Python Decorators

Decorators in Python are special functions which adds additional functionality to an existing function or code.

For example, you had a white car with basic wheel setup and a mechanic changes the color of your car to red and fits alloy wheels to it then the mechanic decorated your car, similarly a decorator in python is used to decorate(or add functionality or feature) to your existing code.

In [20]:

```python
# some function
def first(msg):
    print(msg)

# second function
def second(func, msg):
    func(msg)

# calling the second function with first as argument
second(first, "Hello!")
```

```
Hello!
```

While in the example above the function second took the function first as an argument and used it, a function can also return a function.

When there is nested functions(function inside a function) and the outer function returns the inner function it is known as Closure in python.

Sample 1 - Adding $ to the return value from price() function

If we want a function that does prefix '$', the decorator can help us:

In [22]:

```python
def dollar(fn):
    def new(*args):
        return '$' + str(fn(*args))
    return new

@dollar
def price(amount, tax_rate):
    return amount + amount*tax_rate

print(price(100,0.1))

@dollar
def hello(number):
    return number*2

print(hello(30))
```

```
$110.0
$60
```

The dollar decorator function takes the price() function, and returns enhanced the output from the original price() after modifying the inner working.

Note that the decorator enables us to do it without making any changes on the price() function itself.

So, decorator works as a wrapper, modifying the behavior of the code before and after a target function execution, without the need to modify the function itself, enhancing the original functionality.

In [28]:

```python
def heading(f):
    def wrapped():
        return '<H1>' + f() + '</H1>'
    return wrapped

def bold(f):
    def wrapped():
        return '<b>' + f() + '</b>'
    return wrapped

def italic(f):
    def wrapped():
        return '<i>' + f() + '</i>'
    return wrapped


@heading
@bold
@italic
def welcome():
    return 'Welcome to Decorator'

print(welcome())
```

```
<H1><b><i>Welcome to Decorator</i></b></H1>
```

Welcome to Decorator

```
# Using Decorators in Python

A decorator gives a function a new behavior without changing the function itself. A decorator is used to add functionality to a function or a class. In other words, python decorators wrap another function and extends the behavior of the wrapped function, without permanently modifying it.

Now, Let's understand the decorators using an example:-
```

In [1]:

```python
# a decorator function
def myDecor(func):
    # inner function like in closures
    def wrapper():
        print("Modified function")
        func()
    return wrapper


def myfunc():
    print('Hello!!')

# Calling myfunc()
myfunc()

# decorating the myfunc function
decorated_myfunc = myDecor(myfunc)

# calling the decorated version
decorated_myfunc()
```

```
Hello!!
Modified function
Hello!!
```

In the code example above, we have followed the closure approach but instead of some variable, we are passing a function as argument, hence executing the function with some more code statements.

We passed the function myfunc as argument to the function myDecor to get the decorated version of the myfunc function.

Now rather than passing the function as argument to the decorator function, python provides us with a simple way of doing this, using the @ symbol.

In [3]:

```python
# using the decorator function
@myDecor
def myfunc():
    print('Hello!!')

# Calling myfunc()
myfunc()
```

```
Modified function
Hello!!
```

In the code example above, @myDecor is used to attach the myDecor() decorator to any function you want.

So when we will call myfunc(), instead of execution of the actual body of myfunc() function, it will be passed as an argument to myDecor() and the modified version of myfunc() is returned which will be executed.

So, basically @ is used to attach any decorator with name Decorator_name to any function in python programming language.

Decorators with arguments

Till now we have seen the use of decorators to modify function that hasn't used any argument. Now, let's see how to use argument with a function which is to be decorated.

For this, we are going to use *args and* *kwargs as the arguments in the inner function of the decorator.

The *args in function definition is used to pass a variable number of arguments to any function. It is used to pass a non-keyworded, variable-length argument list.

The **kwargs in function definitions is used to pass a keyworded, variable-length argument list. We use the name kwargs with the double star. The reason is that the double star allows us to pass through keyword arguments (and any number of them).

```
In [4]:
```

```python
def myDecor(func):
    def wrapper(*args, **kwargs):
        print('Modified function')
        func(*args, **kwargs)
    return wrapper

@myDecor
def myfunc(msg):
    print(msg)

# calling myfunc()
myfunc('Hey')


```

```
Modified function
Hey
```

In the example, the myfunc() function is taking an argument msg which is a message that will be printed.

