README.md

Uvod v programiranje v Python-u

KŠOK - Uvod v programiranje v programskem jeziku Python, 2021

V mesecu maju je potekalo 6 predavanj. Soočli smo se z vsemi potrebnimi osnovami programiranja v Pythonu, od podatkovnih tipov, zank pa vse do uporabe zunanjih knjižnic. Pri zadnjem predavanju so znanje uporabili na prikazovanju podatkov z uporabo knjižnice matplotlib.

Kontakt predavatelja:
Liam Mislej
liammislej@gmail.com

Uporabni viri in informacije:

Tečaj je sledil sklopom iz spodnjega vira, s tem da smo sklope podrobneje predelali. https://www.w3schools.com/python/default.asp

Python poberemo na naslovu: https://www.python.org/

Vsi uporabljeni oz. napisani programi se nahajajo v mapi Koda.

Uvod v programiranje, osnovni pojmi, števila, nizi in ostale osnove

Python

Python je interpretni visokoravni večnamenski programski jezik. Interpretni (interpreted) pomeni, da se napisana koda izvaja postopoma brez potrebe po kompilaciji programa. Pomembno je vedeti, da se koda izvaja zaporedoma po vrsticah (prvo se izvede prva vrstica, nato druga, tretja, ...)

Nekateri programski jeziki uporabljajo razne znake (`{ }, ;, ...`) za ločevanje blokov kode. Pri Pythonu to ni tako; uporablja se indentacijo (indente) za ločevanje blokov. Indent je le niz, ki ga (ponavadi) sestavljajo štirje presledki, Krajše napišemo oz. natipkamo z tipko `Tab` , primer:

Python

```
# To ni indentirano in vrne napako
if True:
print("Hello world")


# To je pravilno indentirano
if True:
    print("Hello world")
```

Primer iste kode v programskem jeziku Java kjer se bloke deli z uporabo (`{}`):

```
class HelloWorld {
    public static void main(String[] args) {
        if (true) {
            System.out.println("Hello, World!");
        }
    }
}
```

Pri javi ni potrebno bloke indentirati kot je prikazano odzgoraj. Če bi želeli ba lahko zgornjo kodo napisali v eni sami vrstici, vendar uporabljamo indente zaradi berljivosti.

Seveda bo vse bolj jasno, ko se bomo s tem soočili pri predavanjih in vajah, ko bomo pisali kodo.

Komentiranje (comments)

V Pythonu komentarje označimo na naslednja dva načina:

- Z znakom `#` na začetku vrstice, uporabljamo za enovrstične komentarje
- Z uporabo treh dvojnih narekovajev `"""` na začetku in koncu komentarja, uporabljamo za večvrstične komentarje

```
# To je enovrstični komentar

""" To je večvrstični komentar.
    Še ena vrstica!  """
```

Spremenljivke (variables)

Spremenljivke so lahko kateri koli niz ali pa ena sama črka. V spremenljivkah hranimo vse podatke kot so števila, nizi, seznami in celo funkcije. Spremenljivko definiramo z enačajem. Pišemo jih kot navadne besede. Ne smejo vsebovati presledkov ali se začeti z številom. V Pythonu se držimo ne napisanega pravila, da kadar spremenljivka vsebuje več besed jih povežemo z podčrtaji, prav tako spremenljivke v določenih primerih začnemo z podčrtaji.

Primer:

```
a = 10
to_je_spremenljivka = "Niz"
_spremenljivka = "Niz"
__spremenljivka = "Niz"

# Tega ne počnemo:
00spremenljivka = "Niz"
to je spremenljivka = "Niz"
!tojespremenljivka = "Niz"
```

Podatkovni tipi (data types)

Podatke ločimo na več tipov, vse hranimo v spremenljivkah in vsakemu posameznemu tipu pripadajo določene metode in lastnosti.

- int = cela števila 1, 2, 0, -1, -2300 ...

  ```
  a = 10
  b = 0
  c = 6
  ```

- float = decimalna števila 2.13, 1.5, 0.75, -2.33 ...

  ```
  a = 10.4
  b = 0.5
  c = 6.123
  ```

- str = nizi in črke "a", "To je niz", "0.24"

  ```
  a = 'a'
  b = "beseda"
  c = "Celoten stavek."
  d = "24"  # To ni število vendar niz
  ```

- list = seznam [1, 3, 4], [[1, 2, 3], [4, 5, 6]]

```
a = [1, 2, 3]
b = [1, 1, 1, 1, 0]
```

- tuple = n-terka (1, 2), (3, 4, 5)

```
a = (1, 2)
b = (1, 2, 3, 4)
c = (,1)
```

- set = množica {3, "b", "c"}

```
a = {3, 4, 10, "neki"}
```

- dict = slovar {"ključ": podatek}

```
a = {"ključ 1": 5, "ključ 2": 7}
```

- bool = binarna vrednost True, False

```
a = True = 1
b = False = 0
```

Print in Input

Če želimo nek niz, število, seznam, podatek shranjen v spremenljivki ... izpisati na konzolo to naredimo z uporabo ukaza print()

```
>>> print("Hello world!")
Hello world!

>>> print(42)
42
```

Če želimo iz konzole shraniti nek vhod(nekaj kar napišemo) uporabimo ukaz input(), program pri tej vrstici počaka na vhodni podatek, ko je ta napisan, ga vrne. Podatek pa lahko shranimo v spremenljivko.

Opomba: Podatek se vedno shrani kot niz, torej tipa str.

```
>>> vhodni_podatek = input()
>>> # Napišemo npr.: Pozdravljeni
>>> print(vhodni_podatek)
Pozdravljeni
```

Pretvorbe podatkovnih tipov (casting)

Če želimo pretvoriti določen podatkovni tip v nekega drugega to storimo z ukazi, ki jih pišemo enako kot željeni tip. Torej `str(), int(), list(), ...` .

Tip podatka ali spremenljivke preverimo z ukazom type().

```
>>> a = '5'
>>> print(type(a))
str

>>> b = 5
>>> print(type(b))
int

>>> c = int(a)  # Niz shranjen v sprem. a, torej "5", pretvorimo v celo število 5
```

```
>>> print(type(c))
>>> int
>>> print(c)
5

>>> d = str(b)  # Število shranjeno v sprem. b, torej 5, pretvorimo v niz
>>> print(type(d))
str
>>> print(d)
5  # Nize Python izpisuje brez narekovajev, zato pri tem izpisu izgleda, kot da je to število čeprav je niz

>>> decimalno_stevilo = 2.5
>>> celo_stevilo = int(decimalno_stevilo)
>>> print(type(decimalno_stevilo))
float
>>> print(type(celo_stevilo))
int

>>> print(celo_stevilo)
3
# Python, kadar pretvarjamo iz decimalnih števil v cela števila, ta zaokroži navzgor po osnovnem pravilu zaokroževanja.
```

Operaterji (operators)

Operaterje uporabljamo za izvajanje operacij med podatkovnimi števili in spremenljivkami.

Arithmetic Operators:

Operator	Name	Example
+	Addition	x + y
-	Subtraction	x - y
*	Multiplication	x * y
/	Division	x / y
%	Modulus	x % y
**	Exponentiation	x ** y
//	Floor division	x // y

Assignment Operators:

Operator	Example	Same as
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3

Comparison Operators:

Operator	Name	Example
==	Equal	x == y
!=	Not equal	x != y
>	Greater than	x > y

Operator	Name	Example
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

Logical Operators:

Operator	Description	Example
and	Returns True if both statements are true	x < 5 and x < 10
or	Returns True if one of the statements is true	x < 5 or x < 4
not	Reverse the result, returns False if the result is true	not(x < 5 and x < 10)

Identity Operators:

Operator	Description	Example
is	Returns True if both variables are the same object	x is y
is	not Returns True if both variables are not the same object	x is not y

Membership Operators:

Operator	Description	Example
in	Returns True if a sequence with the specified value is present in the object	x in y
not	in Returns True if a sequence with the specified value is not present in the object	x not in y

Nizi (strings)

Kot pri vseh ostalih programskih jezikih so nizi seznami byte-ov, ki predstavljajo znake.

