ćemo kako koristiti anonimne lambda funkcije, raditi s funkcijama višeg reda, koristiti module, pisati comprehension sintaksu za bržu izgradnju struktura podataka te kako raditi s klasama i objektima.

Posljednje ažurirano: 24.11.2024.

Sadržaj

- [Raspodijeljeni sustavi \(RS\)](#)
- [\(2\) Napredniji Python koncepti](#)
 - [Sadržaj](#)
- [1. Lambda funkcije](#)
 - [1.1 Lambda funkcije kao argumenti drugim funkcijama](#)
 - [1.2 Funkcije višeg reda](#)
 - [1.2.1 Funkcija `map`](#)
 - [1.2.2 Funkcija `filter`](#)
 - [1.2.3 Funkcije `any` i `all`](#)
- [2. Izgradnja struktura kroz `comprehension` sintaksu](#)
 - [2.1 List comprehension](#)
 - [2.2 Dictionary comprehension](#)
- [3. Zadaci za vježbu - lambda izrazi, funkcije višeg reda i comprehension sintaksa](#)
 - [Zadatak 1: Lambda izrazi](#)

- [Zadatak 2: Funkcije višeg reda](#)
- [Zadatak 3: Comprehension sintaksa](#)
- [4. Klase i objekti](#)
 - [4.1 Definiranje klase i stvaranje objekta](#)
 - [4.2 Konstruktor klase](#)
 - [4.3 Metode klase](#)
 - [4.4 Nasljeđivanje](#)
- [5. Moduli i paketi](#)
 - [5.1 Moduli](#)
 - [5.1.1 Ugrađeni moduli](#)
 - [5.2 Paketi](#)
- [6. Zadaci za vježbu - Klase, objekti, moduli i paketi](#)
 - [Zadatak 4: Klase i objekti](#)
 - [Zadatak 5: Moduli i paketi](#)

1. Lambda funkcije

Lambda funkcije su anonimne funkcije koje se u pravilu koriste za jednokratne, male operacije. Funkcije su anonimne jer se ne dodjeljuju imena kao što je to slučaj kod običnih funkcija. Lambda funkcije mogu primiti proizvoljan broj argumenata, ali mogu sadržavati samo jedan izraz (*eng. expression*).

Sintaksa lambda funkcije je sljedeća:

```
lambda arguments : expression
```

Primjerice: Klasičnu funkciju za kvadriranje broja možemo napisati ovako:

```
def kvadriraj(x):
    return x ** 2

print(kvadriraj(5)) # 25
```

Kod lambda funkcije, potrebno je izbaciti ključnu riječ `def` i ime funkcije, a umjesto toga koristimo ključnu riječ `lambda`:

```
lambda x: x ** 2
print((lambda x: x ** 2)(5)) # 25
```

Lambda funkcije se mogu pohranjivati u varijable, a zatim pozivati preko tih varijabli:

```
kvadriraj = lambda x: x ** 2

print(kvadriraj(5)) # 25
```

Lambda funkcije mogu primiti više argumenata:

```
zbroji = lambda x, y: x + y

print(zbroji(5, 3)) # 8

zbroji_kvadrate = lambda x, y: x ** 2 + y ** 2

print(zbroji_kvadrate(3, 4)) # 25
```

Ali i ne moraju primiti niti jedan argument:

- Sljedeći primjer nema puno smisla jer je moguće samo pohraniti vrijednost "Pozdrav!" u varijablu i ispisati je, ali je koristan za demonstraciju:

```
pozdrav = lambda: "Pozdrav!"

print(pozdrav()) # Pozdrav!
```

U lambda funkcijama, kao i običnim, možemo postaviti zadane vrijednosti za argumente:

```
pozdrav = lambda ime="Ivan": f"Pozdrav, {ime}!" # koristimo f-string za formatiranje
stringa

print(pozdrav()) # Pozdrav, Ivan!
print(pozdrav("Marko")) # Pozdrav, Marko!
```

- pa i više njih:

```
circle_area = lambda r=1, pi=3.14: pi * r ** 2

print(circle_area()) # 3.14
print(circle_area(2)) # 12.56
```

Ako lambda funkcija ima više argumenata, argumente s zadanim vrijednostima postavljamo na kraj.

```
multiplier = lambda x, factor = 2: x * factor

print(multiplier(5)) # 10
print(multiplier(5, 3)) # 15
```

Naravno, kao i obične funkcije, lambda funkcije je moguće koristiti sa svim tipovima podataka, uključujući i strukture podataka:

```
tekst = "Ovo je neki tekst"

print((lambda x: x.upper())(tekst)) # OVO JE NEKI TEKST
```

1.1 Lambda funkcije kao argumenti drugim funkcijama

Prava snaga lambda funkcija dolazi do izražaja kada ih koristimo kao argumente drugim funkcijama. To je korisno jer nam omogućuje da napišemo funkcije višeg reda, tj. funkcije koje primaju druge funkcije kao argumente.

Dodatno, moguće ih je koristiti kao anonimne funkcije unutar drugih funkcija, iz opet istog razloga, kako bi se izbjeglo definiranje dodatnih funkcija koje se koriste samo jednom.

Primjerice: Želimo napisati funkciju koja će primati **listu brojeva i funkciju koja će se primijeniti na svaki element** liste. To možemo napraviti ovako:

```
def primjeni_na_sve(lista, funkcija):
    rezultat = []
    for element in lista:
        rezultat.append(funkcija(element)) # u novu listu dodajemo rezultate funkcije
    primijenjene na svaki element
    return rezultat
```

Što je ovdje `funkcija`? Što god želimo i definiramo kao funkciju. Primjer, želimo kvadrirati svaki element liste, za to možemo definirati malo anonimnu lambda funkciju:

```
lambda x: x ** 2 # za svaki element x vraća x na kvadrat
```

- i proslijedimo je kao argument funkciji `primjeni_na_sve`:

```
print(primjeni_na_sve([1, 2, 3, 4], lambda x: x ** 2)) # [1, 4, 9, 16]
```

- ili želimo primijeniti funkciju koja potencira vrijednost na 3. potenciju:

```
print(primjeni_na_sve([1, 2, 3, 4], lambda x: x ** 3)) # [1, 8, 27, 64]
```

- funkciju je moguće pohraniti i u varijablu te potom proslijediti:

```
uvecaj_za_5 = lambda broj: broj + 5

print(primjeni_na_sve([1, 2, 3, 4], uvecaj_za_5)) # [6, 7, 8, 9]
```

Lambda funkcija može biti i povratna vrijednost neke funkcije. Primjerice, funkcija `kvadriraj` vraća lambda funkciju koja kvadrira broj:

```

def kvadriraj():
    return lambda x: x ** 2

kvadriraj_broj = kvadriraj()

print(kvadriraj_broj(5)) # 25

```

OK, nema puno smisla. Međutim, možemo definirati: **funkciju** koja će vraćati: **funkciju** koja će primati broj i množiti ga s nekim faktorom:

```

def mnozi_sa_faktorom(faktor):
    return lambda x: x * faktor

mnozi_sa_5 = mnozi_sa_faktorom(5) # ovo je ekvivalentno: mnozi_sa_5 = lambda x: x * 5

print(mnozi_sa_5(3)) # 15

```

Kako ovo radi?

1. Funkcija `mnozi_sa_faktorom` prima `faktor` kao argument i vraća lambda funkciju koja prima `x` kao argument i množi ga s `faktor`.
2. U varijablu `mnozi_sa_5` pohranjujemo rezultat poziva funkcije `mnozi_sa_faktorom` s argumentom `5`. Rezultat poziva te funkcije je lambda funkcija koja množi broj s 5.
3. Pozivamo funkciju `mnozi_sa_5` s argumentom `3` i dobivamo rezultat `15`.

Ako želimo, možemo definirati i uvjete unutar lambda funkcije:

Sintaksa je sljedeća:

```
lambda arguments: expression if condition else expression
```

Primjerice: Želimo kvadrirati broj samo ako je paran:

```
kvadriraj_parne = lambda x: x ** 2 if x % 2 == 0 else x
```

- ili želimo vratiti duljinu znakovnog niza ako je duljina veća od 5, inače vraćamo sam znakovni niz:

```
dulji_od_5 = lambda niz: len(niz) if len(niz) > 5 else niz
```

- ili želimo vratiti "paran" ako je broj paran, inače "neparan":

```
paran_neparan = lambda x: "paran" if x % 2 == 0 else "neparan"
```

1.2 Funkcije višeg reda

Funkcije višeg reda (*eng. Higher-order functions*) su **funkcije koje primaju druge funkcije kao argumente** ILLI **vraćaju druge funkcije kao rezultat**.

Lambda funkcije su korisne jer nam omogućuju pisanje funkcija višeg reda bez potrebe za definiranjem dodatnih funkcija koje se koriste samo jednom.

Primjerice, funkcija `primjeni_na_sve` iz prethodnog primjera je funkcija višeg reda jer prima drugu funkciju kao argument.

Funkcije višeg reda su korisne jer omogućuju pisanje modularnog koda, tj. koda koji je podijeljen u manje, samostalne dijelove koji obavljaju specifične zadatke.

- Ono što ćemo vjerojatno najčešće raditi, je **koristiti lambda funkcije kao argumente ugrađenim funkcijama višeg reda**, kao što su `map`, `filter`, `reduce`, `sort` itd.

1.2.1 Funkcija `map`

Funkcija `map` prima funkciju i **iterabilni objekt** (npr. listu) i primjenjuje tu funkciju na svaki element tog objekta. Povratna vrijednost je **map objekt** koji se može pretvoriti u listu, tuple ili neki drugi iterabilni objekt.

Sintaksa:

```
map(function, iterables)
```

Primjerice: Želimo kvadrirati svaki element liste:

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

kvadriraj = lambda x: x ** 2

kvadrirana_lista = list(map(kvadriraj, lista)) # map vraća map objekt, zato koristimo
list() za pretvaranje u listu

# ili kraće:

kvadrirana_lista = list(map(lambda x: x ** 2, lista))
```

Kako ovo radi?

1. `map` je funkcija višeg reda koja prima lambda funkciju koja kvadrira broj (`lambda x: x ** 2`) i listu `[1, 2, 3, 4]`.
2. `map` primjenjuje tu funkciju na svaki element liste i vraća map objekt.
3. `list` pretvara map objekt u listu.

Što ako želimo izvući određeni ključ iz neke liste rječnika i spremiti ga u novu listu?

Primjer: Imamo listu studenata s imenom, prezimenom i JMBAG-om. Želimo izvući samo JMBAG-ove:

Kako bismo ovo učinili "ručno"? Bez lambda funkcija i funkcija višeg reda (`map`)?

```

studenti = [
    {"ime": "Ivan", " prezime": "Ivić", "jmbag": "0303077889"}, 
    {"ime": "Marko", " prezime": "Marković", "jmbag": "0303099878"}, 
    {"ime": "Ana", " prezime": "Anić", "jmbag": "0303088777"}]

jmbagovi = []

for student in studenti:
    jmbagovi.append(student["jmbag"])

print(jmbagovi) # ['0303077889', '0303099878', '0303088777']

```

Kako bismo to učinili koristeći `map` i lambda funkciju?

```

jmbagovi = list(map(lambda student: student["jmbag"], studenti)) # student je svaki pojedini element (rječnik) liste studenti

print(jmbagovi) # ['0303077889', '0303099878', '0303088777']

```

Kako ovo radi?

- `map` prima lambda funkciju: `lambda student: student["jmbag"]` i listu `studenti`.
- Lambda funkcija prima svaki pojedini element liste `studenti` (rječnik) i vraća vrijednost ključa `"jmbag"`.
- `map` vraća map objekt.
- `list` pretvara map objekt u listu.

`map` funkcija je korisna jer omogućuje brzu i jednostavnu transformaciju podataka. Može primiti proizvoljni broj iterabilnih objekata, ali mora primiti **točno jednu funkciju**.

Funkcije višeg reda, općenito, ne moraju primiti funkciju u obliku lambda funkcije. Možemo koristiti i običnu referencu na funkciju:

```

def zbroji(a, b):
    return a + b

print(list(map(zbroji, [1, 2, 3], [4, 5, 6])))
# ili kraće:
print(list(map(lambda a, b: a + b, [1, 2, 3], [4, 5, 6])))

```

Ovdje koristimo funkciju `zbroji` koja prima dva argumenta i zbraja ih. `map` prima tu funkciju i dvije liste `[1, 2, 3]` i `[4, 5, 6]` i zbraja elemente na istim pozicijama.

Što će ispisati gornji primjer?

```
[5, 7, 9]
```

1.2.2 Funkcija `filter`

Funkcija `filter` prima funkciju koja vraća `True` ili `False` i **iterabilni objekt**. Vraća **filter objekt** koji se može pretvoriti u listu, tuple ili neki drugi iterabilni objekt.

Ova funkcija će filtrirati elemente iterabilnog objekta prema rezultatu funkcije (**predikata**) koja vraća `True` ili `False`.

Sintaksa:

```
filter(function, iterables)
```

Primjerice: Želimo filtrirati samo parne brojeve iz liste:

Prvo kako bismo to učinili "ručno":

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

parni = []

for broj in lista:
    if broj % 2 == 0:
        parni.append(broj)

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

- ili koristeći `filter` i lambda funkciju:

```
parni = list(filter(lambda x: x % 2 == 0, lista))
```

Naravno možemo kombinirati i različite strukture podataka:

```
studenti = [
    {"ime": "Ivan", "prezime": "Ivić", "jmbag": "0303077889", "godina_rodenja": 2000},
    {"ime": "Marko", "prezime": "Marković", "jmbag": "0303099878", "godina_rodenja": 1999},
    {"ime": "Ana", "prezime": "Anić", "jmbag": "0303088777", "godina_rodenja": 2003},
    {"ime": "Petra", "prezime": "Petrić", "jmbag": "0303088777", "godina_rodenja": 2001}
]

rodeni_prije_2000 = list(filter(lambda student: student["godina_rodenja"] < 2000,
                                studenti))
```

Kako bismo ovo zapisali "ručno"?

```

rodeni_prije_2000 = []

for student in studenti:
    if student["godina_rodenja"] < 2000:
        rodeni_prije_2000.append(student)

print(rodeni_prije_2000) # [{'ime': 'Marko', ' prezime': 'Marković', ' jmbag': '0303099878',
' godina_rodenja': 1999}]

```

Uočite prednosti korištenja funkcija višeg reda i lambda funkcija. Operacije za koje nam treba 4-5 linija koda, u pravilu možemo zapisati u jednoj liniji.

Funkcija `filter` je korisna jer omogućuje brzu i jednostavnu filtraciju podataka. Može primiti proizvoljni broj iterabilnih objekata, ali mora primiti **točno jednu funkciju**.

1.2.3 Funkcije `any` i `all`

Funkcije `any` i `all` su također funkcije višeg reda koje primaju iterabilni objekt i vraćaju `True` ili `False`.

- `any` vraća `True` ako je bilo koji (barem jedan) element iterabilnog objekta istinit, inače vraća `False`.
- `all` vraća `True` ako su svi elementi iterabilnog objekta istiniti, inače vraća `False`.

Primjer korištenja funkcije `any`:

```

print(any([False, False, True])) # True (jer je barem jedan element True)

print(any([False, False, False])) # False (jer niti jedan element nije True)

```

Primjer korištenja funkcije `all`:

```

print(all([True, True, True])) # True (jer su svi elementi True)

print(all([True, False, True])) # False (jer nisu svi elementi True)

```

Kako koristiti ove funkcije s lambda funkcijama?

Recimo da želimo provjeriti jesu li svi brojevi u listi parni. Idemo prvo ručno:

```

def svi_parni(lista):
    for broj in lista:
        if broj % 2 != 0:
            return False
    return True

print(svi_parni([2, 4, 6, 8])) # True
print(svi_parni([2, 4, 6, 7])) # False

```

- ili koristeći `all`, `map` i lambda funkciju:

```
print(all(map(lambda x: x % 2 == 0, [2, 4, 6, 8]))) # True
print(all(map(lambda x: x % 2 == 0, [2, 4, 6, 7]))) # False
```

Kako ovo radi?

1. `map` prima lambda funkciju: `lambda x: x % 2 == 0` i listu `[2, 4, 6, 8]`.
2. `map` primjenjuje tu funkciju na svaki element liste
3. lista sad postaje `[True, True, True, True]`
4. `all` provjerava jesu li svi elementi liste `True` i vraća `True` jer jesu. `all([True, True, True, True])`

Pogledajmo još jedan primjer, gdje želimo provjeriti jesu li svi putnici uplatili aranžman:

```
putnici = [
    {"ime": "Ivan", " prezime": "Ivić", "uplata": True},
    {"ime": "Marko", " prezime": "Marković", "uplata": True},
    {"ime": "Ana", " prezime": "Anić", "uplata": False}
]

print(all(map(lambda putnik: putnik["uplata"], putnici))) # False
```

- ili ručno:

```
def svi_uplatili(putnici):
    for putnik in putnici:
        if not putnik["uplata"]:
            return False
    return True

print(svi_uplatili(putnici)) # False
```

Sličnih funkcija višeg reda ima još mnogo, primjerice `sorted`, `reduce`, `zip` itd. Korisno je istražiti ih i koristiti u praksi jer će vam uvelike ubrzati i olakšati rad.

2. Izgradnja struktura kroz comprehension sintaksu

`Comprehension` Sintaksa je jedan od najmoćnijih alata u Pythonu. Omogućuje nam brzu i jednostavnu izgradnju struktura podataka, kao što su liste, rječnici i skupovi.

Ova sintaksa pruža čitljiv i mnogo kraći način za **izgradnju struktura podataka** u usporedbi s klasičnim načinima korištenja petlji.

Postoje 4 vrste `comprehension` sintakse:

1. **List comprehension** (izgradnja liste)
2. **Dictionary comprehension** (izgradnja rječnika)
3. **Set comprehension** (izgradnja skupa)

4. Generator comprehension (izgradnja generatora)

Nećemo se baviti generatorima, ali ćemo proučiti prve tri vrste.

Najčešće ćemo koristiti **list comprehension**, ali je korisno znati i ostale vrste.

2.1 List comprehension

Krenimo jednostavno: želimo izgraditi **listu kvadrata brojeva od 1 do 10**.

U svim sljedećim primjerima prikazat će se rješenje na **klasičan način** i način **comprehension sintaksom**.

Klasičan način:

```
kvadrati = []

for i in range(1, 11):
    kvadrati.append(i ** 2)

print(kvadrati) # [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Rekli smo da ovo možemo skratiti i korištenjem `map` funkcije višeg reda:

```
kvadrati = list(map(lambda x: x ** 2, range(1, 11)))

print(kvadrati) # [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Ali i korištenjem **list comprehension sintakse**:

```
kvadrati = [x ** 2 for x in range(1, 11)]

print(kvadrati) # [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Idemo usporediti sve tri metode:

1. **Klasičan način:**

- 3 linije koda ukupno:
 - 1 linija za inicijalizaciju liste
 - 1 linija za petlju
 - 1 linija za dodavanje elementa u listu

2. **Korištenjem `map` funkcije:**

- 1 linija koda ukupno:
 - `map` funkcija prima lambda funkciju i range objekt
 - lambda funkcija kvadrira broj, a range objekt vraća listu brojeva od 1 do 10
 - `list` pretvara map objekt u listu

3. Korištenjem list comprehension sintakse:

- 1 linija koda ukupno:
 - poznata sintaksa `for` petlje koja iterira kroz range objekt (listu brojeva od 1 do 10)
 - ispred se dodaje izraz koji se izvršava za svaki element (`x ** 2`)
 - rezultat se dodaje u listu što je definirano uglatim zagradama `[...]`

Osnovna sintaksa list comprehensiona je sljedeća:

```
[expression for element in iterable]
```

Gdje je:

- `expression` izraz koji se izvršava za svaki element
- `element` varijabla koja predstavlja trenutni element
- `iterable` iterabilni objekt (npr. lista, skup, rječnik, generator), u ovom slučaju je lista brojeva od 1 do 10

Recimo da imamo listu znakovnih nizova gdje želimo **izgraditi listu duljina tih nizova**:

Klasičan način:

```
nizovi = ["jabuka", "kruška", "banana", "naranča"]

duljine = []

for niz in nizovi:
    duljine.append(len(niz))

print(duljine) # [6, 6, 6, 7]
```

List comprehension:

```
duljine = [len(niz) for niz in nizovi]

print(duljine) # [6, 6, 6, 7]
```

Ovdje je `len(niz)` izraz koji se izvršava za svaki element `niz` u listi `nizovi`.

Idemo dalje, možemo nadograditi sintaksu list comprehensiona dodavanjem `if` uvjeta.

Kako izgraditi listu kvadrata brojeva od 1 do 10, ali **samo za neparne brojeve**:

Klasičan način:

```

kvadrati_neparnih = []

for i in range(1, 11):
    if i % 2 != 0:
        kvadrati_neparnih.append(i ** 2)

print(kvadrati_neparnih) # [1, 9, 25, 49, 81]

```

List comprehension:

```
kvadrati_neparnih = [x ** 2 for x in range(1, 11) if x % 2 != 0] # uvjet se dodaje na kraj
```

Pomalo je neuobičajeno, ali ove izraze želimo čitati slično kao što bismo ih čitali običnim jezikom:

- "kvadrat broja x za svaki x u rasponu od 1 do 10 ako je x neparni broj"

Međutim, prilikom programiranja često ćemo pisati ove izraze (1) počevši od petlje, (2) zatim izraza i (3) uvjeta na kraju, slično kao što bismo kodirali na klasičan način.

Sintaksa s `if` uvjetom:

```
[expression for element in iterable if condition]
```

Primjer iznad praktično je "graditi" na sljedeći način:

1. Prvo definiramo `for` petlju koja prolazi kroz sve brojeve od 1 do 10, a izraz neka bude samo taj broj x

```
[x for x in range(1, 11)] # [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

2. Zatim želimo kao izraz kvadrat brojeva, pa mijenjamo u $x ** 2$

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

3. Na kraju dodajemo uvjet `if x % 2 != 0` i to nakon `for` petlje

```
[x ** 2 for x in range(1, 11) if x % 2 != 0] # [1, 9, 25, 49, 81]
```

Kako ovo koristiti sa strukturama? Imamo listu rječnika gdje želimo izgraditi **listu imena studenata** koji su rođeni prije 1999. godine:

```

studenti = [
    {"ime": "Ivan", "prezime": "Ivić", "godina_rodjenja": 2000},
    {"ime": "Marko", "prezime": "Marković", "godina_rodjenja": 1990},
    {"ime": "Ana", "prezime": "Anić", "godina_rodjenja": 2003},
    {"ime": "Petra", "prezime": "Petrić", "godina_rodjenja": 2001}
]

```

Klasičan način:

```
rodeni_prije_1999 = []

for student in studenti:
    if student["godina_rodenja"] < 1999:
        rodeni_prije_1999.append(student["ime"])

print(rodeni_prije_1999) # ['Marko']
```

List comprehension:

```
rodeni_prije_1999 = [student["ime"] for student in studenti if student["godina_rodenja"] < 1999]

print(rodeni_prije_1999) # ['Marko']
```

Moguće je dodati i **else izraz**:

Primjer: Želimo izgraditi listu kvadrata brojeva od 1 do 10, ali **za neparne brojeve kvadrat, a za parne brojeve sam broj**:

Klasičan način:

```
kvadrati_neparnih_a_parne_brojevi= []

for i in range(1, 11):
    if i % 2 != 0:
        kvadrati_neparnih_a_parne_brojevi.append(i ** 2)
    else:
        kvadrati_neparnih_a_parne_brojevi.append(i)

print(kvadrati_neparnih_a_parne_brojevi) # [1, 2, 9, 4, 25, 6, 49, 8, 81, 10]
```

List comprehension:

```
kvadrati_neparnih_a_parne_brojevi = [x ** 2 for x in range(1, 11) if x % 2 != 0 else x]

print(kvadrati_neparnih_a_parne_brojevi) # SyntaxError: invalid syntax (zašto ???)
```

Sintaksa s **else izrazom** je nešto drugačija nego kad koristimo samo **if** uvjet:

```
[expression1 if condition else expression2 for element in iterable]
```

- dok smo kod **if** uvjeta imali:

```
[expression for element in iterable if condition]
```

- dakle, moramo prvo definirati `if` izraz, a zatim `else` izraz i to sve ispred `for` petlje:

```
kvadrati_neparnih_a_parne_brojevi = [x ** 2 if x % 2 != 0 else x for x in range(1, 11)] #
[1, 2, 9, 4, 25, 6, 49, 8, 81, 10]
```

Comprehension možemo koristiti i s znakovnim nizovima.

Primjer: Imamo listu voća `fruits`:

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
```

Želimo izgraditi listu voća, ali **samo prva tri slova svakog voća**:

Klasičan način:

```
prva_tri_slova = []

for fruit in fruits:
    prva_tri_slova.append(fruit[:3])

print(prva_tri_slova) # ['app', 'ban', 'che', 'kiw', 'man']
```

List comprehension:

```
prva_tri_slova = [fruit[:3] for fruit in fruits]
```

- Ili želimo izgraditi novu listu voća, npr. koja sadrži samo ono voće koje sadrži slovo `a`:

Klasičan način:

```
sa_slovom_a = []

for fruit in fruits:
    if "a" in fruit:
        sa_slovom_a.append(fruit)

print(sa_slovom_a) # ['apple', 'banana', 'mango']
```

List comprehension:

```
sa_slovom_a = [fruit for fruit in fruits if "a" in fruit]
```

Koji će biti sadržaj sljedeće liste?

```
newlist = [x if x != "banana" else "orange" for x in fruits]

print(newlist) # ?
```

► Spoiler alert! Odgovor na pitanje

2.2 Dictionary comprehension

Dictionary comprehension je vrlo sličan list comprehensionu, ali umjesto liste, gradimo rječnik kroz comprehension sintaksu.

Sintaksa dictionary comprehensiona je sljedeća:

```
{key_expression: value_expression for item in iterable if condition}
```

Uočite :

Gdje je:

- `key_expression` izraz koji se izvršava za ključeve
- `value_expression` izraz koji se izvršava za vrijednosti
- `item` varijabla koja predstavlja trenutni element
- `iterable` iterabilni objekt (npr. lista, skup, rječnik, generator)
- `condition` uvjet koji se može dodati (nije obavezan)

Recimo da imamo listu voća `fruits` i želimo izgraditi rječnik gdje su **ključevi voća**, a **vrijednosti duljina tih voća**:

Klasičan način:

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]

duljine_voca = {}

for fruit in fruits:
    duljine_voca[fruit] = len(fruit)

print(duljine_voca) # {'apple': 5, 'banana': 6, 'cherry': 6, 'kiwi': 4, 'mango': 5}
```

Dictionary comprehension:

```
duljine_voca = {fruit: len(fruit) for fruit in fruits}

print(duljine_voca) # {'apple': 5, 'banana': 6, 'cherry': 6, 'kiwi': 4, 'mango': 5}
```

Možemo napraviti i rječnik gdje su ključevi i vrijednosti brojevi, a petlja ide od 1 do 5:

Ključevi neka budu brojevi od 1 do 5, a vrijednosti kvadrati tih brojeva:

Klasičan način:

```
kvadrati_brojeva = {}

for i in range(1, 6):
    kvadrati_brojeva[i] = i ** 2

print(kvadrati_brojeva) # {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

Dictionary comprehension:

```
kvadrati_brojeva = {i: i ** 2 for i in range(1, 6)}

print(kvadrati_brojeva) # {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

Moguće je unutar comprehension sintakse koristiti vanjske funkcije:

```
def kvadriraj(x):
    return x ** 2

kvadrati_brojeva = {i: kvadriraj(i) for i in range(1, 6)}
```

Ako želimo dodati uvjete, to radimo na kraju na isti način kao i kod list comprehensiona:

Primjer: Želimo izgraditi rječnik gdje su ključevi brojevi, a vrijednosti kvadrati tih brojeva, ali **samo za parne brojeve** od 1 do 10:

Klasičan način:

```
kvadrati_parnih = {}

for i in range(1, 11):
    if i % 2 == 0:
        kvadrati_parnih[i] = i ** 2

print(kvadrati_parnih) # {2: 4, 4: 16, 6: 36, 8: 64, 10: 100}
```

Dictionary comprehension:

```
kvadrati_parnih = {i: i ** 2 for i in range(1, 11) if i % 2 == 0}

print(kvadrati_parnih) # {2: 4, 4: 16, 6: 36, 8: 64, 10: 100}
```

Ako dodamo i else izraz, sintaksa je slična kao kod list comprehensiona:

Primjer: Izradit ćemo rječnik gdje ćemo za svaki broj kao ključ postaviti taj broj, a vrijednost će biti "paran" ako je broj paran, inače "neparan":

Klasičan način:

```
paran_neparan = {}

for i in range(1, 11):
    if i % 2 == 0:
        paran_neparan[i] = "paran"
    else:
        paran_neparan[i] = "neparan"

print(paran_neparan) # {1: 'neparan', 2: 'paran', 3: 'neparan', 4: 'paran', 5: 'neparan',
6: 'paran', 7: 'neparan', 8: 'paran', 9: 'neparan', 10: 'paran'}
```

Dictionary comprehension:

```
paran_neparan = {i: "paran" if i % 2 == 0 else "neparan" for i in range(1, 11)}

print(paran_neparan) # {1: 'neparan', 2: 'paran', 3: 'neparan', 4: 'paran', 5: 'neparan',
6: 'paran', 7: 'neparan', 8: 'paran', 9: 'neparan', 10: 'paran'}
```

Sintaksa **set comprehensiona** je vrlo slična list comprehensionu, ali umjesto liste, gradimo skup koristeći `{}` zagrade, bez `key:value` parova.

3. Zadaci za vježbu - lambda izrazi, funkcije višeg reda i comprehension sintaksa

Zadatak 1: Lambda izrazi

Napišite korespondirajuće **lambda** izraze za sljedeće funkcije:

1. Kvadriranje broja:

```
def kvadriraj(x):
    return x ** 2
```

2. Zbroji pa kvadriraj:

```
def zbroji_pa_kvadriraj(a, b):
    return (a + b) ** 2
```

3. Kvadriraj duljinu niza:

```
def kvadriraj_duljinu(niz):
    return len(niz) ** 2
```

4. Pomnoži vrijednost s 5 pa potenciraj na x:

```
def pomnozi_i_potenciraj(x, y):  
    return (y * 5) ** x
```

5. Vrati True ako je broj paran, inače vrati None:

```
def paran_broj(x):  
    if x % 2 == 0:  
        return True  
    else:  
        return None
```

Zadatak 2: Funkcije višeg reda

Definirajte sljedeće izraze korištenjem funkcija višeg reda i lambda izraza:

1. Koristeći funkciju `map`, kvadrirajte duljine svih nizova u listi:

```
nizovi = ["jabuka", "kruška", "banana", "naranča"]  
  
kvadrirane_duljine = ...  
  
print(kvadrirane_duljine) # [36, 36, 36, 49]
```

2. Koristeći funkciju `filter`, filtrirajte samo brojeve koji su veći od 5:

```
brojevi = [1, 21, 33, 45, 2, 2, 1, -32, 9, 10]  
  
veci_od_5 = ...  
  
print(veci_od_5) # [21, 33, 45, 9, 10]
```

3. Koristeći odgovarajuću funkciju višeg reda i lambda izraz (bez comprehensiona), pohranite u varijablu `transform` rezultat kvadriranja svih brojeva u listi gdje rezultat mora biti rječnik gdje su ključevi originalni brojevi, a vrijednosti kvadrati tih brojeva:

```
brojevi = [10, 5, 12, 15, 20]  
  
transform = ...  
  
print(transform) # {10: 100, 5: 25, 12: 144, 15: 225, 20: 400}
```

4. Koristeći funkcije `all` i `map`, provjerite jesu li svi studenti punoljetni:

```

studenti = [
    {"ime": "Ivan", "prezime": "Ivić", "godine": 19},
    {"ime": "Marko", "prezime": "Marković", "godine": 22},
    {"ime": "Ana", "prezime": "Anić", "godine": 21},
    {"ime": "Petra", "prezime": "Petrić", "godine": 13},
    {"ime": "Iva", "prezime": "Ivić", "godine": 17},
    {"ime": "Mate", "prezime": "Matić", "godine": 18}
]

svi_punoljetni = ...

print(svi_punoljetni) # False

```

5. Definirajte varijablu `min_duljina` koja će pohranjivati `int`. Koristeći odgovarajuću funkciju višeg reda i lambda izraz, pohranite u varijablu `duge_rijeci` sve riječi iz liste `rijeci` koje su dulje od `min_duljina`:

```

rijeci = ["jabuka", "pas", "knjiga", "zvijezda", "priatelj", "zvuk", "čokolada", "ples",
          "pjesma", "otorinolaringolog"]

min_duljina = input("Unesite minimalnu duljinu riječi: ")

# min_duljina = 7
duge_rijeci = ...

# print(duge_rijeci) # ['zvijezda', 'priatelj', 'čokolada', 'otorinolaringolog']

```

Zadatak 3: Comprehension sintaksa

1. Koristeći list comprehension, izgradite listu parnih kvadrata brojeva od 20 do 50:

```

parni_kvadrati = ...

print(parni_kvadrati) # [400, 484, 576, 676, 784, 900, 1024, 1156, 1296, 1444, 1600, 1764,
                      1936, 2116, 2304, 2500]

```

2. Koristeći list comprehension, izgradite listu duljina svih nizova u listi `rijeci`, ali samo za nizove koji sadrže slovo `a`:

```

rijeci = ["jabuka", "pas", "knjiga", "zvijezda", "priatelj", "zvuk", "čokolada", "ples",
          "pjesma", "otorinolaringolog"]

duljine_sa_slovom_a = ...

print(duljine_sa_slovom_a) # [6, 3, 6, 8, 9, 8, 6, 17]

```

3. Koristeći list comprehension, izgradite listu rječnika gdje su ključevi brojevi od 1 do 10, a vrijednosti kubovi tih brojeva, ali samo za neparne brojeve, za parne brojeve neka vrijednost bude sam broj:

```
kubovi = ...
```

```
print(kubovi) # [{1: 1}, {2: 2}, {3: 27}, {4: 4}, {5: 125}, {6: 6}, {7: 343}, {8: 8}, {9: 729}, {10: 10}]
```

4. Koristeći dictionary comprehension, izgradite rječnik iteriranjem kroz listu brojeva od 50 do 500 s korakom 50, gdje su ključevi brojevi, a vrijednosti su korijeni tih brojeva zaokruženi na 2 decimale:

```
korijeni = ...
```

```
print(korijeni) # {50: 7.07, 100: 10.0, 150: 12.25, 200: 14.14, 250: 15.81, 300: 17.32, 350: 18.71, 400: 20.0, 450: 21.21, 500: 22.36}
```

5. Koristeći list comprehension, izgradite listu rječnika gdje su ključevi prezimena studenata, a vrijednosti su zbrojeni bodovi, iz liste `studenti`:

```
studenti = [
    {"ime": "Ivan", "prezime": "Ivić", "bodovi": [12, 23, 53, 64]}, 
    {"ime": "Marko", "prezime": "Marković", "bodovi": [33, 15, 34, 45]}, 
    {"ime": "Ana", "prezime": "Anić", "bodovi": [8, 9, 4, 23, 11]}, 
    {"ime": "Petra", "prezime": "Petrić", "bodovi": [87, 56, 77, 44, 98]}, 
    {"ime": "Iva", "prezime": "Ivić", "bodovi": [23, 45, 67, 89, 12]}, 
    {"ime": "Mate", "prezime": "Matić", "bodovi": [75, 34, 56, 78, 23]}
]
```

```
zbrojeni_bodovi = ...
```

```
print(zbrojeni_bodovi) # [{'Ivić': 152}, {'Marković': 127}, {'Anić': 55}, {'Petrić': 362}, {'Ivić': 236}, {'Matić': 266}]
```

4. Klase i objekti

Klase (*eng. Class*) i **objekti** (*eng. Object*) su temeljna paradigma u objektno orijentiranom programiranju.

- Klase su šablonski opisi objekata, dok su objekti instance klasa. Izradom nove klase, automatski se stvara novi **tip podataka**.

Slično kao i u JavaScriptu, u Pythonu je gotovo su gotovo svi programske konstrukti objekti koji sadrže **atribute** (*eng. attribute*) i **metode** (*eng. method*).

Dakle klase možemo zamisliti kao šablone (*eng. blueprint*) za definiranje atributa i metoda objekata.

Klasu definiramo ključnom riječju `class`, a objekt klase stvaramo **pozivom klase kao funkcije**. Ne koristimo `new` ključnu riječ kao u nekim drugim jezicima.

4.1 Definiranje klase i stvaranje objekta

Primjer jednostavne klase:

```
class Osoba:  
    pass
```

I to je to! Definirali smo klasu `osoba` koja ne sadrži niti jedan atribut ili metodu. Često koristimo `pass` kada želimo definirati praznu klasu koju ćemo kasnije nadograditi.

Objekt stvaramo pozivom klase kao funkcije:

```
osoba = Osoba()  
  
print(osoba) # <__main__.Osoba object> memorijска lokacija objekta
```

Atribute možemo definirati prilikom definicije, navodeći ih kao varijable unutar klase:

```
class Osoba:  
    ime = "Ivan"  
    prezime = "Ivić"  
    godine = 25  
  
osoba = Osoba()  
  
print(osoba.ime) # Ivan  
print(osoba.prezime) # Ivić  
print(osoba.godine) # 25
```

4.2 Konstruktor klase

Primjer iznad nije dobar način definiranja klase jer svi objekti klase `osoba` dijele iste atribute.

Konstruktor (*eng. Constructor*) je posebna metoda koja se koristi za **inicijalizaciju objekta klase**.

Iz tog razloga možemo definirati **konstruktor klase** koji se definira metodom `__init__`. Ova metoda poziva se svaki put prilikom inicijalizacije objekta klase.

Primjer: Nadogradnja klase `osoba` s konstruktorom:

```
class Osoba:  
    def __init__(self, ime, prezime, godine):  
        self.ime = ime  
        self.prezime = prezime  
        self.godine = godine  
  
osoba = Osoba("Ivan", "Ivić", 25)  
  
print(osoba.ime) # Ivan  
print(osoba.prezime) # Ivić
```

```

print(osoba.godine) # 25

osoba2 = Osoba("Marko", "Marković", 30)

print(osoba2.ime) # Marko
print(osoba2.prezime) # Marković
print(osoba2.godine) # 30

```

Primijetite da smo koristili `self` kao prvi argument metode `__init__`. `self` je ključna riječ i **referenca na trenutni objekt klase** i koristi se za pristupanje atributima i metodama objekta. Bez navođenja `self` kao prvog argumenta, Python će baciti grešku.

```

class Osoba:
    def __init__(ime, prezime, godine): # TypeError: __init__() takes 3 positional
arguments but 4 were given
        self.ime = ime
        self.prezime = prezime
        self.godine = godine

osoba = Osoba("Maja", "Majić", 30)

```

4.3 Metode klase

Metode klase su funkcije koje se definiraju unutar klase i koriste se za izvršavanje određenih operacija **nad objektima klase**.

Kada definiramo metode, možemo pristupati vrijednostima atributa objekta pomoću `self` reference.

Primjer metode `pozdrav`:

```

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

    def pozdrav(self):
        return f"Pozdrav, ja sam {self.ime} {self.prezime} i imam {self.godine} godina."

```

Poziv metode:

```

osoba = Osoba("Snješka", "Snježanić", 25)

print(osoba.pozdrav()) # Pozdrav, ja sam Snješka Snježanić i imam 25 godina.

print(pozdrav(osoba)) # oprez, česta greška! Metode pozivamo nad objektima, ovo je primjer
# poziva globalne funkcije koja prima objekt kao argument

```

Metode mogu biti bilo što, od jednostavnih operacija do složenih lambda izraza ili izraza koji pozivaju vanjske funkcije ili unutarnje metode.

4.4 Nasljeđivanje

Nasljeđivanje (*eng. Inheritance*) je ključna paradigma u objektno orijentiranom programiranju. Omogućuje nam **definiranje novih klasa koje nasljeđuju atribute i metode od postojećih klasa**.

Klasa koja nasljeđuje zove se **podkласа** (*eng. subclass*), a klasa koja se nasljeđuje zove se **nadkласа** (*eng. superclass*).

Prilikom definiranja podklase, navodimo nadklasu u zagradama, a koristeći `super()` funkciju možemo nasljediti sve atribute i metode nadklase.

Primjer nasljeđivanja:

```
class Korisnik:
    def __init__(self, ime, prezime):
        self.ime = ime
        self.prezime = prezime
        self.username = f"{ime.lower()}_{prezime.lower()}""

    def pozdrav(self):
        return f"Pozdrav, ja sam {self.ime} {self.prezime}, moj username je
{self.username}."

class Admin(Korisnik):
    def __init__(self, ime, prezime, privilegije):
        super().__init__(ime, prezime) # nasljeđujemo atribute od nadklase
        self.privilegije = privilegije

    def pozdrav(self):
        return f"Pozdrav, ja sam {self.ime} {self.prezime}, moj username je
{self.username} i imam ukupno {len(self.privilegije)} privilegije: {',
'.join(self.privilegije)}."
```

Instaciramo objekt klase `Admin`:

```
root = ["dodavanje_korisnika", "brisanje_korisnika", "dodavanje_postova",
"brisanje_postova"]
admin = Admin("Ivan", "Ivić", root)

print(admin.pozdrav()) #Pozdrav, ja sam Ivan Ivić, moj username je ivan_ivić i imam ukupno
4 privilegije: dodavanje_korisnika, brisanje_korisnika, dodavanje_postova,
brisanje_postova.
```

Objekte i njihova svojstva možemo brisati pomoću `del` ključne riječi:

```
del admin.privilegije

del admin
```

5. Moduli i paketi

5.1 Moduli

Moduli (eng. *Module*) su Python datoteke koje sadrže definicije funkcija, klasa ili varijabli **koje možemo koristiti u drugim Python datotekama**. Moduli nam omogućuju bolju organizaciju koda i ponovnu upotrebu koda koji se ponavlja i potrebno ga je koristiti u više datoteka.

Module možemo učitati u Python skriptu koristeći ključnu riječ `import`, a definiramo ih u vanjskim datotekama s ekstenzijom `.py`.

Primjer modula:

```
# greetings.py

def pozdrav(ime):
    return f"Pozdrav, {ime}!"
```

- Učitavanje modula:

```
# main.py
import greetings

print(greetings.pozdrav("Marko")) # Pozdrav, Marko!
```

Dakle, u "modulima" možemo definirati i varijable/klase, a zatim ih čitati/pozivati u glavnoj skripti na jednak način.

```
# greetings.py

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

    def pozdrav(self):
        return f"Pozdrav, {self.ime}!"

studenti = ["Ana", "Bojan", "Milka", "Dejan", "Ema"]
# main.py

import greetings

student = greetings.studenti # ['Ana', 'Bojan', 'Milka', 'Dejan', 'Ema']

student_objekt = greetings.Student("Ema")

print(student_objekt.pozdrav()) # Pozdrav, Ema!
```

Modulima možemo davati proizvoljna imena, ali moraju imati ekstenziju `.py`.

Moguće je prilikom učitavanja modula koristiti i `as` ključnu riječ za davanje **aliasa** modulu. Ovo je korisno kada imamo modul s dugim imenom ili kada želimo izbjegći konflikte imena.

