Chapter 29
Decorators

29.1 Introduction

The idea behind Decorators comes from the *Gang of Four* Design Patterns book (so-called as there were four people involved in defining these design patterns). In this book numerous commonly occurring object oriented design patterns are presented. One of these design patterns is the Decorator design pattern.

The Decorator pattern addresses the situation where it is necessary to add additional behaviour to specific objects. One way to add such additional behaviour is to decorate the objects created with types that provide the extra functionality. These decorators wrap the original element but present exactly the same interface to the user of that element. Thus the Decorator Design pattern extends the behaviour of an object without using sub classing. This decoration of an object is transparent to the decorators' clients.

In Python Decorators are functions that take another function (or other callable object such as a method) and return a third function representing the decorated behaviour.

This chapter introduces decorators, how they are defined, how they are used and presents built-in decorators.

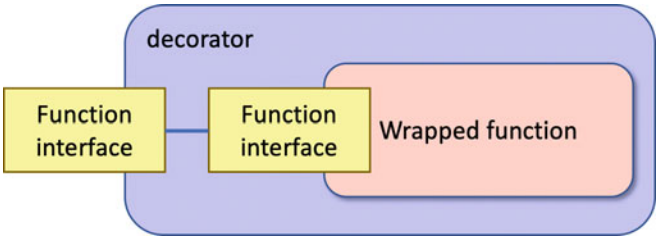
29.2 What Are Decorators?

A Decorator is a piece of code, that is used to mark a callable object (such as a function, method, class or object) typically to enhance or modify its behaviour (potentially replacing it). It thus *decorates* the original behaviour.

Decorators are in fact callable objects themselves and as such behave more like macros in other languages that can be applied to callable objects that then return a new callable object (typically a new function).

J. Hunt, *A Beginners Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-030-20290-3_29

The basic idea is illustrated in the following diagram:



This diagram illustrates a decorator wrapping a callable object (in this case a function). Note that the decorator presents exactly the same interface to the user of the decorator as the original function would present; that is, it takes the same parameters and either returns nothing (`None`) or something.

It should also be noted that the decorator is also at liberty to completely replace a callable object rather than to wrap it; it's a design decision made by the implementor of the decorator.

29.3 Defining a Decorator

To define a decorator you need to define a callable object such as a function that takes another function as a parameter and returns a new function.

An example of the definition of a very simple logger decorator function is given below.

```
def logger(func):
    def inner():
        print('calling ', func.__name__)
        func()
        print('called ', func.__name__)

    return inner
```

In this case the `logger` decorator wraps the original function within a new function, here called `inner`. When this function is executed, a statement is logged before and after the original function is executed.

Every function has an attribute `__name__` that provides the functions *name* and this is used in the `inner()` function above to printout the actual function about to be invoked.

Note that the function `inner()` is defined inside the `logger()` function (this is completely legal). A reference to the `inner()` function is then returned as the result of the `logger()` function. The `inner()` function is not executed at this point!

29.4 Using Decorators

To see what the effect of applying a decorator is; it is useful to explore the basic (explicit) approach to it use. This can be done by defining a function (we will call `target`) that prints out a simple message:

```
def target():
    print('In target function')
```

We can explicitly apply the `logger` decorator to this function by passing the reference to the `target` function (without invoking it), for example:

```
t1 = logger(target)
t1()
```

When we run this code, we actually execute the `inner()` function which was returned by the decorator. This function in turn prints out a message and then calls the function passed into the `logger`. Once this passed in function has executed, it prints another message. The effect of executing the `t1()` function is this to call the `inner()` function which calls the `target` function, thus printing out:

```
calling  target
In target function
called  target
```

This illustrates what happens when a decorator style function is *executed*.

Python provides some syntactic sugar that allows the definition of the function and the association with the decorator to be declared together using the `'@'` syntax, for example:

```
@logger
def target():
    print('In target function')

target()
```

This has the same effect as passing `target` into the `logger` function; but illustrates the role of the `logger` in a rather more Pythonic way. It is thus the more common use of Decorators.

The output of this function is the same as the previous version.

29.5 Functions with Parameters

Decorators can be applied to functions that take parameters; however the decorator function must also take these parameters as well.