Nize označujemo z enojnimi narekovaji `'Niz'` ali dvojnimi narekovaji `"Niz"`. Ne pisano pravilo je, da se posamezne znake označuje z enojnimi narekovaji, več znakov v vrstici pa z dvojnimi narekovaji. Nize lahko shranimo v spremenljivke.

```
>>> niz1 = 'a'
>>> niz2 = "To je niz!"

>>> print(niz1)
a

>>> print(niz2)
To je niz!
```

Večvrstične nize pišemo podobno kot večvrstične komentarje, z uporabo treh dvojnih(ali enojnih) narekovajev.

```
>>> niz1 = """ To je niz
>>> napisan v dveh vrsticah. """

>>> print(niz1)
To je niz napisan v dveh vrsticah.
```

Opazimo, da se prelom v novo vrstico ni izpisal. To pa zato, ker Python tipkane prelome ignorira in se za to uporablja posebna oznaka `\n`.

```
>>> print("To se bo \n prelomlo v novo vrstico.")
To se bo
 prelomlo v novo vrstico.
```

String Operations:

Nize lahko med drugim seštevamo in množimo.

```
>>> niz1 = 'a' + '+' + 'b'
>>> print(niz1)
a+b

>>> niz2 = 'c' * 3
>>> print(niz2)
ccc
```

Strings are Arrays:

Nizo so pravzaprav indeksirani seznami.

Dolžino niza oz. število znakov preverimo z uporabo funkcije `len()`.

```
>>> dolzina = len("a bc1")
>>> print(dolzina)
5

>>> niz1 = "niz"
>>> print(len(niz1))
3
```

Nize lahko pretvorimo v sezname z uporabo funkcije `list()`, kjer je vsak znak posamezen element v seznamu.

```
>>> print(list("niz"))
['n', 'i', 'z']
```

Pri nizih velja indeksiranje, kar pomeni, da če želimo izvedeti ali shraniti znak na določenem mestu pišemo `[i]` zraven spremenljivke oz. niza, kjer je i celo število mesta na katerem se znak nahaja. Indeksiranje se začne z številom 0, tako da je znak na 1. mestu ub. na indeksu 0.

```
>>> print("niz"[0])
n

>>> niz1 = "To je ena vrstica!"
>>> print(niz1[1])
o

# Če želimo izvedeti zadnji znak pišemo kar -1
>>> print(niz1[-1])
!
```

Podatkovni tipi in zanke

Seznami (lists/arrays)

Sezname uporabljamo za shranjevanje več podatkov v spremenljivkah. Posameznemu podatku pravimo element(item). Seznami so urejeni(elementi imajo definiran vrstni red, kateri se ne spremeni), možno je elemente spreminjati in dovoljujejo ponavljajoče podatke. Posamezen element v seznamo je lahko kateri koli podatkovni tip(int, str, celo še en seznam, tuple, ... , itd.)

Ustvarjanje seznama:

Seznam ustvarimo z oglatimi oklepaji in ga shranimo v spremenljivko:

```
>>> prazen_seznam = []
>>> print(prazen_seznam)
[]

>>> seznam_z_vec_elementi = [1, 3, "niz", [4, 5, 6]]
>>> print(seznam_z_vec_elementi)
[1, 3, "niz", [4, 5, 6]]
```

Dostopanje do podatkov(Indexing):

Do podatkov dostopamo z indeksiranjem, kjer pišemo [indeks] zraven spremenljivke oz. seznama, kjer je indeks celo število. V programiranju se štetje začne od 0 naprej, tako se element na 1. mestu v seznamu nahaja na indeksu 0, na 2. mestu na indeksu 1, ..., itd.

```
>>> seznam = [1, 2, 3, 4]
>>> print(seznam[0])
1
>>> print(seznam[2])
3

# Vrednost na določenem indeksu lahko shranimo v spremenljivko
>>> vrednost = seznam[1]
>>> print(vrednost)
2

# Spreminjanje vrednosti posameznega elementa
>>> seznam[0] = 10
>>> print(seznam)
[10, 2, 3, 4]

# POZOR: Če dostopamo do podatko z indeksom, ki presega <code>dolžino seznama - 1</code>(-1, ker štejemo od 0 pri indeksiranju) n
>>> seznam = [1, 2, 3, 4]
>>> print(seznam[4])
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    print(seznam[4])
IndexError: list index out of range
```

Dolžina seznama(list size) Dolžino seznama prederemo z uporabo funkcije `len()`, ta vrne število elementov in to lahko shranimo v spremenljivko.

```
>>> print(len([1, 2, 3, 4, 5]))
5

>>> seznam = [1, 2, 3, 4, 5, 6]
>>> dolzina = len(seznam)
>>> print(dolzina)
6
```

Dodajanje podatkov v seznam:

- metoda append doda element na konec seznama, kot parameter prejme element, pišemo kot: `seznam.append(novi_element)`
- metoda insert vrine element pred določenim indeksom, prejme dva parametra indeks in element, pišemo kot: `seznam.insert(indeks, novi_element)`

```
>>> seznam = [1, 2, 3, 4, 5]
>>> seznam.append(6)
>>> print(seznam)
[1, 2, 3, 4, 5, 6]

>>> seznam = [1, 2, 3, 4, 5]
>>> seznam.insert(1, "element")
>>> print(seznam)
[1, "element", 2, 3, 4, 5]
```

Odstranjevanje elementov:

- metoda remove odstrani željen element iz seznama, kot parameter prejme element, pišemo kot: `seznam.remove(element)`
- metoda pop odstrani element na željenem indeksu, kot parameter prejme indeks, pišemo kot: `seznam.pop(indeks)`

```
>>> seznam = ["kšok", "python", "tečaj"]
>>> seznam.remove("kšok")
>>> print(seznam)
["python", "tečaj"]
```

```
>>> seznam = ["košok", "python", "tečaj"]
>>> seznam.pop(1)
>>> print(seznam)
["košok", "tečaj"]
```

Seštevanje seznamov: Sezname lahko seštevamo z uporabo operatorja `+`. Vrstni red je tle pomemben saj seznama zlepi skupaj.

```
>>> seznam1 = ["košok", "python", "tečaj"]
>>> seznam2 = [1, 2, 3]
>>> seznam_sesteti = seznam1 + seznam2
>>> print(seznam sesteti)
["košok", "python", "tečaj", 1, 2, 3]
```

Urejanje seznamov(Sorting):

Sezname lahko uredimo po velikosti, z uporabo metode sort, pišemo: `seznam.sort()`. To uredi seznam po velikosti tako, da je najmanjši element na prvem mestu(indeksu 0) največji pa na zadnjem. V kolikor želimo obratno sortiranje(od največjega do najmanjšega kot parameter podamo `reverse=True`. Metoda sezname uredi alfanumerično(prvo po številih), oz. če seznam vsebuje le nize, po abecedi(kjer je 'a' najmanjša možna vrednost)

```
>>> seznam = [2, 3, 4, 5, 1, 0]
>>> seznam.sort()
>>> print(seznam)
[0, 1, 2, 3, 4, 5]

>>> seznam = [2, 3, 4, 5, 1, 0]
>>> seznam.sort(reverse=True)
>>> print(seznam)
[5, 4, 3, 2, 1, 0]

>>> seznam = ["a", "1a", "b"]
>>> seznam.sort()
>>> print(seznam)
['1a', 'a', 'b']
```

List comprehension:

