Python - property() function

Traditional object-oriented languages like Java and C# use properties in a class to encapsulate data. Property includes the getter and setter method to access encapsulated data. A class in Python can also include properties by using the property() function.

Consider the following Python script which defines the person class as having the getter and setter methods. The getname() method returns the value of the private instance attribute \_\_name, while the setname() method assigns the value to the \_\_name attribute.

Example: person.py

class person:

def \_\_init\_\_(self, name="Guest"):

self.\_\_name=name

def setname(self, name):

self.\_\_name=name

def getname(self):

return self.\_\_name

The following interactive session shows the implementation of the person class from the above script.

>>> from person import person  
>>> p1=person()  
>>> p1.getname()  
'Guest'  
>>> p1.setname('Bill')  
>>> p1.getname()  
'Bill'

As you can see above, the p1.getname() method returns the value of attribute \_\_name and the setname() method assigns a value to it. However, it would be nice if the getter and setter functions were called implicitly when we access an attribute, just like a property in Java and C#. This is where Python's built-in property() function comes in handy.

The property() method in Python provides an interface to instance attributes. It encapsulates instance attributes and provides a property, same as Java and C#.

The property() method takes the get, set and delete methods as arguments and returns an object of the property class.

Signature:

prop=property(getter, setter, deleter, docstring)

The following example demonstrates how to create a property in Python using the property() function.

Example: property() function

class person:

def \_\_init\_\_(self):

self.\_\_name=''

def setname(self, name):

print('setname() called')

self.\_\_name=name

def getname(self):

print('getname() called')

return self.\_\_name

name=property(getname, setname)

In the above example, property(getname, setname) returns the property object and assigns it to name. Thus, the name property hides the private instance attribute \_\_name. The name property is accessed directly, but internally it will invoke the getname() or setname() method, as shown below.

>>> from person import person  
>>> p1=person()  
>>> p1.name="Steve"  
setname() called  
>>> p1.name  
getname() called  
'Steve'

As you can see above, the getname() method gets called automatically when we access the name property. In the same way, the setname method gets called when we assign a value to the name property. It also hides the instance attribute \_\_name.

In the same way, you can specify a deleter method for the property, as shown in the below script.

Example: property() function

class person:

def \_\_init\_\_(self, name):

self.\_\_name=name

def setname(self, name):

print('setname() called')

self.\_\_name=name

def getname(self):

print('getname() called')

return self.\_\_name

def delname(self):

print('delname() called')

del self.\_\_name

name=property(getname, setname, delname)

The delname() function would be invoked when you delete the name property.

>>> from person import person  
>>> p1=person()  
>>> p1.name="Steve"  
setname() called  
>>> del p1.name  
delname() called

In this way, we can define a property in the class using the property() function in Python.

@classmethod decorator

The @classmethod decorator can be applied on any method of a class. This decorator will allow us to call that method using the class name instead of the object.

Example: @classmethod

class person:

totalObjects=0

def \_\_init\_\_(self):

person.totalObjects=person.totalObjects+1

@classmethod

def showcount(cls):

print("Total objects: ",cls.totalObjects)

In the above example, @classmethod is applied on the showcount() method. The showcount() method has one parameter cls, which refers to the person class. Now, we can call the showcount() method using the class name, as shown below.

>>>p1=person()  
>>>p2=person()  
>>>person.showcount()  
Total objects: 2

However, the same method can be called using an object also.

>>>p1.person()  
>>>p1.showcount()  
Total objects: 2

@staticmethod decorator

The @staticmethod is a built-in decorator in Python which defines a static method. A static method doesn't receive any reference argument whether it is called by an instance of a class or by the class itself. The following notation is used to declare a static method in a class:

Example: Define Static Method

class person:

@staticmethod

def greet():

print("Hello!")

In the above example, @staticmethod is applied to the greet() method. So, the greet() method can be called using the class name person.greet(), as well as using the object.

>>> person.greet()  
Hello!  
>>> p1=person()  
>>> p1.greet()  
Hello!

