Decorators

A decorator takes in a function, adds some functionality and returns it. In this tutorial, you will learn how you can create a decorator and why you should use it.

Decorators in Python

Python has an interesting feature called decorators to add functionality to an existing code.

This is also called metaprogramming because a part of the program tries to modify another part of the program at compile time.

What do you need to know about functions in Python?

Let's take a look at two aspects related to functions in Python.

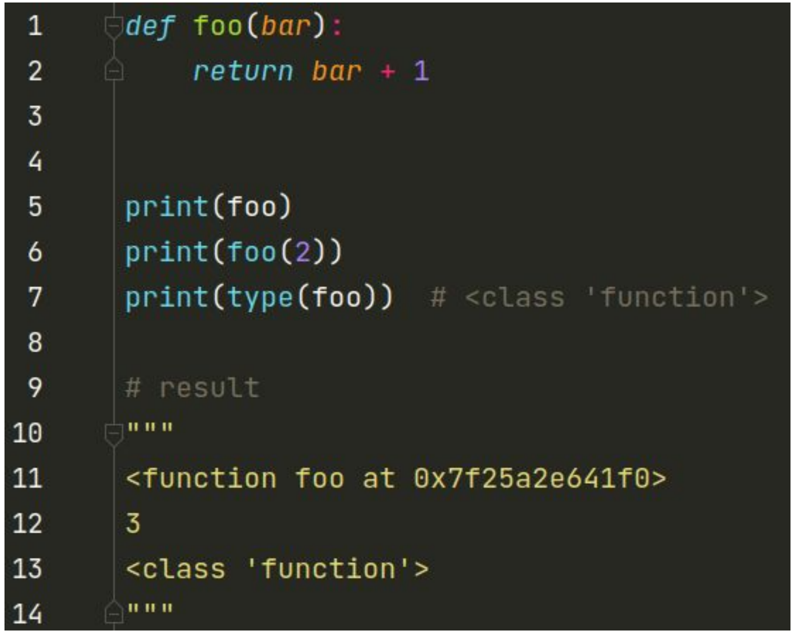
First of all, a function is a special kind of object, so it can be passed as an argument to other functions.

Second of all, inside functions, you can create other functions, call them and return them as a result via return. Let us dwell on these points in more detail.

Function object

Functions can be passed around, and used as arguments, just like any other value

See an example:

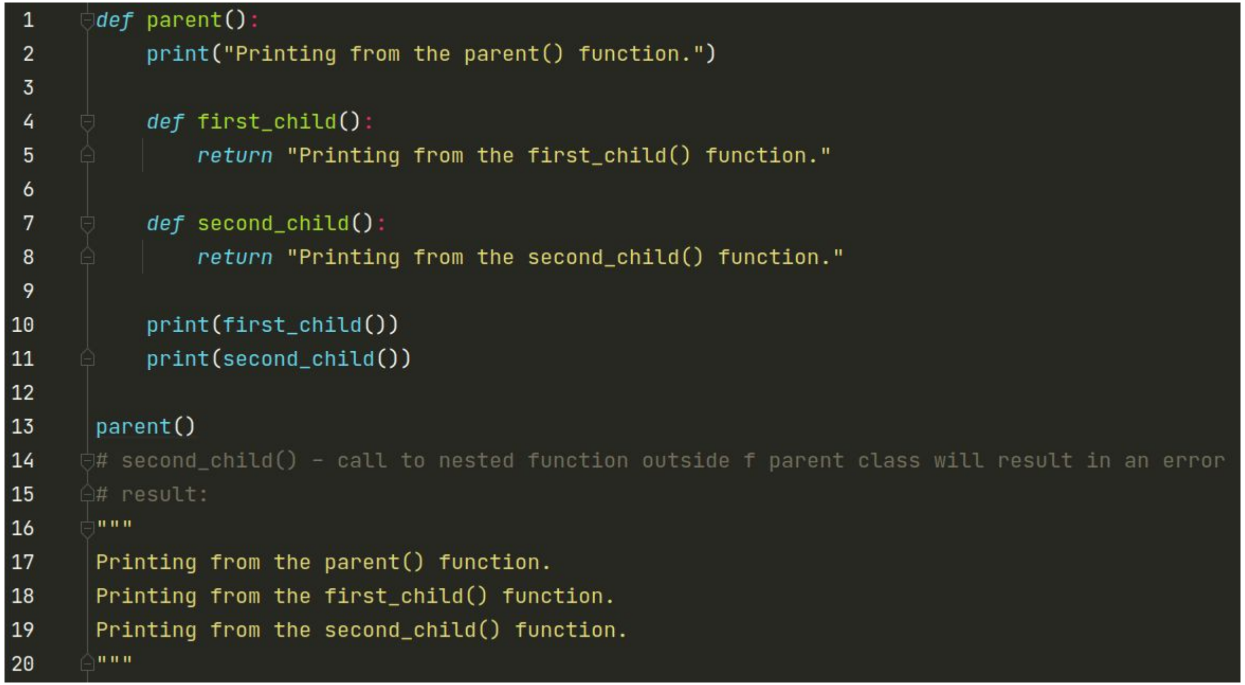


```python
def foo(bar):
    return bar + 1


print(foo)
print(foo(2))
print(type(foo))  # <class 'function'>

# result
"""
<function foo at 0x7f25a2e641f0>
3
<class 'function'>
"""
```

