Closures and Decorators

Austin Bingham
@austin_bingham
austin@sixty-north.com

Presenter

Robert Smallshire
@robsmallshire
rob@sixty-north.com

pluralsight
hardcore dev and IT training

python™

def

define new functions, executed at runtime

Local functions

```
def func():
    x = 1
    y = 2
    return x + y
```

Local functions

```python
def func():
    x = 1
    y = 2
    return x + y
```

```python
def func():
    def local_func():
        a = 'hello, '
        b = 'world'
        return a + b

    x = 1
    y = 2
    return x + y
```

Local functions

```python
def func():
    x = 1
    y = 2
    return x + y
```

```python
def func():
    def local_func():
        a = 'hello, '
        b = 'world'
        return a + b

    x = 1
    y = 2
    return x + y
```

python™

LEGB rule

local, enclosing, global, built-in

LEGB Rule

```
PI = TAU / 2

def func(x):
    def local_func(n):
        a = 'hello, '
        return a + n

    y = 2
    return x + y
```

module.py

LEGB Rule

```
PI = TAU / 2

def func(x):
    def local_func(n):
        a = 'hello, '
        return a + n

    y = 2
    return x + y
```

module.py

LEGB Rule

```
PI = TAU / 2

def func(x):
    def local_func(n):
        a = 'hello, '
        return a + n

    y = 2
    return x + y
```

module.py

LEGB Rule

```
PI = TAU / 2

def func(x):
    def local_func(n):
        a = 'hello, '
        return a + n

    y = 2
    return x + y
```

module.py

Local functions

- Useful for specialized, one-off functions
- Aid in code organization and readability
- Similar to lambdas, but more general
  - May contain multiple expressions
  - May contain statements

Returning functions

```
def outer():
    def inner():
        print('inner')
    inner()
```

Returning functions

```
def outer():
    def inner():
        print('inner')
    return inner

i = outer()
i()
```

python™

First-class Functions

functions can be treated like any other object

Closures

```
def outer():
    x = 3

    def inner(y):
        return x + y

    return inner

i = outer()
```

Closures

```
def outer():
    x = 3

    def inner(y):
        return x + y

    return inner

i = outer()
```

Closures

```
def outer():
    x = 3

    def inner(y):
        return x + y

    return inner

i = outer()
```

Closures

```
x = 3

def inner(y):
    return x + y

i = outer()
```

python™

Closures

maintain references to objects from earlier scopes

python™

Function factory

function that returns new, specialized functions

python™

LEGB does **not apply** when making **new bindings**.

python™

global

introduce names from global namespace
into the local namespace

python™

nonlocal

introduce names from the enclosing namespace
into the local namespace

You get a SyntaxError
if the name doesn't
exist.

python™

decorators

modify or enhance functions without changing their definition

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

Decorators

```
@my_decorator
def my_function(x, y):
    return x + y
```

Decorators

```
@my_decorator
def my_function(x, y):
    return x + y
```

FUNCTION OBJECT

Decorators

```
y):
```

FUNCTION OBJECT

```
def my_decorator(f):
    . . .
    return new_f
```

Decorators

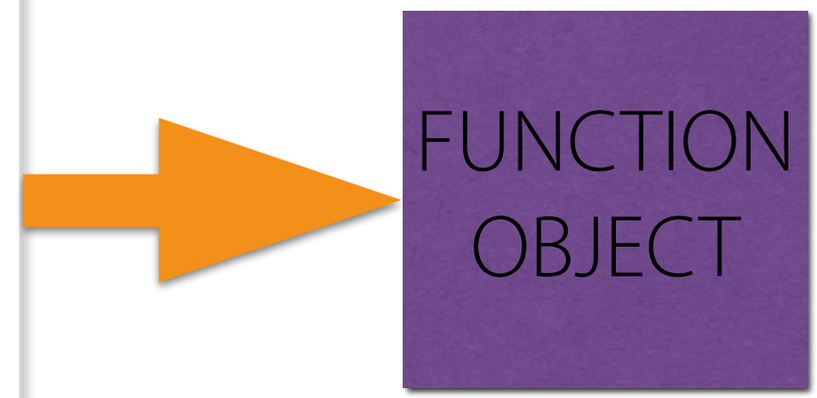
FUNCTION OBJECT

```
def my_decorator(f):
    . . .
    return new_f
```