Even though both person.greet() and p.greet() are valid calls, the static method receives reference of neither. Hence it doesn't have any arguments - neither self nor cls.

Python - Magic Methods

Magic methods in Python are the special methods which add "magic" to your class. Magic methods are not meant to be invoked directly by you, but the invocation happens internally from the class on a certain action. For example, when you add two numbers using the + operator, internally, the \_\_add\_\_() method will be called.

Built-in classes in Python define many magic methods. Use the dir() function to see the number of magic methods inherited by a class. For example, the following lists all the attributes and methods defined in the int class.

>>> dir(int)  
['\_\_abs\_\_', '\_\_add\_\_', '\_\_and\_\_', '\_\_bool\_\_', '\_\_ceil\_\_', '\_\_class\_\_', '\_\_delattr\_\_', '\_\_dir\_\_', '\_\_divmod\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_float\_\_', '\_\_floor\_\_', '\_\_floordiv\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_getnewargs\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_index\_\_', '\_\_init\_\_', '\_\_init\_subclass\_\_', '\_\_int\_\_', '\_\_invert\_\_', '\_\_le\_\_', '\_\_lshift\_\_', '\_\_lt\_\_', '\_\_mod\_\_', '\_\_mul\_\_', '\_\_ne\_\_', '\_\_neg\_\_', '\_\_new\_\_', '\_\_or\_\_', '\_\_pos\_\_', '\_\_pow\_\_', '\_\_radd\_\_', '\_\_rand\_\_', '\_\_rdivmod\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_rfloordiv\_\_', '\_\_rlshift\_\_', '\_\_rmod\_\_', '\_\_rmul\_\_', '\_\_ror\_\_', '\_\_round\_\_', '\_\_rpow\_\_', '\_\_rrshift\_\_', '\_\_rshift\_\_', '\_\_rsub\_\_', '\_\_rtruediv\_\_', '\_\_rxor\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_sub\_\_', '\_\_subclasshook\_\_', '\_\_truediv\_\_', '\_\_trunc\_\_', '\_\_xor\_\_', 'bit\_length', 'conjugate', 'denominator', 'from\_bytes', 'imag', 'numerator', 'real', 'to\_bytes']

As you can see above, the int class includes various magic methods surrounded by double underscores. For example, the \_\_add\_\_ method is a magic method which gets called when we add two numbers using the + operator. Consider the following example.

>>> num=10  
>>> num + 5  
15  
>>> num.\_\_add\_\_(5)  
15

As you can see, when you do num+10, the + operator calls the \_\_add\_\_(10) method. You can also call num.\_\_add\_\_(5) directly which will give the same result. However, as mentioned before, magic methods are not meant to be called directly, but internally, through some other methods or actions.

Magic methods are most frequently used to define overloaded behaviours of predefined operators in Python. For instance, arithmetic operators by default operate upon numeric operands. This means that numeric objects must be used along with operators like +, -, \*, /, etc. The + operator is also defined as a concatenation operator in string, list and tuple classes. We can say that the + operator is overloaded.

In order to make the overloaded behaviour available in your own custom class, the corresponding magic method should be overridden. For example, in order to use the + operator with objects of a user-defined class, it should include the \_\_add\_\_() method.

Let's see how to implement and use some of the important magic methods.

\_\_new\_\_() method

Languages such as Java and C# use the new operator to create a new instance of a class. In Python the \_\_new\_\_() magic method is implicitly called before the \_\_init\_\_() method. The \_\_new\_\_() method returns a new object, which is then initialized by \_\_init\_\_().

Example: \_\_new\_\_()

class employee:

def \_\_new\_\_(cls):

print ("\_\_new\_\_ magic method is called")

inst = object.\_\_new\_\_(cls)

return inst

def \_\_init\_\_(self):

print ("\_\_init\_\_ magic method is called")

self.name='Satya'

The above example will produce the following output when you create an instance of the Employee class.

>>> e1=employee()  
\_\_new\_\_ magic method is called  
\_\_init\_\_ magic method is called

Thus, the \_\_new\_\_() method is called before the \_\_init\_\_() method.

