Function Decorators

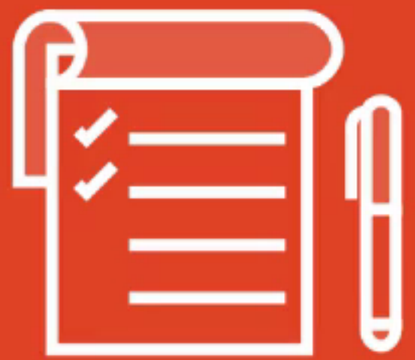
Austin Bingham
COFOUNDER - SIXTY NORTH

@austin_bingham

Robert Smallshire
COFOUNDER - SIXTY NORTH

@robsmallshire

Overview



Function decorators

Applying decorators to functions

Using any callable as a decorator

Applying multiple decorators

Semantics of such application

Standard library support for decorators

Parameterized decorators

Decorators

Modify or enhance an existing function in a non-intrusive and maintainable way

Implemented as a callable that accepts a callable and returns a callable

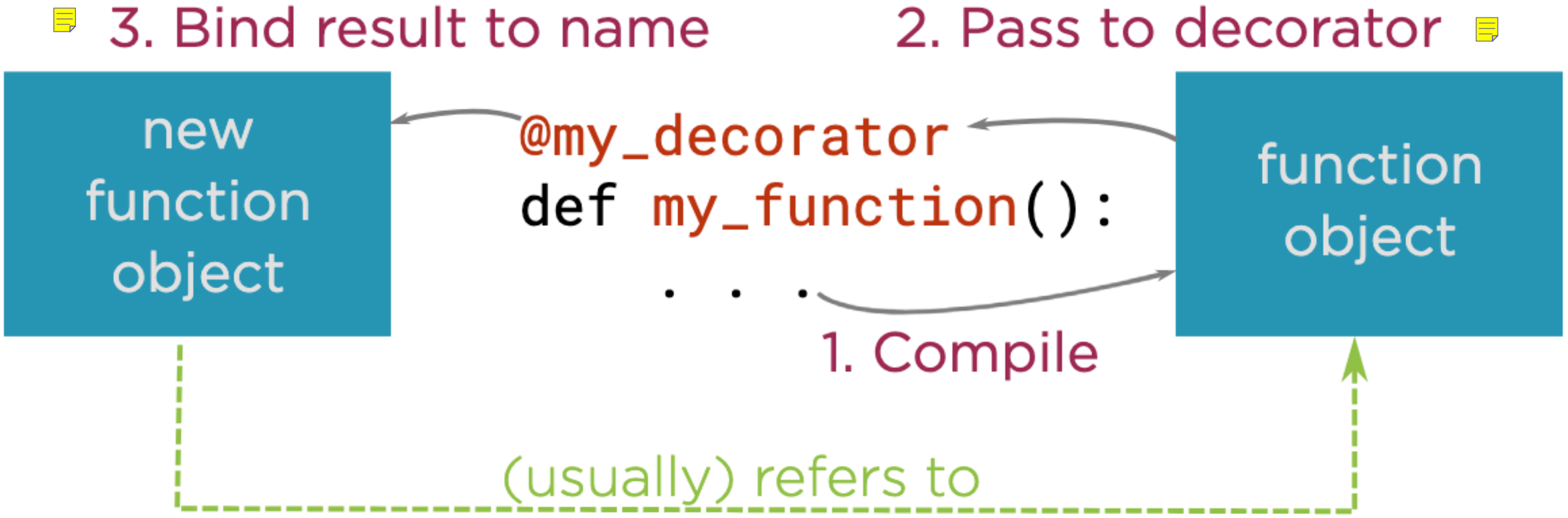
"A function accepting a function and returning a function"

Decorator Syntax

Applies decorator

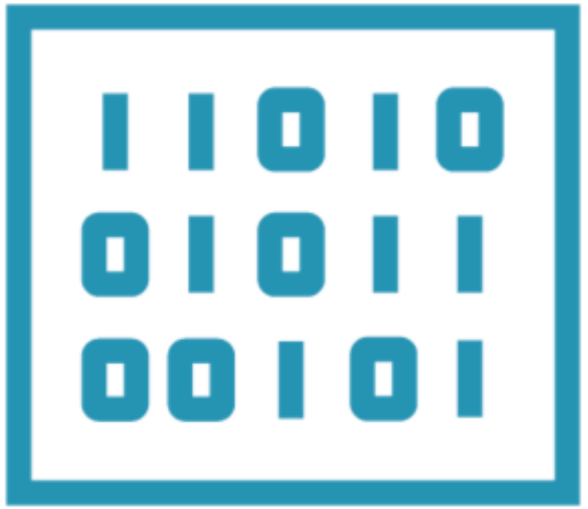
```
@my_decorator
def my_function():
    . . .
```

Decorator Application



Decorators allow you to modify existing functions without changing their definition.

Callers don't need to change when decorators are applied.