For example, if you have a function such as

```
@logger
def my_func(x, y):
    print(x, y)

my_func(4, 5)
```

Then the function returned form the decorator must also take two parameters, for example:

```
def logger(func):
    def inner(x, y):
        print('calling ', func.__name__, 'with', x, 'and', y)
        func(x, y)
        print('returned from ', func.__name__)
    return inner
```

29.6 Stacked Decorators

Decorators can be stacked; that is more than one decorator can be applied to the same callable object. When this occurs, each function is wrapped inside another function; this idea is illustrated by the following code:

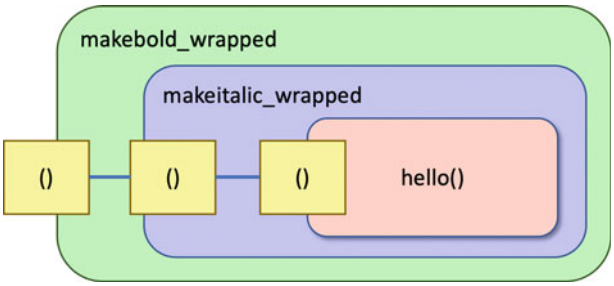
```
# Define the decorator functions
def make_bold(fn):
    def makebold_wrapped():
        return "<b>" + fn() + "</b>"
    return makebold_wrapped

def make_italic(fn):
    def makeitalic_wrapped():
        return "<i>" + fn() + "</i>"
    return makeitalic_wrapped

# Apply decorators to function hello
@make_bold
@make_italic
def hello():
    return 'hello world'

# Call function hello
print(hello())
```

In this example, the function `hello()` is marked with two decorators, `@make_bold` and `@make_italic`.

This means that the function `hello()` is first passed into the `make_italic()` function and wrapped by the `makeitalic_wrapped` function. This function is then returned from the `make_italic` decorator.

The `makeitalic_wrapped` is then passed into the `make_bold()` function which then wraps it inside the `makebold_wrapped` function; which is returned by the `make_bold` decorator.

This means that the function invoked when `hello()` is called is the `makebold_wrapped` function which calls two further functions as shown below:



The end result is that the string returned by the function passed in is first wrapped by `<i>` and `</i>` (indicating italics) and then by `<b>` and `</b>` (indicating bold) in HTML.

Thus the output from `print(hello())` is:

```
<b><i>hello world</i></b>
```

29.7 Parameterised Decorators

Decorators can also take parameters however the syntax for such decorators is a little different; there is essentially an extra layer of indirection. The decorator function takes one or more parameters and returns a function that can use the parameter and takes the callable object that is being wrapped. For example:

```
def register(active=True):
    def wrap(func):
        def wrapper():
            print('Calling ', func.__name__, ' decorator param
', active)
            if active:
                func()
                print('Called ', func.__name__)
            else:
                print('Skipped ', func.__name__)
        return wrapper
    return wrap

@register()
def func1():
    print('func1')

@register(active=False)
def func2():
    print('func2')

func1()
print('-' * 10)
func2()
```

In this example, the wrapped function will only be called if the active parameter is `True`. This is the default and so for `func1()` it is not necessary to specify the parameter (although note that it is now necessary to provide the round brackets).

For `func2()` the `@register` decorator is defined with the active parameter set to False. This means that the wrapper function will not call the provided function.

Note that the usage of the decorator only differs in the need to include the round brackets even if no parameters are being specified; even though there are now two inner functions defined within the `register` decorator.

29.8 Method Decorators

29.8.1 Methods Without Parameters

It is also possible to decorate methods as well as functions (as they are also callable objects). However, it is important to remember that methods take the special parameter `self` as the first parameter which is used to reference the object that the method is being applied to. It is therefore necessary for the decorator to take this parameter into account; i.e. the *inner* wrapped function must take at least one parameter representing `self`:

```
def pretty_print(method):
    def method_wrapper(self):
        return "<p>{0}</p>".format(method(self))

    return method_wrapper
```

The `pretty_print` decorator defines an inner function that takes as its first (and in this case only) parameter the reference to the object (which by convention uses the parameter `self`). This is then passed into the actual method when it is called.

