FLOW CONTROL STATEMENTS

Omogočajo nam kontrolo sprememb in logike programa.

If statement

```
if <expr>:
    <statement>
    <statement>
    ...
    <statement>
<following_statement>
```

```
<expr> je izraz ovrednoten v Boolean kontekstu.
<statement> je Python izraz (nadaljevanje naše kode), ki je pravilno zamaknjen.
Če je <expr> True, potem se izvedejo <statement>. Če je <expr> False, potem se  <s
tatement> preskoči in se ne izvede.
Nato se program nadaljuje z <following_statement>
```

Indentation / Zamikanje

> Pri Pythonu se zamikanje (indentation) uporablja za definiranje blokov kode. Vse vrstice z istim zamikom se smatrajo kot isti blok kode.

> Bloke kode se lahko poljubno globoko "nesta".

> Zamikanje je določeno z tabulatorjem ali presledki. Ni važno točno število, važno je, da je skozi kodo enako.

In [5]:

```
x = 0
y = 5

if x < y:
    print("Smo znotraj if.")
    print("End if")
print("End")
```

```
Smo znotraj if.
End if
End
```

Else

Včasih želimo, da če je nekaj res se izvede določen blok kode, če stvar ni res pa naj se izvede drug del kode.

To dosežemo z else.

```
if <expr>:
    <statement(s)>
else:
    <statement(s)>
<following_statement>
```

Če je <expr> True se izvede blok direktno pod njem, če pa je <expr> False se ta bl
ok kode preskoči in se izvede blok pod else.

In [4]:

```
x = 100

if x < 50:
    print('(first block)')
    print('x is small')
else:
    print('(second block)')
    print('x is large')

print("End")
```

```
(second block)
x is large
End
```

Elif

Če želimo še večjo razvejanost naših možnosti lahko uporabimo elif (else if).

```
if <expr>:
    <statement(s)>
elif <expr>:
    <statement(s)>
elif <expr>:
    <statement(s)>
else:
    <statement(s)>
<following_statement>
```

Python preveri vsak <expr> posebej. Pri ta prvem, ki bo True, bo izvedel njegov bl
ok kode.
Če ni nobeden True se bo izvedel else blok kode.

In [5]:

```
x = 20
if x > 100:
    print('x je večje od 100')
elif x > 50:
    print('x večje od 50 in manjše od 100')
elif x > 30:
    print('x večje od 30 in manjše od 50')
elif x > 10:
    print('x večje od 10 in manjše od 30')
else:
    print("x manjše od 10")

print("End")
```

```
x večje od 10 in manjše od 30
End
```

One-line if statement

Obstaja način zapisa if stavka v eni vrstici ampak se ta način odsvetuje, ker napravi kodo nepregledno.

https://realpython.com/python-conditional-statements/ (https://realpython.com/python-conditional-statements/)

```
<expr1> if <conditional_expr> else <expr2>

z = 1 + x if x > y else y + 2

If <conditional_expr> is true, <expr1> is returned and <expr2> is not evaluated.
If <conditional_expr> is false, <expr2> is returned and <expr1> is not evaluated.
```

In [6]:

```
x = 8
z = 1 + x if x > 10 else x**2
print(z)

x = 20
z = 1 + x if x > 10 else x**2
print(z)
```

```
64
21
```

The pass statements

Uporablja se kot "placeholder", da nam interpreter ne meče napak.

In [11]:

```
if True:

print("Hello") # should give IndentationError
```

```
  File "<ipython-input-11-33a91c099307>", line 3
    print("Hello") # should give IndentationError
        ^
IndentationError: expected an indented block
```

In [12]:

```
if True:
    pass
print("Hello") # should be fine now with the pass added
```

```
Hello
```

Vaja 01

Napišite program, ki bo uporabnika uprašal naj vnese neko celoštevilsko vrednost. Program naj nato izpiše ali je vrednost deljiva z 3 ali ne.

In [ ]:

```

```

In [ ]:

```

```

Vaja 02

Napišite program, ki bo pretvoril stopinje Celzija v Fahrenheit ali obratno.

