2.4 Decorators

A decorator is one of the design patterns that describes the structure of related objects. Python is able to decorate functions, methods, and classes.

The decorator's operation is based on wrapping the original function with a new «decorating» function (or class), hence the name «decoration». This is done by passing the original function (i.e., the **decorated** function) as a parameter to the decorating function so that the **decorating** function can call the passed function. The decorating function returns a function that can be called later.

Of course, the **decorating** function does more, because it can take the parameters of the decorated function and perform additional actions and that make it a real decorating function.

The same principle is applied when we decorate classes. We'll talk about this a bit later.

So from now on, the term 'decorator' will be understood as a decorating class or a decorating function.

Decorators are used to perform operations before and after a call to a wrapped object or even to prevent its execution, depending on the circumstances. As a result, we can change the operation of the packaged object without directly modifying it.

Decorators are used in:

- the validation of arguments;
- the modification of arguments;
- the modification of returned objects;
- the measurement of execution time;
- message logging;
- thread synchronization;
- code refactorization;
- caching.

Function decorators

Let's analyze some examples before we get down to the next dose of theory.

So, let's create a function – `simple_hello()` is one of the simplest functions we could think of. We'll decorate it in a moment.

```
def simple_hello():
    print("Hello from simple function!")
```

```
def simple_decorator(function):
    print('We are about to call "{}"'.format(function.__name__))
    return function
```

The last lines are responsible for both method invocations:

```
decorated = simple_decorator(simple_hello)
decorated()
```

The whole code should look like the code presented in the right pane.

When you run the code, the result should be:

output

```
We are about to call "simple_hello"
Hello from simple function!
```

Now let's create another function, `simple_decorator()`, which is more interesting as it accepts an object as a parameter, displays a name attribute value of the parameter, and returns an accepted object.

We have created a simple decorator – a function which accepts another function as its only argument, prints some details, and returns a function or other callable object.

Well ... you could say ... we wrote so many lines of code just to print two other lines? Where is the simplicity or convenience in this approach?

Well ... we could say ... you should look at the following syntactic sugar:

As you can see, the definition of the `simple_hello()` function is literally decorated with '@simple_decorator' – isn't that a nice syntax?

This means that:

- operations are performed on object names;
- **this is the most important thing to remember**: the name of the simple_function object ceases to indicate the object representing our `simple_function()` and from that moment on it indicates the object returned by the decorator, the simple_decorator.

The implementation of the decorator pattern introduces this syntax, which appears to be very important and useful to developers. That is why decorators have gained great popularity and are widely used in Python code. It should be mentioned that decorators are very useful for refactoring or debugging the code.

```
def simple_decorator(function):
    print('We are about to call "{}"'.format(function.__name__))
    return function


@simple_decorator
def simple_hello():
    print("Hello from simple function!")


simple_hello()
```

output

```
We are about to call "simple_hello"
Hello from simple function!
```

Decorators should be universal

Consider a function that accepts arguments and should also be decorated. Decorators, which should be universal, must support any function, regardless of the number and type of arguments passed. In such a situation, we can use the *args and **kwargs concepts. We can also employ a closure technique to persist arguments.

The code presented in the right pane shows how the decorator can handle the arguments of the function being decorated.

```
def simple_decorator(own_function):

    def internal_wrapper(*args, **kwargs):
        print('"{}" was called with the following
arguments'.format(own_function.__name__))
        print('\t{}\n\t{}\n'.format(args, kwargs))
        own_function(*args, **kwargs)
        print('Decorator is still operating')

    return internal_wrapper


@simple_decorator
def combiner(*args, **kwargs):
    print("\tHello from the decorated function; received arguments:", args, kwargs)

combiner('a', 'b', exec='yes')
```

The output is:

output

```
"combiner" was called with the following arguments
    ('a', 'b')
    {'exec': 'yes'}

    Hello from the decorated function; received arguments: ('a', 'b') {'exec':
'yes'}
Decorator is still operating
```

Arguments passed to the decorated function are available to the decorator, so the decorator can print them. This is a simple example, as the arguments were just printed, but not processed further.

A nested function (internal_wrapper) could reference an object (own_function) in its enclosing scope thanks to the closure.

Decorators can accept their own attributes

In Python, we can create a decorator with arguments. Let's create a program in which the decorator will be more generic – we'll allow you to pass the packing material in the argument.