FUNCTION OBJECT

Decorators

```
@my_decorator
def my_function(x, y):
    return x + y
```



Decorators

```
def my_function(x, y):
```

FUNCTION OBJECT

Decorators

- Replace, enhance, or modify existing functions
- Does not change the original function definition
- Calling code does not need to change
- Decorator mechanism uses the modified function's original name

Decorators

```
def vegetable():
    return 'blomkål'

def animal():
    return 'bjørn'

def mineral():
    return 'stål'
```

Decorators

```
def vegetable():
    return ascii('blomkål')

def animal():
    return ascii('bjørn')

def mineral():
    return ascii('stål')
```

Decorators

```
def vegetable():
    return ascii('blomkål')

def animal():
    return ascii('bjørn')

def mineral():
    return ascii('stål')
```

Not very scalable.

Not very maintainable.

Decorators

We've seen **functions as decorators**…

…but **other objects** can be decorators **as well**.

Classes as decorators

```
class MyDec:
    def __init__(self, f):
        . . .

    def __call__(self):
        . . .
```

```
@MyDec
def func():
    . . .
```

Classes as decorators

Classes are callable…

```
class MyDec:
    def __init__(self, f):
        . . .

    def __call__(self):
        . . .
```

```
@MyDec
def func():
    . . .
```

Classes as decorators

Classes are callable…

```
class MyDec:
    def __init__(self, f):
        . . .

    def __call__(self):
        . . .
```

…so they can be used as decorators.

```
@MyDec
def func():
    . . .
```

Classes as decorators

Applying a class decorator creates a new instance…

```
class MyDec:
    def __init__(self, f):
        . . .

    def __call__(self):
        . . .
```

```
@MyDec
def func():
    . . .
```

Classes as decorators

Applying a class decorator creates a new instance…

```
class MyDec:
    def __init__(self, f):
        . . .

    def __call__(self):
        . . .
```

```
@MyDec
def func():
    . . .
```

…so the instance must be callable.

Instances as decorators

Decorating with an instance *calls* the instance.

```
class AnotherDec:
    def __call__(self, f):
        def wrap():
            . . .
        return wrap
```

```
@AnotherDec()
def func():
    . . .
```

Instances as decorators

```
class AnotherDec:
    def __call__(self, f):
        def wrap():
            . . .
        return wrap
```

```
@AnotherDec()
def func():
    . . .
```

Decorating with an instance calls the instance.

The return value of **`__call__`** must be callable.

Multiple decorators

```
@decorator1
@decorator2
@decorator3
def some_function():
    . . .
```

Multiple decorators

```
@decorator1
@decorator2
@decorator3
def some_function():
    . . .
```

Multiple decorators

```
@decorator1
@decorator2
def some_function():
    . . .
```

Multiple decorators

```
@decorator1
@decorator2
def some_function():
    . . .
```

Multiple decorators

```
@decorator1
def some_function():
    . . .
```

Multiple decorators

```
@decorator1
def some_function():
    . . .
```

Multiple decorators

```
def some_function():
    . . .
```

python™

functools.wrap()

Naive decorators can lose important metadata.

python™

functools.wraps()