Uporabnik naj vnese številko. Nato naj vnese v katerih enotah nam je podal vrednost (**C** ali **F**). Glede na vnešeno črko naj vaš program uporabi pravilno formulo za pretvorbo.

```
T(°F) = T(°C) × 9/5 + 32

T(°C) = (T(°F) - 32) x 5/9
```

Če uporabnik ni vnesel **C** ali **F** naj program izpiše *Prišlo je do napake.*

Primer:

```
Vnesi vrednost: 12
Vnesi enoto: C
```

Rešitev:

```
12 stopinj celzija je enako 53.6 fahrenheit.
```

In [ ]:

```
```

In [ ]:

```
```

In [ ]:

```
```

While

While zanka deluje na podoben princip kot if. While izvaja blok kode, dokler je "expression" True.

```
    while <expr>:
        <statement(s)>
```

In [26]:

```
lepo_vreme = True
while lepo_vreme:
  print('Vreme je lepo.')
  lepo_vreme = False
```

```
Vreme je lepo.
```

In [14]:

```
#the body should be able to change the condition’s value, because if the condition is
#True at the beginning, the body might run continuously to infinity
#while True:
#    print("Neskončna zanka. Se ne ustavim.")

#ustavimo v CTRL + C
```

While zanko se lahko uporabi za ponovitev bloka kode določenega števila korakov.

In [27]:

```
i = 0
while i < 10:
    print(f'Repeated {i} times')
    i += 1
```

```
Repeated 0 times
Repeated 1 times
Repeated 2 times
Repeated 3 times
Repeated 4 times
Repeated 5 times
Repeated 6 times
Repeated 7 times
Repeated 8 times
Repeated 9 times
```

In [16]:

```
#A common use of the while loop is to do things like these:

temperature = 15

while temperature < 20:
    print('Heating...')
    temperature += 1

#Only instead of the temperature increasing continuously, we would e.g. get it from a senso
#Remember to always have a way of exiting the loop! Otherwise it will run endlessly!
```

```
Heating...
Heating...
Heating...
Heating...
Heating...
```

Obstaja tud while else.

```
while <expr>:
    <statement(s)>
else:
    <additional_statement(s)>

The <additional_statement(s)> specified in the else clause will be executed when t
he while loop terminates.
About now, you may be thinking, “How is that useful?” You could accomplish the sam
e thing by putting those statements immediately after the while loop, without the
else:

What’s the difference?

In the latter case, without the else clause, <additional_statement(s)> will be exe
cuted after the while loop terminates, no matter what.

When <additional_statement(s)> are placed in an else clause, they will be executed
only if the loop terminates “by exhaustion”—that is, if the loop iterates until th
e controlling condition becomes false. If the loop is exited by a break statement,
the else clause won’t be executed.
```

Vaja 01

Napišite program, ki izpiše prvih 10 sodih števil.

In [ ]:

In [ ]:

Vaja 02

Uporabnik naj vnese željeno dolžino Fibonaccijevega zaporedja. Program naj nato to zaporedje shrani v list in ga na koncu izpiše.

```
Fibonacci sequence

0, 1, 1, 2, 3, 5, 8, 13, 21, 34
```

In [ ]:

```
```

In [ ]:

```
```

In [ ]:

```
```

For loop

Uporablja se kadar hočemo izvesti blok kode za vnaprej določeno število ponovitev.

Primer: kadar hočemo izvesti blok kode za vsak element v list-u.

```
for <var> in <iterable>:
    <statement(s)>
```

In [17]:

```
primes = [2, 3, 5, 7, 11] #itrable
for prime in primes:
    print(f'{prime} is  a prime number.')
```

```
2 is  a prime number.
3 is  a prime number.
5 is  a prime number.
7 is  a prime number.
11 is  a prime number.
```

In [18]:

```
kid_ages = (3, 7, 12)
for age in kid_ages:
    print(f'I have a {age} year old kid.')
```

```
I have a 3 year old kid.
I have a 7 year old kid.
I have a 12 year old kid.
```

Velikokrat se skupaj z for-loop uporablja funkcija range().

```
range(start, stop, step)
```

- start - Optional. An integer number specifying at which position to start. Default is 0
- stop - An integer number specifying at which position to end, excluding this number.
- step - Optional. An integer number specifying the incrementation. Default is 1

Funkcija range nam zgenerira list števil.

