EPAM Python Software Engineer Training

Lesson 2: Python decorators

Contents

1 Course

1.1 Decorators

In Python decorators are functions which decorate its argument at compile time (i.e. a function which takes something as an argument and returns an a decorated something while JIT-compiler prepares an expression). A syntax for a decorator usage is:

```
@decorator
def function():
    pass

@decorator_with_params('a', 'b')
@decorator
class A(object):
    pass
```

An above code is equivalent to the below one while it is more readable:

```
def function():
    pass

function = decorator(function)

class A(object):
    pass

A = decorator_with_params('a', 'b')(decorator(A))
```

In case if a decorator is to be used with parameters - it shall be a function which takes all necessary parameters as its arguments and returns a pure decorator function.

Decorators, especially parametrized ones, are a very powerful tool making it possible to use an aspect-oriented programming paradigm in Python. Uses cases for decorators are boundless, including transaction systems, context managers, logging systems, exception handlers, attribute markers, code generators etc.

2 Tasks

2.1 Transaction Manager

Write a simple decorator which implements a transaction handler. There shall be no real transaction performed, instead, a decorator shall print messages like `Transaction 1 for my_func started`, `Transaction 2 for my_func complete`, or `Transaction 3 for my_func cancelled` at transaction start, completion, or cancellation (in case of an error in a decorated function) accordingly.

Declare several functions doing different stuff that are decorated with this decorator.

> ***Hint***
>
> For simplicity it is allowed to use a global variable to auto-increment transaction numbers. Use a __name__ function attribute to distinguish transactions for different functions.

2.2 Hello World

Write a simple parametrized decorator so that the below code snippet prints `Hello\nWorld` to the standard output:

```
@decorator('World')
def function(arg):
    print arg

function('Hello')
```

2.3 Context Manager

Redo a transaction handler task in such a way that there is no necessity to wrap each action requiring a transaction into a separate function.

E.g. in the following code snippet there are 3 actions wrapped into a transaction, including nested transactions:

```
def my_func(a, b, c):
    with transaction('root'):
        print a
        with transaction('nested successful'):
            print b
        with transaction('nested with error'):
            print c
            raise Exception
```

> ***Hint***
>
> Use a *contextlib.context_manager* decorator to implement this task.

2.4 Method Generator

Write a decorator which generates a set of methods in a class given a template function and a dictionary of function names and their parameters.

E.g. the following two code snippets shall be equivalent:

```
def template(self, a, b, c):
    print self.x, a, b, c

method_table = {
    'test': dict(a=10, c=20),
    'other_test': dict(b=30),
}

@template_methods(template, method_table):
class A(object):
    x = 10
    pass
```

```
class A(object):
    x = 10

    def test(self, b):
        print self.x, 10, b, 30

    def other_test(self, a, c):
        print self.x, a, 20, c
```

> ***Hint***
>
> You might use a *funcutils.partial* to make this task easier.