Mastering Python Decorators

Welcome

I'm your host, Aaron Maxwell.

- aaron@powerfulpython.com
- On twitter:
    - @powerfulpython (professional)
    - @redsymbol (personal)

Our focus in this class: **Writing Python Decorators**

What is a decorator?

A decorator is a way to **add behavior** around a function or method.

```
@somedecorator
def some_function(x, y, z):
    # ...
```

Once it is written, using a decorator is trivially easy.

Many successful libraries use decorators extensively: Flask, Django, Twisted, pytest, nose, SQLAlchemy, and more.

WHY Write Decorators?

Using decorators is trivial. **Writing** decorators is very challenging. But today, you'll learn how to do it!

What decorators let you do:

- Add rich features to groups of functions and classes
- Untangle distinct, frustratingly intertwined concerns in your code
- Encapsulate code reuse patterns not otherwise possible
- Effectively extend Python syntax in certain limited but powerful ways
- Build easily reusable frameworks

A Code Problem ...

Imagine you're making requests to a remote todo-list API, which speaks JSON over HTTP:

```
import requests
def make_api_request():
    requests.get(API_URL + '/items')

resp = make_api_request()
assert resp.status_code == 200, resp.status_code
```

But it's flakey, and returns a 500 error code 1% of the time.

Retry

Automatic retry scaffolding:

```
# Retry this many times before giving up.
MAX_TRIES = 3
tries = 0
resp = None
while True:
    resp = make_api_request()
    if resp.status_code == 500 and tries < MAX_TRIES:
        tries += 1
        continue
    break
todo_items = resp.json()
for item in todo_items:
    # Do something, like print it out, etc.
```

You have dozens of functions and methods like `make_api_request()`. How do you reusably capture this pattern?

Broad Itinerary

- Generic function arguments and argument unpacking
- Working with Function Objects
- Basic decorator structure and patterns
- Advanced decorators: class-based, accepting arguments, etc.

And throughout... Lots of coding!

How We'll Proceed

Bookmark this URL:

https://goo.gl/MPwpdY

First thing, download the courseware (link in document).

What's included:

- PDF course book - MasteringPythonDecorators.pdf
- Slides - SlidesForMasteringPythonDecorators.pdf
- Labs (i.e., programming exercises - more on that later)

Give you a break every hour-ish (10 minutes).

Python versions

Most code I show you will run in both Python 2 and 3.

Where it's different, I'll code in Python 3, and point out the differences. (There won't be many.)

You can do the programming exercises in either 3.5+, or 2.7.

What makes perfect?

Practice, practice, practice.

- Practice syntax (typing things in)
- Practice programming (higher-level labs)

I expect you to do your part!

You **exponentially** get out of this what you put into it.

Running the labs

Labs are the main programming exercises. You are given a failing automated test; your job is to write Python code to make it pass.

Simply run it as a Python program, any way you like. (For example, "python3 helloworld.py")

Run unmodified first, so you can see the failure report.

When done, put your sign down (so I know you're done).

Then: Move on to the extra credit.

Lab: helloworld.py

Let's do our first lab now: 'helloworld.py`

- In `labs/py3` for Python 3.x, or `labs/py2` for 2.7

When you finish:

- Knock your sign down, so I know you're done.
- Proceed to `helloworld_extra.py`

Getting the most

We'll take some class time for each lab. You may not finish, but it's **critically important** that you at least start when I tell you to.

After we're done for the day, find time to finish all the main labs before tomorrow.

Solutions are provided. Use them wisely, not foolishly:

- After you get the lab passing, compare your code to the official solution.
- Other than that, don't look at them if you can avoid it.
- The more work you can do on your own, the more you will learn. Peek at the solution to get a hint when you really need it.

Optional (**only** for future master Pythonistas): Do all the extra labs as well, as soon as you can manage.

