

Pandy Knight

Automation Panda

Developer Advocate at Applitools





Pandy Knight

Automation Panda

Developer Advocate at Applitools



Regular Function

```
def hello_world():  
    print('Hello World!')
```

Regular Function

```
def hello_world():  
    print('Hello World!')
```

```
>>> hello_world()  
Hello World!
```

Regular Function

```
def hello_world():  
    print('Hello World!')
```

Decorated Function

```
@tracer  
def hello_world():  
    print('Hello World!')
```

Decorated Function

```
def tracer(func):
```

```
@tracer
```

```
def hello_world():
```

```
    print('Hello World!')
```

Decorated Function

```
def tracer(func):
```



```
@tracer
```

```
def hello_world():  
    print('Hello World!')
```


Decorated Function

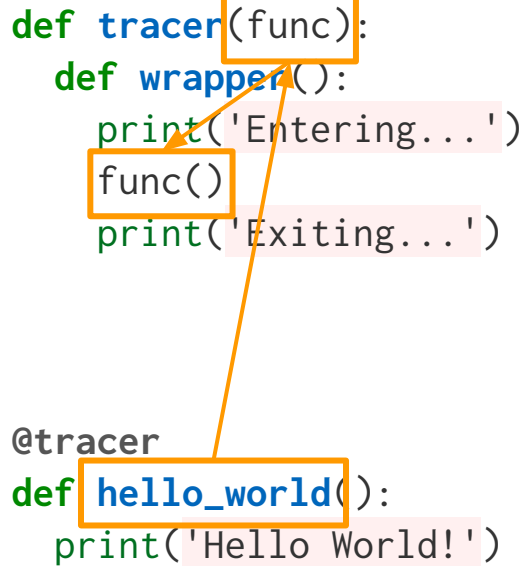
```
def tracer(func):  
    def wrapper():  
        print('Entering...')  
        func()  
        print('Exiting...')
```

@tracer

```
def hello_world():  
    print('Hello World!')
```

Decorated Function

```
def tracer(func):  
    def wrapper():  
        print('Entering...')  
        func()  
        print('Exiting...')  
  
@tracer  
def hello_world():  
    print('Hello World!')
```



Decorated Function

```
def tracer(func):  
    def wrapper():  
        print('Entering...')  
        func()  
        print('Exiting...')  
    return wrapper
```

@tracer

```
def hello_world():  
    print('Hello World!')
```

Decorated Function

```
def tracer(func):  
    def wrapper():  
        print('Entering...')  
        func()  
        print('Exiting...')  
    return wrapper
```

@tracer

```
def hello_world():  
    print('Hello World!')
```

```
>>> hello_world()  
Entering...  
Hello World!  
Exiting...
```

Decorators

Decorators wrap

Decorators wrap functions

Decorators wrap functions around

**Decorators wrap
functions around
functions!**

**Decorators wrap
functions around
functions!**



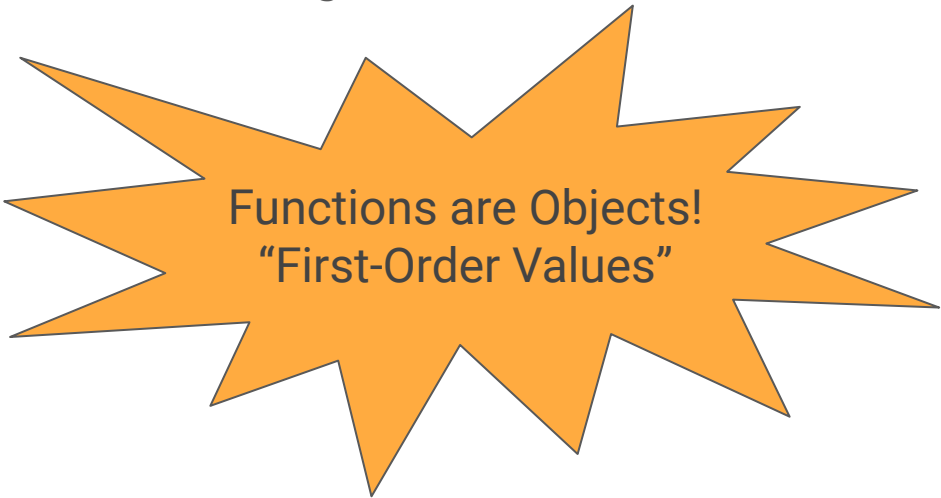


Functions are Objects!

```
def tracer(func):  
    def wrapper():  
        print('Entering...')  
        func()  
        print('Exiting...')  
    return wrapper
```

```
@tracer  
def hello_world():  
    print('Hello World!')
```

```
>>> hello_world()  
Entering...  
Hello World!  
Exiting...
```



Functions are Objects!
"First-Order Values"

Functions are Objects!

```
def tracer(func):  
    def wrapper():  
        print('Entering...')  
        func()  
        print('Exiting...')  
    return wrapper
```

```
@tracer  
def hello_world():  
    print('Hello World!')
```

```
>>> hello_world()  
Entering...  
Hello World!  
Exiting...
```



Pass functions into
functions!

Functions are Objects!

```
def tracer(func):  
    def wrapper():  
        print('Entering...')  
        func()  
        print('Exiting...')  
    return wrapper
```

```
@tracer  
def hello_world():  
    print('Hello World!')
```

```
>>> hello_world()  
Entering...  
Hello World!  
Exiting...
```




Define new functions
inside functions!

Functions are Objects!

```
def tracer(func):  
    def wrapper():  
        print('Entering...')  
        func()  
        print('Exiting...')  
    return wrapper
```

```
@tracer  
def hello_world():  
    print('Hello World!')
```

```
>>> hello_world()  
Entering...  
Hello World!  
Exiting...
```

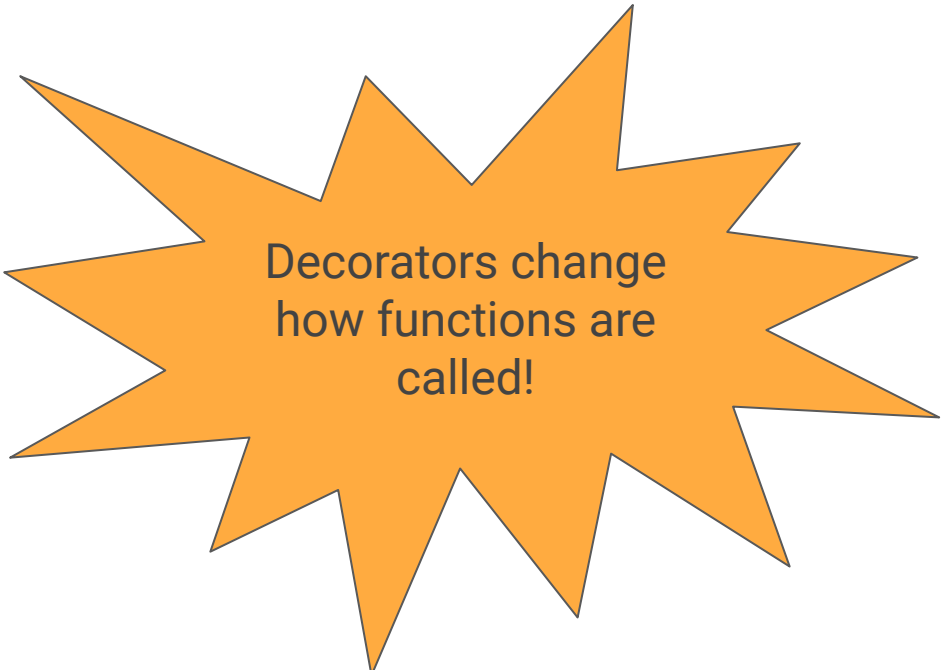


Return a function
from a function!

