Dekoratori, iteratori i generatori

Dekoratori

Dekoratori spadaju među moćnije mogućnosti u Pythonu. Iako ih je jednostavno koristiti, koncept je zapravo komplikovan. Šta je dekorator? Dekorator je vrsta funkcije koja uzima jednu funkciju kao argument i kao povratnu vrednost ima drugu funkciju. Postoje dve vrste dekoratora: dekoratori koji se odnose na klase i dekoratori koji se odnose na funkcije. Kako su klase predmet drugog kursa, u ovoj nastavnoj jedinici ćemo se baviti funkcijskim dekoratorima. Koncept dekoratora je apstraktan, pa ćemo se poslužiti postupnim primerom. Prvo ćemo definisati dve funkcije:

Primer

```
def first_function():
    print('first_function')
def second_function():
    print('second function')
first_function()
second_function()
```

Ako pokrenemo obe funkcije, izvršiće se obe print naredbe. Ovo je veoma jednostavan primer, ali ćemo ga nadograditi funkcijom `change_to_second_function(func)` pa sada naš primer izgleda ovako:

Radno okruženje

```
def change_to_second_function(func):
    return second_function

def first_function():
    print('first_function')

def second_function():
    print('second function')

first_function = change_to_second_function(first_function)
first_function()
```

U ovom koraku dodali smo `change_to_second_function(func)` sa parametrom `func` koji se trenutno ne koristi (prosleđujemo ga, ali nema nikakvu funkcionalnost unutar te funkcije), a koja vraća drugu funkciju. Ovim smo postigli to da funkciju nad kojom pozivamo `change_to_second_function()` zamenimo povratnom vrednošću funkcije `change_to_second_function()`.Dakle, `first_function()` sada postaje `second_function()`, što možemo proveriti naredbom `print(first_function())`, koja vraća u program ispis iz `second_function()` i to `'second function'` string.

Na ovaj način, dekorator smo kreirali ručno. Python podržava @ sintaksu za kreiranje dekoratora, i to:

Primer

```
def second_function():
    print('second function')

def change_to_second_function(func):
    return second_function

@change_to_second_function
def first_function():
    print('first_function')

first_function()
```

Ovde smo uveli dve promene: prva je to da smo dodali liniju kojom zapravo i definišemo dekorator (`@change_to_second_function`), a druga je to da smo uklonili liniju `first_function = change_to_second_function(first_function)` kojom smo u prethodnom primeru „ručno" kreirali dekorator.

Nakon pokretanja ovog primera dobijamo isti rezultat kao i u prethodnom primeru: string 'second function'. Na ovaj način smo dekorisali funkciju `first_function()` funkcijom `change_to_second_function(func)`.

Ovo jeste ispravan način za kreiranje dekoratora, ali nije i način koji se sreće u praksi. Jedan takav primer bi bio situacija gde prvenstveno imamo funkciju kojom pretvaramo string sa malim slovima u string sa velikim slovima:

Primer

```
def all_caps(s):
    return s.upper()
print(all_caps('new york'))
```

Ovo je vrlo jednostavna funkcija koja prima parametar s tipa string i vraća taj string sa velikim slovima.

Ako želimo da umesto stringa prosledimo listu stringova sa malim slovima, a da ne promenimo ovu funkciju – definisaćemo dekorator (`'@decorator'`) funkciju koja će nam omogućiti da funkciju `all_caps()` primenimo na svaki element prosleđene liste. Takav dekorator bi mogao izgledati ovako:

Primer

```
def decorator(func):
    # Lokalna (ugnežđena) funkcija
    def inner_function(l):
       return [func(el) for el in l]
```

```
        return inner_function

    @decorator
    def all_caps(s):
        return s.upper()

    cities= ['london', 'moscow', 'new york']
    print(all_caps(cities))
```

Konvencija je da se unutar dekoratorske funkcije definiše ugnežđena funkcija. Funkcija `decorator(func)` prima argument tipa funkcija (a to je naša funkcija (`all_caps()`)). Unutar decorator() funkcije smo definisali ugnežđenu funkciju `inner_function()` koja prima listu vrednosti za koje želimo da primenimo kapitalizaciju slova i konvertuje ih. Na kraju dekorator() funkcije, vraćamo referencu na ugnežđenu funkciju – `inner_function()`. Pa je tako funkcija `all_caps()` zamenjena sa `inner_function()` funkcijom. Ovo je tipičan način konstruisanja dekoratora.

