GAMAKA

Artificial Intelligence Solutions

Descriptor

Compiled by Mahesh Kaneri

Definition of descriptor:

Python descriptors are created to manage the attributes of different classes which use the object as reference. In descriptors we used three different methods that are `__getters__()`, `__setters__()`, and `__delete__()`. If any of those methods are defined for an object, it can be termed as a descriptor. Normally, Python uses methods like getters and setters to adjust the values on attributes without any special processing. It's just a basic storage system. Sometimes, You might need to validate the values that are being assigned to a value. A descriptor is a mechanism behind properties, methods, static methods, class methods, and `super()`.

Descriptor protocol:

In other programming languages, descriptors are referred to as setter and getter, where public functions are used to Get and Set a private variable. Python doesn't have a private variables concept, and descriptor protocol can be considered as a Pythonic way to achieve something similar. Descriptors are a new way to implement classes in Python, and it does not need to inherit anything from a particular object. To implement descriptors easily in python we have to use at least one of the methods that are defined above. Note that instance below returns to the object where the attribute was accessed, and the owner is the class where the descriptor was assigned as an attribute. There are three protocol in python descriptor for setters, getters and delete method.

- `gfg.__get__(self, obj, type=None)` : This attribute is called when you want to retrieve the information `(value = obj.attr)`, and whatever it returns is what will be given to the code that requested the attribute's value.
- `gfg.__set__(self, obj, value)` : This method is called to set the values of an attribute `(obj.attr = 'value')`, and it will not return anything to you.
- `gfg.__delete__(self, obj)` : This method is called when the attribute is deleted from an object `(del obj.attr)`

Invoking descriptor:

Descriptors are invoked automatically whenever it receives the call for a `set()` method or `get()` method. For example, `obj.gfg` looks up gfg in the dictionary of `obj`. If gfg defines the method `__get__()`, then `gfg.__get__(obj)` is invoked. It can also directly be invoked by method name i.e `gfg.__get__(obj)`

```
def __getattribute__(self, key):
    v = object.__getattribute__(self, key)
    if hasattr(v, '__get__'):
        return v.__get__(None, self)
    return v
```

The important points to remember are:

- Descriptors are invoked by the `__getattribute__()` method.
- Overriding `__getattribute__()` prevents automatic descriptor calls.
- `object.__getattribute__()` and `type.__getattribute__()` make different calls to `__get__()`.
- Data descriptors always override instance dictionaries.
- Non-data descriptors may be overridden by instance dictionaries.

Descriptor Example :

In this Example a data descriptor sets and returns values normally and prints a message logging their access.

```
class Descriptor(object):
    def __init__(self, name =''):
        self.name = name
    def __get__(self, obj, objtype):
        return "{}for{}".format(self.name, self.name)
    def __set__(self, obj, name):
        if isinstance(name, str):
            self.name = name
        else:
            raise TypeError("Name should be string")
class GFG(object):
    name = Descriptor()


g = GFG()
g.name = "Geeks"
print(g.name)
```

Creating a Descriptor using property() :

`property()`, it is easy to create a usable descriptor for any attribute. Syntax for creating `property()`

```
# Python program to explain property() function

# Alphabet class

class Alphabet:

     def __init__(self, value):

          self._value = value


     # getting the values

     def getValue(self):

          print('Getting value')

          return self._value


     # setting the values

     def setValue(self, value):

          print('Setting value to ' + value)

          self._value = value


     # deleting the values

     def delValue(self):

          print('Deleting value')

          del self._value


     value = property(getValue, setValue, delValue, )


# passing the value

x = Alphabet('GeeksforGeeks')
```

```
print(x.value)
x.value = 'GfG'
del x.value
```

Creating a Descriptor using class methods:

In this we create a class and override any of the descriptor methods `__set__`, `__ get__`, and `__delete__`. This method is used when the same descriptor is needed across many different classes and attributes, for example, for type validation.

```
class Descriptor(object):
     def __init__(self, name =''):
          self.name = name
     def __get__(self, obj, objtype):
          return "{}for{}".format(self.name, self.name)
     def __set__(self, obj, name):
          if isinstance(name, str):
               self.name = name
          else:
               raise TypeError("Name should be string")

class GFG(object):
     name = Descriptor()

g = GFG()
g.name = "Geeks"
print(g.name)
```

Creating a Descriptor using @property Decorator:

In this we use the power of property decorators which are a combination of property type method and Python decorators.

```
class Alphabet:
     def __init__(self, value):
          self._value = value

     # getting the values
     @property
     def value(self):
          print('Getting value')
          return self._value

     # setting the values
     @value.setter
     def value(self, value):
          print('Setting value to ' + value)
          self._value = value

     # deleting the values
     @value.deleter
     def value(self):
          print('Deleting value')
          del self._value
```

```
# passing the value
x = Alphabet('Peter')
print(x.value)

x.value = 'Diesel'

del x.value
```