DECORATORS

Part 1

Decorators

Recall the simple closure example we did which allowed to us to maintain a count of how many times a function was called:

```
def counter(fn):
    count = 0
    def inner(*args, **kwargs):
        nonlocal count
        count += 1
        print('Function {0} was called {1} times'.format(fn.__name__, count)
        return fn(*args, **kwargs)
    return inner
```

using `*args, **kwargs` means we can call any function `fn` with any combination of positional and keyword-only arguments

```
def add(a, b=0):
    return a + b
```

```
add = counter(add)
```

```
result = add(1, 2)
```

→ Function add was called 1 times
→ `result = 3`

We essentially modified our `add` function by wrapping it inside another function that added some functionality to it

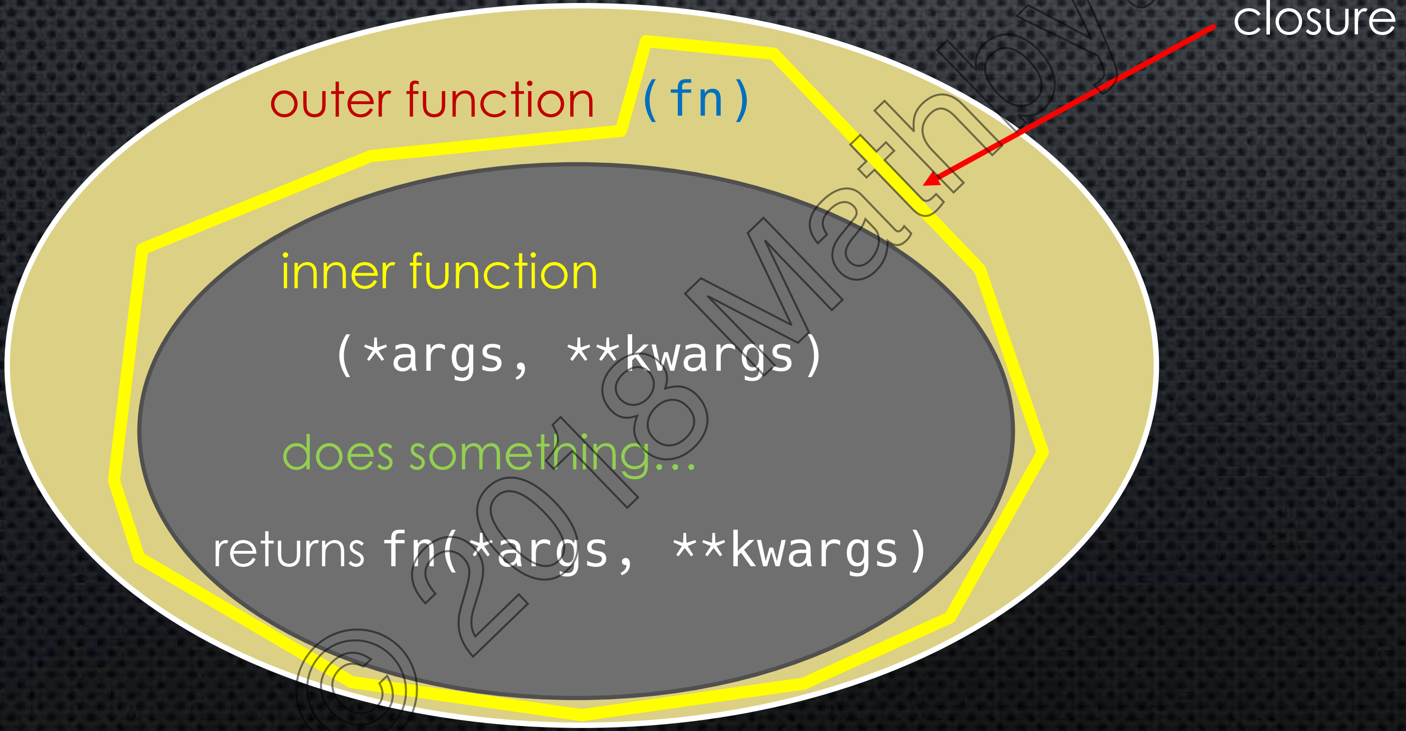
We also say that we decorated our function `add` with the function `counter`

And we call `counter` a decorator function

Decorators

In general a decorator function:

- takes a function as an argument
- returns a closure
- the closure usually accepts any combination of parameters
- runs some code in the inner function (closure)
- the closure function calls the original function using the arguments passed to the closure
- returns whatever is returned by that function call



Decorators and the @ Symbol

In our previous example, we saw that `counter` was a decorator

and we could decorate our `add` function using: `add = counter(add)`

In general, if `func` is a decorator function, we decorate another function `my_func` using:

```
my_func = func(my_func)
```

This is so common that Python provides a convenient way of writing that:

```
@counter
def add(a, b):
    return a + b
```

is the same as writing

```
def add(a, b):
    return a + b

add = counter(add)
```

```
@func
def my_func(…):
    …
```

is the same as writing

```
def my_func(…):
    …

my_func = func(my_func)
```

Introspecting Decorated Functions

Let's use the same count decorator.

```
def counter(fn):
    count = 0
    def inner(*args, **kwargs):
        nonlocal count
        count += 1
        print('{0} was called {1} times'.format(fn.__name__, count))
        return fn(*args, **kwargs)
    return inner
```

```
@counter
def mult(a, b, c=1):
    """
        returns the product of three values
    """
    return a * b * c
```

remember we could equally have written:
`mult = counter(mult)`

`mult.__name__` → `inner` not `mult`

`mult`'s name "changed" when we decorated it
they are not the same function after all

`help(mult)` → Help on function inner in module __main__:
inner(*args, **kwargs)

We have also "lost" our docstring,
and even the original function signature

Even using the `inspect` module's `signature` does not yield better results

One approach to fixing this

We could try to fix this problem, at least for the docstring and function name as follows:

```
def counter(fn):
    count = 0
    def inner(*args, **kwargs):
        nonlocal count
        count += 1
        print('Function {0} was called {1} times'.format(fn.__name__, count)
        return fn(*args, **kwargs)
    inner.__name__ = fn.__name__
    inner.__doc__ = fn.__doc__
    return inner
```

But this doesn't fix losing the function signature – doing so would be quite complicated

Instead, Python provides us with a special function that we can use to fix this

The functools.wraps function

The functools module has a wraps function that we can use to fix the metadata of our inner function in our decorator

```
from functools import wraps
```

In fact, the wraps function is itself a decorator

but it needs to know what was our "original" function – in this case fn

```
def counter(fn):
    count = 0
    def inner(*args, **kwargs):
        nonlocal count
        count += 1
        print(count)
        return fn(*args, **kwargs)
    inner = wraps(fn)(inner)
    return inner
```

```
def counter(fn):
    count = 0
    @wraps(fn)
    def inner(*args, **kwargs):
        nonlocal count
        count += 1
        print(count)
        return fn(*args, **kwargs)
    return inner
```

```
def counter(fn):
    count = 0
    @wraps(fn)
    def inner(*args, **kwargs):
        nonlocal count
        count += 1
        print(count)
        return fn(*args, **kwargs)
    return inner
```

```
@counter
def mult(a:int, b:int, c:int=1):
    """
        returns the product of three values
    """
    return a * b * c
```

```
help(mult)    → Help on function mult in module __main__:
                 mult(a:int, b:int, c:int=1)
                     returns the product of three values
```

And introspection using the `inspect` module works as expected:

```
inspect.signature(mult)    → <Signature (a:int, b:int, c:int=1)>
```

You don't have to use `@wraps`, but it will make debugging easier!

Code