The `pretty_print` decorator can now be used with any method that only takes the `self` parameter, for example:

```
class Person:
    def __init__(self, name, surname, age):
        self.name = name
        self.surname = surname
        self.age = age

    def print_self(self):
        print('Person - ', self.name, ', ', self.age)

    @pretty_print
    def get_fullname(self):
        return self.name + " " + self.surname
```

In the above class the `get_fullname()` method is decorated with `pretty_print`. If we now call `get_fullname()` on an object, the resulting string will be wrapped in `<p>` and `</p>` (which is HTML markup for a paragraph):

```
print('Starting')
p = Person('John', 'Smith', 21)
p.print_self()
print(p.get_fullname())
print('Done')
```

This generates the output:

```
Starting
Person -  John ,  21
<p>John Smith</p>
Done
```

29.8.2 Methods with Parameters

As with functions, methods that take parameters in addition to `self` can also be decorated. In this case the function returned form the decorator must take not only the `self` parameter but also any parameters passed to the method. For example:

```
def trace(method):
    def method_wrapper(self, x, y):
        print('Calling', method, 'with', x, y)
        method(self, x, y)
        print('Called', method, 'with', x, y)
    return method_wrapper
```

Now this `trace` decorator defines an inner function that takes the parameter `self` and two additional parameters. It can be used with any method that also takes two parameters such as the method `move_to()` below:

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    @trace
    def move_to(self, x, y):
        self.x = x
        self.y = y

    def __str__(self):
        return 'Point - ' + str(self.x) + ',' + str(self.y)
```

When a `Point` object is created below, we can call the `move_to()` method and see the result:

```
p = Point(1, 1)
print(p)
p.move_to(5, 5)
print(p)
```

The output from this is:

```
Point - 1,1
Calling <function Point.move_to at 0x110288b70> with 5 5
Called <function Point.move_to at 0x110288b70> with 5 5
Point - 5,5
```

29.9 Class Decorators

As well as being able to decorate functions and methods; it is possible to decorate classes.

A class can be decorated to add required functionality that may be external to that class.

As an example, a common class level operation is to want to indicate that a class should implement the singleton design pattern. The Singleton Design Pattern (again from the Gang of Four Design Patterns book) describes a type that can only have one object constructed for it. That is, unlike other objects it should not be possible to obtain more than one instance within the same program. Thus, the *Singleton* design pattern ensures that only one instance of a class is created. All objects that use an instance of that type use the same instance.

We can define a decorator that implements the singleton design pattern, for example:

```
def singleton(cls):
    print('In singleton for: ', cls)
    instance = None

    def get_instance():
        nonlocal instance
        if instance is None:
            instance = cls()
        return instance

    return get_instance
```

This decorator returns the `get_instance()` function. This function checks to see if the variable instance is set to `None` or not; if it is set to `None` it instantiates the class passed into the decorator and stores this in the instance variable. It then returns the instance. If the instance is already set it merely returns the instance.

We can apply this decorator to whole classes such as `Service` and `Foo` below:

```
@singleton
class Service(object):
    def print_it(self):
        print(self)

@singleton
class Foo(object):
    pass
```

We can now use the classes `Service` and `Foo` as normal; however only one instance of `Service` and one instance of `Foo` will ever be created in the same program:

```
print('Starting')
s1 = Service()
print(s1)
s2 = Service()
print(s2)
f1 = Foo()
print(f1)
f2 = Foo()
print(f2)
print('Done')
```

In the above code snippet, it looks as if we have created two new `Service` objects and two `Foo` objects; however, the `@singleton` decorator will restrict the number of instances created to one and will reuse that instance whenever a request is made to instantiate the given class. Thus when we run this example, we can see that the hexadecimal number representing the location of the object in memory is the same for the two `Service` objects and the same for the two `Foo` objects:

```
In singleton for:  <class '__main__.Service'>
In singleton for:  <class '__main__.Foo'>
Starting
<__main__.Service object at 0x10ac3f780>
<__main__.Service object at 0x10ac3f780>
<__main__.Foo object at 0x10ac3f7b8>
<__main__.Foo object at 0x10ac3f7b8>
Done
```

29.10 When Is a Decorator Executed?

An important feature of decorators is that they are executed right after the decorator function is defined. This is usually at *import time* (i.e. when a module is loaded by Python).

```
def logger(func):
    print('In Logger')
    def inner():
        print('In inner calling ', func.__name__)
        func()
        print('In inner called ', func.__name__)
    print('Finishing Logger')
    return inner

@logger
def print_it():
    print('Print It')

print('Start')
print_it()
print('Done')
```

For example, the decorator `logger` shown above, prints out 'In Logger' and 'Finished Logger' when it is executed. If the output is examined, it can be seen that this output occurs before the program prints ‘Start’.