In [24]:

```
x = range(-5, 10, 1)
print(type(x))
print(list(x))
```

```
<class 'range'>
[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

In [41]:

```
# Primer: Iteracija čez dictionary
pets = {
    'macka': 6,
    'pes': 12,
    'krava': 20
}

for pet, years in pets.items():
    print(f'{pet} je star/a {years} let.')
```

```
macka je star/a 6 let.
pes je star/a 12 let.
krava je star/a 20 let.
```

> **Nasveti**

- Use the enumerate function in loops instead of creating an “index” variable

Programmers coming from other languages are used to explicitly declaring a variable to track the index of a container in a loop. For example, in C++:

```
for (int i=0; i < container.size(); ++i)
{
// Do stuff
}
```

In Python, the enumerate built-in function handles this role.

In [7]:

```
moj_list = ["Anže", "Luka", "Mojca"]
index = 0
for element in moj_list:
    print (f'{index} {element}')
    index += 1
```

```
0 Anže
1 Luka
2 Mojca
```

In [6]:

```
#Idiomatic
moj_list = ["Anže", "Luka", "Mojca"]
for index, element in enumerate(moj_list):
    print (f'{index} {element}')
```

```
0 Anže
1 Luka
2 Mojca
```

Break

Break keyword terminira najbolj notranjo zanko v kateri se nahaja.

In [44]:

```
avti = ["ok", "ok", "ok", "slab", "ok"]

for avto in avti:
    if avto == "slab":
        print("Avto je zanič.")
        break
    print("Avto je ok.")
    print("Naslednji korak zanke")
print("End")
```

```
Avto je ok.
Naslednji korak zanke
Avto je ok.
Naslednji korak zanke
Avto je ok.
Naslednji korak zanke
Avto je zanič.
End
```

Continue

Continue keyword izpusti kodo, ki se more še izvesti, in skoči na naslednjo iteracijo zanke .

In [45]:

```
avti = ["ok", "ok", "ok", "slab", "ok"]

for avto in avti:
    if avto == "slab":
        print("Avto je zanič.")
        continue #continue #lah pokažeš še primer k je stvar zakomentirana
    print("Avto je ok.")
    print("Naslednji korak zanke")
print("End")
```

```
Avto je ok.
Naslednji korak zanke
Avto je ok.
Naslednji korak zanke
Avto je ok.
Naslednji korak zanke
Avto je zanič.
Avto je ok.
Naslednji korak zanke
End
```

Vaja 01

Iz danega dictionary izpišite vse ključe, katerih vrednost vsebuje črko **r**.

```
d = {
    "mačka": "Micka",
    "pes": "Fido",
    "volk": "Rex",
    "medved": "Žan",
    "slon": "Jan",
    "žirafa": "Helga",
    "lev": "Gašper",
    "tiger": "Anže",
    "papagaj": "Črt",
    "ribica": "Elena",
    "krokodil": "Kasper",
    "zajec": "Lars",
    "kamela": "Manca"
}
```

In [ ]:

In [ ]:

Vaja 02

Poiščite vsa praštevila med 2 in 30.

In [ ]:

```
```

In [ ]:

```
```

Funkcije

Funkcija je blok kode, ki izvede specifično operacijo in jo lahko večkrat uporabimo.

> Za primer, če v programu večkrat uporabniku rečemo, naj vnese celo število med 1 in 20. Od njega zahtevamo vnos s pomočjo **input** in nato to spremenimo v celo število z uporabo **int**. Nato preverimo ali je število v pravilnem rangu. To zaporedje kode v programu večkrat ponovimo.

> Če se sedaj odločimo, da naj uporabnik vnese celo število v rangu med 1 in 100, moramo popraviti vsako vrstico posebej, kar hitro lahko privede do napake.

> Za lažje pisanje programa lahko to zaporedje kode shranimo v funkcijo. Če sedaj spremenimo rang, le-tega popravimo samo enkrat, znotraj naše funkcije.

Funkcije nam omogočajo uporabo tuje kode brez globjega razumevanja kako le-ta deluje. Z njihovo pomočjo lahko zelo kompleksne probleme razbijemo na majhne in bolj obvladljive komponente.

Defining a Function

Funkcijo definiramo z uporabo def keyword kateri sledi ime funkcije in oklepaji (). Zaključi se jo z " : ".

Blok kode, katero želimo, da naša funkcija izvede zapišemo z ustreznim zamikom.

```
def ime_funkcije():
    # Naš blok kode katero želimo izvesti
    x = input("...")
    y = int(x) + 5
    ...
```

Po priporočilih se imena funkcije piše na snake_case način (vse male črke, med besedami podčrtaj _)

Funkcijo nato uporabimo tako, da jo pokličemo po imenu in dodamo zraven ().