Functional Programming!

Decorators Change Functions!

```
def tracer(func):  
    def wrapper():  
        print('Entering...')  
        func()  
        print('Exiting...')  
    return wrapper  
  
@tracer  
def hello_world():  
    print('Hello World!')
```



Decorators change
how functions are
called!

Decorators Change Functions!

```
def tracer(func):  
    def wrapper():  
        print('Entering...')  
        func()  
        print('Exiting...')  
    return wrapper
```


```
@tracer  
def hello_world():  
    print('Hello World!')
```

“Outer” Decorator Function

“Inner” Decorated Function

Decorators Change Functions!

```
def tracer(func):  
    def wrapper():  
        print('Entering...')  
        func()  
        print('Exiting...')  
    return wrapper  
  
@tracer  
def hello_world():  
    print('Hello World!')
```



Add "advice" before
and after the inner
function call!

Decorators Change Functions!

```
def tracer(func):  
    def wrapper():  
        print('Entering...')  
        func()  
        print('Exiting...')  
    return wrapper
```

@tracer

```
def hello_world():  
    print('Hello World!')
```

@tracer

```
def goodbye_world():  
    print('Goodbye World!')
```



Apply decorators to
any function!

Aspect-Oriented Programming!

**Decorators wrap
functions around
functions!**



Hold on! We have a problem!



Mistaken Identity?

```
def tracer(func):  
    def wrapper():  
        print('Entering...')  
        func()  
        print('Exiting...')  
    return wrapper
```

@tracer

```
def hello_world():  
    print('Hello World!')
```

```
>>> hello_world()  
Entering...  
Hello World!  
Exiting...
```


Mistaken Identity?

```
def tracer(func):  
    def wrapper():  
        print('Entering...')  
        func()  
        print('Exiting...')  
    return wrapper
```

```
>>> hello_world()  
Entering...  
Hello World!  
Exiting...
```

```
>>> hello_world.__name__  
'wrapper'
```

@tracer

```
def hello_world():  
    print('Hello World!')
```

Mistaken Identity?

```
def tracer(func):  
    def wrapper():  
        print('Entering...')  
        func()  
        print('Exiting...')  
    return wrapper
```

@tracer

```
def hello_world():  
    print('Hello World!')
```

```
>>> hello_world()  
Entering...  
Hello World!  
Exiting...
```

```
>>> hello_world.__name__  
'wrapper'
```

```
>>> help(hello_world)  
Help on function wrapper in module  
decorator:  
wrapper()
```

Corrected Identity!

```
import functools
```

```
def tracer(func):  
    @functools.wraps(func)  
    def wrapper():  
        print('Entering...')  
        func()  
        print('Exiting...')  
    return wrapper
```

```
@tracer
```

```
def hello_world():  
    print('Hello World!')
```

```
>>> hello_world()  
Entering...  
Hello World!  
Exiting...
```

```
>>> hello_world.__name__  
'hello_world'
```

```
>>> help(hello_world)  
Help on function wrapper in module  
decorator:  
hello_world()
```

Wait! There's another problem!



Inputs and Outputs?

```
def tracer(func):  
    @functools.wraps(func)  
    def wrapper():  
        print('Entering...')  
        func()  
        print('Exiting...')  
    return wrapper
```

@tracer

```
def hello(name):  
    return f'Hello {name}!'
```

Inputs and Outputs?

```
def tracer(func):  
    @functools.wraps(func)  
    def wrapper():  
        print('Entering...')  
        func()  
        print('Exiting...')  
    return wrapper
```

@tracer

```
def hello(name):  
    return f'Hello {name}!'
```

```
>>> h = hello('Andy')
```

Inputs and Outputs?

```
def tracer(func):  
    @functools.wraps(func)  
    def wrapper():  
        print('Entering...')  
        func()  
        print('Exiting...')  
    return wrapper
```

@tracer

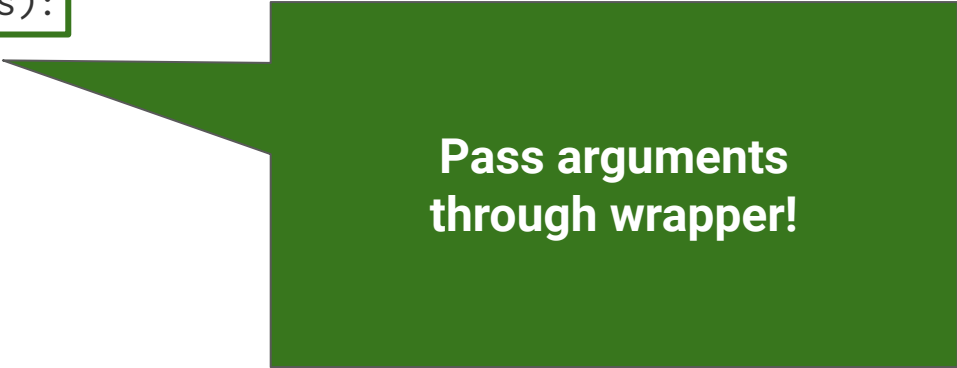
```
def hello(name):  
    return f'Hello {name}!'
```

```
>>> h = hello('Andy')  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
TypeError: wrapper() takes 0  
positional arguments but 1 was given
```

Inputs and Outputs?

```
def tracer(func):  
    @functools.wraps(func)  
    def wrapper(*args, **kwargs):  
        print('Entering...')  
        func(*args, **kwargs)  
        print('Exiting...')  
    return wrapper
```

```
@tracer  
def hello(name):  
    return f'Hello {name}!'
```

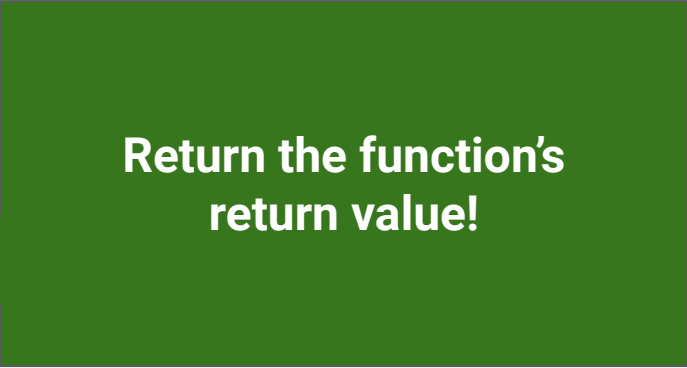


Pass arguments
through wrapper!

Inputs and Outputs?

```
def tracer(func):  
    @functools.wraps(func)  
    def wrapper(*args, **kwargs):  
        print('Entering...')  
        answer = func(*args, **kwargs)  
        print('Exiting...')  
        return answer  
    return wrapper
```

```
@tracer  
def hello(name):  
    return f'Hello {name}!'
```



Return the function's
return value!

