Higher-Order Decorators

Stacking Decorators

You can stack decorators.

```
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
```

Simply write them on separate lines:

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

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
```

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 - Don't Repeat Yourself

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.

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 should be "9".

Extra credit: Create and use a `multiply` 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.

Lab: Decorators With Arguments

Lab file: `decorators/decoratorargs.py`

- In `labs` folder
- When you are done, study the solution - compare to what you wrote.
- ... and then do `decorators/webframework.py`