Komentiranje

`#`	početak linijskog komentara

Osnovni tipovi varijabli / objekata

`str`	✗	(string), (uređeni) niz znakova
`int`	✗	(integer), cijeli broj
`float`	✗	broj s pomičnim zarezom
`bool`	✗	(boolean), logička varijabla **True** ili **False**

✗ - objekt **nije identiteno-promjenjiv** tj. *immutable* je
✓ - objekt **je identiteno-promjenjiv** tj. *mutable* je

Složeni tipovi varijabli / objekata

`range`	✗	raspon - uređen (indeksiran) niz integera
`list`	✓	lista - uređena (indeksirana) podatkovna kolekcija (niz objekata)
`dict`	✓	rječnik - neuređena (neindeksirana) podatkovna kolekcija parova ključ-vrijednost
`tuple`	✗	n-terac - uređena (indeksirana) **nepromjenljiva** podatkovna kolekcija
`set`	✗	skup - neuređena (neindeksirana) podatkovna kolekcija **jedinstvenih** vrijednosti

✗ - objekt **nije promjenjiv** tj. *immutable* je
✓ - objekt **je promjenjiv** tj. *mutable* je

Konverzije varijabli

`str(object)`	konverzija u string
`int(object)`	konverzija u integer
`float(object)`	konverzija u float
`bool(object)`	konverzija u boolean
`list(iterable)`	konverzija (podatke kolekcije) u listu

Funkcije s objektima

`len(`*`object`*`)`	vraća broj znakova u objektu
`print(`*`object`*`)`	ispisuje objekt (ne vraća ništa)
`type(`*`object`*`)`	ispisuje tip objekta
`id(`*`object`*`)`	ispisuje identitet objekta (adresu u memoriji) *

* Svi objekti u Pythonu imaju svoj jedinstveni ID, koji se dodjeljuje objektu kada se objekt **kreira**.
ID objekta je adresa u memoriji i bit će drugačija svaki put kad se pokrene program.
Objekti kojima se vrijednosti mogu mijenjati bez promjene identiteta zovu se promjenljivi (engl. mutable) objekti, a oni kojima se vrijednost ne može mijenjati bez stvaranja novog objekta istog tipa zovu se nepromjenljivi (engl. immutable) objekti.
Promjena vrijednosti objekta obično se događa operatorom pridružbe ili djelovanjem metode na vrijednost objekta.

Ostale funkcije

`range([start], end, [step])`	vraća sekvencu (raspon) brojeva od "start" do "end", s razlikom od "step"

Deklaracija stringova

`"text"`	klasičan način zapisa stringa
`"""text"""`	zapis stringa koji omogućuje prijelaz u novi red
`f"text {`*`value`*`} text"`	formatirani string koji evaluira vrijednost u vitičastim zagradama

Prilikom deklaracije, mogu se koristiti i jednostruki navodnici! Stringovi nisu identiteno-promijenjivi...

```
» string1 = "Ivan"
» string2 = string1
» id(string1) == id(string2)
"True"
» string1 = "Marko"
» string2
"Ivan"
» id(string1) == id(string2)
"False"
```

... što znači da promjena jedne ne veže promjenu druge.

Osnovne operacije sa stringovima

`"text"[0]`	indeksiranje stringova (`"t"`)
`"text"[0:2:1]`	komad stringa (počevši od 0 do 2 (ne uključujući 2), s korakom 1) (`"te"`)
`"abc" + "def"`	povezivanje stringova (`"abcdef"`)
`"abc" * 2`	umnožavanje stringova (`"abcabc"`)

Posljednji član stringa ima indeks -1, predzadnji -2, itd.

Iteriranje kroz stringove

`for` *`char`* `in` *`string`*`:`	iteriranje kroz string

Funkcije sa stringovima

`input(`*`prompt`*`)`	traži vrijednost inputa u prompt (tipa string)