Inner functions

Inner functions, also known as nested functions, are functions that you define inside other functions. In Python, this kind of function has direct access to variables and names defined in the enclosing function. Inner functions have many uses, most notably as closure factories and decorator functions.

Inner functions



```python
def parent():
    print("Printing from the parent() function.")

    def first_child():
        return "Printing from the first_child() function."

    def second_child():
        return "Printing from the second_child() function."

    print(first_child())
    print(second_child())

parent()
# second_child() - call to nested function outside f parent class will result in an error
# result:
"""
Printing from the parent() function.
Printing from the first_child() function.
Printing from the second_child() function.
"""
```

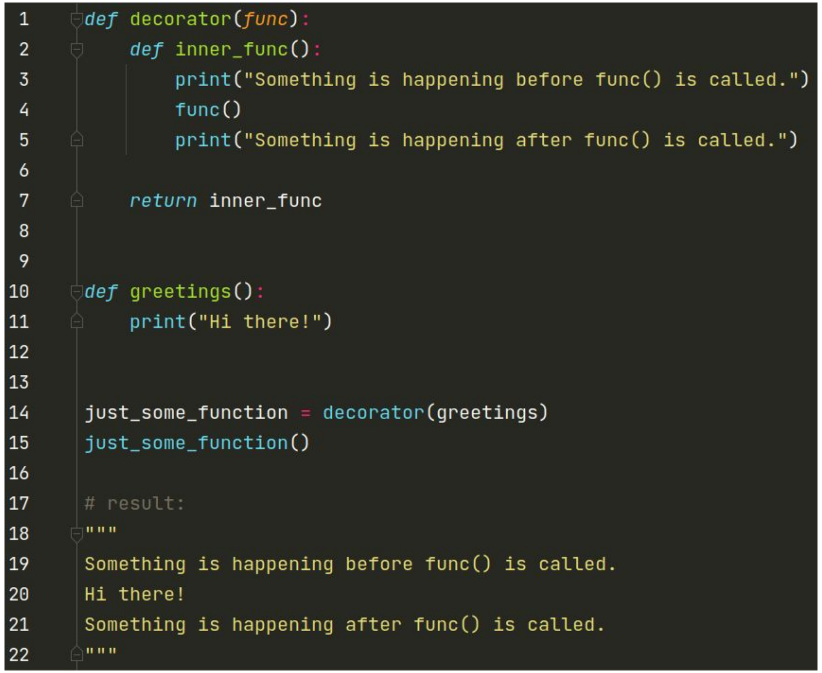
Decorating functions

Functions and methods are called callable as they can be called. In fact, any object which implements the special __call__() method is termed callable. So, in the most basic sense, a decorator is a callable that returns a callable.

Basically, a decorator takes in a function, adds some functionality and returns it. Decorators allow us to wrap another function in order to extend the behavior of the wrapped function, without permanently modifying it.

Decorating functions

We can see that the decorator function added some new functionality to the original function. This is similar to packing a gift. The decorator acts as a wrapper. The nature of the object that got decorated (actual gift inside) does not alter. But now, it looks pretty (since it got decorated).

