DECORATORS

PART 2

Decorator Parameters

In the previous videos we saw some built-in decorators that can handle some arguments:

```
@wraps(fn)
def inner():
    …
```

```
@lru_cache(maxsize=256)
def factorial(n):
    …
```

function call

This should look quite different from the decorators we have been creating and using:

```
@timed
def fibonacci(n):
    …
```

no function call

The timed decorator

```
def timed(fn):
    from time import perf_counter

    def inner(*args, **kwargs):
        total_elapsed = 0
        for i in range(10):
            start = perf_counter()
            result = fn(*args, **kwargs)
            total_elapsed += (perf_counter() - start)
        avg_elapsed = total_elapsed / 10
        print(avg_elapsed)
        return result
    return inner
```

hardcoded value 10

```
@timed
def my_func():
    …
```

OR

```
my_func = timed(my_func)
```

One Approach

extra parameter

free variable

```
def timed(fn, reps):
    from time import perf_counter

    def inner(*args, **kwargs):
        total_elapsed = 0
        for i in range(reps):
            start = perf_counter()
            result = fn(*args, **kwargs)
            total_elapsed += (perf_counter() - start)
        avg_elapsed = total_elapsed / reps
        print(avg_elapsed)
        return result
    return inner
```

```
my_func = timed(my_func, 10)
```

```
@timed(10)
def my_func():
    …
```

Rethinking the solution

```
@timed
def my_func():
    …
```

```
my_func = timed(my_func)
```

So, `timed` is a function that returns that `inner` closure that contains our original function

In order for this to work as intended:

```
@timed(10)
def my_func():
    …
```

`timed(10)` will need to return our original `timed` decorator when called

```
dec = timed(10)
```

`timed(10)` returns a decorator

```
@dec
def my_func():
    …
```

and we decorate our function with `dec`

Nested closures to the rescue!

```
def outer(reps):

    def timed(fn):
        from time import perf_counter

        def inner(*args, **kwargs):
            total_elapsed = 0
            for i in range(reps):
                start = perf_counter()
                result = fn(*args, **kwargs)
                total_elapsed += (perf_counter() - start)
            avg_elapsed = total_elapsed / reps
            print(avg_elapsed)
            return result
        return inner

    return timed
```

our original decorator

free variable bound to reps in outer

calling outer(n) returns our original decorator with reps set to n (free variable)

```
my_func = outer(10)(my_func)
```

✔

OR

```
@outer(10)
def my_func():
    …
```

✔

Decorator Factories

The `outer` function is not itself a decorator

instead it returns a decorator when called

and any arguments passed to `outer` can be referenced (as free variables) inside our decorator

We call this `outer` function a decorator factory function

(it is a function that creates a new decorator each time it is called)

And finally…

To wrap things up, we probably don't want out decorator call to look like:

```
@outer(10)
def my_func():
    …
```

It would make more sense to write it this way:

```
@timed(10)
def my_func():
    …
```

All we need to do is change the names of the `outer` and `timed` functions

```
def timed(reps):

    def dec(fn):
        from time import perf_counter


        @wraps(fn)
        def inner(*args, **kwargs):
            total_elapsed = 0
            for i in range(reps):
                start = perf_counter()
                result = fn(*args, **kwargs)
                total_elapsed += (perf_counter() - start)
            avg_elapsed = total_elapsed / reps
            print(avg_elapsed)
            return result
        return inner
    return dec

@timed(10)
def my_func():
    …
```

def timed(reps): ← this was outer

def dec(fn): ← this was timed

@wraps(fn) ← we can still use @wraps

Code