Metode nad stringovima

string.isnumeric()	✓	ispisuje **True** ako je string broj, a **False** ako nije
string.isalpha()	✓	ispisuje **True** ako je string sastavljen isključivo od slova, a **False** ako ne
string.find(*value*, *[start]*, *[end]*)	✓	vraća indeks prvog pojavljivanja "value"
string.index(*value*, *[start]*, *[end]*)	✓	slično kao find, ali u slučaju nepronalaska izbacuje grešku
string.rfind(*value*, *[start]*, *[end]*)	✓	vraća indeks zadnjeg pojavljivanja "value"
string.rindex(*value*, *[start]*, *[end]*)	✓	slično kao rfind, ali u slučaju nepronalaska izbacuje grešku
string.lstrip(*[character]*)	✓	uklanja niz "character" (ili razmak) s lijeve strane **kopije** stringa
string.rstrip(*[character]*)	✓	uklanja niz "character" (ili razmak) s desne strane **kopije** stringa
string.strip(*[character]*)	✓	uklanja niz "character" (ili razmak) s lijeve i desne strane **kopije** stringa
string.replace(*oldvalue*, *newvalue*, *[count]*)	✓	mijenja "oldvalue" za "newvalue" (u prvih "count" pojavljivanja) **kopije** stringa
string.count(*value*, *[start]*, *[end]*)	✓	ispisuje broj ponavljanja "value"
string.lower()	✓	ispisuje **kopiju** stringa kojemu su sva slova mala
string.upper()	✓	ispisuje **kopiju** stringa kojemu su sva slova velika
string.capitalize()	✓	ispisuje **kopiju** stringa kojemu je prvo slovo veliko
string.split(*delimiter*)	✓	ispisuje **novu listu** nastalu razdvajanjem **kopije stringa** po graničniku
delimiter.join(*iterable*)	✓	ispisuje **novi string** nastao spajanjem elemenata **kopije iterabla** po graničniku

✗ - metoda **nema** povrat tj. `return`
✓ - metoda **ima** povrat tj. `return`

Aritmetički operatori

+	zbrajanje
-	oduzimanje
*	množenje
**	potenciranje
/	dijeljenje
//	cjelobrojno dijeljenje
%	ostatak cjelobrojnog djeljenja

Operatori dodjele

=	a = 5
+=	a = a + 5
-=	a = a - 5
*=	a = a * 5
**=	a = a ** 5
/=	a = a / 5
//=	a = a // 5
%=	a = a % 5

Operatori usporedbe vrijednosti

`==`	jednako
`!=`	nije jednako
`>`	veće
`>=`	veće ili jednako
`<`	manje
`<=`	manje ili jednako

Izlaz može biti `True` ili `False`

Operatori usporedbe adresa u memoriji

`is`	jednakost adresa u memoriji
`is not`	nejednakost adresa u memoriji

Operator provjerava `id(object1) == id(object2)`
Izlaz može biti `True` ili `False`

Logički operatori

`and`	istinito ako su sve tvrdnje točne
`or`	istinito ako je barem jedna tvrdnja točna
`not`	inverzija (negacija) istinitosti tvrdnje

Izlaz može biti `True` ili `False`

Operatori članstva

`in`	točnost postojanja člana u sekvenci
`not in`	netočnost postojanja člana u sekvenci

Izlaz može biti `True` ili `False`

Neistinite (lažne) vrijednosti

`False`	definicijska "laž"
`None`	"vrijednost" varijable bez vrijednosti (a = None)
`0`	0 tipa cijelog broja
`0.0`	0 tipa broja s pomičnim zarezom
`""`	prazni string (u bilo kojem formatu)
`[]`	prazan niz
`()`	prazna n-torka
`{}`	prazan rječnik
`set()`	prazan skup
`range(0)`	prazan raspon

Grananje

`if` *<condition 1>*`:`	postavljanje prvog uvjeta
`<code block 1>`	kod koji se izvršava ako je prvi uvjet zadovoljen
`elif` *<condition 2>*`:`	postavljanje drugog uvjeta
<code block 2>	kod koji se izvršava ako je drugi uvjet zadovoljen
`else:`	pokrivanje svih ostalih uvjeta
<code block 3>	kod koji se izvršava ako prvi i drugi uvjet nisu zadovoljeni

`if`, `elif` i `else` moraju koristiti iste indentacije!
`<code block 1>`, `<code block 2>` i `<code block 3>` moraju koristiti iste indentacije!

Zadovoljavanje bilo kojeg od uvjeta tj. "grane" rezultira izlaskom iz "stabla" i nastavljanjem izvršavanja daljnjeg koda.

Ni `elif` ni `else` sekcije nisu obavezne.
... `elif` sekcija nije nužna ako se kod dijeli u samo dvije grane.
... `else` sekcija nije nužna u slučajevima tipa *else-do-nothing*.

`while` petlja

`while` *<condition>*:	`<condition>` je uvjet iteracije
<code block>	kod koji se izvršava u svakoj iteraciji

`while` petlja se izvršava kad nije unaprijed poznat broj potrebnih iteracija.

