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!'
```

@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: " + 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: " + 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.)

Uppercase or not?

By convention, decorator names are lowercase.

And by convention, class names are uppercase.

Which wins?

Because how a decorator is implemented is an implementation detail, and may even change... I recommend you lowercase decorator names, even if they're classes.

```
# Lowercase class name, defying that convention.
class printlog:
    def __init__(self, func):
    # ...
```

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.

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` folder
- HINT: In `memoize_class.py`, make your wrapper accept just `*args`, not `**kwargs`.
- When you are done, study the solution - compare to what you wrote.
- ... and then optionally do `memoize_class_extra.py`