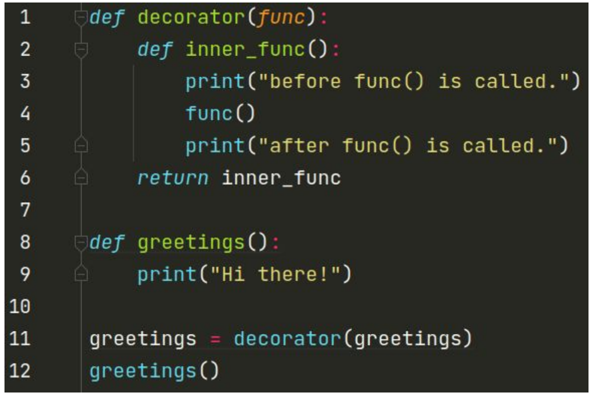


```python
def decorator(func):
    def inner_func():
        print("Something is happening before func() is called.")
        func()
        print("Something is happening after func() is called.")

    return inner_func


def greetings():
    print("Hi there!")


just_some_function = decorator(greetings)
just_some_function()

# result:
"""
Something is happening before func() is called.
Hi there!
Something is happening after func() is called.
"""
```

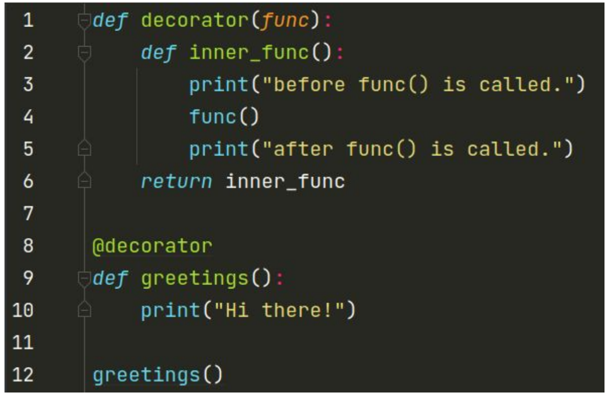
Pie syntax

Generally, we decorate a function and reassign it as `just_some_function = decorator(greetings)`

This is a common construct and for this reason, Python has a syntax to simplify this. We can use the @ symbol along with the name of the decorator function and place it above the definition of the function to be decorated.



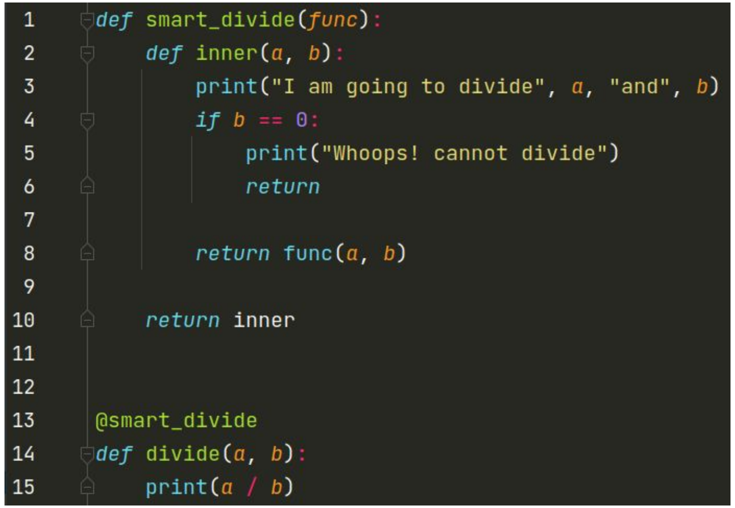
```python
def decorator(func):
    def inner_func():
        print("before func() is called.")
        func()
        print("after func() is called.")
    return inner_func

def greetings():
    print("Hi there!")

greetings = decorator(greetings)
greetings()
```



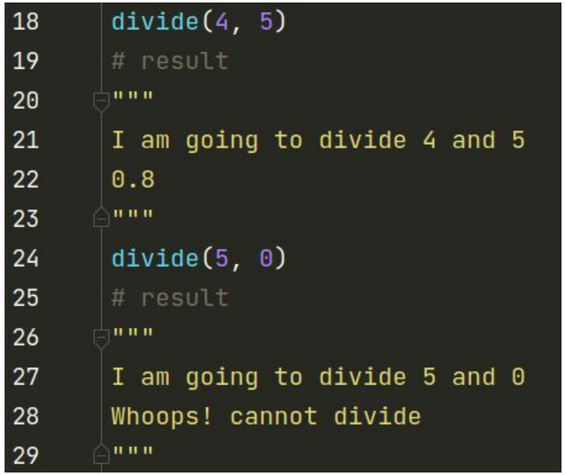
```python
def decorator(func):
    def inner_func():
        print("before func() is called.")
        func()
        print("after func() is called.")
    return inner_func

@decorator
def greetings():
    print("Hi there!")

greetings()
```