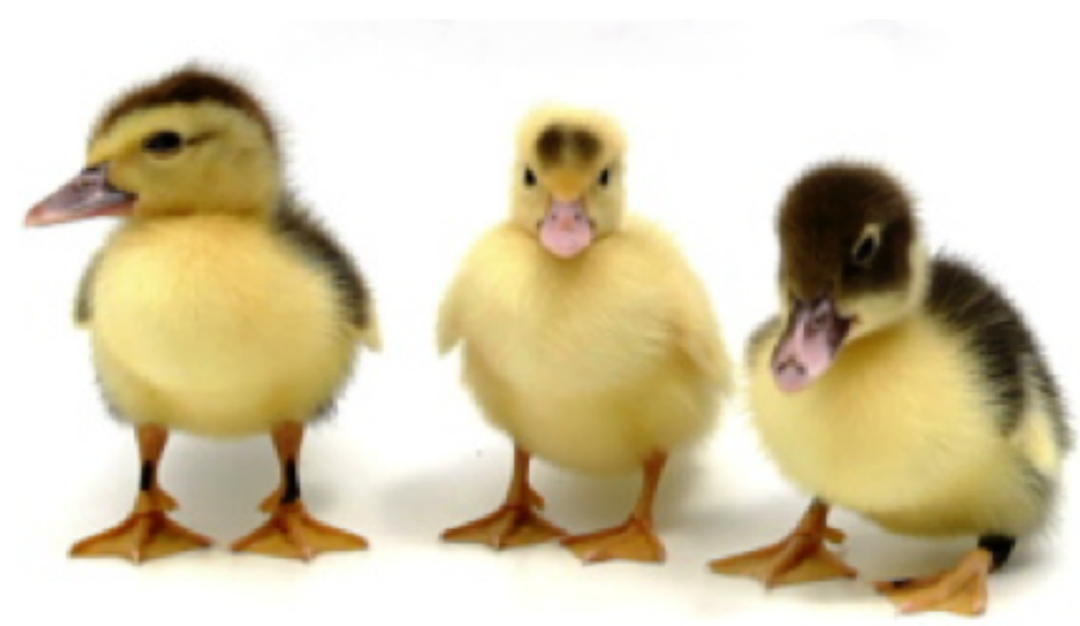
properly update metadata on wrapped functions

Decorators

- Decorators are a powerful tool
- Decorators are widely used in Python
- It's possible to overuse decorators; be mindful
- They can improve maintainability, increase clarity, and reduce complexity