```
In Logger
Finishing Logger
Start
In inner calling  print_it
Print It
In inner called  print_it
Done
```

Note that the decorated function and the wrapped function only execute when they are explicitly invoked.

This highlights the difference between what Pythonistas call *import time* and *runtime*.

29.11 Built-in Decorators

There are numerous built-in decorators in Python 3; some of which we have already seen such as `@classmethod`, `@staticmethod` and `@property`. We also saw some decorators when talking about abstract methods and properties. There are also decorators associated with unit testing and asynchronous operations.

29.12 FuncTools Wrap

One issue with decorated functions may become apparent when debugging or trying to trace what is happening. The problem is that by default the attributes associated with the function being called are actually those of the inner function returned by the decorator function. That is the `name`, `doc` and `module` of the function are those of the function returned by the decorator. The `name` and documentation of the original, decorated function, have been lost.

For example, returning to the original logger decorator we have:

```
def logger(func):
    def inner():
        print('calling ', func.__name__)
        func()
        print('called ', func.__name__)
    return inner


@logger
def get_text(name):
    """returns some text"""
    return "Hello "+name


print('name:', get_text.__name__)
print('doc: ', get_text.__doc__)
print('module; ', get_text.__module__)
```

When we run this code we get:

```
name: inner
doc:  None
module;  __main__
```

It appears that the `get_text` function is called inner and has no *docstring* associated with it. However, if we look at the function it should be called `get_text()` and does have a *docstring* of 'returns some text'!

Python (since version 2.5) has included the `functools` module which contains the `functools.wraps` decorator which can be used to overcome this problem.

Wraps is a decorator for updating the attributes of the wrapping function(inner) to those of the original function (in this case `get_text()`). This is as simple as decorating the 'inner' function with `@wraps(func)`.

```
from functools import wraps

def logger(func):
    @wraps(func)
    def inner():
        print('calling ', func.__name__)
        func()
        print('called ', func.__name__)
    return inner
```

The end result is that in the above example the `name` and `doc` are now updated to the name of the wrapped function and the documentation associated with that function.

If we now rerun the earlier example, we get:

```
name: get_text
doc:  returns some text
module;  __main__
```

29.13 Online Resources

For further information on decorators see:

- https://www.python-course.eu/python3_decorators.php a short introduction to Python decorators
- https://www.python.org/dev/peps/pep-0318/ PEP 318 considering decorators for functions and methods.
- https://www.python.org/dev/peps/pep-3129/ PEP 3129 introducing class decorators.
- https://docs.python.org/3.7/library/functools.html Python Standard Library documentation for `functoools`.
- https://pymotw.com/3/functools/index.html Python module of the Week for `functoools`.

- https://github.com/lord63/awesome-python-decorator A useful page that lists many Python decorators as well as third party contributions.
- https://wiki.python.org/moin/PythonDecoratorLibrary which provides a central repository for decorator examples.

29.14 Book Reference

For more information on the Decorator and Singleton design patterns see the "Patterns" book by the Gang of Four (E. Gamma, R. Helm, R. Johnson and J. Vlissades, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995).

29.15 Exercises

The aim of this exercise its to develop your own decorator.

You will write a `timer` decorator to be used with methods in a class that take the first parameter `self`, followed by one other parameter.

The decorator should log how long a method takes to execute.

To do this you can use the `time` module and import the `default_timer`.

```
from timeit import default_timer
```

You can then obtain a `default_timer` object for the start and end of a function call and use these values to generate the time taken, for example:

```
start = default_timer()
func(self, value)
end = default_timer()
print('returned from ', func, 'it took', end - start,
'seconds')
```

You can then apply the decorator to the `deposit()` and `withdraw()` methods defined in the `Account` class. For example,

```
@timer
def deposit(self, amount):
    self._balance += amount

@timer
def withdraw(self, amount):
    self._balance -= amount
```

These methods will be inherited by the `DepositAccount` and `InvestmentAccout` classes. In the `CurrentAccount` class the `withdraw` method is over written so you will also need to decorate that method with `@timer` as well.

Now when you run your sample application you should get timing information printed out for the `deposit` and `withdraw` methods:

```
calling  deposit on Account[123] - John, current account =
10.05overdraft limit: -100.0 with 23.45
returned from  deposit it took 8.009999999999268e-07 seconds
calling  withdraw on Account[123] - John, current account =
33.5overdraft limit: -100.0 with 12.33
returned from  withdraw it took 1.141999999999116e-06 seconds
```