\_\_str\_\_() method

Another useful magic method is \_\_str\_\_(). It is overridden to return a printable string representation of any user defined class. We have seen str() built-in function which returns a string from the object parameter. For example, str(12) returns '12'. When invoked, it calls the \_\_str\_\_() method in the int class.

>>> num=12  
>>> str(num)  
'12'  
>>> #This is equivalent to  
>>> int.\_\_str\_\_(num)  
'12'

Let us now override the \_\_str\_\_() method in the employee class to return a string representation of its object.

Example:

class employee:

def \_\_init\_\_(self):

self.name='Swati'

self.salary=10000

def \_\_str\_\_(self):

return 'name='+self.name+' salary=$'+str(self.salary)

See how the str() function internally calls the \_\_str\_\_() method defined in the employee class. This is why it is called a magic method!

>>> e1=employee()  
>>> print(e1)  
name=Swati salary=$10000

\_\_add\_\_() method

In following example, a class named distance is defined with two instance attributes - ft and inch. The addition of these two distance objects is desired to be performed using the overloading + operator.

To achieve this, the magic method \_\_add\_\_() is overridden, which performs the addition of the ft and inch attributes of the two objects. The \_\_str\_\_() method returns the object's string representation.

Example: \_\_add\_\_()

class distance:

def \_\_init\_\_(self, x=None,y=None):

self.ft=x

self.inch=y

def \_\_add\_\_(self,x):

temp=distance()

temp.ft=self.ft+x.ft

temp.inch=self.inch+x.inch

if temp.inch>=12:

temp.ft+=1

temp.inch-=12

return temp

def \_\_str\_\_(self):

return 'ft:'+str(self.ft)+' in: '+str(self.inch)

Run the above Python script to verify the overloaded operation of the + operator.

>>> d1=distance(3,10)  
>>> d2=distance(4,4)  
>>> print("d1= {} d2={}".format(d1, d2))  
d1= ft:3 in: 10 d2=ft:4 in: 4  
>>>d3=d1+d2  
>>>print(d3)  
ft:8 in: 2

\_\_ge\_\_() method

The following method is added in the distance class to overload the >= operator.

Example: \_\_ge\_\_()

class distance:

def \_\_init\_\_(self, x=None,y=None):

self.ft=x

self.inch=y

def \_\_ge\_\_(self, x):

val1=self.ft\*12+self.inch

val2=x.ft\*12+x.inch

if val1>=val2:

return True

else:

return False

This method gets invoked when the >= operator is used and returns True or False. Accordingly, the appropriate message can be displayed

>>>d1=distance(2,1)  
>>>d2=distance(4,10)  
>>>d1>=d2  
False

Important Magic Methods

The following tables list important magic methods in Python 3.

| Initialization and Construction | Description |
| --- | --- |
| \_\_new\_\_(cls, other) | To get called in an object's instantiation. |
| \_\_init\_\_(self, other) | To get called by the \_\_new\_\_ method. |
| \_\_del\_\_(self) | Destructor method. |

| Unary operators and functions | Description |
| --- | --- |
| \_\_pos\_\_(self) | To get called for unary positive e.g. +someobject. |
| \_\_neg\_\_(self) | To get called for unary negative e.g. -someobject. |
| \_\_abs\_\_(self) | To get called by built-in abs() function. |
| \_\_invert\_\_(self) | To get called for inversion using the ~ operator. |
| \_\_round\_\_(self,n) | To get called by built-in round() function. |
| \_\_floor\_\_(self) | To get called by built-in math.floor() function. |
| \_\_ceil\_\_(self) | To get called by built-in math.ceil() function. |
| \_\_trunc\_\_(self) | To get called by built-in math.trunc() function. |