The call will result in the decorating of function by myDecor decorator and argument passed to it will be as a result passed to the args of wrapper() function which will again pass those arguments while calling myfunc() function.

And finally, the message passed will be printed after the statement 'Modified function'.

Chaining the Decorators

We can use more than one decorator to decorate a function by chaining them. Let's understand it with an example:-

In [5]:

```python
# first decorator
def star(f):
    def wrapped():
        return '**' + f() + '**'
    return wrapped

# second decorator
def plus(f):
    def wrapped():
        return '++' + f() + '++'
    return wrapped

@star
@plus
def hello():
    return 'hello'

print(hello())
```

```
**++hello++**
```

In [ ]:

```python
class MyClass:
    def method(self):
        return 'instance method called', self

    @classmethod
    def classmethod(cls):
        return 'class method called', cls

    @staticmethod
    def staticmethod():
        return 'static method called'
```

Practical use of Decorators

Decorators are very often used for adding the timing and logging functionalities to the normal functions in a python program. Let's see one example where we will add the timing functionalities to two functions:

In [6]:

```python
import time

def timing(f):
    def wrapper(*args, **kwargs):
        start = time.time()
        result = f(*args,**kwargs)
        end = time.time()
        print(f.__name__ +" took " + str((end-start)*1000) +
        return result
    return wrapper

@timing
def calcSquare(numbers):
    result = []
    for number in numbers:
        result.append(number*number)
    return result

@timing
def calcCube(numbers):
    result = []
    for number in numbers:
        result.append(number*number*number)
    return result

# main method
if __name__ == '__main__':
    array = range(1,100000)
    sq = calcSquare(array)
    cube = calcCube(array)
```

```
calcSquare took 16.63804054260254 mil sec
calcCube took 25.146961212158203 mil sec
```

In the above example, we have created two functions calcCube and calcSquare which are used to calculate square and cube of a list of numbers respectively. Now, we want to calculate the time it takes to execute both the functions, for that we have defined a decorator timing which will calculate the time it took in executing both the functions.

Here we have used the time module and the time before starting a function to start variable and the time after a function ends to end variable. f.**name** gives the name of the current function that is being decorated. The code range(1,100000) returned a list of numbers from 1 to 100000.

So, by using decorators, we avoided using the same code in both the functions separately (to get the time of execution). This helped us in maintaining a clean code as well as reduced the work overhead.

In [11]:

```python
# PythonDecorators/my_decorator.py
class my_decorator(object):

    def __init__(self, f):
        print("inside my_decorator.__init__()")
        f() # Prove that function definition has completed

    def __call__(self):
        print("inside my_decorator.__call__()")

@my_decorator
def aFunction():
    print("inside aFunction()")

print("Finished decorating aFunction()")

aFunction()
```

```
inside my_decorator.__init__()
inside aFunction()
Finished decorating aFunction()
inside my_decorator.__call__()
```

In [8]:

```python
# PythonDecorators/decorator_with_arguments.py
class decorator_with_arguments(object):

    def __init__(self, arg1, arg2, arg3):
        """
        If there are decorator arguments, the function
        to be decorated is not passed to the constructor!
        """
        print("Inside __init__()")
        self.arg1 = arg1
        self.arg2 = arg2
        self.arg3 = arg3

    def __call__(self, f):
        """
        If there are decorator arguments, __call__() is only
        once, as part of the decoration process! You can only
        it a single argument, which is the function object.
        """
        print("Inside __call__()")
        def wrapped_f(*args):
            print("Inside wrapped_f()")
            print("Decorator arguments:", self.arg1, self.ar
            f(*args)
            print("After f(*args)")
        return wrapped_f

@decorator_with_arguments("hello", "world", 42)
def sayHello(a1, a2, a3, a4):
    print('sayHello arguments:', a1, a2, a3, a4)

print("After decoration")

print("Preparing to call sayHello()")
sayHello("say", "hello", "argument", "list")
print("after first sayHello() call")
sayHello("a", "different", "set of", "arguments")
print("after second sayHello() call")
```

```
Inside __init__()
Inside __call__()
After decoration
Preparing to call sayHello()
Inside wrapped_f()

Decorator arguments: hello world 42
sayHello arguments: say hello argument list
After f(*args)
after first sayHello() call
Inside wrapped_f()
Decorator arguments: hello world 42
sayHello arguments: a different set of arguments
After f(*args)
```

```
after second sayHello() call
```

