

Funkcije

Slajdovi za predmet Osnove programiranja

Katedra za informatiku, Fakultet tehničkih nauka, Novi Sad

2022.

Ciljevi

- razumevanje zašto je korisno podeliti program u skup više potprograma (funkcija)
- poznavanje pisanja funkcija u Pythonu
- poznavanje detalja pozivanja funkcija i prenosa parametara
- pisanje programa koji koriste funkcije – radi povećanja modularnosti i izbegavanja ponavljanja koda

Funkcija funkcija

- do sada smo videli različite vrste funkcija:
- u nekim primerima program se sastojao od jedne funkcije `main`
- ugrađene funkcije, npr. `abs`
- funkcije iz paketa, npr. `math.sqrt`

Funkcija funkcija ₂

- imati isti programski kod na više mesta donosi probleme
- problem 1: pisanje istog koda više puta je više posla
- problem 2: isti kod se mora održavati na više mesta istovremeno
- funkcije se mogu upotrebiti protiv ponavljanja istog koda, i pomažu da programi budu čitljiviji i lakši za održavanje

Funkcija funkcija ₃

- funkcija je kao potprogram, mali program unutar većeg programa
- osnovna ideja – napišemo niz naredbi i dodelimo mu ime;
- kasnije izvršavamo taj niz naredbi pozivajući ga po imenu

Funkcija funkcija ₄

- deo programa u kome se funkcija kreira zove se **definicija funkcije**
- kada se funkcija koristi u programu, kažemo da se ona **poziva**

Primer funkcija

- definicija funkcije

```
def main():  
    print("Happy birthday to you!")  
    print("Happy birthday to you!")  
    print("Happy birthday, dear Fred...")  
    print("Happy birthday to you!")
```

- poziv funkcije

```
>>> main()  
Happy birthday to you!  
Happy birthday to you!  
Happy birthday, dear Fred...  
Happy birthday to you!
```

Primer funkcija 2

- imamo ponavljanje u kodu:
`print("Happy birthday to you!")`
- možemo napisati funkciju koja ispisuje ovaj red:

```
def happy():  
    print("Happy birthday to you!")
```

- sada možemo da skratimo naš program

Primer funkcija ₃

- novi program

```
def singFred():  
    happy()  
    happy()  
    print("Happy birthday, dear Fred...")  
    happy()
```

- rezultat je:

```
>>> singFred()  
Happy birthday to you!  
Happy birthday to you!  
Happy birthday, dear Fred...  
Happy birthday to you!
```

Primer funkcija 4

- funkcija happy nam je uštedela dosta kucanja
- šta ako nam treba rođendan za Lucy?
- mogli bismo napisati funkciju za Lucy:

```
def singLucy():  
    happy()  
    happy()  
    print("Happy birthday, dear Lucy...")  
    happy()
```

Primer funkcija 5

- možemo napisati program za Freda i Lucy

```
def main():
    singFred()
    print()
    singLucy()
```

- dobićemo ovaj rezultat

```
>>> main()
Happy birthday to you!
Happy birthday to you!
Happy birthday, dear Fred..
Happy birthday to you!

Happy birthday to you!
Happy birthday to you!
Happy birthday, dear Lucy...
Happy birthday to you!
```

Primer funkcija 5

- još uvek ima ponavljanja koda
- jedina razlika između `singFred` i `singLucy` je ime u trećoj naredbi
- od ove dve funkcije može se napraviti jedna, sa parametrom

Primer funkcija ₆

- opštija funkcija sing

```
def sing(person):  
    happy()  
    happy()  
    print("Happy birthday, dear", person + ".")  
    happy()
```

- ova funkcija ima **parametar** sa nazivom person
- parametar je promenljiva čija vrednost se definiše prilikom poziva funkcije

Primer funkcija 7

- novi izlaz iz programa je:

```
>>> sing("Fred")  
Happy birthday to you!  
Happy birthday to you!  
Happy birthday, dear Fred.  
Happy birthday to you!
```