```
ime_funkcije() # Klic naše funkcije
```

In [28]:

```
def hello():
    print("Hello, World!")

print("Začetek programa")
hello()
print("Nadaljevanje programa")
```

```
Začetek programa
Hello, World!
Nadaljevanje programa
```

Funkcije je v kodi potrebno ustvariti, še predno jo kličemo.

In [30]:

```
print("Začetek programa")
hello2()
print("Nadaljevanje programa")

def hello2():
    print("Hello, World!")
```

```
Začetek programa

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-30-d0c6cd4e0154> in <module>
      1 print("Začetek programa")
----> 2 hello2()
      3 print("Nadaljevanje programa")
      4
      5 def hello2():

NameError: name 'hello2' is not defined
```

In [ ]:

In [ ]:

In [ ]:

Naloga:

Napišite funkcijo, ki od uporabnika zahteva naj vnese svojo EMŠO število.

Funkcija naj nato izpiše koliko let je uporabnik star.

EMŠO ima 14 številk XXXXyyyXXXXXXX. 5.,6.,7. številka predstavljajo letnico rojstva (999 -> 1999 leto rojstva).

Primeri:

```
Input:
Vnesi emšo: 0102999500111

Output:
Star si 22 let

Input:
Vnesi emšo: 0104986505555

Output:
Star si 35 let
```

In [ ]:

```
```

In [ ]:

```
```

Working with Parameters

Funkciji lahko pošljemo določene spremenljivke, katere želimo uporabiti v funkciji.

> Primer: Če vemo ime uporabnika, ga lahko kličemo po imenu, kadar od njega zahtevamo input.

Vrednost, ki jo pošljemo v funkcijo, se reče **argument**. To funkcija sprejme kot **parameter**.

- Parameters are the name within the function definition.
- Arguments are the values passed in when the function is called.

Parametre funkcije definiramo znotraj njenih "( )".

```
def funkcija_1(x, y, z): # x, y, z are parameters
    pass

funkcija_1(1, 2, 3) # 1, 2, 3 are arguments
```

In [9]:

```
def funkcija_1(x, y, z):
    print(f"X vrednost: {x}")
    print(f"Y vrednost: {y}")
    print(f"Z vrednost: {z}")

funkcija_1(1,2,3)
```

```
X vrednost: 1
Y vrednost: 2
Z vrednost: 3
```

V zgornjem primeru se ob klicu funkcije:

- vrednost 1 shrani v spremenljivko x
- vrednost 2 shrani v spremenljivko y
- vrednost 3 shrani v spremenljivko z

Zato je vrstni red argumentov pomemben!

In [10]:

```
def funkcija_1(x, y, z):
    print(f"X vrednost: {x}")
    print(f"Y vrednost: {y}")
    print(f"Z vrednost: {z}")

funkcija_1(1, 2, 3)
print("Zamenjajmo vrstni red.")
funkcija_1(3, 2, 1)
```

```
X vrednost: 1
Y vrednost: 2
Z vrednost: 3
Zamenjajmo vrstni red.
X vrednost: 3
Y vrednost: 2
Z vrednost: 1
```

Pomembno je tudi, da podamo pravilno število argumentov!

Če funkcija pričakuje 3 argumente, ji moramo podatki 3 argumente. Nič več. nič manj. V nasprotnem primeru dobimo napako.

In [18]:

```
# Primer, ko podamo premalo argumentov
def funkcija_1(x, y, z):
    print(f"X vrednost: {x}")
    print(f"Y vrednost: {y}")
    print(f"Z vrednost: {z}")

funkcija_1(1, 2)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-18-e9b6b54ff80a> in <module>
      4     print(f"Z vrednost: {z}")
      5
----> 6 funkcija_1(1, 2)

TypeError: funkcija_1() missing 1 required positional argument: 'z'
```

In [19]:

```
# Primer, ko podamo preveč argumentov
def funkcija_1(x, y, z):
    print(f"X vrednost: {x}")
    print(f"Y vrednost: {y}")
    print(f"Z vrednost: {z}")

funkcija_1(1, 2, 3, 4)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-19-271e80339153> in <module>
      4     print(f"Z vrednost: {z}")
      5
----> 6 funkcija_1(1, 2, 3, 4)

TypeError: funkcija_1() takes 3 positional arguments but 4 were given
```

Naloga:

Napiši funkcijo, ki sprejme 3 argumente.

Funkcija naj izpiše kateri ima največjo vrednost in koliko je ta vrednost.

Primeri:

```
Input:
fun_01(0,-5,6)

Output:
Tretji argument je največji. Vrednost: 6

Input:
fun_01(1, 50, -50)

Output:
Drugi argument je največji. Vrednost: 50
```

In [ ]:

```

```

In [ ]:

```

```

Keyword Arguments

Naše argumente lahko poimenujemo s pravilnim imenom parametra in tako, ko naslednjič kličemo funkcijo, ne potrebujemo argumente podati v pravilnem vrstnem redu.