In [10]:

```python
# PythonDecorators/entry_exit_class.py
class entry_exit(object):

    def __init__(self, f):
        self.f = f

    def __call__(self):
        print("Entering", self.f.__name__)
        self.f()
        print("Exited", self.f.__name__)

@entry_exit
def func1():
    print("inside func1()")

@entry_exit
def func2():
    print("inside func2()")

func1()
func2()
```

```
Entering func1
inside func1()
Exited func1
Entering func2
inside func2()
Exited func2
```

In [9]:

```python
# PythonDecorators/entry_exit_function.py
def entry_exit(f):
    def new_f():
        print("Entering", f.__name__)
        f()
        print("Exited", f.__name__)
    return new_f

@entry_exit
def func1():
    print("inside func1()")

@entry_exit
def func2():
    print("inside func2()")

func1()
func2()
print(func1.__name__)
```

```
Entering func1
inside func1()
Exited func1
Entering func2
inside func2()
Exited func2
new_f
```

In [7]:

```
# PythonDecorators/decorator_function_with_arguments.py
def decorator_function_with_arguments(arg1, arg2, arg3):
    def wrap(f):
        print("Inside wrap()")
        def wrapped_f(*args):
            print("Inside wrapped_f()")
            print("Decorator arguments:", arg1, arg2, arg3)
            f(*args)
            print("After f(*args)")
        return wrapped_f
    return wrap

@decorator_function_with_arguments("hello", "world", 42)
def sayHello(a1, a2, a3, a4):
    print('sayHello arguments:', a1, a2, a3, a4)

print("After decoration")

print("Preparing to call sayHello()")
sayHello("say", "hello", "argument", "list")
print("after first sayHello() call")
sayHello("a", "different", "set of", "arguments")
print("after second sayHello() call")
```

```
Inside wrap()
After decoration
Preparing to call sayHello()
Inside wrapped_f()
Decorator arguments: hello world 42
sayHello arguments: say hello argument list
After f(*args)
after first sayHello() call
Inside wrapped_f()
Decorator arguments: hello world 42
sayHello arguments: a different set of arguments
After f(*args)
after second sayHello() call
```

In [ ]:

```
#/Users/SurendraMac/Python27
```

In [ ]:

```python
# %load trace.py
#trace.py
def trace( aFunc ):
    """Trace entry, exit and exceptions."""
    def loggedFunc(*args, **kw ):
        print("enter", aFunc.__name__)
        try:
            result= aFunc(*args,**kw )
        except Exception, e:
            print("exception",aFunc.__name__, e)
            raise
        print("exit", aFunc.__name__)
        return result
    loggedFunc.__name__= aFunc.__name__
    loggedFunc.__doc__= aFunc.__doc__
```

In [ ]:

```
# %load trace_client.py
##Here's a class which uses our @trace decorator.
##trace_client.py
from trace1 import trace
class MyClass(object):
    #@trace
    def __init__( self, someValue,name='Surendra' ):
        """Create a MyClass instance."""
        self.value= someValue
        self.name=name
        print "Name and Value ",self.name,self.value
    #@trace
    def doSomething( self, anotherValue,age=35):
        """Update a value."""
        self.value += anotherValue
        print "another Value ",self.value


def hello1():
    def hello(*arg):
        '''This is hello document'''
        print "Hi"
    print "Hello1"
    return hello()

hello1()

#mc=MyClass(23)
#mc.doSomething(60)
#m=MyClass(45)
#enter __init__
#exit __init__
#mc.doSomething( 15 )
#m.doSomething(54)
#enter doSomething
#exit doSomething
# print mc.value
#38

