# Pojam, definicija i parametrizacija funkcije

Funkcije u Pythonu predstavljaju deo koda koji izvršava određenu logiku. Do sada smo se već upoznali sa nekim od funkcija kao što su `print()`, `type()` i slične koje su već ugrađene u Python programski jezik. Neke od prednosti korišćenja funkcija su:

- razlaganje kompleksnih problema u manje celine;
- smanjenje ponavljanja istih celina u kodu;
- unapređenje čitljivosti koda.

Funkcije u Pythonu imaju isti status kao i ostali objekti. Mogu biti dodeljene promenljivama, smeštene u kolekcije ili prosleđene kao argumenti funkcije.

## Tipovi funkcija u Pythonu

Postoje dva tipa funkcija u Pythonu:

- ugrađene funkcije – funkcije koje se nalaze u ugrađenim modulima koji dolaze sa instalacijom Pythona; u prethodnim lekcijama njih smo najviše i koristili (`print()`, `type()`);
- korisnički definisane funkcije – funkcije koje mi kao korisnici definišemo zarad rešavanja određenog problema; na njih ćemo se i fokusirati u ovoj nastavnoj jedinici.

## Sintaksa funkcije

Sintaksa funkcije izgleda ovako:

**Primer**

```
def function_name(parameters):
    local variables
    function logic
    return
```

Analiza sintakse:

- def – ključna reč koja prethodi samoj definiciji funkcije; nakon nje je obavezno ispisati i ime funkcije;
- function_name – funkciju možemo nazvati kako god želimo dokle god to ime nije isto kao neka od ključnih (rezervisanih) reči; o pravilima imenovanja funkcija je bilo reči u prethodnim nastavnim jedinicama;
- local variables – promenljive koje koristimo za rad u funkciji i koje ne postoje van nje ili kada funkcija završi izvršavanje;
- function logic – bilo kakva kalkulacija koja treba da se implementira u funkciji;
- return – ova ključna reč je potrebna ako želimo da vratimo neku vrednost iz funkcije dalje u program.

| Primer sintakse funkcije | |
|---|---|
| **Kod** | **Rezultat** |
| `def addition(a, b):`<br>`  print("Addition")`<br>`  return a + b`<br>`print(addition(1, 2))` | `Addition`<br>`3` |

*Tabela 15.1. Primer sintakse funkcije*

U ovom primeru smo prvo definisali funkciju `addition(a, b)` sa dva ulazna argumenta, ispisali njenu logiku i dalje u programu je pozvali koristeći njeno ime i uz to zadali dva proizvoljno određena broja.

## Funkcija kao objekat

Već smo napomenuli da funkcija ima isti status kao i ostali objekti; da metode koje se primenjuju na druge objekte važe i ovde, da funkcije mogu biti elementi u sekvencama i da se same funkcije mogu prosleđivati kao argumenti drugim funkcijama. Ovu funkcionalnost primećujemo u sledećem primeru:

Primer funkcije kao objekta

**Radno okruženje**

```
def f_one():
    pass

def f_two():
    pass

def f_three(f):
    print(id(f))


t = (f_one, f_two, f_three)
for i in t:
    print("Function:", i.__name__)
    print("Object is instance of object:", isinstance(i, object))
    print("Id:", id(i))

f_three(f_one)
```

U ovom primeru prvo primećujemo da je napravljena n-torka čiji su članovi imena funkcija. Nakon toga smo petljom prošli kroz sve članove te sekvence i primenili metode za čitanje imena datog objekta i proverili da li je dati objekat zaista tipa objekat i koji je identifikacioni broj objekta. Nakon toga smo pozvali funkciju `f_three()` sa zadatim argumentom. Nakon njenog izvršenja smo uvideli da se identifikacioni broj zadatog argumenta (a taj argument je zapravo funkcija `f_one()`) poklapa sa brojem `f_one()` funkcije koji je izlistan u prvoj iteraciji petlje.

## Naredba return

Funkcija se kreira kako bi izvršila određeni zadatak. Često se dešava da taj zadatak ima određeni rezultat koji bismo želeli da koristimo dalje u programu. Naredba `return` nam omogućava da se ta vrednost vrati dalje u program. Takođe, bitno je znati da ova naredba nije obavezna i da funkcija ne mora vraćati nikakvu vrednost. Vrednosti koje mogu da se vrate su svi objekti i tipovi podataka podržani u Pythonu.