```
import greetings as g

print(g.pozdrav("Ivan")) # Pozdrav, Ivan!
```

Ako želimo učitati samo određene funkcije iz modula, koristimo `from` ključnu riječ:

- Tada ne moramo navoditi naziv modula prilikom poziva funkcije:

```
from greetings import pozdrav

# pozivamo funkciju bez navođenja imena modula "greetings"
print(pozdrav("Iva")) # Pozdrav, Iva!
```

Možemo definirati više funkcija, a zatim učitati samo one koje želimo:

Definirajmo modul `math_operations.py`:

```
# math_operations.py

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

def oduzimanje(a, b):
    return a - b

def mnozenje(a, b):
    return a * b

def dijeljenje(a, b):
    return a / b

def potenciranje(a, b):
    return a ** b

def korijen(a):
    return a ** 0.5

def kvadrat(a):
    return a ** 2
```

Učitavanje samo funkcija `zbroj` i `oduzimanje`:

```
from math_operations import zbroj, oduzimanje

print(zbroj(5, 3)) # 8
print(oduzimanje(5, 3)) # 2
```

- ili učitavanje svih funkcija iz modula sa zvjezdicom `*`:

```
from math_operations import *

print(zbroj(5, 3)) # 8
print(oduzimanje(5, 3)) # 2
print(mnozenje(5, 3)) # 15
print(dijeljenje(5, 3)) # 1.6666666666666667
print(potenciranje(5, 3)) # 125
print(korijen(25)) # 5.0
print(kvadrat(5)) # 25
```

Moguće je i učitati sve funkcije iz modula i dodati im alias:

```
from math_operations import zbroj as add, oduzimanje as sub

print(add(5, 3)) # 8
print(sub(5, 3)) # 2
```

5.1.1 Ugrađeni moduli

Ugrađenih modula u Pythonu ima mnogo, a neki od najčešće korištenih su:

- `math` - matematičke operacije
- `random` - generiranje slučajnih brojeva
- `datetime` - omogućuje rad s datumima i s vremenom
- `os` - omogućuje interakciju s operacijskim sustavom, npr. manipulaciju datotečnog sustava, environment varijablama, itd.
- `sys` - omogućuje pristup parametrima i funkcijama specifičnim za sustav, kao što su argumenti naredbenog retka, standardni ulaz i izlaz, itd.
- `json` - omogućuje rad s JSON formatom i Python objektima (serijalizacija i deserijalizacija)
- `re` - omogućuje rad s regularnim izrazima (regex)
- `collections` - dodatne kolekcije podataka koje nisu ugrađene u Python, kao što su `namedtuple`, `deque`, `Counter`, `OrderedDict`, itd.
- `itertools` - dodatne funkcije za rad s iterabilnim objektima, kao što su `chain`, `cycle`, `repeat`, `combinations`, `permutations`, itd.

Primjer korištenja nekih ugrađenih modula:

```
import math
```

```

print(math.pi) # 3.141592653589793
print(math.sqrt(25)) # 5.0
import random

print(random.randint(1, 10)) # slučajni broj između 1 i 10
import datetime

print(datetime.datetime.now()) # 2024-11-09 19:36:41.954767
print(datetime.datetime.now().year) # 2024
import os

print(os.getcwd()) # /Users/lukablaskovic/Github/FIPU-RS
print(os.listdir()) # ['RS1 - Ponavljanje Pythona', '.DS_Store', 'RS2 - Napredniji Python koncepti', 'README.md', 'RS3 - Asyncio i Aiohttp', '.git']

os.mkdir("nova_mapa") # stvara novu mapu "nova_mapa"

```

Dokumentaciju ugrađenih modula za Python 3 možete pronaći [ovdje](#).

Na internetu možete pronaći puno dokumentacije i primjera korištenja poznatih modula, a mi ćemo se fokusirati samo na neke od njih u nastavku ovog kolegija.

5.2 Paketi

Paketi (*eng. Packages*) su direktoriji koji sadrže **više modula**. Paketi su nam korisni kada želimo organizirati naš kod u logičke cjeline, gdje više različitih modula radi zajedno.

Zamislite pakete kao foldere koji sadrže više Python datoteka.

Primjer strukture paketa `faculty` koji sadrži module `studenti.py` i `operacije.py`:

```

faculty/
|
├── __init__.py
├── studenti.py
└── operacije.py

```

Uočite da za definiranje paketa moramo imati datoteku `__init__.py` u direktoriju paketa.

U `__init__.py` datoteci možemo definirati varijable, funkcije ili klase koje će biti dostupne prilikom učitavanja paketa, ali to **nije obavezno i ona može biti prazna**.

Idemo u modul `studenti.py` definirati klasu `Student` s atributima `ime`, `prezime` i `kolegiji` i metodama `pozdrav` i `kolegiji`:

```
# studenti.py

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

    def pozdrav(self):
        return f"Pozdrav, ja sam {self.ime} {self.prezime}."

    def ispis_kolegija(self):
        return f"Moji kolegiji su: {', '.join(self.kolegiji)}."
```

Koristimo sintaksu `from` i `import` za učitavanje modula iz paketa:

```
# main.py
from faculty import studenti

student_marko = studenti.Student("Marko", "Marković", ["Web aplikacije", "Raspodijeljeni sustavi", "Operacijska istraživanja"])

print(student_marko.pozdrav()) # Pozdrav, ja sam Marko Marković.

print(student_marko.ispis_kolegija()) # Moji kolegiji su: Web aplikacije, Raspodijeljeni sustavi, Operacijska istraživanja.
```

Unutar modula `operacije.py` možemo recimo definirati neku funkciju koja će za svaki kolegij stvoriti rječnik gdje su ključevi kolegiji, a vrijednosti liste ocjena.

```
# operacije.py

def ocjene(kolegiji):
    return {kolegij: [] for kolegij in kolegiji}
```

Učitavanje modula `operacije.py`:

```
# main.py
from faculty import operacije

ocjene_studenta = operacije.ocjene(student_marko.kolegiji)

print(ocjene_studenta) # {'Web aplikacije': [], 'Raspodijeljeni sustavi': [], 'Operacijska istraživanja': []}
```

- itd. Možemo dodati funkciju koja simulira dodavanje 5 random ocjena studentu za od kolegija:

```
# operacije.py

import random

def simuliraj_ocjene(kolegiji):
    return {kolegij: [random.randint(1, 5) for _ in range(5)] for kolegij in kolegiji}
```

Učitavanje i korištenje funkcije:

```
# main.py
from faculty import studenti, operacije

student_marko = studenti.Student("Marko", "Marković", ["Web aplikacije", "Raspodijeljeni sustavi", "Operacijska istraživanja"])

ocjene_studenta = operacije.ocjene(student_marko.kolegiji)

print(ocjene_studenta) # {'Web aplikacije': [], 'Raspodijeljeni sustavi': [], 'Operacijska istraživanja': []}

simulacija_ocjena = operacije.simuliraj_ocjene(student_marko.kolegiji) # {'Web aplikacije': [2, 3, 1, 4, 4], 'Raspodijeljeni sustavi': [3, 1, 3, 4, 1], 'Operacijska istraživanja': [5, 2, 1, 1, 5]}

print(simulacija_ocjena)
```

Za ispis svih ocjena pojedinog studenta, možemo dodati novu metodu u klasu `Student` unutar modula `studenti.py`:

```
# studenti.py

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

    def pozdrav(self):
        return f"Pozdrav, ja sam {self.ime} {self.prezime}."

    def ispisi_kolegija(self):
        return f"Moji kolegiji su: {', '.join(self.kolegiji)}."

    def ispisi_ocjena(self, ocjene):
        for kolegij, ocjene in ocjene.items():
            print(f"Ocjene iz kolegija {kolegij}: {', '.join(map(str, ocjene))}.")
```

Ispis ocjena studenta:

```

student_marko.ispis_ocjena(simulacija_ocjena)

"""
Ocjene iz kolegija Web aplikacije: 5, 1, 1, 3, 4.
Ocjene iz kolegija Raspodijeljeni sustavi: 4, 4, 2, 5, 2.
Ocjene iz kolegija Operacijska istraživanja: 3, 3, 5, 4, 3.
"""

```

Naravno, postoji i mnoštvo ugrađenih paketa u Pythonu, a mi ćemo se fokusirati samo na neke od njih u nastavku ovog kolegija.

6. Zadaci za vježbu - Klase, objekti, moduli i paketi

Zadatak 4: Klase i objekti

- Definirajte klasu `Automobil` s atributima `marka`, `model`, `godina_proizvodnje` i `kilometraža`. Dodajte metodu `ispis` koja će ispisivati sve attribute automobila.
 - Stvorite objekt klase `Automobil` s proizvoljnim vrijednostima atributa i pozovite metodu `ispis`.
 - Dodajte novu metodu `starost` koja će ispisivati koliko je automobil star u godinama, trenutnu godine dohvate pomoći `datetime` modula.
- Definirajte klasu `Kalkulator` s atributima `a` i `b`. Dodajte metode `zbroj`, `oduzimanje`, `mnozenje`, `dijeljenje`, `potenciranje` i `korijen` koje će izvršavati odgovarajuće operacije nad atributima `a` i `b`.
- Definirajte klasu `Student` s atributima `ime`, `prezime`, `godine` i `ocjene`.

Iterirajte kroz sljedeću listu studenata i za svakog studenta stvorite objekt klase `Student` i dodajte ga u novu listu `studenti_objekti`:

```

studenti = [
    {"ime": "Ivan", "prezime": "Ivić", "godine": 19, "ocjene": [5, 4, 3, 5, 2]},
    {"ime": "Marko", "prezime": "Marković", "godine": 22, "ocjene": [3, 4, 5, 2, 3]},
    {"ime": "Ana", "prezime": "Anić", "godine": 21, "ocjene": [5, 5, 5, 5, 5]},
    {"ime": "Petra", "prezime": "Petrić", "godine": 13, "ocjene": [2, 3, 2, 4, 3]},
    {"ime": "Iva", "prezime": "Ivić", "godine": 17, "ocjene": [4, 4, 4, 3, 5]},
    {"ime": "Mate", "prezime": "Matić", "godine": 18, "ocjene": [5, 5, 5, 5, 5]}
]

```

- Dodajte metodu `prosjek` koja će računati prosječnu ocjenu studenta.
 - U varijablu `najbolji_student` pohranite studenta s najvećim prosjekom ocjena iz liste `studenti_objekti`. Implementirajte u jednoj liniji koda.
- Definirajte klasu `Krug` s atributom `r`. Dodajte metode `opseg` i `povrsina` koje će računati opseg i površinu kruga.
 - Stvorite objekt klase `Krug` s proizvoljnim radiusom i ispišite opseg i površinu kruga.

5. Definirajte klasu `Radnik` s atributima `ime`, `pozicija`, `placa`. Dodajte metodu `work` koja će ispisivati "Radim na poziciji {pozicija}".

- Dodajte klasu `Manager` koja nasljeđuje klasu `Radnik` i definirajte joj atribut `department`. Dodajte metodu `work` koja će ispisivati "Radim na poziciji {pozicija} u odjelu {department}".
- U klasu `Manager` dodajte metodu `give_raise` koja prima parametre `radnik` i `povecanje` i povećava plaću radnika (`Radnik`) za iznos `povecanje`.
- Definirajte jednu instancu klase `Radnik` i jednu instancu klase `Manager` i pozovite metode `work` i `give_raise`.

Zadatak 5: Moduli i paketi

Definirajte paket `shop` koji će sadržavati module `proizvodi.py` i `narudzbe.py`.

Modul `proizvodi.py`:

- definirajte klasu `Proizvod` s atributima `naziv`, `cijena` i `dostupna_kolicina`. Dodajte metodu `ispis` koja će ispisivati sve atribute proizvoda.
- u listu `skladiste` pohranite 2 objekta klase `Proizvod` s proizvoljnim vrijednostima atributa. U ovoj listi ćete pohranjivati instance klase `Proizvod` koje će predstavljati stanje proizvoda u skladištu.
- definirajte funkciju `dodaj_proizvod` van definicije klase koja će dodavati novi `Proizvod` u listu `skladiste`.

U `main.py` datoteci učitajte modul `proizvodi.py` iz paketa `shop` i pozovite pozovite funkciju `dodaj_proizvod` za svaki element iz sljedeće liste:

```
proizvodi_za_dodavanje = [
    {"naziv": "Laptop", "cijena": 5000, "dostupna_kolicina": 10},
    {"naziv": "Monitor", "cijena": 1000, "dostupna_kolicina": 20},
    {"naziv": "Tipkovnica", "cijena": 200, "dostupna_kolicina": 50},
    {"naziv": "Miš", "cijena": 100, "dostupna_kolicina": 100}
]
```

Lista `skladiste` treba sada sadržavati ukupno 6 elemenata.

Nakon što to napravite, pozovite metodu `ispis` za svaki proizvod iz liste `skladiste`.

Modul `narudzbe.py`:

- definirajte klasu `Narudzba` s atributima: `naruceni_proizvodi` i `ukupna_cijena`.
- dodajte funkciju `napravi_narudzbu` van definicije klase koja prima listu proizvoda kao argument i vraća novu instancu klase `Narudzba`.
- dodajte provjeru u funkciju `napravi_narudzbu` koja će provjeravati dostupnost proizvoda prije nego što se napravi narudžba. Ako proizvoda nema na stanju, ispišite poruku: "Proizvod {naziv} nije dostupan!" i ne stvarajte narudžbu.
- dodajte provjere u funkciju `napravi_narudzbu` koja će provjeriti sljedeća 4 uvjeta:
 - argument `naruceni_proizvodi` mora biti lista
 - svaki element u listi mora biti rječnik

- svaki rječnik mora sadržavati ključeve `naziv`, `cijena` i `narucena_kolicina`
- lista ne smije biti prazna
- izračunajte ukupnu cijenu narudžbe koju ćete pohraniti u lokalnu varijablu `ukupna_cijena` u jednoj liniji koda.
- narudžbe (instanca klase `Narudzba`) pohranite u listu rječnika `narudzbe`.
- u klasu `Narudzba` dodajte metodu `ispis_narudzbe` koja će ispisivati nazive svih naručenih proizvoda, količine te ukupnu cijenu narudžbe.
 - npr. "Naručeni proizvodi: Laptop x 2, Monitor x 1, Ukupna cijena: 11000 eur".

U `main.py` datoteci učitajte modul `narudzbe.py` iz paketa `shop` i pozovite funkciju `napravi_narudzbu` s listom proizvoda iz prethodnog zadatka.

Raspodijeljeni sustavi (RS)

Nositelj: doc. dr. sc. Nikola Tanković

Asistent: Luka Blašković, mag. inf.

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



Fakultet informatike u Puli

(3) Asinkroni Python: Osnove *asyncio* biblioteke

#3

RS

Asinkronost je koncept koji označava mogućnost simultanog izvršavanja više zadataka pri čemu se zadaci izvršavaju neovisno jedan o drugome, odnosno ne čekaju jedan na drugi da se završe, već se odvijaju neovisno o međusobnim vremenskim ograničenjima. U Pythonu, asinkrono programiranje omogućuje nam da zadatke izvršavamo konkurentno, bez blokiranja izvršavanja programa i to bez korištenja tradicionalnih multi-threading tehnika kroz programske dretve. Navedeno je korisno za zadatke poput I/O operacija, mrežnih operacija pozivanjem API-eva, obrade velikih količina podataka, upravljanje podacima i sl. Kroz ovu skriptu naučit ćete pisati asinkroni Python kod koristeći biblioteku asyncio.

Posljednje ažurirano: 11.12.2024.

Sadržaj

- [Raspodijeljeni sustavi \(RS\)](#)
- [\(3\) Asinkroni Python: Osnove *asyncio* biblioteke](#)
 - [Sadržaj](#)
- [1. *asyncio* biblioteka](#)
 - [1.1. Korutine \(eng. Coroutines\)](#)
 - [1.2 Konkurentno izvršavanje više korutina](#)
 - [1.3 *asyncio tasks*](#)
 - [1.3.1 Konkurentno izvođenje kroz `asyncio.gather\(\)` i `asyncio.create_task\(\)`](#)
- [2. Zadaci za vježbu - Korutine, Task objekti, `asyncio.gather\(\)`](#)

1. `asyncio` biblioteka

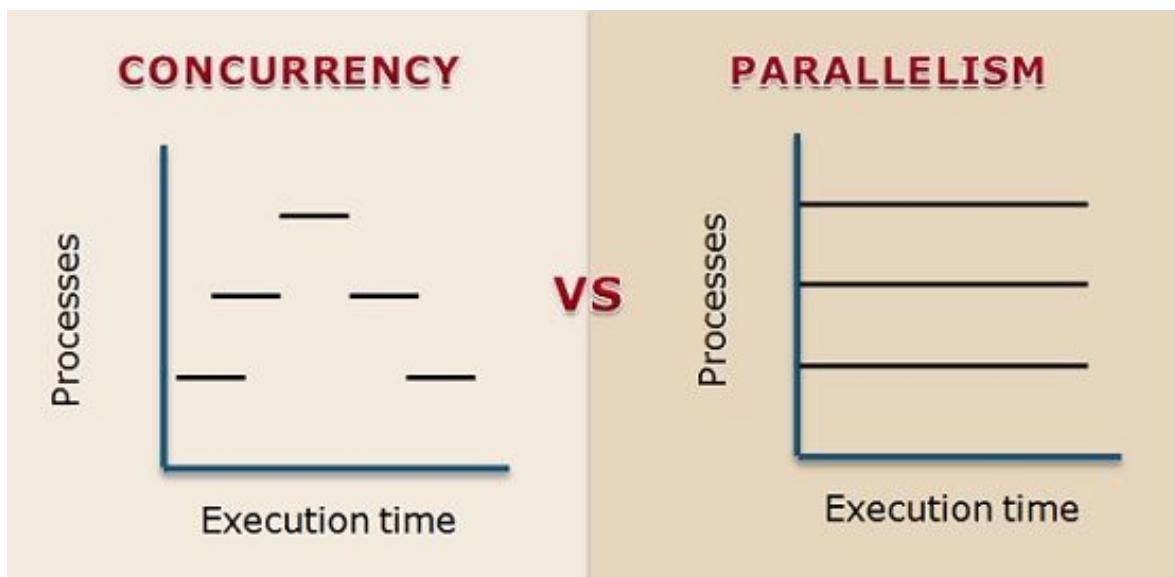
`asyncio` je biblioteka koja se koristi za pisanje konkurentnog koda koristeći `async/await` sintaksu. Ova biblioteka omogućuje nam da pišemo asinkroni kod koji se izvršava konkurentno, bez blokiranja izvršavanja programa te služi kao svojevrsni **temelj za pisanje asinkronih programa u Pythonu**.

Datoteka je uključena u standardnu biblioteku **Pythona 3.7+** pa ju nije potrebno naknadno instalirati.

Kratki osvrt na paralelno i konkurentno izvršavanje:

Paralelno izvršavanje (*eng. Parallelism*) sastoji se od izvršavanja više operacija simultano, odnosno u isto vrijeme. Ovo se postiže korištenjem prvenstveno više procesnih jedinica (*eng. CPU Cores*). Paralelno izvršavanje je fizičko i odvija se na različitim procesorskim jedinicama.

Konkurentno izvršavanje (*eng. Concurrency*) sastoji se od izvršavanja više operacija u isto vrijeme, ali ne nužno simultano. To znači da se operacije mogu međusobno preklapati u vremenu, ali se izmjenjuju u svom izvršavanju, koristeći najčešće jednu procesorsku jedinicu odnosno iste resurse. Konkurentnost se ostvaruje kroz mehanizme kao što su asinkrono programiranje, višedretvenost (*eng. multithreading*) te programiranje bazirano na događajima (*eng. event-driven programming*).



Na ovom kolegiju dotaknuti ćemo se prvenstveno **konkurentnog izvršavanja** kroz asinkrono programiranje, a u nešto manjoj mjeri i na paralelno izvršavanje.

1.1. Korutine (eng. Coroutines)

Ključne riječi `async` i `await` koriste se za:

1. **definiranje asinkronih (`async`) funkcija** (koje vraćaju `coroutine` objekte) te za
2. **čekanje na rezultat izvršavanja asinkronih funkcija (`await`)**.

Kako bismo simulirali asinkrono izvršavanje, iskoristit ćemo funkciju `asyncio.sleep()` koja simulira čekanje određenog vremena zadalog u **sekundama**.

Sintaksa:

```
asyncio.sleep(delay)
```

- `delay` - broj sekundi koliko želimo čekati - odgoditi izvršavanje koda

```
import asyncio

async def main():
    print('Hello')
    await asyncio.sleep(1)
    print('World')
```

U gornjem primjeru, funkcija `main()` je asinkrona funkcija koja ispisuje "Hello", čeka 1 sekundu te ispisuje "World". Međutim, kako bi se funkcija `main()` izvršila, potrebno ju je pokrenuti pomoću `asyncio.run()` funkcije.

`asyncio.run()` je također funkcija iz `asyncio` biblioteke kojom pokrećemo asinkronu (**korutinu**) pokretanjem tzv. *event loopa*. Kao obavezan argument, prima asinkronu funkciju koju želimo pokrenuti - u ovom slučaju to je funkcija `main()`.

Sintaksa:

```
asyncio.run(coroutine)
```

- `coroutine` - asinkrona funkcija koju želimo pokrenuti

Event loop je mehanizam koji upravlja izvršavanjem asinkronih funkcija, odnosno **korutina**.

Korutina (*eng. coroutine*) je specifična vrsta funkcije koja se može zaustaviti i nastaviti izvršavanje u bilo kojem trenutku. Korutine se koriste za pisanje asinkronog koda u Pythonu.

```
import asyncio

async def main():
    print('Hello')
    await asyncio.sleep(1)
    print('World')

asyncio.run(main())
```

Izvršavanjem gornjeg koda, dobit ćemo ispis:

```
Hello
World
```

Kao što vidimo, ispis "Hello" se pojavljuje odmah, dok se ispis "World" pojavljuje nakon 1 sekunde. Na ovaj način, napisali smo najjednostavniji primjer asinkronog izvršavanja koda.

1.2 Konkurentno izvršavanje više korutina

Recimo da imamo više korutina koje želimo pokrenuti. U praksi ćemo htjeti logiku za dohvaćanje podataka s weba (npr. preko API-ja) odvojiti od logike za obradu tih podataka. Idemo simulirati takav primjer:

```
import asyncio

async def fetch_data(): # primjer jednostavne korutine koja simulira dohvaćanje podataka
    print('Dohvaćam podatke...')
    data = {'iznos': '3000', 'stanje': 'uspješno'}
    await asyncio.sleep(2)
    print('Podaci dohvaćeni.')
    return data

async def main():
    data = await fetch_data()
    print(f'Podaci: {data}')

asyncio.run(main())
```

Što će se dogoditi kada pokrenemo gornji kod?

- Spoiler alert! Odgovor na pitanje

Međutim, što ako imamo više asinkronih funkcija koje želimo pokrenuti, a koje imaju **različite duljine trajanja/izvođenja**? U praksi to može biti slučaj kada dohvaćamo podatke s više različitih API-ja, gdje su neki API-evi brži, a neki sporiji.

Idemo simulirati takav primjer.

```
import asyncio

async def fetch_api_1():
    print('Dohvaćam podatke s API-ja 1...')
    await asyncio.sleep(2)
    print('Podaci s API-ja 1 dohvaćeni.')
    return {'api_1': 'uspješno'}

async def fetch_api_2():
    print('Dohvaćam podatke s API-ja 2...')
    await asyncio.sleep(4)
    print('Podaci s API-ja 2 dohvaćeni.')
    return {'api_2': 'uspješno'}
```

Kako ćemo definirati funkciju `main()` koja će pokrenuti obje asinkrone funkcije `fetch_api_1()` i `fetch_api_2()`?

Možemo pokušati na sljedeći način:

```

async def main():
    podaci_1 = await fetch_api_1()
    podaci_2 = await fetch_api_2()

    print(f'Podaci s API-ja 1: {podaci_1}')
    print(f'Podaci s API-ja 2: {podaci_2}')

asyncio.run(main())

```

Pokrenite kod, koliko je vremena potrebno da se dohvate svi podaci? Zašto?

► Spoiler alert! Odgovor na pitanje

Kako bismo riješili ovaj problem, koristit ćemo funkciju `asyncio.gather()` koja omogućuje pokretanje **više korutina konkurentno**. Ova funkcija prima više asinkronih funkcija kao argumente te ih pokreće istovremeno (ne nužno paralelno, ali konkurentno).

Sintaksa:

```
asyncio.gather(*coros)
```

- `*coros` - argumenti su asinkrone funkcije koje želimo pokrenuti

```

async def main():
    podaci_1, podaci_2 = await asyncio.gather(fetch_api_1(), fetch_api_2())

    print(f'Podaci s API-ja 1: {podaci_1}')
    print(f'Podaci s API-ja 2: {podaci_2}')

```

Pokrenite kod, koliko je vremena potrebno da se dohvate svi podaci? Zašto?

► Spoiler alert! Odgovor na pitanje

Primjer: Definirat ćemo korutinu `timer()` koja će simulirati otkucaje timera svake sekunde. Korutina prima 2 argumenta: naziv timera i broj sekundi koliko će trajati, a zatim svake sekunde ispisuje preostale vrijeme.

```

import asyncio

async def timer(name, delay):
    for i in range(delay, 0, -1):
        print(f'{name}: {i} sekundi preostalo...')
        await asyncio.sleep(1)
    print(f'{name}: Vrijeme je isteklo!')

async def main():
    await asyncio.gather( # pokrećemo dvije korutine konkurentno
        timer('Timer 1', 3),
        timer('Timer 2', 5)

```

```
)  
  
asyncio.run(main())
```

- Pokrenite kod i provjerite ispis.

Rezultat:

```
Timer 1: 3 sekundi preostalo...  
Timer 2: 5 sekundi preostalo...  
Timer 1: 2 sekundi preostalo...  
Timer 2: 4 sekundi preostalo...  
Timer 1: 1 sekundi preostalo...  
Timer 2: 3 sekundi preostalo...  
Timer 1: Vrijeme je isteklo!  
Timer 2: 2 sekundi preostalo...  
Timer 2: 1 sekundi preostalo...  
Timer 2: Vrijeme je isteklo!
```

1.3 asyncio tasks

Radni zadatak, odnosno `task` u `asyncio` su temeljni gradivni blokovi asinkronog programiranja u Pythonu. `Task` predstavlja izvršnu jedinicu, odnosno asinkronu operaciju, koja se zakazuje (*eng. schedules*) za izvršavanje u `event loop`-u.

`asyncio.create_task()` je funkcija koja stvara novi `Task` objekt koji izvršava asinkronu funkciju. Ova funkcija je korisna kada želimo definirati korutinu koju ćemo zakazati za konkurentno izvršavanje kasnije u programu.

Sintaksa:

```
asyncio.create_task(coroutine)
```

- `coroutine` - asinkrona funkcija koju želimo zakazati za konkurentno izvršavanje
- vraća `Task` objekt (`<class '_asyncio.Task'>`)

Implementirat ćemo prethodne primjer pozivanja API-ja koristeći `asyncio.create_task()`.

```
import asyncio

async def fetch_api_1():
    print('Dohvaćam podatke s API-ja 1...')
    await asyncio.sleep(2)
    print('Podaci s API-ja 1 dohvaćeni.')
    return {'api_1': 'uspješno'}

async def fetch_api_2():
    print('Dohvaćam podatke s API-ja 2...')
    await asyncio.sleep(4)
    print('Podaci s API-ja 2 dohvaćeni.')
    return {'api_2': 'uspješno'}
```

Korutine `fetch_api_1()` i `fetch_api_2()` su iste kao i prije, ali **postoji razlika u načinu pozivanja korutina**.

```
async def main():
    task_1 = asyncio.create_task(fetch_api_1())
    task_2 = asyncio.create_task(fetch_api_2())

    podaci_1 = await task_1
    podaci_2 = await task_2

    print(f'Podaci s API-ja 1: {podaci_1}')
    print(f'Podaci s API-ja 2: {podaci_2}')

asyncio.run(main())
```

Pokrenite kod, koliko je vremena potrebno da se dohvate svi podaci? Zašto?

► Spoiler alert! Odgovor na pitanje

Općenito, koristeći `asyncio.create_task()` možemo pokrenuti više korutina konkurentno, a zatim čekati na njihov završetak.

Sintaksa:

```
task_1 = asyncio.create_task(coroutine_1())
task_2 = asyncio.create_task(coroutine_2())

await task_1 # čekamo na završetak prve korutine
await task_2 # čekamo na završetak druge korutine
```

Dakle, kod iznad će se izvršiti **konkurentno**, a ne sekvensijalno.

1.3.1 Konkurentno izvođenje kroz `asyncio.gather()` i `asyncio.create_task()`

Kombinirajmo prethodne primjere korištenjem `asyncio.create_task()` i `asyncio.gather()`.

Želimo definirati jednu korutinu `korutina(n)` koja će čekati jednu sekundu, a zatim vratiti poruku o završetku izvođenja.

```
import asyncio

async def korutina(n):
    await asyncio.sleep(1)
    return f'Korutina {n} je završila.'
```

U `main()` funkciji ćemo pohraniti 5 korutina u liste `tasks`. Drugim riječima, želimo pohraniti 5 `Task` objekata koji će izvršavati korutine `korutina(n)`, za `n` od 1 do 5.

```
async def main():
    tasks = []

    for i in range(1, 6):
        task = asyncio.create_task(korutina(i))
        tasks.append(task)

    print(tasks) # ispis svih referenci na Task objekte

asyncio.run(main())
```

Kako ovo možemo napraviti elegantnije? `list comprehension` nam može pomoći.

```
async def main():
    tasks = [asyncio.create_task(korutina(i)) for i in range(1, 6)]
    print(tasks) # ispis svih referenci na Task objekte

asyncio.run(main())
```

Za pokretanje svih korutina konkurentno, ne želimo pisati `await task` za svaki `Task` objekt.

Dakle, **sljedeće nije najbolje rješenje:**

```

async def main():
    tasks = [asyncio.create_task(korutina(i)) for i in range(1, 6)]

    for task in tasks:
        await task

    print('Sve korutine su završile.')

asyncio.run(main())

```

Zašto? Nigdje ne pohranjujemo rezultate korutina, već samo čekamo na njihov završetak.

Stvari možemo riješiti ovako:

```

async def main():
    tasks = [asyncio.create_task(korutina(i)) for i in range(1, 6)]

    results = []

    for task in tasks:
        results.append(await task) # čekamo na završetak svake korutine i pohranjujemo rezultat

    print(results)

asyncio.run(main())

```

Međutim, puno bolje rješenje je koristiti `asyncio.gather()`.

- `asyncio.gather()` osim korutina može primiti i `Task` objekte
- možemo proslijediti jedan ili više `Task` objekata na isti način kao i korutine: `await asyncio.gather(task_1, task_2, task_3)`
- međutim, možemo proslijediti i listu korutina ili `Task` objekata s operatorom `*`: `await asyncio.gather(*tasks)`

```

async def main():
    tasks = [asyncio.create_task(korutina(i)) for i in range(1, 6)]
    results = await asyncio.gather(*tasks)
    print(results)

asyncio.run(main())
# Ispisuje: ['Korutina 1 je završila.', 'Korutina 2 je završila.', 'Korutina 3 je završila.', 'Korutina 4 je završila.', 'Korutina 5 je završila.']

```

Na ovaj način, `asyncio.gather(*tasks)` čeka na završetak svih korutina i vraća **listu rezultata izvođenja korutina**.

Pogledat ćemo još nekoliko jednostavnih primjera i mjeriti vrijeme izvođenja programa koristeći `time` modul.

Primjer: Definirat ćemo korutinu koja će nakon određenog vremena ispisati poruku.

```
import asyncio
import time

async def kaži_nakon(delay, poruka):
    await asyncio.sleep(delay)
    print(poruka)

async def main():
    print(f"Početak: {time.strftime('%X')}")

    await kaži_nakon(1, 'Pozdraav!')
    await kaži_nakon(2, 'Kako si?')

    print(f"Kraj: {time.strftime('%X')}")

asyncio.run(main())
```

- Ako pokrenemo program u ovom obliku u 11:00:00, što će biti ispisano?

```
Početak: 11:00:00
Pozdraav!
Kako si?
Kraj: 11:00:03
```

- Isto možemo pretočiti u Task objekte:

```
async def main():
    print(f"Početak: {time.strftime('%X')}

task1 = asyncio.create_task(kaži_nakon(1, 'Pozdraav!'))
task2 = asyncio.create_task(kaži_nakon(2, 'Kako si?'))

await task1
await task2

print(f"Kraj: {time.strftime('%X')}")

asyncio.run(main())
```

- ili koristeći `asyncio.gather()`:

```

async def main():
    print(f"Početak: {time.strftime('%X')}")

    task1 = asyncio.create_task(kaži_nakon(1, 'Pozdraaav!'))
    task2 = asyncio.create_task(kaži_nakon(2, 'Kako si?'))

    await asyncio.gather(task1, task2)

    print(f"Kraj: {time.strftime('%X')}")

asyncio.run(main())

```

Primjer: Idemo vidjeti kako možemo na isti način koristiti `asyncio.gather()` za pozivanje prethodne korutine `Timer(name, delay)` koja simulira otkucaje timera svake sekunde. Korutinu želimo pokrenuti 3 puta s različitim vremenima trajanja. Potrebno je definirati `Task` objekte i pohraniti ih u listu `tasks`, a zatim koristiti `asyncio.gather()` za pokretanje svih korutina konkurentno.

```

import asyncio

async def timer(name, delay):
    for i in range(delay, 0, -1):
        print(f'{name}: {i} sekundi preostalo...')
        await asyncio.sleep(1)
    print(f'{name}: Vrijeme je isteklo!')

async def main():
    timers = [
        asyncio.create_task(timer('Timer 1', 3)),
        asyncio.create_task(timer('Timer 2', 5)),
        asyncio.create_task(timer('Timer 3', 7))
    ]

    await asyncio.gather(*timers)

asyncio.run(main())

```

2. Zadaci za vježbu - Korutine, Task objekti, asyncio.gather()

- Definirajte korutinu koja će simulirati dohvaćanje podataka s weba.** Podaci neka budu lista brojeva od 1 do 10 koju ćete vratiti nakon 3 sekunde. Listu brojeva definirajte comprehensionom. Nakon isteka vremena, u korutinu ispišite poruku "Podaci dohvaćeni." i vratite podatke. Riješite bez korištenja `asyncio.gather()` i `asyncio.create_task()` funkcija.
- Definirajte dvije korutine koje će simulirati dohvaćanje podataka s weba.** Prva korutina neka vrati listu proizvoljnih rječnika (npr. koji reprezentiraju podatke o korisnicima) nakon 3 sekunde, a druga korutina neka vrati listu proizvoljnih rječnika (npr. koji reprezentiraju podatke o proizvodima) nakon 5 sekundi. Korutine pozovite konkurentno korištenjem `asyncio.gather()` i ispišite rezultate. Program se mora izvršavati ~5 sekundi.
- Definirajte korutinu `autentifikacija()` koja će simulirati autentifikaciju korisnika na poslužiteljskoj strani.** Korutina kao ulazni parametar prima rječnik koji opisuje korisnika, a sastoji se od ključeva `korisnicko_ime`, `email` i `lozinka`. Unutar korutine simulirajte provjeru korisničkog imena na način da ćete provjeriti nalaze li se par `korisnicko_ime` i `email` u bazi korisnika. Ova provjera traje 3 sekunde.

```
baza_korisnika = [
    {'korisnicko_ime': 'mirko123', 'email': 'mirko123@gmail.com'},
    {'korisnicko_ime': 'ana_anic', 'email': 'aanic@gmail.com'},
    {'korisnicko_ime': 'maja_0x', 'email': 'majaaaaaa@gmail.com'},
    {'korisnicko_ime': 'zdeslav032', 'email': 'deso032@gmail.com'}
]
```

Ako se korisnik ne nalazi u bazi, vratite poruku "Korisnik {korisnik} nije pronađen."

Ako se korisnik nalazi u bazi, potrebno je pozvati vanjsku korutinu `autorizacija()` koja će simulirati autorizaciju korisnika u trajanju od 2 sekunde. Funkcija kao ulazni parametar prima rječnik korisnika iz baze i lozinku proslijeđenu iz korutine `autentifikacija()`. Autorizacija simulira dekripciju lozinke (samo provjerite podudaranje stringova) i provjeru s lozinkom iz baze_lozinka. Ako su lozinke jednake, korutine vraća poruku "Korisnik {korisnik}: Autorizacija uspješna.", a u suprotnom "Korisnik {korisnik}: Autorizacija neuspješna."

```
baza_lozinka = [
    {'korisnicko_ime': 'mirko123', 'lozinka': 'lozinka123'},
    {'korisnicko_ime': 'ana_anic', 'lozinka': 'super_teska_lozinka'},
    {'korisnicko_ime': 'maja_0x', 'lozinka': 's324SDFFdsj234'},
    {'korisnicko_ime': 'zdeslav032', 'lozinka': 'deso123'}
]
```

Korutinu `autentifikacija()` pozovite u `main()` funkciji s proizvoljnim korisnikom i lozinkom.

4. Definirajte korutinu `provjeri_parnost` koja će simulirati "super zahtjevnu operaciju" provjere **parnosti** broja putem vanjskog API-ja. Korutina prima kao argument broj za koji treba provjeriti parnost, a vraća poruku "Broj {broj} je paran." ili "Broj {broj} je neparan." nakon 2 sekunde. Unutar `main` funkcije definirajte listu 10 nasumičnih brojeva u rasponu od 1 do 100 (koristite `random` modul). Listu brojeva izgradite kroz list comprehension sintaksu. Nakon toga, pohranite u listu `zadaci` 10 `Task` objekata koji će izvršavati korutinu `provjeri_parnost` za svaki broj iz liste (također kroz list comprehension). Na kraju, koristeći `asyncio.gather()`, pokrenite sve korutine konkurentno i ispišite rezultate.
5. Definirajte korutinu `secure_data` koja će simulirati enkripciju osjetljivih podataka. Kako se u praksi enkripcija radi na poslužiteljskoj strani, korutina će simulirati enkripciju podataka u trajanju od 3 sekunde. Korutina prima kao argument rječnik osjetljivih podataka koji se sastoji od ključeva `prezime`, `broj_kartice` i `cvv`. Definirajte listu s 3 rječnika osjetljivih podataka. Pohranite u listu `zadaci` kao u prethodnom zadatku te pozovite zadatke koristeći `asyncio.gather()`. Korutina `secure_data` mora za svaki rječnik vratiti novi rječnik u obliku: `{'prezime': prezime, 'broj_kartice': 'enkriptirano', 'cvv': 'enkriptirano'}`. Za fake enkripciju koristite funkciju `hash(str)` koja samo vraća hash vrijednost ulaznog stringa.
-

Raspodijeljeni sustavi (RS)

Nositelj: doc. dr. sc. Nikola Tanković

Asistent: Luka Blašković, mag. inf.

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



Fakultet informatike u Puli

(4) Asinkroni Python: Slanje konkurentnih HTTP zahtjeva

#4

RS

HTTP (Hypertext Transfer Protocol) je protokol koji omogućuje prijenos podataka između klijenta i servera na webu. Asinkroni Python omogućuje nam da programiramo poslužitelje koji mogu istovremeno obrađivati više zahtjeva bez blokiranja glavnog toka programa, čime se postiže bolja učinkovitost, osobito u aplikacijama koje zahtijevaju visoku propusnost, kao što su web servisi i API klijenti. Korištenjem biblioteke kao što je `aiohttp`, možemo jednostavno implementirati asinkrone HTTP klijente i poslužitelje u Pythonu. U prošloj skripti upoznali ste se s asinkronim programiranjem u Pythonu pomoću `asyncio` biblioteke, a u ovoj ćete naučiti kako kombinirati asinkrone funkcionalnosti s HTTP zahtjevima i odgovorima.

Posljednje ažurirano: 21.11.2024.

Sadržaj

- [Raspodijeljeni sustavi \(RS\)](#)
- [\(4\) Asinkroni Python: Slanje konkurentnih HTTP zahtjeva](#)
 - [Sadržaj](#)
- [1. Ponavljanje HTTP protokola](#)
 - [1.1. Struktura HTTP zahtjeva \(eng. HTTP request\)](#)
 - [1.2 Struktura HTTP odgovora \(eng. HTTP response\)](#)
- [2. Slanje konkurentnih HTTP zahtjeva pomoću aiohttp biblioteke](#)
 - [2.1 Kako šaljemo HTTP zahtjeve sinkrono?](#)
 - [2.2 Asinkrono slanje HTTP zahtjeva](#)
 - [2.2.1 Context Manager `with`](#)
 - [2.2.2 `clientSession` klasa](#)
 - [2.2.3 Konkurentno slanje kroz `asyncio.gather`](#)

- [2.2.4 Konkurentno slanje kroz `asyncio.Task`](#)
- [3. Zadaci za vježbu - Slanje konkurentnih HTTP zahtjeva](#)

1. Ponavljanje HTTP protokola

HTTP (eng. *Hypertext Transfer Protocol*) odnosi se na protokol koji se koristi za prijenos podataka putem weba. Omogućuje web preglednicima, udaljenim poslužiteljima i ostalim dijelovima sustavne cjeline da komuniciraju međusobno. HTTP je protokol bez stanja (eng. *stateless*), što znači da svaki zahtjev klijenta poslužitelju ne ovisi o prethodnim zahtjevima. Svaki zahtjev se tretira kao zaseban zahtjev, bez obzira na prethodne.

Tipična HTTP komunikacijski model (**klijent** ↔ **poslužitelj**) sastoji se HTTP zahtjeva (eng. *request*) i HTTP odgovora (eng. *response*).

- **HTTP zahtjev** (eng. *HTTP request*): odnosi se na **zahtjev koji klijent šalje poslužitelju**. Npr. web preglednik šalje zahtjev za resurs udaljenom poslužitelju
- **HTTP odgovor** (eng. *HTTP response*): odnosi se na **odgovor koji poslužitelj šalje klijentu**. Npr. poslužitelj šalje odgovor s JSON podacima u tijelu odgovora

1.1. Struktura HTTP zahtjeva (eng. HTTP request)

- **Metoda** (eng. *method*): odnosi se na vrstu zahtjeva (npr. `GET`, `POST`, `PUT`, `PATCH`, `DELETE`)
- **URL** (eng. *Uniform Resource Locator*): odnosi se na adresu resursa na poslužitelju (npr. `https://api.github.com/users/nek_i_korisnik`
 - **Shema** (eng. *scheme*): odnosi se na protokol koji se koristi (npr. `https`)
 - **Domena** (eng. *domain*): odnosi se na ime domene poslužitelja (npr. `api.github.com`)
 - **Route parametar** (eng. *route*): odnosi se na dinamički dio URL-a, najčešće za identifikaciju pojedinog resursa (npr. `/users/:id`)
 - **Query parametar** (eng. *query*): odnosi se na dodatne parametre upita, najčešće za filtriranja, sortiranja i sl. (npr. `?page=1&limit=10`)
 - **Fragment** (eng. *fragment*): odnosi se na oznaku dijela resursa (npr. `#section1`)
- **Zaglavlja** (eng. *headers*): odnose se na dodatne informacije o zahtjevu (npr. `Content-Type: text/html; charset=utf-8`)
- **Tijelo** (eng. *body*): odnosi se na podatke koji se šalju s zahtjevom (npr. `JSON`)
- **Verzija protokola** (eng. *protocol version*): odnosi se na verziju HTTP protokola (npr. `HTTP/1.1`)

HTTP metode koje se najčešće koriste su:

- **GET**: dohvaća resurs/resurse s poslužitelja (npr. podatke o korisniku)
- **POST**: šalje podatke na poslužitelj (npr. podatke iz forme)
- **PUT**: ažurira resurs na poslužitelju u cijelosti (npr. zamjenjuje postojeće podatke o korisniku novima)
- **PATCH**: ažurira resurs na poslužitelju djelomično (npr. izmjenjuje lozinku korisnika)
- **DELETE**: briše resurs s poslužitelja (npr. briše korisnika)

Podsjetnik: Uobičajeno je koristiti **GET** metodu za dohvaćanje podataka, s dodatnim query parametrima za filtriranje, sortiranje, paginaciju i sl. ili s dodatnim route parametrima za identifikaciju pojedinog resursa, npr. kada želimo dohvatiti samo jednog korisnika. Nije po standardu slati **tijelo zahtjeva** unutar **GET** metode.

Podsjetnik: Metodu **POST** koristimo kada želimo poslati podatke na poslužitelj, npr. kada šaljemo podatke iz forme ili kada želimo izraditi novi resurs na poslužitelju. U tom slučaju, **šaljemo podatke u tijelu zahtjeva**, najčešće u JSON formatu iako je moguće koristiti i druge. Razlog zašto šaljemo podatke u tijelu zahtjeva, a ne kao query parametre, jest što je tijelo zahtjeva skriveno od korisnika - ne pojavljuje se u URL-u.

Podsjetnik: Metode **PUT** i **PATCH** koristimo kada želimo ažurirati resurs na poslužitelju. Razlika između njih je što **PUT** zamjenjuje cijeli resurs novim podacima, dok **PATCH** ažurira resurs djelomično. Primjerice, kada ažuriramo korisnika, **PUT** metodom zamijenili bismo sve podatke o korisniku novima, dok bismo **PATCH** metodom mogli ažurirati samo određeni podatak ili više njih. Podatke koje ažuriramo također šaljemo u tijelu zahtjeva.

Primjer: HTTP zahtjev koji dohvaća podatke o korisniku s GitHuba `pero_peric`:

```
GET https://api.github.com/users/pero_peric HTTP/1.1
```

Primjer: HTTP zahtjev koji šalje podatke o novom korisniku na poslužitelj:

```
POST https://api.github.com/users HTTP/1.1
```

Content-Type: application/json

```
{
  "username": "pero_peric",
  "email": "pperic@gmail.com",
  "password": "pero123"
}
```

Primjer: HTTP zahtjev koji ažurira username korisnika `pero_peric` na poslužitelju:

```
PATCH https://api.github.com/users/pero_peric HTTP/1.1
```

```
{
  "username": "pero_peric_2"
}
```

Primjer: HTTP zahtjev koji briše korisnika `pero_peric` s poslužitelja:

```
DELETE https://api.github.com/users/pero_peric HTTP/1.1
```

Primjer: HTTP zahtjev koji dohvaća samo korisnike s imenom `pero`:

```
GET https://api.github.com/users?name=pero HTTP/1.1
```

Primjer: HTTP zahtjev koji zamjenjuje sve podatke o korisniku `pero_peric` novima:

```
PUT https://api.github.com/users/pero_peric HTTP/1.1
```

```
{  
    "username": "pero_peric_2",  
    "email": "pperic2@gmail.com",  
    "password": "pppper01234"  
}
```

1.2 Struktura HTTP odgovora (eng. HTTP response)

- **Statusna linija** (eng. *status line*): odnosi se na **statusni kod i poruku** (npr. `200 OK`) gdje je `200` statusni kod, a `OK` poruka
- **Zaglavla** (eng. *headers*): odnose se na dodatne informacije o odgovoru (npr. `Content-Type: application/json`)
- **Tijelo** (eng. *body*): odnosi se na podatke koji se šalju s odgovorom (npr. `JSON`)
- **Verzija protokola** (eng. *protocol version*): odnosi se na verziju HTTP protokola (npr. `HTTP/1.1`)

Podsjetnik: Statusni kodovi (eng. *HTTP status codes*) su brojevi koji se koriste u **HTTP odgovorima** kako bi klijentu dali informaciju u kojem je stanju zahtjev koji je poslao. Drugim riječima, ako klijent pošalje zahtjev koji rezultira greškom, poslužitelj uz odgovarajuću poruku vraća i statusni kod koji označava vrstu greške.

Statusne kodove možemo podijeliti u sljedeće skupine:

- `1xx` (100 - 199) - Informacijski odgovori (eng. *Informational responses*): Poslužitelj je primio zahtjev te ga i dalje obrađuje
- `2xx` (200 - 299) - Odgovori uspjeha (eng. *Successful responses*): Zahtjev klijenta uspješno primljen i obrađen
- `3xx` (300 - 399) - Odgovori preusmjeravanja (eng. *Redirection messages*): Ova skupina kodova govori klijentu da mora poduzeti dodatne radnje kako bi dovršio zahtjev
- `4xx` (400 - 499) - Greške na strani klijenta (eng. *Client error responses*): Sadrži statusne kodove koji se odnose na greške nastale na klijentskoj strani
- `5xx` (500 - 599) - Greške na strani poslužitelja (eng. *Server error responses*): Sadrži statusne kodove koji se odnose na greške nastale na poslužiteljskoj strani

Statusni kodovi neizbjeglan su dio HTTP komunikacije, a njihovom primjenom **standardiziramo komunikaciju između klijenta i poslužitelja**. Na taj način, klijent može interpretirati odgovor poslužitelja i ovisno o statusnom kodu poduzeti odgovarajuće radnje.

Statusnih kodova ima mnogo, a svaki od njih ima svoje značenje. Možete pronaći **popis i definicije** svih statusnih kodova na [ovoj poveznici](#).

Međutim, u praksi se ne najčešće ne koriste svi statusni kodovi, već nekolicina njih. Evo nekoliko najčešće korištenih statusnih kodova:

- `200` - OK: Zahtjev je uspješno primljen i obrađen (npr. GET zahtjev za dohvata svih resursa)

- **201** - Created: Resurs je uspješno stvoren (npr. nakon slanja POST zahtjeva)
- **400** - Bad Request: Zahtjev nije moguće obraditi zbog neispravnih podataka (npr. korisnik je poslao neispravan ID resursa u zahtjevu)
- **404** - Not Found: Resurs nije pronađen (npr. korisnik je poslao ID resursa koja ne postoji na poslužitelju)
- **500** - Internal Server Error: Opća greška na poslužitelju (npr. greška prilikom obrade zahtjeva, najvjerojatnije zbog greške u kodu na poslužitelju)

Postoji puno varijacija 4xx, 5xx i 2xx statusnih kodova, pa tako imamo:

- **401** - Unauthorized: Korisnik nije autoriziran za pristup resursu (npr. korisnik nema prava pristupa resursu jer nije prijavljen)
- **204** - No Content: Zahtjev je uspješno primljen i obrađen, ali nema sadržaja za prikazati (npr. nakon brisanja resursa)
- **403** - Forbidden: Korisnik nema prava pristupa resursu (npr. korisnik nema prava pristupa resursu jer nije administrator)
- **301** - Moved Permanently: Resurs je trajno premješten na novu lokaciju (npr. kada se mijenja URL resursa)
- **503** - Service Unavailable: Poslužitelj nije dostupan (npr. poslužitelj je preopterećen)
- **409** - Conflict: Zahtjev nije moguće obraditi zbog konflikta (npr. korisnik pokušava ažurirati resurs koji je već ažuriran, npr. kod PUT/PATCH zahtjeva)

VAŽNO: Za studente koji imaju poteškoća s razumijevanjem HTTP protokola iz sažetka danog u ovom poglavlju, preporuka je da prouče skriptu [WA1 iz kolegija Web aplikacije](#) s prijediplomskog studija Informatike u Puli. Potrebno se prijaviti s [AAI@EduHr](#) računom na [UNIPU](#) domeni.

2. Slanje konkurentnih HTTP zahtjeva pomoću aiohttp biblioteke

aiohttp (*Asynchronous HTTP Client/Server for asyncio and Python*) je biblioteka koja omogućuje **asinkrono programiranje HTTP klijenata i poslužitelja u Pythonu**. Ova datoteka izgrađena je na temelju `asyncio` biblioteke s kojom smo se upoznali u skripti `rs3`.

aiohttp biblioteka omogućuje nam da jednostavno implementiramo asinkrone HTTP klijente i poslužitelje u Pythonu, što je korisno u kontekstu razvoja i testiranja malih web servisa koji zahtijevaju visoku propusnost. Dodatno, datoteka pruža podršku za [WebSocket protokol](#), što je korisno za razvoj aplikacija u stvarnom vremenu (eng. *real-time applications*).

Za razliku od `asyncio` biblioteke koja je ugrađena u Python 3.7+, aiohttp biblioteku potrebno je instalirati ručno:

```
pip install aiohttp
```

Napomena, kod instalacije vanjskih paketa **preporučuje se korištenje virtualnog okruženja** kako bi se izbjegli konflikti između paketa.

Ako ste se odlučili koristiti `conda` alat za upravljanje virtualnim okruženjima, napraviti novo okruženje naziva `rs4` prije nego instalirate aiohttp biblioteku:

```
conda create --name rs4 python=3.13
```

Aktivirajte novo okruženje:

```
conda activate rs4
```

Unutar VS Codea promijenite interpreter na novo kreirano okruženje `rs4` kako biste izbjegli [linting greške](#).

Sada možete instalirati biblioteke 

2.1 Kako šaljemo HTTP zahtjeve sinkrono?

Međutim, prije nego se upoznamo s asinkronim načinom definiranja HTTP klijenata, vrijedno je prisjetiti se kako to radimo sinkrono, koristeći biblioteku `requests`.

`requests` je popularna biblioteka za rad s HTTP zahtjevima u Pythonu koja omogućuje jednostavno slanje zahtjeva na poslužitelj i primanje odgovora. Međutim, `requests` je **sinkrona biblioteka**, što znači da će svaki zahtjev blokirati izvođenje programa dok se ne primi odgovor.

Kako bismo poslali HTTP zahtjev koristeći `requests` biblioteku, prvo je potrebno instalirati biblioteku:

```
pip install requests
```

Uključimo `requests` biblioteku:

```
import requests
```

Jednostavni primjer slanja GET zahtjeva na poslužitelj. Zahtjev ćemo poslati na [Cat Facts API](#) koji vraća nasumične činjenice o mačkama:

```
import requests

response = requests.get("https://catfact.ninja/fact")
print(response.text)
```

Ako pokrenemo ovaj kod, dobit ćemo nasumični odgovor u obliku rječnika s ključevima `fact` i `length`:

```
{
    "fact": "The life expectancy of cats has nearly doubled over the last fifty years.",
    "length": 73
}
```

Možemo provjeriti statusni kod odgovora:

```
print(response.status_code) # 200
```

Rekli smo da u sinkronim programima, svaki zahtjev koji pošaljemo čeka na odgovor prethodnog prije nego pošaljemo novi. Ako neki zahtjevi traju dugo, to može značajno usporiti izvođenje programa.

Primjer: Poslat ćemo 5 zahtjeva na *Cat Facts API*, kod za slanje možemo spakirati u jednostavnu funkciju koja šalje GET zahtjev i ispisuje rezultat. Zahtjev možemo dohvatiti pod ključem `fact`, ali prije tog moramo napraviti deserijalizaciju JSON odgovora koristeći metodu `json()`:

```
import requests

def send_request():
    response = requests.get("https://catfact.ninja/fact")
    fact = response.json()["fact"]
    print(fact)

print("Šaljemo 1. zahtjev...")
send_request()

print("Šaljemo 2. zahtjev...")
send_request()

print("Šaljemo 3. zahtjev...")
send_request()

print("Šaljemo 4. zahtjev...")
send_request()

print("Šaljemo 5. zahtjev...")
send_request()
```

Vidimo da je za izvršavanje svakog zahtjeva potrebno pričekati odgovor prethodnog; na taj način smo napisali kod i to je OK. Ukupno vrijeme trajanja ovog programa je prosječno 1-2 sekunde, ovisno o brzini interneta.

Možemo koristiti biblioteku `time` kako bismo preciznije izmjerili vrijeme izvršavanja programa:

```
import requests
import time

def send_request():
    response = requests.get("https://catfact.ninja/fact")
    fact = response.json()["fact"]
    print(fact)

start = time.time()

print("Šaljemo 1. zahtjev...")
send_request()

print("Šaljemo 2. zahtjev...")
send_request()
```

```

print("Šaljemo 3. zahtjev...")
send_request()

print("Šaljemo 4. zahtjev...")
send_request()

print("Šaljemo 5. zahtjev...")
send_request()

end = time.time()
print(f"Izvršavanje programa traje {end - start:.2f} sekundi.")

```

Poslat ćemo 15 zahtjeva, kod možemo i strukturirati u petlju:

```

import requests
import time

def send_request():
    response = requests.get("https://catfact.ninja/fact")
    fact = response.json()["fact"]
    print(fact)

start = time.time()

for i in range(15):
    print(f"Šaljemo {i + 1}. zahtjev...")
    send_request()

end = time.time()
print(f"Izvršavanje programa traje {end - start:.2f} sekundi.")

```

Prosječno vrijeme trajanja programa iznad je 3-4 sekunde. Ako povećamo broj zahtjeva, **vrijeme izvršavanja će se povećati proporcionalno broju zahtjeva**. Obzirom da vrijeme izvođenja programa direktno ovisi o broju iteracija `i`, možemo reći da je vremenska složenost $O(n)$.

Zahtjeve smo slali sinkrono (sekvencijalno), i to je vrlo vidljivo na ovom primjeru. Zamislimo da šaljemo 1000 zahtjeva, ili 10 000 zahtjeva - program bi trajao jako dugo, a poslužitelj bi morao čekati na svaki zahtjev. Na ovaj način, ne iskorištavamo puni potencijal poslužitelja, a aplikacije koje razvijamo nisu skalabilne.

Idemo vidjeti kako bismo to mogli riješiti asinkronim programiranjem odnosno **konkurentnim slanjem zahtjeva** na poslužitelj pomoću `aiohttp` biblioteke.

2.2 Asinkrono slanje HTTP zahtjeva

Cilj nam je poslati više zahtjeva na *Cat Facts API* i postići brže vrijeme izvršavanja programa (ne želimo da slanje i ispis rezultata 15 zahtjeva traje gotovo 4 sekunde).

Ako već niste, instalirajte `aiohttp` biblioteku.

Kako zahtjeve šaljemo konkurentno, najpraktičnije je kod spakirati u korutine. Međutim, kako bi radili s korutinama i asinkronim programiranjem općenito, svakako je potrebno uključiti i `asyncio` biblioteku.

```
import aiohttp
import asyncio
import time
```

Nećemo više koristiti `requests` biblioteku, zamijenili smo je s `aiohttp`.

Biblioteka `requests` u pozadini je definirala korisničku sesiju (eng. *session*) koja je omogućavala **ponovnu upotrebu veze s poslužiteljem**, odnosno pohranu HTTP zaglavlja, autentifikacije, kolačića i drugih objekata koji se ponavljaju u svakom zahtjevu. Na taj način, umjesto da se radi nova sesija za svako slanje zahtjeva, moguće je **ponovno koristiti već postojeću sesiju**.

U `aiohttp` biblioteci, potrebno je naglasiti definiranje sesije - ona nam omogućuje iste funkcionalnosti koje su prethodno navedene.

2.2.1 Context Manager `with`

Koncept **kontekstnog menadžera** (eng. *Context Manager*) u Pythonu omogućavaju nam alokaciju i dealokaciju resursa, odnosno upravljanje resursima koji se koriste u bloku koda.

Najčešće korišteni primjer *context managera* u Pythonu je naredba `with` koju koristimo kako bismo definirali **blok koda za rad s resursima** koje treba eksplicitno **(1) otvoriti, (2) koristiti i (3) zatvoriti**.

Primjerice:

- **datoteke** (otvaranje → čitanje/pisanje → zatvaranje)
- **mrežne veze** (otvaranje → slanje zahtjeva → zatvaranje)
- **baze podataka** (otvaranje → izvršavanje upita → zatvaranje)

Naredba `with` omogućava automatsko upravljanje resursima, osiguravajući da će se resursi pravilno osloboditi i zatvoriti čak i ako dođe do greške u bloku koda. Na taj način, kod postaje čišći i sigurniji.

Sintaksa naredbe `with`:

```
with neki_resurs as alias:
    # rad s resursom koristeći "alias"
```

Tipičan primjer korištenje naredbe `with` je rad s datotekama:

```
with open("datoteka.txt", "r") as file: # otvaramo datoteku za čitanje i koristimo alias  
"file"  
    sadržaj = file.read() # čitamo sadržaj datoteke  
    print(sadržaj)
```

Bez korištenja naredbe `with`, morali bismo eksplicitno zatvoriti datoteku nakon što smo pročitali sadržaj:

```
file = open("datoteka.txt", "r")  
sadržaj = file.read()  
print(sadržaj)  
file.close() # zatvaramo datoteku
```

Međutim kod iznad ne obuhvaća slučaj greške prilikom čitanja ili pisanja u datoteku ako postoji. U tom slučaju, trebali bismo koristiti `try-except-finally` blokove kako bismo osigurali da će se datoteka zatvoriti čak i ako dođe do greške.

```
try:  
    file = open("datoteka.txt", "r")  
    sadržaj = file.read()  
    print(sadržaj)  
except Exception as e:  
    print(f"Greška: {e}")  
finally:  
    file.close()
```

Osim spomenutih primjera resursa, možemo definirati i [vlastite kontekstne menadžere](#), međutim to nije predmet ove skripte.

Naredba `with` **automatski zatvara resurs čak i ako dođe do greške u bloku koda**, što je jedan od razloga zašto se preporučuje njeno korištenje.

Dodatno, vidimo da je kod s naredbom `with` **kraći i lakši za čitanje**.

2.2.2 ClientSession klasa

Vratimo se na naš primjer slanja 15 zahtjeva na *Cat Facts API*. Što je ovdje resurs koji trebamo otvoriti i zatvoriti, u kontekstu `with` naredbe?

- ▶ Spoiler alert! Odgovor na pitanje

U `aiohttp` biblioteci, za rad s HTTP sesijom koristimo klasu `ClientSession`.

Klasa `clientSession` predstavlja asinkroni HTTP klijent koji omogućuje **konkurentno slanje HTTP zahtjeva unutar Python programa**. Ovaj klijent implementiran je kao kontekstni menadžer, što znači da ga možemo koristiti unutar `with` bloka.

- Kako bismo stvorili novu instancu `ClientSession` klase, kao i klase općenito, jednostavno pozivamo njen konstruktor:

U varijablu `session` spremamo instancu klase `ClientSession`:

```
session = aiohttp.ClientSession()
```

Nakon što smo stvorili instancu klase, možemo koristiti `with` blok kako bismo definirali blok koda za asinkroni rad s HTTP sesijom. Jedina razlika je što sad stvari radimo asinkrono pa moramo koristiti `async` ispred kontekstnog menadžera.

- Obzirom da koristimo `with`, možemo definirati alias `session` za **instancu unutar same naredbe**:

```
async with aiohttp.ClientSession() as session:  
    # rad s HTTP sesijom
```

- Nad našom sesijom `session` sad možemo koristiti metodu `get` za slanje GET zahtjeva na isti način kao što smo to radili s `requests` bibliotekom:

```
async with aiohttp.ClientSession() as session:  
    response = await session.get("https://catfact.ninja/fact")  
    print(response)
```

Kako ovo sad pozvati? **Context manager sam po sebi nije funkcija, niti korutina**. Zato ga moramo pozvati unutar `async` funkcije ili korutine.

- jednostavno ga dodajemo unutar `main` korutine

```

async def main(): # definiramo main korutinu
    async with aiohttp.ClientSession() as session: # otvaramo HTTP sesiju koristeći context
        manager "with"
        response = await session.get("https://catfact.ninja/fact")
        print(response)

# pokrećemo main korutinu koristeći asyncio.run()
asyncio.run(main())

```

- Ako pokrenete kod vidjet ćete ogroman ispis, to je zato što smo ispisali cijeli HTTP odgovor, uključujući zaglavlja, statusnu liniju, tijelo itd...

Kako bismo dobili samo tijelo odgovora, možemo na isti način kao i kod `requests` biblioteke koristiti metodu `json()` za deserijalizaciju, ali s jednom razlikom - moramo koristiti `await` ključnu riječ jer je metoda sada asinkrona:

```

async def main():
    async with aiohttp.ClientSession() as session:
        response = await session.get("https://catfact.ninja/fact")
        fact_dict = await response.json() # dodajemo await
        print(fact_dict) # ispisuje nasumičnu činjenicu

```

OK, **sada znamo kako poslati jedan zahtjev asinkrorno**. Vidimo da se trajanje nije promijenilo, ali to je zato što smo poslali samo jedan zahtjev.

Idemo poslati 5 zahtjeva na ovaj način, jednostavno ćemo kod iterirati 5 puta.

```

async def main():
    async with aiohttp.ClientSession() as session:
        for i in range(5):
            response = await session.get("https://catfact.ninja/fact")
            fact_dict = await response.json()
            print(fact_dict)

```

Trebali biste uočiti da stvari rade nešto brže nego prije. Jedino što može biti zbunjujuće je `print` naredba koja ispisuje činjenice sekvencijalno, ali.. **ispisuju li se one uopće sekvencijalno? Kako to možemo znati ako su činjenice nasumične?**

U ispis ćemo dodati vrijednost lokalne varijable `i` kako bismo vidjeli redoslijed ispisivanja činjenica:

```

async def main():
    async with aiohttp.ClientSession() as session:
        for i in range(5):
            response = await session.get("https://catfact.ninja/fact")
            fact_dict = await response.json()
            print(f"{i + 1}: {fact_dict['fact']}")

asyncio.run(main())

```

Primjer ispisa:

```
1: Cats' hearing is much more sensitive than humans and dogs.  
2: A cat named Dusty, aged 17, living in Bonham, Texas, USA, gave birth to her 420th  
kitten on June 23, 1952.  
3: British cat owners spend roughly 550 million pounds yearly on cat food.  
4: There are more than 500 million domestic cats in the world, with approximately 40  
recognized breeds.  
5: A cat's appetite is the barometer of its health. Any cat that does not eat or drink for  
more than two days should be taken to a vet.
```

VAŽNO! Ako pokrenete kod više puta, uočit ćete da se činjenice uvijek ispisuju sekvencijalno, odnosno vrijednost `i` je uvijek: `1 2 3 4 5`.

- Iako je kod iznad tehnički napisan "asinkrono", ova petlja se ustvari **ne izvršava konkurentno** budući da koristimo `await` ključnu riječ ispred svakog slanja zahtjeva `session.get`
- Zbog toga ova petlja sama po sebi **neće slati zahtjeve konkurentno**, već će svaki zahtjev čekati na odgovor prethodnog, što znači da se svaka iteracija petlje izvršava sekvencijalno (jedna za drugom).

Vidjeli smo sličan problem u prošloj skripti gdje smo simulirali slanje zahtjeva na poslužitelj koristeći `asyncio.sleep` funkciju te smo rekli da ga možemo riješiti na 3 načina:

- koristeći `asyncio.gather` funkciju
- koristeći `asyncio.Task` objekte
- kombiniranjem ove dvije metode

2.2.3 Konkurentno slanje kroz `asyncio.gather`

U prošloj skripti ste naučili da možemo koristiti `asyncio.gather` funkciju kako bismo **pozvali više korutina konkurentno i zatim pohraniti sve rezultate u jednu listu**.

Sintaksa `asyncio.gather`:

```
asyncio.gather(*korutine) # ako su unutar liste  
asyncio.gather(korutina1, korutina2, korutina3) # ako su pojedinačno
```

Kako ćemo ovdje definirati korutinu za slanje?

Ideja je da iz sljedeće `main` korutine izvučemo kod za slanje zahtjeva u zasebnu korutinu `get_cat_fact`:

```
async def main():  
    async with aiohttp.ClientSession() as session:  
        for i in range(5):  
            response = await session.get("https://catfact.ninja/fact")  
            fact_dict = await response.json()  
            print(f"{i + 1}: {fact_dict['fact']}")  
  
    asyncio.run(main())
```

Glavno pitanje je **gdje ćemo definirati context manager?** U `main` korutini ili u `get_cat_fact` korutini?

► Spoiler alert! Odgovor na pitanje

U vanjsku korutinu proslijeđujemo alias `session`:

```
async def get_cat_fact(session):  
    response = await session.get("https://catfact.ninja/fact")  
    fact_dict = await response.json()  
    return fact_dict
```

U `main` korutini tada moramo definirati otvaranje same sesije:

```
async def main():  
    async with aiohttp.ClientSession() as session:
```

Napokon, možemo koristiti `asyncio.gather` funkciju kako bismo poslali 5 zahtjeva konkurentno. Kako već znamo dobro *comprehension* sintaksu, iskoristit ćemo *list comprehension* za izradu liste korutina:

```

async def get_cat_fact(session):
    response = await session.get("https://catfact.ninja/fact")
    fact_dict = await response.json()
    print(fact_dict['fact'])

async def main():
    async with aiohttp.ClientSession() as session:
        cat_fact_korutine = [get_cat_fact(session) for i in range(5)]

```

- Pozivamo korutine konkurentno koristeći `asyncio.gather` funkciju

```

async def main():
    async with aiohttp.ClientSession() as session:
        cat_fact_korutine = [get_cat_fact(session) for i in range(5)]
    await asyncio.gather(*cat_fact_korutine)

```

Pokrenite kod - vidimo da se činjenice ispisuju dosta brzo.

A kitten will typically weigh about 3 ounces at birth. The typical male housecat will weigh between 7 and 9 pounds, slightly less for female housecats.
Cats see six times better in the dark and at night than humans.
There are approximately 60,000 hairs per square inch on the back of a cat and about 120,000 per square inch on its underside.
Cats bury their feces to cover their trails from predators.
The Egyptian Mau is probably the oldest breed of cat. In fact, the breed is so ancient that its name is the Egyptian word for "cat."

Ako se prisjetite, prosječno vrijeme trajanja programa s 5 činjenica je bilo 1-2 sekunde, ali tada smo imali i ispisivanje: `print("Šaljemo n. zahtjev...")` u svakoj iteraciji.

Dodat ćemo i ovdje `print` naredbu prije ispisa činjenice i izmjeriti vrijeme koristeći `time` modul:

```

async def get_cat_fact(session):
    print("Šaljemo zahtjev za mačji fact")
    response = await session.get("https://catfact.ninja/fact")
    fact_dict = await response.json()
    print(fact_dict['fact'])

```

I bez dodavanja `time` modula, odmah vidimo razliku u terminalu! Prije smo imali **sekvencialno slanje zahtjeva po zahtjev**:

Sekvencialno slanje (requests):

Šaljemo 1. zahtjev...

Cats often overreact to unexpected stimuli because of their extremely sensitive nervous system.

Šaljemo 2. zahtjev...

The normal body temperature of a cat is between 100.5 ° and 102.5 °F. A cat is sick if its temperature goes below 100 ° or above 103 °F.

Šaljemo 3. zahtjev...

If they have ample water, cats can tolerate temperatures up to 133 °F.

Šaljemo 4. zahtjev...

Cats don't have sweat glands over their bodies like humans do. Instead, they sweat only through their paws.

Šaljemo 5. zahtjev...

The first commercially cloned pet was a cat named "Little Nicky." He cost his owner \$50,000, making him one of the most expensive cats ever.

Izvršavanje programa traje 1.26 sekundi.

Sada vidimo da se svi zahtjevi prvo pošalju **konkurentno**, a zatim ispisuju sve činjenice. **Ne čekamo više na odgovor kroz svaku iteraciju petlje**.

Konkurentno slanje (*aiohttp*):

Šaljemo zahtjev za mačji fact

Lions are the only cats that live in groups, called prides. Every female within the pride is usually related.

A happy cat holds her tail high and steady.

The average cat food meal is the equivalent to about five mice.

The Egyptian Mau is probably the oldest breed of cat. In fact, the breed is so ancient that its name is the Egyptian word for "cat."

A cat's nose pad is ridged with a unique pattern, just like the fingerprint of a human.

Zanima nas još i vrijeme izvođenja programa.

Započeti ćemo mjeriti kad se pozove `main` korutina, a završiti na kraju `main` korutine.

```
async def main():
    start = time.time()
    async with aiohttp.ClientSession() as session:
        cat_fact_korutine = [get_cat_fact(session) for i in range(5)]
        await asyncio.gather(*cat_fact_korutine)
    end = time.time()
    print(f"\nIzvršavanje programa traje {end - start:.2f} sekundi.")

asyncio.run(main())
```

Primjer ispisa:

Šaljemo zahtjev za mačji fact
Šaljemo zahtjev za mačji fact

Cats have "nine lives" thanks to a flexible spine and powerful leg and back muscles
Cats' eyes shine in the dark because of the tapetum, a reflective layer in the eye, which acts like a mirror.

The oldest cat on record was Crème Puff from Austin, Texas, who lived from 1967 to August 6, 2005, three days after her 38th birthday. A cat typically can live up to 20 years, which is equivalent to about 96 human years.

When a cat rubs up against you, the cat is marking you with its scent claiming ownership.

Cats see six times better in the dark and at night than humans.

Izvršavanje programa traje 0.27 sekundi.

Vidimo da se vrijeme izvršavanja programa na ovom jednostavnom primjeru slanja 5 zahtjeva **smanjilo s ~1.26 sekundi na ~0.27 sekundi.**

Razliku možemo izraziti i u postocima:

$$\frac{sekvenzialnoVrijeme - konkurentnoVrijeme}{sekvenzialnoVrijeme} \times 100 \quad (1)$$

odnosno:

$$\frac{1.26 - 0.27}{1.26} \times 100 \approx 78.57\% \quad (2)$$

Dakle, **konkurentni kod se izvršio otprilike 78.57% brže od sekvencijalnog!**

Ako podijelimo staro vrijeme izvršavanja s novim, vidimo da je **konkurentni kod gotovo 5 puta brži od sekvencijalnog.**

$$\frac{sekvenzialnoVrijeme}{konkurentnoVrijeme} \quad (3)$$

$$\frac{1.26}{0.27} = 4.67 \quad (4)$$

Pokušajmo i s 15 zahtjeva:

```
async def main():
    start = time.time()
    async with aiohttp.ClientSession() as session:
        cat_fact_korutine = [get_cat_fact(session) for i in range(15)]
        await asyncio.gather(*cat_fact_korutine)
    end = time.time()
    print(f"\nIzvršavanje programa traje {end - start:.2f} sekundi.")
```

Primjer ispisa:

```
Šaljemo zahtjev za mačji fact
Cat families usually play best in even numbers. Cats and kittens should be acquired in pairs whenever possible.
Cats are subject to gum disease and to dental caries. They should have their teeth cleaned by the vet or the cat dentist once a year.
The biggest wildcat today is the Siberian Tiger. It can be more than 12 feet (3.6 m) long (about the size of a small car) and weigh up to 700 pounds (317 kg).
A group of cats is called a clowder.
The heaviest cat on record is Himmy, a Tabby from Queensland, Australia. He weighed nearly 47 pounds (21 kg). He died at the age of 10.
A cat can jump up to five times its own height in a single bound.
A commemorative tower was built in Scotland for a cat named Towser, who caught nearly 30,000 mice in her lifetime.
Purring does not always indicate that a cat is happy and healthy - some cats will purr loudly when they are terrified or in pain.
Long, muscular hind legs enable snow leopards to leap seven times their own body length in a single bound.
The most traveled cat is Hamlet, who escaped from his carrier while on a flight. He hid for seven weeks behind a panel on the airplane. By the time he was discovered, he had traveled nearly 373,000 miles (600,000 km).
Cats and kittens should be acquired in pairs whenever possible as cat families interact best in pairs.
```

The earliest ancestor of the modern cat lived about 30 million years ago. Scientists called it the Proailurus, which means "first cat" in Greek. The group of animals that pet cats belong to emerged around 12 million years ago.

There are up to 60 million feral cats in the United States alone.

The strongest climber among the big cats, a leopard can carry prey twice its weight up a tree.

The name "jaguar" comes from a Native American word meaning "he who kills with one leap".

Izvršavanje programa traje 0.61 sekundi.

Vidimo da se vrijeme izvršavanja programa s 15 zahtjeva **smanjilo s 3-4 sekunde na 0.61 sekundi**.

Ovdje nam je također program gotovo 5 puta brži, odnosno poboljšanje je ~80%.

2.2.4 Konkurentno slanje kroz `asyncio.Task`

Naučili smo kako koristiti `asyncio.gather` funkciju za konkurentno izvođenje korutina. Međutim, u prošloj skripti smo rekli da možemo definirati i tzv. **Taskove** koji predstavljaju izvršenje korutina unutar `asyncio` petlje.

Rekli smo da `Task` objekti, (možemo ih zvati i *Taskovi*) omogućuju bolju kontrolu nad izvršavanjem korutina jer možemo pratiti njihov status, upravljati njima pojedinačno, i eventualno čekati da završe pomoći `await` ključne riječi.

Taskove definiramo koristeći `asyncio.create_task` funkciju koja prima korutinu kao argument.

Sintaksa:

```
task = asyncio.create_task(korutina)
```

U našem primjeru dohvaćanja činjenica o mačkama, korutine su `get_cat_fact`. Možemo ih jednostavno pohraniti u listu i zatim izraditi `Task` objekte za svaku, koristeći *list comprehension*.

Nakon toga ćemo ih pozvati koristeći `await` ključnu riječ jednostavnim iteriranjem kroz listu `Task` objekata.

```
async def main():
    start = time.time()
    async with aiohttp.ClientSession() as session:
        cat_fact_tasks = [asyncio.create_task(get_cat_fact(session)) for _ in range(5)] # pohranjujemo Taskove u listu
        for task in cat_fact_tasks: # ovaj kod izvršava se konkurentno jer smo koristili Taskove
            await task # pozivamo svaki Task
    end = time.time()
    print(f"\nIzvršavanje programa traje {end - start:.2f} sekundi.")
```

Rezultat je identičan kao i kod `asyncio.gather` funkcije:

Šaljemo zahtjev za mačji fact

70% of your cat's life is spent asleep.

Cats eat grass to aid their digestion and to help them get rid of any fur in their stomachs.

In 1987 cats overtook dogs as the number one pet in America.

In ancient Egypt, when a family cat died, all family members would shave their eyebrows as a sign of mourning.

A cat can't climb head first down a tree because every claw on a cat's paw points the same way. To get down from a tree, a cat must back down.

Izvršavanje programa traje 0.28 sekundi.

Ako izvrstimo kod više puta, vidjet ćete da je rezultat izvođenja u pravilu identičan (~0,27 sekundi) kao što je to bio slučaj s `asyncio.gather` funkcijom.

Rekli smo da je moguće i kombinirati ova dva pristupa, odnosno koristiti `asyncio.gather` funkciju za konkurentno izvođenje *Taskova*:

```
async def main():
    start = time.time()
    async with aiohttp.ClientSession() as session:
        cat_fact_tasks = [asyncio.create_task(get_cat_fact(session)) for _ in range(5)] # pohranjujemo Taskove u listu
        await asyncio.gather(*cat_fact_tasks) # pozivamo Taskove konkurentno
    end = time.time()
    print(f"\nIzvršavanje programa traje {end - start:.2f} sekundi.")
```

Možemo maknuti `print` naredbe unutar korutine `get_cat_fact` te vratiti samo činjenicu kao rezultat te korutine:

```
async def get_cat_fact(session):
    response = await session.get("https://catfact.ninja/fact")
    fact_dict = await response.json()
    return fact_dict['fact']

async def main():
    start = time.time()
    async with aiohttp.ClientSession() as session:
        cat_fact_tasks = [asyncio.create_task(get_cat_fact(session)) for _ in range(5)] # pohranjujemo Taskove u listu
        actual_cat_facts = await asyncio.gather(*cat_fact_tasks) # pohranit ćemo rezultate u listu
    end = time.time()
    print(actual_cat_facts)
    print(f"\nIzvršavanje programa traje {end - start:.2f} sekundi.")

asyncio.run(main())
```

Rezultat je lista činjenica:

```
[ 'The Maine Coone is the only native American long haired breed.', 'The Amur leopard is one of the most endangered animals in the world.', "A cat's normal pulse is 140–240 beats per minute, with an average of 195.", 'The cat has 500 skeletal muscles (humans have 650).', 'A happy cat holds her tail high and steady.' ]
```

```
Izvršavanje programa traje 0.27 sekundi.
```

U slučaju da nam ispisi i vrijeme izvođenja nisu dovoljan dokaz da su zahtjevi uistinu poslani konkurentno, možemo još provjeriti **redoslijed ispisivanja činjenica** koji nam je, ako se prisjetite, kod sekvencijalnog slanja uvijek bio isti: 1 2 3 4 5.

Ovdje to možemo testirati na način da ćemo jednostavno proslijediti `i` lokalnu varijablu iz petlje u korutinu `get_cat_fact`:

```
async def get_cat_fact(session, i):
    response = await session.get("https://catfact.ninja/fact")
    fact_dict = await response.json()
    print(f"{i + 1}: {fact_dict['fact']}") # dodajemo ispis u formatu: "redniBroj:
    činjenica"
    return fact_dict['fact']

async def main():
    start = time.time()
    async with aiohttp.ClientSession() as session:
        cat_fact_tasks = [asyncio.create_task(get_cat_fact(session, i)) for i in range(5)] # u
        korutinu proslijedujemo "i"
        actual_cat_facts = await asyncio.gather(*cat_fact_tasks)
    end = time.time()
    print(f"\nIzvršavanje programa traje {end - start:.2f} sekundi.")
asyncio.run(main())
```

Primjer ispisa 1:

```
2: It is estimated that cats can make over 60 different sounds.
1: According to a Gallup poll, most American pet owners obtain their cats by adopting
strays.
5: Cats are the world's most popular pets, outnumbering dogs by as many as three to one
3: The oldest cat to give birth was Kitty who, at the age of 30, gave birth to two
kittens. During her life, she gave birth to 218 kittens.
4: Cats can jump up to 7 times their tail length.
```

Primjer ispisa 2:

```
1: In Japan, cats are thought to have the power to turn into super spirits when they die.
This may be because according to the Buddhist religion, the body of the cat is the
temporary resting place of very spiritual people.i
4: A cat sees about 6 times better than a human at night, and needs 1/6 the amount of light
that a human does - it has a layer of extra reflecting cells which absorb light.
3: Cats lived with soldiers in trenches, where they killed mice during World War I.
5: In relation to their body size, cats have the largest eyes of any mammal.
2: Female felines are \superfecund
```

Primjer ispisa 3:

- 4: Mountain lions are strong jumpers, thanks to muscular hind legs that are longer than their front legs.
- 2: Cats' hearing stops at 65 khz (kilohertz); humans' hearing stops at 20 khz.
- 3: Retractable claws are a physical phenomenon that sets cats apart from the rest of the animal kingdom. In the cat family, only cheetahs cannot retract their claws.
- 5: A cat uses its whiskers for measuring distances. The whiskers of a cat are capable of registering very small changes in air pressure.
- 1: Tylenol and chocolate are both poisionous to cats.

Ako se prisjetimo ilustracije konkurentnog izvođenja na samom početku skripte `RS3`, da se zaključiti zašto su rezultati ovakvi.

Naime, **svaki zahtjev se šalje u isto vrijeme raspodjelom računalnih resursa unutar event loopa, a odgovori se vraćaju u različito vrijeme**. Zbog toga je redoslijed ispisivanja činjenica **nasumičan**. Pitanje je, kako upravljati ovakvim nasumičnim ponašanjem? Za sada ćemo ostaviti ovu temu otvorenom.

Sada definitivno možemo reći da je kod koji smo definirali **konkurentan** te možemo uočiti **jasna poboljšanja u brzini izvođenja programa** 

3. Zadaci za vježbu - Slanje konkurentnih HTTP zahtjeva

1. **Definirajte korutinu** `fetch_users` koja će slati GET zahtjev na [JSONPlaceholder API](https://jsonplaceholder.typicode.com/users) na URL-u:

`https://jsonplaceholder.typicode.com/users`. Morate simulirati slanje 5 zahtjeva konkurentno unutar `main` korutine. Unutar `main` korutine izmjerite vrijeme izvođenja programa, a rezultate pohranite u listu odjedanput koristeći `asyncio.gather` funkciju. Nakon toga koristeći `map` funkcije ili `list comprehension` izdvojite u zasebne 3 liste: samo imena korisnika, samo email adrese korisnika i samo username korisnika. Na kraju `main` korutine ispišite sve 3 liste i vrijeme izvođenja programa.

2. **Definirajte dvije korutine**, od kojih će jedna služiti za dohvaćanje činjenica o mačkama koristeći `get_cat_fact` korutinu koja šalje GET zahtjev na URL: `https://catfact.ninja/fact`. Izradite 20 `Task` objekata za dohvaćanje činjenica o mačkama te ih pozovite unutar `main` korutine i rezultate pohranite odjednom koristeći `asyncio.gather` funkciju. Druga korutina `filter_cat_facts` ne šalje HTTP zahtjeve, već mora primiti gotovu listu činjenica o mačkama i vratiti novu listu koja sadrži samo one činjenice koje sadrže riječ "cat" ili "cats" (neovisno o velikim/malim slovima).

Primjer konačnog ispisa:

Filtrirane činjenice o mačkama:

- A 2007 Gallup poll revealed that both men and women were equally likely to own a cat.
- The first cat in space was a French cat named Felicette (a.k.a. "Astrocat") In 1963, France blasted the cat into outer space. Electrodes implanted in her brains sent neurological signals back to Earth. She survived the trip.
- The lightest cat on record is a blue point Himalayan called Tinker Toy, who weighed 1 pound, 6 ounces (616 g). Tinker Toy was 2.75 inches (7 cm) tall and 7.5 inches (19 cm) long.
- The first commercially cloned pet was a cat named "Little Nicky." He cost his owner \$50,000, making him one of the most expensive cats ever.
- In the 1750s, Europeans introduced cats into the Americas to control pests.
- A group of cats is called a clowder.

3. Definirajte korutinu `get_dog_fact` koja dohvaća činjenice o psima sa [DOG API](#).

Korutina `get_dog_fact` neka dohvaća činjenicu o psima na URL-u: <https://dogapi.dog/api/v2/facts>. Nakon toga, definirajte korutinu `get_cat_fact` koja dohvaća činjenicu o mačkama slanjem zahtjeva na URL: <https://catfact.ninja/fact>.

Istovremeno pohranite rezultate izvršavanja ovih *Taskova* koristeći `asyncio.gather(*dog_facts_tasks, *cat_facts_tasks)` funkciju u listu `dog_cat_facts`, a zatim ih koristeći *list slicing* odvojite u dvije liste obzirom da znate da je prvih 5 činjenica o psima, a drugih 5 o mačkama.

Na kraju, definirajte i **treću korutinu `mix_facts`** koja prima liste `dog_facts` i `cat_facts` i vraća **novu listu** koja za vrijednost indeksa `[i]` sadrži činjenicu o psima ako je duljina činjenice o psima veća od duljine činjenice o mačkama na istom indeksu, inače vraća činjenicu o mački. Na kraju ispišite rezultate filtriranog niza činjenica. Liste možete paralelno iterirati koristeći `zip` funkciju, npr. `for dog_fact, cat_fact in zip(dog_facts, cat_facts):`.

Primjer konačnog ispisa:

```
Mixane činjenice o psima i mačkama:
```

```
If they have ample water, cats can tolerate temperatures up to 133 °F.  
Dogs with little human contact in the first three months typically don't make good pets.  
The most popular dog breed in Canada, U.S., and Great Britain is the Labrador retriever.  
An estimated 1,000,000 dogs in the U.S. have been named as the primary beneficiaries in  
their owner's will.  
When a cat rubs up against you, the cat is marking you with its scent claiming  
ownership.
```

Raspodijeljeni sustavi (RS)

Nositelj: doc. dr. sc. Nikola Tanković

Asistent: Luka Blašković, mag. inf.

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



Fakultet informatike u Puli

(6) Razvojni okvir FastAPI

#6

RS

FastAPI je moderni web okvir za izgradnju API-ja koji se temelji na modernom Pythonu i tipovima (*type hints*). Radi se o relativnoj novom razvojnem okviru koji je prvi put objavljen 2018. godine te je od onda u aktivnom razvoju, a bilježi sve veću popularnost među Python programerima. Glavne funkcionalnosti FastAPI-ja uključuju automatsku generaciju dokumentacije, odličnu brzinu izvođenja koja je mjerljiva sa brzinom izvođenja razvojnih okvira temeljenih na Node-u i Go-u, kao i mogućnost korištenja tipova podatka za definiranje ulaznih i izlaznih očekivanih vrijednosti, validaciju podataka temeljenu na Pydantic modelima, automatsko generiranje dokumentacije itd. Konkretno u sklopu ovog kolegija, naučit ćemo kako razvijati s FastAPI-jem u svrhu implementacije robusnih mikroservisa koji se koriste u raspodijeljenim sustavima.

 **Posljednje ažurirano: 16.1.2025.**

- manji ispravci

Sadržaj

- [Raspodijeljeni sustavi \(RS\)](#)
- [\(6\) Razvojni okvir FastAPI](#)
 - [Sadržaj](#)
- [1. Uvod u FastAPI](#)
 - [1.1 Instalacija](#)
 - [1.2 Definiranje ruta](#)
 - [1.2.1 Parametri ruta \(eng. route parameters\)](#)
 - [Primitivni tipovi koji podržavaju type hinting](#)
 - [Kolekcije koje podržavaju type hinting](#)
 - [Tijelo zahtjeva \(eng. request body\)](#)
 - [1.2.2 Query parametri \(eng. query parameters\)](#)

- [1.2.3 Kako razlikovati route i query parametre te tijelo zahtjeva?](#)
- [2. Pydantic](#)
 - [2.1 Input/Output modeli](#)
 - [2.2 Zadaci za vježbu - Osnove definicije ruta i Pydantic modela](#)
 - [2.3 Složeniji Pydantic modeli](#)
 - [2.3.1 Tablica osnovnih tipova](#)
 - [2.3.2 Tablica kolekcija](#)
 - [2.3.3 Primjeri složenijih Pydantic modela](#)
 - [Zadane vrijednosti \(eng. default values\)](#)
 - [Rječnici, n-torke i skupovi](#)
 - [Složeni tipovi iz biblioteke typing](#)
 - [2.4 Nasljeđivanje Pydantic modela](#)
 - [2.5 Zadaci za vježbu: Definicija složenijih Pydantic modela](#)
 - [2.6 Field polje Pydantic modela](#)
- [3. Obrada grešaka \(eng. Error Handling\)](#)
 - [3.1 Validacija parametara rute i query parametra](#)
 - [3.2 Zadaci za vježbu: Obrada grešaka](#)
- [4. Strukturiranje poslužitelja i organizacija koda](#)
 - [4.1 Dependency Injection \(DI\)](#)
 - [4.2 API Router](#)
 - [4.3 Zadatak za vježbu: Razvoj mikroservisa za dohvaćanje podataka o filmovima](#)

1. Uvod u FastAPI

FastAPI je moderni web okvir za izgradu brzih i učinkovitih API-ja. Temelji se na Python anotacije zvane [type hints](#) kako bi omogućio lakšu validaciju dolaznih HTTP zahtjeva i odgovora što smanjuje greške tijekom razvoja i egzekucije programa te povećava sigurnost i olakšava održavanje koda. Jedna od ključnih značajki FastAPI-ja je i **automatska generacija dokumentacije** putem alata Swagger UI, ali i mogućnost korištenja Pydantic modela za validaciju složenijih podatkovnih struktura.

Po svom dizajnu, FastAPI je *non-blocking*, što znači da je sposoban obrađivati više zahtjeva istovremeno (konkurentno) bez blokiranja izvođenja glavne dretve. Kao temelj koristi [Starlette](#) web okvir koji je lagan i brz asinkroni web okvir. Pozadinska tehnologija koja omogućuje ovakvo ponašanje je [ASGI](#), odnosno *Asynchronous Server Gateway Interface*. Radi se o relativnoj novoj konvenciji za razvoj web poslužitelja u Pythonu koja je zamijenila stariju WSGI konvenciju. Glavna mana je što **WSGI nije bio dizajniran za asinkrono izvođenje**.

Primjeri razvojnih okvira koji su temeljeni i prvenstveno razvijani na WSGI konvenciji uključuju [Django](#) i [Flask](#) (iako se danas mogu učiniti asinkronim uz određene ekstenzije).

Projekt iz kolegija Raspodijeljeni sustavi moguće je napraviti koristeći FastAPI kao temeljni web okvir za izgradnju mikroservisa. U nastavku slijedi upute za instalaciju FastAPI-ja te primjere kako ga kvalitetno koristiti u praksi.



FastAPI logotip

1.1 Instalacija

FastAPI je odlično dokumentiran te postoji mnoštvo resursa na internetu koji vam mogu pomoći u njegovom učenju i razvoju. Preporučuje se korištenje FastAPI dokumentacije kao primarnog izvora informacija.

Dostupno na: <https://fastapi.tiangolo.com/learn/>

Za početak, potrebno je pripremiti **virtualno okruženje**. Mi ćemo ovdje koristiti `conda` modul:

```
conda create --name rs_fastapi python=3.13
conda activate rs_fastapi
```

Isto možete napraviti i kroz `Anaconda Navigator` grafičko sučelje.

Nakon što smo aktivirali virtualno okruženje, instaliramo FastAPI:

```
pip install "fastapi[standard]"
```

Napravite novi direktorij, npr. `rs_fastapi` i u njemu izradite datoteku `main.py`:

Uključujemo FastAPI modul i definiramo instancu aplikacije:

```
from fastapi import FastAPI

app = FastAPI()
```

FastAPI koristi [Uvicorn](#) kao ASGI server. **Uvicorn** podržava HTTP/1.1 standard te WebSockets protokole. Dolazi instaliran s FastAPI-jem (ako ste ga instalirali sa `[standard]` zastavicom kao što je prikazano iznad). U tom slučaju, možete pokrenuti FastAPI poslužitelj koristeći sljedeću naredbu:

```
fastapi dev main.py
```

Naredba `fastapi dev` čita datoteku `main.py` i pokreće FastAPI poslužitelj koristeći *uvicorn*. U pravilu, FastAPI poslužitelj će biti pokrenut portu `8000`, ako je slobodan.

FastAPI servis je moguće pokrenuti i direktnim pozivanjem `uvicorn` modula:

```
uvicorn main:app --reload
```

gdje je:

- `main` ime datoteke bez ekstenzije
- `app` instanca FastAPI aplikacije
- `--reload` zastavica označava da se poslužitelj ponovno pokrene nakon svake promjene u kodu (*hot reload*)

Ako želimo definirati port na kojem će se poslužitelj pokrenuti, možemo to učiniti dodavanjem zastavice `--port`:

```
uvicorn main:app --reload --port 3000
```

Možete otvoriti web preglednik i posjetiti <http://localhost:8000> odnosno <http://localhost:8000/docs> kako biste vidjeli **generiranu dokumentaciju** ([Swagger UI](#)).

- kao alternativa, možete pristupiti i [ReDoc](#) dokumentaciji na <http://localhost:8000/redoc>.

Swagger UI i **Redoc** su alati za generiranje dokumentacije iz [OpenAPI specifikacije](#). FastAPI generira OpenAPI specifikaciju automatski na temelju definiranih ruta i Pydantic modela, a Swagger UI i ReDoc su alati koji tu specifikaciju prikazuju na korisnički prihvatljiv način - **u obliku web stranice s interaktivnim elementima**.

Ako pokušate otvoriti dokumentaciju, vidjet ćete da trenutno nema definiranih ruta.

FastAPI 0.1.0 OAS 3.1
[/openapi.json](#)

No operations defined in spec!

1.2 Definiranje ruta

FastAPI koristi **dekoratore** za definiranje ruta. U Pythonu, dekoratori (eng. *decorators*) su **funkcije ili klase koje proširuju funkcionalnost druge funkcije ili klase** bez promjene njene implementacije. Dekoratori omogućuju dodavanje funkcionalnosti na postojeće funkcije na čitljiviji način.

U kontekstu funkcionskog programiranja, **dekoratori su funkcije višeg reda** (eng. *higher-order functions*) koje rade sljedeće:

1. Primaju funkciju (ili klasu) kao argument
2. Dodaju neku funkcionalnost (ponašanje) toj funkciji
3. Vraćaju "modificiranu" funkciju (ili klasu)

Dekoratori se koriste prije definiranja funkcije kojoj želimo dodati funkcionalnost, **oznakom @ prije naziva dekoratora**.

Konkretno, FastAPI koristi dekoratore za definiranje ruta. Na primjer, sljedeći kod definira jednostavnu GET rutu koja vraća JSON odgovor s porukom "Hello, world!"

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/") # dekorator za GET metodu na korijenskoj ruti
def read_root(): # funkcija koja se poziva kada se posjeti korijenska ruta
    return {"message": "Hello, world!"} # vraća JSON odgovor u tijelu HTTP odgovora
```

Ekvivalentan kod koji smo pisali prilikom definiranja `aiohttp` rute izgledao bi ovako:

```
from aiohttp import web

def handle(request):
    return web.json_response({"message": "Hello, world!"})

app = web.Application()
app.router.add_get('/', handle)
```

Dakle, FastAPI koristi dekoratore za definiciju:

1. **Metode** HTTP za rute (`GET`, `POST`, `PUT`, `PATCH`, `DELETE`, itd.)
2. **Putanje** ruta (npr. `/`, `/items/{item_id}`, `/users/{user_id}/items/{item_id}`, itd.)

Handler funkciju koja se mora izvršiti pišemo neposredno ispod dekoratora.

U FastAPI-ju možemo koristiti sljedeće dekoratore za definiranje ruta:

- `@app.get(path)` - definira GET rutu
- `@app.post(path)` - definira POST rutu
- `@app.put(path)` - definira PUT rutu
- `@app.delete(path)` - definira DELETE rutu
- `@app.patch(path)` - definira PATCH rutu
- `@app.options(path)` - definira OPTIONS rutu
- `@app.head(path)` - definira HEAD rutu

1.2.1 Parametri ruta (eng. route parameters)

Parametre ruta definiramo na isti način kao i u `aiohttp` biblioteci, koristeći vitičaste zagrade `{}`. Na primjer, sljedeći kod definira rutu koja očekuje `proizvod_id` kao parametar:

```
@app.get("/proizvodi/{proizvod_id}")
def get_proizvod(proizvod_id):
    return {"proizvod_id": proizvod_id}
```

HTTP zahtjev možete poslati koristeći bilo koji alat, međutim kad već radimo s FastAPI-jem, **dobra je praksa koristiti ugrađenu interaktivnu dokumentaciju** koju generira **Swagger** ili **ReDoc**.

- otvorite <http://localhost:8000/docs> u web pregledniku kako biste pristupili generiranoj dokumentaciji.

Ako je kod ispravan, trebali biste vidjeti definiranu rutu u dokumentaciji: `GET /proizvodi/{proizvod_id}`
`Get Proizvod`

- gdje je `Get Proizvod` ustvari **naziv handler funkcije** koju smo definirali, a ruta `GET /proizvodi/{proizvod_id}` je **definirana dekoratorom**.

Odaberite rutu i kliknite na `Try it out` kako biste mogli poslati HTTP zahtjev.

- u polje `proizvod_id` unesite neku vrijednost i kliknite na `Execute`.
- ukoliko je sve ispravno, trebali biste vidjeti HTTP odgovor s definiranom vrijednosti `proizvod_id`.

default

GET /proizvodi/{proizvod_id} Get Proizvod

Parameters

Name	Description
proizvod_id * required (path)	proizvod_id

Responses

Code	Description	Links
200	Successful Response	No links
422	Validation Error	No links

Generirana FastAPI Swagger dokumentacija, dostupna na <http://localhost:8000/docs>

Vidimo da generirana dokumentacija nudi **pregled svih podataka koje očekuje i vraća naša ruta**, odnosno sve podatke o HTTP zahtjevu koji se očekuje te o odgovoru koji će se vratiti.

Server response

Code	Details
200	<p>Response body</p> <pre>{ "proizvod_id": "3" }</pre> <p>Response headers</p> <pre>content-length: 19 content-type: application/json date: Tue, 07 Jan 2025 21:19:00 GMT server: uvicorn</pre>

Responses

Code	Description	Links
200	Successful Response	No links
422	Validation Error	No links

U interaktivnoj dokumentaciji možemo vidjeti detaljan pregled HTTP odgovora koji vraća FastAPI poslužitelj

U Swagger interaktivnoj dokumentaciji možemo vidjeti sljedeće elemente HTTP odgovora:

- **Response body:** JSON odgovor koji je vraćen, u ovom slučaju: `{"proizvod_id": "3"}`
- **Response code:** HTTP statusni kod koji je vraćen, u ovom slučaju: `200 OK`
- **Response headers:** zaglavlja HTTP odgovora

Uz to možemo vidjeti i primjere ispravnog i neispravnog odgovora te definirane **Pydantic podatkovne modele** (`schemas`), ako postoje. Više o tome u nastavku.

Primijetite sljedeće, FastAPI je automatski **parsirao parametar** `proizvod_id` iz URL-a i proslijedio ga kao argument funkciji `get_proizvod`.

```
@app.get("/proizvodi/{proizvod_id}")
def get_proizvod(proizvod_id):
    return {"proizvod_id": proizvod_id}
```

Ako pogledate odgovor, vidjet ćete da je vrijednost `proizvod_id` ustvari: `string: "proizvod_id": "3"`.

- **FastAPI automatski parsira parametre ruta u odgovarajući tip podatka**, ovisno o tipu koji je *hintan* u Python funkciji. Kako mi nismo definirali ništa, prepostavlja se da je tip `str`.

Ako bi htjeli naglasiti da je očekivani parametar `proizvod_id` tipa `int`, možemo to napraviti koristeći **Python type hinting**.

- to radimo na način da pišemo **tip podataka odvojen dvotočjem (:) nakon imena parametra**

Sintaksa:

```
@app.get("/ruta/{parametar}")
def funkcija(parametar: tip): # type hinting
    # tijelo funkcije
```

Primjer: Želimo/hintamo da je `proizvod_id` tipa `int`:

```
@app.get("/proizvodi/{proizvod_id}")
def get_proizvod(proizvod_id: int): # "hintamo" da je proizvod_id tipa int
    return {"proizvod_id": proizvod_id}
```

Pošaljite opet zahtjev u dokumentaciji i vidjet ćete da je sada vrijednost `proizvod_id` tipa `int`.

type hinting u FastAPI-ju **nije samo dekorativna značajka**, već ima i praktičnu svrhu na način da održuje **automatsko parsiranje i validaciju podataka**.

Međutim, ako se vratimo na dokumentaciju i pošaljemo sljedeći zahtjev: `GET /proizvodi/Marko`. Vidjet ćemo da poslužitelj baca grešku jer je očekivani tip podataka `int`, a mi smo poslali `str`.

The screenshot shows a FastAPI application's error response. A curl command is provided to reproduce the error:

```
curl -X 'GET' \
'http://localhost:8000/proizvodi/Marko' \
-H 'accept: application/json'
```

The Request URL is <http://localhost:8000/proizvodi/Marko>. The Server response details a 422 Error: Unprocessable Entity. The Response body is a JSON object:

```
{
  "detail": [
    {
      "type": "int_parsing",
      "loc": [
        "path",
        "proizvod_id"
      ],
      "msg": "Input should be a valid integer, unable to parse string as an integer",
      "input": "Marko"
    }
  ]
}
```

The Response headers are:

```
content-length: 158
content-type: application/json
date: Tue, 07 Jan 2025 21:39:54 GMT
server: uvicorn
```

FastAPI automatski baca grešku ako se očekivani tip podataka ne podudara s onim što je poslano

Dobili smo detaljnu grešku, sa statusnim kodom `422 Unprocessable Entity` i složenim JSON objektom HTTP odgovora koji opisuje grešku:

```
{
  "detail": [
    {
      "type": "int_parsing",
      "loc": [
        "path",
        "proizvod_id"
      ],
      "msg": "Input should be a valid integer, unable to parse string as an integer",
      "input": "Marko"
    }
  ]
}
```

FastAPI poslužitelj automatski obrađuje ovu grešku za nas (**ne moramo ih obrađivati ručno kao do sada**) i sadrži sve potrebne informacije o grešci, uključujući tip greške, lokaciju greške, poruku greške i ulazne podatke koji su uzrokovali grešku.

Primitivni tipovi koji podržavaju type hinting

- `str` - string
- `int` - cijeli broj
- `float` - decimalni broj
- `bool` - logička vrijednost

- `bytes` - niz bajtova
- `None` - nema vrijednosti

Kolekcije koje podržavaju type hinting

- `list` - lista
- `tuple` - uređeni par
- `set` - skup
- `frozenset` - nepromjenjivi skup
- `dict` - rječnik

Više o tipovima podataka u poglavlju [2. Pydantic](#).

Primjer: Nadogradit ćemo postojeću aplikaciju tako da pronađe odgovarajući proizvod u *in-memory* listi proizvoda te omogućiti korisniku da ga **dohvati prema imenu**. Također, dodat ćemo rutu za **dodavanje novog proizvoda** u listu.

Definirajmo nekoliko proizvoda u listi. Svaki proizvod sadrži ključeve `id`, `naziv`, `boja` i `cijena`:

```
proizvodi = [
    {"id": 1, "naziv": "majica", "boja": "plava", "cijena": 50},
    {"id": 2, "naziv": "hlače", "boja": "crna", "cijena": 100},
    {"id": 3, "naziv": "tenisice", "boja": "bijela", "cijena": 150},
    {"id": 4, "naziv": "kapa", "boja": "smeđa", "cijena": 20}
]
```

1. Definirat ćemo prvo rutu koja će omogućiti dohvaćanje svih proizvoda:

```
@app.get("/proizvodi")
def get_proizvodi(): # funkcija ne prima argumente jer nemamo parametre
    return proizvodi
```

2. Zatim ćemo definirati rutu koja će omogućiti dohvaćanje proizvoda prema imenu, dakle:

`/proizvodi/{naziv}`:

Možemo koristiti ugrađenu Python funkciju `next()` koja će nam omogućiti pronađak **prvog proizvoda koji zadovoljava uvjet**. Sintaksa nalikuje na *list comprehension*, ali s dodatnim parametrom `default` koji se vraća ako se ne pronađe nijedan element koji zadovoljava uvjet.

- nakon pronađaska prvog elementa koji zadovoljava uvjet, `next()` vraća taj element i **iteriranje se zaustavlja**

Sintaksa:

```
next((expression for iterator in iterable if condition), default)
```

- `expression` - izraz koji se evaluira

- `iterator` - iterator koji prolazi kroz elemente
- `iterable` - kolekcija elemenata (lista, rječnik, skup, tuple, itd.)
- `condition` - uvjet koji mora biti zadovoljen
- `default` - vrijednost koja se vraća ako se ne pronađe nijedan element koji zadovoljava uvjet

Definirajmo rutu za dohvaćanje proizvoda prema imenu:

```
@app.get("/proizvodi/{naziv}") # route parametar "naziv"
def get_proizvod_by_name(naziv: str): # očekujemo string kao naziv proizvoda (ako ne
naglasimo se podrazumijeva da je str)
    # pronalazimo proizvod gdje se njegov naziv poklapa s nazivom iz parametra rute "naziv"
    pronadjeni_proizvod = next((proizvod for proizvod in proizvodi if proizvod["naziv"] ==
naziv), None) # None ako se ne pronađe proizvod
    return pronadjeni_proizvod
```

Tijelo zahtjeva (eng. request body)

3. **Dodavanje proizvoda u listu proizvoda** možemo odraditi definicijom POST zahtjeva na `/proizvodi`:

Tijelo HTTP zahtjeva možemo definirati kao argument funkcije te *hintamo* da je tijelo zahtjeva tipa `dict` (rječnik) jer očekujemo JSON objekt.

Ne navodimo tijelo zahtjeva u dekoratoru (kao što je slučaj kod parametara rute), već ga očekujemo kao argument funkcije *hintanjem* `dict` ili Pydantic modela (više u nastavku).

```
@app.post("/proizvodi") # ne definiramo tijelo zahtjeva u dekoratoru
def add_proizvod(proizvod: dict): # očekujemo JSON objekt kao proizvod u tijelu zahtjeva
pa hintamo rječnik (dict)
    proizvod["id"] = len(proizvodi) + 1 # dodajemo novi ID (broj proizvoda + 1)
    proizvodi.append(proizvod) # dodajemo proizvod u listu
    return proizvod
```

Otvorite dokumentaciju, uočit ćete sve tri definirane rute (`GET /proizvodi`, `GET /proizvodi/{naziv}`, `POST /proizvodi`). Isprobajte svaku od definiranih ruta.

FastAPI 0.1.0 OAS 3.1
[/openapi.json](#)

default

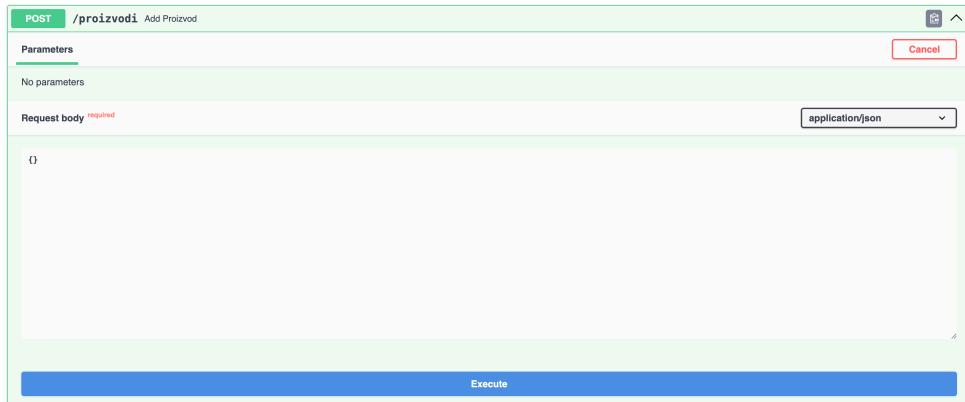
<code>GET</code>	<code>/proizvodi</code>	Get Proizvodi	▼
<code>POST</code>	<code>/proizvodi</code>	Add Proizvod	▼
<code>GET</code>	<code>/proizvodi/{naziv}</code>	Get Proizvod By Name	▼

Schemas

<code>HTTPValidationError</code> > Expand all <code>object</code>
<code>ValidationError</code> > Expand all <code>object</code>

Generirana dokumentacija s tri definirane rute (`GET /proizvodi`, `GET /proizvodi/{naziv}`, `POST /proizvodi`)

Ako otvorite sučelje za rutu `POST /proizvodi`, vidjet ćete da vam se nudi opcija za unos JSON tijela zahtjeva, budući da nismo naveli parametre rute u dekoratoru:



Sučelje za unos tijela zahtjeva u dokumentaciji za rutu `POST /proizvodi`

```
{ "naziv": "šal", "boja": "plava", "cijena": 30 }
```

HTTP Odgovor će biti novi proizvod s automatski dodijeljenim ID-em:

```
{
    "naziv": "šal",
    "boja": "plava",
    "cijena": 30,
    "id": 5 // automatski dodijeljen ID
}
```

1.2.2 Query parametri (eng. query parameters)

Query parametri su parametri koji se šalju u URL-u HTTP zahtjeva, nakon znaka `?`. Na primjer, u URL-u `/proizvodi?boja=plava` query parametar je `boja` s vrijednošću `plava`. Uobičajeno je koristiti query parametre za filtriranje podataka, sortiranje, paginaciju i slične operacije.

Na FastAPI poslužitelju, **query parametre** možemo definirati koristeći Python *type hinting* na način da ih dodamo kao argumente funkcije, **bez dodavanja u URL putanju kroz dekorator**.

- **FastAPI će takve argumente automatski interpretirati kao query parametre.**

Primjer definiranja rute koja očekuje query parametar `boja`:

```
@app.get("/proizvodi") # u FastAPI-ju ne navodimo query parametre u URL putanji
def get_proizvodi_by_query(boja: str): # očekujemo query parametar "boja"
    pronadeni_proizvodi = [proizvod for proizvod in proizvodi if proizvod["boja"] == boja] # koristimo list comprehension, a ne next() jer možemo imati više proizvoda s istom bojom
    return pronadeni_proizvodi
```

Možemo definirati i više query parametara:

```

@app.get("/proizvodi") # u FastAPI-ju ne navodimo query parametre u URL putanji
def get_proizvodi_by_query(boja: str, max_cijena: int): # očekujemo query parametre "boja"
    i "max_cijena"
        # koristimo list comprehension, a ne next() jer možemo imati više proizvoda s istom
        bojom i cijenom manjom ili jednako od max_cijena
        pronadjeni_proizvodi = [proizvod for proizvod in proizvodi if proizvod["boja"] == boja
        and proizvod["cijena"] <= max_cijena]
        return pronadjeni_proizvodi

```

Identični procesi primjenjuju se i za query parametre kao i za route parametre kada koristimo *type hinting*:

- automatsko parsiranje podataka
- automatska validacija podataka
- automatsko generiranje dokumentacije

Query parametrima možemo dodjeljivati i **zadane (defaultne) vrijednosti**:

```

@app.get("/proizvodi") # u FastAPI-ju ne navodimo query parametre u URL putanji
def get_proizvodi_by_query(boja: str = None, max_cijena: int = 100): # očekujemo query
    parametre "boja" i "max_cijena", ali su im zadane vrijednosti None odnosno 100
        pronadjeni_proizvodi = [proizvod for proizvod in proizvodi if (boja is None or
        proizvod["boja"] == boja) and (max_cijena is None or proizvod["cijena"] <= max_cijena)]
        return pronadjeni_proizvodi

```

Svi navedeni query parametri na ovaj način postaju **opcionalni**. Ako ih ne navedemo u URL-u, poslužitelj će ih automatski postaviti na `None`.

Vidimo da se FastAPI ponaša vrlo slično kao i `aiohttp` biblioteka, ali s mnogo više **automatskih značajki** koje olakšavaju razvoj i održavanje koda. Dodatno, tu je dokumentacija koja nam već u ovoj fazi pomaže u razvoju i testiranju API-ja. Konkretno, za primjer rute iznad možemo u dokumentaciji odmah vidjeti:

- koji se query parametri očekuju (`boja`, `max_cijena`)
- koji su tipovi podataka očekivani (`string`, `integer`)
- koje su defaultne vrijednosti (`None`, `100`)

1.2.3 Kako razlikovati route i query parametre te tijelo zahtjeva?

U FastAPI-ju može biti zbumujuće razlikovati route parametre, query parametre i tijelo zahtjeva budući da ne navodimo eksplisitno "što je što" već se oslanjamo na *type hinting*. **Evo kratkog pregleda:**

- **Route parametri - obavezno se navode u URL putanji (dekoratoru)**, npr.

```
@app.get("/proizvodi/{proizvod_id}") .
```

- moraju imati odgovarajući **ekvivalent u deklaraciji funkcije** i to istog naziva, npr. `def get_proizvod(proizvod_id: int): .`
- sada se može poslati sljedeći zahtjev: `GET /proizvodi/3`.
- mogu sadržavati *type hinting*, inače se podrazumijeva `str`.
- FastAPI automatski parsira i validira podatke iz parametra rute.

- **Query parametri - ne navode se u URL putanji (dekoratoru):** `@app.get("/proizvodi")`

- deklariraju se kao argumenti funkcije, npr. `def get_proizvodi_by_query(boja: str): .`
- sada se može poslati sljedeći zahtjev: `GET /proizvodi?boja=plava`.
- query parametri ako su navedeni bez zadanih vrijednosti postaju obavezni.
- Zadane vrijednosti možemo postaviti dodjeljivanjem vrijednosti u deklaraciji funkcije, npr. `def get_proizvodi_by_query(boja: str = "plava") .`
- FastAPI automatski parsira i validira podatke iz query parametara.

- **Tijelo zahtjeva - ne navode se u URL putanji (dekoratoru)**, npr. `@app.post("/proizvodi")`.

- deklariraju se kao argumenti funkcije hintanjem `dict` ili Pydantic modela, npr. `def add_proizvod(proizvod: dict): .`
- FastAPI automatski parsira i validira podatke iz tijela zahtjeva.
- u nastavku ćemo vidjeti kako koristiti Pydantic modele za hintanje tijela zahtjeva.

Moguće je kombinirati sva 3 pristupa.

Primjerice: Recimo da želimo definirati rutu koja će omogućiti ažuriranje podataka o proizvodu iz skladišta gdje su proizvodi podijeljeni u kategorije.

Podaci su definirani na sljedeći način:

- `id_skladiste` - cijeli broj (route parametar)
- `kategorija` - string (query parametar)
- `proizvod` - proizvod koji ažuriramo (tijelo zahtjeva)

Odabrali bi metodu PATCH budući da djelomično ažuriramo resurse (proizvode) u skladištu.

1. Definirat ćemo dekorator za PATCH metodu na `/skladiste`:

```
@app.patch("/skladiste")
```

2. Prva filtracija odnosi se na dohvat određenog skladišta prema `id_skladiste`:

- nadograđujemo dekorator
- dodajemo ekvivalentni argument funkcije

```
@app.patch("/skladiste/{id_skladiste}")
def update_skladiste(id_skladiste: int):
```

3. Druga filtracija odnosi se na dohvat proizvoda u određenoj kategoriji:

- dodajemo query parametar u deklaraciji funkcije, **ali ne u dekoratoru**

```
@app.patch("/skladiste/{id_skladiste}")
def update_skladiste(id_skladiste: int, kategorija: str):
```

4. Možemo postaviti zadanu vrijednost za query parametar:

- npr. `kategorija: str = "gradevinski_materijal"`

```
@app.patch("/skladiste/{id_skladiste}")
def update_skladiste(id_skladiste: int, kategorija: str = "gradevinski_materijal"):
```

5. Na kraju, dodajemo tijelo zahtjeva kao argument funkcije:

- hintmo da je tijelo zahtjeva tipa `dict`
- dodajemo na početak funkcije jer vrijede ista pravila kao i za zadane argumente običnih Python funkcija (zadani argumenti dolaze na kraju)

```
@app.patch("/skladiste/{id_skladiste}")
def update_skladiste(proizvod: dict, id_skladiste: int, kategorija: str =
"gradevinski_materijal"):
```

Provjerimo kako je dokumentirana definirana ruta u FastAPI dokumentaciji.

PATCH /skladiste/{id_skladiste} Update Skladiste

Cancel

Parameters

Name	Description
id_skladiste * required	integer (path)
kategorija	string (query)

Request body required

application/json

```
{}
```

Execute

Responses

The screenshot shows a REST API testing interface. At the top, it specifies a PATCH request to the endpoint /skladiste/{id_skladiste} with the sub-action Update Skladiste. Below this, there's a 'Parameters' section with two entries: 'id_skladiste' (required, integer, path) and 'kategorija' (string, query). Under 'Request body' (which is marked as required), there's a dropdown set to 'application/json' and a text area containing an empty object '{}'. At the bottom, a prominent blue button labeled 'Execute' is available for sending the request.

U nastavku ćemo vidjeti kako validirati tijelo zahtjeva koristeći **Pydantic modele**.

2. Pydantic

Pydantic je najrasprostranjenija Python biblioteka za **validaciju podataka** koja se bazira na *type hintingu* za definiranje očekivanih tipova podataka te automatski vrši validaciju podataka prema tim definicijama. Pydantic je posebno koristan u FastAPI-ju jer se može koristiti za definiranje **modela podataka** koji se koriste za validaciju dolaznih i odlaznih podataka odnosno **tijela HTTP zahtjeva i odgovora**.

Napomena! Kada govorimo o **modelima** u kontekstu FastAPI-ja, mislimo na **Pydantic modele** koji se koriste za definiranje složenijih struktura podataka koje želimo "hintati" u različitim dijelovima aplikacije. Model u ovom kontekstu **ne predstavlja matematički model** koji se odnosi na statističke analize, model strojnog učenja ili sl. već predstavlja složenu strukturu podataka koja se koristi za validaciju, serijalizaciju te deserijalizaciju podataka te osigurava da su podaci u skladu s očekivanim tipovima. U nastavku ove skripte koristit će se termin "model" za danu definiciju.



Dokumentacija dostupna na: <https://docs.pydantic.dev/latest/>

Jedna od glavnih prednosti Pydantic-a je njegovo ponašanje u IDE razvojnim okruženjima kao što su **VS Code** ili **PyCharm**. IDE-ovi koji podržavaju Python *type hinting* automatski će prepoznati Pydantic modele i pružiti korisne informacije o očekivanim tipovima podataka, što olakšava razvoj i održavanje koda.

Pydantic klase definiramo nasljeđivanjem `pydantic.BaseModel` klase.

Uobičajeno je Pydantic klase odvojiti o `main.py` datoteke kako bi kod bio bolje organiziran te kako bi klase mogli koristiti u više datoteka.

- **Pydantic modele ćemo definirati u zasebnoj datoteci**, npr. `models.py` ili `schemas.py`.

Napravite novu datoteku `models.py`:

Definirajte klasu `Proizvod` koja će predstavljati model podataka za proizvod koji smo prije *hintali* kao rječnik.

- Prvo uključujemo `BaseModel` **kojeg nasljeđuju sve Pydantic klase**:

```
# models.py

from pydantic import BaseModel
```

Pišemo definiciju klase koja nasljeđuje `BaseModel`:

```
# models.py

class Proizvod(BaseModel):
    pass
```

Unutar definicije klase navodimo, koristeći *type-hinting*, atribute koje očekujemo za proizvod, to su:

- `id` - cijeli broj (`int`)
- `naziv` - string (`str`)
- `boja` - string (`str`)
- `cijena` - decimalni broj (`float`)

```
# models.py

class Proizvod(BaseModel):
    id: int
    naziv: str
    boja: str
    cijena: float
```

Uključujemo ovu klasu u `main.py` datoteku:

```
from fastapi import FastAPI

from models import Proizvod # uključujemo Pydantic model koji smo definirali
```

Međutim, kojoj je svrha ovog modela? U kojoj definiciji rute ćemo ga koristiti? **To ovdje nije jasno naglašeno.**

Primjerice: Kod POST rute za dodavanje proizvoda u listu, do sad smo koristili `dict` kao tip podataka za proizvod koristeći *type hinting*.

```
@app.post("/proizvodi")
def add_proizvod(proizvod: dict):
    proizvod["id"] = len(proizvodi) + 1
    proizvodi.append(proizvod)
    return proizvod
```

Ipak, to nije najbolji pristup budući da korisnik može poslati bilo kakav JSON objekt, odnosno objekt s proizvoljnim ključevima. Želimo ograničiti korisnika na slanje samo točno određenih ključeva u objektu, konkretno na one definirane Pydantic modelom `Proizvod`.

- jednostavno ćemo zamijeniti `dict` s `Proizvod` u definiciji rute:

```

@app.post("/proizvodi")
def add_proizvod(proizvod: Proizvod): # zamijenili smo dict s Proizvod
    proizvod["id"] = len(proizvodi) + 1
    proizvodi.append(proizvod)
    return proizvod

```

No postoji problem. Ako pokušate poslati isti zahtjev za dodavanje novog proizvoda, vidjet ćete da će FastAPI izbaciti grešku:

```
TypeError: 'Proizvod' object does not support item assignment
```

Zašto dolazi do ove greške?

► Spoiler alert! Odgovor na pitanje

Problem je što **Pydantic generira *read-only* modele**, odnosno modele koji ne podržavaju dodavanje novih ključeva (ili brisanje/ažuriranje postojećih) u objekt nakon što je objekt inicijaliziran.

Međutim, ako bolje pogledamo vidimo da je inicijalni problem što smo definirali `id` u samom modelu, a zatim *hintamo* taj tip podataka prilikom dodavanja novog proizvoda **iako znamo da se `id` automatski dodjeljuje na poslužiteljskoj strani**, odnosno vjerojatno bazi podataka u stvarnom svijetu.

Izbacit ćemo `id` iz modela `Proizvod` budući da želimo da se on automatski dodjeljuje:

```

# models.py

class Proizvod(BaseModel):
    naziv: str
    boja: str
    cijena: float

```

Ako bolje pogledate, problem i dalje postoji jer pokušavamo dodati `id` u objekt `proizvod`:

```
proizvod["id"] = len(proizvodi) + 1
```

Ulagna struktura:

```
{
    "naziv": "šal",
    "boja": "plava",
    "cijena": 30
}
```

Očekivana izlazna struktura:

```
{  
    "id": 5,  
    "naziv": "šal",  
    "boja": "plava",  
    "cijena": 30  
}
```

2.1 Input/Output modeli

Samim time, **uobičajena praksa je definirati više Pydantic modela za svaku strukturu**, ovisno u kojoj fazi obrade se nalazi.

Što trebamo? Korisnik šalje podatke bez `id`-a, a poslužitelj vraća podatke s `id`-om.

Input Model koji korisnik šalje uobičajeno je nazvati s prefiksom `Create`, `Update`, `In` ovisno o kojoj se CRUD operaciji radi:

```
# models.py  
  
class CreateProizvod(BaseModel):  
    naziv: str  
    boja: str  
    cijena: float
```

Output Model koji se vraća s poslužitelja natrag korisniku uobičajeno je nazvati s prefiksom `Response` ili `Out`:

```
# models.py  
  
class Proizvod(BaseModel):  
    id: int  
    naziv: str  
    boja: str  
    cijena: float
```

Vratimo se na `main.py` datoteku i uključimo oba modela:

```
# main.py  
from fastapi import FastAPI  
  
from models import CreateProizvod, Proizvod
```

Zamjenit ćemo `dict` s `CreateProizvod` u definiciji rute:

```

@app.post("/proizvodi")
def add_proizvod(proizvod: CreateProizvod): # "ulazni proizvod" mora sadržavati naziv,
boju i cijenu
    proizvod["id"] = len(proizvodi) + 1
    proizvodi.append(proizvod)
    return proizvod

```

Međutim, **sada je potrebno napraviti novu instancu klase `Proizvod`** kako bi se mogao dodati `id`:

- izdvojiti ćemo generiranje `id`-a u samostalnu naredbu
- instancirati ćemo novi objekt `Proizvod` s dodijeljenim `id`-om te preostalim podacima iz `proizvod`
- **objekte Pydantic klasa instanciramo na identičan način kao i obične Python klase**

```

@app.post("/proizvodi")
def add_proizvod(proizvod: CreateProizvod):
    new_id = len(proizvodi) + 1 # generiramo novi ID u samostalnoj naredbi
    proizvod_s_id = Proizvod(id=new_id, naziv=proizvod.naziv, boja=proizvod.boja,
    cijena=proizvod.cijena) # instanciramo novi objekt Proizvod s dodijeljenim ID-om
    return proizvod_s_id

```

Kod radi, ali možemo skratiti posao koristeći *unpacking sintaksu* i pretvorbu Pydantic modela u rječnik.

Važno! Umjesto da navodimo svaki atribut modela `createProizvod` prilikom instanciranja `Proizvod`, možemo prvo **pretvoriti** Pydantic model u rječnik koristeći `model_dump()` metodu a potom raspakirati taj rječnik operatorom `**`

Sintaksa:

```
rjecnik = model.model_dump() # pretvaramo Pydantic model u rječnik
```

Dakle, **kod za instanciranje objekta klase `Proizvod`** možemo skratiti na sljedeći način:

```

@app.post("/proizvodi")
def add_proizvod(proizvod: CreateProizvod):
    new_id = len(proizvodi) + 1
    proizvod_s_id = Proizvod(id=new_id, **proizvod.model_dump()) # koristimo ** za
    raspakiravanje rječnika "proizvod"
    return proizvod_s_id

```

Vraćamo korisniku `proizvod_s_id` koji je tipa `Proizvod`, a ne `CreateProizvod`!

Dodatno, moguće je naglasiti da je povratna vrijednost funkcije `add_proizvod` tipa `Proizvod` unutar dekoratora koristeći `response_model` argument:

Sintaksa:

```
@app.metoda("/ ruta", response_model=PydanticModel)
```

Konkretno za naš primjer:

```

@app.post("/proizvodi", response_model=Proizvod) # naglašavamo da je povratna vrijednost
tipa Proizvod
def add_proizvod(proizvod: CreateProizvod):
    new_id = len(proizvodi) + 1
    proizvod_s_id = Proizvod(id=new_id, **proizvod.model_dump())
    return proizvod_s_id

```

Ovo je korisno jer FastAPI automatski vrši validaciju podataka koje vraćamo korisniku, također **generira dokumentaciju na temelju ove informacije**.

The screenshot shows the 'Schemas' section of the FastAPI documentation. It displays the Pydantic models defined in the code:

- CreateProizvod** (object):
 - `naziv*`: string
 - `boja*`: string
 - `cijena*`: number
- HTTPValidationError** (object): Expand all
- Proizvod** (object):
 - `naziv*`: string
 - `boja*`: string
 - `cijena*`: number
 - `id*`: integer
- ValidationError** (object): Expand all

Na dnu dokumentirane rute možete vidjeti **definirane Pydantic podatkovne modele** pod `Schemas` sekcijom

The screenshot shows the configuration for the `/proizvodi` endpoint:

- Method**: POST
- Path**: `/proizvodi` Add Proizvod
- Parameters**: No parameters
- Request body**: required (application/json)
- Body Content** (JSON example):


```
{
        "naziv": "string",
        "boja": "string",
        "cijena": 0
      }
```
- Buttons**: Cancel, Execute

Uočite da je struktura JSON objekta koji se očekuje (prema Pydantic modelu `CreateProizvod`) odmah prikazana u dokumentaciji

Važno je još naglasiti sljedeće: Nakon što smo validirali podatke koje korisnik šalje (ulazni model `CreateProizvod`), **nije potrebno izrađivati novi objekt** `Proizvod` s dodijeljenim `id`-om budući da bi onda opet trebali pozvati metodu `model_dump()` kako bismo pohranili čisti rječnik u listu proizvoda.

```

@app.post("/proizvodi", response_model=Proizvod)
def add_proizvod(proizvod: CreateProizvod):
    new_id = len(proizvodi) + 1
    proizvod_s_id = Proizvod(id=new_id, **proizvod.model_dump()) # redundantno stvaranje novog objekta Proizvod
    proizvodi.append(proizvod_s_id.model_dump()) # dodajemo rječnik "čistih podataka" u listu proizvoda, a ne Pydantic model!
    return proizvod_s_id

```

Umjesto toga, ako nemamo posebnu potrebnu izrađivati novu instancu klase `Proizvod`, napraviti ćemo samo ono što je potrebno - **validacija podataka**.

U tom slučaju nećemo stvarati instancu, **već samo hintati vrijednost** `proizvod_s_id`!

- uočite da kad ne stvaramo novu instancu, moramo stvarati rječnik vitičastim zagradama `{}` i držati se pravila za definiranje rječnika, možemo i koristiti konstruktor `dict()`:

```

@app.post("/proizvodi", response_model=Proizvod)
def add_proizvod(proizvod: CreateProizvod):
    new_id = len(proizvodi) + 1
    proizvod_s_id : Proizvod = {"id" : new_id, **proizvod.model_dump()} # samo hintamo vrijednost, ne stvaramo novu instancu!
    proizvodi.append(proizvod_s_id) # dodajemo Pydantic model u listu proizvoda
    return proizvod_s_id

```

2.2 Zadaci za vježbu - Osnove definicije ruta i Pydantic modela

- Definirajte novu FastAPI rutu `GET /filmovi` koja će klijentu vraćati listu filmova definiranu u sljedećoj listi:

```
filmovi = [
    {"id": 1, "naziv": "Titanic", "genre": "drama", "godina": 1997},
    {"id": 2, "naziv": "Inception", "genre": "akcija", "godina": 2010},
    {"id": 3, "naziv": "The Shawshank Redemption", "genre": "drama", "godina": 1994},
    {"id": 4, "naziv": "The Dark Knight", "genre": "akcija", "godina": 2008}
]
```

- Nadogradite prethodnu rutu na način da će **output** biti validiran Pydantic modelom `Film` kojeg definirate u zasebnoj datoteci `models.py`.
- Definirajte novu FastAPI rutu `GET /filmovi/{id}` koja će omogućiti pretraživanje novog filma prema `id`-u definiranom u parametru rute `id`. Dodajte i ovdje validaciju Pydantic modelom `Film`.
- Definirajte novu rutu `POST /filmovi` koja će omogućiti dodavanje novog filma u listu filmova. Napravite novi Pydantic model `CreateFilm` koji će sadržavati atribute `naziv`, `genre` i `godina`, a kao output vraćajte validirani Pydantic model `Film` koji predstavlja novododani film s automatski dodijeljenim `id`-em.
- Dodajte query parametre u rutu `GET /filmovi` koji će omogućiti filtriranje filmova prema `genre` i `min_godina`. Zadane vrijednosti za query parametre neka budu `None` i `2000`.

2.3 Složeniji Pydantic modeli

Pydantic modeli mogu sadržavati i **složenije strukture podataka** kao što su liste, rječnici, ugniježđeni modeli i slično. U nastavku ćemo vidjeti kako definirati složenije modele i kako ih koristiti u FastAPI aplikaciji.

U zadatku 2.2 susreli smo se s jednostavnim modelom `Film` koji sadrži samo osnovne atribute, odnosno primitivne tipove podataka. Ako želimo odraditi validaciju podataka za rutu koja vraća više filmova gdje svaki film rječnik validiran instancom klase `Film`, možemo to definirati i ugrađenom `List` klasom.

