Python - Decoradores

Um Decorator em Python é uma função que recebe outra função como argumento. A função de argumento é aquela a ser decorada pelo decorador. O comportamento da função de argumento é estendido pelo decorador sem realmente modificá-lo.

Neste capítulo, aprenderemos como usar o decorador Python.

Função em Python é um objeto de primeira ordem. Isso significa que pode ser passado como argumento para outra função, assim como outros tipos de dados, como número, string ou lista, etc. Também é possível definir uma função dentro de outra função. Essa função é chamada de função aninhada. Além disso, uma função também pode retornar outra função.

Sintaxe

A definição típica de uma função de decorador é a seguinte -

```
def decorator(arg_function): #arg_function to be decorated
   def nested_function():
      #this wraps the arg_function and extends its behaviour
      #call arg_function
      arg_function()
   return nested_function
```

Aqui está uma função Python normal -

```
def function():
   print ("hello")
```

Agora você pode decorar esta função para estender seu comportamento, passando-a para decorador -

```
function=decorator(function)
```

Se esta função for executada agora, ela mostrará a saída estendida pelo decorador.

Exemplo 1

O código a seguir é um exemplo simples de decorador -

```
def my_function(x):
   print("The number is=",x)

def my_decorator(some_function,num):
```

```
   def wrapper(num):
      print("Inside wrapper to check odd/even")
      if num%2 == 0:
         ret= "Even"
      else:
         ret= "Odd!"
      some_function(num)
      return ret
   print ("wrapper function is called")
   return wrapper

no=10
my_function = my_decorator(my_function, no)
print ("It is ",my_function(no))
```

A my_function() apenas imprime o número recebido. No entanto, seu comportamento é modificado passando-o para my_decorator. A função interna recebe o número e retorna se é ímpar/par. A saída do código acima é -

```
wrapper function is called
Inside wrapper to check odd/even
The number is= 10
It is Even
```

Exemplo 2

Uma maneira elegante de decorar uma função é mencionar, logo antes de sua definição, o nome do decorador precedido pelo símbolo @. O exemplo acima foi reescrito usando esta notação -

```
def my_decorator(some_function):
   def wrapper(num):
      print("Inside wrapper to check odd/even")
      if num%2 == 0:
         ret= "Even"
      else:
         ret= "Odd!"
      some_function(num)
      return ret
   print ("wrapper function is called")
   return wrapper

@my_decorator
def my_function(x):
   print("The number is=",x)
```

```
no=10
print ("It is ",my_function(no))
```

A biblioteca padrão do Python define os seguintes decoradores integrados -

@classmethod Decorador

O método de classe é uma função integrada. Ele transforma um método em um método de classe. Um método de classe é diferente de um método de instância. O método de instância definido em uma classe é chamado por seu objeto. O método recebeu um objeto implícito referido por self. Um método de classe, por outro lado, recebe implicitamente a própria classe como primeiro argumento.

Sintaxe

Para declarar um método de classe, a seguinte notação de decorador é usada -

```
class Myclass:
   @classmethod
   def mymethod(cls):
   #....
```

A forma @classmethod é a do decorador de função conforme descrito anteriormente. O mymethod recebe referência à classe. Ele pode ser chamado pela classe e também por seu objeto. Isso significa que Myclass.mymethod e Myclass().mymethod são chamadas válidas.

Exemplo 3

Vamos entender o comportamento do método de classe com a ajuda do exemplo a seguir -

```
class counter:
   count=0
   def __init__(self):
      print ("init called by ", self)
      counter.count=counter.count+1
      print ("count=",counter.count)
   @classmethod
   def showcount(cls):
      print ("called by ",cls)
      print ("count=",cls.count)

c1=counter()
c2=counter()
print ("class method called by object")
c1.showcount()
```

```
print ("class method called by class")
counter.showcount()
```

Na definição de classe, count é um atributo de classe. O método __init__() é o construtor e obviamente é um método de instância, pois recebeu self como referência de objeto. Cada objeto declarado chama esse método e aumenta a contagem em 1.