See the code presented in the right pane.

```
def warehouse_decorator(material):
    def wrapper(our_function):
        def internal_wrapper(*args):
            print('<strong>*</strong> Wrapping items from {} with
{}'.format(our_function.__name__, material))
            our_function(*args)
            print()
```

```
        return internal_wrapper
    return wrapper


@warehouse_decorator('kraft')
def pack_books(*args):
    print("We'll pack books:", args)


@warehouse_decorator('foil')
def pack_toys(*args):
    print("We'll pack toys:", args)


@warehouse_decorator('cardboard')
def pack_fruits(*args):
    print("We'll pack fruits:", args)


pack_books('Alice in Wonderland', 'Winnie the Pooh')
pack_toys('doll', 'car')
pack_fruits('plum', 'pear')
```

The `warehouse_decorator()` function created in this way has become much more flexible and universal than 'simple_decorator', because it can handle different materials.

Note that our decorator is enriched with one more function to make it able to handle arguments at all call levels.

The `pack_books` function will be executed as follows:

- the `warehouse_decorator('kraft')` function will return the wrapper function;
- the returned wrapper function will take the function it is supposed to decorate as an argument;
- the wrapper function will return the internal_wrapper function, which adds new functionality (material display) and runs the decorated function.

output

```
<strong>*</strong> Wrapping items from pack_books with kraft
We'll pack books: ('Alice in Wonderland', 'Winnie the Pooh')

<strong>*</strong> Wrapping items from pack_toys with foil
We'll pack toys: ('doll', 'car')

<strong>*</strong> Wrapping items from pack_fruits with cardboard
We'll pack fruits: ('plum', 'pear')
```

The biggest advantage of decorators is now clearly visible:

- we don’t have to change every 'pack' function to display the material being used;
- we just have to add a simple one liner in front of each function definition.

Decorator stacking

Python allows you to apply multiple decorators to a callable object (function, method or class).

The most important thing to remember is the order in which the decorators are listed in your code, because it determines the order of the executed decorators. When your function is decorated with multiple decorators:

```
@outer_decorator
@inner_decorator
def function():
    pass

abcd = subject_matter_function()
```

the call sequence will look like the following:

- the outer_decorator is called to call the inner_decorator, then the inner_decorator calls your function;
- when your function ends it execution, the inner_decorator takes over control, and after it finishes its execution, the outer_decorator is able to finish its job.

This routing mimics the classic stack concept.

The syntactic sugar presented above is the equivalent of the following nested calls:

```
subject_matter_function =
outer_decorator(inner_decorator(subject_matter_function()))
abcd = subject_matter_function()
```

It’s less readable than a simple call to your function, isn't it?

Another advantage becomes clear when you think about the number of modifications you should add to gain the same functionality, because you'd have to modify each call to your function.

In the right pane, you'll find a real example of stacked decorators.

```
def big_container(collective_material):
    def wrapper(our_function):
        def internal_wrapper(*args):
            our_function(*args)
            print('<strong>*</strong> The whole order would be packed with',
collective_material)
            print()
        return internal_wrapper
    return wrapper

def warehouse_decorator(material):
    def wrapper(our_function):
        def internal_wrapper(*args):
            our_function(*args)
            print('<strong>*</strong> Wrapping items from {} with
{}'.format(our_function.__name__, material))
        return internal_wrapper
    return wrapper

@big_container('plain cardboard')
@warehouse_decorator('bubble foil')
def pack_books(*args):
    print("We'll pack books:", args)

@big_container('colourful cardboard')
@warehouse_decorator('foil')
```

```
def pack_toys(*args):
    print("We'll pack toys:", args)

@big_container('strong cardboard')
@warehouse_decorator('cardboard')
def pack_fruits(*args):
    print("We'll pack fruits:", args)


pack_books('Alice in Wonderland', 'Winnie the Pooh')
pack_toys('doll', 'car')
pack_fruits('plum', 'pear')
```

We've created two decorators:

- `big_container` – which packs boxes into the collective material passed
- `warehouse_decorator` – which wraps single items into different materials.

We've also created functions for packaging different kinds of items, each decorated with two decorators.