Primjerice, ako je struktura podataka sljedeća:

```
[  
    {  
        "id": 1,  
        "naziv": "Titanic",  
        "genre": "drama",  
        "godina": 1997  
    },
```

```
{
    "id": 2,
    "naziv": "Inception",
    "genre": "akcija",
    "godina": 2010
}
]
```

Definiramo model `FilmResponse` koji opisuje danu strukturu filma:

```
# models.py

from pydantic import BaseModel

class FilmResponse(BaseModel):
    id: int
    naziv: str
    genre: str
    godina: int
```

Definicija rute (bez Pydantic validacije) u `main.py` izgleda ovako:

```
@app.get("/filmovi", )
def get_filmovi():
    return filmovi
```

Nije potrebno svaki element rječnika eksplicitno pretvarati u instancu modela `FilmResponse`, kao što bi to radili na sljedeći način:

```
@app.get("/filmovi")
def get_filmovi():
    filmovi_objekti = [FilmResponse(**film) for film in filmovi] # pretvaramo svaki rječnik
    iz filmovi u instancu modela FilmResponse
    return filmovi_objekti
```

Iako je kod iznad ispravan, ako bismo dodali novi film u listu `filmovi` kojemu nedostaje neki atribut, primjerice `godina`, poslužitelj će "puknuti" prilikom pokušaja pretvaranja rječnika u instancu modela.

```
filmovi = [
{
    "id": 1,
    "naziv": "Titanic",
    "genre": "drama",
    "godina": 1997
},
{
    "id": 2,
    "naziv": "Inception",
    "genre": "akcija",
```

```

        "godina": 2010
    },
    {
        "id": 3,
        "naziv": "The Matrix",
        "genre": "sci-fi",
    }
]

@app.get("/filmovi")
def get_filmovi():
    filmovi_objekti = [FilmResponse(**film) for film in filmovi] # greška prilikom
    pretvaranja rječnika u instancu modela za film s ID-em 3
    return filmovi_objekti

```

Poslužitelj vraća grešku 500, što je u redu jer je greška na strani poslužitelja.

Ono što ustvari želimo je da FastAPI automatski vrši validaciju i serijalizaciju podataka u JSON prema definiranom modelu `FilmResponse`, **bez eksplisitnog stvaranja instanci modela** za svaki film u listi te na taj način **skratiti kod**.

Rekli smo da to postižemo koristeći parametar `response_model` koji se **dodaje u dekorator rute**:

```

@app.get("/filmovi", response_model=FilmResponse) # ali što je rezultat?
def get_filmovi():
    return filmovi

```

Kako je rezultat ove rute ustvari lista rječnika, moramo to navesti i u `response_model` kako ne bi dobili grešku. **FastAPI će automatski pretvoriti svaki rječnik u listi u instancu modela `FilmResponse`** kako bi se osigurala validacija i serijalizacija podataka, bez potrebe za eksplisitnim stvaranjem instanci modela.

U poglavlju [1.2.1 Parametri ruta \(eng. route parameters\)](#) vidjeli smo da je moguće koristiti kolekciju `list` za *type-hinting* složenijih struktura podataka.

Koristeći uglate zagrade s `list` klasom, možemo definirati da se očekuje lista rječnika, odnosno lista modela `FilmResponse`:

Sintaksa:

```
kolekcija[model]
```

Dakle, ruta sad izgleda ovako:

```

@app.get("/filmovi", response_model=list[FilmResponse]) # povratna vrijednost je lista
rječnika, sada konkretno validirana lista modela FilmResponse
def get_filmovi():
    return filmovi

```

```

Schemas
FilmResponse ^ Collapse all object
  id* integer
  naziv* string
  genre* string
  godina* integer

```

Rezultat je isti, a naš kod je puno kraći i čišći. Dodatno, **na ovaj način FastAPI prikazuje u dokumentaciji strukturu uspješnog odgovora**, međutim nismo riješili problem obrade greške što je u redu, jer je greška nastala na strani poslužitelja, što znači da se radi o pogrešci u implementaciji koju treba ispraviti.

U nastavku ćemo vidjeti na koje sve načine možemo definirati Pydantic modele i to kombiniranjem osnovnih tipova, kolekcija, ugniježđenih modela i drugih složenijih tipova.

2.3.1 Tablica osnovnih tipova

Python Tip	Opis	<i>type-hinting</i> primjer
<code>int</code>	Cijeli brojevi	<code>starost: int = 25</code>
<code>float</code>	Decimalni brojevi	<code>cijena: float = 19.99</code>
<code>str</code>	Znakovni nizovi (tekstualni podaci)	<code>ime: str = "John"</code>
<code>bool</code>	Logičke vrijednosti	<code>je_aktivran: bool = True</code>
<code>bytes</code>	Nepromjenjivi Bajtovi	<code>nepromjenjivi_binarni_podatak: bytes = b"binary data"</code>
<code>bytearray</code>	Promjenjivi (eng. mutable) bajtovi	<code>promjenjivi_binarni_podatak: bytearray = bytearray(b"data")</code>

2.3.2 Tablica kolekcija

Python Tip	Opis	Primjer
<code>list</code>	Lista elemenata bilo kojeg tipa	<code>tags: list[str] = ["tag1", "tag2"]</code>
<code>tuple</code>	Nepromjenjivi niz elemenata	<code>koordinate: tuple[float, float] = (1.0, 2.0)</code>
<code>dict</code>	Rječnik ključ-vrijednost parova	<code>config: dict[str, int] = {"key": 42}</code>
<code>set</code>	Skup jedinstvenih elemenata	<code>kategorije: set[str] = {"A", "B"}</code>
<code>frozenset</code>	Nepromjenjivi skup jedinstvenih elemenata	<code>frozen_kategorije: frozenset[str] = frozenset({"A", "B"})</code>

2.3.3 Primjeri složenijih Pydantic modela

Primjer: Želimo definirati Pydantic model `Korisnik` koji će sadržavati osnovne podatke o korisniku:

Korisnik:

- `id` - cijeli broj
- `ime` - string
- `prezime` - string
- `email` - string
- `dob` - cijeli broj
- `aktivran` - logička vrijednost

Rješenje:

```
class Korisnik(BaseModel):  
    id: int  
    ime: str  
    prezime: str  
    email: str  
    dob: int  
    aktivran: bool
```

Primjer: Želimo definirati Pydantic model `Narudžba` koji će sadržavati osnovne podatke o narudžbi i listu imena naručenih proizvoda:

Narudžba:

- `id` - cijeli broj
- `datum` - string
- `proizvodi` - lista stringova
- `ukupna_cijena` - decimalni broj
- `isporučeno` - logička vrijednost

Rješenje:

```
class Narudzba(BaseModel):  
    id: int  
    datum: str  
    proizvodi: list[str] # lista stringova  
    ukupna_cijena: float  
    isporuceno: bool
```

Osim osnovnih tipova i kolekcija, Pydantic modeli mogu sadržavati i **ugniježđene modele**, odnosno druge Pydantic modele. Ovo je korisno kada želimo definirati složenije strukture podataka koje se sastoje od više manjih dijelova.

Primjer: Želimo definirati Pydantic modele `Proizvod` i `Narudžba` gdje narudžba može sadržavati više proizvoda:

Proizvod:

- `id` - cijeli broj
- `naziv` - string
- `cijena` - decimalni broj
- `kategorija` - string
- `boja` - string

Narudžba:

- `id` - cijeli broj
- `ime_kupca` - string
- `prezime_kupca` - string
- `proizvodi` - lista Proizvoda
- `ukupna_cijena` - decimalni broj

Rješenje:

```
class Proizvod(BaseModel):  
    id: int  
    naziv: str  
    cijena: float  
    kategorija: str  
    boja: str  
  
class Narudzba(BaseModel):  
    id: int  
    ime_kupca: str  
    prezime_kupca: str  
    proizvodi: list[Proizvod] # lista proizvoda  
    ukupna_cijena: float
```

Primjer: Želimo definirati Pydantic modele `Proizvod`, `Narudžba`, `StavkaNarudžbe` gdje narudžba može sadržavati više stavki narudžbe, a svaka stavka narudžbe sadrži jedan proizvod.

Proizvod:

- `id` - cijeli broj
- `naziv` - string
- `cijena` - decimalni broj

- `kategorija` - string
- `boja` - string

StavkaNarudžbe:

- `id` - cijeli broj
- `proizvod` - Proizvod
- `narucena_kolicina` - cijeli broj
- `ukupna_cijena` - decimalni broj

Narudžba:

- `id` - cijeli broj
- `ime_kupca` - string
- `prezime_kupca` - string
- `stavke` - lista StavkaNarudžbe
- `ukupna_cijena` - decimalni broj

Rješenje:

```
class Proizvod(BaseModel):
    id: int
    naziv: str
    cijena: float
    kategorija: str
    boja: str

class StavkaNarudzbe(BaseModel):
    id: int
    proizvod: Proizvod
    narucena_kolicina: int
    ukupna_cijena: float

class Narudzba(BaseModel):
    id: int
    ime_kupca: str
    prezime_kupca: str
    stavke: list[StavkaNarudzbe]
    ukupna_cijena: float
```

Zadane vrijednosti (eng. default values)

Jednako kao kod definicije query parametra, moguće je koristiti **zadane vrijednosti** za atribute Pydantic modela. Zadane vrijednosti se postavljaju na isti način kao i kod običnih Python funkcija, dodavanjem `=` nakon tipa podatka.

Primjer: Definirajmo Pydantic model `Korisnik` koji će sadržavati osnovne podatke o korisničkom računu, a zadana vrijednost će biti za atribut `racun_aktiviran`.

Korisnik:

- `id` - cijeli broj
- `ime` - string
- `prezime` - string
- `email` - string
- `dob` - cijeli broj
- `racun_aktivran` - logička vrijednost, zadana vrijednost `True`

Rješenje:

```
class Korisnik(BaseModel):  
    id: int  
    ime: str  
    prezime: str  
    email: str  
    dob: int  
    racun_aktivran: bool = True
```

Rječnici, n-torke i skupovi

U tablici kolekcija vidimo da, osim lista, Pydantic modeli mogu sadržavati i rječnike, n-torke i skupove. U nastavku ćemo vidjeti kako definirati modele koji sadrže ove složenije strukture podataka.

Primjer: Definirajmo Pydantic model `Loto` koji će sadržavati rezultate loto izvlačenja, a rezultati će biti pohranjeni u rječniku gdje su ključevi cijeli brojevi, a vrijednosti broj pojavljivanja tog broja u izvlačenju.

Loto:

- `id` - cijeli broj
- `rezultati` - rječnik cijelih brojeva i njihovih pojavljivanja

Sintaksa:

```
dict[key_type, value_type]
```

Rješenje:

```
class Loto(BaseModel):  
    id: int  
    rezultati: dict[int, int]
```

Primjer: Definirat ćemo Pydantic model `GeoLokacija` koji će sadržavati informacije o geografskoj lokaciji u obliku n-torke (`latitude`, `longitude`).

GeoLokacija:

- `id` - cijeli broj
- `koordinate` - n-torka decimalnih brojeva

Sintaksa:

```
tuple[type1, type2]
```

Rješenje:

```
class GeoLokacija(BaseModel):
    id: int
    koordinate: tuple[float, float]
```

Primjer: Definirat ćemo Pydantic model `Inventura` koji će sadržavati naziv skladišta i rječnik proizvoda s nazivima proizvoda i njihovim količinama.

Inventura:

- `id` - cijeli broj
- `naziv_skladista` - String
- `proizvodi` - rječnik stringova i cijelih brojeva

Sintaksa:

```
dict[key_type, value_type]
```

Rješenje:

```
class Inventura(BaseModel):
    id: int
    naziv_skladista: str
    proizvodi: dict[str, int]
```

Složeni tipovi iz biblioteke `typing`

U Pythonu postoji biblioteka `typing` koja sadrži dodatne tipove podataka koji se koriste za *type hinting*. Ovi tipovi su korisni kada želimo definirati složenije strukture podataka koje nisu obuhvaćene osnovnim tipovima ili kolekcijama.

Biblioteka `typing` uključena je od Pythona 3.5 te ju nije potrebno naknadno instalirati.

typing Tip	Opis	type-hinting primjer
<code>Union[T1, T2, T3, ... Tn]</code>	Unija se koristi kada vrijednost može biti jedna od više specificiranih podataka. Dakle, u primjeru <code>vrijednost</code> , ona može biti ili <code>int</code> ili <code>str</code> .	<code>vrijednost: Union[int, str] = 42</code>

<code>Optional</code>	Vrijednost može biti opcionalna, ako nije navedena moguće je definirati i zadalu vrijednost. Ekvivalentno: <code>Union[T, None]</code>	<code>ime: Optional[str] = "Nije navedeno pa se zovem Pero"</code>
<code>Any</code>	Vrijednost može biti bilo kojeg tipa podataka	<code>podatak: Any = "Može biti bilo što"</code>
<code>Callable</code>	Funkcija ili "pozivljivi" objekt (Callable). Moguće je navesti argumente funkcije te povratnu vrijednost	<code>funkcija: Callable[[int, str], str] = lambda x, y: f'{x}, {y}'</code>
<code>Literal</code>	Ograničavanje vrijednosti na unaprijed definirane opcije	<code>smjer: Literal['gore', 'dolje'] = "gore"</code>
<code>TypedDict</code>	Specijalni s definiranim tipovima ključeva i vrijednosti	<code>osoba: TypedDict('osoba', {'ime': str, ' prezime': str})</code>

Vrijednosti `typing` biblioteke ima jako puno. Ovdje su navedeni samo neki od najčešće korištenih tipova. Opsežnu dokumentaciju možete pronaći na [službenoj stranici](#).

Primjer: Definirat ćemo Pydantic model `Kolegij` koji će sadržavati informacije o kolegiju. Semestar može biti samo između vrijednosti `[1, 2, 3, 4, 5, 6]`, a vrijednost `ECTS` ne mora biti navedena, u slučaju da nije navedena, zadana vrijednost je `6`.

Kolegij:

- `id` - cijeli broj
- `naziv` - string
- `semestar` - cijeli broj, unutar `[1, 2, 3, 4, 5, 6]`
- `ECTS` - cijeli broj, opcionalan, zadana vrijednost `6`
- `opis` - string
- `profesor` - string

Rješenje:

```
from typing import Optional, Literal

class Kolegij(BaseModel):
    id: int
    naziv: str
    semestar: Literal[1, 2, 3, 4, 5, 6]
    ECTS: Optional[int] = 6
    opis: str
    profesor: str
```

Primjer: Definirat ćemo Pydantic model `Automobil` koji će sadržavati informacije o automobilu. Boja automobila može biti samo jedna od unaprijed definiranih opcija, godina proizvodnje ne mora biti navedena, snaga motora je rječnik s ključevima `kw` i `ks`, a `cijena` je rječnik s ključevima `osnovna` i `sa_pdv`.

Automobil:

- `id` - cijeli broj
- `marka` - string
- `model` - string
- `boja` - string, jedna od opcija `["crvena", "plava", "zelena", "bijela", "crna"]`
- `godina_proizvodnje` - cijeli broj, optionalan
- `snaga_motora` - rječnik s ključevima `kw` i `ks`
- `cijena` - rječnik s ključevima `osnovna` i `sa_pdv`

Rješenje:

```
from typing import Optional, Literal

class Automobil(BaseModel):
    id: int
    marka: str
    model: str
    boja: Literal["crvena", "plava", "zelena", "bijela", "crna"]
    godina_proizvodnje: Optional[int] # godina proizvodnje nije obavezna, ali ako se navede
    mora biti cijeli broj
    snaga_motora: dict[str, int] # zašto je dovoljno samo [str i int]?
    cijena: dict[str, float]
```

Kako bismo instancirali ovu klasu, potrebno je navesti ključeve rječnika `snaga_motora` i `cijena`:

- svaki ključ rječnika `snaga_motora` mora biti string, a vrijednost cijeli broj, međutim **dozvoljeno je navesti neograničeno ključ-vrijednost parova**

```
automobil = Automobil(
    id=1,
    marka="Audi",
    model="A4",
    boja="crvena",
    godina_proizvodnje=2018,
    snaga_motora={"kw": 100, "KS": 136},
    cijena={"osnovna": 25000, "sa_pdv": 30000}
)
```

Kada bi htjeli **ograničiti ključeve** atributa `snaga_motora` i `cijena`, morali bismo definirati zasebne Pydantic modele:

```

class SnagaMotora(BaseModel):
    kW: int
    KS: int

class Cijena(BaseModel):
    osnovna: float
    sa_pdv: float

class Automobil(BaseModel):
    id: int
    marka: str
    model: str
    boja: Literal["crvena", "plava", "zelena", "bijela", "crna"]
    godina_proizvodnje: Optional[int]
    snaga_motora: SnagaMotora
    cijena: Cijena

```

Ovaj automobil instancirali bi na sljedeći način:

```

automobil = Automobil(
    id=1,
    marka="Audi",
    model="A4",
    boja="crvena",
    godina_proizvodnje=2018,
    snaga_motora=SnagaMotora(kW=100, KS=136),
    cijena=Cijena(osnovna=25000, sa_pdv=30000)
)

```

Vidimo da `snaga_motora` i `cijena` više nisu rječnici, već su **ugniježđeni modeli** `SnagaMotora` i `Cijena`.

Ipak, moguće ih je definirati kao posebne rječnike tipa `TypedDict` iz modula `typing` koji omogućuje definiranje rječnika s točno određenim ključevima.

- sintaksa je ista, jedino što klase nasljeđuju `TypedDict` umjesto `BaseModel`

```

from typing import TypedDict

class SnagaMotora(TypedDict):
    kW: int
    KS: int

class Cijena(TypedDict):
    osnovna: float
    sa_pdv: float

class Automobil(BaseModel):
    id: int
    marka: str
    model: str
    boja: Literal["crvena", "plava", "zelena", "bijela", "crna"]

```

```
godina_proizvodnje: Optional[int]
snaga_motora: SnagaMotora
cijena: Cijena
```

Ovaj automobil instancirali bi na sljedeći način:

```
automobil = Automobil(
    id=1,
    marka="Audi",
    model="A4",
    boja="crvena",
    godina_proizvodnje=2018,
    snaga_motora={"kW": 100, "KS": 136},
    cijena={"osnovna": 25000, "sa_pdv": 30000}
)
```

2.4 Nasljeđivanje Pydantic modela

Nasljeđivanje (*eng. inheritance*) je koncept u programiranju gdje jedan objekt (klasa) može naslijediti atribute i metode drugog objekta (klase). Već smo vidjeli na početku kolegija da je moguće nasljeđivati atribute i metode klase A na način da ju navodimo u zagradama prilikom definicije klase B.

Ista pravila vrijede za Pydantic modele. Ako želimo definirati novi Pydantic model koji će naslijediti atribute i metode nekog drugog modela, to možemo učiniti na sljedeći način:

Sintaksa:

```
# Pydantic model A
class A(BaseModel):
    atribut_a: str
    atribut_b: int

# Pydantic model B koji nasljeđuje atribute i metode modela A i dodaje vlastiti atribut

class B(A):
    atribut_c: float
```

Objekte ovakvih modela instanciramo na isti način kao i obične modele:

```
objekt_a = A(atribut_a="vrijednost_a", atribut_b=42)
objekt_b = B(atribut_a="vrijednost_a", atribut_b=42, atribut_c=3.14)
```

U kontekstu FastAPI poslužitelja i modeliranja podataka, uobičajeno je koristiti prefix `Base` za osnovne modele koji sadrže zajedničke atribute, a zatim nasljeđivati te modele u specifičnijim modelima, npr.

`Create`, `Update`, `Response`, `In`, `Out` i slično.

Ako se vratimo na model `Proizvod` koji smo imali u dosadašnjim primjerima, možemo definirati modele `BaseProizvod`, `RequestProizvod` i `ResponseProizvod`.

- kako prilikom dodavanja proizvoda ne želimo da korisnik unosi `id`, niti cijenu s PDV-om koju ćemo računati na poslužitelju (u ovom slučaju 25% PDV-a), možemo definirati `BaseProizvod` model koji sadrži osnovne atribute proizvoda
- u tom slučaju, klasa `RequestProizvod` nasljeđuje atribute iz `BaseProizvod` modela i ne dodaje ništa novo (jer to su atribute koje klijent šalje poslužitelju)
- klasa `ResponseProizvod` nasljeđuje atribute iz `BaseProizvod` modela i dodaje `id` atribut te računa cijenu s PDV-om u atributu `cijena_pdv`

```

class BaseProizvod(BaseModel):
    naziv: str
    cijena: float
    kategorija: str
    boja: str

class RequestProizvod(BaseProizvod): # nasljeđujemo attribute iz BaseProizvod modela
    pass # ne dodajemo niti jedan novi atribut

class ResponseProizvod(BaseProizvod): # nasljeđujemo attribute iz BaseProizvod modela
    id: int
    cijena_pdv: float

```

Primjer rute za dodavanje proizvoda s ukupnom validacijom podataka:

```

@app.post("/proizvodi", response_model=ResponseProizvod) # validacija i serijalizacija
HTTP odgovora prema ResponseProizvod modelu
def dodaj_proizvod(proizvod: RequestProizvod): # RequestProizvod model koristimo za
validaciju podataka koje klijent šalje
    PDV_MULTIPLIER = 1.25
    some_id = random.randrange(1, 100) # simuliramo dodjelu ID-a
    cijena_pdv = proizvod.cijena * PDV_MULTIPLIER # računamo cijenu s PDV-om
    proizvod_spreman_za_pohranu : ResponseProizvod = {**proizvod.model_dump(), "id":some_id,
"cijena_pdv":cijena_pdv} # ne instanciramo novi ResponseProizvod, već koristimo type-
hinting
    proizvodi.append(proizvod_spreman_za_pohranu)
    return proizvod_spreman_za_pohranu

```

Curl

```
curl -X 'POST' \
  'http://localhost:8000/proizvodi' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
    "naziv": "Sot",
    "cijena": 300,
    "kategorija": "sotovi",
    "boja": "siva"
}'
```

Request URL

<http://localhost:8000/proizvodi>

Server response

Code	Details
200	<p>Response body</p> <pre>{ "naziv": "Sot", "cijena": 300, "kategorija": "sotovi", "boja": "siva", "id": 64, "cijena_pdv": 375 }</pre> <p>Download</p> <p>Response headers</p> <pre>content-length: 93 content-type: application/json date: Sun, 12 Jan 2025 19:08:23 GMT server: uvicorn</pre>

Responses

Code	Description	Links
200	Successful Response	No links

U dokumentaciji vidimo da su poslati atributi `naziv`, `cijena`, `kategorija` i `boja`, a vraćeni atributi su `id`, `naziv`, `cijena`, `kategorija`, `boja` i `cijena_pdv`.

Primjer: Definirajmo Pydantic modele `KorisnikBase`, `KorisnikCreate` i `KorisnikResponse` koji će sadržavati osnovne podatke o korisniku, podatke koje korisnik šalje prilikom registracije i podatke koje korisnik dobiva kao odgovor prilikom registracije. Dodatno, `KorisnikResponse` model sadrži i atribut `datum_registracije` koji predstavlja trenutni datum i vrijeme registracije korisnika.

- lozinka koju korisnik šalje prilikom registracije je u tekstuallnom obliku, međutim, prilikom registracije u bazi podataka, lozinka se sprema kao heširana vrijednost
- osim heširane lozinke, povratna vrijednost nakon uspješne registracije sadrži i datum registracije koji će biti objekt tipa `datetime`

KorisnikBase:

- `ime` - string
- `prezime` - string
- `email` - string

KorisnikCreate: nasljeđuje atribute iz `KorisnikBase` modela

- `lozinka_text` - string

KorisnikResponse: nasljeđuje atribute iz `KorisnikBase` modela

- `lozinka_hash` - string
- `datum_registracije` - objekt tipa `datetime`

Rješenje:

```
from datetime import datetime

class KorisnikBase(BaseModel):
    ime: str
    prezime: str
    email: str

class KorisnikCreate(KorisnikBase):
    lozinka_text: str

class KorisnikResponse(KorisnikBase):
    lozinka_hash: str
    datum_registracije: datetime # hintamo složeni objekt tipa datetime
```

Primjer rute za registraciju korisnika:

```

@app.post("/korisnici", response_model=KorisnikResponse) # validacija i serijalizacija
HTTP odgovora prema KorisnikResponse modelu
def registracija_korisnika(korisnik: KorisnikCreate):

    lozinka_hash = str(hash(korisnik.lozinka_text)) # simuliramo heširanje lozinke
    datum_registracije = datetime.now() # trenutni datum i vrijeme registracije
    korisnik_spreman_za_pohranu : KorisnikResponse = {**korisnik.model_dump(),
"lozinka_hash" : lozinka_hash, "datum_registracije": datum_registracije} # uzimamo sve iz
KorisnikCreate + lozinka_hash i datum_registracije kako bismo zadovoljili KorisnikResponse
model

    print(f"Korisnik spremam za pohranu: {korisnik_spreman_za_pohranu}")

    korisnici.append(korisnik_spreman_za_pohranu)
    return korisnik_spreman_za_pohranu # vraćamo KorisnikResponse model

```

Validacijom podataka kroz ova tri modela postigli smo sljedeće:

- klijent šalje podatke o korisniku prilikom registracije te unosi **ime, prezime, lozinku u tekstualnom obliku i email**
- podaci koje klijent šalje se validiraju prema `KorisnikCreate` modelu
- na poslužitelju se **hešira lozinka i dodaje datum registracije** te se podaci validiraju prema `KorisnikResponse` modelu
- u bazu podataka (u ovom slučaju *in-memory* lista) sprema se heširana lozinka i datum registracije, **bez lozinke u tekstualnom obliku!**
- klijent dobiva odgovor s podacima o korisniku, **bez lozinke u tekstualnom obliku!**

```

Curl
curl -X 'POST' \
'http://localhost:8000/korisnici' \
-H 'accept: application/json' \
-H 'Content-Type: application/json' \
-d '{
    "ime": "string",
    "prezime": "string",
    "email": "string",
    "lozinka_text": "string"
}'
Request URL
http://localhost:8000/korisnici
Server response
Code Details
200 Response body
{
    "ime": "string",
    "prezime": "string",
    "email": "string",
    "lozinka_hash": "2698794031908662734",
    "datum_registracije": "2025-01-12T21:32:43.472310"
}
Download
Response headers
content-length: 129
content-type: application/json
date: Sun, 12 Jan 2025 20:32:43 GMT
server: unicorn

```

U dokumentaciji vidimo da su poslani atributi `ime`, `prezime`, `email` i `lozinka_text`, a vraćeni atributi su `ime`, `prezime`, `email`, `lozinka_hash` i `datum_registracije`.

2.5 Zadaci za vježbu: Definicija složenijih Pydantic modela

- Definirajte Pydantic modele `Knjiga` i `Izdavač` koji će validirati podatke o knjigama i izdavačima. Svaka knjiga sastoji se od naslova, imena autora, prezimena autora, godine izdavanja, broja stranica i izdavača. Izdavač se sastoji od naziva i adrese. Ako godina izdavanja nije navedena, zadana vrijednost je trenutna godina.
- Definirajte Pydantic model `Admin` koji validira podatke o administratoru sustava. Administrator se sastoji od imena, prezimena, korisničkog imena, emaila te ovlasti. Ovlasti su lista stringova koje mogu sadržavati vrijednosti: `dodavanje`, `brisanje`, `ažuriranje`, `čitanje`. Ako ovlasti nisu navedene, zadana vrijednost je prazna lista. Za ograničavanje ovlasti koristite `Literal` tip iz modula `typing`.
- Definirajte Pydantic model `Restaurantorder` koji se sastoji od informacija o narudžbi u restoranu. Narudžba se sastoji od identifikatora, imena kupca, `stol_info`, liste jela i ukupne cijene. Definirajte još jedan model za jelo koje se sastoji od identifikatora, naziva i cijene. Za `stol_info` pohranite rječnik koji očekuje ključeve `broj` i `lokacija`. Primjerice, `stol_info` može biti `{"broj": 5, "lokacija": "terasa"}`. Za definiciju takvog rječnika koristite `TypedDict` tip iz modula `typing`.
- Definirajte Pydantic modela `cctv_frame` koji će validirati podatke o trenutnoj slici s CCTV kamere. Trenutna slika se sastoji od identifikatora, vremena snimanja, te koordinate x i y. Koordinate validirajte kao n-torku decimalnih brojeva. Ako koordinate nisu navedene, zadana vrijednost je `(0.0, 0.0)`.

2.6 Field polje Pydantic modela

U prethodnim primjerima vidjeli smo kako definirati Pydantic modele koristeći atribute i nasljeđivanje. U nekim slučajevima, možda ćemo htjeti definirati dodatne podatke o atributima, kao što su:

- zadane vrijednosti
- opisi atributa
- ograničenja
- aliasi
- ...

Za to koristimo `Field` polje koje se nalazi u modulu `pydantic`. `Field` polje koristi se za **definiranje dodatne informacije o atributima** Pydantic modela.

Sintaksa:

```
from pydantic import Field

class NekiModel(BaseModel):
    neki_atribut: tip = Field()
```

Primjerice, ako se vratimo na model `Korisnik` koji smo definirali ranije, možemo dodati dodatne informacije (`description`) o atributima koje bi željeli poslati korisniku u slučaju da dođe do validacijske pogreške:

```
from pydantic import Field

class Korisnik(BaseModel):
    id: int = Field(description="Jedinstveni identifikator korisnika")
    ime: str = Field(description="Ime korisnika")
    prezime: str = Field(description="Prezime korisnika")
    email: str = Field(description="Email adresa korisnika")
    dob: int = Field(description="Datum rođenja korisnika")
    aktivan: bool = Field(description="Je li korisnik aktivan")
```

```
Korisnik ▾ Collapse all object
  id* ▾ Collapse all integer
    Jedinstveni identifikator korisnika
  ime* ▾ Collapse all string
    Ime korisnika
  prezime* ▾ Collapse all string
    Prezime korisnika
  email* ▾ Collapse all string
    Email adresa korisnika
  dob* ▾ Collapse all integer
    Datum rođenja korisnika
  aktivan* ▾ Collapse all boolean
    Je li korisnik aktivan
```

U dokumentaciji vidimo definirane opise atributa

Ako bismo ovdje sada htjeli dodati zadane vrijednosti, koristimo `default` parametar u `Field` polju:

```
from pydantic import Field

class Korisnik(BaseModel):
    id: int = Field(description="Jedinstveni identifikator korisnika", default=1)
    ime: str = Field(description="Ime korisnika", default="John")
    prezime: str = Field(description="Prezime korisnika", default="Doe")
    email: str = Field(description="Email adresa korisnika", default="JohnDoe@gmail.com")
    dob: int = Field(description="Datum rođenja korisnika", default=1990)
    aktivan: bool = Field(description="Je li korisnik aktivan", default=True)
```

Ukoliko želimo **ograničiti vrijednosti numeričkih atributa**, koristimo `ge` i `le` parametre u `Field` polju:

- `ge` - greater than or equal to
- `gt` - greater than
- `le` - less than or equal to
- `lt` - less than

```

from pydantic import Field

class Korisnik(BaseModel):
    id: int = Field(description="Jedinstveni identifikator korisnika", ge=1, le=100) # id
    mora biti između 1 i 100
    ime: str = Field(description="Ime korisnika")
    prezime: str = Field(description="Prezime korisnika")
    email: str = Field(description="Email adresa korisnika")
    dob: int = Field(description="Datum rođenja korisnika", ge=1900, le=2021) # datum
    rođenja mora biti između 1900 i 2021
    aktivavan: bool = Field(description="Je li korisnik aktivan")

```

Ukoliko želimo ograničiti duljine znakovnih nizova, koristimo `max_length` i `min_length` argumente u `Field` polju:

```

from pydantic import Field

class Korisnik(BaseModel):
    id: int = Field(description="Jedinstveni identifikator korisnika", ge=1, le=100)
    ime: str = Field(description="Ime korisnika", min_length=2, max_length=50) # ime mora
    imati između 2 i 50 znakova
    prezime: str = Field(description="Prezime korisnika", min_length=2, max_length=50) # prezime mora imati između 2 i 50 znakova
    email: str = Field(description="Email adresa korisnika")
    dob: int = Field(description="Datum rođenja korisnika", ge=1900, le=2021)
    aktivavan: bool = Field(description="Je li korisnik aktivan")

```

U sljedećoj tablici dani su česti parametri koji se koriste u `Field` polju:

Field Parametar	Opis parametra	Primjer
<code>default</code>	Zadana vrijednost za polje.	<code>ime: str = Field("Ivan Horvat")</code>
<code>default_factory</code>	Funkcija koja dinamički generira zadanu vrijednost.	<code>kreirano: datetime = Field(default_factory=datetime.utcnow)</code>
<code>title</code>	Naslov za polje, koristi se za dokumentaciju.	<code>ime: str = Field(..., title="Puno ime")</code>
<code>description</code>	Opis polja, koristi se za dokumentaciju.	<code>dob: int = Field(..., description="Dob osobe, mora biti 18 ili više")</code>
<code>alias</code>	Alternativni naziv za polje u serijaliziranim podacima.	<code>email: str = Field(..., alias="email_adresa")</code>
<code>const</code>	Ako je <code>True</code> , vrijednost se ne može mijenjati nakon inicijalizacije.	<code>uloga: str = Field("admin", const=True)</code>
<code>gt</code>	Vrijednost mora biti veća od ove.	<code>rezultat: int = Field(..., gt=0)</code>
<code>ge</code>	Vrijednost mora biti veća ili jednaka ovoj.	<code>dob: int = Field(..., ge=18)</code>
<code>lt</code>	Vrijednost mora biti manja od ove.	<code>postotak: float = Field(..., lt=100)</code>
<code>le</code>	Vrijednost mora biti manja ili jednaka ovoj.	<code>ocjena: int = Field(..., le=10)</code>
<code>min_length</code>	Minimalna duljina stringa ili liste.	<code>korisnicko_ime: str = Field(..., min_length=3)</code>

max_length	Maksimalna duljina stringa ili liste.	lozinka: str = Field(..., max_length=20)
regex	Regex obrazac koji polje mora zadovoljiti.	email: str = Field(..., regex=r'^\S+@\S+\.\S+\$')

3. Obrada grešaka (eng. Error Handling)

Do sad smo naučili kako definirati osnovne FastAPI rute koje prihvaćaju parametre rute, query parametre i tijelo zahtjeva. Također smo naučili kako definirati Pydantic modele koji služe za validaciju dolaznih podataka, automatsku deserijalizaciju i serijalizaciju podataka te automatsku generaciju dokumentacije.

U ovom poglavlju ćemo se upoznati s dodatnim sigurnosnim mehanizmima koje svaki robusni poslužitelj mora imati u svojim definicijama ruta. To je naravno obrada grešaka koje mogu nastati korisničkom pogreškom (`4xx`) ili greškom na poslužitelju (`5xx`).

FastAPI ima gotovu podršku za obradu grešaka kroz `HTTPException` klasu.

```
from fastapi import HTTPException
```

Ova klasa koristi se za podizanje iznimke u slučaju greške, ustvari se radi o običnoj Python iznimci (`Exception`) koja se podiže kada dođe do greške, ali u našem slučaju sadrži dodatne informacije o statusu greške i poruci koja se vraća korisniku u kontekstu HTTP protokola.

Za **vraćanje iznimke** u Pythonu, općenito koristimo ključnu riječ `raise`:

```
raise Exception("Došlo je do greške")
```

Primjerice: ako korisnik pokuša pristupiti resursu koji ne postoji, možemo podići iznimku `HTTPException` s odgovarajućim statusom (`status_code`) i porukom (`detail`):

```
raise HTTPException(status_code=404, detail="Resurs nije pronađen")
```

Uzet ćemo sljedeći primjer: korisnik pokušava dohvatiti podatke o knjigama, međutim zatraži knjigu s naslovom koji ne postoji u bazi podataka. U tom slučaju, podižemo iznimku `HTTPException` s statusom `404` i porukom `Knjiga nije pronađena`.

```
from fastapi import FastAPI, HTTPException

app = FastAPI()

knjige = [
    {"id": 1, "naslov": "Ana Karenjina", "autor": "Lav Nikolajević Tolstoj"},
    {"id": 2, "naslov": "Kiklop", "autor": "Ranko Marinković"},
    {"id": 3, "naslov": "Proces", "autor": "Franz Kafka"}
]

@app.get("/knjige/{naslov}", response_model=Knjiga)
def dohvati_knjigu(naslov: str):
    for knjiga in knjige:
        if knjiga["naslov"] == naslov:
            return knjiga # vraćamo knjigu ako je pronađena
    raise HTTPException(status_code=404, detail="Knjiga nije pronađena") # podižemo iznimku
    # ako knjiga nije pronađena s odgovarajućom porukom i statusnim kodom
```

Definirat ćemo rutu za dodavanje nove knjige, međutim, ako korisnik pokuša dodati knjigu koja već postoji u bazi podataka, podići ćemo iznimku `HTTPException` s statusom `400` i porukom `Knjiga već postoji`.

Definirat ćemo prvo odgovarajuće Pydantic modele:

```
# models.py

from pydantic import BaseModel

class KnjigaRequest(BaseModel):
    naslov: str
    autor: str

class KnjigaResponse(KnjigaRequest):
    id: int
```

```
# main.py

@app.post("/knjige", response_model=KnjigaResponse)
def dodaj_knjigu(knjiga_request: KnjigaRequest):
    for pohranjena_knjiga in knjige: # prolazimo kroz sve knjige u "bazi podataka"
        if pohranjena_knjiga["naslov"] == knjiga_request.naslov:
            raise HTTPException(status_code=400, detail="Knjiga već postoji")
    new_id = knjige[-1]["id"] + 1
    nova_knjiga : KnjigaResponse = {"id": new_id, **knjiga_request.model_dump()} # ne
instanciramo novi KnjigaResponse, već koristimo type-hinting
    knjige.append(nova_knjiga) # dodajemo rječnik koji predstavlja knjigu
    return nova_knjiga
```

Općenito, klasa `HTTPException` ima sljedeće parametre:

- `status_code` - statusni kod HTTP odgovora
- `detail` - poruka koja se vraća korisniku
- `headers` - dodatna zaglavlja HTTP odgovora

Naravno, moguće je strukturirati rutu i na način da može podići više različitih iznimki, ovisno o situaciji:

```
@app.get("/knjige/{id}", response_model=KnjigaResponse)
def dohvati_knjigu(id: int):

    if id < 1:
        raise HTTPException(status_code=400, detail="ID mora biti veći od 0")

    for knjiga in knjige:
        if knjiga["id"] == id:
            return knjiga # vraćamo knjigu ako je pronađena
    raise HTTPException(status_code=404, detail=f"Knjiga s id-em {id} nije pronađena") # podižemo iznimku ako knjiga nije pronađena s odgovarajućom porukom i statusnim kodom
```

Osim direktnog upisa statusnih kodova, postoji konvencija korištenja specijalnog `status` modula iz FastAPI paketa koji sadrži gotove statusne kodove.

- na ovaj način povećavamo čitljivost koda i smanjujemo mogućnost greške
- također, ovim principom naš IDE može bolje prepoznati statusne kodove te ga sam editor može pronaći

```
from fastapi import status

@app.get("/knjige/{id}", response_model=KnjigaResponse)
def dohvati_knjigu(id: int):

    if id < 1:
        raise HTTPException(status_code=status.HTTP_400_BAD_REQUEST, detail="ID mora biti veći od 0") # koristimo status modul za statusni kod

    for knjiga in knjige:
        if knjiga["id"] == id:
            return knjiga # vraćamo knjigu ako je pronađena
    raise HTTPException(status_code=status.HTTP_404_NOT_FOUND, detail=f"Knjiga s id-em {id} nije pronađena") # koristimo status modul za statusni kod
```

Sve statusne kodove unutar ovog modula možete pronaći na sljedećoj [poveznicu](#)

Za kraj, ako radite vaš projekt koristeći WebSocket protokol, FastAPI ima podršku za podizanje iznimki kroz `WebSocket` klasu:

```
from fastapi import WebSocketException
```

Međutim, to nije predmet ovih vježbi. Za sve kojih zanima više o WebSocket protokolu, posjetite sljedeću [poveznicu](#).

3.1 Validacija parametara rute i query parametra

U primjeru iznad validirali smo tijelo zahtjeva kroz Pydantic model `KnjigaResponse`, odnosno `KnjigaRequest` za POST rutu. Međutim, ponekad želimo validirati i parametre rute i query parametre koje korisnik šalje u URL-u na sličan način kao što smo validirali tijelo zahtjeva.

U tu svrhu postoje `Path` i `Query` polja iz modula `fastapi` koja koristimo za validaciju parametara rute i query parametara.

Primjer: Vidjeli smo kako možemo validirati parametre rute i query parametre u FastAPI ruti koristeći `type-hinting`. No, što ako moramo provjeriti kao u primjeru iznad je li ID veći od 0? Upotrijebit ćemo `Path` polje za validaciju parametara rute.

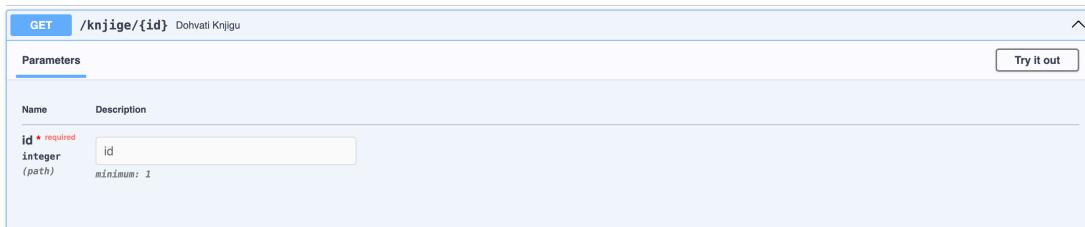
```

from fastapi import Path

@app.get("/knjige/{id}", response_model=KnjigaResponse)
def dohvati_knjigu(id: int = Path(title="ID knjige", ge=1)): # koristimo isti "ge"
    parametar kao u Field polju
    for knjiga in knjige:
        if knjiga["id"] == id:
            return knjiga # vraćamo knjigu ako je pronađena
    raise HTTPException(status_code=404, detail=f"Knjiga s id-em {id} nije pronađena") # podižemo iznimku ako knjiga nije pronađena s odgovarajućom porukom i statusnim kodom

```

Na ovaj način, osim čišćeg koda, dobivamo i oznaku `"minimum : 1"` u dokumentaciji koja korisniku daje informaciju o minimalnoj vrijednosti ovog parametra.



Dobivamo oznaku `"minimum : 1"` u dokumentaciji koja korisniku daje informaciju o minimalnoj vrijednosti ovog parametra.

Više u ovom obliku validacije parametra rute na [FastAPI dokumentaciji](#).

Na isti način možemo validirati i query parametre koristeći `Query` polje. Malo ćemo proširiti podatke o našim knjigama na način da sadrže i informaciju o broju stranica i godini izdavanja.

```

knjige = [
    {"id": 1, "naslov": "Ana Karenjina", "autor": "Lav Nikolajević Tolstoj",
     "broj_stranica": 864, "godina_izdavanja": 1877},
    {"id": 2, "naslov": "Kiklop", "autor": "Ranko Marinković", "broj_stranica": 488,
     "godina_izdavanja": 1965},
    {"id": 3, "naslov": "Proces", "autor": "Franz Kafka", "broj_stranica": 208,
     "godina_izdavanja": 1925}
]

```

Nadogradit ćemo i Pydantic modele:

```
# models.py

from pydantic import BaseModel, Field

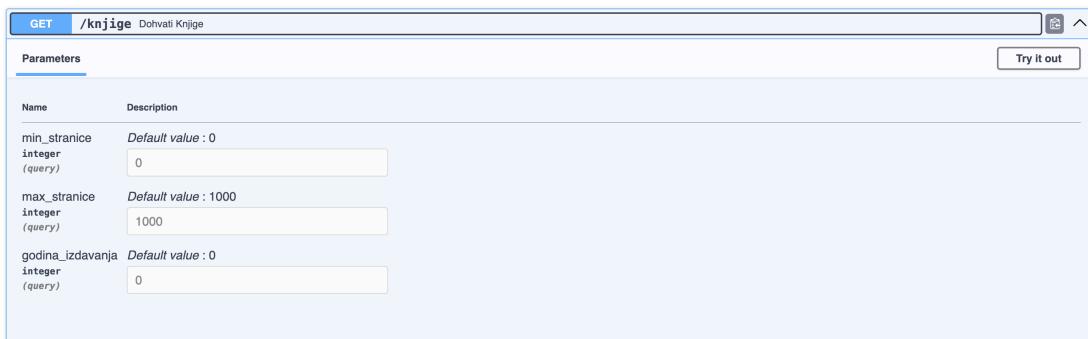
class KnjigaRequest(BaseModel):
    naslov: str
    autor: str
    broj_stranica: int = Field(ge=1) # broj stranica mora biti veći od 1
    godina_izdavanja: int = Field(ge=0, le=2024) # godina izdavanja mora biti između 0 i
    2024
```

Idemo definirati rutu za dohvaćanje svih knjiga s 3 query parametra: `min_stranice`, `max_stranice` i `godina_izdavanja`.