- sada možemo da preradimo i glavni program

Primer funkcija ₈

- novi glavni program glasi:

```
def main():  
    sing("Fred")  
    print()  
    sing("Lucy")
```

- rezultat je sada sledeći:

```
>>> main()  
Happy birthday to you!  
Happy birthday to you!  
Happy birthday, dear Fred.  
Happy birthday to you!
```

```
Happy birthday to you!  
Happy birthday to you!  
Happy birthday, dear Lucy.  
Happy birthday to you!
```

Funkcije i parametri

- **opseg** (scope) promenljive definiše deo programa u kome se promenljiva može koristiti
- svaka funkcija je poseban potprogram
- promenljive koje se koriste unutar funkcije su **lokalne** za tu funkciju
- makar imale isto ime kao i neke promenljive izvan te funkcije
- jedini način da funkcija „vidi“ podatak izvan sebe je da ga dobije kao parametar

Funkcije i parametri 2

- definicija funkcije izgleda ovako:

```
def <name>(<formal-parameters>):  
    <body>
```
- ime funkcije (`name`) mora biti identifikator
- `formal-parameters` je lista imena parametara (lista može biti prazna)

Funkcije i parametri ₃

- formalni parametri su dostupni samo u telu funkcije
- promenljive sa istim imenom u drugim delovima programa su različite od parametara i lokalnih promenljivih u funkciji

Funkcije i parametri ₄

- funkcija se poziva po imenu za kojim sledi lista stvarnih parametara ili argumenata:
`<name>(<actual-parameters>)`

Poziv funkcije

- kada Python naiđe na poziv funkcije, otpočinje proces od 4 koraka:
 - 1 pozivajući program zaustavlja izvršavanje u tački poziva funkcije
 - 2 formalni parametri funkcije dobijaju vrednosti stvarnih parametara iz poziva
 - 3 telo funkcije se izvršava
 - 4 kontrola se vraća u tačku odmah iza poziva funkcije

Poziv funkcije 2

- prođimo kroz sledeći kod:

```
sing("Fred")  
print()  
sing("Lucy")
```


- kada Python naiđe na `sing("Fred")`, izvršavanje `main` se privremeno zaustavlja
- Python traži definiciju funkcije `sing` i vidi da ona ima jedan formalni parametar, `person`

Poziv funkcije ₃

- formalni parametar dobija vrednost stvarnog parametra
- to je kao da smo izvršili sledeću naredbu

```
person = "Fred"
```

- promenljiva person je upravo inicijalizovana



The diagram shows a red arrow pointing from the string "Fred" in the `main` function's `sing("Fred")` call to the `person` parameter in the `sing` function's definition. Above the arrow, the text `person="Fred"` is written in red.

```
def main():
    sing("Fred")
    print()
    sing("Lucy")

def sing(person):
    happy()
    happy()
    print("Happy birthday, dear",
          person + ".")
    happy()
```

Poziv funkcije 4

- u ovom trenutku Python počinje izvršavanje tela funkcije `sing`
- prva naredba je poziv funkcije `happy`
- Python zaustavlja izvršavanje funkcije `sing` i prebacuje kontrolu na `happy`
- funkcija `happy` se sastoji iz jedne `print` naredbe koja se izvrši i kontrola se vraća nazad

Poziv funkcije 5

```
def main():
    sing("Fred")
    print()
    sing("Lucy")

def sing(person):
    happy()
    happy()
    print("dear", person)
    happy()

def happy():
    print("Happy birthday to you!")
```

- izvršavanje se nastavlja sa još dva poziva funkcije happy
- kada Python dođe do kraja funkcije sing, vraća kontrolu nazad funkciji main
- tamo nastavlja izvršavanje odmah posle poziva funkcije sing

Poziv funkcije 6

```
def main():
    sing("Fred")
    print()
    sing("Lucy")

def sing(person):
    happy()
    happy()
    print("Happy birthday, dear",
          person + ".")
    happy()
```

- promenljiva person u funkciji main nije vidljiva!
- memorija koju zauzimaju lokalne promenljive dok se funkcija izvršava se oslobađa nakon završetka funkcije
- lokalne promenljive ne čuvaju vrednost između dva poziva funkcije