Part 1

Variable Arguments & Argument Unpacking

The basics

I'll assume you know how to define functions with default arguments.

```
>>> def foo(a, b, x=3, y=2):
...     return (a+b)/(x+y)
...
>>> foo(5, 0)
1.0
>>> foo(10, 2, y=3)
2.0
>>> foo(b=4, x=3, a=1)
1.0
```

Accepting 0 or more arguments

Sometimes you want to define a function that can take any number of arguments. Python's syntax for doing that looks like this:

```
# Note the asterisk. That's the magic part
def takes_any_args(*args):
    print("Type of args: " + str(type(args)))
    print("Value of args: " + str(args))
```

```
>>> takes_any_args(1)
Type of args: <class 'tuple'>
Value of args: (1,)
>>> takes_any_args("x", "y", "z")
Type of args: <class 'tuple'>
Value of args: ('x', 'y', 'z')
>>> takes_any_args()
Type of args: <class 'tuple'>
Value of args: ()
```

Stored in tuple

Within the function, `args` is a tuple:

```
def takes_any_args(*args):
    print("Type of args: " + str(type(args)))
    print("Value of args: " + str(args))
```

```
>>> takes_any_args(5, 4, 3, 2, 1)
Type of args: <class 'tuple'>
Value of args: (5, 4, 3, 2, 1)
>>> takes_any_args(["first", "list"], ["another","list"])
Type of args: <class 'tuple'>
Value of args: (['first', 'list'], ['another', 'list'])
```

Single Argument vs. *args

```
>>> def takes_any_args(*args):
...     print("Type of args: " + str(type(args)))
...     print("Value of args: " + str(args))
...
>>> def takes_a_list(items):
...     print("Type of items: " + str(type(items)))
...     print("Value of items: " + str(items))
...
>>> data = ["x", "y", "z"]
>>> takes_a_list(data)
Type of items: <class 'list'>
Value of items: ['x', 'y', 'z']
>>> takes_any_args(data)
Type of args: <class 'tuple'>
Value of args: (['x', 'y', 'z'],)
```

*args

By convention, the tuple argument's default name is `args`. But it doesn't have to be.

```
def read_files(*paths):
    data = ""
    for path in paths:
        with open(path) as handle:
            data += handle.read()
    return data

# ch1.txt has text of Chapter 1; ch2.txt for Ch. 2, etc.
story = read_files("ch1.txt", "ch2.txt", "ch3.txt", "ch4.txt")
```

Quick exercise

Open a file named `varargs1.py` and type in the following:

```
def print_args(*args):
    for arg in args:
        print(arg)

print_args("red", "blue", "green")
```

You should see this output, one color per line:

```
red
blue
green
```

Extra credit: Can you find a way to call `print_args`, with different arguments, that triggers an error?

Variable Keyword Arguments

Keyword arguments can't be captured by the `*args` idiom:

```
def print_args(*args):
    for arg in args:
        print(arg)
```

```
>>> print_args(a=4, b=7)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: print_args() got an unexpected keyword argument 'a'
```

What do we do?

kwargs

For keyword arguments, use `**` syntax:

```
def print_kwargs(**kwargs):
    for key, value in kwargs.items():
        print("{} -> {}".format(key, value))
```

`kwargs` is a **dictionary**.

```
>>> print_kwargs(hero="Homer", antihero="Bart", genius="Lisa")
hero -> Homer
genius -> Lisa
antihero -> Bart
```

Notice this is normal Python syntax for calling a function. Has nothing to do with `**kwargs`.

Combine them

Combine them together:

```
def print_all(*args, **kwargs):
    for arg in args:
        print(arg)
    for key, value in kwargs.items():
        print("{} -> {}".format(key, value))
```

A defined function can use either `*args`, or `**kwargs`, or both.

Some notes...

- `args` and `kwargs` are conventional names. Use them unless it's more readable to do something different.
- Always be clear on the types:
  - `args` is a tuple, in the same order as passed in
  - `kwargs` is a dictionary, unordered

Quick exercise

Create a file named `varargs2.py` and type in the following (on the left). When run, it should print the output on the right.

varargs2.py source

```
def print_all(*args, **kwargs):
    for arg in args:
        print(arg)
    for key, value in kwargs.items():
        print("{} -> {}".format(key, value))

print_all("dog", "cat")
print_all(x=7, z=3)
print_all("red", "green", x=7, z=3)
```

program output

```
dog
cat
z -> 3
x -> 7
red
green
z -> 3
x -> 7
```

EXTRA CREDIT: How can you write a function taking from 0 to 4 arguments (but no more)? How about 0 to 20? What are the different approaches and trade-offs?

Better: Arg Unpacking

Python provides a better way.

```
>>> def normal_function(a, b, c):
...     print("a: {} b: {} c: {}".format(a,b,c))
...
>>> numbers = (7, 5, 3)
>>> normal_function(*numbers)
a: 7 b: 5 c: 3
>>> # Exactly equivalent to:
... normal_function(numbers[0], numbers[1], numbers[2])
a: 7 b: 5 c: 3
```

Notice `normal_function` is just a regular function! This is called **argument unpacking**.