Ensure that functions only return ASCII strings

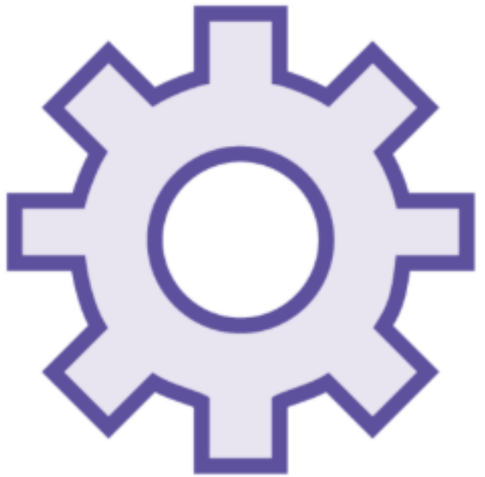
We could use `ascii()` to do that

Intrusive and hard to maintain

Decorator Example

```
>>> def escape_unicode(f):
...     def wrap(*args, **kwargs):
...         x = f(*args, **kwargs)
...         return ascii(x)
...     return wrap
...
>>> def northern_city():
...     return 'Tromsø'
...
>>> print(northern_city())
Tromsø
>>> @escape_unicode
... def northern_city():
...     return 'Tromsø'
...
>>> print(northern_city())
'Troms\xf8'
>>>
```

Decorator Mechanics



Decorators take a callable argument and return a callable

In this example, the callable we return is the local function `wrap()`

`wrap()` uses a closure to access `f` after `escape_unicode()` returns

Classes as Decorators

Classes as Decorators

1. Classes are **callable objects**
2. Functions decorated with a class are replaced by an **instance of the class**
3. These instances **must themselves be callable**

We can decorate with a class as long as instances of the class implement `__call__()`.

Classes as Decorators

```
>>> class CallCount:
...     def __init__(self, f):
...         self.f = f
...         self.count = 0
...     def __call__(self, *args, **kwargs):
...         self.count += 1
...         return self.f(*args, **kwargs)
...
>>> @CallCount
... def hello(name):
...     print('Hello, {}!'.format(name))
...
>>> hello('Fred')
Hello, Fred!
>>> hello('Wilma')
Hello, Wilma!
>>> hello('Betty')
Hello, Betty!
>>> hello('Barney')
Hello, Barney!
>>> hello.count
4
>>>
```

Instances as Decorators

Instances as Decorators

Python calls an instance's `__call__()` when it's used as a decorator

`__call__()`'s return value is used as the new function

Create groups of callables that you can dynamically control as a group

Instances as Decorators

```
>>> l = [1, 2, 3]
>>> l = rotate_list(l)
Calling <function rotate_list at 0x10eb4cf70>
>>> l
[2, 3, 1]
>>> l = rotate_list(l)
Calling <function rotate_list at 0x10eb4cf70>
>>> l
[3, 1, 2]
>>> l = rotate_list(l)
Calling <function rotate_list at 0x10eb4cf70>
>>> l
[1, 2, 3]
>>> tracer.enabled = False
>>> l = rotate_list(l)
>>> l
[2, 3, 1]
>>> l = rotate_list(l)
>>> l
[3, 1, 2]
>>> l = rotate_list(l)
>>> l
[1, 2, 3]
>>>
```

Which to Use?

Power and flexibility

The available callable objects give you a lot of flexibility when implementing decorators.

Design space

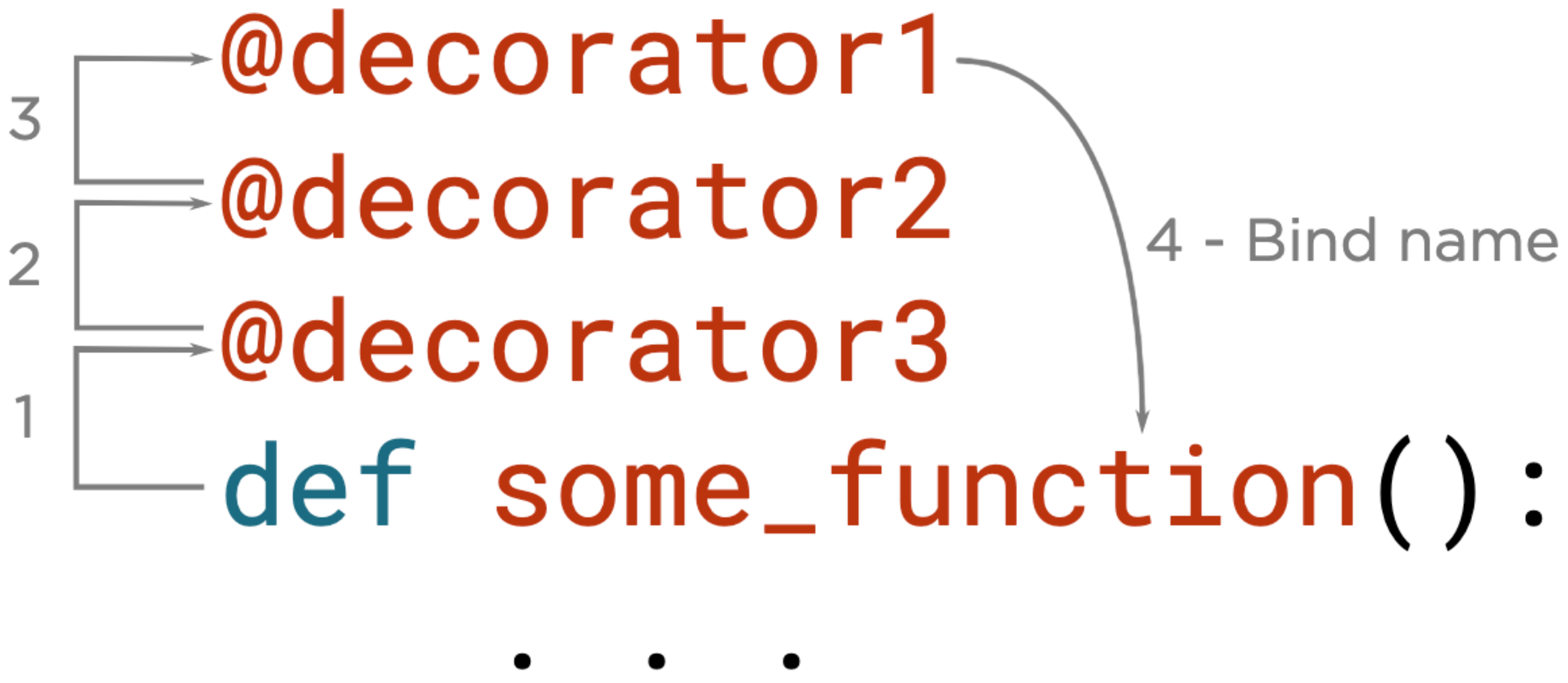
Deciding which to use will depend a great deal upon what you're trying to accomplish.

