Python Decorators

Summary: in this tutorial, you'll learn about Python decorators and how to develop your own decorators.

What is a decorator in Python

A decorator is a function (https://www.pythontutorial.net/python-basics/python-functions/) that takes another function as an argument and extends its behavior without changing the original function explicitly.

Let's take a simple example to understand the concept.

A simple Python decorator example

The following defines a `net_price` function:

```
def net_price(price, tax):
    """ calculate the net price from price and tax
    Arguments:
        price: the selling price
        tax: value added tax or sale tax
    Return
```

```
        the net price
    """
    return price * (1 + tax)
```

The `net_price` function calculates the net price from selling price and tax. It returns the `net_price` as a number.

Suppose that you need to format the net price using the USD currency. For example, `100` becomes `$100` . To do it, you can use a decorator.

By definition, a decorator is a function that takes a function as an argument:

```
def currency(fn):
    pass
```

And it returns another function:

```
def currency(fn):
    def wrapper(*args, **kwargs):
        fn(*args, **kwargs)

    return wrapper
```

The `currency` function returns the `wrapper` function. The `wrapper` function has the `*args` and `**kwargs` parameters.

These parameters allow you to call any `fn` function with any combination of positional and keyword-only arguments (https://www.pythontutorial.net/python-basics/python-kwargs/) .

In this example, the `wrapper` function essentially executes the `fn` function directly and doesn't change any behavior of the `fn` function.

In the `wrapper` function, you can call the `fn` function, get its result, and format the result as a currency string:

```
def currency(fn):
    def wrapper(*args, **kwargs):
        result = fn(*args, **kwargs)
        return f'${result}'
    return wrapper
```

The `currency` function is a decorator.

It accepts any function that returns a number and formats that number as a currency string.

To use the `currency` decorator, you need to pass the `net_price` function to it to get a new function and execute the new function as if it were the original function. For example:

```
net_price = currency(net_price)
print(net_price(100, 0.05))
```

Output:

```
$105.0
```

Python decorator definition

In general, a decorator is:

- A function that takes another function (original function) as an argument and returns another function (or closure (https://www.pythontutorial.net/advanced-python/python-closures/))
- The closure typically accepts any combination of positional and keyword-only arguments.
- The closure function calls the original function using the arguments passed to the closure and returns the result of the function.

The inner function is a closure because it references the `fn` argument from its enclosing scope or the decorator function.

The @ symbol

In the previous example, the currency is a decorator. And you can decorate the `net_price` function using the following syntax:

```
net_price = currency(net_price)
```

Generally, if `decorate` is a decorator function and you want to decorate another function `fn`, you can use this syntax:

```
fn = decorate(fn)
```

To make it more convenient, Python provides a shorter way like this:

```
@decorate
def fn():
    pass
```

For example, instead of using the following syntax:

```
net_price = currency(net_price)
```

... you can decorate the `net_price` function using the `@currency` as follows:

```
@currency
def net_price(price, tax):
    """ calculate the net price from price and tax
    Arguments:
        price: the selling price
        tax: value added tax or sale tax
    Return
        the net price
```

```
    """
    return price * (1 + tax)
```

Put it all together.

```
def currency(fn):
    def wrapper(*args, **kwargs):
        result = fn(*args, **kwargs)
        return f'${result}'
    return wrapper


@currency
def net_price(price, tax):
    """ calculate the net price from price and tax
    Arguments:
        price: the selling price
        tax: value added tax or sale tax
    Return
        the net price
    """
    return price * (1 + tax)


print(net_price(100, 0.05))
```

Introspecting decorated functions

When you decorate a function:

```
@decorate
def fn(*args,**kwargs):
    pass
```

It's equivalent:

```
fn = decorate(fn)
```

The `decorate` function returns a new function, which is the wrapper function.

If you use the built-in `help` function to show the documentation of the new function, you won't see the documentation of the original function. For example:

```
help(net_price)
```

Output:

```
wrapper(*args, **kwargs)

None
```

Also, if you check the name of the new function, Python will return the name of the inner function returned by the decorator:

```
print(net_price.__name__)
```

Output:

```
wrapper
```

So when you decorate a function, you'll lose the original function signature and documentation.

To fix this, you can use the `wraps` function from the `functools` standard module. In fact, the `wraps` function is also a decorator.

The following shows how to use the `wraps` decorator:

```
from functools import wraps


def currency(fn):
    @wraps(fn)
    def wrapper(*args, **kwargs):
        result = fn(*args, **kwargs)
        return f'${result}'
    return wrapper


@currency
def net_price(price, tax):
    """ calculate the net price from price and tax
    Arguments:
        price: the selling price
        tax: value added tax or sale tax
    Return
        the net price
    """
    return price * (1 + tax)


help(net_price)
print(net_price.__name__)
```

Output:

```
net_price(price, tax)
    calculate the net price from price and tax
    Arguments:
        price: the selling price
        tax: value added tax or sale tax
```

```
    Return
        the net price

net_price
```

Summary

- A decorator is a function that changes the behavior of another function without explicitly modifying it.
- Use the `@` symbol to decorate a function.
- Use the `wraps` function from the `functools` built-in module to retain the documentation and name of the original function.