Prvo **primjer s osnovnom validacijom** query parametara kroz *type-hinting*:

```
@app.get("/knjige")
def dohvati_knjige(min_stranice: int = 0, max_stranice: int = 1000, godina_izdavanja: int
= 0):
    filtrirane_knjige = []
    for knjiga in knjige:
        if knjiga["broj_stranica"] >= min_stranice and knjiga["broj_stranica"] <= max_stranice
        and knjiga["godina_izdavanja"] == godina_izdavanja:
            filtrirane_knjige.append(knjiga)
    return filtrirane_knjige
```

Primjer dokumentirane rute:



U dokumentaciji vidimo da su query parametri `min_stranice`, `max_stranice` i `godina_izdavanja` s zadanim vrijednostima.

Međutim, možemo dodatno **proširiti validaciju query parametara** kroz `Query` polje:

- `min_stranice` mora biti veći od 0
- `max_stranice` mora biti veći od 0
- `godina_izdavanja` mora biti između 0 i 2024
- `min_stranice` mora biti manji od `max_stranice` (ovo radimo u samoj funkciji)

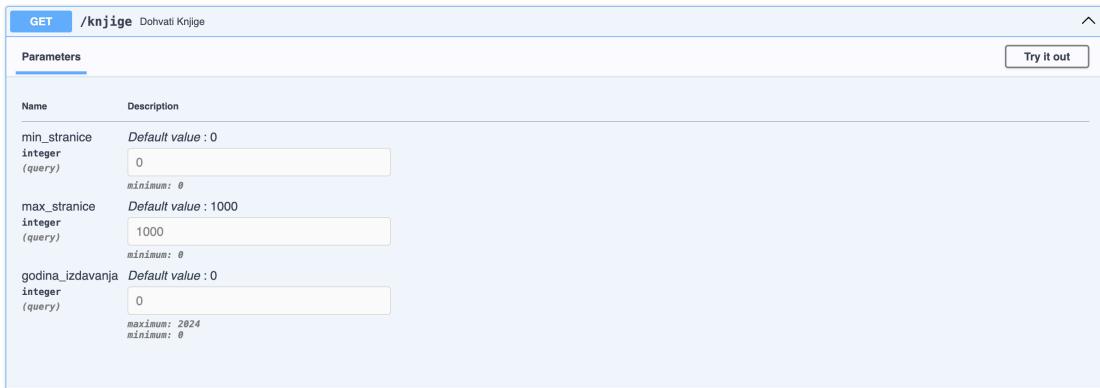
```

from fastapi import Query

@app.get("/knjige")
def dohvati_knjige(min_stranice: int = Query(0, ge=1), max_stranice: int = Query(1000, ge=1), godina_izdavanja: int = Query(0, ge=0, le=2024)):
    if min_stranice > max_stranice:
        raise HTTPException(status_code=400, detail="Minimalni broj stranica mora biti manji od maksimalnog")
    filtrirane_knjige = []
    for knjiga in knjige:
        if knjiga["broj_stranica"] >= min_stranice and knjiga["broj_stranica"] <= max_stranice and knjiga["godina_izdavanja"] == godina_izdavanja:
            filtrirane_knjige.append(knjiga)
    return filtrirane_knjige

```

Primjer dokumentirane rute s dodatnim validacijama:



3.2 Zadaci za vježbu: Obrada grešaka

- Definirajte rutu i odgovarajući Pydantic model za dohvaćanje podataka o automobilima. Svaki automobil ima sljedeće atribute: `id`, `marka`, `model`, `godina_proizvodnje`, `cijena` i `boja`. Ako korisnik pokuša dohvatiti automobil s ID-em koji ne postoji, podignite iznimku `HTTPException` s statusom `404` i porukom `Automobil nije pronađen`.
- Nadogradite prethodnu rutu s query parametrima `min_cijena`, `max_cijena`, `min_godina` i `max_godina`. Implementirajte validaciju query parametra za cijenu i godinu proizvodnje. Minimalna cijena mora biti veća od 0, a minimalna godina proizvodnje mora biti veća od 1960. Unutar funkcije obradite iznimku kada korisnik unese minimalnu cijenu veću od maksimalne cijene ili minimalnu godinu proizvodnje veću od maksimalne godine proizvodnje te vratite odgovarajući `HTTPException`.
- Definirajte rutu za dodavanje novog automobila u bazu podataka. `id` se mora dodati na poslužitelju, kao i atribut `cijena_pdv` (definirajte dodatni Pydantic model za to). Ako korisnik pokuša dodati automobil koji već postoji u bazi podataka, podignite odgovarajuću iznimku. Implementirajte ukupno 3 Pydantic modela, uključujući `BaseCar` model koji će nasljeđivati preostala 2 modela.

4. Strukturiranje poslužitelja i organizacija koda

U ovom poglavlju ćemo se upoznati s organizacijom koda u FastAPI poslužitelju. Kako bi naš poslužitelj bio čitljiviji i lakši za održavanje, bitno je organizirati kod na način da bude strukturiran i pregledan.

4.1 Dependency Injection (DI)

FastAPI ima moćan **Dependency Injection** sustav koji omogućuje da se kod poslužitelja strukturira na način da se smanji ponavljanje koda i poveća čitljivost.

Dependency Injection (*DI*) je dizajnerski obrazac u softverskom inženjerstvu koji omogućava bolju modularnost programskog proizvoda.

DI je ustvari način upravljanja ovisnostima (*eng. Dependency*) u aplikaciji tako da se vanjske ovisnosti klase ili objekta "ubrizgavaju" izvana, umjesto da ih klasa sama stvara ili pronalazi.

Ovakav dizajnerski obrazac je koristan kada:

- želimo smanjiti ovisnost između klasa
- postoji logika koja se ponavlja u više klasa, odnosno koju je potrebno dijeliti
- dijeljenje konekcije na bazu podataka
- dijeljenje konfiguracijskih postavki
- dijeljenje autorizacijske logike

Kada koristimo FastAPI, DI možemo ostvariti koristeći modul `Depends` iz FastAPI paketa.

```
from fastapi import Depends
```

Dependency Injection koristimo tako da definiramo **funkciju koja vraća ovisnost**, a zatim tu funkciju koristimo kao argument u rutu.

Primjerice: Zamislimo da imamo poslužitelj koji sadrži nekoliko administratorskih ruta, ali za pristup tim rutama **korisnik mora biti autoriziran**. Simulirat ćemo funkciju koja vraća korisničko ime na temelju tokena koji pristiže s HTTP zahtjevom.

Ideja je sljedeća:

- korisnik šalje **token** s HTTP zahtjevom kojim dokazuje da je autoriziran i da je on administrator
- ako se token ne podudara s tokenom koji je potreban za pristup administratorskim rutama, korisniku se vraća greška

```

@app.get("/tajni_podaci")
def get_tajni_podaci(token: str):
    if token != "super_secret_admin_token007": # provjeravamo je li token ispravan
        # simuliramo samo naravno
        raise HTTPException(status_code=401, detail="Nemate ovlasti za pristup ovim podacima")
    return {"tajni_podaci": "šifra za sef je 1234"}

```

Ako dodamo još nekoliko ruta, primjerice za ažuriranje i brisanje tajnih podataka, morat ćeemo ponavljati ovu provjeru u svakoj ruti.

```

@app.put("/tajni_podaci")
def update_tajni_podaci(token: str, podaci: dict):
    if token != "super_secret_admin_token007":
        raise HTTPException(status_code=401, detail="Nemate ovlasti za pristup ovim podacima")
    # ažuriramo podatke...
    return {"poruka": "Podaci uspješno ažurirani"}

@app.delete("/tajni_podaci")
def delete_tajni_podaci(token: str):
    if token != "super_secret_admin_token007":
        raise HTTPException(status_code=401, detail="Nemate ovlasti za pristup ovim podacima")
    # brišemo podatke...
    return {"poruka": "Podaci uspješno obrisani"}

```

Možemo jednostavno izdvojiti kod za provjeru tokena u zasebnu funkciju i **koristiti je kao ovisnost u svakoj ruti**.

```

def provjeri_token(token: str):
    if token != "super_secret_admin_token007":
        raise HTTPException(status_code=401, detail="Nemate ovlasti za pristup ovim podacima")
    return token

```

Ili možemo simulirati vraćanje korisnika koji se nalazi u bazi podataka na temelju tokena:

```

from pydantic import BaseModel

class Admin(BaseModel):
    korisnicko_ime: str
    token: str

administratori = [
    {"korisnicko_ime": "secret_admin_007", "token": "super_secret_admin_token007"},
    {"korisnicko_ime": "secret_admin_123", "token": "admin_token123"},
    {"korisnicko_ime": "secret_admin_456", "token": "admin_token456"}
]

def provjeri_token(token: str):
    for admin in administratori:
        if admin["token"] == token:

```

```
    return Admin(**admin) # vraćamo instancu Admin klase
    raise HTTPException(status_code=401, detail="Nemate ovlasti za pristup ovim podacima")
```

Sada možemo koristiti ovu funkciju kao ovisnost u svakoj ruti koja zahtjeva autorizaciju.

```
@app.get("/tajni_podaci")
def get_tajni_podaci(admin: Admin = Depends(provjeri_token)): # koristimo Depends funkciju
    za "ubrizgavanje ovisnosti"
    return {"tajni_podaci": "šifra za sef je 1234"}


@app.put("/tajni_podaci")
def update_tajni_podaci(podaci: dict, admin: Admin = Depends(provjeri_token)): # "podaci"
    su tijelo HTTP zahtjeva
    # ažuriramo podatke...
    print(f"Podatke ažurirao admin {admin.korisnicko_ime}")
    return {"poruka": "Podaci uspješno ažurirani"}


@app.delete("/tajni_podaci")
def delete_tajni_podaci(admin: Admin = Depends(provjeri_token)):
    # brišemo podatke...
    print(f"Podatke izbrisao admin {admin.korisnicko_ime}")
    return {"poruka": "Podaci uspješno obrisani"}
```

Naravno, **ovo je samo simulacija**, u pravom projektu moramo koristiti stvarnu bazu podataka, sa sigurnim mehanizmima za autentifikaciju i autorizaciju zahtjeva!

DI se često koristi za potrebe autorizacije i autentifikacije dolaznih zahtjeva te za dijeljenje konekcije na bazu podataka, međutim ima i mnoge druge svrhe o kojima možete više pročitati u FastAPI dokumentaciji na sljedećoj [poveznici](#).

Što se tiče implementacije sigurnosnih mehanizama, FastAPI nude gotove module za autentifikaciju i autorizaciju, kao što su `OAuth2PasswordBearer` i `OAuth2PasswordRequestForm`. Više o tome također možete pronaći u dokumentaciji na sljedećoj [poveznici](#).

4.2 API Router

Osim Dependency Injection sustava, FastAPI nudi i mogućnost strukturiranja koda kroz `APIRouter` klasu. Slično kao Express.Router u Express.js, `APIRouter` omogućuje grupiranje srodnih ruta i resursa u jednu cjelinu.

Različite rute je potrebno grupirati u odgovarajuće "podaplikacije" u zasebnim datotekama, unutar zajedničkog direktorija. Direktorij možemo nazvati `routers` ili `routes`.

```
mkdir routers
```

Kako bi naglasili da se radi o modulu, možemo dodati praznu `__init__.py` datoteku unutar direktorija.

```
touch routers/__init__.py
```

U direktoriju `routers` možemo kreirati zasebne datoteke za svaku grupu ruta. Primjerice, dodajemo rutu za korisnike:

```
# routers/korisnici.py
from fastapi import APIRouter

router = APIRouter() # router je podaplikacija koju instanciramo na isti način
```

Ili dodajemo rutu za knjige:

```
# routers/knjige.py
from fastapi import APIRouter

router = APIRouter()
```

Rute definiramo na identičan način kao i do sada, samo što ih grupiramo unutar `router` objekta.

```
# routers/korisnici.py
from fastapi import APIRouter

router = APIRouter()

@router.get("/korisnici")
def get_korisnici():
    return {"poruka": "Dohvaćeni korisnici"}

@router.post("/korisnici")
def create_korisnik():
    return {"poruka": "Korisnik uspješno kreiran"}
```

odnosno:

```
# routers/knjige.py

from fastapi import APIRouter

router = APIRouter()

@router.get("/knjige")
def get_knjige():
    return {"poruka": "Dohvaćene knjige"}

@router.post("/knjige")
def create_knjiga():
    return {"poruka": "Knjiga uspješno kreirana"}
```

Obzirom da sve rute počinju istim prefiksom (npr. `/korisnici` ili `/knjige`), možemo to naglasiti prilikom definicije `APIRouter` objekta. Tada je potrebno maknuti prefiks iz svake rute unutar datoteke.

```
# routers/korisnici.py

from fastapi import APIRouter

router = APIRouter(prefix="/korisnici")

@router.get("/") # ustvari je /korisnici/
def get_korisnici():
    return {"poruka": "Dohvaćeni korisnici"}

@router.post("/") # ustvari je /korisnici/
def create_korisnik():
    return {"poruka": "Korisnik uspješno kreiran"}

@router.get("/{id}") # ustvari je /korisnici/{id}
def get_korisnik(id: int):
    return {"poruka": f"Dohvaćen korisnik s ID-em {id}"}
```

Ove rute možemo učitati u glavnu aplikaciju koristeći `include_router` metodu.

```
# main.py

from fastapi import FastAPI
from routers.korisnici import router as korisnici_router # uključujemo router iz datoteke korisnici.py
from routers.knjige import router as knjige_router # uključujemo router iz datoteke knjige.py
app = FastAPI()

app.include_router(korisnici_router) # uključujemo rute za korisnike
app.include_router(knjige_router) # uključujemo rute za knjige

# nastavljamo dalje s definicijom rute na "main" razini
```

```
@app.get("/")
def home():
    return {"poruka": "Dobrodošli na FastAPI poslužitelj"}
```

Konačna struktura projekta sada izgleda ovako:

```
.
├── main.py
├── routers
│   ├── __init__.py
│   ├── korisnici.py
│   └── knjige.py
└── models.py
```

Ovako organizirani poslužitelj je čitljiviji, lakši za održavanje i skalabilan. Svaka grupa ruta je odvojena u zasebnoj datoteci, a svaka ruta je odvojena u zasebnoj funkciji.

Više o organizaciji koda u velikim aplikacijama možete pročitati u FastAPI dokumentaciji na sljedećoj [poveznicu](#).

4.3 Zadatak za vježbu: Razvoj mikroservisa za dohvaćanje podataka o filmovima

Implementirajte mikroservis za dohvaćanja podataka o filmovima koristeći FastAPI. Mikroservis treba biti organiziran u zasebnim datotekama unutar direktorija `routers` i `models`. Glavni resurs jesu filmovi, a podatke možete preuzeti u JSON obliku sa sljedeće [poveznice](#).

1. Implementirajte odgovarajuće Pydantic modele za filmove prema atributima koji se nalaze u JSON datoteci.
2. Za svaki atribut filma definirajte odgovarajuće polje u Pydantic modelu.
3. Učitajte filmove iz JSON datoteke i [odradite deserijalizaciju podataka](#), a zatim ih pohranite u *in-memory* listu filmova.
4. Dodajte provjere za sljedeće attribute filma unutar Pydantic modela za film:
 - `Images` mora biti lista stringova (javnih poveznica na slike)
 - `type` mora biti odabir između "movie" i "series"
 - Obavezni atributi su: `Title`, `Year`, `Rated`, `Runtime`, `Genre`, `Language`, `Country`, `Actors`, `Plot`, `Writer`
 - Ostali atributi su neobavezni, a ako nisu navedeni, postavite im zadanu vrijednost
 - Dodajte validacije za `Year` i `Runtime` atribut (godina mora biti veća od 1900, a trajanje filma mora biti veće od 0)
 - Dodajte validacije za `imdbRating` i `imdbVotes` (ocjena mora biti između 0 i 10, a broj glasova mora biti veći od 0)
5. Definirajte Pydantic model `Actor` koji će sadržavati attribute `name` i `surname`.
6. Definirajte Pydantic model `Writer` koji će sadržavati attribute `name` i `surname`.
7. Strukturirajte kod u zasebnim datotekama unutar direktorija `routers` i `models`. U direktoriju `routers` dodajte datoteku `filmovi.py` u kojoj ćete definirati rute za dohvaćanje svih filmova i pojedinog filma po `imdbID`-u i rutu za dohvaćanje filma prema naslovu (`Title`).
8. Za rutu koja dohvaća sve filmove, implementirajte mogućnost filtriranja filmova prema query parametrima: `min_year`, `max_year`, `min_rating`, `max_rating` te `type` (film ili serija). Implementirajte validaciju query parametra.
9. U glavnoj aplikaciji učitajte rute iz datoteke `filmovi.py` i uključite ih u glavnu FastAPI aplikaciju.
10. Dodajte iznimke (`HTTPException`) za slučaj kada korisnik pokuša dohvatiti film koji ne postoji u bazi podataka, po `imdbID`-u ili `Title`-u.
11. Testirajte aplikaciju koristeći generiranu interaktivnu dokumentaciju (Swagger ili ReDoc).

Rješenje učitajte na GitHub i predajte na Merlin, uz pripadajuće screenshotove dokumentacije koja se generira automatski na `/docs` ruti.

Nema univerzalnog rješenja za organizaciju koda i implementaciju API-ja, a zadaća nosi do 2 dodatna boda ovisno o kvaliteti izrade FastAPI mikroservisa.

Raspodijeljeni sustavi (RS)

Nositelj: doc. dr. sc. Nikola Tanković

Asistent: Luka Blašković, mag. inf.

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



#7

RS

Naučili smo kako definirati asinkrone mikroservise kroz razvojne okvire poput aiohttp i FastAPI. Jednom kad imamo robusne mikroservise, sljedeći korak je njihovo raspoređivanje i upravljanje resursima, bilo na lokalnom ili u proizvodnjkom okruženju. Kontejnerizacija predstavlja tehnologiju koja omogućuje doslovno pakiranje aplikacija i svih njenih ovisnosti u jednu samostalnu i lako-prenosivu cjelinu, tzv. kontejner (*eng. Container*). Kontejneri osiguravaju konzistentnost i predvidljivost ponašanja aplikacija u različitim okruženjima, smanjujući mogućnost grešaka. S druge strane, uravnotežavanje opterećenja (*eng. Load balancing*) osigurava ravnomjernu raspodjelu zahtjeva između više mikroservisa odnosno instanci kontejnera. Kombinacija ovih dvaju tehnologija omogućuje skaliranje i optimizaciju modernih softverskih rješenja u dinamičnim okruženjima, kao što je to računarstvo u oblaku (*eng. Cloud computing*).

Posljednje ažurirano: 23.1.2025.

Sadržaj

- [Raspodijeljeni sustavi \(RS\)](#)
- [\(7\) Kontejnerizacija i Load balancing](#)
 - [Sadržaj](#)
- [1. Kontejnerizacija mikroservisa \(Docker\)](#)
 - [1.1 Instalacija Docker platforme](#)
 - [1.2 Dockerfile](#)
 - [1.2.1 Osnovne naredbe u Dockerfileu](#)
 - [1.3 Kontejnerizacija osnovnog Python programa](#)
 - [1.4 Kontejnerizacija aiohttp mikroservisa](#)
 - [1.4.1 Mapiranje portova](#)
 - [1.5 Tablica osnovnih Dockerfile naredbi](#)

- [1.6 Tablica osnovnih Docker naredbi](#)
- [1.7 Kontejnerizacija FastAPI mikroservisa](#)
 - [1.7.1 Implementacija mikroservisa](#)
 - [1.7.2 Kontejnerizacija mikroservisa](#)
- [1.8 Zadaci za vježbu: Kontejnerizacija mikroservisa](#)
- [2. Docker Compose](#)
 - [2.1 Kako spakirati više mikroservisa u jednu cjelinu](#)
 - [2.1.1 Sintaksa `docker-compose.yml` datoteke](#)
 - [2.2 Interna komunikacija mikroservisa](#)
 - [2.3 Varijable okruženja u Dockeru](#)
 - [2.4 Zadaci za vježbu: Docker Compose](#)
- [3 Load balancing \(`nginx`\)](#)
 - [3.1 Horizontalno skaliranje koristeći samo Docker Compose](#)

1. Kontejnerizacija mikroservisa (Docker)

Docker je popularna platforma otvorenog koda koja se koristi za automatizaciju razvoja i isporuke koristeći tehnologiju kontejnerizacije (*eng. Containerization*). U računarstvu, kontejnerizacija predstavlja vrstu virtualizacije na razini operacijskog sustava koja omogućuje pokretanje i izvršavanje aplikacija u izoliranim okruženjima zvanim **kontejneri** (*eng. Container*).



Kontejner (*eng. Container*) je standardizirana, samostalna i izolirana softverska jedinica koja sadrži sve potrebne datoteke, biblioteke, konfiguracije i druge ovisnosti potrebne za pokretanje aplikacije. Kontejneri služe za brzo pakiranje i distribuciju aplikacija u različitim okruženjima, primjerice na razvojnom računalu, testnom poslužitelju ili proizvodnjkom sustavu, ili različitim operacijskim sustavima.

U usporedbi s virtualnim mašinama (*eng. Virtual Machine (VM)*), kontejneri su znatno memorijski efikasniji, brže se pokreću te su portabilni. Međutim, kontejneri pokrenuti na našim računalima (ili u Cloud okruženju) direktno ovise o operacijskom sustavu domaćina te dijele resurse s njim, što ne predstavlja potpuni izolacijski sloj kao kod virtualnih mašina koje imaju vlastiti operacijski sustav, programe, aplikacije itd.

Ipak, upravo ovo dijeljenje kernela operacijskog sustava domaćina omogućuje brže pokretanje i manju potrošnju resursa, što je čini idealnom tehnologijom za razvoj i isporuku mikroservisa.

1.1 Instalacija Docker platforme

Kako bi nastavili, potrebno je prvo instalirati Docker platformu koja dolazi s grafičkim korisničkim sučeljem (**Docker Desktop**) za sve operacijske sisteme.

- [Docker Desktop za Windows](#)
- [Docker Desktop za macOS](#)
- [Docker Desktop za Linux](#)

Ako ste na Windows OS-u, Docker Desktop zahtjeva instalaciju **WSL-2** (Windows Subsystem for Linux) koji se može instalirati preko PowerShell naredbe:

```
wsl --install
```

Dodatno, je potrebno omogućiti **virtualizaciju** za Windows računala.

Ovisno o proizvođaču maticne ploče, postupak se razlikuje, ali BIOS-u se obično pristupa pritiskom tipke **F2**, **F10**, **F12** ili **DEL** na samom pokretanju računala (**ovo se ne radi za macOS računala**).

Najbolji način je pretražiti na internetu kako pristupiti BIOS-u za vaš model računala. Nakon toga pratite upute na linku iznad, ovisno o operacijskom sustavu.

System requirements

Tip

Should I use Hyper-V or WSL?

Docker Desktop's functionality remains consistent on both WSL and Hyper-V, without a preference for either architecture. Hyper-V and WSL have their own advantages and disadvantages, depending on your specific set up and your planned use case.

WSL 2 backend, x86_64 Hyper-V backend, x86_64 WSL 2 backend, Arm (Beta)

- WSL version 1.1.3.0 or later.
- Windows 11 64-bit: Home or Pro version 22H2 or higher, or Enterprise or Education version 22H2 or higher.
- Windows 10 64-bit: Minimum required is Home or Pro 22H2 (build 19045) or higher, or Enterprise or Education 22H2 (build 19045) or higher.
- Turn on the WSL 2 feature on Windows. For detailed instructions, refer to the [Microsoft documentation](#).
- The following hardware prerequisites are required to successfully run WSL 2 on Windows 10 or Windows 11:
 - 64-bit processor with [Second Level Address Translation \(SLAT\)](#)
 - 4GB system RAM
 - Enable hardware virtualization in BIOS. For more information, see [Virtualization](#).

For more information on setting up WSL 2 with Docker Desktop, see [WSL](#).

Note

Docker only supports Docker Desktop on Windows for those versions of Windows that are still within [Microsoft's servicing timeline](#). Docker Desktop is not supported on server versions of Windows, such as Windows Server 2019 or Windows Server 2022. For more information on how to run containers on Windows Server, see [Microsoft's official documentation](#).

Important

To run Windows containers, you need Windows 10 or Windows 11 Professional or Enterprise edition. Windows Home or Education editions only allow you to run Linux containers.

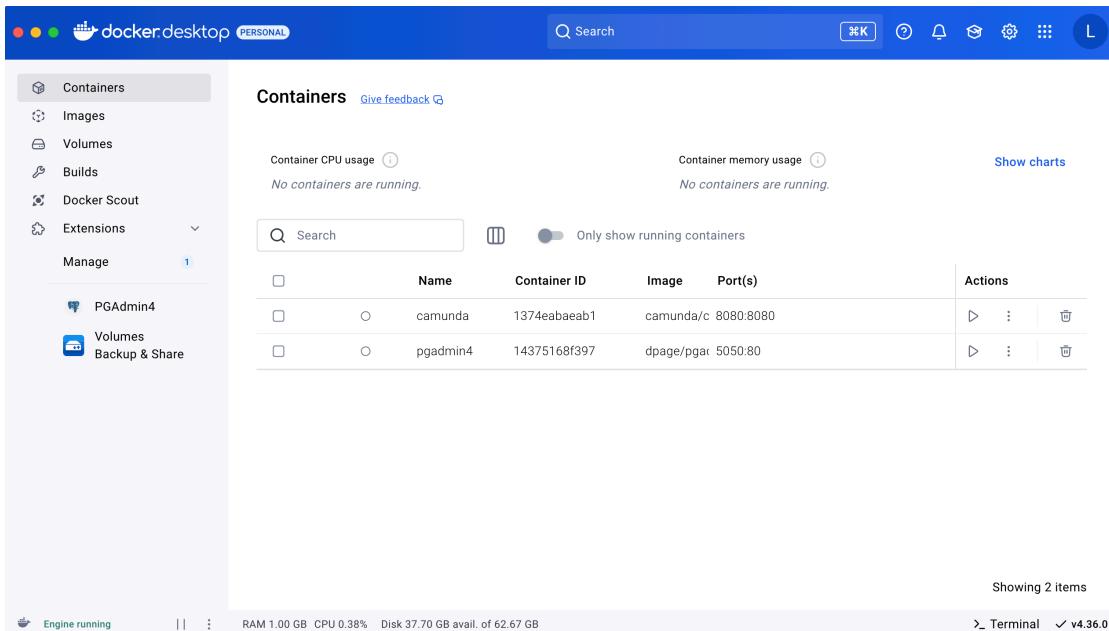
Na Windowsu je moguće koristiti **WSL** (Windows Subsystem for Linux) ili **Hyper-V** platformu za virtualizaciju, detaljne upute: <https://docs.docker.com/desktop/setup/install/windows-install/>

Docker je moguće koristiti i na **Linux** (dostupno za: Ubuntu, Debian, RHEL, Fedora) i **macOS** (dostupno za: Apple silicon, Intel chip) operacijskim sustavima bez dodatnih postavki. [Na Linuxu možete instalirati Docker i bez grafičkog sučelja preko terminala](#), međutim za početnike je preporuka instalirati grafičko sučelje - **Docker Desktop**.

Nakon što ste uspješno instalirati **Docker Desktop**, provjerite je li uspješno instaliran preko naredbe:

```
docker --version
```

Pokrenite Docker Desktop aplikaciju i prijavite se s vašim Docker računom. Ako nemate Docker račun, možete ga besplatno kreirati na [Docker Hub-u](#).



Grafičko sučelje Docker Desktop aplikacije

Grafičko sučelje Docker Desktop aplikacije sastoji se od nekoliko osnovnih elemenata:

- Container** - prikaz svih pokrenutih kontejnera (eng. *Docker container*). Docker Container je svaka instanca izgrađenog Docker predloška (*image*) koja se pokreće u izoliranom okruženju
- Images** - prikaz svih preuzetih Docker predložaka (eng. *Docker image*). Docker Image je nepromjenjivi predložak za definiranje i pokretanje kontejnera.
- Volumes** - prikaz svih Docker "volumena" (eng. *Docker volumes*). Docker Volume koristi se za trajno pohranjenje podataka, obzirom da se podaci unutar kontejnera brišu prilikom gašenja kontejnera.
- Builds** - prikaz svih provedenih Docker "buildova" (eng. *Docker build*). Ovdje su pohranjeni svi Docker buildovi koji su se izvršavali na vašem računalu.
- Docker Scout** - napredna analiza pohranjenih docker predložaka, u svrhu pronalaska potencijalnih ranjivosti (eng. *vulnerabilities*).
- Extensions** - dodatne ekstenzije za Docker Desktop aplikaciju. Za sada nam nisu potrebne.

U pravilu, za sada će nam najzanimljiviji biti `Container` i `Images` tabovi.

U nastavku ove skripte, za Docker Images neće se koristiti termin Docker "slika" već **predložak**.

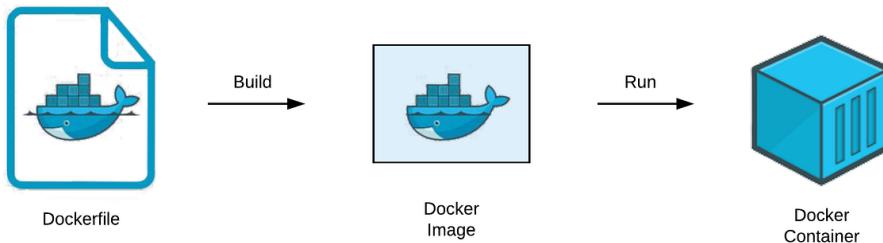
1.2 Dockerfile

Dockerfile je tekstualna datoteka koju koristimo za definiranje predložaka kontejnera. **Predložak** (Docker image) je ništa drugo nego **tekstualna datoteka koja se sastoji od niza naredbi** koje se izvršavaju prilikom izgradnje kontejnera.

`Dockerfile` može biti vrlo jednostavan, ali i vrlo složen, ovisno o mikroservisu kojeg definiramo, ali i o ovisnostima koje ima (npr. baza podataka, vanjski servisi, itd.).

U kontekstu ovog kolegija, mi ćemo naučiti kako definirati Dockerfileove za naše mikroservise, koje smo definirali u Pythonu, konkretno koristeći `FastAPI` i `aiohttp`, ali to može biti bilo koji drugi jezik ili tehnologija, ili oblik softverskog rješenja (ne mora biti API).

Upravo je to i **glavni cilj Docker platforme** - omogućiti jednostavno pakiranje i distribuciju bilo koje aplikacije, neovisno o njenim karakteristikama, ovisnostima ili tehnologijama koje koristi.



Dockerfile definira **predložak** kontejnera, a **kontejner** je instanca tog predloška koja se pokreće u izoliranom okruženju

Dockerfile definiramo **doslovnim nazivom datoteke**: `Dockerfile` (bez ekstenzije), a on se nalazi u korijenskom direktoriju mikroservisa.

Sintaksa *Dockerfile* naredbe:

```
# komentar
INSTRUCTION argument
```

- `INSTRUCTION` - naredba koja se izvršava prilikom izgradnje Docker predložka
- `argument` - argument naredbe

1.2.1 Osnovne naredbe u Dockerfileu

FROM

- Svrha: definira bazni predložak (tzv. **base image**) na kojem se gradi naš predložak
- Svaki Docker predložak mora početi s `FROM` naredbom, dakle to je prva naredba u Dockerfileu

```
FROM <image>:<tag>
```

Uobičajeno je koristiti službene verzije predložaka koje su dostupne na [Docker Hubu](#), konkretno za Python ih ima jako puno, npr. `python:3`:

```
# koristi Python 3 kao bazni predložak
FROM python:3
```

WORKDIR

- postavlja radni direktorij **unutar datotečnog sustava (eng. File system)** kontejnera
- sve naredbe nakon `WORKDIR` naredbe izvršavaju odnose se na taj direktorij, odnosno svi relativni putevi (*eng. path*) odnose se na taj direktorij

```
WORKDIR <path>
```

Primjer: postavljanje radnog direktorija na `/app` znači da će se sve naredbe koje slijede izvršavati unutar `/app` direktorija kontejnera.

```
# postavlja radni direktorij na /app
WORKDIR /app
```

COPY

- kopira datoteke i/ili direktorije **s računala domaćina** (eng. host) **u datotečni sustav kontejnera**
- naredba prima dva argumenta: `<src>` putanju do datoteke/direktorija na računalu domaćina i `<dest>` putanju do datoteke/direktorija unutar kontejnera
- ako želimo kopirati sve datoteke/direktorije iz trenutnog direktorija, možemo koristiti točku `.` kao `<src>`

```
# kopira datoteku app.py iz trenutnog direktorija (<src>) u destinacijski direktorij
kontejnera (<dest>
COPY <src> <dest>
```

Primjer: kopiranje ukupnog sadržaja iz trenutnog direktorija u `/app` direktorij kontejnera:

```
# kopira sve datoteke i direktorije iz trenutnog direktorija u /app direktorij kontejnera
COPY . /app
```

CMD

- definira **bilo koju naredbu** koja će se izvršiti **prilikom pokretanja kontejnera**
- može se koristiti **samo jednom** u Dockerfileu
- tipično se koristi za pokretanje aplikacije prilikom pokretanja kontejnera.
- naredba se **ne pokreće prilikom stvaranja predloška, već prilikom pokretanja kontejnera**

```
# pokreće aplikaciju prilikom pokretanja kontejnera
CMD ["executable", "arg1", "arg2"]
```

Primjer: pokretanje Python aplikacije `app.py` prilikom pokretanja kontejnera:

```
# pokreće Python aplikaciju prilikom pokretanja kontejnera
CMD ["python", "app.py"]
```

RUN

- izvršava naredbu **prilikom izgradnje Docker predloška**

- uobičajeno se koristi za instalaciju ovisnosti, konfiguraciju okruženja i sl.
- rezultati izvršene naredbe se pohranjuju u predložak, odnosno postaju dostupni prilikom pokretanja kontejnera
- u usporedbi s naredbom `CMD`, `RUN` se izvršava prilikom izgradnje predloška, dok se `CMD` izvršava prilikom pokretanja kontejnera

```
RUN <command>
```

Primjer: instalacija Python paketa `requests` prilikom izgradnje predloška:

```
# instalira Python paket requests prilikom izgradnje predloška
RUN pip install requests
```

EXPOSE

- služi za dokumentaciju porta na kojem će kontejner slušati u mreži.
- **neće otvoriti port na hostu**, već samo **dokumentira** koji port koristi kontejner

```
EXPOSE <port>
```

Primjer: dokumentiranje porta `8080`

```
# dokumentira port 8080 na kojem će kontejner slušati
EXPOSE 8080
```

Dakle, osnovne naredbe su `FROM`, `WORKDIR`, `COPY`, `CMD`, `RUN` i `EXPOSE`. Krenut ćemo s jednostavnim primjerima koji koriste samo ove naredbe.

1.3 Kontejnerizacija osnovnog Python programa

[Docker Hub](#) je servis koji omogućuje preuzimanje gotovih predložaka (**bazni predlošci**), ali i dijeljenje vlastitih. Na njemu možete pronaći veliki broj gotovih Docker predložaka koje možemo koristiti kao bazne (u svrhu definicije vlastitog predloška) ili kao gotove servise (npr. baze podataka, AI modele, mikroservise, desktop aplikacije ili bilo što drugo).

Međutim, mi ćemo koristiti osnovni Python 3 `Dockerfile` koji možemo jednostavno izgraditi kloniranjem `python:3` predloška.

Zamislimo da radimo na jednostavnom Python programu koji ispisuje "Hello, World!" poruku. Naš Python program `app.py` izgleda ovako:

```
# app.py
if __name__ == '__main__':
    print("Hello, World!")
```

Program pokrećemo jednostavno naredbom `python app.py` u terminalu.

Kako bi razumjeli kako Docker radi, prvo ćemo običnim tekstom napisati "niz naredbi" koji ćemo potom preslikati u odgovarajuće Docker naredbe.

1. Prvo kloniramo postojeći Python 3 predložak koji će biti predložak za naš kontejner.
2. Zatim definiramo radni direktorij unutar kontejnera gdje će se naša aplikacija pokrenuti, uobičajeno je to `/app`.
3. Kopiramo datoteku `app.py` s našeg računala u radni direktorij kontejnera.
4. Definiramo naredbu koja će se izvršiti prilikom pokretanja kontejnera, u našem slučaju to je `python app.py`.

Kreirajte novu datoteku `Dockerfile` u korijenskom direktoriju vašeg Python programa i unesite sljedeće naredbe koje preslikavaju tekst iznad:

```
# 1. Prvo kloniramo postojeći Python 3 predložak koji će biti predložak za naš kontejner.
FROM python:3

# 2. zatim definiramo radni direktorij unutar kontejnera gdje će se naša aplikacija
# pokrenuti, uobičajeno je to `/app`.

WORKDIR /app

# 3. Kopiramo datoteku `app.py` s našeg računala u radni direktorij kontejnera.

COPY app.py /app

# 4. Definiramo naredbu koja će se izvršiti prilikom pokretanja kontejnera, u našem
# slučaju to je `python app.py`.

CMD [ "python", "app.py" ]
```

Brisanjem komentara, `Dockerfile` svodimo na sljedeće:

```
FROM python:3
WORKDIR /app
COPY app.py /app
CMD ["python", "app.py"]
```

Struktura direktorija bi trebala izgledati ovako:

```
.
├── Dockerfile
└── app.py
```

`Dockerfile` dodajemo u korijenski direktorij našeg Python programa

Kako bismo **izgradili predložak** (eng. *build*) iz definiranog `Dockerfile`-a, koristimo naredbu `docker build -t <ime>:<verzija> .`:

- opcionalnom zastavicom `-t` možemo odrediti ime i verziju našeg predložka
- točka `.` označava trenutni direktorij gdje se nalazi `Dockerfile` (pazite da se u terminalu nalazite u direktoriju gdje se nalazi `Dockerfile`!)

```
cd /putanja/do/direktorija/sa/Dockerfileom
docker build -t hello-world:1.0 .
```

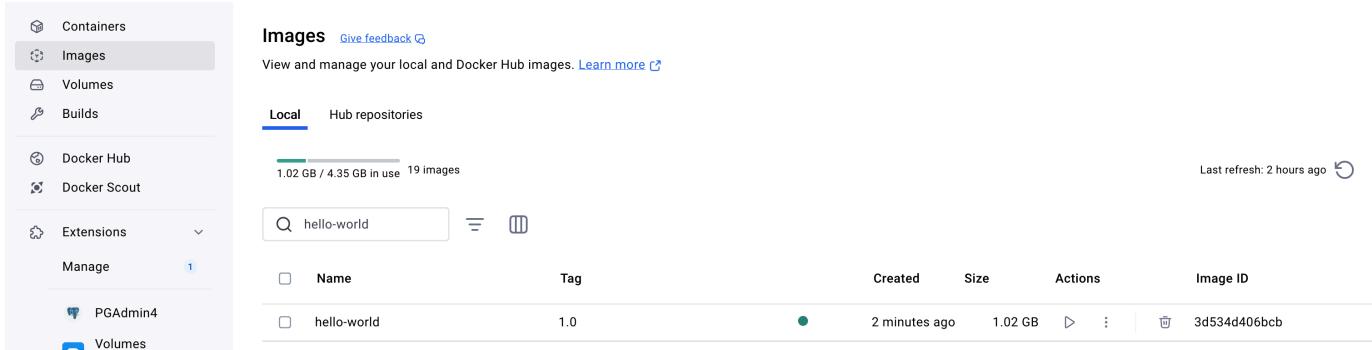
Čitaj: "izgradi Docker predložak s imenom `hello-world` i verzijom `1.0` na temelju `Dockerfile`-a iz trenutnog direktorija"

Ako dobijete grešku prilikom izgradnje: `ERROR: Cannot connect to the Docker daemon at unix:///Users/lukablaskovic/.docker/run/docker.sock. Is the docker daemon running?`, to znači da Docker daemon nije pokrenut. Pokrenite Docker Desktop aplikaciju i pokušajte ponovno.

Izgradnja Docker predložka potrajat će neko vrijeme budući da je prvi korak preuzimanje i priprema baznog predložka `python:3`.

- nakon što se izgradi predložak, možete ga vidjeti u Docker Desktop aplikaciji pod tabom `Images`.

Jednom kad je predložak izgrađen, otvorite **Docker Desktop** i provjerite je li vaš predložak uspješno izgrađen u tabu `Images`.



Vidimo da je predložak `hello-world:1.0` uspješno izgrađen i ima oko 1GB, to je zato što je bazni predložak `python:3` dosta velik!

Kontejner možemo pokrenuti odabirom `Actions -> Run` ili preko terminala naredbom `docker run <ime>:<verzija>`:

```
docker run hello-world:1.0
```

Napomena: Naredbu je moguće pokrenuti bilo kojem terminalu, ne samo u terminalu gdje se nalazite u direktoriju s `Dockerfile`-om.

Pokretanjem kontejnera trebali biste vidjeti ispis "Hello, World!" poruke u terminalu, odnosno u Docker Desktop aplikaciji u tabu `container`.

The screenshot shows the Docker Desktop application window. On the left, a sidebar menu is open with several options: Containers (selected), Images, Volumes, Builds, Docker Hub, Docker Scout, Extensions (with Manage and PGAdmin4 listed), and Volumes & Backup & Share. The main content area is titled 'Containers' and shows a table of running containers. The table has the following data:

Name	Container ID	Image	Port(s)	Actions
dreamy_montalcini	2d6efdc50d9b	hello-world:1.0		

Kontejner `hello-world:1.0` je uspješno pokrenut i ispisuje "Hello, World!" poruku

Pokretanjem kontejnera na ovaj način, Docker automatski dodjeljuje **naziv** i **ID kontejnera**.

1.4 Kontejnerizacija aiohttp mikroservisa

Na ovaj način možemo kontejnerizirati bilo koji Python program koji se sinkrono izvršava. Međutim, kako bismo kontejnerizirali asinkroni mikroservis, poput aiohttp mikroservisa koji smo imali priliku razvijati na prethodnim vježbama, **potrebit će malo drugačije pristupi izradi Dockerfilea**.

U ovom primjeru, kontejnerizirat ćemo jednostavan aiohttp mikroservis koji sadrži dva endpointa: `GET /proizvodi` i `POST /proizvodi`.

Napraviti ćemo novi direktorij `aiohttp-microservice` u kojem ćemo kreirati novi Python program `app.py` koji sadrži aiohttp mikroservis:

```
mkdir aiohttp-microservice
cd aiohttp-microservice
```

Budući da koristimo aiohttp, potrebno je instalirati ovaj paket u virtualno okruženje:

- navodimo verziju Pythona (3.11)

```
conda create -n aiohttp-microservice python=3.11
conda activate aiohttp-microservice
```

Instalirajte aiohttp paket:

```
pip install aiohttp
```

Mikroservis ćemo definirati u datoteci `app.py`:

- `GET /proizvodi` - vraća listu proizvoda
- `POST /proizvodi` - dodaje novi proizvod
- podaci su pohranjeni u listi `proizvodi`
- poslužitelj sluša na portu `8080`

```
import asyncio
from aiohttp import web

proizvodi = [
    {"id": 1, "naziv": "Laptop", "cijena": 1500},
    {"id": 2, "naziv": "Miš", "cijena": 20},
    {"id": 3, "naziv": "Tipkovnica", "cijena": 50},
    {"id": 4, "naziv": "Monitor", "cijena": 300},
    {"id": 5, "naziv": "Slušalice", "cijena": 100},
]

app = web.Application()

async def get_proizvodi(request):
    return web.json_response(proizvodi)
```

```

async def add_proizvod(request):
    data = await request.json()

    if data["naziv"] in [proizvod["naziv"] for proizvod in proizvodi]:
        return web.json_response({"error": "Proizvod već postoji!"}, status=400)

    proizvod = {
        "id": proizvodi[-1]["id"] + 1,
        "naziv": data['naziv'],
        "cijena": data['cijena']
    }
    proizvodi.append(proizvod)
    return web.json_response(proizvod)

app.router.add_routes([
    web.get('/proizvodi', get_proizvodi),
    web.post('/proizvodi', add_proizvod)
])

web.run_app(app, host='localhost', port=8080)

```

Napravite novu datoteku `Dockerfile` u korijenskom direktoriju `aiohttp-microservice`.

Sada je naš program je složeniji, imamo asinkroni mikroservis koji sluša na portu `8080`, stoga je potrebno definirati nekoliko dodatnih naredbi u Dockerfileu. Osim toga, imamo i ovisnost o `aiohttp` paketu, stoga je potrebno instalirati ovaj paket prilikom izgradnje predloška.

