



# Decorators in Python

# Introduction to decorators - First Class Objects

In functional programming, you work almost entirely with pure functions that don't have side effects.

While not a purely functional language, Python supports many functional programming concepts, **including treating functions as first-class objects.**

This means that *functions can be passed around and used as arguments*, just like any other object like `str`, `int`, `float`, `list`, and so on.

# Introduction to decorators - First Class Objects

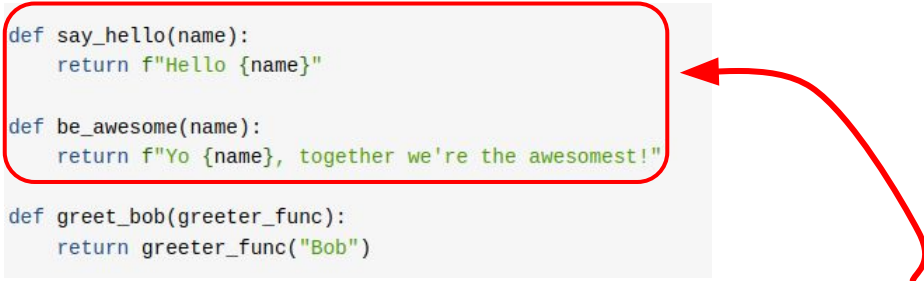
Consider the following three functions:

```
def say_hello(name):  
    return f"Hello {name}"  
  
def be_awesome(name):  
    return f"Yo {name}, together we're the awesomest!"  
  
def greet_bob(greeter_func):  
    return greeter_func("Bob")
```

# Introduction to decorators - First Class Objects

Consider the following three functions:

```
def say_hello(name):  
    return f"Hello {name}"  
  
def be_awesome(name):  
    return f"Yo {name}, together we're the awesomest!"  
  
def greet_bob(greeter_func):  
    return greeter_func("Bob")
```



**Here, `say_hello()` and `be_awesome()` are regular functions that expect a name given as a string.**

# Introduction to decorators - First Class Objects

Consider the following three functions:

```
def say_hello(name):  
    return f"Hello {name}"  
  
def be_awesome(name):  
    return f"Yo {name}, together we're the awesomest!"  
  
def greet_bob(greeter_func):  
    return greeter_func("Bob")
```

The `greet_bob()` function, however, expects a function as its argument.

You can, for example, pass it the `say_hello()` or the `be_awesome()` function.

# Introduction to decorators - First Class Objects

The `say_hello` function is named without parentheses.

**This means that only a reference to the function is passed.**

**The function isn't executed.**

```
def say_hello(name):  
    return f"Hello {name}"  
  
def be_awesome(name):  
    return f"Yo {name}, together we're the awesomest!"  
  
def greet_bob(greeter_func):  
    return greeter_func("Bob")
```

```
>>> greet_bob(say_hello)  
'Hello Bob'  
  
>>> greet_bob(be_awesome)  
'Yo Bob, together we're the awesomest!'
```

# Introduction to decorators - First Class Objects


The `say_hello` function is named without parentheses.

**This means that only a reference to the function is passed.**

**The function isn't executed.**

The `greet_bob()` function, on the other hand, is written with parentheses, so it will be called as usual.

```
def say_hello(name):  
    return f"Hello {name}"  
  
def be_awesome(name):  
    return f"Yo {name}, together we're the awesomest!"  
  
def greet_bob(greeter_func):  
    return greeter_func("Bob")
```



```
>>> greet_bob(say_hello)  
'Hello Bob'  
  
>>> greet_bob(be_awesome)  
'Yo Bob, together we're the awesomest!'
```

# Introduction to decorators - Inner Functions

## Inner Functions

It's possible to define functions *inside other functions*.

Such functions are called inner functions.

Here's an example of a function with two inner functions

```
def parent():  
    print("Printing from parent()")  
  
    def first_child():  
        print("Printing from first_child()")  
  
    def second_child():  
        print("Printing from second_child()")  
  
    second_child()  
    first_child()
```



# Introduction to decorators - Inner Functions

## Inner Functions

It's possible to define functions *inside other functions*.

Such functions are called inner functions.

Here's an example of a function with two inner functions

Outer function



```
def parent():  
    print("Printing from parent()")  
  
    def first_child():  
        print("Printing from first_child()")  
  
    def second_child():  
        print("Printing from second_child()")  
  
    second_child()  
    first_child()
```

# Introduction to decorators - Inner Functions

## Inner Functions

It's possible to define functions *inside other functions*.