Sezname lahko definiramo tudi enovrtično, to je v določenih primerih celo bolje. To naredimo tako, da v seznam kar napišemo for zanko, spremenljivka, ki se sprehaja se shrani na določen indeks(po vrstnem redu). Najbolje prikazati direktno na primeru:

```
seznam = [i for i in range(10)]

print(seznam)

# Izpiše:
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Lahko vstavimo tudi pogojne stavke pred for zanko:

```
seznam = ["Manj od 4" if i < 4 else i for i in range(10)]

print(seznam)

# Izpiše:
['Manj od 4', 'Manj od 4', 'Manj od 4', 'Manj od 4', 4, 5, 6, 7, 8, 9]
```

Tupli (tuples)

Tuple prav tako uporabljamo za shranjevanje več podatkov v spremenljivkah. Tupli so urejeni(elementi imajo definiran vrstni red, kateri se ne spremeni), dovoljujejo ponavljajoče podatke in podatkov **ni mogoče spreminjati**. Posamezen element v tuplu je lahko kateri koli podatkovni tip(int, str, celo še en seznam, tuple, ... , itd.)

Ustvarjanje tupla:

Tuple ustvarimo z okroglimi oklepaji in ga shranimo v spremenljivko:

```
>>> primer_tupla = (1, "a", "Ksok", 3.14)
>>> print(primer_tupla)
(1, "a", "Ksok", 3.14)

>>> tuple_en_element = ("Niz",)  # Opazimo vejico, brez vejice bi Python ignoriral oklepaj in se to nebi shranilo kot tuple
>>> print(tuple_en_element)
("niz")
```

Dostopanje do podatkov(Indexing):

Do podatkov dostopamo z indeksiranjem, kjer pišemo [indeks] zraven spremenljivke oz. seznama, kjer je indeks celo število. V programiranju se štetje začne od 0 naprej, tako se element na 1. mestu v seznamu nahaja na indeksu 0, na 2. mestu na indeksu 1, ..., itd.

```
>>> primer = (1, 2, 3)
>>> print(primer[0])
1

>>> primer2 = (2, 4, 6, 7)
>>> print(primer2[3])
7

# Do zadnjega elementa v seznamih in tuplih lahko dostopamo tudi tako, da pišemo -1 za indeks
>>> primer3 = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
>>> print(primer3[-1])
10
```

Dolžina tupla(tuple size) Dolžino tupla preberemo z uporabo funkcije `len()`.

```
>>> print(len((1, 2, 3)))
3

>>> seznam = (1, 23, 4, 5, "Niz", [1, 2, 3])
>>> dolzina = len(seznam)
>>> print(dolzina)
6
```

Množice (sets)

Množice niso urejene(elementi nimajoimajo definiran vrstni red, saj se lahko spremeni), ne dovoljujejo ponavljajoče podatke in podatkov ni mogoče spreminjati. Posamezen element v množici je lahko kateri koli podatkovni tip(int, str, celo še en seznam, tuple, ... , itd.)

Ustvarjanje množice:

Množico ustvarimo z zavitimi oklepaji in jo shranimo v spremenljivko:

```
>>> primer_mnozice = {1, 3, "neki", True, "neki"}
>>> print(primer_mnozice)
{1, 3, "neki", True}  # Opazimo da se pojavi le en niz "neki", to pa zato ker v množici ne dovoljujemo ponavljajočih podatkov.
```

Dostopanje do podatkov:

Do podatkov v množici lahko dostopamo z iteracijo oz. z uporabo for zanke, lahko pa preverimo če je podatek vsebovan v množici.

```
>>> mnoz = {"podatek1", 2, 3, "kšok"}
>>> print("kšok" in mnoz)  # Preverimo če je podatek "kšok" v množici
True

# Iteriramo skozi množico
>>> for podatek in mnoz:
>>>   print(podatek)
"podatek1"
2
3
"ksok"
```

Dodajanje elementov v množico

V množico dodajamo elemente tako da na spremenljivki kjer jo hranimo uporabimo metodo add, primer: `mnoz.add(element)`.

```
>>> mnoz = {1, 2, 3}
>>> mnoz.add("Kšok")
>>> print(mnoz)
{1, 2, 3, "Kšok"}
```

Odstranjevanje elementov iz množice

Elmente iz množice odstranimo z uporabo metode remove, primer: `mnoz.remove("Kšok")`.

```
>>> mnoz = {1, 2, "Kšok", 3}
>>> mnoz.remove("Kšok")
>>> print(mnoz)
{1, 2, 3}
```

Združevanje množic

Množice lahko združujemo na dva načina, z uporabo metode union, ki naredi unijo med množicami ali z uporabo metode update. To naredimo tako, da na spremenljivki kateri želimo dodati elemente iz neke druge množice uporabimo zgornji metodi.

```
>>> mnoz_1 = {1, 2, 3}
>>> mnoz_2 = {"k", "o", "p"}
>>> mnoz_1.union(mnoz_2)
>>> print(mnoz_1)
{1, 2, 3, "k", "o", "p"}

>>> mnoz_1 = {1, 2, 3}
>>> mnoz_2 = {"k", "o", "p"}
>>> mnoz_2.update(mnoz_1)
>>> print(mnoz_1)
{"k", "o", "p", 1, 2, 3}
```

Slovarji (dictionaries)

Slovarje uporabljamo za hranjenje podatkov v parih ključ:podatek(key:value). Slovarji so urejeni(od Python 3.7 naprej), ne dovoljujejo ponavljajočih podatkov in podatke lahko spreminjamo. Prav tako lahko v njih hranimo vse možne tipe podatkov.

Ustvarjanje slovarja:

Slovar ustvarimo z uporabo zavitih oklepajev, posamezne pare ločimo z vejco, ključ v paru pa od podatko ločimo z dvopičjem.

```
>>> slovar = {"ključ1": "podatek1", "ključ2": "podatek2", "ključ3": "podatek3"}
>>> print(slovar)
{"ključ1": "podatek1", "ključ2": "podatek2", "ključ3": "podatek3"}
```

Dostopanje do podatkov:

Do podatkov pri slovarjih dostopamo z ključi in sicer zraven spremenljivke pišemo oglati oklepaj, notri pa ključ. To nam vrne podatek shranjen pri tistem ključu.

```
>>> slovar = {"ključ1": 42, "ključ2": [1, 2, 3], 4: 5}
>>> print(slovar["ključ1"])
42
>>> print(slovar["ključ2"])
[1, 2, 3]
>>> print(slovar[4])
5
```

Spreminjanje podatkov

Podatke spreminjamo tako da pri posameznem ključu definiramo novo vrednost.

```
>>> slovar = {"ključ1": 42, "ključ2": [1, 2, 3], 4: 5}
>>> print(slovar)
{"ključ1": 42, "ključ2": [1, 2, 3], 4: 5}

# Spremenimo vrednost na ključu "ključ2" iz [1, 2, 3] v število 1
>>> slovar["ključ2"] = 1
>>> print(slovar)
{"ključ1": 42, "ključ2": 1, 4: 5}
```

Dodajanje in spreminjanje podatkov

Novi podatek dodamo tako, da enostavno napišemo novi ključ in enačimo z podatkom, enako kot pri spreminjanju podatkov.

```
>>> slovar = {"ključ1": 42, "ključ2": [1, 2, 3], 4: 5}
>>> slovar["ključ4"] = "Novi podatek"
{"ključ1": 42, "ključ2": [1, 2, 3], 4: 5, "ključ4": "Novi podatek"}
```

Podatek iz slovarja odstranimo z uporabo metode pop, kot parameter vstavimo ključ podatka katerega želimo izbrisati.

