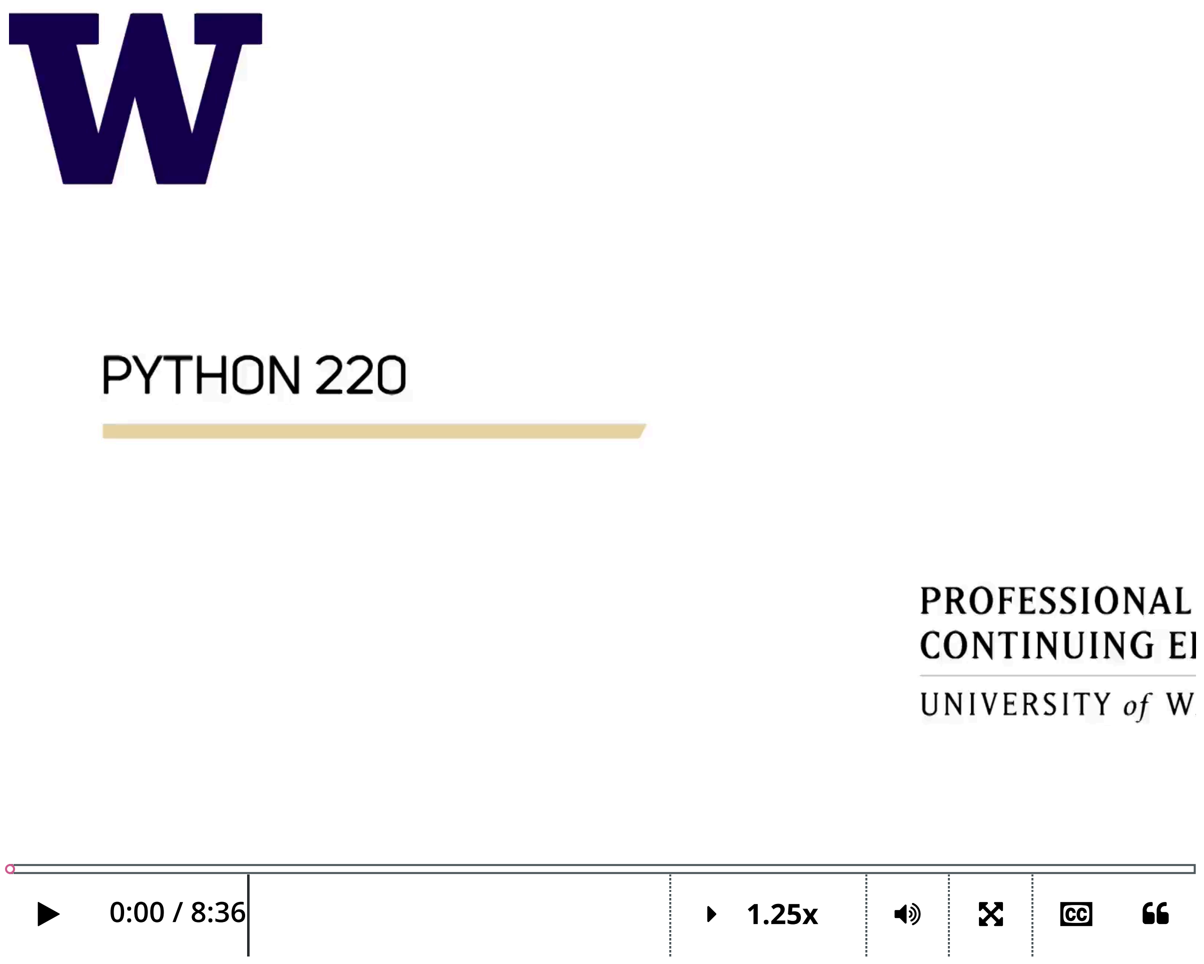
Part 1: Decorators

Bookmark this page

Part 1: Decorators

Decorators

Start of transcript. Skip to the end.



[MUSIC PLAYING]

All right.

We've seen closures.

And now we're going to look at a related idea that I think is really neat.

And it's called decorators.

In preparation, I'm going to throw up something that's really simple.

Basically, we're going to look at the way Python can

rebind references, rebind variables.

Functions are things that generate values based on arguments. In Python, functions are first-class objects. This means that you can bind names to them, pass them around, etc., just like other objects. Thanks to this you can write functions that take functions as arguments or return functions as values.

```
def substitute(a_function):
    def new_function(*args, **kwargs):
        return "I'm not that other function"
    return new_function
```

There are many things you can do with a simple pattern like this, so many, that we give it a special name: a Decorator.
“A decorator is a function that takes a function as an argument and returns a function as a return value.”

That’s nice, but why is it useful? Imagine you are trying to debug a module with a number of functions like this one:

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

You want to see when each function is called, with what arguments and
with what result. So you rewrite each function as follows:

```
def add(a, b):
    print("Function 'add' called with args: {}, {}".format(a, b) )
    result = a + b
    print("\tResult --> {}".format(result))
    return result
```

That is not particularly nice, especially if you have lots of functions
in your module. Now imagine we defined the following, more generic *decorator*:

```
def logged_func(func):
```

```
def logged_func(func):
    def logged(*args, **kwargs):
        print("Function {} called".format(func.__name__))
        if args:
            print("\twith args: {}".format(args))
        if kwargs:
            print("\twith kwargs: {}".format(kwargs))
        result = func(*args, **kwargs)
        print("\t Result --> {}".format(result))
        return result
    return logged
```

We could then make logging versions of our module functions.

```
logging_add = logged_func(add)
```

Then, where we want to see the results, we can use the logged version:

```
In []: logging_add(3, 4)
Function 'add' called
    with args: (3, 4)
     Result --> 7
Out[]: 7
```

This is nice, but we must now call the new function wherever we originally called the old one. It would be nicer if we could just call the old function and have it log. Remembering that you can easily rebind symbols in Python using simple assignment statements leads to this form:

```
def logged_func(func):
    # implemented above

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

add = logged_func(add)
```

And now you can simply use the code you've already written and calls to add will be logged:

```
In []: add(3, 4)
Function 'add' called
    with args: (3, 4)
     Result --> 7
Out[]: 7
```