This example demonstrates that packaging functions are called simply (and could be called many times in different places in your code) and every time those functions' behavior would be extended in a relevant way.

output

```
We'll pack books: ('Alice in Wonderland', 'Winnie the Pooh')
<strong>*</strong> Wrapping items from pack_books with bubble foil
<strong>*</strong> The whole order would be packed with plain cardboard

We'll pack toys: ('doll', 'car')
<strong>*</strong> Wrapping items from pack_toys with foil
<strong>*</strong> The whole order would be packed with colourful cardboard

We'll pack fruits: ('plum', 'pear')
<strong>*</strong> Wrapping items from pack_fruits with cardboard
<strong>*</strong> The whole order would be packed with strong cardboard
```

Lab

Objectives

- Improving the student's skills in creating decorators and operating with them.

Scenario

- Create a function decorator that prints a timestamp (in a form like year-month-day hour:minute:seconds, eg. 2019-11-05 08:33:22)
- Create a few ordinary functions that do some simple tasks, like adding or multiplying two numbers.
- Apply your decorator to those functions to ensure that the time of the function executions can be monitored.

Hint

To print the current time, you could use the following code:

```
# import module responsible for time processing
from datetime import datetime

# get current time using now() method
timestamp = datetime.now()

# convert timestamp to human-readable string, following passed pattern:
string_timestamp = timestamp.strftime('%Y-%m-%d %H:%M:%S')

print(string_timestamp)
```

resposta

```
# import module responsible for time processing
from datetime import datetime


def multiplica_decorator(funcio):
    def internal_wrapper(*args, **kwargs):
        timestamp = datetime.now()
        string_timestamp = timestamp.strftime('%Y-%m-%d %H:%M:%S')
        print(string_timestamp)
        funcio(*args,**kwargs)

    return internal_wrapper

@multiplica_decorator
def multiplica(a,b):
    print(a*b)

multiplica(2,3)
```

Decorating functions with classes

A decorator does not have to be a function. In Python, it could be a class that plays the role of a decorator as a function.

We can define a decorator as a class, and in order to do that, we have to use a `__call__` special class method. When a user needs to create an object that acts as a function (i.e., it is callable) then the function decorator needs to return an object that is callable, so the `__call__` special method will be very useful.

Our previous example code:

```
def simple_decorator(own_function):

    def internal_wrapper(*args, **kwargs):
        print('"{}" was called with the following
arguments'.format(own_function.__name__))
```

```
        print('\t{}\n\t{}\n'.format(args, kwargs))
        own_function(*args, **kwargs)
        print('Decorator is still operating')

    return internal_wrapper
```

could be transcribed to the code presented on the right. Run it to see the output and compare it to the output of the previously retrieved output.

```
class SimpleDecorator:
    def __init__(self, own_function):
        self.func = own_function

    def __call__(self, *args, **kwargs):
        print('"{}" was called with the following
arguments'.format(self.func.__name__))
        print('\t{}\n\t{}\n'.format(args, kwargs))
        self.func(*args, **kwargs)
        print('Decorator is still operating')


@SimpleDecorator
def combiner(*args, **kwargs):
    print("\tHello from the decorated function; received arguments:", args, kwargs)


combiner('a', 'b', exec='yes')
```

A short explanation of special methods:

- the `__init__` method assigns a decorated function reference to the self.attribute for later use;
- the `__call__` method, which is responsible for supporting a case when an object is called, calls a previously referenced function.

The advantage of this approach, when compared to decorators expressed with functions, is:

- classes bring all the subsidiarity they can offer, like inheritance and the ability to create dedicated supportive methods.

output

```
"combiner" was called with the following arguments
    ('a', 'b')
    {'exec': 'yes'}

    Hello from the decorated function; received arguments: ('a', 'b') {'exec':
'yes'}
Decorator is still operating
```

Decorators with arguments

Another previously discussed snippet showed that decorators can accept arguments:

```
def warehouse_decorator(material):
    def wrapper(our_function):
        def internal_wrapper(*args):
            print('* Wrapping items from {} with {}'.format(our_function.__name__,
material))
            our_function(*args)
            print()
        return internal_wrapper
    return wrapper
@warehouse_decorator('kraft')
def pack_books(*args):
    print("We'll pack books:", args)
```

And this code could be transcribed to a decorator expressed as a class, presented in the right pane.

```
class WarehouseDecorator:
    def __init__(self, material):
        self.material = material

    def __call__(self, own_function):
        def internal_wrapper(*args, **kwargs):
            print('<strong>*</strong> Wrapping items from {} with
{}'.format(own_function.__name__, self.material))
            own_function(*args, **kwargs)
            print()
        return internal_wrapper


@WarehouseDecorator('kraft')
def pack_books(*args):
    print("We'll pack books:", args)


@WarehouseDecorator('foil')
def pack_toys(*args):
    print("We'll pack toys:", args)


@WarehouseDecorator('cardboard')
def pack_fruits(*args):
    print("We'll pack fruits:", args)


pack_books('Alice in Wonderland', 'Winnie the Pooh')
pack_toys('doll', 'car')
pack_fruits('plum', 'pear')
```