Inputs and Outputs Included!

```
def tracer(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        print('Entering...')
        answer = func(*args, **kwargs)
        print('Exiting...')
        return answer
    return wrapper
```

```
@tracer
def hello(name):
    return f'Hello {name}!'
```

```
>>> h = hello('Andy')
```

Inputs and Outputs Included!

```
def tracer(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        print('Entering...')
        answer = func(*args, **kwargs)
        print('Exiting...')
        return answer
    return wrapper
```

@tracer

```
def hello(name):
    return f'Hello {name}!'
```

```
>>> h = hello('Andy')
Entering...
Exiting...
```

Inputs and Outputs Included!

```
def tracer(func):  
    @functools.wraps(func)  
    def wrapper(*args, **kwargs):  
        print('Entering...')  
        answer = func(*args, **kwargs)  
        print('Exiting...')  
        return answer  
    return wrapper
```

```
@tracer
```

```
def hello(name):  
    return f'Hello {name}!'
```

```
>>> h = hello('Andy')  
Entering...  
Exiting...
```

```
>>> h
```

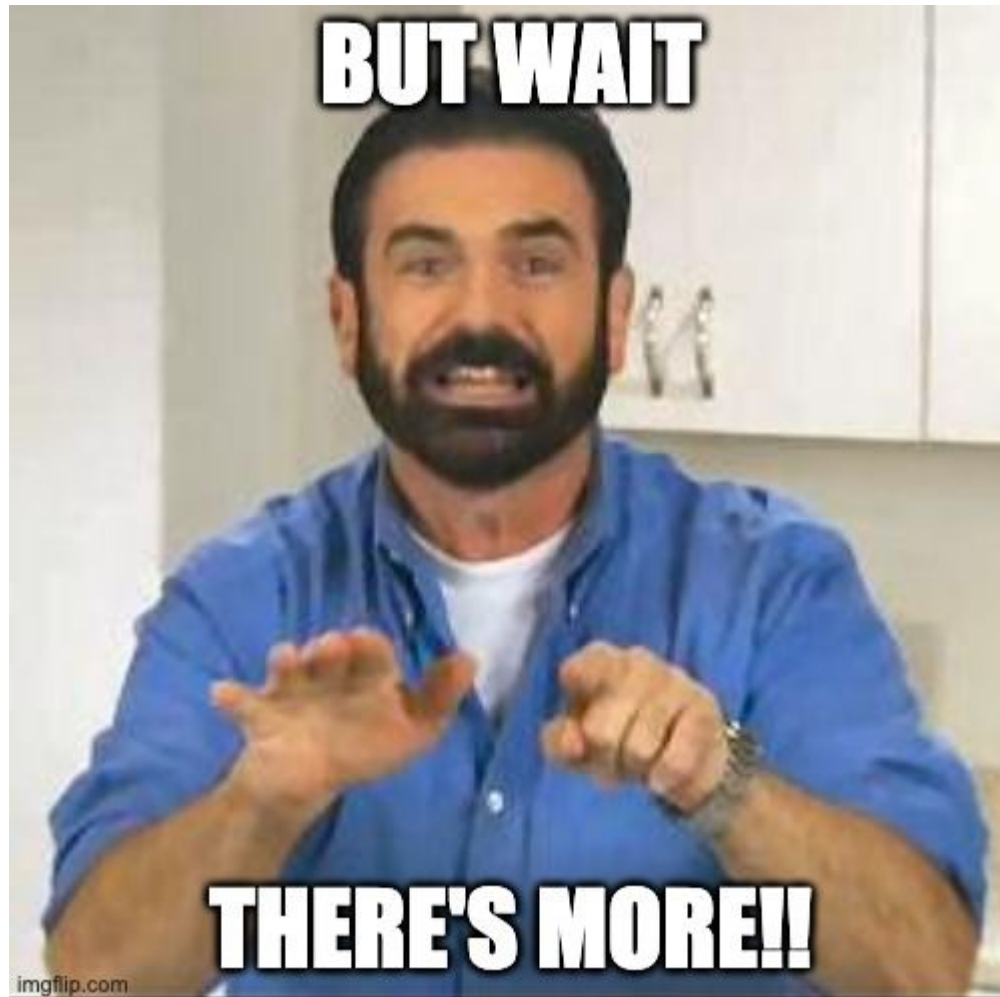
Inputs and Outputs Included!

```
def tracer(func):  
    @functools.wraps(func)  
    def wrapper(*args, **kwargs):  
        print('Entering...')  
        answer = func(*args, **kwargs)  
        print('Exiting...')  
        return answer  
    return wrapper
```

```
@tracer  
def hello(name):  
    return f'Hello {name}!'
```

```
>>> h = hello('Andy')  
Entering...  
Exiting...  
  
>>> h  
'Hello Andy!'
```





Call a Function Twice

```
@call_twice  
def hello_world():  
    print('Hello World!')
```


Call a Function Twice

```
def call_twice(func):  
    @functools.wraps(func)  
    def wrapper(*args, **kwargs):  
  
        return wrapper  
  
@call_twice  
def hello_world():  
    print('Hello World!')
```

Call a Function Twice

```
def call_twice(func):  
    @functools.wraps(func)  
    def wrapper(*args, **kwargs):  
        func(*args, **kwargs)  
        return func(*args, **kwargs)  
    return wrapper
```

```
@call_twice  
def hello_world():  
    print('Hello World!')
```

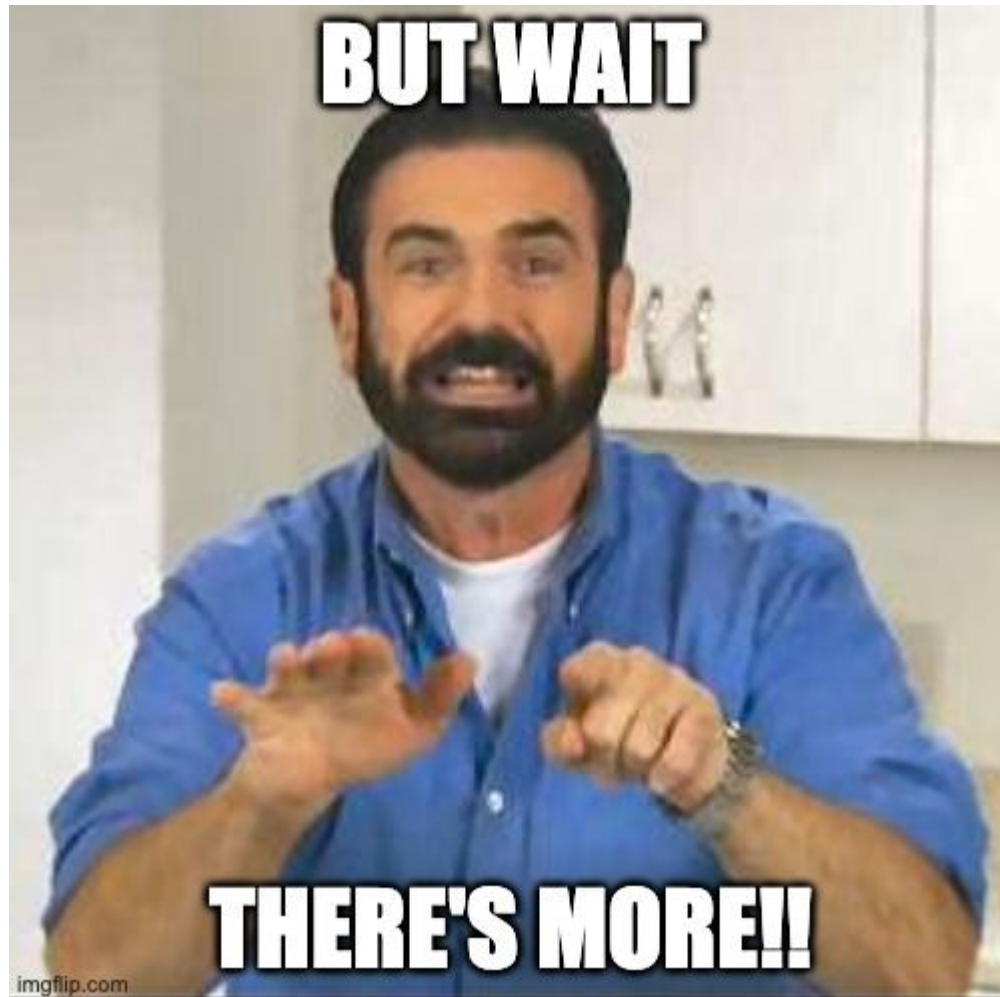
Call a Function Twice

```
def call_twice(func):  
    @functools.wraps(func)  
    def wrapper(*args, **kwargs):  
        func(*args, **kwargs)  
        return func(*args, **kwargs)  
    return wrapper
```

```
@call_twice
```

```
def hello_world():  
    print('Hello World!')
```

```
>>> hello_world()  
Hello World!  
Hello World!
```