Decorating Functions with Parameters

Sometimes we might need to define a decorator that accepts arguments. We achieve this by passing the arguments to the wrapper function. The arguments will then be passed to the function that is being decorated at call time.



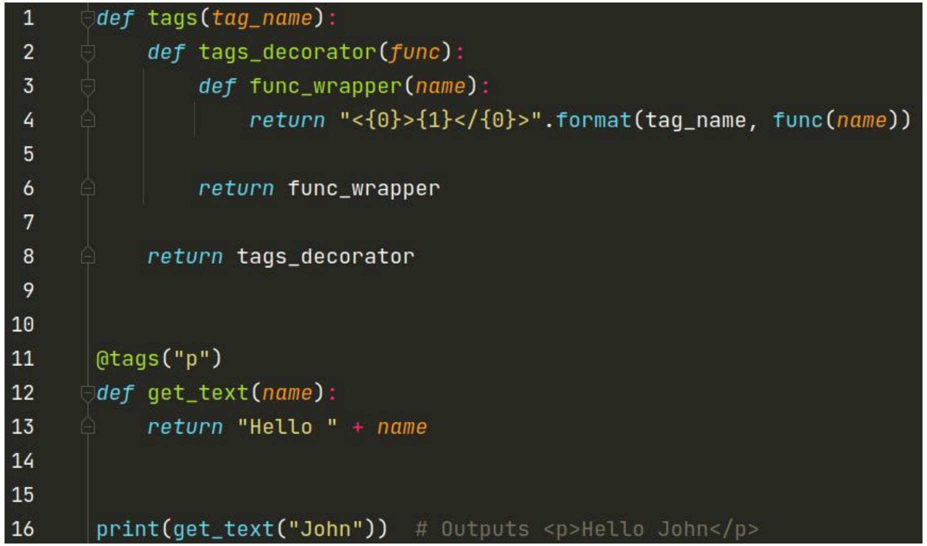
```python
def smart_divide(func):
    def inner(a, b):
        print("I am going to divide", a, "and", b)
        if b == 0:
            print("Whoops! cannot divide")
            return

        return func(a, b)

    return inner


@smart_divide
def divide(a, b):
    print(a / b)
```



```python
divide(4, 5)
# result
"""
I am going to divide 4 and 5
0.8
"""
divide(5, 0)
# result
"""
I am going to divide 5 and 0
Whoops! cannot divide
"""
```

Pass args to decorator

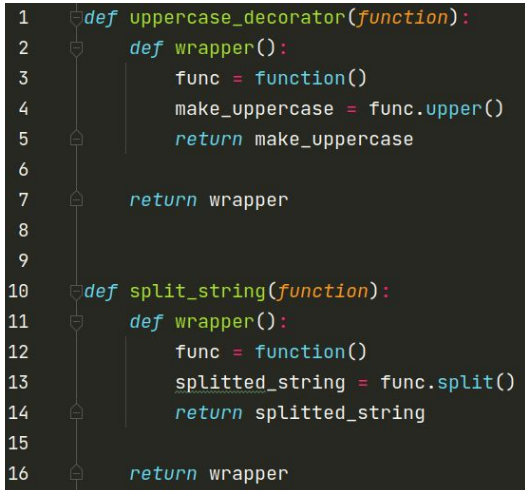
Decorators expect to receive a function as an argument, that is why we have to build a function that takes those extra arguments and generate our decorator on the fly.



