Skript jezici V03 – OOP

Milan Tomić

Sadržaj

- Objektno orijentisano programiranje u Pythonu
  - Klase
  - Atributi i metode
  - Konstruktor i specijalne metode
  - Nadjačavanje operatora
- Izuzeci
- Dekoratori

Klase

- Klase u Pythonu predstavljaju način za definisanje novih tipova objekata
- Python omogućuje mnogo više dinamičnosti nego što je to slučaj kod nekih drugih jezika
  - Objekat se često može proširivati proizvoljnim svojstvima
  - Klasa se takođe može nadograđivati posle definicije
- Python podržava višestruko nasleđivanje, sva polja i metode su po defaultu public i nasleđuju se po utvrđenom redu (MRO – method resolution order)
  - Ispis redosleda utvrđivanja metoda svake klase može se dobiti pozivom klasne metode mro()

Definicija klase

- Sve klase u Python-u 3 implicitno nasleđuju baznu klasu object ukoliko nije drugačije naglašeno
- Klase su nalik na strukture podataka sa pridruženim ponašanjima specifičnim za tu strukturu
- Najkraća definicija klase sadrži naziv klase i ključnu reč pass koja služi da označi prazan blok koda (na mestu gde se inače očekuje neki kod)
- Većina operatora sa posebnom sintaksom može da bude nadjačana koristeći specijalne metode
- Pored toga često se nadjačava __init__ metoda koja služi kao inicijalizator, tj. ima efekat parametrizovanog konstruktora u nekim programskim jezicima
- Kao što je ranije pomenuto, promenljive se u Pythonu ponašaju kao pokazivači na objekte, što znači da se prosleđivanje objekta u funkcije (ili metode) vrši pokazivački i sve promene su vidljive van opsega funkcije (metode)

Definicija klase

- Kada se klasa definiše, formira se klasni objekat koji se skladišti u lokalnom prostoru imena sa ostalim promenljivama
- Klasni objekat može da se iskoristi za
  - Pristup imenima definisanim u klasi (atributi, metode) – tačkasta notacija
  - Instancijaciju (kreiranje objekta te klase) – poziv kao poziv funkcije
- Sve metode kao prvi argument primaju ključnu reč self
  - self predstavlja objekat na kome je metoda pozvana, a u pitanju je konvencija (koju bi valjalo poštovati zbog statičke analize koda isl.)
  - pri pozivu metode self se automatski prosleđuje

Definicija i instancijacija klase

```
#%% Definicija klase

class PraznaKlasa:
    pass

# klasa sa konstruktorom
class Covek:
    def __init__(self, ime, prezime):
        self.ime = ime
        self.prezime = prezime

# instancijacija
prazno = PraznaKlasa()
print(prazno)  # <__main__.PraznaKlasa object>
covek = Covek('Mika', 'Mikić')
print(covek.ime, covek.prezime)  # Mika Mikić
```

Instance klase

- Objekti koji predstavljaju instance neke klase razumeju samo tačkastu notaciju – pristup atributima i to atributima (podacima) objekta i metodama (ponašanjima) objekta
- Za svaku funkciju definisanu unutar definicije klase, svakoj instanci te klase pridružuje se odgovarajuća metoda
  - Bitno: Klasa.f je objekat koji predstavlja funkciju (i nije vezan ni za jedan specifičan objekat tj. self nije definisano); objekat.f je objekat koji predstavlja metodu (i vezan je za objekat tj. self je definisano)
  - Klase, funkcije i metode su takođe objekti u Python-u i mogu se pridruživati promenljivama

Primer proširenja klase i poziva metode

```
#%% Proširenje klase i poziv metode

# definišemo funkciju
def predstavi_se(self):
    print('Ja sam', self.ime, self.prezime)

# dodeljujemo objekat funkcije kao atribut objektu klase
Covek.predstavi_se = predstavi_se

# instanca te klase sada može da koristi tu funkciju kao metodu
covek.predstavi_se()  # Ja sam Mika Mikić
```