Argument unpacking

Given these:

```
one_args = [ 42 ]
two_args = (7, 10)
three_args = [1, 2, 3]

def f(n): return n / 2
def g(a, b): return a + b
def h(x, y, z): return x * y * z
```

The following pairs are all equivalent:

```
f(one_args[0])
f(*one_args)

g(two_args[0], two_args[1])
g(*two_args)

h(three_args[0], three_args[1], three_args[2])
h(*three_args)
```

Keyword Unpacking

Just like with `*args`, double-star works the other way too. We can take a regular function, and pass it a dictionary using two asterisks:

```
>>> def normal_function(a, b, c):
...     print("a: {} b: {} c: {}".format(a,b,c))
...
>>> numbers = {"a": 7, "b": 5, "c": 3}
>>> normal_function(**numbers)
a: 7 b: 5 c: 3
>>> # Exactly equivalent to:
... normal_function(a=numbers["a"], b=numbers["b"], c=numbers["c"])
a: 7 b: 5 c: 3
```

Matching Keys

Keys of the dictionary *must* match up with how the function was declared:

```
>>> bad_numbers = {"a": 7, "b": 5, "z": 3}
>>> normal_function(**bad_numbers)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: normal_function() got an unexpected keyword argument 'z'
```

Calling both

You can call a function using both - and both will be unpacked:

```
>>> def addup(a, b, c=1, d=2, e=3):
...     return a + b + c + d + e
...
>>> nums = (3, 4)
>>> extras = {"d": 5, "e": 2}
>>> addup(*nums, **extras)
15
```

Two different things

Python uses `*` and `**` for two *very different* things:

- Variable arguments (when *defining* a function), and
- Argument unpacking (when *calling* a function).

These look similar in code. **But they are completely different things.**

Lab: Variable Arguments

Lab file: `functions/varargs.py`

- In labs/py3 for 3.x; labs/py2 for 2.7

When you finish:

- Knock your sign down, so I know you're done.
- Proceed to `varargs_extra.py`
- When that's done, open and skim through **MasteringPythonDecorators.pdf** - just notice what topics interest you.

Part 2

Functions As Objects

Function object

In Python, **everything** is an object. Including functions.

```
>>> def f(n): return n+2
...
>>> id(f)
4314937816
>>> g = f
>>> print(g(3))
5
>>> id(g)
4314937816
```

You can use this fact to do **amazing things**.

Function Factory

You can create functions inside of functions.

```
def mk_greeter(greeting):
    def greet(name):
        print("Good {}, {}!".format(greeting, name))
    return greet
```

```
>>> good_morning = mk_greeter("morning")
>>> type(good_morning)
<class 'function'>
>>> good_morning("Joe")
Good morning, Joe!
```

Generate Several

We can call `mk_greeter()` many times, to define many functions:

```
>>> good_morning = mk_greeter("morning")
>>> good_evening = mk_greeter("evening")
>>> good_afternoon = mk_greeter("afternoon")
>>>
>>> good_morning("Joe")
Good morning, Joe!
>>> good_evening("Sandra")
Good evening, Sandra!
>>> good_afternoon("Tim")
Good afternoon, Tim!
```

Independent Objects

Each function has its own identity. Each is its own object.

```
>>> id(mk_greeter)
4322785752
>>> id(good_morning)
4322785344
>>> id(good_evening)
4322785480
>>> id(good_afternoon)
4322785208
```

Function Factory Practice

Create a file named `funcfact.py`. Type in the following:

```
def mk_greeter(greeting):
    def greet(name):
        print("Good {}, {}!".format(greeting, name))
    return greet
good_morning = mk_greeter("morning")
good_afternoon = mk_greeter("afternoon")
good_morning("Your Name")
good_afternoon("Your Neighbor's Name")
```

Run, and this is the output:

```
Good morning, Your Name!
Good afternoon, Your Neighbor's Name!
```

EXTRA CREDIT: Can you use `mk_greeter` to print "Good day, Aaron!" directly, without making a `good_day()` function?

Indirection

This works too:

```
>>> mk_greeter("evening")("Bart")
Good evening, Bart!
```

Python parses this left to right.

Function objects can be **anonymous** - meaning, we don't have to give them a name (store them in a variable) to work with them.