Adding a Timer

```
@timer  
def hello_world():  
    print('Hello World!')
```

Adding a Timer

```
def timer(func):  
    @functools.wraps(func)  
    def wrapper(*args, **kwargs):
```

```
        return wrapper
```

```
@timer  
def hello_world():  
    print('Hello World!')
```

Adding a Timer

```
def timer(func):  
    @functools.wraps(func)  
    def wrapper(*args, **kwargs):  
  
        answer = func(*args, **kwargs)  
  
        return answer  
    return wrapper  
  
@timer  
def hello_world():  
    print('Hello World!')
```

Adding a Timer

```
import time

def timer(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        start = time.time()
        answer = func(*args, **kwargs)
        end = time.time()
        print(f'Elapsed: {end - start}')
        return answer
    return wrapper

@timer
def hello_world():
    print('Hello World!')
```

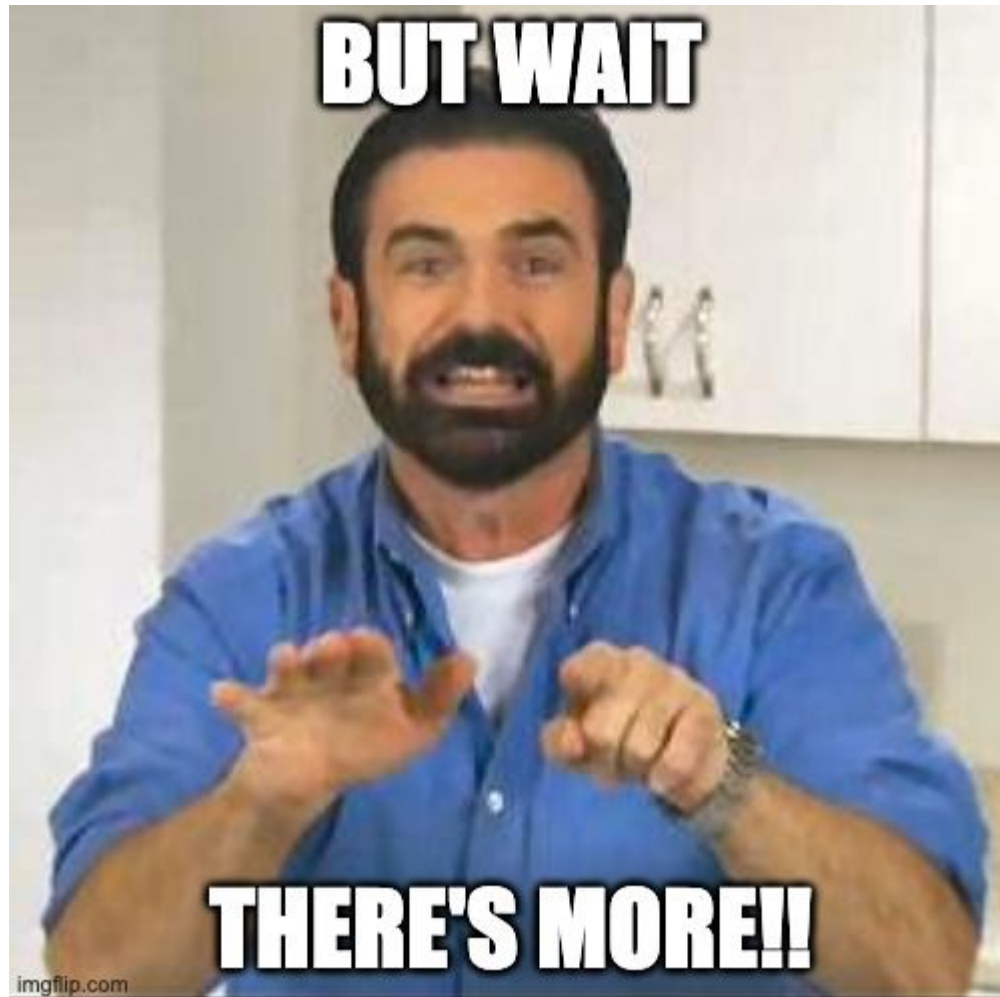

Adding a Timer

```
import time

def timer(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        start = time.time()
        answer = func(*args, **kwargs)
        end = time.time()
        print(f'Elapsed: {end - start}')
        return answer
    return wrapper
```

```
@timer
def hello_world():
    print('Hello World!')
```

```
>>> hello_world()
Hello World!
Elapsed: 0.00013303756713867188
```



imgflip.com

Nesting Decorators

```
@timer  
@call_twice  
def hello_world():  
    print('Hello World!')
```

Nesting Decorators

```
@timer
@call_twice
def hello_world():
    print('Hello World!')
```

```
>>> hello_world()
Hello World!
Hello World!
Elapsed: 7.82012939453125e-05
```

Nesting Decorators

```
@timer
@call_twice
def hello_world():
    print('Hello World!')
```

```
>>> hello_world()
Hello World!
Hello World!
Elapsed: 7.82012939453125e-05
```

```
@call_twice
@timer
def hello_world():
    print('Hello World!')
```