```
>>> slovar = {"ključ1": 42, "ključ2": [1, 2, 3], 4: 5}
>>> slovar.pop("ključ2")
>>> print(slovar)
{"ključ1": 42, 4: 5}
```

if, elif in else

Z if stavki določamo, če se bo določen blok kode izvedel. Tukaj bodo prvič nastopili indenti, s katerimi ločimo bloke kode. Indenti so skupek presledkov - v Pythonu ponavadi uporabimo 4 presledke ali en klik na tipko `tab`.

If stavke pišemo tako, da najprej napišemo `if` nato `pogoj` in na koncu `:`. Pod samim stavkom pa določen blok kode, ki je seveda indentiran. Pogoj je lahko karkoli, ki vrne boolean(binarno) vrednost, torej `True` ali `False`. V primeru, da pogoj vrne `True` se bo blok kode pod if stavkom izvedel.

Pogoje ponavadi pišemo z uporabo operatorjev tipov:

- Comparisson(<, >, ==, !=, ...)
- Logical(and, or, not)
- Membership(in)

Konkretno na primeru:

Opomba: Zaradi indentacij, naslednji primeri ne bodo več pisani v shell-u, vendar v konkretnem Python programu(datoteka s končnico .py), tako ne bodo več nastopili znaki `>>>`, ki predstavljajo izvedeno kodo. Kadar bo nekaj izpisano, bo to označeno z komentarjem.

```
starost = 17
if starost >= 18:
    print("Starost je večja ali enaka 18")

# To ne izpiše ničesar saj pogoj ni izpovnjen(17 >/= 18)

# Če starost spremenimo na npr. 19
starost = 19
if starost >= 18:
    print("Starost je večja ali enaka 18")

# Izpiše:
Starost je večja ali enaka 18

# V tem primeru se je blok kode pod if stavkom izvedel, saj je pogoj veljal.
```

elif:
Elif stavke uporabljamo kadar potrebujemo dodatne pogoje. Uporabljamo jih lahko le po if stavkih delujejo pa enako. Prav tako jih lahko uporabimo toliko kot želimo. Najprej se izvede if pogoj, če ta ne velja nastopi prvi elif stavek, če še ta ne velja drugi elif stavek, ... in tako do prvega ki velja, ko se ta izvede se vsi preostali ne izvedejo.

```
starost = 19

if starost < 18:
    print("Starost je manj od 18")
elif 18 <= starost < 21:
        print("Starost je več ali enako 18 ampak manj od 21")
elif starost >= 21:
    print("Starost je več ali enako 21")

# Izpiše:
Starost je več ali enako 18 ampak manj od 21
```

else:

Else uporabljamo, kadar želimo, da se izvede del kode kadar noben od zgornjih if, elif stavkov ne velja. Else pišemo zadnje in se izvede le takrat ko se noben zgornji ukaz ne izvede.

```
starost = 46

if starost < 18:
    print("Starost je manj od 18")
elif 18 <= starost < 21:
        print("Starost je več ali enako 18 ampak manj od 21")
elif 21 <= starost < 45:
    print("Starost je več ali enako 21 ampak manj od 45")
else:
    print("Starost je večja ali enaka 45")

# Izpiše:
Starost je večja ali enaka 45
```

Zanke in funkcije

While zanka (while loop)

While zanka se izvaja dokler je pogoj pri njej izpolnjen. V slovenščini bi to lahko brali kot: Dokler je to res, se izvajaj!. Pišemo: `while pogoj:` pri tem gremo v novo vrstico, ki jo indentiramo, da ločimo blok kode. Torej blok kode pod zanko se bo izvajal dokler pogoj v sami definiciji zanke velja. Blok kode se izvaja v krogih, najprej se izvede prva vrstica, nato druga, tretja, ... ko pridemo na konec, se vrne na začetek, preveri če pogoj velja, v primeru da velja, ponovi enako kakor prej.

```
starost = 0

while starost <= 6:
    print(starost)
    starost += 1  # Starosti prištevamo 1, zapisano je enako kakor satarost = starost + 1

# Izpiše:
0
1
2
3
4
5
6

# Izpišejo se le števila do 6, saj ko je število > 6 pogoj (starost <= 6) ne velja več ==> zanka se zaključi
```

Vse zanke lahko zaključimo znotraj same zanke:

Z uporabo ukaza `break`, lahko zanko predhodno zaključimo. In sicer na tistem koraku kjer je ukaz napisan.

Denimo, da ustvarimo neskončno zanko, tako, da kot pogoj pišemo True(kar bo vedno res). V spremenljivki starost hranimo število, na začetku je enako 0. Ob vsakem koraku v zanki, to število povečamo za 1. Znotraj same zanke pa imamo pogoj `if`, ki preverja če je vrednost spremenljivke `starost` presegla vrednost 100. V kolikor pogoj velja zaključimo zanko z uporabo ukaza break.

```
starost = 0

while True:  # Se izvaja v nedogled
    print(starost)
    if starost > 100:
        break
    starost += 1

# Izpiše:
0
1
2
...
98
99
100
101
```

For zanka (for loop)

For zanke uporabljamo za iteriranje skozi sekvence ali iteratorje, kot so seznami, množice, ...

Pišemo: `for i in seznam:`, kjer je `i` spremenljivka ki bo zavzemala določeno vrednost ob vsakem krogu zanke, `in` pomeni v nečemu, `seznam` pa nek seznam z podatki.

```
seznam = ["k", "š", "o", "k"]

for i in seznam:
    print(i)

# Izpiše
k
š
o
k
```

Torej spremenljivka i zavzame vsako vrednost elementov v seznamu po vrstnem redu. Spremenljivko, ki iterira, lahko pišemo poljubno npr.

```
seznam = ["k", "š", "o", "k"]

for crka in seznam:
    print(crka)

# Izpiše
k
š
o
k
```

range: Pri for zankah pogostokrat uporabljamo ukaz `range(od, do, korak)`. Ta ustvari iterator števil, ki si ga lahko predstavljamo kot seznam. Ustvari zaporedna števila od parametra `od` do **in ne vključno** parametra `do`, s koraki `korak`. Parametri `od, do, korak` morajo biti števila.

Range ni seznam, ampak iterator. V to se ne bomo poglabljali, če pa želimo prikazati range kot seznam, ga moramo najprej pretvoriti.

```
>>> print(list(range(0, 10)))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

# Če dodamo še parameter, ki predstavlja korak. Tega nastavimo na vrednost 2, se 'generira' vsako drugo število
>>> print(list(range(0, 10, 2)))
[0, 2, 4, 6, 8]
```

Opazimo, da se število 10 ne izpiše, to pa zato, ker je zadnje število po vrsti, to število ukaz range nikoli ne izpiše.

Če to uporabimo v for zanki:

```
for i in range(1, 5):
    print(i)

# Izpiše
1
2
3
4
```

Funkcije (functions)

Funkcije so bloki kode, ki se izvedejo le ko jih pokličemo.

Funkcijo definiramo z uporabo ukaza `def` za tem podamo ime funkcije in prazna oklepaja `ime_funkcije()` na koncu pa še dvopičje `:` . V blok kode od funkcije se vključuje vse kar je pod definicijo indentirano. Definicije funkcij pišemo brez indentacije in se nahajajo izven zank, if stavkov, ... itd.

Primer:

```
def izpisi_pozdrav():
    print("Pozdravljeni!")
```

Če poženemo .py datoteko, ki vsebje zgornjo definicijo funkcije, se ne izpiše nič. Če želimo, da se funkcija izvede jo moramo poklicati. To storimo tako, da napišemo ime funkcije z oklepaji.

Spremenjen zgornji primer:

```
def izpisi_pozdrav():
    print("Pozdravljeni!")

izpisi_pozdrav()

# Ipiše:
Pozdravljeni!
```

Parametri:

Funkcijam lahko podajamo parametre tako, da jih napišemo v oklepajih. Parametre pišemo kot spremenljivke in jih v definicij funkcije lahko uporabljamo. Število parametrov ni omejeno, torej jih lahko imamo poljubno.