Moguće je iskoristiti naredbu `RUN` za instalaciju paketa, primjerice:

```
RUN pip install aiohttp
```

Međutim to nije uobičajeno raditi, obzirom da **stvarni mikroservisi imaju često puno više od jedne ovisnosti**. Uz to, na ovaj način ne navodimo direktno o kojoj se verziji biblioteke radi, što može dovesti do problema u budućnosti prilikom ažuriranja međuovisnosti paketa.

Bolja opcija je izlistati **sve ovisnosti** koje koristi naš mikroservis te ih instalirati jednom `RUN` naredbom.

Ovisnosti je uobičajeno definirati u posebnoj datoteci: `requirements.txt`

To možemo napraviti naredbom `pip freeze` koja će nam u terminal izlistati **sve pakete** koje koristi trenutno aktivno virtualno okruženje i **njihove verzije**:

```
aiohappyeyeballs==2.4.4
aiohttp==3.11.11
aiosignal==1.3.2
attrs==24.3.0
frozenlist==1.5.0
idna==3.10
multidict==6.1.0
propcache==0.2.1
setuptools==75.1.0
wheel==0.44.0
yarl==1.18.3
```

Možemo ih kopirati u ručno izrađenu datoteku `requirements.txt`, ili možemo koristiti naredbu `pip freeze > requirements.txt` koja će ih automatski zapisati u datoteku tog naziva.

Struktura direktorija bi trebala izgledati ovako:

```
.
├── Dockerfile
└── app.py
└── requirements.txt
```

Sada ćemo uzeti prethodni `Dockerfile` i prilagoditi ga za naš `aiohttp` mikroservis:

```
# Dockerfile za osnovni Python program
FROM python:3
WORKDIR /app
COPY app.py /app
CMD [ "python", "app.py" ]
```

1. korak je zamjena `python:3` baznog predloška s `python:3.11`, kako bi se poklapao s verzijom Pythona koju koristimo. Osim toga, možemo koristiti neki neku od službenih distribucija Pythona koje su memorijski efikasnije, npr. `python:3.11-slim`:

```
FROM python:3.11-slim
```

2. korak je postavljanje **radnog direktorija u kontejneru** na `/app`:

```
WORKDIR /app
```

3. Kako sad osim `app.py` imamo i `requirements.txt`, potrebno je kopirati oba u radni direktorij kontejnera. Za to smo rekli da koristimo `COPY` naredbu s točkom `.` za `<src>`

```
# kopiraj sve datoteke iz trenutnog direktorija u /app direktorij kontejnera
COPY . /app
```

4. Sada ćemo instalirati sve ovisnosti iz `requirements.txt` datoteke. To ćemo napraviti naredbom `RUN pip install -r requirements.txt`:

- kada ne bismo koristili zastavicu `-r`, `pip` bi pokušao instalirati paket `requirements.txt` iz PyPi repozitorija, što nije ono što želimo

```
# instaliraj sve ovisnosti iz requirements.txt datoteke
RUN pip install -r requirements.txt
```

5. Iako je već u servisu definiran port `8080`, dobra praksa je dokumentirati ga koristeći naredbu `EXPOSE`:

```
# dokumentiraj port 8080
EXPOSE 8080
```

6. Na kraju, definiramo naredbu koja se koristi za pokretanje mikroservisa, u ovom slučaju ista je kao i prije.

```
# pokreće Python aplikaciju prilikom pokretanja kontejnera
CMD ["python", "app.py"]
```

Konačni `Dockerfile` izgleda ovako:

```
FROM python:3.11-slim
WORKDIR /app
COPY . /app
RUN pip install -r requirements.txt
EXPOSE 8080
CMD ["python", "app.py"]
```

Navigirat ćemo u direktorij `aiohttp-microservice` i izgradit ćemo predložak `aiohttp-microservice:1.0`:

```
docker build -t aiohttp-microservice:1.0 .
```

U terminalu možete vidjeti kako se izgrađuje predložak u 4 koraka:

1. Preuzimanje baznog predloška `python:3.11-slim`
2. Postavljanje radnog direktorija na `/app`
3. Kopiranje datoteka iz trenutnog direktorija u kontejnerski `/app`
4. Instalacija ovisnosti iz `requirements.txt`

Docker desktop interface showing the Images section. Local repository contains 22 images, including 'aiohttp-microservice' version 1.0 (194.46 MB, created 2 minutes ago). Last refresh: 2 minutes ago.

Otvorite Docker desktop i provjerite je li predložak uspješno izgrađen.

Vidimo da je predložak `aiohttp-microservice:1.0` uspješno izgrađen i zauzima znatno manje memorije (~200MB) obzirom da smo koristili `slim` veziju za bazni predložak.

Kontejner možemo pokrenuti naredbom:

```
docker run aiohttp-microservice:1.0
```

i to radi!

Docker desktop interface showing the Containers section. Two containers are running: 'dreamy_montalcini' (hello-world:1.0) and 'vigorous_kilby' (aiohttp-microservice:1.0).

1.4.1 Mapiranje portova

Naredbom `docker ps` možemo vidjeti sve pokrenute kontejnere na našem računalu:

```
docker ps
```

Ispisuje **aktivne** kontejnere:

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
PORTS	NAMES			
a604911ac56a	aiohttp-microservice:1.0	"python app.py"	2 seconds ago	Up 2 seconds
8080/tcp	trusting_spence			

Oznake u ispisu:

- `CONTAINER ID` - jedinstveni identifikator kontejnera
- `IMAGE` - ime i verzija predloška
- `COMMAND` - naredba koja se izvršava prilikom pokretanja kontejnera (definirana u `CMD` naredbi)
- `CREATED` - vrijeme kada je kontejner pokrenut
- `STATUS` - status kontejnera (npr. `Up 2 seconds` znači da je kontejner pokrenut prije 2 sekunde)
- `PORTS` - portovi na kojima kontejner sluša
- `NAMES` - naziv kontejnera

Vidimo da je kontejner pokrenut i sluša na portu `8080`. Međutim, ako pokušamo pristupiti `localhost:8080/proizvodi` u web pregledniku ili kroz HTTP klijent pošaljemo zahtjev, dobit ćemo grešku povezivanja, što mislite zašto? 😕

► Spoiler alert! Odgovor na pitanje

U stupcu `PORTS` vidimo označku `8080/tcp`, što znači da je port `8080` otvoren (*eng. exposed*) unutar kontejnera, ali ne prema domaćinu (*eng. host*).

Mapiranje portova možemo obaviti pomoću zastavice `-p` u naredbi `docker run`:

Sintaksa:

```
docker run -p <host_port>:<container_port> <image>:<tag>
```

Nekoliko primjera da bude jasnije:

- ako mikroservis interno radi na portu `8080`, možemo ga mapirati na isti port domaćina (ako je slobodan):

```
docker run -p 8080:8080 aiohttp-microservice:1.0
```

- ako mikroservis interno radi na portu `8080`, a želimo ga mapirati na port `8083` domaćina:

```
docker run -p 8083:8080 aiohttp-microservice:1.0
```

- ako mikroservis interno radi na portu 4000, a želimo ga mapirati na port 3000 domaćina:

```
docker run -p 3000:4000 aiohttp-microservice:1.0
```

Zastavicom `--name` moguće je i dodijeliti ime kontejneru, kako ga Docker ne bi generirao nasumično:

```
docker run --name aiohttp-microservice -p 8080:8080 aiohttp-microservice:1.0
```

Redoslijed zastavica u ovom slučaju nije bitan, ali je dobra praksa prvo navesti zastavice za mapiranje portova, a zatim ime i verziju predloška:

```
docker run -p 8080:8080 --name aiohttp-microservice aiohttp-microservice:1.0
```

Kako je ovaj kontejner već pokrenut, možemo ga zaustaviti naredbom `docker stop <container_id_or_name>`:

```
docker stop a604911ac56a  
# ili  
docker stop aiohttp-microservice
```

Pokrenut ćemo kontejner s mapiranim portom i provjeriti stanje naredbom `docker ps`:

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
PORTE	NAMES			
702711364e85	aiohttp-microservice:1.0	"python app.py"	4 seconds ago	Up 4 seconds
0.0.0.0:8080->8080/tcp	aiohttp-microservice			

- `0.0.0.0:8080->8080/tcp` port je uspješno mapiran na port 8080 domaćina!

Praktično je koristiti Docker desktop sučelje budući da ono pamti kontejnere koje smo pokrenuli ili ugasili, **odnosno pamti parametre koje smo pritom koristili**. Tako možemo jednostavno ponovno pokrenuti kontejner klikom na `Actions -> Start` ili `Actions -> Restart`, na kontejneru gdje smo **već definirali mapiranje portova** u prvom pokretanju.



Pokretanje kontejnera s mapiranim portom iz Docker Desktop sučelja (tab `Containers`)

Ipak, stvari niti sada neće raditi! 😊

Ako otvorimo implementaciju mikroservisa, vidjet ćemo sljedeću naredbu za pokretanje:

```
web.run_app(app, host='localhost', port=8080)
```

- "slušaj na `localhost` hostu". `localhost` je ustvari *loopback* adresa mrežnog sučelja na računalu, a najčešće se asocira s IPv4 adresom `127.0.0.1`.
- port je `8080` i to je u redu.

Problem: mikroservis se pakira u kontejner, a kontejner je izolirano okruženje, odnosno **ne koristi mrežne postavke domaćina**. Prema tome, `localhost` u kontejneru se odnosi na sam kontejner, a ne na domaćinu!

Kada definiramo `localhost` kao *host*, mikroservis će prihvati samo zahtjeve koji dolaze iz samog kontejnera, a ne izvana.

Kako bismo definirali da mikroservis sluša na svim mrežnim sučeljima, **uključujući i domaćinu**, koristimo adresu `0.0.0.0`.

U produkcijskim okruženjima, ovo može biti sigurnosni rizik budući da mikroservis sluša na svim mrežnim sučeljima, ali za potrebe razvoja i testiranja, to je sasvim u redu.

Prema tome, izmijenit ćemo kod u mikroservisu:

```
web.run_app(app, host='0.0.0.0', port=8080) # zamijenili smo 'localhost' s '0.0.0.0'
```

Kontejner možemo izbrisati direktno u Docker Desktop aplikaciji ili naredbom `docker rm <container_id_or_name>`:

```
docker rm aiohttp-microservice
```

Nakon što izmjenimo kod mikroservisa, moramo **ponovno izraditi predložak** budući da je izmijenjen programski kod, a **Docker predložak je nepromjenjiv** - nije ga moguće izmjeniti nakon što je izgrađen.

Izgradimo ponovo predložak:

```
docker build -t aiohttp-microservice:1.0 .
```

Nakon što je predložak izgrađen, pokrenimo kontejner s mapiranim portom:

```
docker run -p 8080:8080 --name aiohttp-microservice aiohttp-microservice:1.0
```

Sada možemo poslati zahtjev na Docker kontejner s našeg računala koristeći `localhost:8080/proizvodi` u web pregledniku ili kroz HTTP klijent.

The screenshot shows the Postman interface with a successful GET request to `http://localhost:8080/proizvodi`. The response body is a JSON array containing five products:

```

1  [
2   {
3     "id": 1,
4     "naziv": "Laptop",
5     "cijena": 1500
6   },
7   {
8     "id": 2,
9     "naziv": "Miš",
10    "cijena": 20
11  },
12  {
13    "id": 3,
14    "naziv": "Tipkovnica",
15    "cijena": 50
16  },
17  {
18    "id": 4,
19    "naziv": "Monitor",
20    "cijena": 300
21  },
22  {
23    "id": 5,
24    "naziv": "Slušalice",
25    "cijena": 100
26  }

```

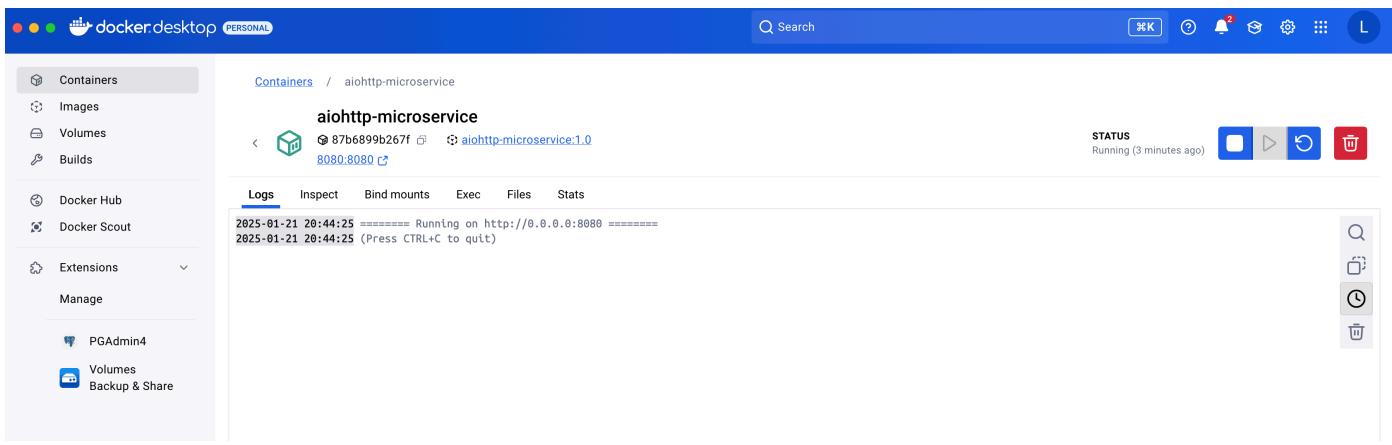
Poslali smo `GET /proizvodi` zahtjev na `localhost:8080` preko Postmana. Vidimo da kontejnerizirani mikroservis uspješno vraća listu proizvoda.

Detaljne mrežne postavke aktivnog Docker kontejnera možete provjeriti naredbom: `docker inspect <container_id_or_name>`:

```
docker inspect aiohttp-microservice
```

Osim toga, Docker Desktop pruža praktično sučelje za pregled drugih detalja aktivnog kontejnera, kao što su:

- logovi (terminal)
- detalji mrežnih postavki i druge informacije o kontejneru
- interni datotečni sustav kontejnera
- statistike o korištenju resursa

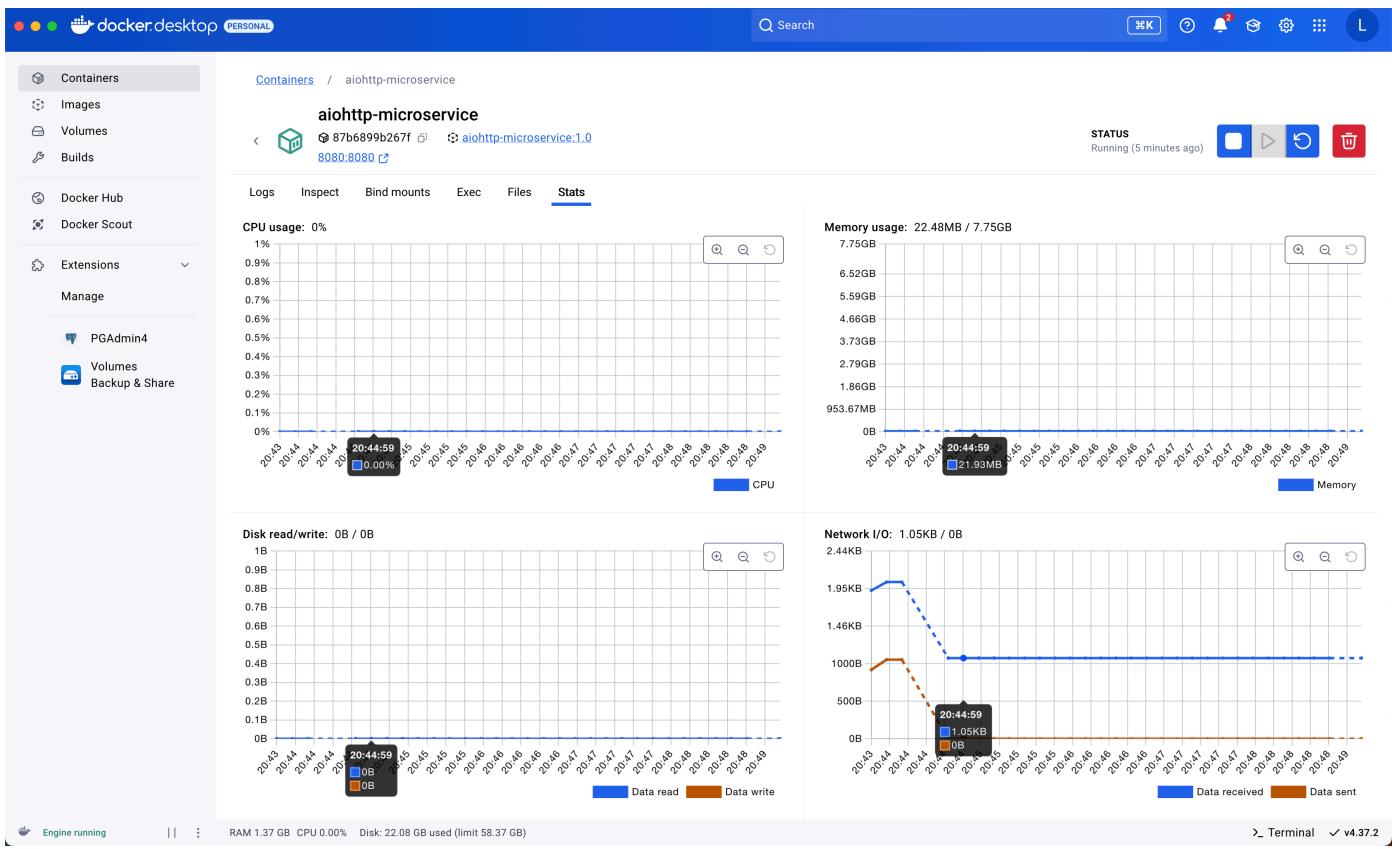


Pregled logova aktivnog kontejnera iz Docker Desktop sučelja

Name	Note	Size	Last modified	Mode
.dockerenv		0 Bytes	4 minutes ago	-rwxr-xr-x
app			4 minutes ago	drwxr-xr-x
app.py		945 Bytes	18 minutes ago	-rw-r--r--
Dockerfile		119 Bytes	46 minutes ago	-rw-r--r--
requirements.txt		181 Bytes	22 hours ago	-rw-r--r--
bin -> usr/bin		7 Bytes	9 days ago	Lrwxrwxrwx
boot			21 days ago	drwxr-xr-x
dev			3 minutes ago	drwxr-xr-x
etc			4 minutes ago	drwxr-xr-x
home			21 days ago	drwxr-xr-x
lib -> usr/lib		7 Bytes	9 days ago	Lrwxrwxrwx
media			9 days ago	drwxr-xr-x
mnt			9 days ago	drwxr-xr-x
opt			9 days ago	drwxr-xr-x
proc			3 minutes ago	dr-xr-xr-x
root			4 minutes ago	drwx-----
run			9 days ago	drwxr-xr-x
sbin -> usr/sbin		8 Bytes	9 days ago	Lrwxrwxrwx
srv			9 days ago	drwxr-xr-x
sys			3 minutes ago	dr-xr-xr-x
tmp			4 minutes ago	dtrwxrwxrwx

RAM 1.38 GB CPU 0.00% Disk: 22.08 GB used (limit 58.37 GB) > Terminal ✓ v4.37.2

Pregled internog datotečnog sustava aktivnog kontejnera iz Docker Desktop sučelja (uočite da je `app.py` datoteka unutar datoteke `/app` koju smo definirali naredbom `WORKDIR`)



Pregled statistika o korištenju resursa aktivnog kontejnera iz Docker Desktop sučelja

Iz statistika je moguće pratiti korištenje resursa kao što su **CPU, memorija, mreža i disk**.

Uočite da je kod graf-a **Network I/O** prikazan promet podataka u i iz kontejnera, a *spike* koji vidimo odnosi se na HTTP zahtjev koji smo poslali mikroservisu kroz Postman malo ranije.

1.5 Tablica osnovnih Dockerfile naredbi

U nastavku je tablica osnovnih **Dockerfile** naredbi s primjerima i sintaksom, koje smo naučili u ovom poglavlju za definiranje **Docker predložaka**:

Naredba	Sintaksa	Objašnjenje	Primjer
FROM	<code>FROM <image>: <tag></code>	Definira bazni predložak koji će se koristiti za definiciju vlastitog	<code>FROM ubuntu:20.04</code>
WORKDIR	<code>WORKDIR <path></code>	Postavlja radni direktorij unutar kontejnera	<code>WORKDIR /app</code>
COPY	<code>COPY <src> <dest></code>	Kopira datoteke ili direktorije s domaćina u datotečni sustav kontejnera.	<code>COPY . /app</code>
CMD	<code>CMD ["executable", "arg1"]</code>	Definira bilo koju naredbu koja će se izvršiti prilikom pokretanja kontejnera	<code>CMD ["python", "app.py"]</code>
		Izvršava bilo koju naredbu koja se	<code>RUN apt-get update &&</code>

RUN	<code>RUN <command></code>	poziva za vrijeme izgradnje Docker predloška	<code>apt-get install -y python3</code>
EXPOSE	<code>EXPOSE <port></code>	Deklarira portove koje će kontejner koristiti.	<code>EXPOSE 8080</code>

1.6 Tablica osnovnih Docker naredbi

U nastavku je tablica osnovnih Docker naredbi s primjerima i sintaksom, koje smo naučili u ovom poglavlju za **izgradnju predložaka i upravljanje kontejnerima**.

Naredba	Sintaksa	Objašnjenje	Primjer
build	<code>docker build -t <image_name>:<tag> <path></code>	Kreira Docker predložak iz <code>Dockerfile</code> -a i dodjeljuje mu ime i tag (opcionalno).	<code>docker build -t myapp:1.0 .</code>
run	<code>docker run -p <host_port>: <container_port> --name <container_name> <image></code>	Pokreće kontejner na temelju Docker predloška, mapira portove (<code>-p</code>) i daje ime (<code>--name</code>) kontejneru.	<code>docker run -p 8080:80 --name mycontainer myapp</code>
docker ps	<code>docker ps</code>	Prikazuje listu trenutno aktivnih kontejnera.	<code>docker ps</code>
docker inspect	<code>docker inspect <container_id_or_name></code>	Prikazuje detaljne informacije o određenom kontejneru ili image-u.	<code>docker inspect mycontainer</code>
docker rm	<code>docker rm <container_id_or_name></code>	Briše zaustavljeni kontejner.	<code>docker rm mycontainer</code>
docker stop	<code>docker stop <container_id_or_name></code>	Zaustavlja aktivni kontejner.	<code>docker stop mycontainer</code>
docker start	<code>docker start <container_id_or_name></code>	Pokreće zaustavljeni kontejner.	<code>docker start mycontainer</code>
docker logs	<code>docker logs <container_id_or_name></code>	Prikazuje logove aktivnog kontejnera.	<code>docker logs mycontainer</code>

1.7 Kontejnerizacija FastAPI mikroservisa

Pokazat ćemo kako kontejnerizirati i nešto složenije mikroservise, poput `FastAPI` mikroservisa sa svim njegovim ovisnostima. Kod `aiohttp`-a proces je bio jednostavniji jer nam je jedina ovisnost bila `aiohttp` paket, dok su drugi uključeni standardnu biblioteku Pythona (npr. `asyncio`).

`FastAPI` mikroservis je složeniji jer koristi više ovisnosti, poput `uvicorn` poslužitelja, `pydantic` za validaciju podataka, `SQLAlchemy` ako radite s relacijskom bazom podataka, itd. Osim toga, dobro razvijeni `FastAPI` poslužitelj gotovo uvijek sadrži strukturirani kod s više datoteka, što znači da je potrebno kopirati više datoteka u kontejner.

1.7.1 Implementacija mikroservisa

Definirat ćemo `FastAPI` mikroservis koji vraća podatke o vremenu preko otvorenog API-ja **Državnog hidrometeorološkog zavoda** (DHMZ).

DHMZ nudi besplatan API za pristup meteorološkim podacima koji su pohranjeni u XML formatu, jedini uvjet korištenja je obavezno navođenje DHMZ-a kao izvora korištenih podataka. Odlučili smo koristiti DHMZ API i napraviti moderni `FastAPI` mikroservis budući da DHMZ API vraća podatke u XML formatu, što je pomalo nečitljivo i danas se sve rjeđe koristi.

Podaci su javno dostupni na sljedećoj poveznici: https://meteo.hr/proizvodi.php?section=podaci¶m=xml_korisnici

Uzet ćemo podatke o `Prognozi` za `Hrvatska/Zagreb sutra`, koji su dostupni na:
https://prognoza.hr/prognoza_sutra.xml

Struktura XML-a slična je JSON strukturi, ali se umjesto `{}` koriste `<>` zagrade za definiranje početnog i završnog elementa, nalik HTML-u.

XML sadrži `metadata` podatke koji pokazuju datum i vrijeme kada su podaci izrađeni:

```
<metadata>
<datatime>210125</datatime>
<creationtime>Tue Jan 21 00:00:00 2025</creationtime>
</metadata>
```

Te podatke unutar `section` elementa, gdje je svako mjerjenje definirano elementom `station`:

```
<section name="All">
<param name="datum" value="220125"/>

<station name="sredisnja" lon="16.03" lat="45.82">
<param name="vrijeme" value="4"/>
<param name="Tmn" value="-1"/>
<param name="Tmx" value="4"/>
<param name="wind" value="6"/>
</station>

<station name="istocna" lon="18.63" lat="45.53">
```

```

<param name="vrijeme" value="6"/>
<param name="Tmn" value="-1"/>
<param name="Tmx" value="3"/>
<param name="wind" value="0"/>
</station>

<station name="gorska" lon="15.37" lat="44.55">
<param name="vrijeme" value="6"/>
<param name="Tmn" value="0"/>
<param name="Tmx" value="5"/>
<param name="wind" value="6"/>
</station>

<station name="unutrasnjost Dalmacije" lon="16.2" lat="44.03">xml
<param name="vrijeme" value="6"/>
<param name="Tmn" value="4"/>
<param name="Tmx" value="10"/>
<param name="wind" value="0"/>
</station>

</section>

```

Prvi korak je izrada direktorija i virtualnog okruženja:

```

mkdir weather-fastapi
cd weather-fastapi

conda create -n weather-fastapi python=3.11
conda activate weather-fastapi

```

Instalirat ćemo `FastAPI` s opcijom `[standard]`:

- pazite na navodne znakove

```
pip install "fastapi[standard]"
```

U datoteku `main.py` dodajemo osnovni kod za pokretanje:

```

# main.py
from fastapi import FastAPI
app = FastAPI()

@app.get("/")
def read_root():
    return {"message": "Hello, weather API!"}

```

Ako pogledate XML podatke, uočite da svaki `station` element ima sljedeće atribute:

- `name` - ime mjernog mjesto
- `lon` - geografska dužina

- `lat` - geografska širina
- `vrijeme` - prognoza vremena (npr. 4 - oblačno, 6 - sunčano)
- `Tmn` - minimalna temperatura
- `Tmx` - maksimalna temperatura
- `wind` - stupanjska jačina vjetra

Recimo da nas zanimaju samo podaci o nazivu mjesta, b i **maksimalnoj temperaturi, prognozi i jačini vjetra**.

Definirat ćemo Pydantic model `vrijeme` koji predstavlja te podatke:

```
# models.py

from pydantic import BaseModel

class Vrijeme(BaseModel):
    mjesto: str
    temperatura_min: int
    temperatura_max: int
    vjetar: int
```

Definirat ćemo endpoint `GET /vrijeme` koji će vraćati podatke o vremenu:

Povratna vrijednost endpointa je lista `Vrijeme` objekata:

```
# main.py
from models import Vrijeme

@app.get("/vrijeme", response_model = list[Vrijeme])
async def get_vrijeme():
    pass
```

Potrebno je slati HTTP zahtjev na `https://prognoza.hr/prognoza_sutra.xml` i parsirati XML podatke u `Vrijeme` objekte.

Za slanje zahtjeva možemo koristiti sinkronu biblioteku `requests` ili još bolje, ono što smo već naučili - asinkronu biblioteku `aiohttp`.

Instalirajmo `aiohttp` paket:

```
pip install aiohttp
```

Moramo otvoriti `clientSession` gdje ćemo slati `GET` zahtjev na URL `https://prognoza.hr/prognoza_sutra.xml`:

```
# main.py
from fastapi import FastAPI, HTTPException
from models import Vrijeme
import aiohttp
```

```

app = FastAPI()

@app.get("/vrijeme", response_model = list[Vrijeme])
async def get_vrijeme():
    url = "https://prognoza.hr/prognoza_sutra.xml"

    async with aiohttp.ClientSession() as session:
        response = await session.get(url)
        if response.status != 200: # u slučaju greške
            raise HTTPException(status_code=response.status, detail="Greška u dohvaćanju XML
podataka s DHMZ API-ja")
        xml_data = await response.text()

```

Možemo omotati kod u `try-except` blok kako bismo uhvatili eventualne greške prilikom slanja zahtjeva:

```

# main.py
from fastapi import status
try:
    async with aiohttp.ClientSession() as session:
        response = await session.get(url)
        if response.status != 200: # u slučaju greške
            raise HTTPException(status_code=response.status, detail="Greška u dohvaćanju XML
podataka s DHMZ API-ja")
        xml_data = await response.text()
except Exception as e: # Uhvati sve greške ako dođe do problema u slanju zahtjeva
    raise HTTPException(status_code=status.HTTP_500_INTERNAL_SERVER_ERROR, detail="Greška
u slanju HTML zahtjeva na DHMZ API")

```

Ako isprintamo `xml_data`, trebali bi dobiti XML podatke u terminalu.

Za samo parsiranje XML-a u Python objekte, možemo koristiti modul iz paketa `xml` - `xml.etree.ElementTree`.

```

# main.py
import xml.etree.ElementTree as ET

```

Pronaći ćemo sve oznake `station`, iterirati ih, te za svaku izvući podatke o `name`, `Tmn`, `Tmx` i `wind`:

```

# main.py
from fastapi import status
try:
    async with aiohttp.ClientSession() as session:
        response = await session.get(url)
        if response.status != 200: # u slučaju greške
            raise HTTPException(status_code=response.status, detail="Greška u dohvaćanju XML
podataka s DHMZ API-ja")
        xml_data = await response.text()
except Exception as e: # Uhvati sve greške ako dođe do problema u slanju zahtjeva

```

```

        raise HTTPException(status_code=status.HTTP_500_INTERNAL_SERVER_ERROR, detail="Greška
u slanju HTML zahtjeva na DHMZ API")

root = ET.fromstring(xml_data)
stations = root.findall("./station")
weather_list = []

for station in stations: # iteriraj kroz sve station elemente i izvuci podatke
    mjesto = station.attrib.get("name")
    temperatura_min = int(station.find("./param[@name='Tmn']").attrib.get("value"))
    temperatura_max = int(station.find("./param[@name='Tmx']").attrib.get("value"))
    vjetar = int(station.find("./param[@name='wind']").attrib.get("value"))

```

- nakon toga ćemo u listu dodati `Vrijeme` objekte koje definiramo dohvaćenim podacima

```

# main.py
@app.get("/vrijeme", response_model = list[Vrijeme])
async def get_vrijeme():
    """
    Dohvaća podatke o vremenu sa DHMZ API-ja, ali u JSON-u!

    Podaci dostupni na https://prognoza.hr/prognoza_sutra.xml
    """

    url = "https://prognoza.hr/prognoza_sutra.xml"

    try:
        async with aiohttp.ClientSession() as session:
            response = await session.get(url)
            if response.status != 200: # u slučaju greške
                raise HTTPException(status_code=response.status, detail="Greška u dohvaćanju XML
podataka s DHMZ API-ja")
            xml_data = await response.text()
    except Exception as e: # Uhvati sve greške ako dođe do problema u slanju zahtjeva
        raise HTTPException(status_code=status.HTTP_500_INTERNAL_SERVER_ERROR, detail="Greška
u slanju HTML zahtjeva na DHMZ API")

    root = ET.fromstring(xml_data)
    stations = root.findall("./station")
    weather_list = []

    for station in stations:
        mjesto = station.attrib.get("name")
        temperatura_min = int(station.find("./param[@name='Tmn']").attrib.get("value"))
        temperatura_max = int(station.find("./param[@name='Tmx']").attrib.get("value"))
        vjetar = int(station.find("./param[@name='wind']").attrib.get("value"))
        weather_list.append(Vrijeme(
            mjesto=mjesto,
            temperatura_min=temperatura_min,
            temperatura_max=temperatura_max,
            vjetar=vjetar

```

```
)  
return weather_list
```

Otvorite dokumentaciju mikroservisa na `http://localhost:8000/docs` i provjerite radi li sve kako treba, trebali biste vidjeti dokumentiranu rutu `/vrijeme` koja vraća podatke o vremenu u JSON formatu.

default

GET /vrijeme Get Vrijeme

Dohvaća podatke o vremenu sa DHMZ API-ja, ali u JSON-u!
Podaci dostupni na https://prognoza.hr/prognoza_sutra.xml

Parameters

No parameters

Responses

Curl

```
curl -X 'GET' \
'http://127.0.0.1:8000/vrijeme' \
-H 'accept: application/json'
```

Request URL

```
http://127.0.0.1:8000/vrijeme
```

Server response

Code	Details
200	Response body [{ "mjesto": "sredisnja", "temperatura_min": -1, "temperatura_max": 4, "vjetar": 6 }, { "mjesto": "istocna", "temperatura_min": -1, "temperatura_max": 3, "vjetar": 0 }, { "mjesto": "jugo", "temperatura_min": -1, "temperatura_max": 3, "vjetar": 0 }]

Tu ćemo stati, jer ovo nam je dovoljno složeno za pokazati kako kontejnerizirati mikroservis s više ovisnosti i strukturiranim kodom.

1.7.2 Kontejnerizacija mikroservisa

Prvi korak je izrada `requirements.txt` datoteke gdje ćemo pohraniti sve ovisnosti:

```
pip freeze > requirements.txt
```

Vidimo da `FastAPI` ima puno više ovisnosti od `aiohttp` mikroservisa:

```
aiohappyeyeballs==2.4.4
aiohttp==3.11.11
aiosignal==1.3.2
annotated-types==0.7.0
anyio==4.8.0
attrs==24.3.0
certifi==2024.12.14
click==8.1.8
dnspython==2.7.0
email_validator==2.2.0
fastapi==0.115.6
fastapi-cli==0.0.7
frozenlist==1.5.0
h11==0.14.0
httpcore==1.0.7
httptools==0.6.4
httpx==0.28.1
idna==3.10
Jinja2==3.1.5
markdown-it-py==3.0.0
MarkupSafe==3.0.2
mdurl==0.1.2
multidict==6.1.0
propcache==0.2.1
pydantic==2.10.5
pydantic_core==2.27.2
Pygments==2.19.1
python-dotenv==1.0.1
python-multipart==0.0.20
PyYAML==6.0.2
rich==13.9.4
rich-toolkit==0.13.2
shellingham==1.5.4
sniffio==1.3.1
starlette==0.41.3
typer==0.15.1
typing_extensions==4.12.2
uvicorn==0.34.0
uvloop==0.21.0
watchfiles==1.0.4
websockets==14.2
yarl==1.18.3
```

Napraviti ćemo `Dockerfile` u direktoriju mikroservisa, struktura direktorija treba izgledati ovako:

```
weather-fastapi/
├── main.py
├── models.py
└── requirements.txt
└── Dockerfile
```

Prvo ćemo uzeti prethodni `Dockerfile` za `aiohttp` mikroservisa, a zatim ga malo prilagoditi:

```
FROM python:3.11-slim
WORKDIR /app
COPY . /app
RUN pip install -r requirements.txt
EXPOSE 8080
CMD ["python", "app.py"]
```

- `FROM python:3.11-slim` - OK
- `WORKDIR /app` - OK
- `COPY . /app` - OK
- `RUN pip install -r requirements.txt` - OK

FastAPI u pravilu radi na portu `8000`, a za pokretanje koristi `uvicorn` poslužitelj. Moramo izmijeniti `EXPOSE` i `CMD` naredbe i ručno pokrenuti poslužitelj i definirati port.

```
EXPOSE 8000
```

Naredba za pokretanje je: `uvicorn main:app`, međutim ako bismo dodali zastavice u `CMD` naredbu, moramo ih odvojiti zarezom, a ne razmakom:

Sintaksa:

```
CMD[naredba, argument1, argument2, ...]
```

odnosno:

```
CMD["neka_naredba", "--argument1", "--argument2", ...]
```

U našem slučaju, definirati ćemo `host` na `0.0.0.0` kao i kod `aiohttp` mikroservisa, a port postaviti na `8000`:

```
CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"]
```

Konačni `Dockerfile` izgleda ovako:

```
FROM python:3.11-slim
WORKDIR /app
COPY . /app
RUN pip install -r requirements.txt
EXPOSE 8000
CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"]
```

Izgradite predložak naredbom `docker build`

- pazite da se nalazite u točnom direktoriju!

```
docker build -t weather-fastapi:1.0 .
```

Pokrenut ćemo kontejner s mapiranim portom:

```
docker run -p 8000:8000 --name weather-fastapi weather-fastapi:1.0
```

```
lukablaskovic ~ docker run -p 8000:8000 --name weather-fastapi weather-fastapi:1.0
INFO:     Started server process [1]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
INFO:     Uvicorn running on http://0.0.0.0:8000 (Press CTRL+C to quit)
```

Pokrenut `FastAPI` mikroservis u globalnom terminalu u obliku Docker kontejnera

To je to! Ako otvorimo web preglednik i posjetimo `localhost:8000/docs`, trebali bismo vidjeti dokumentaciju mikroservisa.

1.8 Zadaci za vježbu: Kontejnerizacija mikroservisa

1. Definirajte jednostavni `aiohttp` mikroservis `authAPI` koji će slušati na portu `9000`. Mikroservis pohranjuje *in-memory* podatke o korisnicima, s hashiranim lozinkama. U komentarima pored svakog zapisa možete pronaći stvarnu lozinku koja je korištena za generiranje hash vrijednosti funkcijom `hash_data`.

```

import hashlib

korisnici = [
    {"korisnicko_ime": "admin", "lozinka_hash" :
     "8d43d8eb44484414d61a18659b443fbfe52399510da4689d5352bd9631c6c51b"}, # lozinka =
     "lozinka123"
    {"korisnicko_ime": "markoMaric", "lozinka_hash" :
     "5493c883d2b943587ea09ab8244de7a0a88d331a1da9db8498d301ca315d74fa"}, # lozinka =
     "markoKralj123"
    {"korisnicko_ime": "ivanHorvat", "lozinka_hash" :
     "a31d1897eb84d8a6952f2c758cdc72e240e6d6d752b33f23d15fd9a53ae7c302"}, # lozinka =
     "1111111111lozinka_123"
    {"korisnicko_ime": "Nada000",
     "lozinka_hash": "492f3f38d6b5d3ca859514e250e25ba65935bcdd9f4f40c124b773fe536fee7d"} # lozinka =
     "blablabla"
]

def hash_data(data: str) -> str:
    return hashlib.sha256(data.encode()).hexdigest()

```

- implementirajte rutu `POST /register` koja dodaje novog korisnika u listu korisnika. Pohranite samo hashiranu lozinku korisnika.
- implementirajte rutu `POST /login` koja pronalazi korisnika po korisničkom imenu u listi korisnika i provjerava je li unesena lozinka u tijelu HTTP zahtjeva ispravna, odnosno podudaraju li se hash vrijednosti. Ako se pokuša prijaviti korisnik koji ne postoji, vratite odgovarajući statusni kod i poruku. Ako se lozinke ne podudaraju, vratite odgovarajući statusni kod i poruku.
- definirajte `Dockerfile` za `authAPI` mikroservis i pokrenite ga u Docker kontejneru. Servis treba slušati na portu `9000` domaćina.

2. Definirajte `FastAPI` mikroservis `socialAPI` koji će služiti za dohvaćanje izmišljenih objava na društvenoj mreži. Objave su pohranjene u listi rječnika, gdje svaki rječnik predstavlja jednu objavu. Svaka objava ima sljedeće atribute:

- `id` - jedinstveni identifikator objave (integer)
- `korisnik` - korisničko ime autora objave (do 20 znakova)
- `tekst` - tekst objave (do 280 znakova)
- `vrijeme` - vrijeme kada je objava napravljena (`timestamp`)
- definirajte odgovarajuće Pydantic modele za izradu nove objave i dohvaćanje objave.
- implementirajte rutu `POST /objava` koja dodaje novu objavu u listu objava. Prije dodavanja u listu, obavezno validirajte ulazne podatke. Prilikom dodavanja objave, sve vrijednosti su obavezne, osim `id` atributa koji se automatski dodjeljuje. Logiku dodjeljivanja jedinstvenog identifikatora možete implementirati sami po želji.
- implementirajte rutu `GET /objava/{id}` koja dohvaća objavu po jedinstvenom identifikatoru.

- implementirajte rutu `GET /korisnici/{korisnik}/objave` koja dohvaća sve objave korisnika s određenim korisničkim imenom.
- definirajte `Dockerfile` za `socialAPI` mikroservis i pokrenite ga u Docker kontejneru. Servis treba slušati na portu `3500` domaćina.

2. Docker Compose

Docker Compose je alat koji omogućuje definiranje i pokretanje **više kontejnera kao cjeline** pomoću samo jedne konfiguracijske datoteke.

Prednost ovog alata je što značajno pojednostavljuje *multi-container* aplikacije, jer omogućuje definiranje svih kontejnera, mreže, volumena i drugih resursa unutar jedne datoteke. Bez obzira na to, svaki kontejner je i dalje izolirano okruženje.

Na ovaj način možemo praktično definirati složene raspodijeljene sustave koji se sastoje od više mikroservisa, baza podataka i drugih posrednika.

Datoteka koju koristi Docker Compose za definiranje kontejnera i drugih resursa naziva se `docker-compose.yml`.

Primjer 1: Raspodijeljeni sustav za e-trgovinu s tri mikroservisa, frontendom i bazom podataka:

- `frontend` Docker kontejner s frontend aplikacijom (npr. Vue.js)
- `backend` Docker kontejner s backend aplikacijom (npr. FastAPI) koji je posrednik između cjelokupnog sustava
- `paymentAPI` Docker kontejner s mikroservisom za plaćanje
- `accountingAPI` Docker kontejner s mikroservisom za računovodstvo
- `database` Docker kontejner s bazom podataka (npr. PostgreSQL)

Primjer 2: Raspodijeljeni sustav za analizu podataka s tri mikroservisa i bazom podataka:

- `dataAPI` Docker kontejner s mikroservisom za dohvaćanje podataka
- `analysisAPI` Docker kontejner s mikroservisom za analizu podataka
- `visualizationAPI` Docker kontejner s mikroservisom za vizualizaciju podataka
- `database` Docker kontejner s bazom podataka (npr. MongoDB)

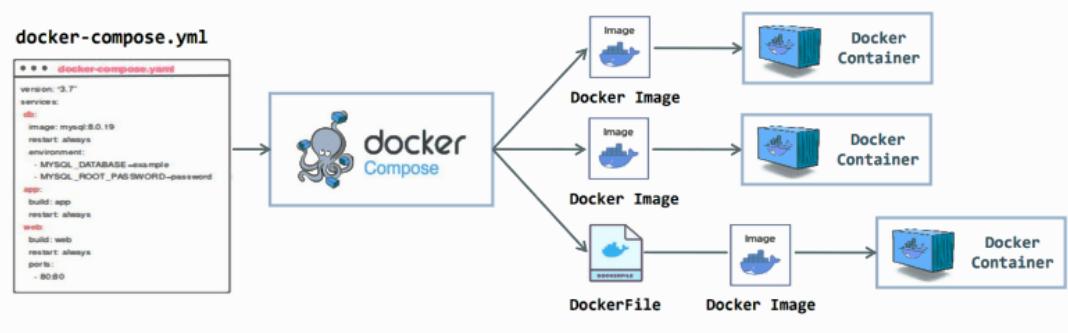
Primjer 3: Raspodijeljeni sustav za sustav za pohranu i dijeljenje datoteka koji se sastoji od četiri mikroservisa i baze podataka:

- `fileAPI` Docker kontejner s mikroservisom za pohranu i dijeljenje datoteka
- `encryptionAPI` Docker kontejner s mikroservisom za enkripciju i dekripciju datoteka
- `userAPI` Docker kontejner s mikroservisom za upravljanje korisnicima
- `notificationAPI` Docker kontejner s mikroservisom za obavijesti
- `database` Docker kontejner s bazom podataka (npr. MySQL)

Uočite zajedničke termine u svim ovim primjerima: to su **raspodijeljeni sustav, mikroservisi i docker kontejner**.