O decorador @classmethod transforma o método showcount() em um método de classe que recebe referência à classe como argumento mesmo que seja chamado por seu objeto. Pode ser visto mesmo quando o objeto c1 chama showcount, ele exibe a referência da classe do contador.

Ele exibirá a seguinte **saída** -

```
init called by <__main__.counter object at 0x000001D32DB4F0F0>
count= 1
init called by <__main__.counter object at 0x000001D32DAC8710>
count= 2
class method called by object
called by <class '__main__.counter'>
count= 2
class method called by class
called by <class '__main__.counter'>
```

@staticmethod Decorador

O staticmethod também é uma função integrada na biblioteca padrão do Python. Ele transforma um método em um método estático. O método estático não recebe nenhum argumento de referência, seja chamado pela instância da classe ou pela própria classe. A seguinte notação usada para declarar um método estático em uma classe -

Sintaxe

```
class Myclass:
@staticmethod
def mymethod():
#....
```

Embora Myclass.mymethod e Myclass().mymethod sejam chamadas válidas, o método estático não recebe referência de nenhuma delas.

Exemplo 4

A classe do contador é modificada conforme -

```
class counter:
   count=0
   def __init__(self):
      print ("init called by ", self)
      counter.count=counter.count+1
      print ("count=",counter.count)
   @staticmethod
   def showcount():
      print ("count=",counter.count)

c1=counter()
c2=counter()
print ("class method called by object")
c1.showcount()
print ("class method called by class")
counter.showcount()
```

Como antes, a contagem de atributos da classe é incrementada na declaração de cada objeto dentro do método __init__(). Porém, como mymethod(), ser um método estático não recebe parâmetro self ou cls. Conseqüentemente, o valor da contagem do atributo de classe é exibido com referência explícita ao contador.

A **saída** do código acima é a seguinte -

```
init called by <__main__.counter object at 0x000002512EDCF0B8>
count= 1
init called by <__main__.counter object at 0x000002512ED48668>
count= 2
class method called by object
count= 2
class method called by class
count= 2
```

@property Decorador

A função interna property() do Python é uma interface para acessar variáveis de instância de uma classe. O decorador @property transforma um método de instância em um "getter" para um atributo somente leitura com o mesmo nome e define a docstring da propriedade como "Obter o valor atual da variável de instância".

Você pode usar os três decoradores a seguir para definir uma propriedade -

- **@property** - Declara o método como uma propriedade.
- **@<nome da propriedade>.setter:** - Especifica o método setter para uma propriedade que define o valor para uma propriedade.

- **@<property-name>.deleter** - Especifica o método delete como uma propriedade que exclui uma propriedade.

Um objeto de propriedade retornado pela função property() possui métodos getter, setter e delete.

```
property(fget=None, fset=None, fdel=None, doc=None)
```

O argumento fget é o método getter, fset é o método setter. Opcionalmente, pode ter fdel como método para excluir o objeto e doc é a string da documentação.

O setter e o getter do objeto property() também podem ser atribuídos com a seguinte sintaxe.

```
speed = property()
speed=speed.getter(speed, get_speed)
speed=speed.setter(speed, set_speed)
```

Onde get_speed() e set_speeds() são os métodos de instância que recuperam e definem o valor para uma variável de instância speed na classe Car.

As declarações acima podem ser implementadas pelo decorador @property. Usar a classe de carro decorador é reescrito como -

```
class car:
   def __init__(self, speed=40):
      self._speed=speed
      return
   @property
   def speed(self):
      return self._speed
   @speed.setter
   def speed(self, speed):
      if speed<0 or speed>100:
         print ("speed limit 0 to 100")
         return
      self._speed=speed
      return

c1=car()
print (c1.speed) #calls getter
c1.speed=60 #calls setter
```

O decorador de propriedades é um método muito conveniente e recomendado para lidar com atributos de instância.