Experimentation

Use experiments and small examples to come to grips with different decorator implementations.

Multiple Decorators

Multiple Decorators



Multiple Decorators

```
...
>>> tracer = Trace()
>>> @tracer
... @escape_unicode
... def norwegian_island_maker(name):
...     return name + 'øy'
...
>>> norwegian_island_maker('Llama')
Calling <function escape_unicode.<locals>.wrap at 0x10687a1f0>
'Llama\\xf8y'
>>> norwegian_island_maker('Python')
Calling <function escape_unicode.<locals>.wrap at 0x10687a1f0>
'Python\\xf8y'
>>> norwegian_island_maker('Troll')
Calling <function escape_unicode.<locals>.wrap at 0x10687a1f0>
'Troll\\xf8y'
>>> tracer.enabled = False
>>> norwegian_island_maker('Llama')
'Llama\\xf8y'
>>> norwegian_island_maker('Python')
'Python\\xf8y'
>>> norwegian_island_maker('Troll')
'Troll\\xf8y'
>>>
```

Decorating Methods

Decorating Methods

```
>>> class IslandMaker:
...     def __init__(self, suffix):
...         self.suffix = suffix
...     @tracer
...     def make_island(self, name):
...         return name + self.suffix
...
>>> im = IslandMaker(' Island')
>>> im.make_island('Python')
Calling <function IslandMaker.make_island at 0x10b4bc1f0>
'Python Island'
>>> im.make_island('Llama')
Calling <function IslandMaker.make_island at 0x10b4bc1f0>
'Llama Island'
>>>
```

Callable Metadata

Losing Metadata

There are subtle problems with our use of decorators

We lose important metadata from the original callable

Losing Metadata

```
>>> import functools
>>> def noop(f):
...     @functools.wraps(f)
...     def noop_wrapper():
...         return f()
...     return noop_wrapper
...
>>> @noop
... def hello():
...     "Print a well-known message."
...     print('hello world!')
...
>>> help(hello)
Help on function hello in module __main__:

hello()
    Print a well-known message.
(END)

>>> hello.__name__
'hello'
>>> hello.__doc__
'Print a well-known message.'
>>>
```

We can copy `__name__` and `__doc__` from our wrapped function to our wrapper function.

`functools.wraps()`

Replace decorator metadata with that of the decorated callable.

It is a decorator that you apply to your wrapper function.

It does the hard work for you!

A More Complex Example

Validating Arguments

```
>>> def check_non_negative(index):
...     def validator(f):
...         def wrap(*args):
...             if args[index] < 0:
...                 raise ValueError(
...                     'Argument {} must be non-negative.'.format(index))
...             return f(*args)
...         return wrap
...     return validator
...
>>> @check_non_negative(1)
... def create_list(value, size):
...     return [value] * size
...
>>> create_list('a', 3)
['a', 'a', 'a']
>>> create_list(123, -6)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 5, in wrap
ValueError: Argument 1 must be non-negative.
>>>
```

Closing Thoughts

Decorators are a powerful tool in Python programming

Decorators are very widely used

Like other language features, it's possible to overuse them

Use them when they improve maintainability, add clarity, and simplify your code

Avoid using them for the sake of using them

Summary

Decorators modify existing callables extrinsically

Apply decorators with "@"

Callables that accept a callable and return a callable

Implement them with functions, classes, or instances

Apply multiple decorators to a callable

Recursive application of basic rules

`functools.wraps()`

"Parameterized decorators"