When you pass arguments to the decorator, the decorator mechanism behaves quite differently than presented in example of decorator that does not accept arguments (previous slide):

- the reference to function to be decorated is passed to `__call__` method which is called only once during decoration process,
- the decorator arguments are passed to `__init__` method

Class decorators

Class decorators strongly refer to function decorators, because they use the same syntax and implement the same concepts.

Instead of wrapping individual methods with function decorators, class decorators are ways to manage classes or wrap special method calls into additional logic that manages or extends instances that are created.

If we consider syntax, class decorators appear just before the 'class' instructions that begin the class definition (similar to function decorators, they appear just before the function definitions).

The simplest use can be presented as follows:

```
@my_decorator
class MyClass:

obj = MyClass()
```

and it is adequate for the following snippet:

```
def my_decorator(A):
   ...

class MyClass:
   ...

MyClass = my_decorator(MyClass())

obj = MyClass()
```

Like function decorators, the new (decorated) class is available under the name 'MyClass' and is used to create an instance. The original class named 'MyClass' is no longer available in your name space. The callable object returned by the class decorator creates and returns a new instance of the original class, extended in some way.

Now we'll talk about a class decorated with a function that allows us to monitor the fact that some code gets access to the class object attributes. When you're debugging your code or optimizing it, you might be curious how many times the object attributes are accessed. In such a situation, a class decorator might be handy.

Let's create a simple class representing a car. Each object should own two attributes: mileage and VIN, and it should be possible to read the values of those attributes.

```
class Car:
    def __init__(self, VIN):
        self.mileage = 0
        self.VIN = VIN

car = Car('ABC123')
print('The mileage is', car.mileage)
print('The VIN is', car.VIN)
```

Now let's create a function that will decorate a class with a method that issues alerts whenever the 'mileage' attribute is read.

```
def object_counter(class_):
    class_.__getattr__orig = class_.__getattribute__
```

```
    def new_getattr(self, name):
        if name == 'mileage':
            print('We noticed that the mileage attribute was read')
        return class_.__getattr__orig(self, name)

    class_.__getattribute__ = new_getattr
    return class_
```

Look at the code in the editor. Let's analyze it:

- line 1: `def object_counter(class_)`: – this line defines a decorating function that accepts one parameter 'class_' (note the underscore)
- line 2: `class_.__getattr__orig = class_.__getattribute__ ''– the decorator makes a copy of the reference to the ''__getattribute__` special method. This method is responsible for returning the attribute values. The reference to this original method will be used in a modified method;
- line 4: `def new_getattr(self, name)`: – a definition of the method playing the role of the new `__getattribute__` method starts here. This method accepts an attribute name – it's a string;
- line 5: `if name == 'mileage'`: – in case some code asks for the 'mileage' attribute, the next line will be executed;
- line 6: `print('We noticed that the mileage attribute was read')` – a simple alert is issued;
- line 7: `return class_.__getattr__orig(self, name)` – the original method `__getattribute__` referenced by `class.__getattr__orig` is called. This ends the 'new_getattr' function definition;
- line 9: `class_.__getattribute__` = new_getattr – now the 'new_getattr' is defined, so it can now be referenced as the new '__getattribute__' method by a decorated class;
- line 10: `return class_` – every well behaved and developed decorator should return the decorated object – in our case it is a decorated class.

The last thing we should do is decorate the Car class:

```
@object_counter
class Car:
```

When you run the code, you can see that access to the 'mileage' attribute has been detected:

output

```
We noticed that the mileage attribute was read
The mileage is 0
The VIN is ABC123
```

Decorators – summary

A decorator is a very powerful and useful tool in Python, because it allows programmers to modify the behavior of a function, method, or class.

Decorators allow us to wrap another callable object in order to extend its behavior.

Decorators rely heavily on closures and *args and **kwargs.

Interesting note:

- the idea of decorators was described in two documents – PEP 318 and PEP 3129. Don't be discouraged

that the first PEP was prepared for Python 2, because what matters here is the idea, not the implementation in a specific Python.