Identity != Name

In fact, the **name** we give a function is just a label - not the function itself.

```
>>> def calculate(x): return x + 2
>>> calculate.__name__
'calculate'

>>> good_morning = mk_greeter("morning")
>>> good_morning.__name__
'greet'
```

Think of `calculate` as a variable, storing an object... which happens to be a function object.

Passing To Functions

If we can store a function object in a variable, we can pass it to another function.

```
def calltwice(func, arg):
    func(arg)
    func(arg)
```

```
>>> calltwice(good_morning, "everyone")
Good morning, everyone!
Good morning, everyone!
```

Passing To Functions

The function object will have a different name in different contexts.

But it's still the same function object.

```
def calltwicewithid(func, arg):
    print("ID of func: " + str(id(func)))
    func(arg)
    func(arg)
```

```
>>> calltwicewithid(good_morning, "everyone")
ID of func: 4314397008
Good morning, everyone!
Good morning, everyone!

>>> id(good_morning)
4314397008

>>> id(calltwicewithid)
4314396736
```

Call with arguments

Create a file called `callwithargs.py` - type in the following, and run:

```
def sayhello(name):
    print("Hello, " + name)
def callwithargs(func, first, second):
    func(first)
    func(second)

callwithargs(sayhello, "Your Name", "Your Neighbor")
```

Output:

```
Hello, Your Name
Hello, Your Neighbor
```

EXTRA CREDIT: Can you make `callwithargs` take any number of names?

Call with many arguments

```
def callwithargs(func, *args):
    for arg in args:
        func(arg)
```

```
>>> callwithargs(sayhello, "Tim")
Hello, Tim
>>> callwithargs(sayhello, "Bob", "John")
Hello, Bob
Hello, John
>>> callwithargs(sayhello, "Luke", "Darth", "Ben")
Hello, Luke
Hello, Darth
Hello, Ben
```

Wrapping

An interesting meta-pattern: wrapper functions.

```
def tell(func):
    def newfunc(arg):
        print("CALLING: " + func.__name__)
        return func(arg)
    return newfunc
```

```
>>> tell_sum = tell(sum)
>>> tell_sum([2, 4, 6])
CALLING: sum
12
```

Lab: Function Objects

Lab file: `functions/funcobj.py`

- In labs/py3 for 3.x; labs/py2 for 2.7

When you finish:

- Knock your sign down, so I know you're done.
- Proceed to `functions/funcobj_extra.py`

Part 3

Basics of Decorators

Example: property

```
>>> class Person:
...     def __init__(self, first_name, last_name):
...         self.first_name = first_name
...         self.last_name = last_name
...
...     @property
...     def full_name(self):
...         return self.first_name + " " + self.last_name
...
>>> person = Person("John", "Smith")
>>> print(person.full_name)
John Smith
```

Example: Flask

```
@app.route("/")
def hello():
    return "<html><body>Hello World!</body></html>"
```

Example: thread locking

```
@withlock
def first_method_in_group(self, arg):
    ...
@withlock
def another_method_in_group(self, arg):
    ...
```

@ is a Shorthand

This:

```
@some_decorator
def some_function(arg):
    # blah blah
```

is equivalent to this:

```
def some_function(arg):
    # blah blah
some_function = some_decorator(some_function)
```

It's just a function

A decorator is **just a function**. That's all.

It is a function that takes exactly one argument, which is a function object.

And it returns a *different* function.

```
def some_function(arg):
    # blah blah
some_function = some_decorator(some_function)
```

Terminology

```
@some_decorator
def some_function(arg):
    # blah blah
```

- **decorator** - What comes after the @. It's a function.
- **bare function** - the one `def`'ed on the next line. The function being decorated.
- The result of decorating a function is the **decorated function**. It's what you actually call in your code.

Remember one thing

A decorator is just a normal, boring function.

It happens to be a function that takes exactly one argument, which is itself a function.

And when called, the decorator returns a *different* function.

Logging decorator

```
def printlog(func):
    def wrapper(arg):
        print("CALLING: " + func.__name__)
        return func(arg)
    return wrapper

@printlog
def f(n):
    return n+2

# Same as:
def f(n):
    return n+2
f = printlog(f)
```

```
>>> print(f(3))
CALLING: f
5
```

Structure

```
def printlog(func):
    def wrapper(arg):
        print("CALLING: " + func.__name__)
        return func(arg)
    return wrapper
```

Body of printlog does just two things:

- Define a function called `wrapper`, and
- Return it.

