



Decorators  $\therefore$  Miki Tebeka [miki@353solutions.com](mailto:miki@353solutions.com)\*

## Contents

Understanding Functions	1
Decorators	5
TL;DR	8

## Understanding Functions

Before we start talking about decorators, let's talk about Python's function. In Python function are *first class objects*. This means that functions are like any other object, we can create new ones, pass them around as function parameters, query attributes ...

Say we have the following code

```
def add(x, y):  
    """Addition"""  
    return x + y
```

What Python does when we do a `def` is to compile the function code and create a function object. Then it assigns the name `add` to point to this object.

---

\*<mailto:miki@353solutions.com>

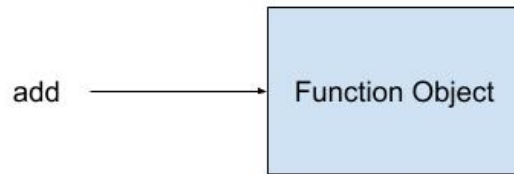


Figure 1:

We can access attributes

```
>>> add.__name__  
'add'
```

We can assign another variable to this function object.

```
plug = add
```

And how we have both `add` and `plus` pointing to the same function object.

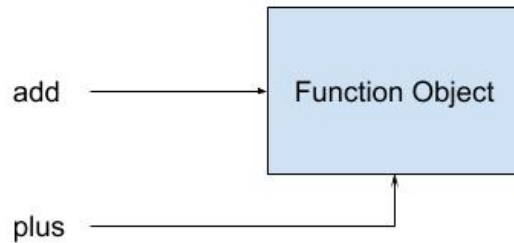


Figure 2:

We can even create functions on the fly

```
def make_adder(n):  
    """Returns a functions that add n to the argument"""  
    def adder(val):  
        return val + n  
  
    return adder
```

The inner `adder` function remember the value of `n` at the time of it's creation. This is called closure<sup>1</sup>.

When we do

```
add7 = make_adder(7)
```

When the function is called, we create a new function object that remember that `n` is 7 and have the local variable `adder` point to it

---

<sup>1</sup>[https://en.wikipedia.org/wiki/Closure\\_\(computer\\_programming\)](https://en.wikipedia.org/wiki/Closure_(computer_programming))

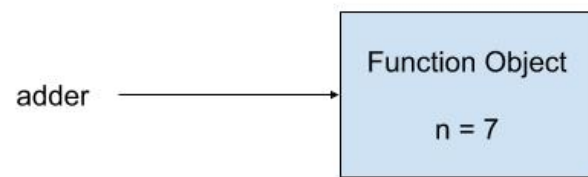


Figure 3:

`make_adder` return value is this function object and now `adder` points to it.

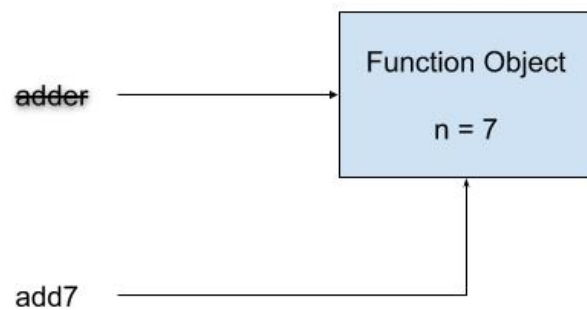


Figure 4:

And now we can use it

```
>>> add7(10)
17
```

## Decorators

A decorator is a function<sup>2</sup> that gets a function as a parameter and returns a function.

Confusing? Let's see an example. Say you have a menu and you'd like to register function to menu items, you might do the following:

```
menu = {}
```

```
def register(fn):
    """Register function to menu"""
    menu[fn.__name__] = fn
    return fn
```

---

<sup>2</sup>Most of the time :)

```

def copy():
    """Copy text"""
    print('>>> COPY <<<')

def paste():
    """Paste text"""
    print('>>> PASTE <<<')

# Register menu functions
register(copy)
register(paste)

def ui_loop():
    """Runs the UI loop"""
    while True:
        print('=== Menu ===')
        print('\n'.join(sorted(menu)))
        name = input('# ').strip() # Use raw_input in Python 2
        if name == 'quit':
            break
        fn = menu.get(name)
        if not fn:
            print('ERROR: Unknown action - {}'.format(name))
            continue
        fn()

```

We can use decorators to make this code nicer

```

menu = {}

def register(fn):
    """Register function to menu"""
    menu[fn.__name__] = fn
    return fn

@register
def copy():
    """Copy text"""
    print('>>> COPY <<<')

```

```

@register
def paste():
    """Paste text"""
    print('>>> PASTE <<<')

def ui_loop():
    """Runs the UI loop"""
    while True:
        print('=== Menu ===')
        print('\n'.join(sorted(menu)))
        name = input('# ').strip() # Use raw_input in Python 2
        if name == 'quit':
            break
        fn = menu.get(name)
        if not fn:
            print('ERROR: Unknown action - {}'.format(name))
            continue
        fn()

```

This code has the same functionality. Every time you see

```

@some_decorator
def some_function(a, b):
    ....

```

It's like writing

```

def some_function(a, b):
    ...

some_function = some_decorator(some_function)

```

Registration is one common use for decorators, the other one is adding functionality to functions without changing the source code. Let's say we'd like to measure how long a function is running. Instead of adding the timing code inside the function, we'll write a decorator that will start a time, call the function and at the end will print how long it took<sup>3</sup>.

```

from time import monotonic, sleep
from functools import wraps

```

```

def timed(fn):
    """Timing decorator"""
    @wraps(fn)

```

---

<sup>3</sup>In real life we'll probably send a metric to a system like InfluxDB

```

def wrapper(*args, **kw):
    start = monotonic()
    try:
        return fn(*args, **kw)
    finally:
        duration = monotonic() - start
        print('{} took {:.2f}sec'.format(fn.__name__, duration))
return wrapper

@timed
def add(a, b):
    """Does addition"""
    sleep(a/10.0) # Simulate work
    return a + b

```

Notes:

- The `wrapper` decorator copies the function name and documentation string to the newly created function. This means the `help(add)` will show the help for `add` and not for the inner `wrapper` function that `add` is pointing to.
- We use `wrapper(*args, **kw)` so we can apply the decorator on different functions with different signatures<sup>4</sup>.
- We use `time.monotonic` to measure time, it is more accurate than `time.time`

## TL;DR

- Functions in Python are “first class objects”. You can assign them to variables, pass them as parameters to other functions and create new ones during program run time
- A “closure” is the environment where a function was created. Functions in Python know their closure.
- A decorator is a function that get a function as argument and returns a function
- When you see ~ python @some\_decorator def some\_function(a, b): .... ~  
It means

```

def some_function(a, b):
    ...

```

---

<sup>4</sup>A function **signature** is the number of arguments it accepts



```
some_function = some_decorator(some_function)
```

- Commons uses for decorators are registration and adding functionality to functions without changing their source code