**Primer korišćenja return funkcije**

Sledi primer korišćenja naredbe return unutar funkcije koja kvadrira broj koji se prosleđuje kao parametar.

```
def to_square(x):
    y = x * x
    return y

nbr = 5
result = to_square(nbr)
print('Result of square function is: ', result)
```

**Ispis:**

```
Result of square function is:  25
```

## Redefinicija funkcije

Ovo je takođe jedno od bitnih svojstava funkcija. Kako je Python dinamički programski jezik, moguće je da jednom definisanu funkciju redefinišemo u drugu.

| Primer redefinisanja funkcije | |
|---|---|
| **Kod** | **Rezultat** |
| `def f_one():`<br>`  print("First function")`<br>`f_one()`<br>`def f_one():`<br>`  print("Second function")`<br>`f_one()` | `First function`<br>`Second function` |

*Tabela 15.3. Primer redefinisanja funkcije*

Prvo smo definisali funkciju `f_one()`, pozvali je i dobili ispis `'First function'`. Dalje smo ponovo definisali funkciju po istom imenu, ali smo joj promenili logiku izvršavanja tako da na ekran ispisuje `Second function`. Posmatrajući ispis na ekranu nakon izvršavanja ovog programa primećujemo i ovu funkcionalnost.

**Pitanje**

Definisati funkciju na ovakav način je:

```
def f_one(a, b): return a + b
```

- pogrešno, jer će izbaciti sintaksičku grešku
- **dopustivo**

**Objašnjenje:**

*Još u pravilima kodiranja smo naučili da Python potpuno podržava više naredbi u jednoj liniji, ali da je to loša praksa i treba je izbegavati. Takođe, u ovom slučaju, moguće je definisati funkciju i ispisati njenu implementaciju u jednoj liniji, ali to dovodi do loše čitljivosti koda.*

## Argumenti funkcije (prosleđivanje po referenci)

Python podržava tri tipa argumenata; to su:

- podrazumevani (implicitni) argumenti;
- argumenti kao ključne reči;
- neodređeni (proizvoljni) broj argumenata.

## Podrazumevani (implicitni) argumenti

Argumenti u funkciji mogu imati podrazumevane vrednosti u slučaju da u trenutnom pozivu ne želimo ili nemamo potrebnu vrednost za prosleđivanje. Podrazumevanu vrednost argumentu funkcije dodeljujemo operatorom dodele (=) i tu vrednost ubacujemo prilikom definicije funkcije.

| **Primer upotrebe implicitnih argumenata** | |
|---|---|
| **Kod** | **Rezultat** |
| `def addition(a=1, b=2):`<br>`  print("Addition")`<br>`  return a + b`<br>`print(addition())`<br>`print(addition(3, 4))` | `Addition`<br>`3`<br>`Addition`<br>`7` |

*Tabela 15.4. Primer upotrebe implicitnih argumenata*

U ovom primeru smo prvo pozvali funkciju bez argumenata, pa su promenljive a i b unutar funkcije `addition()` iskoristile svoje podrazumevane vrednosti. Nakon toga smo istoj funkciji prosledili proizvoljne vrednosti za argumente a i b.

Kod ovog pristupa je važno napomenuti da, kada se parametri postave na podrazumevanu vrednost, svaki naredni parametar mora imati podrazumevanu vrednost, jer će u suprotnom doći do sintaksičke greške.

| Primer | |
|---|---|
| **Kod** | **Rezultat** |
| `def addition(a=1, b):`<br>`  print("Addition")`<br>`  return a + b`<br><br>`print(addition(2,3))` | `SyntaxError: non-default argument follows default argument` |

*Tabela 15.5. Primer greške kod postavljanja podrazumevanih vrednosti*

## Argumenti kao ključne reči

Ako argumente prosleđujemo kao ključne reči, možemo menjati redosled argumenata koji prosleđujemo:

| Primer argumenta kao ključne reči | |
|---|---|
| **Kod** | **Rezultat** |
| `def addition(a, b):`<br>`  print("Addition")`<br>`  return a + b`<br>`print(addition(b=2, a=1))` | `Addition`<br>`3` |

*Tabela 15.6. Primer argumenata kao ključnih reči*

Upišite kod u radno okruženje i proverite rezultate ovih primera:

| **Radno okruženje** |
|---|

Ako se opredelimo za ovakav način prosleđivanja argumenata, moramo ga se i držati. Ako bismo naredbu `print(addition(b=2, a=1))` promenili u `print(addition(b=2, 1))`, dobili bismo sintaksičku grešku.

## Neodređeni (proizvoljni) broj argumenata

Kada ne znamo koliko argumenata će funkcija primiti, koristimo operator (*) i na taj način prosleđujemo n-torku argumenata neodređene dužine. Takođe, konstrukcija (**) je validna i koristi se kada, umesto da samo prosledimo vrednosti proizvoljnog broja argumenata, uz njih prosleđujemo i njihove ključne reči. Na ovaj način zapravo prosleđujemo rečnik gde su vrednosti ključeva – ključne reči, a vrednosti na koje oni ukazuju vrednosti promenljivih koje prosleđujemo:

| Primer | |
|---|---|
| **Kod** | **Rezultat** |
| `def addition(*arguments):`<br>`  s = 0`<br>`  for i in arguments:`<br>`  s += i`<br>`  return s`<br>`print("Addition:",addition(2,1,3))` | `Addition: 6` |

*Tabela 15.7. Primer funkcije sa proizvoljnim brojem argumenata*

Ispis ovog programa će biti 6. Ulazni argument funkcije addition() – `'*arguments'` je tipa tuple, kroz koji iteriramo i vrednost svakog elementa dodajemo na promenljivu s (zbir).