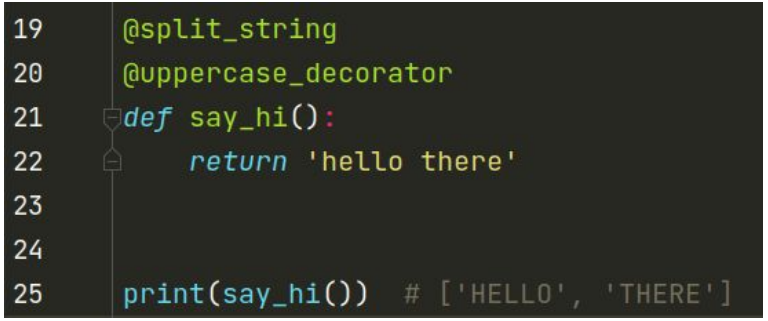
```python
def tags(tag_name):
    def tags_decorator(func):
        def func_wrapper(name):
            return "<{0}>{1}</{0}>".format(tag_name, func(name))

        return func_wrapper

    return tags_decorator


@tags("p")
def get_text(name):
    return "Hello " + name


print(get_text("John"))  # Outputs <p>Hello John</p>
```

Applying Multiple Decorators to a Single Function

We can use multiple decorators for a single function. However, the decorators will be applied in the order that we've called them. Below we'll define another decorator that splits the sentence into a list. We'll then apply the uppercase_decorator and split_string decorator to a single function. The application of decorators is from the bottom up.



```python
def uppercase_decorator(function):
    def wrapper():
        func = function()
        make_uppercase = func.upper()
        return make_uppercase

    return wrapper


def split_string(function):
    def wrapper():
        func = function()
        splitted_string = func.split()
        return splitted_string

    return wrapper
```



```python
@split_string
@uppercase_decorator
def say_hi():
    return 'hello there'


print(say_hi())  # ['HELLO', 'THERE']
```

Features of work with Decorators

- Decorators slow down the function call a little.
- You cannot "undecorate" a function. There are certain tricks to create a decorator that can be detached from a function, but this is bad practice. It would be more correct to remember that if the function is decorated, it will not be canceled.
- Decorators wrap functions, which can make debugging difficult.

Decorators use cases

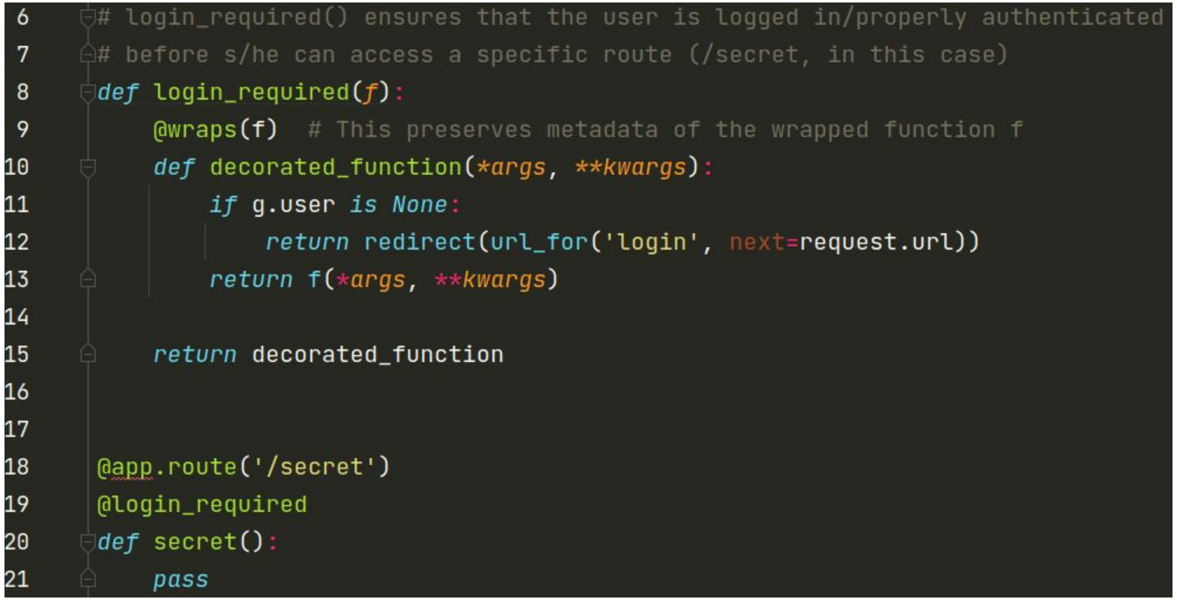
1. As a timer — to see how long it took to run the function