Petlja se izvršava dok je uvjet petlje zadovoljen. Kako bi završila, unutar same petlje mora doći do izmijene uvijeta.

`for` petlja

`for` *<iterator>* `in` *<iterable>*:	`<iterator>` je jedinični element strukture tj. podatkovne kolekcije `<iterable>`
<code block>	kod koji se izvršava u svakoj iteraciji

`for` petlja se izvršava kad je unaprijed poznat broj potrebnih iteracija.

U svakoj iteraciji iterator dobiva novu vrijednost. Petlja završava tek kad završe sve iteracije.

Struktura	Iterable	Iterator
Range	`range(3, 20, 2)`	`3, 5, 7, 9, 11, 13, 15, 17, 19`
String	`"apple"`	`a, p, p, l, e`
List	`["apple", "banana", "cherry"]`	`apple, banana, cherry`
Tuple	`("apple", "banana", "cherry")`	`apple, banana, cherry`
Set	`{"apple", "banana", "cherry"}`	`apple, banana, cherry`
Dictionary	`{"brand": "Ford", "model": "Mustang", "year": 1964}`	`brand, model, year`

Naredba `break`

`for` *<iterator>* `in` *<iterable>*:	
`if` *<condition>*:	postavljanje uvjeta koji prekida petlju
`break`	prekid petlje
<code block>	kod koji se (u suprotnom) izvršava u svakoj iteraciji

`break` služi kako bi se **prekinulo** izvršavanje **petlje**.

Naredba `continue`

`for` *<iterator>* `in` *<iterable>*`:`	
`if` *<condition>*`:`	postavljanje uvjeta kojim se preskače trenutna iteracija
`continue`	preskok iteracije
<code block>	kod koji se (u suprotnom) izvršava u svakoj iteraciji

`continue` služi kako bi se **preskočilo** izvršavanje **iteracije**.

Definiranje funkcije

`def` *name*`(`*params*`):`	definiranje imena funkcije i postavljanje parametara odvojenih zarezom, koji se koriste u bloku koda
<code block>	blok koda kojeg će funkcija izvršavati svakim pozivanjem
`return` *value*	povrat (rezultat) funkcije

`return` vraća rezultat funkcije i **izlazi iz funkcije**, slično kao i `break`
funkcija ne mora imati `return` ako ne vraća rezultat, npr. ako radi samo `print`
funkcija ne mora imati `params` ako nema ulazne podatke.

```
z = f(x,y)
z - value
f - name
x,y - params
```

Definiranje funkcije s *args

`def` *name*`(`*params, *args*`):`	... `*args` sakuplja **ostatak pozicijskih parametara** u **n-torku** (nepoznatog broj članova)
<code block>	blok koda kojeg će funkcija izvršavati svakim pozivanjem
`return` *value*	povrat (rezultat) funkcije

Budući da `*args` sakuplja **ostatak pozicijskih parametara**, nije nužno da se prilikom poziva funkcije u tom parametru nešto nađe, tj. **nije obavezan**.

Definiranje funkcije s **kwargs

`def` *name*`(`*params, **kwargs*`):`	... `**kwargs` sakuplja **ostatak parametara ključnih riječi** u **rječnik** (nepoznatog broj članova)
<code block>	blok koda kojeg će funkcija izvršavati svakim pozivanjem
`return` *value*	povrat (rezultat) funkcije

Budući da `*args` sakuplja **ostatak pozicijskih parametara**, nije nužno da se prilikom poziva funkcije u tom parametru nešto nađe, tj. **nije obavezan**.

Redoslijed parametara prilikom definiranja

```
def name(params, *args, default_params, **kwargs)
```

Pozivanje funkcije

name`(`*args*`)`	pozivanje funkcije koja nije imala `return`
variable `=` *name*`(`*args*`)`	pozivanje funkcije koja je imala `return` i pohranjivanje njenog rezultata u `variable`

```
a = f(2,5)
a - variable
  f - name
2,5 - args
```

Raspakiravanje liste u pojedinačne argumente

name`(*`*list*`)`	članovi liste će se **raspakirati** u **pojedinačne argumente** funkcije

Parametri funkcije

`def student(ime, prezime="Horvat", godina=1):`	definiranje funkcije `student` s 1 obaveznim i 2 opcionalna parametara
`  print(ime, prezime, "je", godina, '.  godina')`	ispis funkcije

Prvo se definiraju svi obavezni parametri, a zatim svi opcionalni parametri.