```

In [ ]:

```python
# %load trace2.py
#trace.py
def trace(a):
    """Trace entry, exit and exceptions."""
    def hello(*b):
        return b
    return hello


print trace(78)

```

In [ ]:

```python
# %load trace_client3.py
##Here's a class which uses our @trace decorator.
##trace_client.py
from trace1 import trace
class MyClass:
    #@trace
##    def __init__( self, someValue):
##        """Create a MyClass instance."""
##        self.value= someValue
##        print "Value ",self.value
    #@trace
##    def doSomething( self, anotherValue):
##        """Update a value."""
##        self.value += anotherValue
##        print "another Value ",self.value
    @trace
    def hello(self):
        pass


mc=MyClass()
#mc.doSomething(60)
#m=MyClass(45)
#enter __init__
#exit __init__
#mc.doSomething( 15 )
#m.doSomething(54)
#enter doSomething
#exit doSomething
# print mc.value
#38

```

In [ ]:

```python
# %load trace1.py
#trace.py
def trace( aFunc ):
    """Trace entry, exit and exceptions."""
    def loggedFunc(*args,**kwarg):
        print "enter", aFunc.__name__
        print "Document of Function", aFunc.__doc__
        try:
            result= aFunc(*args,**kwarg)
        except Exception, e:
            print "exception",aFunc.__name__, e
            raise
        print "exit", aFunc.__name__
        #print result
        return result
    #loggedFunc.__name__= aFunc.__name__
    #loggedFunc.__doc__= aFunc.__doc__
    #print "Returning Log "
    return loggedFunc


@trace
def add(x,y):
    '''This is addition of Two Variable'''
    return x+y


@trace
def sub(x,y):
    ''' This is Substraction '''
    return x-y


print add(5,8)
print sub(9,3)


```

Class Methods

Let’s compare that to the second method, MyClass.classmethod. I marked this method with a @classmethod decorator to flag it as a class method.

Instead of accepting a self parameter, class methods take a cls parameter that points to the class—and not the object cls parameter that points to the class and not the object instance—when the method is called.

Because the class method only has access to this cls argument, it can’t modify object instance state. That would require access to self. However, class methods can still modify class state that applies across all instances of the class.

Static Methods
The third method, MyClass.staticmethod was marked with a @staticmethod decorator to flag it as a static method.

This type of method takes neither a self nor a cls parameter (but of course it’s free to accept an arbitrary number of other parameters).

Therefore a static method can neither modify object state nor class state. Static methods are restricted in what data they can access – and they’re primarily a way to namespace your methods.

Let’s See Them In Action!
I know this discussion has been fairly theoretical up to this point. And I believe it’s important that you develop an intuitive understanding for how these method types differ in practice. We’ll go over some concrete examples now.

Let’s take a look at how these methods behave in action when we call them. We’ll start by creating an instance of the class and then calling the three different methods on it.

MyClass was set up in such a way that each method’s implementation returns a tuple containing information for us to trace what’s going on — and which parts of the class or object the method can access.

Here’s what happens when we call an instance method:

```
>>> obj = MyClass()
>>> obj.method()
('instance method called', <MyClass instance at 0x101a2f4c8>)
```
This confirmed that method (the instance method) has access to the object instance (printed as <MyClass instance>) via the self argument.

When the method is called, Python replaces the self argument with the instance object, obj. We could ignore the syntactic sugar of the dot-call syntax (obj.method()) and pass the instance object manually to get the same result:

```
>>> MyClass.method(obj)
('instance method called', <MyClass instance at 
0x101a2f4c8>)
```

Can you guess what would happen if you tried to call the method without first creating an instance?

By the way, instance methods can also access the class itself through the self.__class__ attribute. This makes instance methods powerful in terms of access restrictions - they can modify state on the object instance and on the class itself.

Let’s try out the class method next:

```
>>> obj.classmethod()
('class method called', <class MyClass at 0x101a2f4c8>)
```

Calling classmethod() showed us it doesn’t have access to the <MyClass instance> object, but only to the <class MyClass> object, representing the class itself (everything in Python is an object, even classes themselves).

Notice how Python automatically passes the class as the first argument to the function when we call MyClass.classmethod(). Calling a method in Python through the dot syntax triggers this behavior. The self parameter on instance methods works the same way.

