Python Class Decorators

Summary: in this tutorial, you'll learn about Python class decorators. After the tutorial, you'll know how to define classes as decorators.

Introduction to the Python class decorators

So far you have learned how to use functions (https://www.pythontutorial.net/python-basics/python-functions/) to define decorators (https://www.pythontutorial.net/advanced-python/python-decorators/) .

For example, the following `star` function prints out a number of `*` characters before and after calling the decorated function:

```
def star(n):
    def decorate(fn):
        def wrapper(*args, **kwargs):
            print(n*'*')
            result = fn(*args, **kwargs)
            print(result)
            print(n*'*')
            return result
```

```
        return wrapper
    return decorate
```

The `star` is a decorator factory (https://www.pythontutorial.net/advanced-python/python-decorator-arguments/) that returns a decorator. It accepts an argument that specifies the number of `*` characters to display.

The following illustrates how to use the `star` decorator factory:

```
@star(5)
def add(a, b):
    return a + b


add(10, 20)
```

Output:

```
*****
30
*****
```

The `star()` decorator factory takes an argument and returns a callable. The callable takes an argument (`fn`) which is a function that will be decorated. Also, the callable can access the argument (`n`) passed to the decorator factory.

A class instance can be a callable when it implements the `__call__` (https://www.pythontutorial.net/python-built-in-functions/python-callable/) method. Therefore, you can make the `__call__` method as a decorator.

The following example rewrites the `star` decorator factory using a class instead:

```
class Star:
    def __init__(self, n):
        self.n = n
```

```
    def __call__(self, fn):
        def wrapper(*args, **kwargs):
            print(self.n*'*')
            result = fn(*args, **kwargs)
            print(result)
            print(self.n*'*')
            return result
        return wrapper
```

And you can use the `Star` class as a decorator like this:

```
@Star(5)
def add(a, b):
    return a + b
```

The `@Star(5)` returns an instance of the `Star` class. That instance is a callable, so you can do something like:

```
add = Star(5)(add)
```

So you can use callable classes to decorate functions.

Put it all together:

```
from functools import wraps


class Star:
    def __init__(self, n):
        self.n = n

    def __call__(self, fn):
```

```
        @wraps(fn)
        def wrapper(*args, **kwargs):
            print(self.n*'*')
            result = fn(*args, **kwargs)
            print(result)
            print(self.n*'*')
            return result
        return wrapper


@Star(5)
def add(a, b):
    return a + b


add(10, 20)
```

Summary

- Use callable classes as decorators by implementing the `__call__` method.
- Pass the decorator arguments to the `__init__` method.