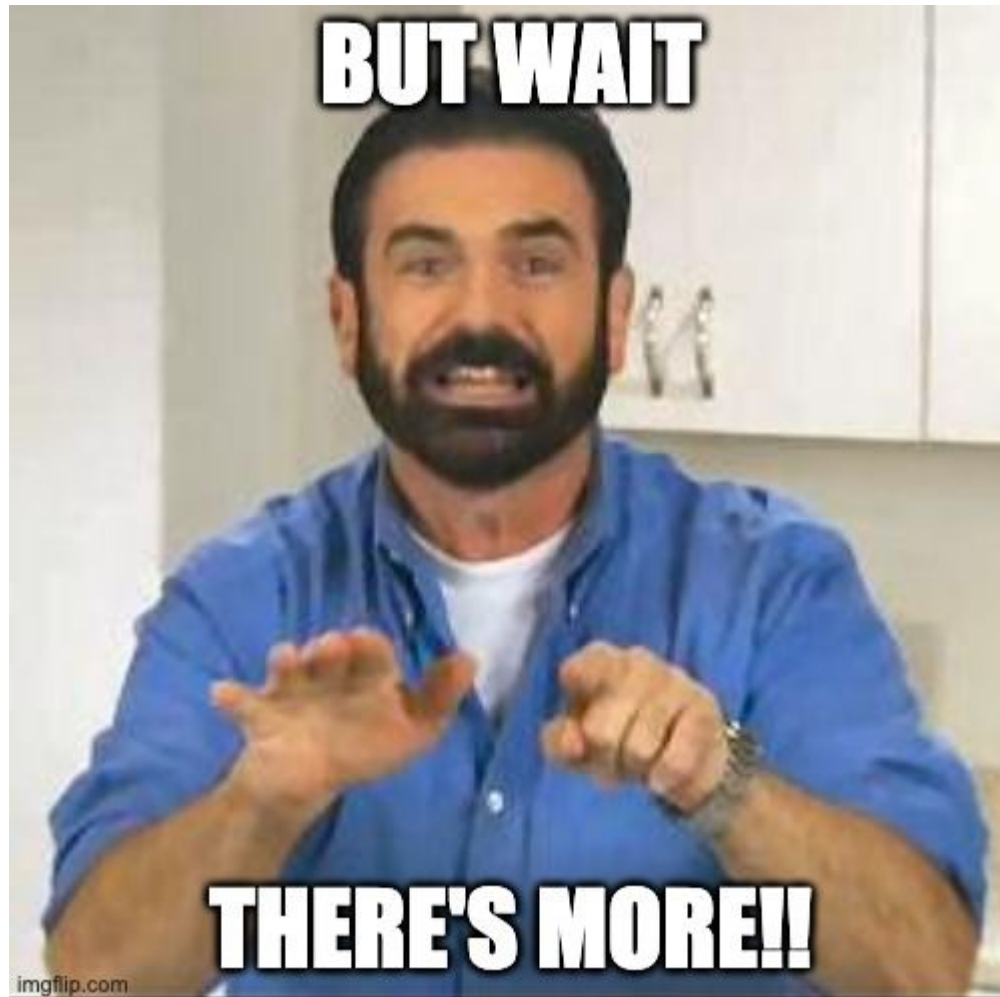
Nesting Decorators

```
@timer
@call_twice
def hello_world():
    print('Hello World!')
```

```
>>> hello_world()
Hello World!
Hello World!
Elapsed: 7.82012939453125e-05
```

```
@call_twice
@timer
def hello_world():
    print('Hello World!')
```

```
>>> hello_world()
Hello World!
Elapsed: 5.1021575927734375e-05
Hello World!
Elapsed: 8.821487426757812e-06
```



Scrubbing and Validating Arguments

```
def add(a, b):  
    return a + b
```

```
def subtract(a, b):  
    return a - b
```


Scrubbing and Validating Arguments

```
def int_inputs(func):
```

```
@int_inputs
```

```
def add(a, b):  
    return a + b
```

```
@int_inputs
```

```
def subtract(a, b):  
    return a - b
```

Scrubbing and Validating Arguments

```
def int_inputs(func):  
    @functools.wraps(func)  
    def wrapper(*args):  
  
        return wrapper
```

```
@int_inputs  
def add(a, b):  
    return a + b
```

```
@int_inputs  
def subtract(a, b):  
    return a - b
```

Scrubbing and Validating Arguments

```
def int_inputs(func):  
    @functools.wraps(func)  
    def wrapper(*args):  
        newargs = [int(a) for a in args]  
        return func(*newargs)  
    return wrapper
```

```
@int_inputs  
def add(a, b):  
    return a + b
```

```
@int_inputs  
def subtract(a, b):  
    return a - b
```

Scrubbing and Validating Arguments

```
def int_inputs(func):  
    @functools.wraps(func)  
    def wrapper(*args):  
        newargs = [int(a) for a in args]  
        return func(*newargs)  
    return wrapper
```

```
@int_inputs
```

```
def add(a, b):  
    return a + b
```

```
@int_inputs
```

```
def subtract(a, b):  
    return a - b
```

```
>>> add(1.5, 2.9)
```

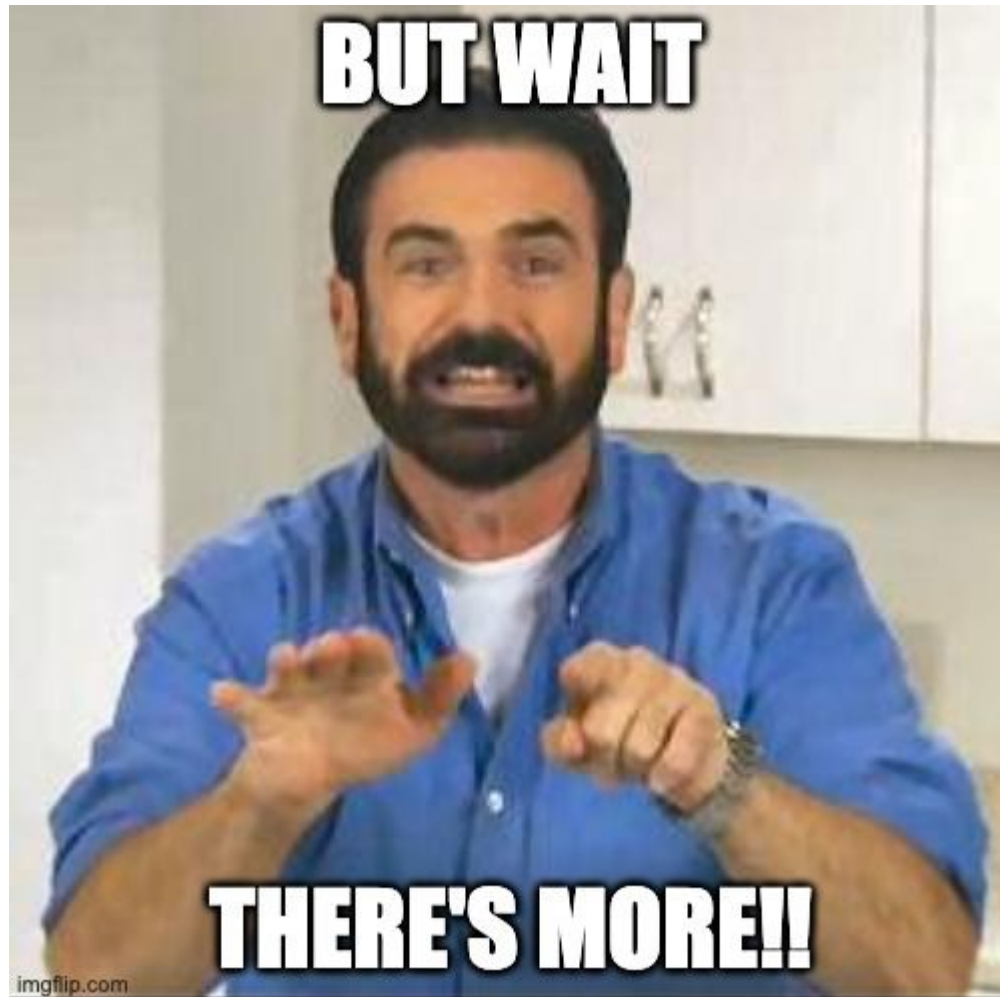
```
3
```

```
>>> subtract('5.4', '3.9')
```

```
2
```

```
>>> add('2', 'abc')
```

```
ValueError: invalid literal for int()  
with base 10: 'abc'
```



imgflip.com

Decorators with Arguments

```
@repeat(5)
def hello_world():
    print('Hello World!')
```

Decorators with Arguments

```
def repeat(count):
```

```
@repeat(5)
```

```
def hello_world():
```

```
    print('Hello World!')
```

Decorators with Arguments

```
def repeat(count):  
    def repeat_decorator(func):
```

```
        return repeat_decorator
```

```
@repeat(5)
```

```
def hello_world():  
    print('Hello World!')
```


Decorators with Arguments

```
def repeat(count):  
    def repeat_decorator(func):  
        @functools.wraps(func)  
        def wrapper(*args, **kwargs):  
  
            return wrapper  
        return repeat_decorator  
  
@repeat(5)  
def hello_world():  
    print('Hello World!')
```