Pozicijski argumenti

`student("Ivan")`	Ivan Horvat je 1. godina
`student("Ivan", "Kovač", 2)`	Ivan Kovač je 2. godina
`student("Ivan", "Kovač")`	Ivan Kovač je 1. godina
`student("Ivan", 2)`	Ivan 2 je 1. godina

Pozicijski argumenti zahtjevaju definirani redoslijed i dodjelujuje se s lijevo na desno.
"Nespareni" argumenti dobivaju podrazumjevanu vrijednost.

Argumenti ključnih riječi

`student(ime="Ivan")`	Ivan Horvat je 1. godina
`student(ime="Ivan", godina=2)`	Ivan Horvat je 2. godina
`student(prezime="Kovač", ime="Ivan")`	Ivan Kovač je 1. godina

Argumenti ključnih riječi ne zahtjevaju definirani redoslijed.
Nedefinirani argumenti dobivaju podrazumjevanu vrijednost.

Miješani argumenti

`student("Ivan", godina=2)`	Ivan Horvat je 2. godina
`student("Ivan", "Kovač", godina=2)`	Ivan Kovač je 2. godina

Pozicijski argumenti se definiraju prije argumenata ključnih riječi.

Primjeri krivog pozivanja funkcija

`student()`	pozivanje funkcije bez obaveznih argumenata
`student(ime="Ivan", 2)`	definiranje pozicijskog argumenta nakon onog s ključnom riječi
`student("Ivan", 2, prezime="Kovač")`	dvostruko definiranje argumenta (pozicija 2 i ključna riječ "prezime")
`student(kolegij="Matematika")`	definiranje nepostojećeg parametra

Vidljivost varijabli

- **globalne** varijable definirane su u glavnom tijelu programa i vidljive su svim funkcijama
  - varijable definirane unutar neke petlje (npr. iteratori) vidljive su i izvan te petlje
  - varijable definirane u bloku koda unutar grananja vidljive su i izvan grananja
- **lokalne** varijable definirane su unutar neke funkcije i vidljve su toj funkciji i njenim pod-funkcijama
  - varijable definirane unutar funkcije mogu se postaviti globalnima korištenjem ključne riječi `global`

`global` *`variable`*	postavljanje `variable` globalnom
`variable``=`*`value`*	definiranje vrijednosti varijable

Ako je određena varijabla definirana globalno, a zatim i više puta lokalno (rekurzivno) unutar funkcija, njena vrijednost u najunutarnjijoj funkciji imat će "najlokalniju" vidljivu vrijednost.

Liste

`list` `= [1, 2, 3]`	definiranje liste
`list` `= [[1, 2, 3], [1, 2, 3], [1, 2, 3]]`	definiranje ugniježđene liste

Liste su **uređene** strukture podataka što znači da je postoji redoslijed članova.
Članovi liste ne moraju biti isti tipovi podataka.
Liste su identiteno-promijenjive...

```
» list1 = [12, 9, 3, 7]
» list2 = list1
» id(list1) == id(list2)
True
» list1.append(1)
» list2
[12, 9, 3, 7, 1]
» id(list1) == id(list2)
True
```

... što znači da promjena jedne veže promjenu druge.

Osnovne operacije s listama

`list[index]`	indeksiranje lista
`list[[start]:[end][:step]]`	komad liste
`list[index] = value`	postavljanje nove vrijednosti člana niza
`list[[start]:[end][:step]] = list`	postavljanje nove vrijednosti komada liste s drugom listom (brisanje i umetanje)
`[1, 2, 3] + [4, 5, 6]`	povezivanje listi (`[1, 2, 3, 4, 5, 6]`)
`[1, 2, 3] * 2`	umnožavanje listi (`[1, 2, 3, 1, 2, 3]`)
`del list[start[:end][:step]]`	briše član ili komad liste
`color = [255, 43, 19]` `red, green, blue = color`	definiranje liste #... ... i raspakiravanje - pridruživanje po elemantima
`item = [4, "Pizza", "Plain", 16.98]` `quantity, *others, price = item`	definiranje liste ##... ... i raspakiravanje - pridruživanje po elemantima

Posljednji član liste ima indeks -1, predposljednji -2, itd.
Lista se može jednostavo rastaviti ako ima jednak broj elemenata.
Lista se može "složeno" rastaviti tako da jedan element (označen s *) sakuplja sav višak.

