Raspodijeljeni sustavi (RS)

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

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

(2) Napredniji Python koncepti

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: 9.11.2024.

Sadržaj

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 primijeni_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 definirait malo anonimnu lambda funkciju:

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

- i proslijedimo je kao argument funkciji `primijeni_na_sve`:

```
print(primijeni_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(primijeni_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(primijeni_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** ILI **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 `primijeni_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_rodenja": 2000},
    {"ime": "Marko", "prezime": "Marković", "godina_rodenja": 1990},
    {"ime": "Ana", "prezime": "Anić", "godina_rodenja": 2003},
    {"ime": "Petra", "prezime": "Petrić", "godina_rodenja": 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", "prijatelj", "zvuk", "čokolada", "ples",
"pjesma", "otorinolaringolog"]

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

# min_duljina = 7
duge_rijeci = ...

# print(duge_rijeci) # ['zvijezda', 'prijatelj', '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", "prijatelj", "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 programski 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 > memorijska 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 **podklasa** (*eng. subclass*), a klasa koja se nasljeđuje zove se **nadklasa** (*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 izbjeći 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`:

```python
# 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:

```python
# 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.

```python
# operacije.py

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

Učitavanje modula `operacije.py`:

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

    def ispis_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

1. Definirajte klasu `Automobil` s atributima `marka`, `model`, `godina_proizvodnje` i `kilometraža`. Dodajte metodu `ispis` koja će ispisivati sve atribute 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 dohvatite pomoću `datetime` modula.
2. 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`.
3. 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.

4. Definirajte klasu `Krug` s atributom `r`. Dodajte metode `opseg` i `povrsina` koje će računati opseg i površinu kruga.
    - Stvorite objekt klase `Krug` s proizvoljnim radijusom 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 `kolicina`. Dodajte metodu `ispis` koja će ispisivati sve atribute proizvoda.
- u listu `proizvodi` dodajte 2 objekta klase `Proizvod` s proizvoljnim vrijednostima atributa.
- definirajte funkciju `dodaj_proizvod` van definicije klase koja će dodavati novi `Proizvod` u listu `proizvodi`.

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

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

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

Modul `narudzbe.py`:

- definirajte klasu `Narudzba` s atributima: `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 `proizvodi` mora biti lista
    - svaki element u listi mora biti rječnik
    - svaki rječnik mora sadržavati ključeve `naziv`, `cijena` i `kolicina`

    - lista ne smije biti prazna
- izračunajte ukupnu cijenu narudžbe koju ćete pohraniti u `ukupna_cijena` u jednoj liniji koda.
- narudžbe 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.