Decorators with Arguments

```
def repeat(count):  
    def repeat_decorator(func):  
        @functools.wraps(func)  
        def wrapper(*args, **kwargs):  
            for _ in range(count):  
                func(*args, **kwargs)  
            return wrapper  
        return repeat_decorator
```

```
@repeat(5)
```

```
def hello_world():  
    print('Hello World!')
```

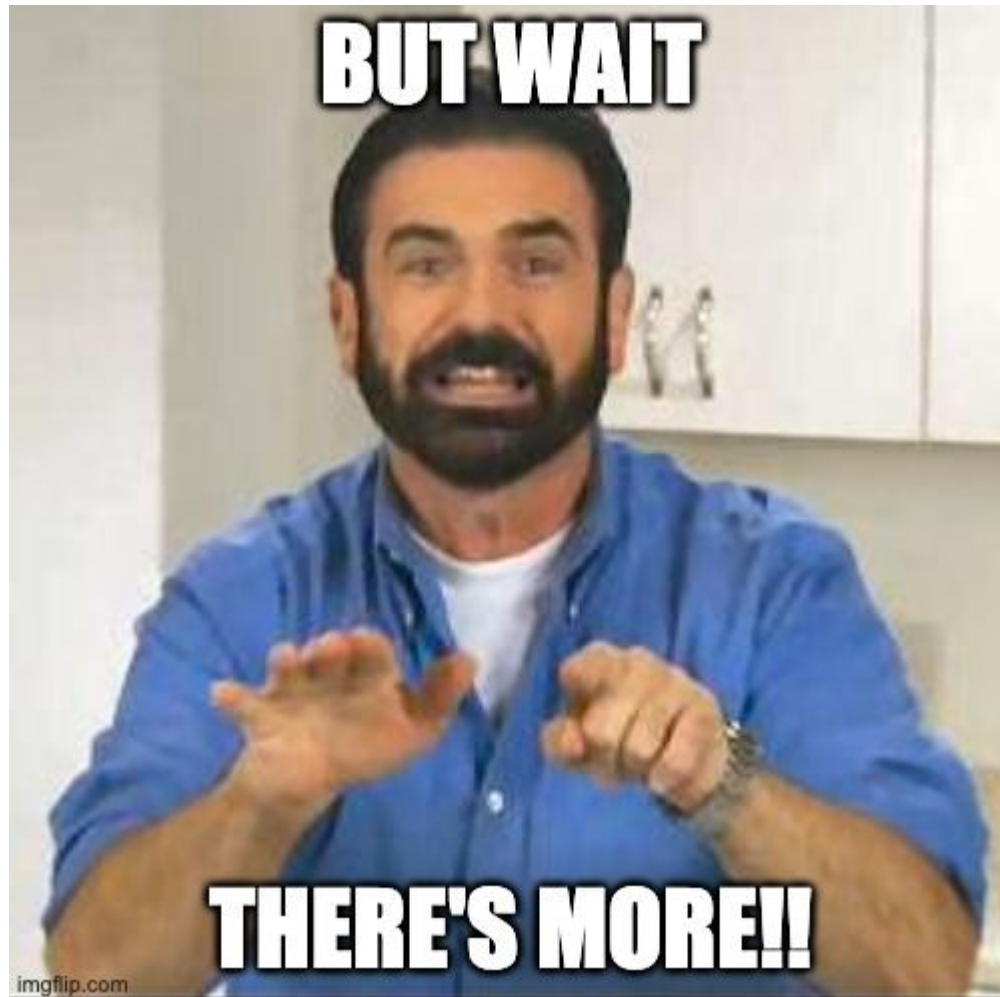
Decorators with Arguments

```
def repeat(count):  
    def repeat_decorator(func):  
        @functools.wraps(func)  
        def wrapper(*args, **kwargs):  
            for _ in range(count):  
                func(*args, **kwargs)  
            return wrapper  
        return repeat_decorator
```

```
@repeat(5)
```

```
def hello_world():  
    print('Hello World!')
```

```
>>> hello_world()  
Hello World!  
Hello World!  
Hello World!  
Hello World!  
Hello World!
```



Decorators that Save State

```
@count_calls  
def hello_world():  
    print('Hello World!')
```

Decorators that Save State

```
def count_calls(func):  
    @functools.wraps(func)  
    def wrapper(*args, **kwargs):
```

```
        return wrapper
```

```
@count_calls
```

```
def hello_world():  
    print('Hello World!')
```

Decorators that Save State

```
def count_calls(func):  
    @functools.wraps(func)  
    def wrapper(*args, **kwargs):
```

```
        wrapper.count = 0  
        return wrapper
```

```
@count_calls
```

```
def hello_world():  
    print('Hello World!')
```

Decorators that Save State

```
def count_calls(func):  
    @functools.wraps(func)  
    def wrapper(*args, **kwargs):  
        wrapper.count += 1  
        return func(*args, **kwargs)  
    wrapper.count = 0  
    return wrapper
```

@count_calls

```
def hello_world():  
    print('Hello World!')
```


Decorators that Save State

```
def count_calls(func):  
    @functools.wraps(func)  
    def wrapper(*args, **kwargs):  
        wrapper.count += 1  
        return func(*args, **kwargs)  
    wrapper.count = 0  
    return wrapper
```

@count_calls

```
def hello_world():  
    print('Hello World!')
```

```
>>> hello_world.count  
0
```

Decorators that Save State

```
def count_calls(func):  
    @functools.wraps(func)  
    def wrapper(*args, **kwargs):  
        wrapper.count += 1  
        return func(*args, **kwargs)  
    wrapper.count = 0  
    return wrapper
```

@count_calls

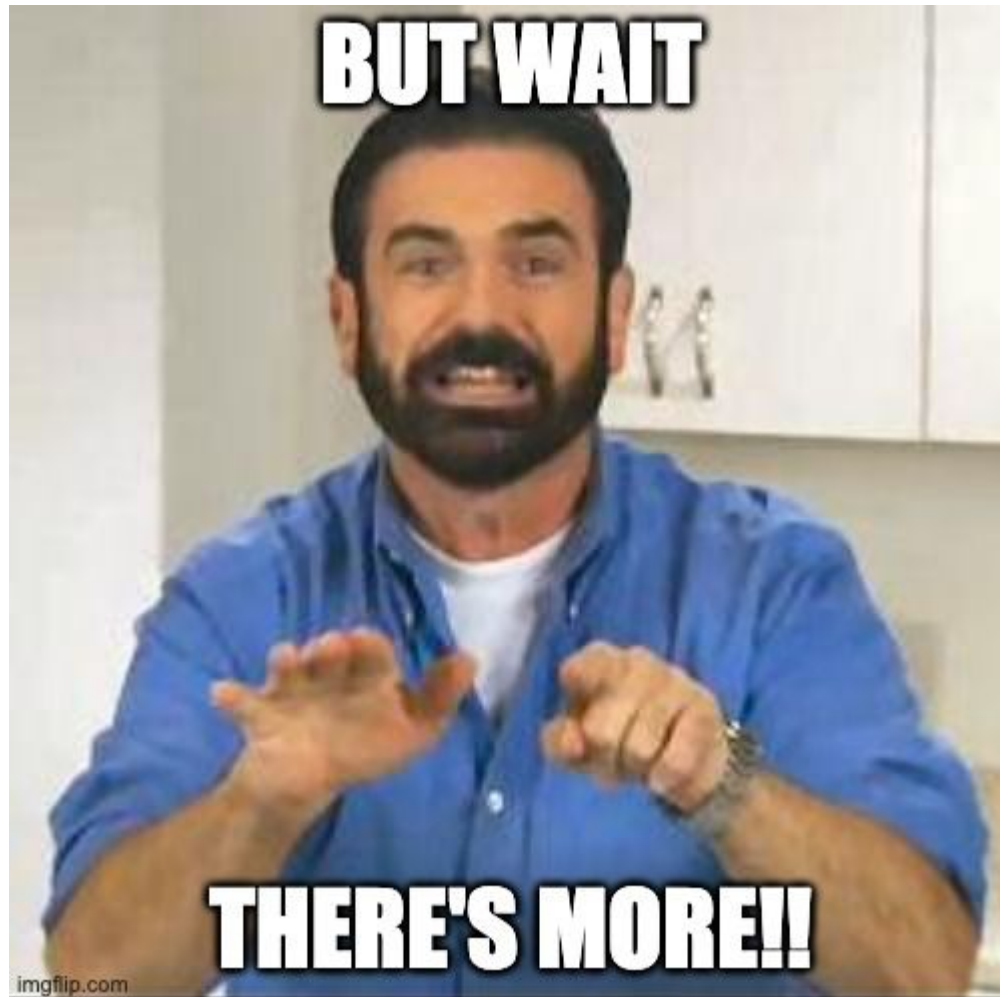
```
def hello_world():  
    print('Hello World!')
```

```
>>> hello_world.count  
0
```

```
>>> hello_world()  
Hello World!
```

```
>>> hello_world()  
Hello World!
```

```
>>> hello_world.count  
2
```