Poziv funkcije 7

- sledeća naredba je `print()` koja ispisuje prazan red
- Python nailazi na još jedan poziv funkcije `sing`
- prenosi kontrolu na funkciju `sing`, a parametar ima vrednost "Lucy"

```
def main():
    sing("Fred")
    print()
    sing("Lucy")

def sing(person):
    happy()
    happy()
    print("Happy birthday, dear",
          person + ".")
    happy()
```

Poziv funkcije ₈

- telo funkcije `sing` sada se izvršava sa parametrom "Lucy"
- to uključuje tri poziva funkcije `happy`
- na kraju se kontrola prenosi nazad u `main`

```
def main():
    sing("Fred")
    print()
    sing("Lucy")
    ↓

def sing(person):
    happy()
    happy()
    print("Happy birthday, dear",
          person + ".")
    happy()
```

The diagram illustrates the execution flow. A red arrow points from the `sing("Lucy")` call in `main()` to the `def sing(person):` definition. From the definition, three red arrows point to the `happy()` calls, which then point back to the `sing` function body, showing the recursive nature of the calls. Finally, a red arrow points from the end of the `sing` function back to the line following `sing("Lucy")` in `main()`, indicating the return of control.

Poziv funkcije ₉

- prosleđivanje parametara je mehanizam za inicijalizaciju promenljivih u funkciji
- parametri predstavljaju **ulazne** podatke za funkciju
- možemo pozivati funkciju više puta sa različitim parametrima i dobiti različite rezultate

Rezultat funkcije

- videli smo primere funkcija koje vraćaju vrednost onome ko ih poziva
`root = math.sqrt(b*b - 4*a*c)`
- vrednost `b*b - 4*a*c` je stvarni parametar poziva funkcije `math.sqrt`
- kažemo da `math.sqrt` **vraća** kvadratni koren svog argumenta

Rezultat funkcije ₂

- ova funkcija vraća kvadrat datog broja:

```
def square(x):  
    return x*x
```

- kada Python naiđe na `return` vraća se nazad iz funkcije i predaje kontrolu u tački gde je funkcija pozvana
- vrednost data u `return` naredbi se šalje nazad onome ko je pozvao funkciju

Rezultat funkcije ₃

```
>>> square(3)
9
>>> print(square(4))
16
>>> x = 5
>>> y = square(x)
>>> print(y)
25
>>> print(square(x) + square(3))
34
```

Rezultat funkcije ₄

- možemo da iskoristimo funkciju `square` da napravimo funkciju za rastojanje dve tačke (x_1, y_1) i (x_2, y_2)

```
def distance(x1, y1, x2, y2):  
    dist = math.sqrt(square(x2 - x1) + square(y2 - y1))  
    return dist
```


Rezultat funkcije ₅

- nekad funkcija mora da vrati više od jedne vrednosti
- rezultat funkcije može biti lista ili sekvenca vrednosti

```
def sumDiff(x, y):  
    sum = x + y  
    diff = x - y  
    return sum, diff
```

Rezultat funkcije ₆

- kada se poziva ovakva funkcija, koristi se istovremena dodela više vrednosti

```
n1, n2 = eval(input("Unesite dva broja (n1, n2): "))  
s, d = sumDiff(n1, n2)  
print("Zbir je", s, "a razlika je", d)
```

- vrednosti se dodeljuju prema poziciji
- s će dobiti prvu vrednost, a d drugu

Rezultat funkcije 7

- sitna fora: sve Python funkcije vraćaju vrednost
- bez obzira da li postoji return ili ne
- ako ne postoji return vraća se specijalna vrednost None
- return se često zaboravi u pisanju koda!
 - ako se prilikom poziva funkcije dobijaju čudne greške, prvo proveriti ovo

Izmena parametara

- rezultat funkcije je osnovni način da se informacije vrate nazad onome ko poziva funkciju
- rezultat rada funkcije se može preneti nazad i tako što se parametar funkcije promeni
- razumevanje ovog postupka traži dublje poznavanje mehanizma dodele vrednosti i prenosa parametara

