Python @Decorators and Digressions

https://noclevercode.svbtle.com/python-decorators-and-diversions

Decorators Are a Simple Mechanism

```
@a_decorator
def a_fuction(*args, **kwargs):
    pass
```

equals

```
a_function = a_decorator(a_function)
```

Decorators Are an Inconsistent Mechanism

```
@a_decorator(foo)
def a_fuction(*args, **kwargs):
    pass
```

equals

```
a_function = a_decorator(foo)(a_function)
```

@Decorators, Good God! What Are They Good For?

- logging & timing
- authorization
- self-registering objects
- mixins
- ifdefs
- rate limitations
- singletons
- caching
- translating results (e.g. units of measure)
- validating parameters and results

Stacking Decorators -- If You Feel Lucky

If every decorator in the stack quickly returns with a proxy object (e.g. a wrapper function), then they are relatively safe to stack

```
@timing
@logging
def do_something():
```

A timing decorator does not know that it is wrapping a logging decorator, so it will count the logging time as part of the call. Since the original caller will not get the result back until the logging completes, this is not inaccurate. What if the timing decorator is inside the logging decorator? It will measure the time the wrapped function took, not the time the caller was waiting

Stacking Decorators, cont

If the decorator sets attributes or injects behavior into the wrapped function, then it needs to be on the “bottom” of the stack so it affects the actual function rather than another wrapper

If the decorator moves the execution of the wrapped function into threads or distributes it, it had better be at the top of the stack so it executes all of the other wrappers in the same context

If a wrapper allows any runtime configurations (generally a Bad Move), then it, too, must be at the top of the stack

The Ugly Truth

This is what meta-programming looks like

```
def a_decorator(func):
    def wrapper(*args, **kwargs):
        print("Before decorated function")
        result = func(*args, **kwargs)
        print("After decorated function")
        return result

    return wrapper
```

It’s a lot of nested function definition and passing object references around

Objects In Disguise

As we are wrapping one object with another, we might want to make the wrapper look like the original

The **functools** library has a decorator named **@wraps** used inside decorator implementations (!) that copies the __name__, __doc__, and other attributes from the wrapped object into the wrapper

```
def a_decorator(func):
    @wraps(func)   #  <-----------------------
    def wrapper(*args, **kwargs):
        print("Before decorated function")
        result = func(*args, **kwargs)
        print("After decorated function")
        return result

    return wrapper
```

Decorator Naming

Because we use one part of Python (functions and classes) as part of the interface of another (annotations), sometimes the naming conventions of one don't fit with the other. To make a decorator implemented with a class look right, it should have a snake-case name even though class names use CorrectCase

```
class simple_cache:   # noqa
    def __init__(self, func):
        self.wrapped = func
    def __call__(self, *args, **kwargs):
        result = _get_cached_value(*args, **kwargs)
        if result is None:
            result = self.wrapped(*args, **kwargs)
            _set_cached_value(result, *args, **kwargs)
        return result
```

The Decorator Mechanism: Mode 1 and Mode 2

	Mode 1 **@decorator**	**Mode 2** **@decorator()**
decorator function	● The function receives the object to decorate ● It returns a callable object that wraps the original, decorated object	● The function receives the decorator parameters ● It returns a callable object that receives the object to decorate ● That object returns a callable object that wraps the original, decorated object
decorator class	● The new instance of the decorator class's `__init__` method receives the original object to decorate ● That new instance's `__call__` method acts as the callable wrapper	● The new instance of the decorator class's `__init__` method receives the decorator parameters ● Its `__call__` method receives the original object to decorate ● and returns a callable object that wraps the original, decorated object

The Caller Sets the Mode, Not the Implementation

The expectation of parameters on the implementing
decorator function/class, or lack thereof
MAKES NO DIFFERENCE

You can write a decorator that can handle both cases or just let the inevitable type error teach your users what to do

The Difference: Classes, Instances, and Functions

Not much, actually

They all have attributes (which they can copy from other objects), and they return something when called:

`x = MyClass()`	execute the `__init__` method
`y = x()`	execute the `call` method
`z = a_function()`	