Objekti metoda

- Objekti metoda su zapravo reference na metode vezane za konkretnu instancu neke klase
- Obično se metoda poziva onog trenutka kada je i tražimo
- U Pythonu postoji mogućnost da ponašanje jedne konkretne instance dodelimo nekoj promenljivoj i da preko te promenljive aktiviramo to ponašanje onda kada nam je to potrebno (bez reference na samu instancu za koju je metoda vezana)

Primer objekta metode

```
#%% Objekat metode i poziv bez referenciranja instance
predstavi_direktora = covek.predstavi_se

print('Obraćanje direktora:', end=' ')
predstavi_direktora() # Obraćanje direktora: Ja sam Mika Mikić
```

Atributi (promenljive u klasama)

- Klasni atributi određeni su u definiciji klase i deljeni su za sve njene instance
- Instancni atributi su specifični za svaku instancu
- Potrebno je voditi računa o tome da se klasni atribut ne pomeša sa instancnim, naročito ukoliko je u pitanju mutabilni tip, jer mutacija na toj instanci može dovesti do neželjenih efekata u drugim instancama
- Atributima klase ili instance nije moguće pristupiti unutar objekta te klase bez referenciranja konkretne instance (uglavnom pomoću self.)
  - Ovo otklanja opasnost od mogućnosti da se pomešaju lokalne promenljive u metodi sa promenljivama-atributima objekta
- Treba voditi računa da se naziv metode ne pomeša sa nazivom atributa, da se slučajno dodelom vrednosti ne bi pregazila definicija metode
  - Pametnim dizajnom klase može se uslovno sprečiti, ali na kraju je sve stvar konvencije

Nasleđivanje

- Pored definisanja ponašanja, klase su tu da nam omoguće i njihovo proširivanje pomoću potklasa
- Ponašanje nasleđivanja u Pythonu je jednostavno – nasleđeni atributi (promenljive i metode) koji nisu nadjačani u potklasi dobijaju se od natklase
- Moguće je višestruko nasleđivanje, pri čemu se poštuje specifičan redosled obilaska natklasa ukoliko atribut nije definisan u potklasi
- Ispis redosleda utvrđivanja metoda svake klase može se dobiti pozivom klasne metode mro()
- Poziv klasi iz natklase nad objektom klase može se izvršiti direktnim referenciranjem ili korišćenjem metode super()
  - super() dinamički određuje prvu natklasu nad kojom može da se pozove metoda
  - super(type, obj) ima i mogućnost da se u prvom argumentu naglasi od koje iz klasa u hijerarhiji se traži natklasa (po MRO) i da se za nju veže odgovarajuća instanca na kojoj će raditi

Primer nasleđivanja (klasa Test)

```
#%% Nasleđivanje

class MojTest(Test):
    # override
    def test(self):
        print('Moj test je uspeo!')

    def test2(self):
        print('Imam i drugi test')

t2 = MojTest()
t2.test()  # Moj test je uspeo!
t2.test = 'Glupost'  # Pokušaj gaženja sprečen!
t2.test2()  # Imam i drugi test
t2.atribut = 'Ok'  # Setovano atribut = Ok
print(t2.atribut) # Ok
```

Specijalne metode

- Specijalne (takođe magične) metode su metode koje interpreter poziva u trenutku kada se izvodi neka od ugrađenih operacija, kao što su:
  - len – dužina (veličina) kolekcije
  - in – provera pripadnosti
  - str – konverzija u string
  - + - sabiranje
  - == - poređenje za jednakost
  - . – tačkasta notacija (pristup atributu ili metodi)
  - [ ] – indeksiranje (pristup po ključu)
  - [:] – slajs
  - ( ) – poziv (kao poziv funkcije)
  - *I mnoge druge*

Specijalne metode

- Ove metode se retko (skoro nikad) ne pozivaju direktno, već posredstvom ugrađenih operacija koje pruža jezik
- Istovremeno, nadjačavanjem tih metoda omogućeno nam je da (re)definišemo ponašanja odgovarajućih operacija (tj. operatora) sa objektima naše klase
- Popis specijalnih metoda može se naći u dokumentaciji ovde: https://docs.python.org/3/reference/datamodel.html#special-method-names