That's all. Most decorators you create will follow this pattern.

Multiple Targets

Decorators are normally applied to many functions or methods.

```
@printlog
def f(n):
    return n+2
@printlog
def g(x):
    return 5 * x
@printlog
def h(arg):
    return 10 + arg
```

```
>>> print(f(3))
CALLING: f
5
>>> print(g(4))
CALLING: g
20
>>> print(h(5))
CALLING: h
15
```

Masking

```
def check_id(func):
    def wrapper(arg):
        print("ID of func: {}".format(id(func)))
        return func(arg)
    print("ID of wrapper: {}".format(id(wrapper)))
    return wrapper
```

```
>>> @check_id
... def f(x): return x * 3
ID of wrapper: 4329698984
>>>
>>> f(2)
ID of func: 4329698576
6
>>> id(f)
4329698984
```

Practice syntax

Open a file named `decorators1.py`, and type this in:

```
def printlog(func):
    def wrapper(arg):
        print("CALLING: " + func.__name__)
        return func(arg)
    return wrapper
@printlog
def f(n):
    return n+2

print(f(3))
```

Run the script. Output should be:

```
CALLING: f
5
```

Extra credit: Define & decorate new functions. Can you trigger interesting errors?

A shortcoming

```
>>> @printlog
... def baz(x, y):
...     return x ** y
...
>>> baz(3,2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: wrapper() takes 1 positional argument but 2 were given
```

What went wrong?

Generalizing

```
# A MUCH BETTER printlog.
def printlog(func):
    def wrapper(*args, **kwargs):
        print("CALLING: " + func.__name__)
        return func(*args, **kwargs)
    return wrapper
```

Rule of thumb: always define your `wrapper` function to accept `*args` and `**kwargs`, unless you have a specific reason not to.

Generalized

This decorator is compatible with *any* Python function:

```
>>> @printlog
... def foo(x):
...     print(x + 2)
...
>>> @printlog
... def baz(x, y):
...     return x ** y
...
>>> foo(7)
CALLING: foo
9
>>> baz(3, 2)
CALLING: baz
9
```

Practice syntax

Open a file named `decorators2.py`, and type this in:

```
def printlog(func):
    def wrapper(*args, **kwargs):
        print("CALLING: " + func.__name__)
        return func(*args, **kwargs)
    return wrapper
@printlog
def g(a, b, c):
    return a + b + c
print(g(1,2,3))
```

Run the script. Output should be:

```
CALLING: g
6
```

Why *args and **kwargs?

Two words: flexibility and power.

A decorator written to take arbitrary arguments can work with functions and methods written *years* later - code the original developer never could have anticipated.

This structure has proven very powerful and versatile.

```
# The prototypical form of Python decorators.
def prototype_decorator(func):
    def wrapper(*args, **kwargs):
        return func(*args, **kwargs)
    return wrapper
```

State in decorators

```
def history(func):
    return_vals = set()
    def wrapper(*args, **kwargs):
        return_val = func(*args, **kwargs)
        return_vals.add(return_val)
        print('Return values: ' + str(sorted(return_vals)))
        return return_val
    return wrapper

@history
def foo(x):
    return x + 2

# Remember, same as:
def foo(x):
    return x + 2
foo = history(foo)
```

History

```
>>> print(foo(3))
Return values: [5]
5
>>> print(foo(2))
Return values: [4, 5]
4
>>> print(foo(3))
Return values: [4, 5]
5
>>> print(foo(7))
Return values: [4, 5, 9]
9
```

Memoization

A function design pattern.

Given an expensive function `f`, you can cache its value.

```
def f(x, y, z):
    # do something expensive

cache = {}
def cached_f(x, y, z):
    # tuples can be dictionary keys.
    key = (x, y, z)
    if key not in cache:
        cache[key] = f(x, y, z)
    return cache[key]
```

This has been around for decades. It's still useful.