```
    def pozdrav(naslavljanje, ime, priimek):
        print(f"Pozdravljeni {naslavljanje} {ime} {priimek}.")

    pozdrav(priimek="Novak", naslavljanje="gospod", ime="Miha")
```

In [46]:

```
def pozdrav(naslavljanje, ime, priimek):
    print(f"Pozdravljeni {naslavljanje} {ime} {priimek}.")

pozdrav("gospod", "Miha", "Novak")
print("\nUporaba Keyword arguments\n")
pozdrav(priimek="Novak", naslavljanje="gospod", ime="Miha")
```

```
Pozdravljeni gospod Miha Novak.

Uporaba Keyword arguments

Pozdravljeni gospod Miha Novak.
```

Če podamo napačno ime, dobimo napako.

In [47]:

```
def pozdrav(naslavljanje, ime, priimek):
    print(f"Pozdravljeni {naslavljanje} {ime} {priimek}.")

pozdrav(zadnje_ime="Novak", naslavljanje="gospod", ime="Miha")
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-47-89652f3d516a> in <module>
      2     print(f"Pozdravljeni {naslavljanje} {ime} {priimek}.")
      3
----> 4 pozdrav(zadnje_ime="Novak", naslavljanje="gospod", ime="Miha")

TypeError: pozdrav() got an unexpected keyword argument 'zadnje_ime'
```

Pri klicanju funkcije lahko uporabimo oba načina podajanja argumentov. Vendar je pomemben vrstni red.

In [48]:

```
def pozdrav(naslavljanje, ime, priimek):
    print(f"Pozdravljeni {naslavljanje} {ime} {priimek}.")

pozdrav("gospod", "Miha", priimek="Novak")
```

```
Pozdravljeni gospod Miha Novak.
```

In [49]:

```
def pozdrav(naslavljanje, ime, priimek):
    print(f"Pozdravljeni {naslavljanje} {ime} {priimek}.")

pozdrav("gospod", priimek="Novak", "Miha")
```

```
  File "<ipython-input-49-d1b39220fd0c>", line 4
    pozdrav("gospod", priimek="Novak", "Miha")
                                       ^
SyntaxError: positional argument follows keyword argument
```

Default Argument Values

Za naše parametre lahko določimo default vrednost, v primeru, da ob klicu funkcije argumenta ne podamo.

```
def funkcija(x=1, y=2):
    print(x + y)

funkcija() # Funkcijo kličemo brez argumentov

Output: 3 # Privzeti vrednosti sta x=1 in y=2
```

In [56]:

```
def pozdrav(naslavljanje="gospod", ime="Miha", priimek="Novak"):
    print(f"Pozdravljeni {naslavljanje} {ime} {priimek}.")

pozdrav()

pozdrav("g.", "Andrej", "Kovač")
pozdrav(ime="Gregor")
```

```
Pozdravljeni gospod Miha Novak.
Pozdravljeni g. Andrej Kovač.
Pozdravljeni gospod Gregor Novak.
```

Potrebno je paziti, da so parametri z default vrednostjo definirani za parametri brez default vrednosti.

In [60]:

```
def funkcija(x, y, z=0):
    print(x + y + z)

funkcija(1, 2)
```

```
3
```

In [62]:

```
def funkcija(x, y=0, z):
    print(x + y + z)

funkcija(1, 2, 3)
```

```
  File "<ipython-input-62-d290ea3a79c4>", line 1
    def funkcija(x, y=0, z):
                 ^
SyntaxError: non-default argument follows default argument
```

Naloga:

Napišite funkcijo, ki izpiše prvih N največjih vrednosti v podanem listu.

Funkcija naj ima dva parametra. Prvi parameter je list, znotraj katerega bomo iskali največje vrednosti. Drugi parameter število, ki nam pove koliko prvih največjih števil naj izpišemo. Če vrednost ni podana, naj se izpiše prvih 5 največjih števil.

Primeri:

```
Input:
vaja([1,5,7,-2,3,8,2-5,12,-22])

Output:
12
8
7
5
3

Input:
vaja([1,5,7,-2,3,8,2-5,12,-22], 3)

Output:
12
8
7
```

In [ ]:

In [ ]:

*args and **kwargs

Ta dva parametra nam omogočata, da funkciji pošljemo poljubno število argumentov.

*args nam pove, da naj neznane argumente zapakira v touple imenovan args.

**kwargs nam pove, da naj neznane argumente zapakira v dictionary imenovan kwargs.

> http://book.pythontips.com/en/latest/args_and_kwargs.html (http://book.pythontips.com/en/latest/args_and_kwargs.html)

> The idiom is also useful when maintaining backwards compatibility in an API. If our function accepts arbitrary arguments, we are free to add new arguments in a new version while not breaking existing code using fewer arguments. As long as everything is properly documented, the “actual” parameters of a function are not of much consequence.

> First of all let me tell you that it is not necessary to write *args or **kwargs. Only the * (asterisk) is necessary. You could have also written *var and **vars. Writing *args and **kwargs is just a convention.

In [77]:

```
def test_args(a, b, c, *args):
    print(f"a = \t {a}")
    print(f"b = \t {b}")
    print(f"c = \t {c}")
    print(f"args = \t {args}")

test_args(1, 2, 3, 4, 5, 6, 7, 8, 9)
```

```
a =      1
b =      2
c =      3
args =   (4, 5, 6, 7, 8, 9)
```

In [75]:

```
# Primer *ARGS

def sestevalnik(*args):
    value = 0
    for ele in args:
        value += ele
    print(value)

sestevalnik(1, 2, 3)

sestevalnik(1, 2, 3, 4, 5, 6, 7, 8, 9)
```

```
6
45
```

In [ ]:

```

```

In [78]:

```
def test_kwargs(a, b, c, **kwargs):
    print(f"a = \t {a}")
    print(f"b = \t {b}")
    print(f"c = \t {c}")
    print(f"kwargs = \t {kwargs}")


test_kwargs(a=1, b=2, c=3, d=4, e=5, f=6, g=7, h=8, i=9)
```

```
a =      1
b =      2
c =      3
kwargs =         {'d': 4, 'e': 5, 'f': 6, 'g': 7, 'h': 8, 'i': 9}
```

**kwargs pridejo prav pri posodabljanju kode in ohranjanju podpore za starejše verzije kode.

Primer: ustvarimo funkcijo **moja_funkcija**, ki ima parameter *barva_grafa*. Drugi programerij uporabijo mojo funkcijo.

Kasneje se odločim posodobiti mojo funkcijo tako, da spremenim ime parametra v *barva*. Sedaj bi morali vsi drugi programerij, ki so uporabili mojo funkcijo prav tako posodobiti njihovo kodo. Z uporabo **kwargs pa lahko še vedno zajamemo njihove argumente.

In [86]:

```
def moja_funkcija(podatki, barva_grafa="črna"):
    print(f"Barva grafa je {barva_grafa}.")

moja_funkcija([1,2,3], barva_grafa="rdeča")
```

```
Barva grafa je rdeča.
```

In [87]:

```
# Želi se posodobit to funcijo
def moja_funkcija(podatki, barva="črna"):
    print(f"Barva grafa je {barva_grafa}.")

moja_funkcija([1,2,3], barva_grafa="rdeča")
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-87-8300209fd37d> in <module>
      3     print(f"Barva grafa je {barva_grafa}.")
      4
----> 5 moja_funkcija([1,2,3], barva_grafa="rdeča")

TypeError: moja_funkcija() got an unexpected keyword argument 'barva_grafa'
```

In [89]:

```
# Želi se posodobit to funcijo
def moja_funkcija(podatki, barva="črna", **kwargs):
    if "barva_grafa" in kwargs.keys():
        print(f"Barva grafa je {kwargs['barva_grafa']}.")
    else:
        print(f"Barva grafa je {barva}.")

moja_funkcija([1,2,3], barva_grafa="rdeča")
```

```
Barva grafa je rdeča.
```

In [ ]:

```

```

In [ ]:

```

```

Returning a Value

Vsaka funkcija tudi vrne določeno vrednost.

Če funkciji nismo eksplicitno določili katero vrednost naj vrne, vrne vrednost **None**.

In [90]:

```
def funkcija():
    print("Pozdrav")

x = funkcija()
print(x)
```

```
Pozdrav
None
```

Da vrnemo specifično vrednost uporabimo besedo **return**.