Ono što se razlikuje u ovom primeru od prvobitnog (first_function(), second_function(), change_to_second_function()) je to što smo ovu ugnežđenu funkciju – `inner_function()` zamenili globalno definisanom funkcijom – `second_function()` u tom prvobitnom primeru.

Dekoratori možda deluju nepotrebno jer se takva funkcionalnost može postići na drugi način; njihova upotreba najčešća je u web rešenjima kao što su npr. Django i Flask. Jedan od uobičajenih načina korišćenja dekoratora u Djangu pokazuje koliko su korisni:

```
@route('home/products')
def get_products():
    pass
```

Na ovaj način smo rekli programu da server, kada se korisnik nađe na home/products stranici, izvrši funkciju `get_products()`.

Vežba

Imamo osnovnu funkciju operations(a,b) koja prima dva argumenta i ispisuje njihove vrednosti. Potrebno je ovu funkciju dekorisati sa dve funkcije: addition, u kojoj ćemo sabrati ova dva broja, i subtraction, u kojoj ćemo oduzeti ova dva broja. Izgled funkcije operations(a,b) je:

```
def operations(a,b):
      print("Passed numbers are: {} i {}".format(a,b))
```

Radno okruženje

```
def addition(func):
   def wrapper_function(a,b):
      print("Subtraction result is {}".format(a+b))
      return func(a,b)
```

```
    return wrapper_function



def subtraction(func):
    def wrapper_function(a,b):
        print("Subtraction result is {}".format(a-b))
        return func(a,b)

    return wrapper_function



@addition
@subtraction
def operations(a,b):
    print("Passed numbers are: {} i {}".format(a,b))



operations(9, 15)
```

Objašnjenje

Osnovnu funkciju smo dekorisali sa dve tražene funkcije. Upotrebili smo unutrašnje funkcije u kojima vršimo sabiranje, odnosno oduzimanje.

Iteratori

Iteratori su objekti koji omogućavaju prolazak kroz sve elemente date kolekcije bez obzira na poznavanje strukture te kolekcije. Ako možemo dobiti iterator od datog objekta, taj objekat možemo zvati i iterabilni objekat (`iterable`). U ove objekte spadaju: liste, n-torke, setovi, rečnici i stringovi, kao i fajl objekti.

Da bi objekat bio iterabilan, potrebno je da podržava metodu `iter()` koja će od datog objekta vratiti tip `iter`, kao i metodu `next()` koja će u program vratiti element iz sekvence.

Prednosti iteratora su:

- jasniji kod;
- mogućnost rada sa sekvencama neodređene dužine;
- manji utrošak resursa.

Sa iteratorima smo se ranije susretali kad god bismo iskoristili `for` petlju nad sekvencom. Naime, u `for` petlji se funkcija `iter()` primenjuje na prosleđeni objekat i tu dolazimo do dva

scenarija: ili dati objekat nije iterabilni objekat, u kom slučaju će doći do greške, ili on to jeste. U slučaju da objekat jeste iterabilni objekat – poziv funkcije `iter(objekat)` će vratiti iterabilni objekat. Sada, `for` petlja koristi upravo taj objekat za iteraciju koristeći metodu `next()` nad iterabilnim objektom. Ovo će se ponavljati dokle god `next()` metoda ne vrati `StopIteration` grešku.

Primer
Kod

`cities = ["new york", "london", "moscow"]`
`cities_iterator = iter(cities)`
`print(next(cities_iterator))`
`print(next(cities_iterator))`
`print(next(cities_iterator))`
`print(next(cities_iterator))`

Tabela 17.1. Primer korišćenja iter metode

Rezultat naredni je:

```
new york
london
moscow
Traceback (most recent call last):
  File "Iterators.py", line 6, in <module>
    print(next(cities_iterator))
```

Ovde uočavamo da nam je pri svakom pozivu `next()` metode nad iteratorom `cities_iterator` vraćen naredni element sve dok nismo i četvrti put pozvali tu metodu. U tom trenutku nam je vraćena `StopIteration` greška.

Funkcionalnost for petlje možemo implementirati u `while` petlji koristeći `iter() i next()` metode:

Primer

```
cities = ["new york", "london", "moscow"]
cities_iterator = iter(cities)

while cities_iterator:
    try:
        print(next(cities_iterator))

    except StopIteration:
        break
```

Ovaj put ćemo dobiti ispis čitave liste cities, ali bez izbacivanja greške.

Takođe, iteratori su jako bitni kod čitanja fajla, jer pomoću njih dobavljamo narednu liniju fajla, bez prethodnog učitavanja čitavog fajla u memoriju. Na taj način štedimo resurse sistema.