Decorators in Classes

```
# Decorator on a method
```

```
class Greeter:  
  
    @timer  
    def hello(self):  
        print('hello')
```

Decorators in Classes

```
# Decorator on a method
```

```
class Greeter:  
  
    @timer  
    def hello(self):  
        print('hello')
```

```
>>> g = Greeter()
```

```
>>> g.hello()
```

```
hello
```

```
Elapsed: 8.606910705566406e-05
```

Decorators in Classes

```
# Decorator on a class
```

```
@timer
```

```
class Greeter:
```

```
    def hello(self):
```

```
        print('hello')
```

Decorators in Classes

```
# Decorator on a class
```

```
@timer
```

```
class Greeter:
```

```
    def hello(self):  
        print('hello')
```

```
>>> g = Greeter()
```

```
Elapsed: 3.814697265625e-06
```

Decorators in Classes

```
# Decorator on a class
```

```
@timer
```

```
class Greeter:
```

```
    def hello(self):  
        print('hello')
```

```
>>> g = Greeter()
```

```
Elapsed: 3.814697265625e-06
```

```
>>> g.hello()
```

```
hello
```

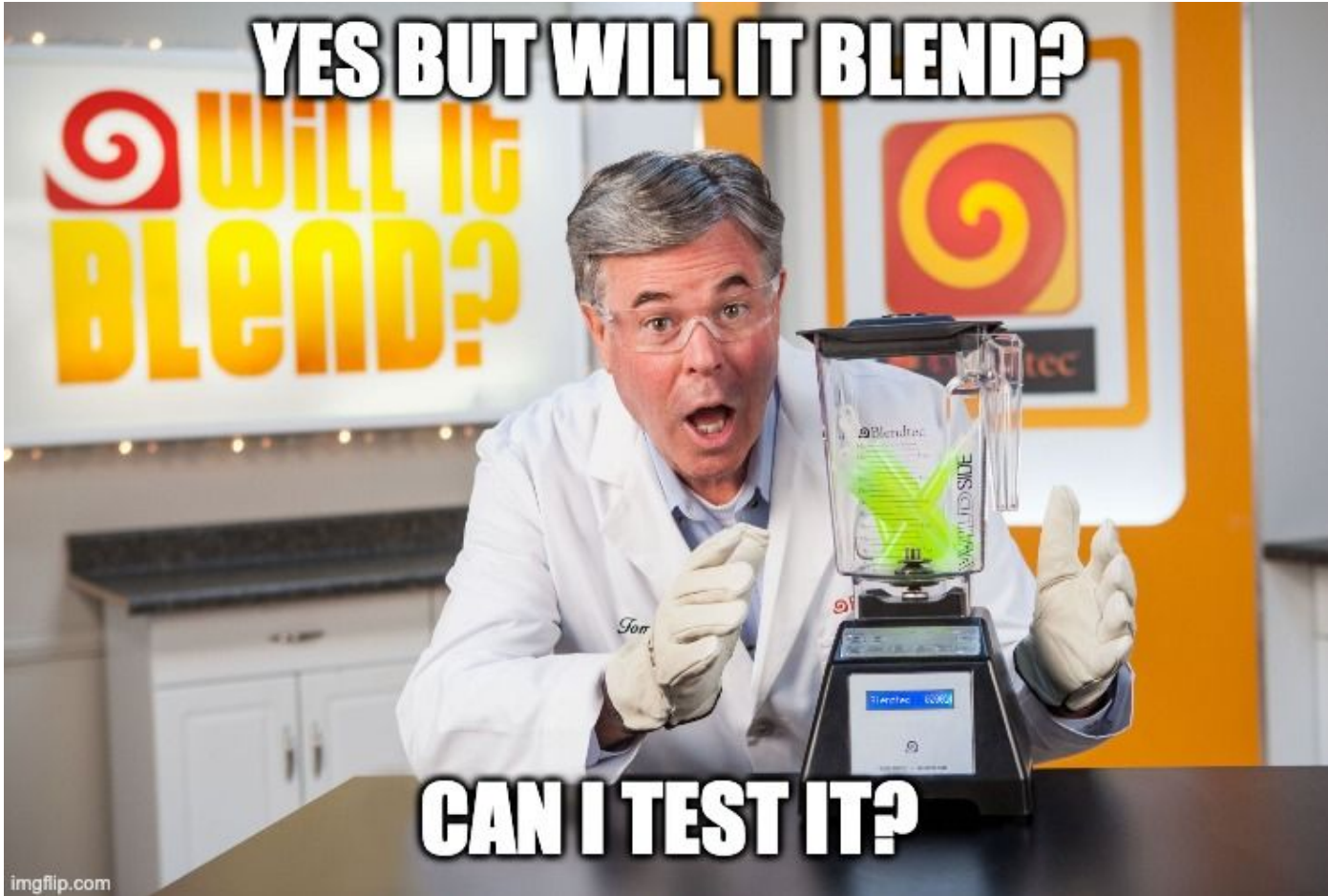
```
# Decorators on classes apply to the  
constructor, not each method
```


Decorators wrap
~~functions around
functions!~~

callables



YES BUT WILL IT BLEND?



CAN I TEST IT?

imgflip.com

Testing Decorators

Testing decorators is challenging!

Here's some advice:

Testing Decorators

Testing decorators is challenging!

Here's some advice:

1. Separate tests for decorator functions from tests for decorated functions.

Testing Decorators

Testing decorators is challenging!

Here's some advice:

1. Separate tests for decorator functions from tests for decorated functions.
2. Apply decorators to “fake” functions used only for testing.

Testing Decorators

Testing decorators is challenging!

Here's some advice:

1. Separate tests for decorator functions from tests for decorated functions.
2. Apply decorators to “fake” functions used only for testing.
3. Cover all possible ways the decorators can be used.
 - a. Decorator parameters?
 - b. Inner function arguments?
 - c. Return values?
 - d. Identity?
 - e. Methods and classes?
 - f. Etc.