Iteriranje kroz liste

`for item in list:`	iteriranje kroz listu
`for item1, item2 in listOfPairs:`	iteriranje kroz listu sastavljenu od parova, tipa `[[1, 2], [3, 4], [5, 6]]`

Metode nad listama

`list.append(object)`	✗	dodavanje objekta na kraj **originalne** liste
`list.extend(iterable)`	✗	dodavanje rastavljene iterable na kraj **originalne** liste
`list.insert(index, object)`	✗	dodavanje objekta ispred člana pod indeksom na **originalnoj** listi
`list.index(value)`	✓	vraća prvi indeks na kojem se nalazi **vrijednost**
`list.clear()`	✗	prazni **originalnu** listu
`list.remove(value)`	✗	briše prvi član u **originalnoj** listi koji ima vrijednost *value*
`list.pop([index])`	✓	uklanja zadnji član u **originalnoj** listi (član pod indeksom) i vraća uklonjenu vrijednost
`list.count(value)`	✓	ispisuje broj ponavljanja "value"
`list.reverse()`	✗	invertira **originalnu** listu
`list.sort([reverse=True])`	✗	(naopako) sortira **originalnu** listu
`list.copy()`	✓	kopira listu (korisno jer su liste identiteno-promjenljive)

✗ - metoda **nema** povrat tj. `return`
✓ - metoda **ima** povrat tj. `return`

Rječnici

dict = {
  key: *value*,
  key: *value*
}
	definiranje rječnika

dict = {
  outer_key: {
  inner_key: *value*,
  inner_key: *value*
  },
  outer_key: {
  inner_key: *value*,
  inner_key: *value*
  }
}
	definiranje ugniježđenog rječnika

key mora biti **nepromjenljivi** tip objekta, *value* može biti bilo koji tip objekta.
Ako liste promatramo kao parove indeks-vrijednost, onda rječnike možemo promatrati kao parove ključ-vrijednost.
Drugim riječima, u listama je vrijednost pohranjena na lokaciji indeksa, a u rječnicima na lokaciji ključa.
Iz tog razloga, rječnici služe za grupiranje podataka, ali rječnici nisu **nisu uređeni** objekti. Rječnici su identiteno-promijenjivi...

```
» dict1 = 1:  "one"
  » dict2 = dict1
» id(dict1) == id(dict2)
        True
  » dict2[2] = "two"
      » dict1
{1:  "one", 2:  "two"}
» id(dict1) == id(dict2)
```

`True` ... što znači da promjena jedne veže promjenu druge.

Osnovne operacije s rječnicima

`dict[key]`	"indeksiranje" rječnika, odnosno dohvaćanje **vrijednosti** ključa
`dict[key] = value`	postavljanje nove **vrijednosti** već postojećeg ili novog **para**
`del dict[key]`	briše **par**
`dict3 = {**dict1, **dict2}`	spajanje rječnika 1 i 2 u rječnik 3
`dict3 = dict1 | dict2`	spajanje rječnika 1 i 2 u rječnik 3

Ključevi moraju biti jedinstveni.
Prilikom manipulacije rječnika, mijenjaju se njihove **vrijednosti**, a ne **ključevi**.

Metode nad rječnicima

`dict.get(key)`	✓	vraća vrijednost ključa ako taj ključ postoji, a u suprotnom vraća `None`
`dict.pop(key)`	✓	uklanja **par** ključ-vrijednost u **originalnom** rječniku i vraća uklonjenu **vrijednost**
`dict.popitem()`	✓	uklanja zadnje dodan **par** u **originalnom** rječniku i vraća uklonjen **par** kao **tuple**
`dict.clear()`	✗	prazni **originalni** rječnik
`dict.keys()`	✓	vraća "listu" **ključeva** (objekt tipa `dict_keys`)
`dict.values()`	✓	vraća "listu" **vrijednosti** (objekt tipa `dict_values`)
`dict.items()`	✓	vraća "listu tupleova", tj. "listu" **parova** (objekt tipa `dict_items`)
`dict.update(dict)`	✗	osvježavanje **originalnog** rječnika s parovima drugog rječnika

✗ - metoda **nema** povrat tj. `return`
✓ - metoda **ima** povrat tj. `return`

Iteriranje kroz rječnike

`for key in dict:`	iteriranje kroz rječnik po **ključevima**
`for key in dict.keys():`	iteriranje kroz rječnik po **ključevima**
`for value in dict.values():`	iteriranje kroz rječnik po **vrijednostima**
`for key, value in dict.items():`	iteriranje kroz rječnik po **parovima**

N-terci

`tuple = (1, 2, 3,)`	definiranje n-terca
`tuple = ((1, 2, 3),(1, 2, 3),(1, 2, 3),)`	definiranje ugniježđenog n-terca

N-terci su **uređene** strukture podataka što znači da je postoji redoslijed članova.
Za razliku od listi, kad se jednom kreiraju, ne mogu se mijenjati.
Preporuka je koristiti zarez na kraju zadnjeg elemeneta. Ako postoji samo jedan element, zarez je **obavezan**.
Elementi n-terca mogu biti bilo koji tipovi objekta.