Primer nadjačavanja str i len

```
#%% Specijalne metode str, len

class MojaKolekcija:
    def __init__(self, *args):
        self._abc = list(args)

    def __len__(self):
        return len(self._abc)

    def __str__(self):
        s = ''
        for x in self._abc:
            s += str(x) + ', '
        return s[:-2]

k = MojaKolekcija(1, 2, 3, 4, 'stop')
print('Moja kolekcija %s duzine %d' % (k, len(k)))
# Moja kolekcija 1, 2, 3, 4, stop duzine 5
```

Primer nadjačavanja + (spolja)

```
#%% Specijalna metoda za operaciju +

def add(self, other):
    if isinstance(other, MojaKolekcija):
        return MojaKolekcija(*self._abc, *other._abc)
    else:
        raise TypeError('Unsupported operand types')

MojaKolekcija.__add__ = add
j = MojaKolekcija('dalje', 'opet')
i = k + j
print('Moja kolekcija %s duzine %d' % (i, len(i)))
# Moja kolekcija 1, 2, 3, 4, stop, dalje, opet duzine 7
```

BONUS: Jedan način da se iznutra „spreči” slučajno „gaženje” metode atributom

```
#%% Sprečavanje dodele vrednosti metodi

class Test:

    def test(self):
        print('Test je uspeo')

    def __setattr__(self, name, value):
        a = getattr(self, name, None)
        if a and callable(a):
            print('Pokušaj gaženja sprečen!!')
        else:
            super.__setattr__(self, name, value)
            print('Setovano %s = %s' % (name, value))

t = Test()
t.test()  # Test je uspeo
t.test = 'Glupost'  # Pokušaj gaženja sprečen!
t.test()  # Test je uspeo
t.atribut = 'Ok'  # Setovano atribut = Ok
print(t.atribut)  # Ok
```

Izuzeci

- Izuzeci su čest način da se upravlja nepredviđenim ponašanjima programa
- Kada se desi izuzetak, interpreter ispisuje poruku o izuzetku i informacije o tome gde se desio (*stack traceback*) i prestaje sa izvršavanjem programa, osim ako se izuzetak uhvati i na neki način obradi, posle čega program može nesmetano da nastavi sa radom
  - Listu ugrađenih izuzetaka možete pronaći ovde: https://docs.python.org/3/library/exceptions.html#bltin-exceptions
  - Neki česti ugrađeni izuzeci koji se pojavljuju u radu su:
    - NameError – promenljiva je korišćena (za čitanje) ali prethodno nije definisana
    - TypeError – tip korišćenog podatka ne podržava korišćenu operaciju
    - ValueError – operacija ne može da se izvrši jer korišćeni argument nema prihvatljivu vrednost

Upravljanje izuzecima

- Python ne zahteva upravljanje izuzecima (na način na koji to radi Java)
  - Ali pojava svakog neuhvaćenog izuzetka prekida izvršavanje programa
- Očekivani izuzeci mogu se hvatati pomoću try-except konstrukta

```
#%% Upravljanje izuzecima
try:
    vrednost = 1/0
except Exception as ex:
    print('Uhvaćen izuzetak:', ex)  # Uhvaćen izuzetak: division by zero
```

try-except-else-finally

- try:
  - Izvršavaju se sve naredbe između try i except
  - Ako nema nikakvih izuzetaka, except se preskače
  - Ako se pojavi izuzetak, preskače se ostatak try i skače se na except; ako se tip uhvaćenog izuzetka slaže sa tipom izuzetka navedenim u except (*isinstance* veza), onda se izvršavaju naredbe u except
  - Inače, izuzetak se prosleđuje u spoljašnje try naredbe i tako sve dok se ne uhvati u neki except ili ako nema excepta da ga uhvati, program se zaustavlja
- except:
  - Može ih biti više za isti try, sa različitim upravljanjima za različite tipove
  - Redosled navođenja treba da ide od specijalnih ka opštim (tj. od potklasa ka natklasama)
  - Poslednji može biti i bez navođenja izuzetka (hvata sve izuzetke), ali to treba izbegavati