Pitanje

Dekorator je:

- objekat tipa rečnik
- objekat tipa iterator
- **objekat tipa funkcija ili klasa**

Objašnjenje:

Dekoratori su posebna vrsta funkcija (ili klasa, u zavisnosti od toga gde se primenjuju) koje uzimaju za argument jednu funkciju i proširuju njenu funkcionalnost bez menjanja originalnog koda prosleđene funkcije.

Generatori

Generatori su specijalne rutine koje se koriste pri kontroli ponašanja iteratora u petlji. Generator je najsličniji funkciji koja vraća sekvencu u program. Razlika je u načinu na koji to radi. Standardne funkcije vrednost u program vraćaju pomoću ključne reči `return`, dok generatorske funkcije vrednost u program vraćaju pomoću ključne reči `yield`, i to jednu po jednu vrednost, a ne celu sekvencu.

Primer	
Kod	**Rezultat**

`def even_numbers(a):`
`  for x in range(a):`
`  if x % 2==0:`
`  yield x`
`for e in even_numbers(10):`
`  print(e,end=', ')`
	`0, 2, 4, 6, 8,`

Tabela 17.2. Primer generatora

Ovim smo definisali generator `even_numbers()`, funkciju koja nam uvek u program vraća parnu vrednost. Ključna reč `yield` završava izvršavanje generator funkcije i vraća vrednost. Nakon što se izvrši `print()` naredba unutar `for` petlje, poziva se `next()` metoda nad generator funkcijom `even_numbers()` i ta funkcija ponovo vraća vrednost u program na osnovu datog uslova.

Iz ovoga se vidi da se generator definiše sa def ključnom rečju, kao i ostale funkcije, i da umesto `return` koristi `yield` naredbu za povratak vrednosti.

Takođe je moguće da generator ima dve i više `yield` naredbi, isto kao što i obična funkcija može imati više `return` naredbi.

Ako bismo želeli da rezultat generator funkcije ipak smestimo u listu, dovoljno je napraviti sledeći poziv:

```
even_numbers_list = list((even_numbers(10)))
```

Razlike sa listama i funkcijama

Razlika između liste i generatora se ogleda u tome što lista ili bilo koja druga sekvenca drži sve svoje elemente odjednom, dok generatori drže uvek jednu vrednost.

Razlika između funkcije i generatora se prvo ogleda u povratnoj vrednosti: u funkcijama vrednost vraćamo koristeći `return` naredbe, dok u generatorima to postižemo koristeći `yield` naredbe, što smo i pokazali na prethodnom primeru. Druga razlika između funkcije i generatora je što nakon izvršenja funkcije sve lokalne promenljive i njihove vrednosti nestaju, a kada se generator funkcija završi i vrati vrednost naredbom `yield`, program se ponovo vraća u istu tu funkciju i nastavlja izvršavanje od sledeće vrednosti. Primer generatora koji vraća vrednost unutrašnje (lokalne) promenljive nakon svake `yield` naredbe:

Primer	
Kod	**Rezultat**

`def func():`
`  i = 10`
`  while i > 0:`
`  yield i`
`  i -= 1`
`for i in func(): print(i, end=' ')`
	`10 9 8 7 6 5 4 3 2 1`

Tabela 17.3. Primer generatora koji vraća vrednost unutrašnje (lokalne) promenljive nakon svake `yield` naredbe

Ovde vidimo da promenljiva i tokom čitavog izvršavanja for petlje zadržava svoju vrednost.

Vežba

Napisati generatorsku funkciju koja ispisuje sve elemente deljive sa 3 od 1 do 50.

Radno okruženje

```
def div(a):
   for x in range(a):
      if x % 3==0:
         yield x

for e in div(50):
   print(e,end=', ')
```

Objašnjenje:

Definisali smo generatorsku funkciju po imenu deljivo(). Yieldovali smo svaku broj do 50. Poziv ove funkcije se koristi kao iterabilni objekat u for petlji.

Rezime

- Dekoratori su funkcije kojima „dekorišemo”, tj. proširujemo funkcionalnost određene funkcije, bez njenog menjanja.
- Iteratori su svi objekti koji podržavaju dve metode: iter() i next().
- Iterator metoda iter() će nam dati objekat, ako je to moguće, pretvoriti u iterabilni objekat.
- Iterator metoda next() će nam iz datog iterabilnog objekta vratiti sledeći element.
- Generatori su posebne funkcije koje sadrže barem jednu yield naredbu kojom vraćaju iterator.
- Razlika između generatora i liste je u tome što liste drže sve svoje vrednosti, dok generatori drže samo trenutnu vrednost u sebi.