| Augmented Assignment | Description |
| --- | --- |
| \_\_iadd\_\_(self, other) | To get called on addition with assignment e.g. a +=b. |
| \_\_isub\_\_(self, other) | To get called on subtraction with assignment e.g. a -=b. |
| \_\_imul\_\_(self, other) | To get called on multiplication with assignment e.g. a \*=b. |
| \_\_ifloordiv\_\_(self, other) | To get called on integer division with assignment e.g. a //=b. |
| \_\_idiv\_\_(self, other) | To get called on division with assignment e.g. a /=b. |
| \_\_itruediv\_\_(self, other) | To get called on true division with assignment |
| \_\_imod\_\_(self, other) | To get called on modulo with assignment e.g. a%=b. |
| \_\_ipow\_\_(self, other) | To get called on exponentswith assignment e.g. a \*\*=b. |
| \_\_ilshift\_\_(self, other) | To get called on left bitwise shift with assignment e.g. a<<=b. |
| \_\_irshift\_\_(self, other) | To get called on right bitwise shift with assignment e.g. a >>=b. |
| \_\_iand\_\_(self, other) | To get called on bitwise AND with assignment e.g. a&=b. |
| \_\_ior\_\_(self, other) | To get called on bitwise OR with assignment e.g. a|=b. |
| \_\_ixor\_\_(self, other) | To get called on bitwise XOR with assignment e.g. a ^=b. |

| Type Conversion Magic Methods | Description |
| --- | --- |
| \_\_int\_\_(self) | To get called by built-int int() method to convert a type to an int. |
| \_\_float\_\_(self) | To get called by built-int float() method to convert a type to float. |
| \_\_complex\_\_(self) | To get called by built-int complex() method to convert a type to complex. |
| \_\_oct\_\_(self) | To get called by built-int oct() method to convert a type to octal. |
| \_\_hex\_\_(self) | To get called by built-int hex() method to convert a type to hexadecimal. |
| \_\_index\_\_(self) | To get called on type conversion to an int when the object is used in a slice expression. |
| \_\_trunc\_\_(self) | To get called from math.trunc() method. |

| String Magic Methods | Description |
| --- | --- |
| \_\_str\_\_(self) | To get called by built-int str() method to return a string representation of a type. |
| \_\_repr\_\_(self) | To get called by built-int repr() method to return a machine readable representation of a type. |
| \_\_unicode\_\_(self) | To get called by built-int unicode() method to return an unicode string of a type. |
| \_\_format\_\_(self, formatstr) | To get called by built-int string.format() method to return a new style of string. |
| \_\_hash\_\_(self) | To get called by built-int hash() method to return an integer. |
| \_\_nonzero\_\_(self) | To get called by built-int bool() method to return True or False. |
| \_\_dir\_\_(self) | To get called by built-int dir() method to return a list of attributes of a class. |
| \_\_sizeof\_\_(self) | To get called by built-int sys.getsizeof() method to return the size of an object. |

| Attribute Magic Methods | Description |
| --- | --- |
| \_\_getattr\_\_(self, name) | Is called when the accessing attribute of a class that does not exist. |
| \_\_setattr\_\_(self, name, value) | Is called when assigning a value to the attribute of a class. |
| \_\_delattr\_\_(self, name) | Is called when deleting an attribute of a class. |

| Operator Magic Methods | Description |
| --- | --- |
| \_\_add\_\_(self, other) | To get called on add operation using + operator |
| \_\_sub\_\_(self, other) | To get called on subtraction operation using - operator. |
| \_\_mul\_\_(self, other) | To get called on multiplication operation using \* operator. |
| \_\_floordiv\_\_(self, other) | To get called on floor division operation using // operator. |
| \_\_div\_\_(self, other) | To get called on division operation using / operator. |
| \_\_mod\_\_(self, other) | To get called on modulo operation using % operator. |
| \_\_pow\_\_(self, other[, modulo]) | To get called on calculating the power using \*\* operator. |
| \_\_lt\_\_(self, other) | To get called on comparison using < operator. |
| \_\_le\_\_(self, other) | To get called on comparison using <= operator. |
| \_\_eq\_\_(self, other) | To get called on comparison using == operator. |
| \_\_ne\_\_(self, other) | To get called on comparison using != operator. |
| \_\_ge\_\_(self, other) | To get called on comparison using >= operator. |