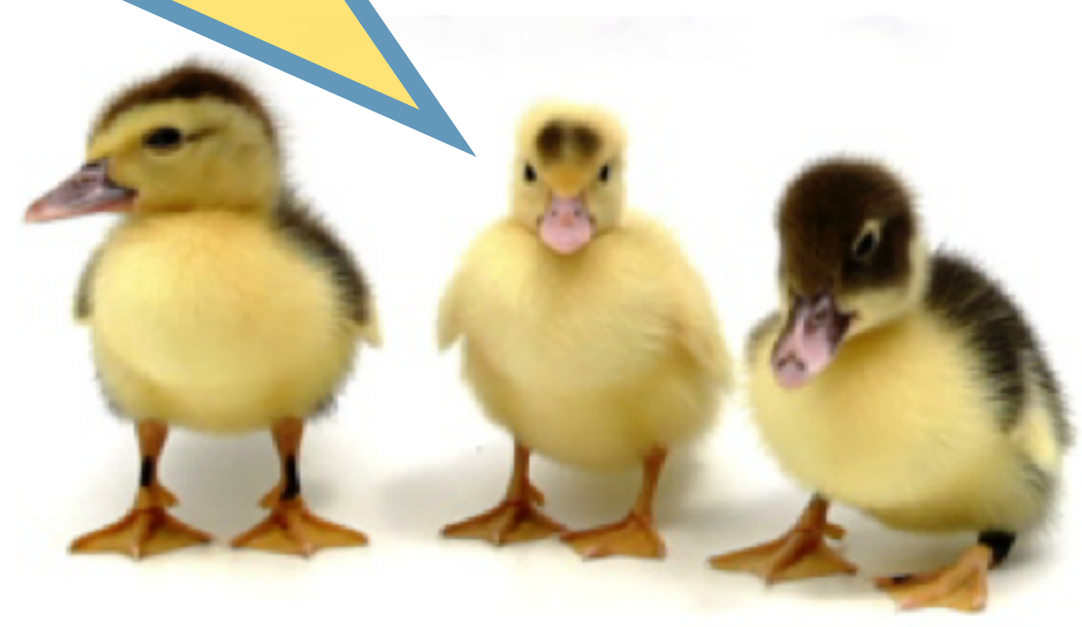
Duck Tails

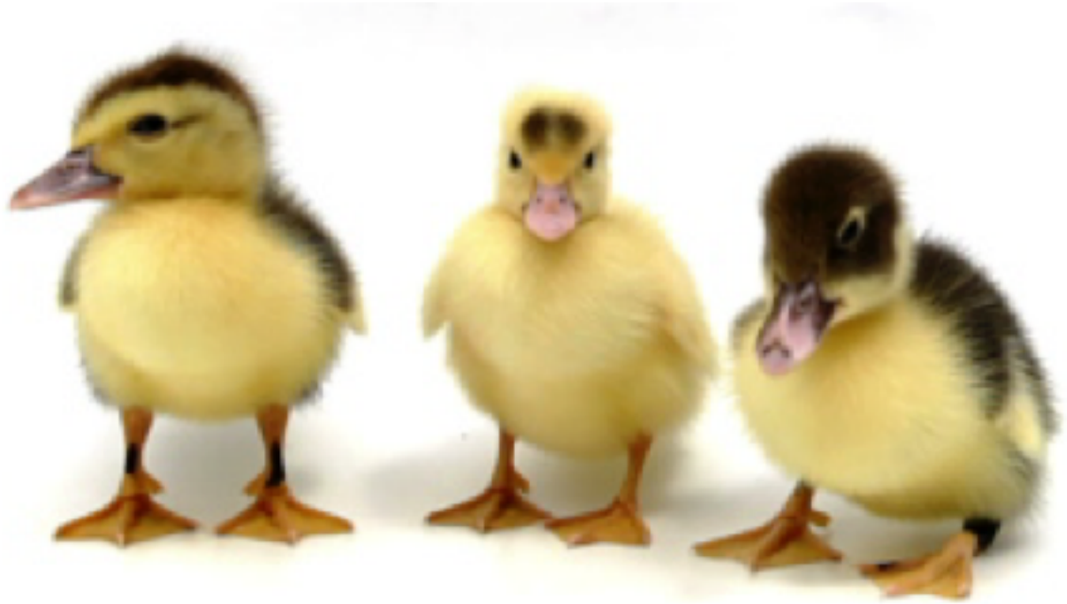
Factory functions which produce
decorator functions which make
wrapper functions which wrap functions
(with added closures!)



Duck Tails

WHEW!





Closures and Decorators

```
def function(a, b):
    print("A function")

    def local_function(x, y):
        print("A local function")
        return x * y + a * b

    return local_function
```

```
>>> p = function(5, 7)
A function
>>> p
<function function.<locals>.local_function at 0x10299f320>
>>> p.__closure__
(<cell at 0x1029bc8d8: int object at 0x100233120>,
 <cell at 0x1029bc980: int object at 0x100233160>)
```

Closures and Decorators

```
def log_to(stream):
    def logging_decorator(func):

        @wraps(func)
        def logging_wrapper(*args, **kwargs):
            print(func.__name__ + " was called", file=stream)
            return func(*args, **kwargs)

        return logging_wrapper

    return logging_decorator
```

```
@second_decorator
@logging_decorator
def some_function(x):
    """A decorated function"""
    return x * x
```

```
>>> print(some_function.__name__)
some_function
>>> print(some_function.__doc__)
A decorated function
```

```
@log_to(sys.stderr)
def another_function(x):
    """A decorated function"""
    return x + x
```