Izmena parametara 2

- recimo da pišemo program koji upravlja bankovnim računima
- treba nam funkcija koja dodaje kamatu na račun
- prva verzija funkcije

```
def addInterest(balance, rate):  
    newBalance = balance * (1 + rate)  
    balance = newBalance
```

Izmena parametara 3

- ideja je da se pomeni balance tako da se poveća za iznos kamate
- napišimo program za test:

```
def test():  
    amount = 1000  
    rate = 0.05  
    addInterest(amount, rate)  
    print(amount)
```

Izmena parametara 4

- nadamo se da će 5% biti dodato na račun i da će rezultat biti 1050

```
>>> test()
```

```
1000
```

- nema greške, sve je u redu!

Izmena parametara 5

1 prve dve linije u test-u kreiraju dve lokalne promenljive, amount i rate

2 one dobijaju vrednosti 1000 i 0.05

```
def addInterest(balance, rate):
    newBalance = balance * (1 + rate)
    balance = newBalance
```

```
def test():
    amount = 1000
    rate = 0.05
    addInterest(amount, rate)
    print(amount)
```


Izmena parametara 5

3 kontrola se dalje
prenosi na funkciju
addInterest

4 formalni parametri
balance i rate
dobijaju vrednosti
stvarnih parametara
amount i rate

5 iako rate postoji u
obe funkcije, to su
dve različite
promenljive

```
def addInterest(balance, rate):
    newBalance = balance * (1 + rate)
    balance = newBalance
```

```
def test():
    amount = 1000
    rate = 0.05
    addInterest(amount, rate)
    print(amount)
```

Izmena parametara 6

<p>6 dodela vrednosti za parametre balance i rate funkcije addInterest koristi vrednosti stvarnih parametara</p>	<pre>def addInterest(balance, rate): newBalance = balance * (1 + rate) balance = newBalance def test(): amount = 1000 rate = 0.05 addInterest(amount, rate) print(amount)</pre>
--------------------------------------------------------------------------------------------------------------------------------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

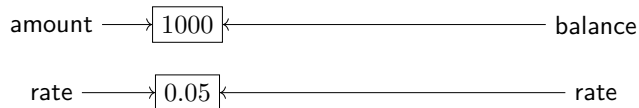
Izmena parametara 7

```
def test():
    amount = 1000
    rate = 0.05
    addInterest(amount, rate)
    print(amount)

def addInterest(balance, rate):
    newBalance = balance * (1 + rate)
    balance = newBalance
```

Diagram illustrating parameter passing and mutation:

- A red arrow points from `addInterest(amount, rate)` in the `test` function to the `addInterest` function definition.
- Red text annotations above the arrow indicate the mapping: `balance=amount` and `rate=rate`.



Izmena parametara 8

- 7 izvršavanje prve linije u `addInterest` će kreirati novu promenljivu `newBalance`
- 8 u drugoj liniji će `balance` primiti vrednost od `newBalance`

```
def addInterest(balance, rate):
    newBalance = balance * (1 + rate)
    balance = newBalance

def test():
    amount = 1000
    rate = 0.05
    addInterest(amount, rate)
    print(amount)
```

Izmena parametara 9

<p>9 balance sada pokazuje na istu vrednost kao i newBalance</p>	<pre>def addInterest(balance, rate): newBalance = balance * (1 + rate) balance = newBalance</pre>
<p>10 ali to nema uticaja na amount u test-u</p>	<pre>def test(): amount = 1000 rate = 0.05 addInterest(amount, rate) print(amount)</pre>

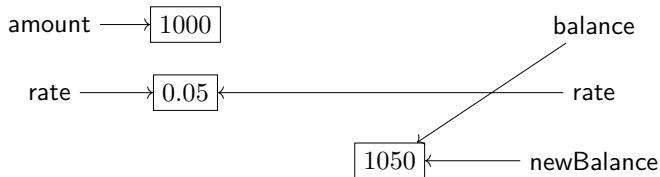
Izmena parametara 10

```
def test():
    amount = 1000
    rate = 0.05
    addInterest(amount, rate)
    print(amount)

def addInterest(balance, rate):
    newBalance = balance * (1 + rate)
    balance = newBalance
```

Diagram illustrating parameter passing and variable mutation:

- `test()` calls `addInterest(amount, rate)`.
- `addInterest` receives `balance` (initially `amount`) and `rate`.
- `addInterest` calculates `newBalance` and updates `balance`.
- `test()` prints `amount` (1000).