U mikroservisnoj arhitekturi, granularnost je ključna. Svaki mikroservis trebao bi obavljati jednu specifičnu funkciju, ili nekoliko srodnih funkcija. Mikroservis u ovom kontekstu može biti bilo koja aplikacija, a mi smo sad vidjeli kako to definirati različite API mikroservise.

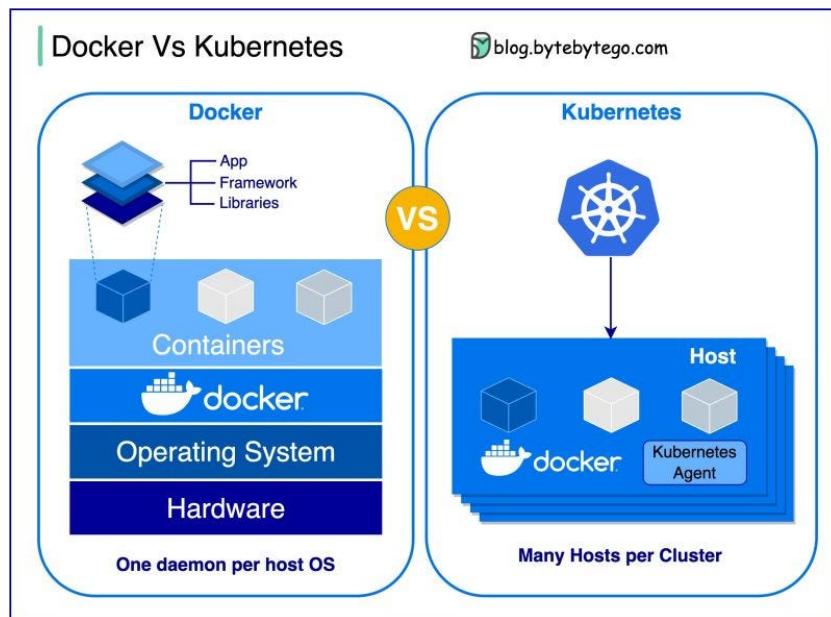
Raspodijeljeni sustav je skupina više mikroservisa, a svaki želimo "spakirati" u zaseban Docker kontejner. Kako se sustavi opisani u 3 prethodna primjera sastoje od više mikroservisa, a sustav u cijelosti ne može funkcionirati ako nedostaje barem jedan, praktično je koristiti **Docker Compose** za definiranje i upravljanje svim kontejnerima kao cjelinom 🚀



Ilustracija rada Docker Compose alata

Međutim, **važno je naglasiti sljedeće**: Docker Compose alat nam omogućuje pokretanje više kontejnera kao cjeline, međutim ta cjelina se izvodi na **jednom računalu**. Dakle, ako se jedno računalo pokvari, cijeli sustav će prestati raditi, bez obzira što je on na aplikacijskog razini raspodijeljen na više mikroservisa.

Postoje sofisticirana programska rješenja koja omogućuju **orkestraciju raspodijeljenog sustava** na više računala, kao što su **Kubernetes** i **Docker Swarm**. Ova složena rješenja omogućuju automatsko upravljanje kontejnerima, skaliranje, nadzor i druge napredne značajke. Međutim, to je tema sama za sebe i izlazi iz okvira ovog kolegija.



Ilustracija usporedbe Docker i Kubernetes alata

2.1 Kako spakirati više mikroservisa u jednu cjelinu

Docker Compose dolazi već instaliran s najnovijom verzijom Docker Desktop aplikacije, a dostupan je na svim operacijskim sustavima.

Možete provjeriti verziju Docker Compose alata naredbom:

```
docker compose version
```

Na linux sustavima je potencijalno potrebno naknadno instalirati Docker Compose alat, izvorni kod možete pronaći na sljedećoj poveznici: <https://github.com/docker/compose/releases>

Docker Compose koristi `docker-compose.yml` datoteku za definiranje kontejnera i drugih resursa koji će se pokrenuti kao cjelina.

Zašto ne bismo kombinirali `aiohttp` i `FastAPI` mikroservise koje smo ranije definirali u jedan "raspodijeljeni sustav" pomoću Docker Compose alata?

Napravit ćemo novi direktorij `compose-example` i unutar njega kreirati `docker-compose.yml` datoteku:

```
mkdir compose-example
cd compose-example
touch docker-compose.yml
```

Struktura direktorija treba izgledati ovako:

```
compose-example/
└── docker-compose.yml
```

Kako bi stvari imale više smisla, možemo malo redizajnirati `aiohttp` mikroservis na način da vraća podatke o regijama, umjesto o proizvodima.

Kopirat ćemo `aiohttp` mikroservis u novi direktorij `aiohttp-regije` koji se nalazi unutar `compose-example` direktorija:

Struktura direktorija `compose-example` treba izgledati ovako:

```
compose-example/
├── aiohttp-regije/
│   ├── app.py
│   └── Dockerfile
└── docker-compose.yml
```

U `aiohttp` mikroservisu, malo ćemo izmjeniti definiciju ruta i podatke koje vraća:

```
# compose-example/aiohttp-regije/app.py

import asyncio
from aiohttp import web
```

```

app = web.Application()

dummy_podaci_regije = [
    {"kljuc": "sredisnja", "naziv": "Središnja Hrvatska", "gradovi": ["Zagreb", "Karlovac", "Sisak"]},
    {"kljuc": "istocna", "naziv": "Istočna Hrvatska", "gradovi": ["Osijek", "Slavonski Brod", "Vinkovci", "Vukovar"]},
    {"kljuc": "gorska", "naziv": "Gorska Hrvatska", "gradovi": ["Delnice", "Čabar", "Vrbovsko"]},
    {"kljuc": "unutrasnjost Dalmacije", "naziv": "Unutrašnjost Dalmacije", "gradovi": ["Knin", "Sinj", "Imotski"]},
    {"kljuc": "sjeverni Jadran", "naziv": "Sjeverni Jadran", "gradovi": ["Rijeka", "Pula", "Opatija", "Rovinj"]},
    {"kljuc": "srednji Jadran", "naziv": "Srednji Jadran", "gradovi": ["Split", "Zadar", "Šibenik"]},
    {"kljuc": "južni Jadran", "naziv": "Južni Jadran", "gradovi": ["Dubrovnik", "Metković", "Ploče"]}
]

async def get_regije(request):
    return web.json_response(dummy_podaci_regije)

async def get_regija(request):
    kljuc = request.match_info['kljuc']
    for regija in dummy_podaci_regije:
        if regija['kljuc'] == kljuc:
            return web.json_response(regija)
    return web.json_response({"error": "Regija nije pronađena"}, status=404)

app.router.add_get("/regije", get_regije)
app.router.add_get("/regije/{kljuc}", get_regija)

web.run_app(app, host='0.0.0.0', port=4000) # promijenili smo port na 4000, čisto tako

```

Naravno, moramo generirati i `requirements.txt` datoteku:

```
pip freeze > requirements.txt
```

Definirajmo `Dockerfile` za `aiohttp-regije` mikroservis:

```
# compose-example/aiohttp-regije/Dockerfile

FROM python:3.11-slim
WORKDIR /app
COPY . /app
RUN pip install -r requirements.txt
EXPOSE 4000
CMD ["python", "app.py"]
```

Riješili smo `aiohttp-regije` mikroservis, struktura direktorija `compose-example` treba izgledati ovako:

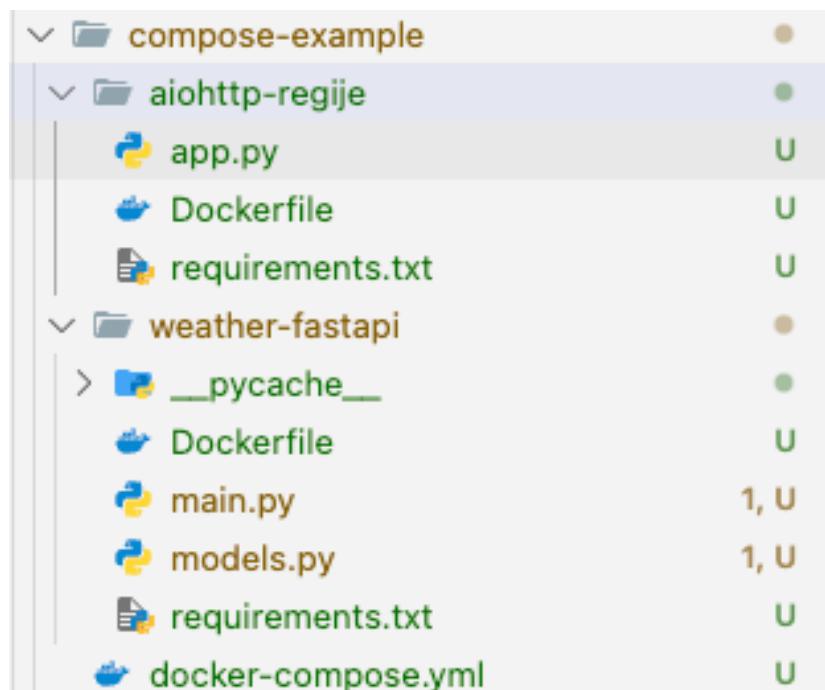
```
compose-example/
├── aiohttp-regije/
│   ├── app.py
│   ├── requirements.txt
│   └── Dockerfile
└── docker-compose.yml
```

FastAPI mikroservis nećemo mijenjati, već ga jednostavno kopiramo u `compose-example` direktorij:

```
compose-example/
├── aiohttp-regije/
│   ├── app.py
│   ├── requirements.txt
│   └── Dockerfile
├── weather-fastapi/
│   ├── main.py
│   ├── models.py
│   ├── requirements.txt
│   └── Dockerfile
└── docker-compose.yml
```

Ako koristite VS Code, preporuka je instalirati Material Icon Theme ekstenziju kako bi direktoriji i datoteke imali ikone:

- [Material Icon Theme](#)



Struktura direktorija `compose-example` u VS Code okruženju, `__pycache__` direktoriji su generirani od strane Python interpretera i možemo ih ignorirati

To je to, struktura je spremna, a sada možemo ova dva mikroservisa pokrenuti kao cjelinu pomoću Docker Compose alata!

2.1.1 Sintaksa docker-compose.yml datoteke

Otvorite `docker-compose.yml` datoteku u `compose-example` direktoriju.

Na početku svake `docker-compose.yml` datoteke obično se nalazi verzija Docker Compose alata, mi ćemo koristiti verziju `3.8`:

`docker-compose.yml` datoteka:

```
version: '3.8'
```

Mikroservise ćemo definirati unutar ključa `services`:

```
version: '3.8'

services:
  naziv_servisa:
    image: ime_docker_predloska
    ports:
      - "host_port:container_port"
```

Svaki mikroservis je ustvari kontejner, a **za svaki kontejner** moramo obavezno definirati koji Docker predložak koristi te koji portovi su mapirani:

```
version: '3.8'

services:
  aiohttp-regije: # ime kontejnera
    image: aiohttp-regije:1.0 # ime Docker predloška
    ports: # mapiranje portova
      - "4000:4000" # host_port:container_port
```

Dodat ćemo i FastAPI mikroservis:

```
version: '3.8'

services:
  aiohttp-regije:
    image: aiohttp-regije:1.0
    ports:
      - "4000:4000"

  weather-fastapi:
    image: weather-fastapi:1.0
    ports:
      - "8000:8000"
```

Moramo paziti da postoje dva različita Docker predloška definirana na našem računalu, `aiohttp-regije:1.0` i `weather-fastapi:1.0`, koje smo definirali u prethodnim koracima.

Aktivne Docker predloške možemo provjeriti naredbom:

```
docker images
```

Ako ih nema, izgradite prvo oba predloška:

- pazite da se nalazite u direktoriju gdje se nalazi `Dockerfile` određenog mikroservisa!

```
cd aiohttp-regije  
docker build -t aiohttp-regije:1.0 .
```

```
cd ..  
cd weather-fastapi  
docker build -t weather-fastapi:1.0 .
```

Nakon što smo izgradili oba predloška, možemo pokrenuti oba mikroservisa kao cjelinu pomoću Docker Compose alata. Navigirajte u `compose-example` direktorij i pokrenite sljedeću naredbu:

```
docker compose up
```

Ova naredba pokreće sve mikroservise definirane u `docker-compose.yml` datoteci kao cjelinu. Moguće da će vas Docker tražiti autentifikaciju kako bi pristupio vašim predlošcima, u tom slučaju unesite:

```
docker login
```

Nakon što se uspješno autentificirate, Docker Compose će pokrenuti oba mikroservisa kao cjelinu! 🚀

The screenshot shows the Docker Desktop interface. On the left, there's a sidebar with options like Containers, Images, Volumes, Builds, Docker Hub, Docker Scout, Extensions, Manage, PGAdmin4, Volumes, Backup & Share. The main area is titled 'Containers' with a 'Give feedback' link. It displays system-wide resource usage: Container CPU usage (0.22% / 800%) and Container memory usage (69.96MB / 7.57GB). A search bar and a filter button ('Only show running containers') are present. Below is a table listing the three containers:

	Name	Container ID	Image	Port(s)	CPU (%)	Last started	Actions
compose-example	-	-	-	-	0.22%	2 minutes ago	[...]
weather-fastapi-1	af9a9466ed7c	weather-fastapi:1.0		8000:8000	0.22%	2 minutes ago	[...]
aiohttp-regije-1	1f11677d0dde	aiohttp-regije:1.0		4000:4000	0%	2 minutes ago	[...]

The screenshot shows the Docker Desktop interface with the 'compose-example' service selected. The logs pane displays the startup logs for the 'weather-fastapi-1' container:

```
2025-01-22 00:17:46 weather-fastapi-1 | INFO: Started server process [1]  
2025-01-22 00:17:46 weather-fastapi-1 | INFO: Waiting for application startup.  
2025-01-22 00:17:46 weather-fastapi-1 | INFO: Application startup complete.  
2025-01-22 00:17:46 weather-fastapi-1 | INFO: Uvicorn running on http://0.0.0.0:8000 (Press CTRL+C to quit)
```

Pokrenuti mikroservisi kao cjelina pomoću Docker Compose alata. Prikaz unutar Docker Desktop aplikacije

Vidimo da svaki servis ima svoj vlastiti kontejner i da su mapirani portovi definirani u `docker-compose.yml` datoteci.

Mikroservise možemo zaustaviti naredbom:

```
docker compose down
```

2.2 Interna komunikacija mikroservisa

Jedna od ključnih značajki mikroservisne arhitekture je **interni komunikacija** između mikroservisa. Svaki mikroservis trebao bi biti izolirano okruženje, a komunikacija između mikroservisa trebala bi biti sigurna i pouzdana.

U našem primjeru, `aiohttp-regije` mikroservis vraća podatke o regijama, a `weather-fastapi` mikroservis vraća podatke o vremenu, a pristup im možemo preko domaćina i odgovarajućih portova.

Što ako želimo da `weather-fastapi` mikroservis dohvata podatke o regijama iz `aiohttp-regije` mikroservisa?

- u tom slučaju pričamo o internoj komunikaciji između mikroservisa
- dakle, servis A i B komuniciraju između sebe, a ne preko vanjskog korisnika (domaćina)
- ovo je **ključna značajka mikroservisne arhitekture**

Recimo da želimo da `weather-fastapi` mikroservis dohvata podatke o regijama iz `aiohttp-regije` mikroservisa jednom kad domaćin pošalje zahtjev na `/vrijeme` rutu mikroservisa `weather-fastapi`.

Domaćin ↔ weather-fastapi ↔ aiohttp-regije

Premda nije potrebno eksplicitno navoditi, uobičajeno je definirati **bridge network** unutar `docker-compose.yml` datoteke kako bi svi mikroservisi bili povezani na istoj mreži.

Mreže dodajemo pod ključ `networks`:

```
version: '3.8'

services:
  aiohttp-regije:
    image: aiohttp-regije:1.0
    ports:
      - "4000:4000"
    networks:
      - interna_mreza

  weather-fastapi:
    image: weather-fastapi:1.0
    ports:
      - "8000:8000"
    networks:
      - interna_mreza

networks:
```

```
interna_mreza: # proizvoljno ime mreže
  driver: bridge # tip mreže
```

Docker compose nam omogućuje da koristimo sam naziv kontejnera kao domenu, odnosno *hostname* prilikom definiranja internih komunikacija.

Dakle, `weather-fastapi` mikroservis može poslati HTTP zahtjev na `aiohttp-regije` mikroservis, putem rute:

```
http://aiohttp-regije:4000/regije
```

S druge strane, `aiohttp-regije` mikroservis može poslati HTTP zahtjev na `weather-fastapi` mikroservis, putem rute:

```
http://weather-fastapi:8000/vrijeme
```

Idemo ovo testirati, nadogradit ćemo mikroservis `weather-fastapi` tako da dohvaća podatke o regijama iz `aiohttp-regije` mikroservisa.

U `weather-fastapi` mikroservisu, dodajemo novu rutu `/vrijeme-regije` koja će dohvaćati podatke o regijama iz `aiohttp-regije` mikroservisa:

```
# compose-example/weather-fastapi/main.py

@app.get("/regije")
async def get_regije():
    async with aiohttp.ClientSession() as session:
        response = await session.get("http://aiohttp-regije:4000/regije") # koristimo naziv
kontejnera kao domenu
        regije = await response.json()
        return regije
```

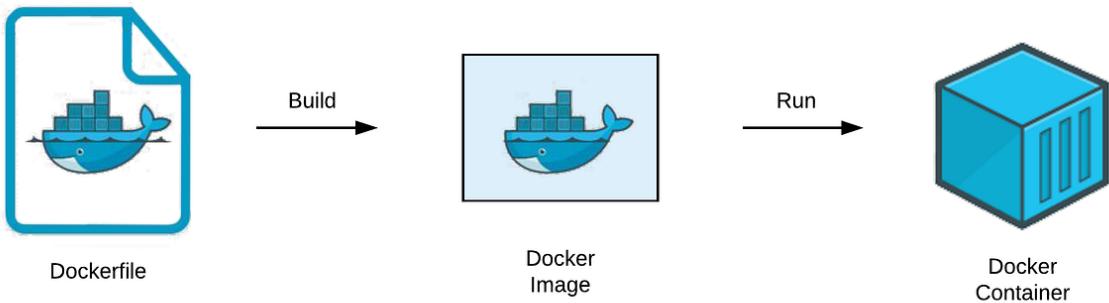
Obzirom da smo izmijenili kod, moramo ponovno izgraditi predložak:

```
cd weather-fastapi
docker build -t weather-fastapi:1.0 .
```

Nakon što izgradimo predložak, možemo ponovno pokrenuti mikroservise kao cjelinu:

```
docker compose up
```

Otvorite dokumentaciju mikroservisa na `http://localhost:8000/docs` i pokušajte pozvati rutu `/regije`. Trebali biste dobiti podatke o regijama koje vraća `aiohttp-regije` mikroservis.



Interna komunikacija između mikroservisa pomoću Docker Compose alata

2.3 Varijable okruženja u Dockeru

Varijable okruženja (eng. *environment variables*) su ključne za konfiguraciju mikroservisa, jer nam omogućuju da postavimo različite vrijednosti za različite okoline (npr. razvoj, testiranje, produkcija)

Stvari su trivijalne kada definiramo varijable okruženja za svaki mikroservis zasebno. Ako verzioniramo kod, svakako je uobičajena praksa koristiti ih za osjetljive podatke, poput lozinki, privatnih ključeva i drugih tajnih informacija.

Varijable okruženja u Pythonu možemo postaviti pomoću `os` modula ili pomoću `.env` datoteke i `python-dotenv` paketa.

```

import os

os.environ['VARIJABLA'] = 'vrijednost'

```

Ipak, u pravilu ih u kodu želimo samo čitati, ne i postavljati. Varijable okruženja možemo definirati unutar datoteke `.env`:

Vratimo se na primjer s `aiohttp-regije` mikroservisom. Definirat ćemo varijablu okruženja `PORT` unutar `.env` datoteke. Recimo da želimo koristiti različiti PORT ovisno o okolini.

- u lokalnom razvoju koristimo port `4000`
- u kontejneriziranoj okolini koristimo port `5000`

Instalirat ćemo paket `python-dotenv` u okruženju `aiohttp-microservice`:

```

conda activate aiohttp-microservice
pip install python-dotenv

```

Kako smo sad izmijenili ovisnosti, odmah ćemo ažurirati naš `requirements.txt`:

```

pip freeze > requirements.txt

```

Nakon toga, kreiramo `.env` datoteku u `aiohttp-regije` direktoriju:

```

compose-example/
    ├── aiohttp-regije/
    │   ├── app.py
    │   ├── requirements.txt
    │   ├── Dockerfile
    │   └── .env
    ├── weather-fastapi/
    │   ├── main.py
    │   ├── models.py
    │   ├── requirements.txt
    │   └── Dockerfile
    └── docker-compose.yml

```

Unutar datoteke `.env` definiramo varijablu okruženja `PORT` i postavljamo vrijednost na `4000`:

```
PORT=4000
```

U `app.py` datoteci, čitamo varijablu okruženja `PORT` i koristimo je za postavljanje poslužitelja:

```

# compose-example/aiohttp-regije/app.py

import os,
from dotenv import load_dotenv

load_dotenv() # učitavamo varijable iz .env datoteke

PORT = os.getenv("PORT") # čitamo varijablu okruženja PORT

```

Sada ju možemo koristi za pokretanje mikroservisa:

```

# compose-example/aiohttp-regije/app.py

web.run_app(app, host='0.0.0.0', port=int(PORT)) # koristimo varijablu okruženja PORT

```

To je to, `Dockerfile` možemo ostaviti nepromijenjen bez obzira na naredbu `EXPOSE 4000` - rekli smo da je to samo informativno i ne utječe na rad kontejnera.

Ipak, moramo ažurirati `docker-compose.yml` datoteku kako bismo izmjenili port u kontejnerskom okruženju:

Možemo definirati varijable okruženja unutar `environment` ključa:

```

version: '3.8'

services:
  aiohttp-regije:
    image: aiohttp-regije:1.0
    ports:
      - 4000:4000 # onda ovo možemo izmjeniti na način da čitamo varijablu okruženja

```

```

environment:
  - PORT=4000 # definiramo varijablu okruženja PORT i postavljamo vrijednost na 4000
networks:
  - interna_mreza

weather-fastapi:
  image: weather-fastapi:1.0
  ports:
    - "8000:8000"
  networks:
    - interna_mreza

```

Sada je potrebno ažurirati ključ `ports` unutar `aiohttp-regije` mikroservisa kako bi čitao varijablu okruženja `PORT`:

```

version: '3.8'

services:
  aiohttp-regije:
    image: aiohttp-regije:1.0
    ports:
      - "${PORT}:${PORT}" # koristimo varijablu okruženja PORT i za host i za kontejner port
    environment:
      - PORT=4000
    networks:
      - interna_mreza

  weather-fastapi:
    image: weather-fastapi:1.0
    ports:
      - "8000:8000"
    networks:
      - interna_mreza

```

Ipak, ako želimo pregaziti vrijednost varijable okruženja unutar `environment`, možemo to učiniti pomoću `.env` datoteke i `env_file` ključa:

```

version: '3.8'

services:
  aiohttp-regije:
    image: aiohttp-regije:1.0
    ports:
      - "${PORT}:${PORT}" # čitamo varijablu okruženja PORT iz .env datoteke
    env_file:
      - .env # učitavamo varijable okruženja iz .env datoteke
    networks:
      - interna_mreza

  weather-fastapi:

```

```

image: weather-fastapi:1.0
ports:
- "8000:8000"
networks:
- interna_mreza

```

Važno je ovdje uočiti sljedeće: U ovom kontekstu (datoteke `docker-compose.yml`), `.env` datoteka se nalazi u istom direktoriju kao i `docker-compose.yml` datoteka, a ne u direktoriju mikroservisa!

Dakle, možemo ju premjestiti u `compose-example` direktorij:

```

compose-example/
├── aiohttp-regije/
│   ├── app.py
│   ├── requirements.txt
│   └── Dockerfile
├── weather-fastapi/
│   ├── main.py
│   ├── models.py
│   ├── requirements.txt
│   └── Dockerfile
└── .env
└── docker-compose.yml

```

Izgradit ćemo ponovno predložak `aiohttp-regije`:

```

cd aiohttp-regije
docker build -t aiohttp-regije:1.0 .

```

Pokrećemo mikroservise:

```
docker compose up
```

Provjerite radi li kontejner `aiohttp-regije` na ispravnom portu koji ste definirali u `.env` datoteci.

```
docker ps
```

To je to! Dobivamo ispravni port koji smo definirali unutar `.env` datoteke:

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
PORTS		NAMES		
71a1a86ccd89	weather-fastapi:1.0	"uvicorn main:app --..."	About a minute ago	Up 10 seconds
94d7df51696f	aiohttp-regije:1.0	"python app.py"	About a minute ago	Up 10 seconds

2.4 Zadaci za vježbu: Docker Compose

1. Napravite novi direktorij `social-network` i unutar njega kopirajte mikroservise izrađene u **Zadacima za vježbu 1.8**: `authAPI` i `socialAPI`.

Definirajte `docker-compose.yml` datoteku koja će pokrenuti oba mikroservisa kao cjelinu. Mikroservisi trebaju biti povezani na istoj mreži i svaki raditi na svom portu.

Jednom kad ste pokrenuli mikroservise zajedno koristeći Docker Compose i to uredno radi, napravite sljedeće izmjene:

- u mikroservisu `socialAPI` izmjenite rutu `GET /korisnici/{korisnik}/objave` na način da se očekuje **tijelo HTTP zahtjeva** s korisničkim imenom i lozinkom, isto validirajte koristeći novi Pydantic model.
- prije nego ruta `GET /korisnici/{korisnik}/objave` vrati podatke, mikroservis `socialAPI` treba poslati HTTP zahtjev na `authAPI` mikroservis (ruta `/login`) kako bi provjerio korisničke podatke.
- implementirajte *dummy* autorizaciju u `authAPI` mikroservisu, tako da vraća `True` ako su korisničko ime i lozinka ispravni, inače vraća `False`.

Dakle, mikroservis `socialAPI` treba poslati HTTP zahtjev na `authAPI` mikroservis kako bi provjerio korisničke podatke prije nego što vrati podatke o objavama korisnika. Ako korisničko ime i lozinka nisu ispravni, `socialAPI` mikroservis treba vratiti grešku.

Nakon toga pokrenite oba mikroservisa zajedno koristeći Docker Compose i provjerite radi li nova funkcionalnost. **Napomena:** morate implementirati internu komunikaciju između 2 kontejnera, kao što je opisano u **poglavlju 2.2**.

3 Load balancing (`nginx`)

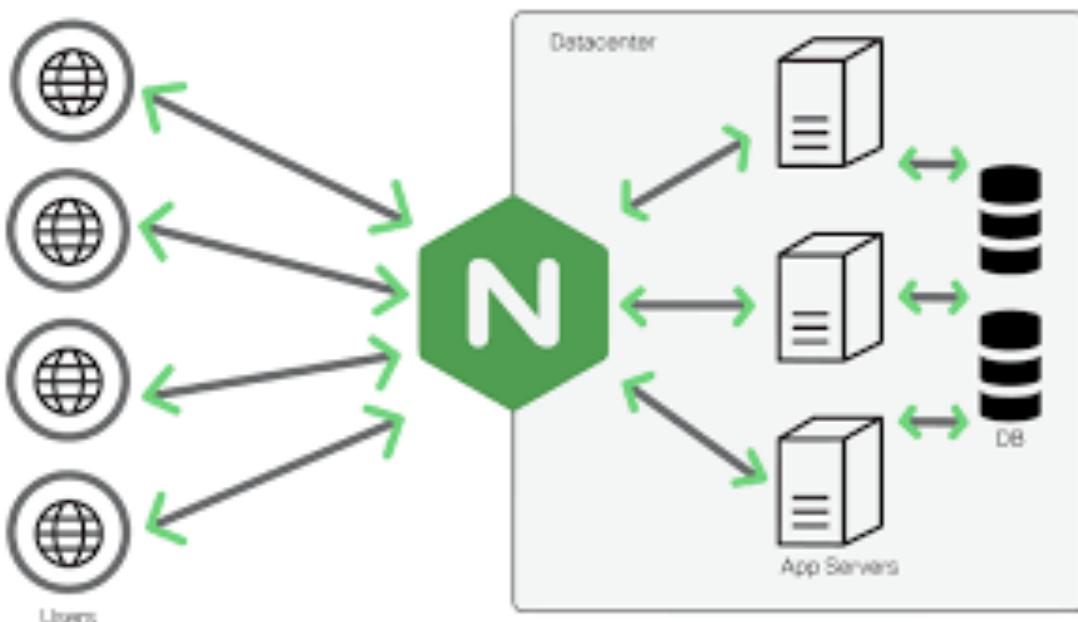
Load balancing je tehnika koja se koristi za distribuciju opterećenja između više poslužitelja, računala ili mrežnih uređaja. Ova tehnika omogućuje da se opterećenje ravnomjerno raspodijeli između više poslužitelja, kako bi se osigurala visoka dostupnost i pouzdanost sustava.

Ciljevi load balancinga su sljedeći:

- **Ravnomjerna raspodjela opterećenja** - svaki poslužitelj dobiva jednaku količinu zahtjeva
- **Visoka dostupnost** - ako jedan poslužitelj prestane raditi, drugi preuzimaju njegovo opterećenje
- **Prevencija da jedan poslužitelj postane usko grlo** - ako jedan poslužitelj postane preopterećen, load balancer preusmjerava zahtjeve na druge poslužitelje
- **Povećanje performansi** - load balancer može koristiti različite algoritme za raspodjelu opterećenja, ovisno o potrebama sustava

Postoje različite vrste load balančera, međutim mi se nećemo baviti detaljima. U ovom primjeru koristit ćemo **nginx** kao load balancer za naše mikroservise.

nginx je popularan web poslužitelj i *reverse proxy server* koji se koristi za posluživanje web stranica, aplikacija i API-ja. Osim toga, **nginx** se može koristiti kao load balancer za distribuciju opterećenja između više poslužitelja.



Ilustracija rada load balančera

`nginx` nije dio Dockera, niti Pythona, već je zaseban softver koji se može instalirati na računalo.

Međutim, možemo koristiti `nginx` kao Docker kontejner i konfigurirati ga kao load balancer za naše mikroservise.

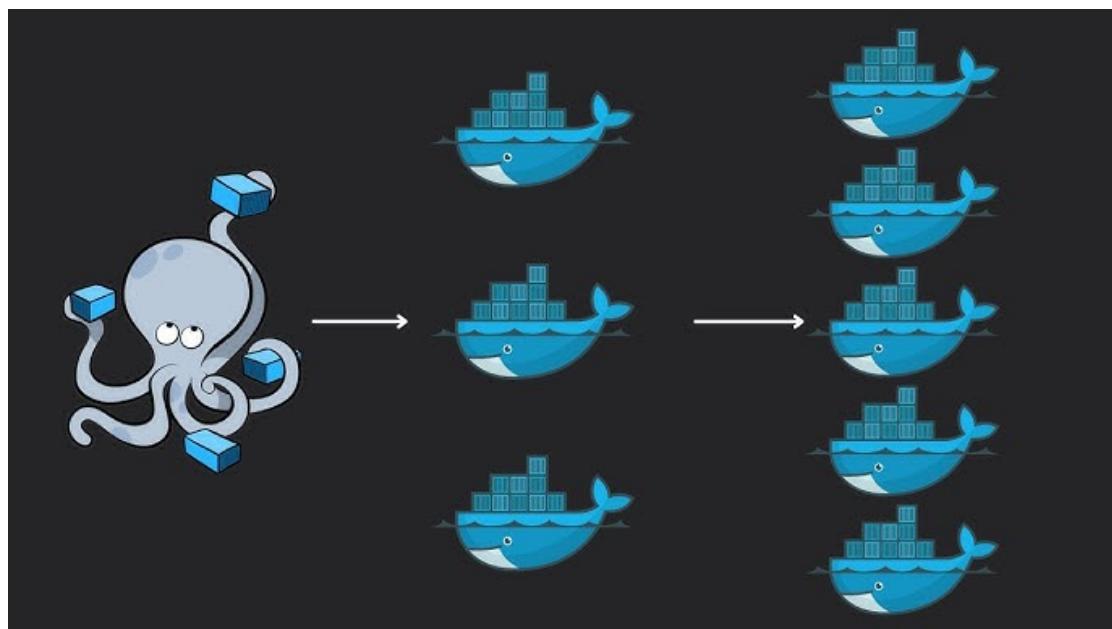
Možemo ga preuzeti preko Docker Huba, na sljedećoj poveznici: https://hub.docker.com/_/nginx

```
docker pull nginx
```

3.1 Horizontalno skaliranje koristeći samo Docker Compose

Horizontalno skaliranje (eng. *Horizontal scaling*) mikroservisa odnosi se na povećanje broja instanci mikroservisa kako bi se povećala dostupnost i performanse sustava. Primjerice, ako iz našeg primjera imamo samo jednu instancu `weather-fastapi` mikroservisa, možemo dodati još jednu instancu u slučaju da prva prestane raditi.

Dakle, u ovom kontekstu samo povećavamo **broj instanci mikroservisa**.



Ilustracija horizontalnog skaliranja mikroservisa

Na primjer, želimo dodati 3 replike `weather-fastapi` mikroservisa i 2 replike `aiohttp-regije` mikroservisa. To radimo kroz `docker-compose.yml` datoteku:

Sintaksa:

```
version: '3.8'

services:
  naziv_servisa:
    image: ime_docker_predloska
    ports:
      - "host_port:container_port"
    deploy:
      replicas: broj_replika
```

Odnosno na našem primjeru:

```
version: '3.8'

services:
```

```

aiohttp-regije:
  image: aiohttp-regije:1.0
  ports:
    - "${PORT}:${PORT}"
  env_file:
    - .env
  networks:
    - interna_mreza
  deploy:
    replicas: 2

weather-fastapi:
  image: weather-fastapi:1.0
  ports:
    - "8000:8000"
  networks:
    - interna_mreza
  deploy:
    replicas: 3

networks:
  interna_mreza: # proizvoljno ime mreže
  driver: bridge # tip mreže

```

Možemo pokrenuti ove kontejnere, međutim dobit ćemo **grešku** prilikom pokretanja budući da Docker pokušava mapirati isti port na više kontejnera prema domaćinu, što nije dozvoljeno.

Containers [Give feedback](#)

View all your running containers and applications. [Learn more](#)

Container CPU usage		Container memory usage		Show charts	
0.39% / 800% (8 CPUs available)		69.95MB / 7.57GB			
<input type="checkbox"/>	<input type="checkbox"/> Search	<input type="checkbox"/>	<input checked="" type="checkbox"/> Only show running containers		
	Name	Container ID	Image	Port(s)	Actions
<input type="checkbox"/>	compose-example	-	-	-	<input type="checkbox"/> ⋮ <input type="checkbox"/> ⌂
<input type="checkbox"/>	aiohttp-regije-1	952a5232b867	aiohttp-regije:1.0	4000:4000 ⌂	<input type="checkbox"/> ⋮ <input type="checkbox"/> ⌂
<input type="checkbox"/>	aiohttp-regije-2	3786219894d8	aiohttp-regije:1.0	4000:4000	<input type="checkbox"/> ⋮ <input type="checkbox"/> ⌂
<input type="checkbox"/>	weather-fastapi-1	6687eb09a7a7	weather-fastapi:1.0	8000:8000	<input type="checkbox"/> ⋮ <input type="checkbox"/> ⌂
<input type="checkbox"/>	weather-fastapi-2	fe83aa3c38ba	weather-fastapi:1.0	8000:8000	<input type="checkbox"/> ⋮ <input type="checkbox"/> ⌂
<input type="checkbox"/>	weather-fastapi-3	c588b7ed2691	weather-fastapi:1.0	8000:8000 ⌂	<input type="checkbox"/> ⋮ <input type="checkbox"/> ⌂

Problem možemo riješiti koristeći **nginx** kao load balancer koji će **distribuirati zahtjeve na različite mikroservise**.

Prvo ćemo dodati `nginx` kontejner u `docker-compose.yml` datoteku:

- radi pojednostavljenja, trenutno ćemo maknuti dinamičko mapiranje portova i staviti fiksne portove za svaki mikroservis

```

version: '3.8'

services:

```

```

aiohttp-regije:
    image: aiohttp-regije:1.0
    ports:
        - "4000:4000" # fiksni port za aiohttp-regije
    networks:
        - interna_mreza
    deploy:
        replicas: 2

weather-fastapi:
    image: weather-fastapi:1.0
    ports:
        - "8000:8000" # fiksni port za weather-fastapi
    networks:
        - interna_mreza
    deploy:
        replicas: 3

nginx: # dodajemo nginx load balancer
    image: nginx
    ports:
        - "80:80"
    volumes: # mapiramo konfiguracijsku datoteku
        - ./nginx.conf:/etc/nginx/nginx.conf # konfiguracijska datoteka za nginx je
nginx.conf
    networks:
        - interna_mreza

networks:
    interna_mreza: # proizvoljno ime mreže
        driver: bridge # tip mreže

```

`nginx` definiramo unutar konfiguracijske datoteke `nginx.conf` koja se mora nalaziti u istom direktoriju kao i `docker-compose.yml` datoteka:

Struktura direktorija:

```

compose-example/
    └── aiohttp-regije/
        ├── app.py
        ├── requirements.txt
        └── Dockerfile
    └── weather-fastapi/
        ├── main.py
        ├── models.py
        ├── requirements.txt
        └── Dockerfile
    └── nginx.conf
    └── .env
    └── docker-compose.yml

```

Reverse proxy odnosi se na tehniku koja omogućuje da se zahtjevi preusmjere s jednog poslužitelja na drugi. U našem slučaju, `nginx` će **preusmjeravati zahtjeve na različite mikroservise**. Više o ovoj temi pročitajte na sljedećoj [poveznici](#).

Unutar `nginx.conf` datoteke, prvo ćemo definirati `upstream` blok u kojem ćemo navesti sve mikroservise na koje će `nginx` preusmjeravati zahtjeve, to su `aiohttp-regije` i `weather-fastapi` mikroservisi:

VAŽNO! Bez obzira na interne portove unutar kontejnera, ovdje možemo definirati na koje portove će `nginx` preusmjeravati zahtjeve, odnosno koje portove će koristiti domaćin (**krajnji korisnik**).

Trenutni portovi definirani unutar `docker-compose.yml` su:

- `aiohttp-regije: 4000`
- `weather-fastapi: 8000`

Otvorite `nginx.conf` datoteku:

1. korak: definicija `events` bloka gdje navodimo najveći broj konekcija koje `nginx` može obraditi istovremeno

```
events {  
    worker_connections 1024;  
}
```

2. korak: definicija `http` bloka gdje navodimo `upstream` blok i `server` blok

Prvo ćemo navesti `upstream` blokove u kojima navodimo naše mikroservise:

```
http {  
    upstream aiohttp-regije {  
        server aiohttp-regije:4000;  
    }  
  
    upstream weather-fastapi {  
        server weather-fastapi:8000;  
    }  
}
```

3. korak: definiramo *reverse-proxy* na proizvolnjom portu (npr. `80`) i **preusmjeravamo sve zahtjeve** na `aiohttp-regije` i `weather-fastapi` mikroservise:

- na lokaciji `/aiohttp` preusmjeravamo sve zahtjeve na `aiohttp-regije` mikroservis
- na lokaciji `/fastapi` preusmjeravamo sve zahtjeve na `weather-fastapi` mikroservis

Ukupna konfiguracija `nginx.conf` datoteke:

```
events {  
    worker_connections 1024;  
}
```

```

http {

upstream aiohttp-regije {
    server aiohttp-regije:4000;
}

upstream weather-fastapi {
    server weather-fastapi:8000;
}

server {
    listen 80;

    location /aiohttp {
        proxy_pass http://aiohttp-regije;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    }

    location /fastapi {
        proxy_pass http://weather-fastapi;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    }
}
}
}

```

Jednostavno pokrećemo opet mikroservise koristeći `docker-compose up` naredbu:

```
docker compose up
```

Otvorite `http://localhost/aiohttp` i `http://localhost/fastapi` u web pregledniku i provjerite radi li load balancer kako treba.

Vidimo da nema grešaka, `nginx` uspješno preusmjerava zahtjeve na `aiohttp-regije` i `weather-fastapi` mikroservise.

```

nginx-1          | /docker-entrypoint.sh: /docker-entrypoint.d/ is not empty, will
attempt to perform configuration
nginx-1          | /docker-entrypoint.sh: Looking for shell scripts in /docker-
entrypoint.d/
nginx-1          | /docker-entrypoint.sh: Launching /docker-entrypoint.d/10-listen-on-
ipv6-by-default.sh
nginx-1          | 10-listen-on-ipv6-by-default.sh: info: IPv6 listen already enabled
nginx-1          | /docker-entrypoint.sh: Sourcing /docker-entrypoint.d/15-local-
resolvers.envsh

```

```

nginx-1          | /docker-entrypoint.sh: Launching /docker-entrypoint.d/20-envsubst-on-
templates.sh
nginx-1          | /docker-entrypoint.sh: Launching /docker-entrypoint.d/30-tune-worker-
processes.sh
nginx-1          | /docker-entrypoint.sh: Configuration complete; ready for start up
weather-fastapi-1 | INFO:      Started server process [1]
weather-fastapi-1 | INFO:      Waiting for application startup.
weather-fastapi-1 | INFO:      Application startup complete.
weather-fastapi-1 | INFO:      Uvicorn running on http://0.0.0.0:8000 (Press CTRL+C to
quit)
nginx-1           | 172.20.0.1 -- [22/Jan/2025:08:12:35 +0000] "GET /aiohttp HTTP/1.1"
404 14 "-" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/132.0.0.0 Safari/537.36"
weather-fastapi-1 | INFO:      172.20.0.4:59704 - "GET /fastapi HTTP/1.0" 404 Not Found
nginx-1           | 172.20.0.1 -- [22/Jan/2025:08:12:38 +0000] "GET /fastapi HTTP/1.1"
404 22 "-" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/132.0.0.0 Safari/537.36"
nginx-1           | 172.20.0.1 -- [22/Jan/2025:08:16:49 +0000] "GET /aiohttp HTTP/1.1"
404 14 "-" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/132.0.0.0 Safari/537.36"
weather-fastapi-1 | INFO:      172.20.0.4:33340 - "GET /fastapi HTTP/1.0" 404 Not Found
nginx-1           | 172.20.0.1 -- [22/Jan/2025:08:16:51 +0000] "GET /fastapi HTTP/1.1"
404 22 "-" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/132.0.0.0 Safari/537.36"

```

Vidimo da u Docker Desktopu nemamo više duple instance `weather-fastapi` i `aiohttp-regije` mikroservisa, već samo jednu instancu svakog mikroservisa, a `nginx` uspješno preusmjerava zahtjeve na njih.

Dakle, **horizontalno skaliranje** mikroservisa možemo postići kroz `docker-compose.yml` datoteku i `nginx` kao load balancer, a cijelu apstrakciju balansiranja izvršava sam `nginx` kontejner 😎

<input type="checkbox"/>	Name	Container ID	Image	Port(s)	Actions		
<input type="checkbox"/>	compose-example	-	-	-	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
<input type="checkbox"/>	nginx-1	e10776ff5f12	nginx	80:80 ⚡	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
<input type="checkbox"/>	weather-fastapi-1	7f490a630a08	weather-fastapi:1.0	8000:8000 ⚡	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
<input type="checkbox"/>	aiohttp-regije-1	74fabd5e00d3	aiohttp-regije:1.0	8001:8001 ⚡	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Load balancer `nginx` uspješno preusmjerava zahtjeve na `aiohttp-regije` i `weather-fastapi` mikroservise

Zadaci iz Load Balancinga neće biti na kolokviju budući da je ovo naprednija tema. Međutim, preporuka je da studenti samostalno istraže ovu temu i pokušaju implementirati *load balancer* u svojim projektima. Ovdje imate dobar primjer od kuda započeti.

GOTOVO! 🎉🎉🎉