Iteriranje kroz n-torke

`for item in tuple:`	iteriranje kroz n-torku
`for item1, item2 in tupleOfPairs:`	iteriranje kroz n-torku sastavljenu od parova, tipa `((1, 2), (3, 4), (5, 6))`

Osnovne operacije s n-tercima

`tuple[index]`	indeksiranje n-terca
`tuple[[start]:[end][:step]]`	komad liste
`color = (255, 43, 19)` `red, green, blue = color`	definiranje n-torke #... ... i raspakiravanje - pridruživanje po elemantima
`item = (4, "Pizza", "Plain", 16.98)` `quantity, *others, price = item`	definiranje n-torke ##... ... i raspakiravanje - pridruživanje po elemantima

N-torka se može jednostavo rastaviti ako ima jednak broj elemenata.
N-torka se može "složeno" rastaviti tako da jedan element (označen s *) sakuplja sav višak.

Metode nad n-tercima

`tuple.index(value)`	✓	vraća prvi indeks na kojem se nalazi **vrijednost**
`tuple.count(value)`	✓	ispisuje broj ponavljanja **vrijednosti**

✗ - metoda **nema** povrat tj. `return`
✓ - metoda **ima** povrat tj. `return`

Skupovi

`set = {1, 2, 3}`	definiranje skupa
`set = set()`	definiranje praznog skupa (jer je `{}` zauzeto za definiranje rječnika)

Svi elementi skupa moraju biti **nepromjenljivi** tipovi objekta.
Set je kao rječnik, no ključevi nemaju par.
Setovi se ne mogu indeksirati jer su elementi "nasumično poslagani".
Set se zbog **sintakse definiranja**, **neindeksiranja** te **jedinstvenosti** članova može promatrati kao **niz ključeva**.

Osnovne operacije sa skupovima

`set = set(list)`	pretvaranje liste u skup kako bi se uklonili duplikati

Metode nad skupovima

`set.add(value)`	✗	dodaje **vrijednost** u **originalni** skup
`set.remove(value)`	✗	uklanja **vrijednost** u **originalnom** skupu i generira grešku ako je nema
`set.discard(value)`	✗	uklanja **vrijednost** u **originalnom** skupu, ali ne generira grešku ako je nema
`set.clear()`	✗	briše sadržaj liste
`set.len()`	✓	vraća broj elemenata skupa
`set1.union(set2)`	✓	**unija**, `set1` ∪ `set2` (sve vrijednosti skupova) (radi s kopijom skupa `s1`)
`set1.intersection(set2)`	✓	**presjek**, `set1` ∩ `set2` (zajedničke vrijednosti skupova) (radi s kopijom skupa `s1`)
`set1.difference(set2)`	✓	**skupovna razlika**, `set1` \ `set2` (jedinstveni elementi `set1`) (radi s kopijom skupa `s1`)

✗ - metoda **nema** povrat tj. `return`
✓ - metoda **ima** povrat tj. `return`

Operacija sa skupovima

`set1 | set2`	**unija**, `set1` ∪ `set2` (sve vrijednosti skupova)
`set1 & set2`	**presjek**, `set1` ∩ `set2` (zajedničke vrijednosti skupova)
`set1 - set2`	**skupovna razlika**, `set1` \ `set2` (jedinstveni elementi `set1`)

Ovo je alternativi način rada sa skupovima. Drugi način je korištenjem metoda `intersection`, `union` i `difference`.