Izmena parametara 11

10 izvršavanje

addInterest se
završava i kontrola
se vraća test-u

```
def addInterest(balance, rate):
    newBalance = balance * (1 + rate)
    balance = newBalance
```

11 lokalne promenljive

iz addInterest
više ne postoje, ali
amount i rate u
test-u pokazuju i
dalje na stare
vrednosti

```
def test():
    amount = 1000
    rate = 0.05
    addInterest(amount, rate)
    print(amount)
```

Izmena parametara ₁₂

- rezime: formalni parametri funkcije primaju samo **vrednosti** stvarnih parametara
- funkcija nema pristup promenljivoj koja čuva stvarni parametar
- kaže se da Python sve parametre **prenosi po vrednosti**
- „pass by value“

Izmena parametara 13

- neki drugi jezici (C++, itd) omogućavaju da se same promenljive (a ne njihove vrednosti) prenesu kao parametar poziva funkcije
- ovaj mehanizam se zove **prenos po referenci**
- „pass by reference“
- tada se izmene parametara unutar funkcije vide nakon njenog poziva

Izmena parametara 14

- Python nema ovu mogućnost
- možemo izmeniti addInterest tako da vraća newBalance

```
def addInterest(balance, rate):  
    newBalance = balance * (1 + rate)  
    return newBalance  
  
def test():  
    amount = 1000  
    rate = 0.05  
    amount = addInterest(amount, rate)  
    print(amount)
```

Izmena parametara 15

- umesto da radimo sa jednim bankovnim računom, možemo da radimo sa više računa
- račune možemo čuvati u listi
- i kamatu dodavati na svaki račun u listi
- prvi račun bi se mogao menjati ovako:

```
balances[0] = balances[0] * (1 + rate)
```

Izmena parametara 16

```
balances[0] = balances[0] * (1 + rate)
```

- ovo znači „pomnoži vrednost nultog elementa liste sa $1+rate$ i to smesti nazad u nulti element liste“
- opštiji način da ovo uradimo bio bi pomoću petlje koja ide kroz listu sa indeksom 0, 1, ..., dužina-1

Izmena parametara 17

```
def addInterest(balances, rate):  
    for i in range(len(balances)):  
        balances[i] = balances[i] * (1+rate)  
  
def test():  
    amounts = [1000, 2200, 800, 360]  
    rate = 0.05  
    addInterest(amounts, 0.05)  
    print(amounts)
```

Izmena parametara 18

- početne vrednosti računa su bile
[1000, 2200, 800, 360]
- program je vratio
[1050.0, 2310.0, 840.0, 378.0]
- šta se desilo? izgleda kao da je amounts promenjen?

Izmena parametara 19

1 prve dve linije u
test-u inicijalizuju
promenljive
amounts i rate

2 vrednost
promenljive
amounts je lista
koja sadrži 4 int
vrednosti

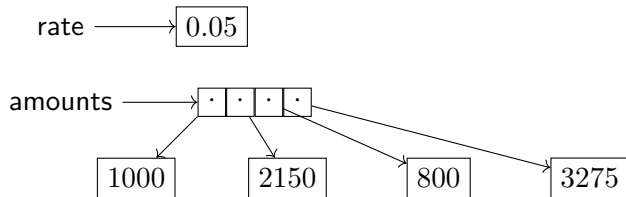
```
def addInterest(balances, rate):
    for i in range(len(balances)):
        balances[i] = balances[i] *
            (1+rate)
```

```
def test():
    amounts = [1000, 2200, 800, 360]
    rate = 0.05
    addInterest(amounts, 0.05)
    print(amounts)
```