```
def izpisi_stevila(a, b):  # Kot parametre v def. funkcije pišemo spremenljivke
    print(a)  # Te spremenljivke lahko nato uporabimo v sami funkciji
    print(b)

izpisi_stevila(5, 4)  # Ko pokličemo funkcijo podamo konktretne vrednosti

# Ipiše:
5
4
```

Vračanje podatkov (return):

Funkcije lahko tudi vrnejo vrednosti, to v definicij funkcije označimo s stavkom return, ko se ta izvede, program izstopi iz funkcije oz. se vse ostalo ne izvede. Če funkcijo, ki vrne vrednost pokličemo, lahko to vrednost shranimo v spremenljivki

```
def kvadriraj(x):
    kvadrat = x ** x
    return kvadrat

kvadrirano_stevilo = kvadriraj(5)

print(kvadrirano_stevilo)
# Ipiše:
25
```

Funkcija ima lahko več `return` stavkov, izvedel pa se bo le eden. V funkcijah lahko definiramo tudi nove spremenljivke vendar bodo te obstajale le znotraj funkcije.

```
def pomnozi_z_pet(x):
    pet = 5
    return x * pet  # Lahko vrnemo tudi operacijo, ta na koncu vrne izračunano število.

print(pomnozi_z_pet(4))
# Ipiše:
20
```

Lambda funkcije in naivna igra Križec Krožec

Lambda funkcije (lambdas) in rešitev naloge ter Križec Krožec

Podobno kot navadne funkcije, lahko še enovrstične funkcije, ki jim pravimo lambda funkcije. Te uporabljamo, kadar potrebujemo enostavnejše funkcije, ki jih lahko zapišemo v eni vrstici. Te funkcije definiramo brez besede `def` tako, da uporabimo izraz `lambda`. Njihovo funkcionalnost shranimo v spremenljivko.

Pišemo: `ime_sprem = lambda parameter: operacija, ki jo funkcija izvede ter vrne podatek`

Če si pogledamo na primerih:

```
kvadrat = lambda x: x**2  # Funkcija prejme število in vrne njegov kvadrat

print(kvadrat(2))

# Izpiše:
4

# Rezultat lahko shranimo v novo spremenljivko:
rez = kvadrat(3)

print(rez)

# Izpiše:
9
```

Rešitev naloge izrisovanja kvadrata

Rešitev naloge izrisovanja kvadrata iz znakov, poljubne dolžine stranic in poljubne širine stranic.

Navodila:

Napiši funkcijo ki sprejme dve števili n in k in izpiše kvadrat znakov "*" velikosti n x n, debeline(stranic) k. V primeru, da je debelina stranic večja od dolžine stranice, naj je kvadrat poln, oz. če je 2k >= n.

Rešitev:

```
def izrisi_kvadrat(n, k):
    """
    Funkcija izrise kvadrat iz znakov *, z stranicami velikosti n
    debeline k
    """
    zgoraj = k
    spodaj = n - k
    for i in range(n):
        if i >= zgoraj and i < spodaj:
            print("*" * k + " " * (n - 2 * k) + "*" * k)
        else:
            print("*" * n)
```

Enako lahko rešimo z uporabo "list comprehension", kjer seznam definiramo kar s for zanko, na koncu pa seznam zlepimo skupaj.

```
def izrisi_kvadrat_oneliner(n, k):
    """
    Funkcija izrise kvadrat iz znakov *, z stranicami velikosti n
    debeline k
    """
    print("".join(["*"*k+" "*(n-2*k)+"*"*k+"\n" if (i>=k and i<n-k) else "*"*n+"\n" for i in range(n)]))
```

Metoda `join(seznam)` elemente v seznamu zlepi skupaj v niz tako, da med posameznimi znaki nastopi prazen niz `""` ,na kateremu kličemo metodo.

Zgornje lahko z uporabo lambda funkcije zapišemo dobesedno v eni vrstici:

```
izrisi_kvadrat_dobesedno_oneliner = lambda n, k: "".join(["*"*k+" "*(n-2*k)+"*"*k+"\n" if (i>=k and i<n-k) else "*"*n+"\n" for i
```

Pri obeh enovrstičnih rešitvah uporabljamo znak `"\n"` kar predstavlja prelom v novo vrstico.

Križec krožec

Med predavanji smo z naučenim znanjem ustvarili igro Križec Krožec.

```
# Igra krizec krozec

polje = [[" ", " ", " "], [" ", " ", " "], [" ", " ", " "]]
# Polje:
#          1.st  2.st  3.st
#  1. vr   [" ", " ", " "]
#  2. vr   [" ", " ", " "]
#  3. vr   [" ", " ", " "]
zmagovalni_znak = ""
konec_igre = False

krog = 1
while krog < 9:
    # Izpišemo polje
    print("{}|{}|{}".format(polje[0][0], polje[0][1], polje[0][2]))
    print("-----")
    print("{}|{}|{}".format(polje[1][0], polje[1][1], polje[1][2]))
    print("-----")
    print("{}|{}|{}".format(polje[2][0], polje[2][1], polje[2][2]))

    # Preberemo ukaz igralca
    if krog % 2 == 0:
        ukaz = input("Na vrsti je 2. igralec: ")
        znak = "X"
    else:
        ukaz = input("Na vrsti je 1. igralec: ")
        znak = "O"

    # Iz ukaza dolocimo vrstico in stolpec
    vrstica = int(ukaz[0]) - 1
    stolpec = int(ukaz[1]) - 1
    # Spremenimo polje na doloceni vrsticici in stolpcu
    polje[vrstica][stolpec] = znak

    # Pregledamo ce je kdo dosegel 3 v vrsti
    # Ustvarimo seznam stolpcev, da je bolj pregledno
    stolpci = [[], [], []]
    # Gremo po vrsticah
    for vrstica in polje:
        # Za vsako vrstico preverimo ce vsebuje 3 enake znake
        # Tako da jo pretvorimo v mnozico
        if len(set(vrstica)) == 1 and vrstica[0] != " ":
            zmagovalni_znak = vrstica[0]
            konec_igre = True
        # Sproti ustvarjamo transponirano polje oz. seznam stolpcev
        # Posameznemu stolpcu dodamo znak iz polja
        stolpci[0].append(vrstica[0])
        stolpci[1].append(vrstica[1])
        stolpci[2].append(vrstica[2])
```

```
    # Preverimo še stolpce
    for stolpec in stolpci:
        # Za vsak stolpec preverimo ce vsebuje 3 enake znake
        # Tako da ga pretvorimo v mnozico
        if len(set(stolpec)) == 1 and stolpec[0] != " ":
            zmagovalni_znak = stolpec[0]
            konec_igre = True
    # Na dolgo preverimo še dve diagonali
    znak = polje[0][0]  # Znak v levem zgornjem kotu
    if polje[1][1] == znak and polje[2][2] == znak and znak != " ":
        zmagovalni_znak = znak
        konec_igre = True
    znak = polje[0][2]  # Znak v desnem zgornjem kotu
    if polje[1][1] == znak and polje[2][0] == znak and znak != " ":
        zmagovalni_znak = znak
        konec_igre = True

    if konec_igre:
        break

    krog += 1


# Ko pridemo ven iz zanke se enkrat izpisemo polje in dolocimo zmagovalca
print("{}|{}|{}".format(polje[0][0], polje[0][1], polje[0][2]))
print("-----")
print("{}|{}|{}".format(polje[1][0], polje[1][1], polje[1][2]))
print("-----")
print("{}|{}|{}".format(polje[2][0], polje[2][1], polje[2][2]))

if zmagovalni_znak == "O":
    print("Zmagal je 1. igralec!")
elif zmagovalni_znak == "X":
    print("Zmagal je 2. igralec!")
else:
    print("Igra je bila izenacena!")
```

Objektno programiranje: Razredi, objekti in dedovanje

Razredi (classes)

Razred si najlažje predstavljamo kot nek načrt za določene objekte. Razredi vsebujejo atribute(podatki shranjeni v spremenljivkah razreda) in metode(funkcije, ki izvajajo operacije nad samim objektom)

Razred definiramo z besedo `class`, nakar podamo ime razreda kot spremenljivko ter dvopičje; vse kar spada v razred indentiramo. V Pythonu imena razredov pišemo tako, da je prva črka posamezne besede napisana z veliko, posamezne besede pa ne ločujemo s podčrtaji. Temu pravilu pravimo CamelCase.

Če sedaj za primer razreda definiramo razred Oseba in mu podamo nekaj atributov, ki jih pišemo enako kot spremenljivke.

