Decorators

Flaky APIs

You write this code to talk with an API:

```
import requests
# API endpoint
API_URL = 'https://example.com/api'

def get_items():
    return requests.get(API_URL + '/items')
```

Now imagine that 3rd-party API is flaky. Sometimes a request randomly breaks, then works when you make it again.

What do you do?

Retry

```
# Original:
def get_items():
    return requests.get(API_URL + '/items')

# New version. Automatically retry, so it's more reliable.
def get_items():
    MAX_TRIES = 3
    tries = 0
    resp = None
    while True:
        resp = requests.get(API_URL + '/items')
        if resp.status_code == 500 and tries < MAX_TRIES:
            tries += 1
            continue
        break
    return resp
```

What if we have many functions like `get_items()`?
How can you make this pattern reusable?

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.

Writing decorators

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

What it lets 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

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>"
```

@ 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
```

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?

And: How do you fix it?

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
```

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
```

Masking

```
def check_id(func):
    def wrapper(*args, **kwargs):
        print("ID of func: " + str(id(func)))
        return func(*args, **kwargs)
    print("ID of wrapper: " + str(id(wrapper)))
    return wrapper
```

```
>>> @check_id
... def f(x): return x * 3
ID of wrapper: 4429514272
>>> f(2)
ID of func: 4429514136
6
>>> id(f)
4429514272
```

Lab: Simple Decorators

Lab file: `decorators/decorators.py`

- In `labs` folder
- When you are done, study the solution - compare to what you wrote.