Izmena parametara 20

```
def test():
    amounts = [1000, 2150, 800, 3275]
    rate = 0.05
    ↓ addInterest(amounts, rate)
    print(amounts)
```

```
def addInterest(balance, rate):
    for i in range(len(balances)):
        balances[i] = balances[i]*(1+rate)
```



Izmena parametara 21

- 3 izvršava se
addInterest
- 4 petlja ide kroz listu
0, 1, 2, 3 i ažurira
tu vrednost u
balances

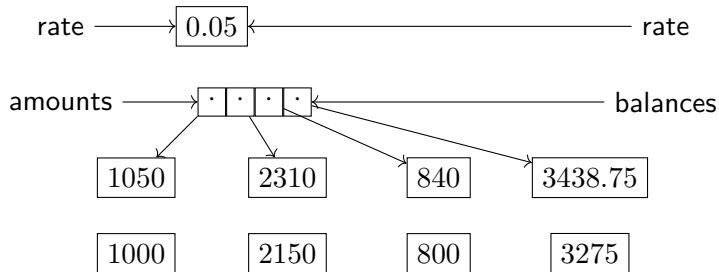
```
def addInterest(balances, rate):
    for i in range(len(balances)):
        balances[i] = balances[i] *
            (1+rate)

def test():
    amounts = [1000, 2200, 800, 360]
    rate = 0.05
    addInterest(amounts, 0.05)
    print(amounts)
```

Izmena parametara 22

```
def test():
    amounts = [1000, 2150, 800, 3275]
    rate = 0.05
    addInterest(amounts, rate)
    print(amounts)

def addInterest(balances, rate):
    for i in range(len(balances)):
        balances[i] = balances[i]*(1+rate)
```



Izmena parametara 23

5 stare vrednosti na dijagramu nisu obrisane već su ostavljene da „vise“ da bismo naglasili da se nisu promenili brojevi u svojim kutijama, već su novi dodeljeni elementima postojeće liste

6 stare vrednosti će iz memorije osloboditi [garbage collection](#) mehanizam

```
def addInterest(balances, rate):
    for i in range(len(balances)):
        balances[i] = balances[i] *
            (1+rate)

def test():
    amounts = [1000, 2200, 800, 360]
    rate = 0.05
    addInterest(amounts, 0.05)
    print(amounts)
```

Izmena parametara 24

- kada se `addInterest` završi i kontrola prenese nazad u test
- `amounts` nije promenjen (i dalje je ona stara lista)
- ali je stanje te stare liste promenjeno, i to je vidljivo u test-u

Izmena parametara 25

- parametri se **uvek** prenose po vrednosti
- ako je vrednost parametra promenljivi objekat, promene na tom objektu će biti vidljive nakon povratka iz funkcije

Funkcije sa promenljivim brojem parametara

```
def my_sum(*args):  
    result = 0  
    # parametrima pristupamo kao elementima sekvence  
    for x in args:  
        result += x  
    return result  
  
print(my_sum(1, 2, 3))
```

- ime args nije obavezno – može bilo koje drugo
- obavezno je navesti `*` – *unpacking operator*

Funkcije sa promenljivim brojem parametara

- `*args` mora doći posle obaveznih parametara

```
def print_greeting(greeting, *names):  
    for name in names:  
        print(greeting + " " + name)  
  
print_greeting("Happy birthday, dear",  
              "Joey", "Johnny", "Dee Dee", "Tommy")
```

Funkcije sa imenovanim parametrima

- `**kwargs` predstavlja imenovane parametre
- nepoznate u momentu pisanja funkcije

```
def concatenate(**kwargs):  
    result = ""  
    for arg in kwargs:  
        result += arg  
    result += " "  
    for arg in kwargs.values():  
        result += arg  
    return result  
  
print(concatenate(a="Real", b="Python", c="Is", d="Great", e="!"))  
>>> abcde RealPythonIsGreat!
```