Lab: memoize

```
# Turn this:
cache = {}
def cached_f(x, y, z):
    # tuples can be dictionary keys.
    key = (x, y, z)
    if key not in cache:
        cache[key] = f(x, y, z)
    return cache[key]

# ... into this:
@memoize
def f(x, y, z):
   # ...
```

Lab file: `decorators/memoize.py`

- In `labs/py3` for Python 3.x, or `labs/py2` for 2.7

When you finish:

- Knock your sign down, so I know you're done.
- Proceed to `decorators/memoize_extra.py`

Stacking Decorators

You can stack decorators. Simply write them on separate lines.

```
@add2
@mult3
def foo(n):
    return n + 1
# That's shorthand for this:
foo = add2(mult3(foo))
```

`add2` adds two, and `mult3` multiplies by three:

```
def add2(func):
    def wrapper(*args, **kwargs):
        return func(*args, **kwargs) + 2
    return wrapper
```

```
def mult3(func):
    def wrapper(*args, **kwargs):
        return func(*args, **kwargs) * 3
    return wrapper
```

What will `foo(3)` return?

Stacking Order

The order of stacking matters.

```
>>> # shorthand for "foo = add2(mult3(foo))"
... @add2
... @mult3
... def foo(n):
...     return n + 1
>>> foo(3)
14

>>> # shorthand for "foo = mult3(add2(foo))"
... @mult3
... @add2
... def foo(n):
...     return n + 1
>>> foo(3)
18
```

Part 4

Decorators That Take Arguments

Decorators That Take Arguments

Remember this:

```
@app.route("/")
def hello():
    return "<html><body>Hello World!</body></html>"
```

This is different from the decorators we've written so far, because it takes an argument. How do we do that?

Simpler example

Imagine a family of "adding" decorators.

```
def add2(func):
    def wrapper(*args, **kwargs):
        return func(*args, **kwargs) + 2
    return wrapper

def add4(func):
    def wrapper(*args, **kwargs):
        return func(*args, **kwargs) + 4
    return wrapper

@add2
def foo(x):
    return x ** 2

@add4
def bar(n):
    return n * 2
```

DRY

There is literally only one character difference between `add2` and `add4`; it's very repetitive, and poorly maintainable.

Better:

```
@add(2)
def foo(x):
    return x ** 2

@add(4)
def bar(n):
    return n * 2
```

How do we do that?

Generating decorators

```
@add(2)
def foo(x):
    return x ** 2
```

add is actually *not* a decorator; it is a function that *returns* a decorator.

In other words, add is a function that returns another function. (Since the returned decorator is, itself, a function).

Nesting functions

Write a function called `add`, which creates and returns the decorator.

```
def add(increment):
    def decorator(func):
        def wrapper(*args, **kwargs):
            return increment + func(*args, **kwargs)
        return wrapper
    return decorator
```

Using add()

These all mean the exact same thing:

```
# This...
@add(2)
def f(n):
    # ....

# ... is the same as this...
add2 = add(2)
@add2
def f(n):
    # ....

# ... and the same as this.
def f(n):
    # ....
f = add(2)(f)
```

Break it down...

```
def add(increment):
    def decorator(func):
        def wrapper(*args, **kwargs):
            return increment + func(*args, **kwargs)
        return wrapper
    return decorator
```

- `wrapper`: just like in the other decorators
- `decorator`: What's applied to the bare function
- (Hint: we could say `add2 = add(2)`, then apply `add2` as a decorator)
- `add`: This is not a decorator. It's a function that returns a decorator.

Closure

```
def add(increment):
    def decorator(func):
        def wrapper(*args, **kwargs):
            return increment + func(*args, **kwargs)
        return wrapper
    return decorator
```

`increment` variable is encapsulated in the scope of the `add` function.

We can't access its value outside the decorator, in the calling context. But we don't need to.

Practice syntax

Create a file `decoratoradd.py`, and write in the following:

```
def add(increment):
    def decorator(func):
        def wrapper(*args, **kwargs):
            return increment + func(*args, **kwargs)
        return wrapper
    return decorator


@add(3)
def f(n):
    return n + 2

print(f(4))
```

Output shoud be "9".

Extra credit: Create and use a `multiply` decorator.

Lab: The returns decorator

Runtime type checking:

```
# Raises TypeError if return value is not an int
@returns(int)
def f(x, y):
    if x > 3:
        return -1.5
    return x + y
```

(Hint: use `isinstance()`)

Lab file: `decorators/returns.py`

- In labs/py3 for 3.x; labs/py2 for 2.7 When you finish:
- Knock your sign down, so I know you're done.
- Proceed to `decorators/webframework.py`

Part 5

Class-Based Decorators

Class-Based Decorators

So far, we've made each decorator by defining a function.

It turns out, you can also create one using a class.