Such functions are called inner functions.

Here's an example of a function with two inner functions

Inner function 1



```
def parent():  
    print("Printing from parent()")  
  
    def first_child():  
        print("Printing from first_child()")  
  
    def second_child():  
        print("Printing from second_child()")  
  
    second_child()  
    first_child()
```

# Introduction to decorators - Inner Functions

## Inner Functions

It's possible to define functions *inside other functions*.

Such functions are called inner functions.

Here's an example of a function with two inner functions

Inner function 2




```
def parent():  
    print("Printing from parent()")  
  
    def first_child():  
        print("Printing from first_child()")  
  
    def second_child():  
        print("Printing from second_child()")  
  
    second_child()  
    first_child()
```

# Introduction to decorators - Functions as Return Values

Python also allows you to return functions from functions.

In the following example, you rewrite `parent()` to return one of the inner functions




```
def parent(num):  
    def first_child():  
        return "Hi, I'm Elias"  
  
    def second_child():  
        return "Call me Ester"  
  
    if num == 1:  
        return first_child  
    else:  
        return second_child
```

# Introduction to decorators - Functions as Return Values

Python also allows you to return functions from functions.

In the following example, you rewrite `parent()` to return one of the inner functions

First function



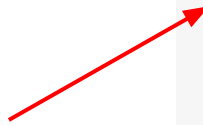
```
def parent(num):  
    def first_child():  
        return "Hi, I'm Elias"  
  
    def second_child():  
        return "Call me Ester"  
  
    if num == 1:  
        return first_child  
    else:  
        return second_child
```

# Introduction to decorators - Functions as Return Values

Python also allows you to return functions from functions.

In the following example, you rewrite `parent()` to return one of the inner functions

Second function



```
def parent(num):  
    def first_child():  
        return "Hi, I'm Elias"  
  
    def second_child():  
        return "Call me Ester"  
  
    if num == 1:  
        return first_child  
    else:  
        return second_child
```

# Introduction to decorators - Functions as Return Values

Note that you're returning `first_child` without the parentheses.


Recall that this means that you're *returning a reference to the function* `first_child`.

```
def parent(num):  
    def first_child():  
        return "Hi, I'm Elias"  
  
    def second_child():  
        return "Call me Ester"  
  
    if num == 1:  
        return first_child  
    else:  
        return second_child
```

```
>>> first = parent(1)  
>>> second = parent(2)  
  
>>> first  
<function parent.<locals>.first_child at 0x7f599f1e2e18>  
  
>>> second  
<function parent.<locals>.second_child at 0x7f599dad5268>
```

# Introduction to decorators - Simple Decorators

Here, you've defined two regular functions, `decorator()` and `say_whee()`, and one inner `wrapper()` function.




```
def decorator(func):  
    def wrapper():  
        print("Something is happening before the function is called.")  
        func()  
        print("Something is happening after the function is called.")  
    return wrapper  
  
def say_whee():  
    print("Whee!")  
  
say_whee = decorator(say_whee)
```



# Introduction to decorators - Simple Decorators


Here, you've defined two regular functions, `decorator()` and `say_whee()`, and one inner `wrapper()` function.



```
def decorator(func):  
    def wrapper():  
        print("Something is happening before the function is called.")  
        func()  
        print("Something is happening after the function is called.")  
    return wrapper  
  
def say_whee():  
    print("Whee!")  
  
say_whee = decorator(say_whee)
```

# Introduction to decorators - Simple Decorators

Here, you've defined two regular functions, `decorator()` and `say_whee()`, and one inner `wrapper()` function.

The so-called **decoration** happens at the following line 

```
def decorator(func):  
    def wrapper():  
        print("Something is happening before the function is called.")  
        func()  
        print("Something is happening after the function is called.")  
    return wrapper  
  
def say_whee():  
    print("Whee!")  
  
say_whee = decorator(say_whee)
```

# Introduction to decorators - Simple Decorators

Here, you've defined two regular functions, `decorator()` and `say_whee()`, and one inner `wrapper()` function.

The so-called **decoration** happens at the following line

**Put simply, a decorator is a function that wraps a function, modifying its behavior.**

```
def decorator(func):  
    def wrapper():  
        print("Something is happening before the function is called.")  
        func()  
        print("Something is happening after the function is called.")  
    return wrapper  
  
def say_whee():  
    print("Whee!")  
  
say_whee = decorator(say_whee)
```