Funkcija kao parametar

- funkcija – ne poziv funkcije! – se može proslediti kao parametar
- taj parametar se može iskoristiti za poziv dobijene funkcije
- ne znamo koja funkcija će biti pozvana u trenutku pisanja tog koda!

```
def greeter():  
    print("Hello")
```

```
def repeater(func, times):  
    for i in range(times):  
        func()
```

```
repeater(greeter, 3)
```

Funkcija kao parametar

```
def greeter():  
    print("Hello")
```

```
def repeater(func, times):  
    for i in range(times):  
        func()
```

```
repeater(greeter, 3)
```

- greeter() je poziv funkcije
- greeter je ime funkcije, koje se može proslediti kao parametar

Funkcija kao parametar

- primer: `sort()` ume da sortira, ali će za poređenje elemenata pozivati
- `compare()` koja se može proslediti kao parametar
- \Rightarrow možemo da menjamo logiku poređenja, dok algoritam za sortiranje ostaje isti

Funkcija u funkciji

- funkcija definisana unutar funkcije je vidljiva samo u toj funkciji

```
def print_integers(values):  
    def is_integer(value):  
        if type(value) is int:  
            return True  
        else:  
            return False  
  
    for v in values:  
        if is_integer(v):  
            print(v)  
  
print_integers([1,2,3,"4","tekst", 3.14])
```

Obmotavanje funkcije

- možemo „obmotati“ funkciju drugom funkcijom
- time dodati ponašanje funkciji nakon što je napisana

```
def print_call(fn):
    # funkcija prima bilo kakve argumente
    def fn_wrap(*args, **kwargs):
        print("Pozivam %s" % fn.__name__)
        # prosledi bilo kakve argumente funkciji fn
        return fn(*args, **kwargs)

    return fn_wrap # rezultat funkcije je funkcija!

suma = print_call(sum) # sum je ugradjena funkcija
print(suma([1, 2, 3, 4]))
```

Primer obmotane funkcije

```
def print_call(fn):
    def fn_wrap(*args, **kwargs):
        print("Pozivam %s" % fn.__name__)
        retval = fn(*args, **kwargs)
        print("Završen poziv")
        return retval

    return fn_wrap  # rezultat funkcije je funkcija!

def greeter(name):
    return "Hello, %s" % name

greeter = print_call(greeter)  # zadržavamo staro ime
print(greeter("Branko"))
```

Dekorator

- **dekorator** je funkcija koja prima funkciju i vraća funkciju
- u našem primeru `print_call` je dekorator
- dodavanje dekoratora na drugu funkciju je lako:

```
@print_call
def neka_funkcija(a, b, c):
    ...
```

- ovo je isto što i

```
def neka_funkcija(a, b, c):
    ...
neka_funkcija = print_call(neka_funkcija)
```

Zatvaranje (closure)

- funkcija „vidi“ promenljive iz opsega u kome se nalazi
- ali ne može da ih menja

```
>>> a = 0
>>> def get_a():
...     return a
...
>>> def set_a(val):
...     a = val
...
>>> get_a()
0
>>> a = 3
>>> get_a()
3
>>> set_a(4)
>>> a
3
```


Primer zatvaranja

- funkcija `nth_power` je *zatvorena* nad promenljivom `n`
- ima pristup ovoj promenljivoj nakon što izvršavanje programa napusti funkciju `generate_power_func`

```
def generate_power_func(n):
    def nth_power(x):
        return x**n
    return nth_power
```

```
raised_to_4 = generate_power_func(4)
del generate_power_func  # ukloni definiciju ove funkcije
print(raised_to_4(2))
```

```
# vrednost n je sacuvana za funkciju raised_to_4
print(raised_to_4.__closure__[0].cell_contents)
```

Modularni programi

- do sada smo funkcije koristili kao sredstvo da sprečimo ponavljanje koda
- drugi razlog za upotrebu funkcija je pisanje **modularnih** programa
- kako algoritmi koje kreiramo postaju sve složeniji, sve je teže čitati programski kod
- jedan način da se ovo reši je podela algoritma na manje potprograme gde je svaki od njih autonoman (ima smisla sam za sebe)