Advantages:

- Can leverage inheritance, encapsulation, etc.
- Can sometimes be more readable for complex decorators

The call hook

Any object with a `__call__` method can be treated like a function.

```
class Prefixer:
    def __init__(self, prefix):
        self.prefix = prefix
    def __call__(self, message):
        return self.prefix + message
```

It's called a **callable**, meaning you can call it like a function:

```
>>> simonsays = Prefixer("Simon says: ")
>>> simonsays("Get up and dance!")
'Simon says: Get up and dance!'
```

The call hook

It's not a function! It's just callable like one.

```
>>> type(simonsays)
<class '__main__.Prefixer'>
```

When you call it like a function, this dispatches to the `__call__` method.

```
>>> simonsays("High five!")
'Simon says: High five!'
>>> simonsays.__call__("High five!")
'Simon says: High five!'
```

Important Note

It's possible to apply decorators to classes, just like you've applied them to functions.

This is a COMPLETELY DIFFERENT THING than class-based decorators.

@printlog as a function

As a reminder (the same code as before):

```
def printlog(func):
    def wrapper(*args, **kwargs):
        print("CALLING: " + func.__name__)
        return func(*args, **kwargs)
    return wrapper
```

```
>>> @printlog
... def foo(x):
...     print(x + 2)
...
>>> foo(7)
CALLING: foo
9
```

@PrintLog as a class

```
class PrintLog:
    def __init__(self, func):
        self.func = func
    def __call__(self, *args, **kwargs):
        print('CALLING: {}'.format(self.func.__name__))
        return self.func(*args, **kwargs)

# Compare to the function version (from last slide):
def printlog(func):
    def wrapper(*args, **kwargs):
        print("CALLING: " + func.__name__)
        return func(*args, **kwargs)
    return wrapper
```

Works the same!

To use this:

```
>>> @printlog
... def foo_func(x):
...     print(x + 2)
...
>>> @PrintLog
... def foo_class(x):
...     print(x + 2)
...
>>> foo_func(7)
CALLING: foo_func
9
>>> foo_class(7)
CALLING: foo_class
9
```

Another look

```
class PrintLog:
    def __init__(self, func):
        self.func = func
    # ...
```

Constructor takes one arg: the function being decorated. Remember, this:

```
@PrintLog
def foo_class(x):
    print(x+2)
```

is shorthand for this:

```
def foo_class(x):
    print(x+2)
foo_class = PrintLog(foo_class)
```

The wrapped "function" is actually a `PrintLog` object.

Another look

```
class PrintLog:
    def __init__(self, func):
        self.func = func
    def __call__(self, *args, **kwargs):
        print('CALLING: {}'.format(self.func.__name__))
        return self.func(*args, **kwargs)
```

The function being decorated is stored as `self.func`.

`__call__` is, in essence, the wrapper function.

Uses

Some reasons to use class-based decorators instead of functions:

1) To leverage inheritance, or other OO features

2) To store state in the decorator (as object attributes)

3) You feel it's more readable. (Some people like one form better than the other.)

Lab: Classy Memoizing

```
# Turn this:
cache = {}
def cached_f(x, y, z):
    # tuples can be dictionary keys.
    key = (x, y, z)
    if key not in cache:
        cache[key] = f(x, y, z)
    return cache[key]
# ... into this (a Memoize class, instead of a memoize function):
@Memoize
def f(x, y, z):
   # ...
```

Lab file: `decorators/memoize_class.py`

- In labs/py3 for 3.x; labs/py2 for 2.7

When you finish:

- Knock your sign down, so I know you're done.
- Proceed to `decorators/memoize_class_extra.py` (after `memoize_extra.py`)

Remember This?

Now that you're experts in Python decorators:

```
# Retry this many times before giving up.
MAX_TRIES = 3
tries = 0
resp = None
while True:
    resp = make_api_request()
    if resp.status_code == 500 and tries < MAX_TRIES:
        tries += 1
        continue
    break
todo_items = resp.json()
for item in todo_items:
    # Do something, like print it out, etc.
```

You have dozens of functions and methods like `make_api_request()`. How do you reusably capture this pattern?

Retry Decorator

```
def retry(func):
    def wrapper(*args, **kwargs):
        MAX_TRIES = 3
        tries = 0
        while True:
            resp = func(*args, **kwargs)
            if resp.status_code == 500 and tries < MAX_TRIES:
                tries += 1
                continue
            break
        return resp
    return wrapper

@retry
def make_api_request():
    return requests.get(API_URL + '/items')
```