try-except-else-finally

- except:
  - Isti upravljač izuzecima može da hvata više različitih tipova izuzetaka iako nisu u istoj hijerarhiji, navođenjem u n-torci

```
#%% Hvatanje više klasa izuzetaka istim upravljačem
while True:
    try:
        vrednost = int(input('Uneti vrednost: '))
        print(2 / vrednost)
        break
    except (ValueError, ArithmeticError) as e:
        print('Uhvaćen izuzetak: ', e)
```

try-except-else-finally

- else:
  - Izvršava se posle try bloka ako nije došlo do izuzetka
  - U njemu je opet moguće da dođe do izuzetka koji su hvatani u except, ali se oni ne hvataju istim except-om
- finally
  - Izvršava se kao „čišćenje” nakon try, bilo da se try završio ili je nastupio except
  - Ako pred izvršavanje finally postoji izuzetak koji nije bačen, on se privremeno čuva
  - Ako se u finally desi novi izuzetak, stari izuzetak se postavlja kao kontekst novog i novi se baca
  - Ako se u finally naiđe na return ili break, prethodno sačuvani izuzetak se odbacuje (kao da se nije ni desio)
  - Ako se pre finally naiđe na return, finally se izvršava pre returna
  - Ako se u finally naiđe na return, on prikriva bilo koji raniji return

try-except-else-finally

```
#%% Kompletan try-except-else-finally
for r, b in ((False, False), (True, False), (True, True)):
    try:
        print(f'try ({r})')
        if r:
            raise ValueError('r je True')
    except ValueError as e:
        print('except', e)
        if b:
            print('raise')
            raise e
    else:
        print('else')
    finally:
        print('finally')
        if b:
            print('break')
            break
```

raise

- Za bacanje izuzetka koristi se ključna reč raise, pri čemu se nakon nje daje instanca klase izuzetka (ili samo klasa izuzetka kao prečica, ako nije potrebna specijalna inicijalizacija)
- Ukoliko želimo da bacimo izuzetak iz except bloka, možemo to učiniti i samo pozivajući raise, bez prosleđivanja instance
  - Biće prosleđen izuzetak koji se trenutno obrađuje

Dekoratori

- Dekorator je način da se određeno ponašanje „obogati” spoljnim ponašanjem bez potrebe da se to eksplicitno navodi
- U osnovi, dekorator je funkcija koja radi nad drugom funkcijom i uređuje (ili obogaćuje) njeno ponašanje
- Upotreba dekoratora u Python-u pojednostavljena je *sintaksnim šećerom* @dekorator, koji se stavlja iznad naziva funkcije (ili metode)
- Dekoratori mogu biti i parametrizovani, ali onda su to uglavnom dekoratori primenjeni na druge dekoratore (videćemo u nastavku)
- Dekoratori se u principu mogu primeniti i na metode i na klase (jer svaka funkcija kao argument može da primi bilo koji objekat, pa i objekat metode, odnosno klase)

Primer dekoratora

```
#%% Primer dekoratora

def pre_poziva(dekorisana_funkcija):
    def dekorisi(*args, **kw):
        print('Pre poziva ' + dekorisana_funkcija.__name__)
        x = dekorisana_funkcija(*args, **kw)
        print('Povratna vrednost: ' + repr(x))
        return x

    return dekorisi

# "novi" način
@pre_poziva
def saberi(x, y):
    return x + y

print(saberi(1, 2))    # Pre poziva saberi
                       # Povratna vrednost: 3
                       # 3
```

```
def oduzmi(x, y):
    return x - y

# "stari" način
oduzmi = pre_poziva(oduzmi)

print(oduzmi(3, 2))    # Pre poziva oduzmi
                       # Povratna vrednost: 1
                       # 1
```