Tipovi grešaka

`SyntaxError`	korištenje zabranjenih znakova (kao što je @), indentacijske greške, nepostojeće ":" nakon petlji, grana
`NameError`	korištenje nepostojećih naredbi, ključnih riječi ili varijabli
`IndexError`	pokušaj pristupanja nepostojećem indeksu liste ili n-torke
`KeyError`	pokušaj pristupanja nepostojećeg ključa u rječniku
`TypeError`	pokušaj manipulacije pogrešnog tipa objekata, npr. zbrajanja integera i stringa
`ValueError`	korištenje pogrešne vrijednosti (ali dobrog tipa) unutar funkcije

Greške

`Raise SyntaxError("`*message*`")`	podiže grešku tipa `SyntaxError` i ispisuje poruku
`Raise NameError("`*message*`")`	podiže grešku tipa `NameError` i ispisuje poruku
`Raise IndexError("`*message*`")`	podiže grešku tipa `IndexError` i ispisuje poruku
`Raise KeyError("`*message*`")`	podiže grešku tipa `KeyError` i ispisuje poruku
`Raise TypeError("`*message*`")`	podiže grešku tipa `TypeError` i ispisuje poruku
`Raise ValueError("`*message*`")`	podiže grešku tipa `ValueError` i ispisuje poruku

`Raise` se ne koristi zbog korisnika program već zbog ostalih koji na programu rade.
Koristi se kako bi se prekinulo daljnje izvršavanje koda.

try i except

`try:`
`    <code block>`
`except [ErrorType]:`
`    <code block>`
	kod koji bi potencijalno mogao generirati grešku
	
	kod koji se izvršava ako se u gornjem bloku generira greška [tipa ErrorType]

`try:`
`    num = int(input("Unesite broj:  "))`
`except ValueError:`
`    num = 1`
`    print("Pogrešan unos.  Odabran je 1.")`
`except EOFError:`
`    num = 1`
`    print("Izlazak iz programa.  Odabran je 1.")`
	kod koji će generirati grešku ako korisnik unese string
	
	broj koji se odabire ako je korisnik unio string
	
	broj koji se odabire ako je korisnik izašao iz programa

Korištenjem `try` i `except` blokova, izbjegnuto je prekidanje izvršavanja koda.

Pristupi programiranju

`year = input("Enter a year:  ")`
`if year.isnumeric():`
`    year = int(year)`
`else:`
`    year = 2025`
	`Look Before You Leap (LBYL)`

`try:`
`    year = int(input("Enter a year:  "))`
`except ValueError:`
`    year = 2025`
	`Easier to Ask Forgiveness than Permission (EAFP)`

`EAFP` se smatra "više Pythonski".

Uvoz cijelih modula

`import random` `random.randint(1, 100)`	`import random as rand` `rand.randint(1, 100)`	uvoz **modula** `random` (pod nazivom `rand`) korištenje **metode** `randint` **modula** `random`
`import calendar` `calendar.isleap(2023)`	`import calendar as cal` `cal.isleap(2023)`	uvoz **modula** `calendar` (pod nazivom `cal`) korištenje **metode** `isleap` **modula** `calendar`
`import math` `math.sqrt(2)`	`import math as m` `m.sqrt(2)`	uvoz **modula** `math` (pod nazivom `m`) korištenje **metode** `sqrt` **modula** `math`

Moduli su Python skripte koje **importanjem** donose nove funkcionalnosti.
Import se može shvatiti kao da se cijela skripta zalijepi u header.
Nakon uvoza, dostupne su sve metode, funkcije i varijable te skripte.
Moduli su tipa `class 'module'`

Uvoz specifične metode modula

`from random import randint` `randint(1, 100)`	`from random import randint as ri` `ri(1, 100)`	uvoz **metode** `randint` (pod nazivom `ri`) korištenje **metode** `randint`
`from calendar import isleap` `isleap(2023)`	`from calendar import isleap as il` `il(2023)`	uvoz **metode** `isleap` (pod nazivom `il`) korištenje **metode** `isleap`
`from math import sqrt` `sqrt(2)`	`from math import sqrt as sq` `sq(2)`	uvoz **metode** `sqrt` (pod nazivom `sq`) korištenje **metode** `sqrt`

Argument od `from` je **modul**, odnosno **Python skripta**.
Argument od `import` je **funkcionalnost**, odnosno **metoda, funkcija ili varijabla** iz te Python skripte.
Moguće je uvesti i **više** metoda nekog modula tako se **metode odvoje zarezima**.
Moduće je uvesti i **sve** metode nekog modula tako da se **zadaje argument** `*`. To je korisno jer zatim nije potrebno koristiti ime modula kao prefiks.

Uvoz druge skripte

`import script`
`script.func()`
`script.var`
	`from script import func`
	`func()`
	`var`
		uvoz **skripte** `script.py`
		korištenje **funkcije** `func` definirane u `script.py`
		korištenje **varijable** `var` definirane u `script.py`

Skripte koja se uvozi mora se nalaziti **u istom direktoriju** kao i skripta u koju se uvozi.