Na sledećem primeru, gde prosleđujemo zapravo rečnik (**) vrednosti, uviđamo sličnu funkcionalnost.

Primer funkcije sa prosleđivanjem rečnika:

**Radno okruženje**

```
def addition(**kwargs):
    for i in kwargs:
        print("Key:",i, "Value:",kwargs[i])

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

## Prosleđivanje po referenci

Argumenti se funkcijama prosleđuju po referenci. Neki programski jezici kreiraju kopije objekata koji se prosleđuju, ali ne i Python. Na ovakav način se ostvaruju dve prednosti:

- sam proces je brži jer se ne troši vreme na kreiranje kopija;
- objekti koji mogu da se menjaju, a promene se u funkciji, permanentno su promenjeni.

**Primer prosleđivanja promenljive po referenci**

```
def func(y):
    y[0] = y[0]**2

x = [5]
func(x)
print ("X now holds the value of: ", x[0])
```

**Ispis:**

```
X now holds the value of:  25
```

**Objašnjenje:**

Ovde smo permanentno promenili vrednost promenljive x, koju smo prosledili funkciji func po referenci. Dakle, svaka promena koja se desi u toj funkciji sa našom promenljivom ostaje permanentna i nakon izvršenja te funkcije. Treba napomenuti da smo ovde koristili liste i radili isključivo na jednom elementu, jer ovakve radnje nad tipovima int i float nisu dozvoljene pošto ovi tipovi ne podržavaju menjanje njihovih vrednosti (immutable) bez ponovne dodele (operator dodele: =). Da smo x definisali kao x=5, vrednost bi ostala 5 na kraju programa.

## Opseg promenljivih (lokalne i globalne promenljive)

Na osnovu toga da li promenljive važe u trenutnoj funkciji ili u čitavoj skripti, promenljive u Pythonu se dele na lokalne i globalne. Razlika između ta dva tipa promenljivih se najbolje vidi na primeru funkcije:

**Primer**

```
a = 5 # promenljiva koja se vidi u celoj skripti (globalna)

def f():
    a = 10
    print(a)

print(a)
f()
```

U ovom primeru, iako smo prvo inicijalizovali promenljivu a na pet, u telu funkcije nije rečeno da želimo da koristimo upravo tu promenljivu, nego smo definisali lokalnu promenljivu pod istim imenom. Ako bismo želeli da upravljamo vrednošću prethodno definisane promenljive, u telo funkcije f() treba dodati liniju global a kao na sledećem primeru:

**Radno okruženje**

```
a = 5

def f():
    global a
    a = 10
    print(a)

print(a)
f()

print(a)
```

**Objašnjenje:**

Promenljiva a će biti vidljiva u celoj skripti jer uz pomoć ključne reči global postaje globalna. To znači da možemo da imamo pristup unutar funkcije promenljivi kao i da ako joj promenimo vrednost unutar funkcije ona će takva i biti van nje.

Tom naredbom global a smo rekli funkciji da na tu promenljivu gleda kao na globalnu i da svaka promena vrednosti te promenljive koja se desi u toj funkciji utiče i na vrednost te iste globalne promenljive.

## Rezime

- Funkcije u Pythonu mogu biti ugrađene (koje dolaze pri instalaciji) ili korisnički definisane (koje mi definišemo shodno svojim potrebama).
- Sintaksa definisanja funkcije podrazumeva navođenje ključne reči def, koju prati ime funkcije, nakon čega slede zagrade u kojima možemo, ali i ne moramo proslediti parametre: def function_name().
- Funkcija može, ali i ne mora imati return naredbu, ali mora sadržati barem jednu naredbu.
- Funkcije u Pythonu su takođe objekti i njihov tip je function.
- Argumenti funkcije se mogu proslediti kao ključne reči, kao implicitni (podrazumevani) i kao neodređeni.
- Prosleđivanje neodređenog broja argumenata se vrši jednostrukom ili dvostrukom zvezdicom (*, **) ispred argumenta pri definiciji funkcije, u zavisnosti od toga da li je reč o imenovanim argumentima (sa ključnim rečima) ili ne.
- Argumenti funkcija u Pythonu funkcionišu po principu prosleđivanja po referenci, što nam omogućava da promena promenljive koju prosleđujemo datoj funkciji i čiju vrednost menjamo važi i nakon završetka te funkcije.
- Kada želimo da nam promenljiva bude dostupna u svim funkcijama u trenutnom fajlu, definisaćemo je pomoću ključne reči global.