```
class Oseba:
    ime = "Liam"
    priimek = "Mislej"
```

Če bi želeli do podatkov dostopati moremo razred najprej inicializirati. To storimo tako, da s spremenljivko enačimo ime razreda z oklepaji. Kadar inicializiramo razred, pravimo temu objekt.

```
o1 = Oseba()
```

Tako je v sapremenljivki o1 shranjen objekt razreda Oseba.

Če želimo do posameznih atributov dostopati, zraven spremenljivke objekta napišemo ime atributa ter povežemo s piko.

```
ime_o1 = o1.ime
priimek_o1 = o1.priimek
```

```
print(ime_o1, priimek_o1)

# Izpiše:
Liam Mislej
```

Zgornji primer ni zelo praktičen, saj vsakič ko naredimo objekt tipa Oseba, ima ta pri atributih ime in priimek vrednosti "Liam" in "Mislej".

Za tem nastopi konstruktor **`init`** . Tega v razredu definiramo podobno kot funkcije. Pomembno je vedeti, da se izvede kadar objekt inicializiramo. Funkcijam definiranim znotraj razredov pravimo metode(o tem kasneje). Konstruktorju **`init`** lahko podajamo parametre (tukaj nastopi nekoliko nenavadna sintaksa).

Torej če bi zgornji primer razširili, da lahko razredu podamo določene vrednosti:

```
class Oseba:
    def __init__(self, ime, priimek):
        self.ime = ime
        self.priimek = priimek
```

Opazimo novo spremenljivko self. Ta se nanaša na sam objekt znotraj razreda. Ko želimo poizvedovati po določenih atributih pred imenom pišemo še self. Sicer bi lahko namesto self pisali karkoli vendar je taka navada in je prav, da se je držimo.

Prvi parameter v konstruktorju **`init`** je namenjen imenu spremenljivke, ki se nanaša na sam objekt, v našem primeru na self.

Če inicializiramo en objekt tega tipa in želimo dostopati, do kakšnega atributa to storimo enako kot v prejšnjem primeru.

```
o1 = Oseba("Janez", "Novak")  # prarameter ime = "Janez", priimek = "Novak"

print(o1.ime)
print(o1.priimek)

# Izpiše:
Janez
Novak
```

Parameter, ki je na mestu `self` v definiciji razreda oz. konstruktorju **`init`** , ne podajamo kadar inicializiramo objekt.

Metode:

Metode so funkcije definirane v samem razredu. Pišemo jih enako kot funkcije. Če jim podamo parameter self lahko metoda manipulira z atributi razreda. V kolikor tega ne podamo metoda funkcionira kot navadna funkcija, le da jo kličemo malenkost drugače.

Zgornjemu razredu dodajmo atribut starost in metodo `rojstni_dan` , ki objektu poveča vrednost spremenljivke starost za 1.

```
class Oseba:
    def __init__(self, ime, priimek, starost):
        self.ime = ime
        self.priimek = priimek
        self.starost = starost

    def rojstni_dan(self):  # Sprejme parameter self, saj bo metoda spreminjala atribute
        """
        Metoda postara osebo za 1 leto
        """
        self.starost += 1
```

Če želimo metodo uporabiti oz. klicati na že obstoječem objektu jo napišemo zraven spremenljivke objekta, ločimo s piko in dodamo oklepaje, podobno kot pri atributih.

Primer:

```
o1 = Oseba("Janez", "Novak", 42)

print(o1.starost)

o1.rojstni_dan()  # Postaramo za 1 leto
```

```
print(o1.starost)

# Izpiše:
42
43
```

Metode se obnašajo enako kot funkcije s tem, da lahko dodatno operirajo na objektu. Torej z metodami lahko vračamo vrednosti in jim podajamo parametre.

Dedovanje (inheritence)

Pri razredih lahko uporabljamo dedovanje. Kot namiguje ime, lahko določen razred deduje po nekem drugem razredu. Kadar razred deduje, ta prevzame vse metode in atribute, ki jih razred po katerem dedujemo ima.

Dedovanje pišemo v definiciji razreda, in sicer v oklepajih. Poleg imena, napišemo ime razreda po kateremu dedujemo.

Recimo, da bi radi naredili razred, ki deduje po razredu oseba oz. tisti razred razširi. Novi razred bomo poimenovali delavec in imel bo dodatne atribute in metode.

Pri tem moramo biti pozorni, saj razredu Oseba podajamo parametre. Če bi želeli ustvariti objekt Delavec, ki privzame določene atribute iz razreda Oseba moramo te parametre tudi novem razredu podati. To naredimo s stavkom `super().__init__(ime, priimek, starost)` znotraj konstruktorja `__init__`, in pri parametrih init podamo še parametre razreda Oseba.

```
class Delavec(Oseba):
    def __init__(self, delovna_leta, ime, priimek, starost):
        super().__init__(ime, priimek, starost)
        self.delovna_leta = delovna_leta  # Atribut predstavlja število let ko je oseba delala, to bo torej celo število int
        self.zaposlen = False  # Novi atribut, ki pove če je oseba trenutno zaposlena, to bo binarna vrednost bool

    def zaposli(self):
        """
        Metoda nastavi atribut zaposlen na True
        """
        self.zaposlen = True

    def odpusti(self):
        """
        Metoda nastavi atribut zaposlen na False
        """
        self.zaposlen = False

    def opisi(self):
        """
        Metoda izpiše niz s katerim opiše objekt oz. osebo, ki je hkrati delavec
        """
        if self.zaposlen:
            zap = "je"
        else:
            zap = "ni"
        print("Delavec {} {} {} zaposlen in je v življenju delal {} let.".format(self.ime, self.priimek, zap,
self.delovna_leta))
```

Če sedaj ustvarimo objekt in kličemo metode definirane v razredu Delavec in tiste def. v razredu Oseba:

```
o = Delavec(ime="Janez", priimek="Novak", starost=21, delovna_leta=2)

o.opisi()
o.zaposli()
o.opisi()
print(o.starost)
o.rojstni_dan()
print(o.starost)

# Izpiše:

Delavec Janez Novak ni zaposlen in je v življenju delal 2 let.
Delavec Janez Novak je zaposlen in je v življenju delal 2 let.
21
22
```

Knjižnice, branje podatkov iz datotek, csv in matplotlib

PIP

PIP je Pythonov paketni menedžer (package manager), vsebuje vse objavljene pakete, ki si jih lahko poberemo.

Paketi so skupki modulov, kjer ima vsak modul neko uporabnost. Če želimo analizirati podatke lahko(med drugim) poberemo paket Pandas, če nas zanima ustvarjanje video iger za to obstaja PyGame, za manipulacijo slik obstaja paket PIL ali pa OpenCV... Na kratko nam paketi podajajo neke funcionalnosti, ki nam delo olajšajo. Če bi želeli izrisati histogram ne bi to na novo izumljali ampak bi samo pobrali knjižnjico oz. paket matplotlib.

Pobiranje paketov in knjižnjic

Od Pythona 3.4. dalje je PIP vsebovan v samem Pythonu, ko ga prenesemo.

Ukaze, ki jih omogoča PIP lahko preverimo tako, da v ukazno vrstico sistema(cmd na Windows sistemu) vpišemo pip.

Primer:

```
C:\Users\Liam>pip
```

Če želimo določeno knjižnjico pobrati napišemo:

```
C:\Users\Liam>pip install matplotlib
```

Kjer podamo pravo ime knjižnjice. Tako se nam knjižnjica naloži na računalnik oz. tam kjer se Python nahaja.

Moduli in uvažanje (modules and imports)

Ko smo določen paket/modul/knjižnjico pobrali z ukazom pip, lahko v kodi uporabljamo vse funkcije/razrede/karkoli, ki jih knjižnjica ponuja. Če želimo v naši .py datoteki uporabiti knjižnjico moramo to Pythonu povedati z ukazom `import` ,kjer zraven podamo ime knjižnjice.

Primer:

```
import csv
```

Če imamo v isti mapi več datotek .py v katerih se nahajajo funkcije lahko v drugih datotekah te funkcije uporabljamo tako, da jih najprej uvozimo z zgornjim ukazom.

Branje in pisanje v datoteke

Med predavanji bomo uporabljali podatke iz SiStat, konkretno:

Bruto domači proizvod na prebivalca, Slovenia, letno

https://pxweb.stat.si/SiStatData/pxweb/sl/Data/-/H280S.px Pobrana datoteka tipa csv(ločeno z vejico in vsebuje glavo), v samem programu preimenovana v bdp_na_preb.csv

Prebivalstvo po izbranih starostnih skupinah in spolu, statistične regije, Slovenija, polletno

https://pxweb.stat.si/SiStatData/pxweb/sl/Data/-/05C2006S.px Poberemo vse podatke za 1. polletje leta 2020. Pobrana datoteka tipa csv(ločeno z vejico in vsebuje glavo) v samem programu preimenovana v preb_po_star_in_spol.csv

Datoteke lahko odpremo na različne načine(preden lahko iz datoteke beremo mora biti odprta), najbolj pogosta metoda je s stavkom `with open("pot/do/datoteke/test.txt", "r/w/a") as ime_spremenljivke:` . Če se datoteka nahaja v isti mapi kot naš program, lahko navedemo le ime datoteke.

Če želimo odpreti datoteko `test.txt` :

```
with open("test.txt", 'r') as file:
    # Tukaj lahko potem upravljamo z datoteko
```

Pri drugem parameteru pri `open<c/ode> navedemo kaj bomo z datoteko počeli:`

- 'r' read, le brali podatke
- 'w' write, pisali v datoteko nove podatke oz. vrstice
- 'a' append, dodajali na konec datoteke nove podatke

Če želimo prebrati eno vrstico lahko uporabimo metodo `readline` :

```
with open("test.txt", 'r') as file:
    # Tukaj lahko potem upravljamo z datoteko
    line = file.readline()
    print(line)

Izpiše niz prve vrstice v datoteki
```

Če želimo prebrati vse vrstice, uporabimo metodo `readline` :

```
with open("test.txt", 'r') as file:
    # Tukaj lahko potem upravljamo z datoteko
    lines = file.readlines()
    print(line)

Izpiše seznam, kjer je vsaka vrstica en element
```

Ker bomo podatke brali iz datotek tipa csv, moramo najprej uvoziti integrirano knjižnico `csv` .

```
import csv
```

Ko beremo datoteke tipa csv, nam ta knjižnjica olajša branje tako, da ignroira določene nevidne znake ...

Podatke iz odprte datoteke nato preberemo z ukazom:

```
with open("bdp_na_preb.csv", "r") as f:
    data = csv.reader(f)
    for row in data:
        print(row)

# Izpiše vse vrstice v datoteki, kjer je vsaka vrstica seznam podatkov.
```

`csv.reader()` vrne seznam vseh vrstic, kjer vsak element predstavlja podatek na določenem stolpcu v datoteki.

Matplotlib

Matplotlib je Python knjižnjica za izrisovanje podatkov, od grafov do histogramov in še veliko več. Uporabljali bomo modul pyplot. Po tem, ko smo knjižnjico pobrali z uporabo PIP, lahko modul v programu uporabimo tako, da ga uvozimo in shranimo v spremenljivko `plt` z ukazom `as` :

```
from matplotlib import pyplot as plt
```

Risanje navadnega grafa

Za izrisovanje grafa uporabimo funkcijo `plot(x, y)` , kjer je x seznam x vrednosti točk, y pa seznam y vrednosti točk.

Funkcijo kličemo tako, da ime pišemo zraven spremenljivke `plt` .

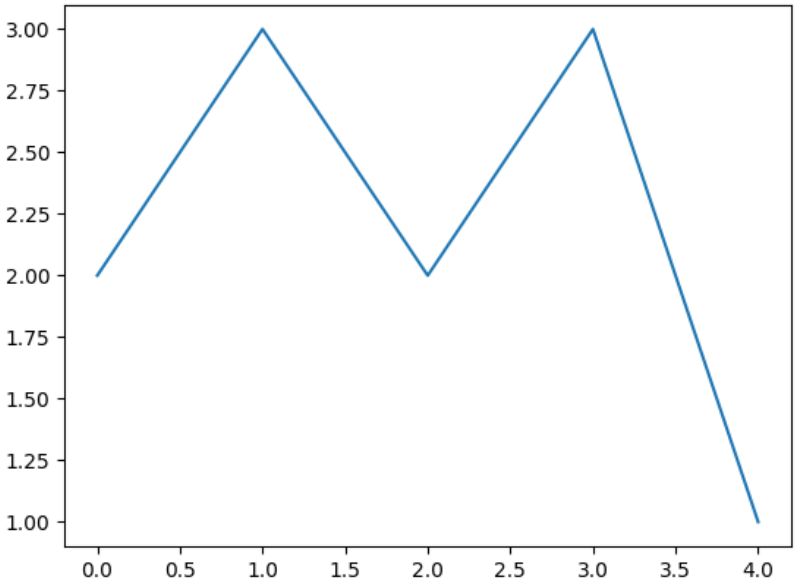
Primer:

```
x = [0, 1, 2, 3, 4]
y = [2, 3, 2, 3, 1]

plt.plot(x, y)
```

```
plt.show()  # Če želmo, da se graf prikaže kličemo plt.show()
```

Zgornje nam izriše sledeči graf:



Risanje histograma

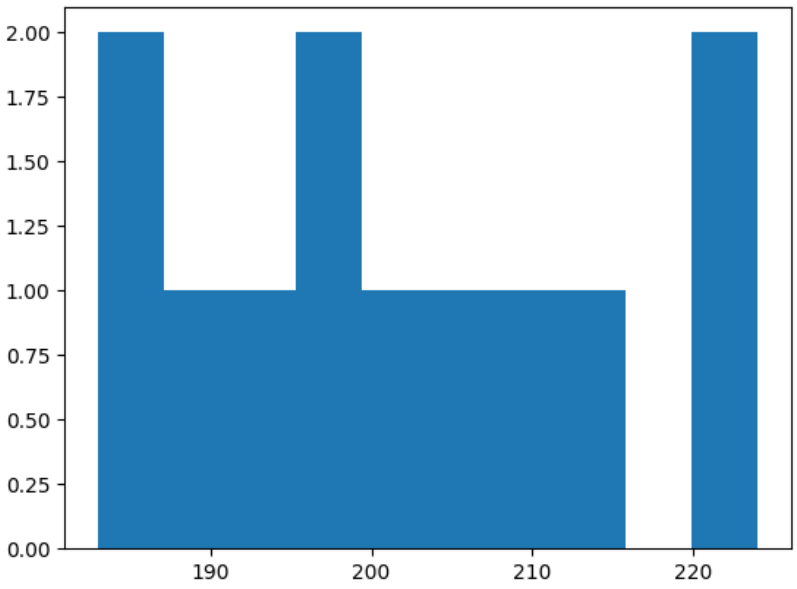
Za izrisovanje histograma uporabimo funkcijo `hist(podatki, bins=10)` . Prvi parameter funkcije je seznam vseh podatkov, bins pa določa število stolpcev iste širine, nastavljeno na 10. Namesto števila za bins lahko podamo seznam pozicij stolpcev.

Primer na podatkih višine košarkarjev Dallas Mavericks.