2. Rate limiting

3. Augment functions with code that logs calls made to them

4. Check the types of passed arguments during debugging

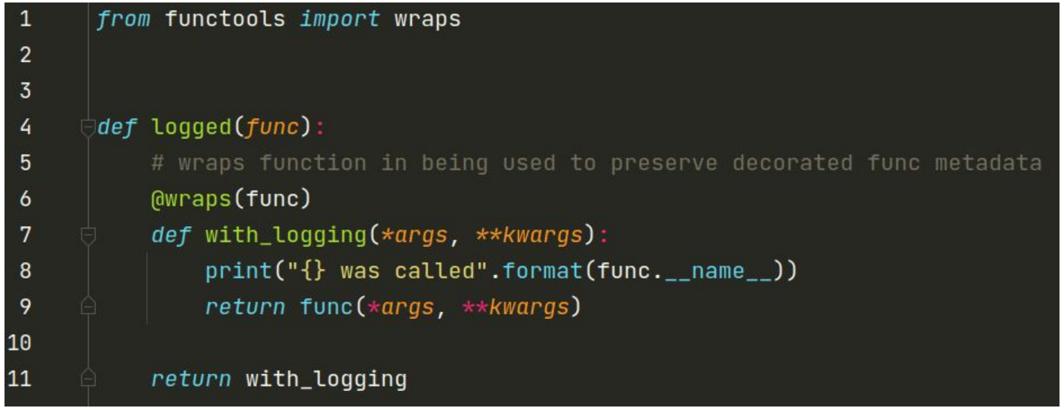
5. Some authentication logic before entering function



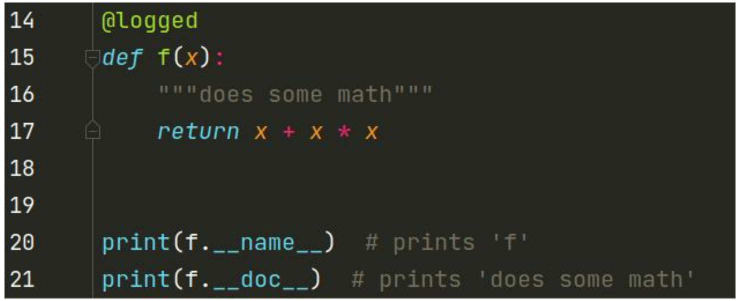
```python
# login_required() ensures that the user is logged in/properly authenticated
# before s/he can access a specific route (/secret, in this case)
def login_required(f):
    @wraps(f)  # This preserves metadata of the wrapped function f
    def decorated_function(*args, **kwargs):
        if g.user is None:
            return redirect(url_for('login', next=request.url))
        return f(*args, **kwargs)

    return decorated_function


@app.route('/secret')
@login_required
def secret():
    pass
```

wraps function

wraps() is being used as a decorator in order to preserve decorated function metadata



```python
from functools import wraps


def logged(func):
    # wraps function in being used to preserve decorated func metadata
    @wraps(func)
    def with_logging(*args, **kwargs):
        print("{} was called".format(func.__name__))
        return func(*args, **kwargs)

    return with_logging
```



```python
@logged
def f(x):
    """does some math"""
    return x + x * x


print(f.__name__)  # prints 'f'
print(f.__doc__)  # prints 'does some math'
```

Built-in decorators

Python contains several built-in decorators. Of all these decorators, the most important trinity are:

- @classmethod
- @staticmethod
- @property

Built-in decorators

The **@classmethod** decorator is primarily used as an interleaved constructor or an initialization helper.

The **@staticmethod** decorator is just a function within a class. You can call both of them with or without class initialization. This is usually used when you have a function that you believe has a relationship to a class. For the most part, it's about style choices.

The **@property** decorator which is helpful in defining the properties effortlessly without manually calling the inbuilt function property(). Which is used to return the property attributes of a class from the stated getter, setter and deleter as parameters.

References

1. https://www.programiz.com/python-programming/decorator
2. https://www.datacamp.com/community/tutorials/decorators-python
3. https://www.geeksforgeeks.org/decorators-in-python/
4. https://wiki.python.org/moin/PythonDecorators
5. https://tproger.ru/translations/demystifying-decorators-in-python/
6. https://habr.com/ru/post/141411/
7.