Example Decorator Tests

```
# Decorated Functions
```

```
@count_calls
```

```
def no_op():
```

```
    pass
```

```
@count_calls
```

```
def same(a):
```

```
    return a
```

Example Decorator Tests

```
# Decorated Functions
```

```
@count_calls  
def no_op():  
    pass
```

```
@count_calls  
def same(a):  
    return a
```

```
# Test Cases
```

```
def test_count_calls_0():  
    assert no_op.count == 0
```


Example Decorator Tests

```
# Decorated Functions
```

```
@count_calls  
def no_op():  
    pass
```

```
@count_calls  
def same(a):  
    return a
```

```
# Test Cases
```

```
def test_count_calls_0():  
    assert no_op.count == 0
```

```
def test_count_calls_3():  
    for _ in range(3):  
        no_op()  
    assert no_op.count == 3
```

Example Decorator Tests

```
# Decorated Functions
```

```
@count_calls
def no_op():
    pass
```

```
@count_calls
def same(a):
    return a
```

```
# Test Cases
```

```
def test_count_calls_0():
    assert no_op.count == 0
```

```
def test_count_calls_3():
    for _ in range(3):
        no_op()
    assert no_op.count == 3
```

```
def test_count_calls_with_args_and_return():
    answer = same('hello')
    assert answer == 'hello'
    assert same.count == 1
```

MORE DECORATORS



ABSOLUTELY FREE!

Common Decorators

<i>Decorator</i>	<i>Applies To</i>	<i>Purpose</i>
@classmethod	Methods	Makes method callable from class with class parameter
@staticmethod	Methods	Makes method callable from class without class parameter
@property	Methods	Adds getters and setters for attributes
@app.route	Functions	Flask: binds a function to a URL
@pytest.mark.parametrize	Functions	pytest: runs tests with different input combos

@classmethod and @staticmethod

```
class Greeter:
```

@classmethod and @staticmethod

```
class Greeter:
```

```
    @classmethod
    def hello(cls):
        name = cls.__name__
        print(f'hello from {name}')
```

```
>>> Greeter.hello()
hello from Greeter
```

@classmethod and @staticmethod

```
class Greeter:
```

```
    @classmethod
    def hello(cls):
        name = cls.__name__
        print(f'hello from {name}')
```

```
    @staticmethod
    def goodbye():
        print('goodbye')
```

```
>>> Greeter.hello()
hello from Greeter
```

```
>>> Greeter.goodbye()
goodbye
```

@classmethod and @staticmethod

```
class Greeter:
```

```
    @classmethod
    def hello(cls):
        name = cls.__name__
        print(f'hello from {name}')

    @staticmethod
    def goodbye():
        print('goodbye')
```

```
>>> Greeter.hello()
hello from Greeter
```

```
>>> Greeter.goodbye()
goodbye
```


The @property Decorator

```
class Accumulator:
```

The @property Decorator

```
class Accumulator:  
  
    def __init__(self):  
        self.count = 0
```

The @property Decorator

```
class Accumulator:  
  
    def __init__(self):  
        self.count = 0  
  
    def add(self, amount):  
        self.count += amount
```

The @property Decorator

```
class Accumulator:  
  
    def __init__(self):  
        self.count = 0  
  
    def add(self, amount):  
        self.count += amount  
  
    @property  
    def count(self):  
        return self._count
```

The @property Decorator

```
class Accumulator:

    def __init__(self):
        self.count = 0

    def add(self, amount):
        self.count += amount

    @property
    def count(self):
        return self._count

    @count.setter
    def count(self, value):
        if value < 0:
            raise ValueError('count must be >= 0')
        self._count = value
```

The @property Decorator

```
class Accumulator:

    def __init__(self):
        self.count = 0

    def add(self, amount):
        self.count += amount

    @property
    def count(self):
        return self._count

    @count.setter
    def count(self, value):
        if value < 0:
            raise ValueError('count must be >= 0')
        self._count = value
```

```
>>> a = Accumulator()
>>> a.count
0
```

The @property Decorator

```
class Accumulator:

    def __init__(self):
        self.count = 0

    def add(self, amount):
        self.count += amount

    @property
    def count(self):
        return self._count

    @count.setter
    def count(self, value):
        if value < 0:
            raise ValueError('count must be >= 0')
        self._count = value
```

```
>>> a = Accumulator()
>>> a.count
0

>>> a.add(5)
>>> a.count
5
```

The @property Decorator

```
class Accumulator:

    def __init__(self):
        self.count = 0

    def add(self, amount):
        self.count += amount

    @property
    def count(self):
        return self._count

    @count.setter
    def count(self, value):
        if value < 0:
            raise ValueError('count must be >= 0')
        self._count = value
```

```
>>> a = Accumulator()
>>> a.count
0

>>> a.add(5)
>>> a.count
5

>>> a.count = 2
>>> a.count = -3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File
"/Users/andy1pk247/Programming/automation-panda/py
gotham-tv/decorator.py", line 107, in count
    raise ValueError('Accumulator count must not
be negative')
ValueError: count must be >= 0
```


A Flask Example

Flask is a Web micro-framework written in Python. It enables you to write Web APIs easily with very little code.

A Flask Example

Flask is a Web micro-framework written in Python. It enables you to write Web APIs easily with very little code.

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello_world():
    return 'Hello, World!'
```

Code source: <https://flask.palletsprojects.com/en/1.1.x/quickstart/>

A `pytest` Example

pytest is a Pythonic test framework. It enables you to write tests as simple functions anywhere in your project.

A `pytest` Example

pytest is a Pythonic test framework. It enables you to write tests as simple functions anywhere in your project.

pytest supports parametrized tests!

A pytest Example

pytest is a Pythonic test framework. It enables you to write tests as simple functions anywhere in your project.

pytest supports parametrized tests!

```
def test_addition(a, b, c):  
    assert a + b == c
```

A pytest Example

pytest is a Pythonic test framework. It enables you to write tests as simple functions anywhere in your project.

pytest supports parametrized tests!

```
values = [  
    (1, 2, 3),  
    (0, 1, 1),  
    (5, -2, 3),  
    (3.14, 9.5, 12.64)  
]
```

```
def test_addition(a, b, c):  
    assert a + b == c
```

A pytest Example

pytest is a Pythonic test framework. It enables you to write tests as simple functions anywhere in your project.

pytest supports parametrized tests!

```
import pytest
```

```
values = [  
    (1, 2, 3),  
    (0, 1, 1),  
    (5, -2, 3),  
    (3.14, 9.5, 12.64)  
]
```

```
@pytest.mark.parametrize("a,b,c", values)  
def test_addition(a, b, c):  
    assert a + b == c
```



When should you use
decorators?

Use decorators for **aspects**.

Use decorators for **aspects**.

(special cross-cutting concerns)

Decorator Candidates

Good examples:

- Logging
- Profiling
- Input validation
- Retries
- Registries

Decorator Candidates

Good examples:

- Logging
- Profiling
- Input validation
- Retries
- Registries

Should the code “wrap” something else?

Decorator Candidates

Good examples:

- Logging
- Profiling
- Input validation
- Retries
- Registries

Bad examples:

- “Main” behaviors
- Complicated logic
- Heavy conditional logic
- Avoiding the wrapped function

Should the code “wrap” something else?

Decorator Candidates

Good examples:

- Logging
- Profiling
- Input validation
- Retries
- Registries

Should the code “wrap” something else?

Bad examples:

- “Main” behaviors
- Complicated logic
- Heavy conditional logic
- Avoiding the wrapped function

Is the code a wrapper or a candy bar?





Primer on Python Decorators

by Geir Arne Hjelle 186 Comments intermediate python

Tweet Share Email

Thanks!

Pandy Knight

Automation Panda

Developer Advocate at Applitools