```
visine_igralcev =  [183, 185, 188, 193, 196, 198, 201, 206, 208, 213, 221, 224]  # V cm

plt.hist(visine_igralcev)
plt.show()
```

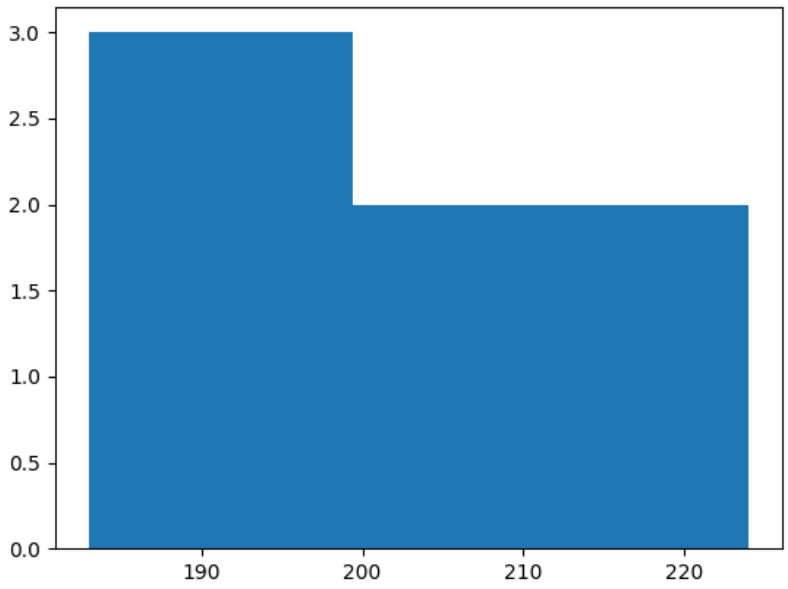
Izriše naslednje:



Če spremenimo število stolpcev na 5:

```
visine_igralcev =  [183, 185, 188, 193, 196, 198, 201, 206, 208, 213, 221, 224]  # V cm

plt.hist(visine_igralcev, bins=5)
plt.show()
```



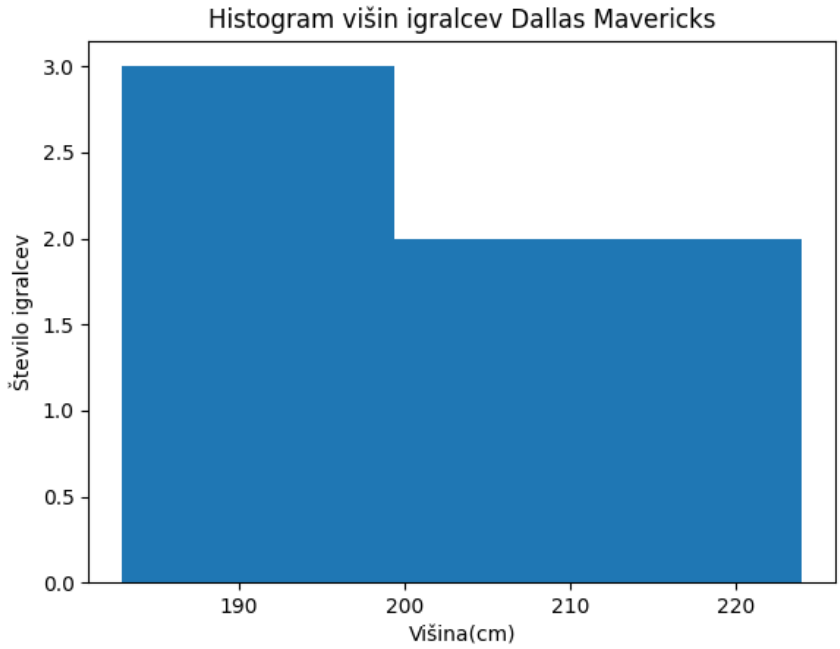
Imenovanje osi in naslov

Osi na grafu lahko poimenujemo z uporabo `xlabel` in `ylabel`. Naslov določimo z `title`.

```
visine_igralcev =  [183, 185, 188, 193, 196, 198, 201, 206, 208, 213, 221, 224]  # V cm

plt.hist(visine_igralcev, bins=5)
plt.xlabel("Višina(cm)")
plt.ylabel("Število igralcev")
plt.title("Histogram višin igralcev Dallas Mavericks")
plt.show()
```

Izriše:



Risanje blokovnega grafa

Pri histogramu podamo vse merjene vrednosti, če imamo vrednosti že preštete in urejene uporabimo `bar`. Funkcija parameter prejme x pozicije stolpcev kot seznam, višine stolpcev določimo tako, da parametru height podamo seznam vrednosti.

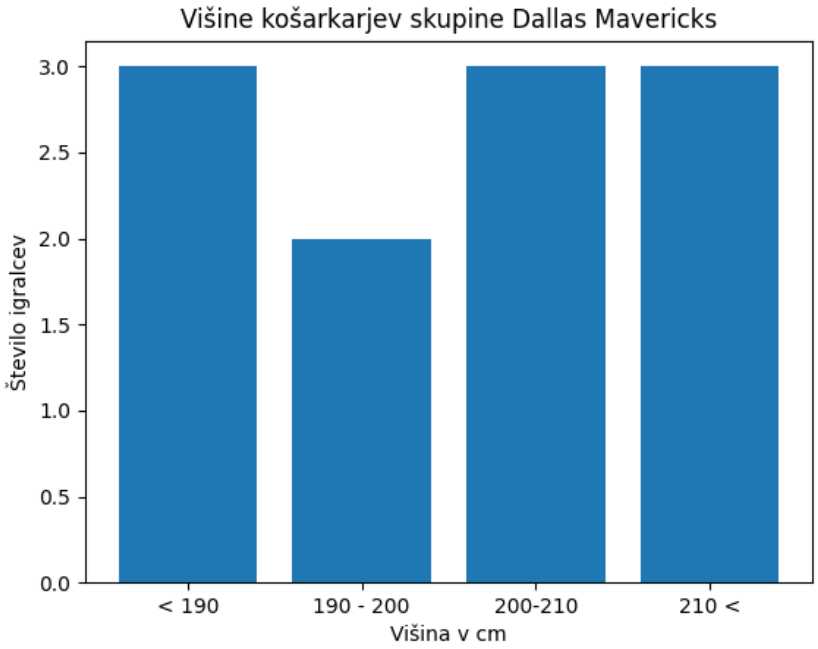
Primer: Če uzamemo zgornje podatke višin košarkašev le, da so sedaj prešteti.

```
# Višine = [183, 185, 188, 193, 196, 198, 201, 206, 208, 213, 221, 224]

bars_x = ["< 190", "190 - 200", "200-210", "210 <"]  # Opisi stolpcev
st_po_visini = [3, 2, 3, 3] # 3 igralci so nižji od 190cm, 2 med 190cm in 200cm, ...

plt.bar(bars_x, height=st_po_visini)
# Nastavimo opise in naslov
plt.xlabel("Višina v cm")
plt.ylabel("Število igralcev")
plt.title("Višine košarkašev skupine Dallas Mavericks")
# Izrišemo
plt.show()
```

Izriše:



Liam Mislej, 25.5.2021