```
def sestevalnik(x, y):
    vsota = x + y
    return vsota

x = sestevalnik(1, 2)
print(x)

Output: 3
```

In [93]:

```
def sestevalnik(x, y):
    print("Seštevam...")
    vsota = x + y
    return vsota

x = sestevalnik(1, 2)
print(x)
```

```
Seštevam...
3
```

Ko se izvede ukaz **return** se vrne vrednost in koda znotraj funkcije se neha izvajati.

In [94]:

```
def sestevalnik(x, y):
    print("Seštevam...")
    vsota = x + y
    return vsota
    print("Končano")

x = sestevalnik(1, 2)
print(x)
```

```
Seštevam...
3
```

Znotraj funkcije imamo lahko tudi več **return** statements, ki vrnejo različne vrednosti, glede na logiko funkcije.

In [98]:

```
def vecje_od_5(x):
    if x > 5:
        return True
    elif x <= 5:
        return False

print(vecje_od_5(1))
print(vecje_od_5(10))
```

```
False
True
```

In [ ]:

```

```

Returning Multiple Values

Funkcija lahko vrne le eno vrednost (bolje rečeno: le en objekt).

Če želimo vrniti več vrednosti jih preprosto zapakiramo v list, touple, dictionary in posredujemo tega.

In [100]:

```
def add_numbers(x, y, z):
    a = x + y
    b = x + z
    c = y + z
    return a, b, c # isto kot return (a, b, c)

sums = add_numbers(1, 2, 3)
print(sums)
print(type(sums))
```

```
(3, 4, 5)
<class 'tuple'>
```

Naloga:

Napišite funkcijo, ki sprejme nabor podatkov v obliki dictionary in vrne največjo vrednost vsakega ključa.

Primeri:

```
Input:
data = {"prices": [41970, 40721, 41197, 41137, 43033],
       "volume": [49135346712, 50768369805, 47472016405, 34809039137, 38700661463
]}
funkcija(data)

Output:
[43033, 50768369805]
```

In [ ]:

```

```

In [ ]:

```

```

Zanimivosti

Python funkcije so objekti. Lahko jih shranimo v spremenljivke, lahko jih posredujemo kot argumente ali vrnemo kot vrednost funkcije.

In [100]:

```
def hello(name):
    return f'My name is {name}'
```

In [101]:

```
print(hello("Gregor"))
```

```
My name is Gregor
```

In [102]:

```
funkcija = hello
print(funkcija("Gregor"))
print(funkcija)
print(type(funkcija))
```

```
My name is Gregor
<function hello at 0x0000015411EE6A60>
<class 'function'>
```

In [103]:

```
func = [hello, 2 ,3, 'Janez']
print(func[0](func[3]))
```

```
My name is Janez
```

In [ ]:

```

```

Naloga:

Ustvarite funkcijo, ki kot parametra vzeme list števil in neko število **m**, ki predstavlja zgornjo mejo.

Funkcija naj se sprehodi skozi podan list in vsako število, ki je večje od m, spremeni v m.

Funkcija naj na koncu vrne spremenjen list.

Primeri:

```
Input:
funkcija([1,12,-3,54,12,-22,65,32], 33)

Output:
[1, 12, -3, 33, 12, -22, 33, 32]
```

In [ ]:

```

```

Naloga:

Ustvari funkcijo, ki uredi list po vrstnem redu. Sprejme naj list in ukaz **asc** (naraščajoči vrstni red) ali **desc** (padajoči vrstni red). List naj nato ustrezno uredi. V kolikor ukaz ni posredovan naj bo default vrednost **asc**.

Primeri:

```
Input:
fun_03([1,4,2,8,4,0], ukaz="desc")

Output:
[8, 4, 4, 2, 1, 0]


Input:
fun_03([1,4,2,8,4,0], ukaz="asc")

Output:
[0, 1, 2, 4, 4, 8]


Input:
fun_03([5,8,-2,13,6,-6])

Output:
[-6, -2, 5, 6, 8, 13]
```

In [ ]:

```
```

In [ ]:

```
```

In [ ]:

```
```

In [ ]:

```
```