`pip` modul

`sudo apt install -y python3-pip`	instalacija `pip` modula
`python3 -m pip --version`	provjera verzije `pip` modula (za Python 3)
`python3 -m pip install` *`package`*	instalacija paketa pomoću `pip` modula (za Python 3) s **pypi.org**

Nakon instalacije, paket se može standardno uvoziti u Python s `import` *`package`*.

Klase i objekti - definiranje

`class Dog:`	definiranje **klase** `Dog`
`    species = "C. familiaris"`	definiranje **atributa klase**, `species`, zajedničkog svim instancama klase
`    num = 0`	definiranje **atributa klase**, `num`, zajedničkog svim instancama klase
`    location = "Pet centre"`	definiranje **atributa klase**, `location`, zajedničkog svim instancama klase
`    @classmethod`	dekorator koji govori da se iduća definicija funkcije odnosi na cijelu klasu
`    def relocate(cls, new_location):`	definiranje **metode** koja se može izvršavati nad **cijelom klasom** (ali i njeim instancama)
`        cls.location = new_location`	
`        return None`	
`    def __init__(self, name):`	definiranje **metode** `__init__`, koja **inicijalizira** pojedinu **instancu** klase
`        self.name = name`	definiranja **atributa instance**, **name** dodijeljenog prilikom inicijalizacije (iz argumenta)
`        self.tricks = []`	definiranja **atributa instance**, **tricks** dodijeljenog prilikom inicijalizacije
`        Dog.num += 1`	promjena **atributa klase**, **species** prilikom svake inicijalizacije instance
`    def learn(self, new_trick):`	definiranje **metode** koja se može izvršavati nad instancama klase
`        self.tricks.append(new_trick)`	
`        return None`	
`    def perform(self, trick):`	definiranje **metode** koja se može izvršavati nad instancama klase
`        print(f"{self.name} performs {trick}!")`	
`        return None`	

Kod definiranja metoda i init-a, nužno je postaviti parametar `self`.
`self` znači da se odnosi na **pojedinu instancu klase**.
Svaka **metoda** ima uvijek ima jedan "nevidljivi" argument - koji pripada parametru `self`.
Nad svim pripadnicima određene **klase** mogu se izvršavati pripadajuće **metode**.
Analogno vrijedi i za klase, ali ulogu `self`-a preuzima `cls`

Podklase (nasljeđivanje)

`class Puppy(Dog):`	definiranje **podklase** `Puppy`, **klase** `Dog`
`    def __init__(self, name, age):`	definiranje **metode** `__init__`, koja **inicijalizira** pojedinu **instancu podklase**
`        super().__init__(name)`	pozivanje `__init__` metode nad **klasom** (lat. *super* - iznad), s argumentom `name` **podkalse**
`        self.age = age`	definiranja **atributa** `age`, svojstvenog **podklasi**, definiranog **inicijalizacijom podklase**
`    def whine(self):`	definiranje **metode** koja je svojstvena samo **podklasi**
`        print(f"{self.name}" whines!)`	
`        return None`	

Podklasa nasljeđuje sve funkcionalnosti **klase**, uz dodatak sebi svojstvenih.
Podklasa ne mora imati i vlastiti `__init__` - može naslijediti samo onaj od **klase**.
`super().__init__(name)` nema argument `self` jer se `self` upisuje samo tijekom **definiranja metode**, dok se ovdje **poziva metoda**. Metoda se poziva, naravno, bez argumenta `self`, kao i u "tijelu" skripte.

Klase i objekti - korištenje

`print(Dog.species)`	ispis **atributa klase** `Dog`
`Dog.relocate("Zoo City")`	izvršavanje **metode** `relocate` nad klasom `Dog`
`elton = Dog("Elton", "Australian Shepherd")`	inicijalizacija **instance** `elton` uz argumenate
`print(elton.name)`	ispis **atributa instance** `elton`
`print(elton.species)`	ispis **atributa instance** `elton`, odnosno **atributa klase**, budući da joj pripada
`elton.learn("sit")`	izvršavanje **metode** `learn` nad instancom `elton`
`elton.perform("sit")`	izvršavanje **metode** `perform` nad instancom `elton`

OOP

`isinstance(`*`object`*`, `*`class`*`)`	provjerava pripadnost **objekta klasi**