Please note that naming these parameters self and cls is just a convention. You could just as easily name them the_object and the_class and get the same result. All that matters is that they’re positioned first in the parameter list for the method.

Time to call the static method now:

```
>>> obj.staticmethod()
'static method called'
```

Did you see how we called staticmethod() on the object and were able to do so successfully? Some developers are surprised when they learn that it’s possible to call a static method on an object instance.

Behind the scenes Python simply enforces the access restrictions by not passing in the self or the cls argument

```
when a static method gets called using the dot syntax.

This confirms that static methods can neither access the
object instance state nor the class state. They work like
regular functions but belong to the class’s (and every
instance’s) namespace.

Now, let’s take a look at what happens when we attempt to
call these methods on the class itself – without creating
an object instance beforehand:

>>> MyClass.classmethod()
('class method called', <class MyClass at 0x101a2f4c8>)

>>> MyClass.staticmethod()
'static method called'

>>> MyClass.method()
TypeError: unbound method method() must
    be called with MyClass instance as first
    argument (got nothing instead)
```

class method vs static method in Python

Class Method

The @classmethod decorator, is a builtin function decorator that is an expression that gets evaluated after your function is defined. The result of that evaluation shadows your function definition. A class method receives the class as implicit first argument, just like an instance method receives the instance Syntax:

class C(object): @classmethod def fun(cls, arg1, arg2, ...): .... fun: function that needs to be converted into a class method returns: a class method for function. A class method is a method which is bound to the class and not the object of the class. They have the access to the state of the class as it takes a class parameter that points to the class and not the object instance. It can modify a class state that would apply across all the instances of the class. For example it can modify a class variable that will be applicable to all the instances. Static Method

A static method does not receive an implicit first argument. Syntax:

class C(object): @staticmethod def fun(arg1, arg2, ...): ... returns: a static method for function fun. A static method is also a method which is bound to the class and not the object of the class. A static method can't access or modify class state. It is present in a class because it makes sense for the method to be present in class. Class method vs Static Method

A class method takes cls as first parameter while a static method needs no specific parameters. A class method can access or modify class state while a static method can't access or modify it. In general, static methods know nothing about class state. They are utility type methods that take some parameters and work upon those parameters. On the other hand class methods must have class as parameter. We use @classmethod decorator in python to create a class method and we use @staticmethod decorator to create a static method in python. When to use what?

We generally use class method to create factory methods. Factory methods return class object (similar to a constructor) for different use cases. We generally use static methods to create utility functions. How to define a class method and a static method?

To define a class method in python, we use @classmethod decorator and to define a static method we use @staticmethod decorator. Let us look at an example to understand the difference between both of them. Let us say we want to create a class Person. Now, python doesn't support method overloading like C++ or Java so we use class methods to create factory methods. In the below example we use a class method to create a person object from birth year.

As explained above we use static methods to create utility functions. In the below example we use a static method to check if a person is adult or not.

Implementation

filter_none edit play_arrow

brightness_4

Python program to demonstrate

use of class method and static method.

from datetime import date

class Person: def **init**(self, name, age): self.name = name self.age = age

```
# a class method to create a Person object by birth year.
@classmethod
def fromBirthYear(cls, name, year):
    return cls(name, date.today().year - year)

# a static method to check if a Person is adult or not.
@staticmethod
def isAdult(age):
    return age > 18
```

person1 = Person('mayank', 21) person2 = Person.fromBirthYear('mayank', 1996)

print person1.age print person2.age

print the result

print Person.isAdult(22) Output

21 21 True This article is contributed by Mayank Agrawal. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org (mailto:contribute@geeksforgeeks.org). See your article appearing on the GeeksforGeeks main page and help other Geeks.

In [30]:

```python
def function1(*arg, **kwarg):
    print(arg)
    print(kwarg)


function1()
function1(5,7)
function1('hello','hi',23,34)
function1(2,3,4, var1=20, var2=40 )
```

```
()
{}
(5, 7)
{}
('hello', 'hi', 23, 34)
{}
(2, 3, 4)
{'var1': 20, 'var2': 40}
```

In [ ]:

```python

```