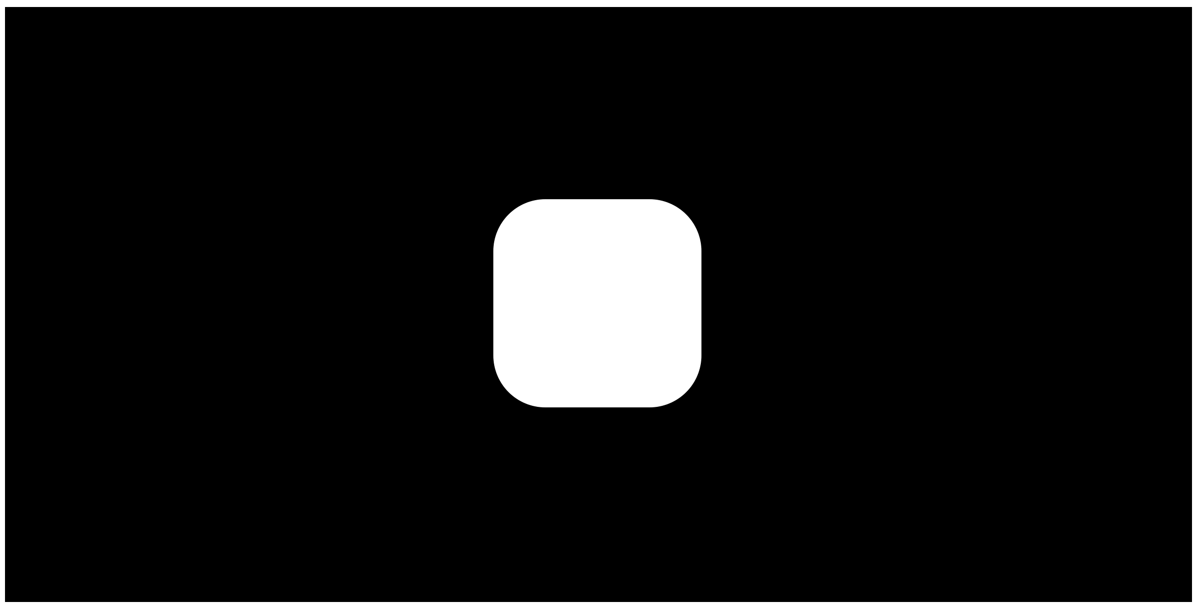
Part 2: Currying

Bookmark this page

Part 2: Currying

Currying



Start of transcript. Skip to the end.

[MUSIC PLAYING]

Now we're going to talk a little bit about currying,

and we're going to touch into the functools module.

Currying is a method by which--

let me start with the objective.

The objective is to reduce the arity, the number of arguments in a function

"Currying" is a special case of closures:

The idea behind currying is that you may have a function with a number of parameters, and you want to make a specialized version of that function with a couple parameters pre-set.

Let's go back to the multiplier example we used when learning closures:

```
def multiplier(x):
    return x * 3
```

As before, we can use closures to create a multiplier factory, one that requires very little code to create new multipliers that do not necessarily return x times 3 all the time. Another option, if you don't feel like using closures, would be to modify the *multiplier* function to take a second parameter:

```
def multiplier(x, n=3):
    return x * n
```

This one would still be able to work in exactly the same way as the original function, but now it also allows you to override the default multiplication by 3 with a different factor:

```
multiplier(4) # The result will be 4 * 3 = 12
multiplier(4, 2) # The result will be 4 * 2 = 8 since the default was overridden
```

Still a bit clunky – it forces us to keep passing the value of the factor, even if it's not changing (unless the factor happens to have the default value of 3). And there are places, like map that require a function that takes only one argument!

Real-world-example

What if I could create a function, on the fly, that had a particular factor "baked in"?

Enter Currying – Currying is a technique where you reduce the number of parameters a function takes, creating a specialized function with one or more of the original parameters set to a particular value. Here is that technique, applied to the multiplier problem:

```
def get_multiplier(n = 3):
    def multiplier(x)
        return x * n
    return multiplier
```

Now, in this case, we still had to define *multiplier* inside of *get_multiplier*, which presented no major issues, but could become more cumbersome in the next scenario. Imagine you have a Python module called *introductions.py*. Inside of that module, there is a single function called *introduce_person()*:

```
# introductions.py
def introduce_person(name, age, job, location):
    return "This is %s, a %d-year-old %s living in %s" % (name, age, job, location)
```

The introduce_person method allows you to create a basic string to introduce a person. It takes four parameters.

```
>>> from introductions import introduce_person
>>> introduce_person("Elisa", 28, "engineer", "Portland")
'This is Elisa, a 28-year-old engineer living in Portland'
```

This works well, but it might seem less convenient if you have a list of people you need to introduce, all with the same job (for example, 'student'), all with the same age (maybe 4th grade children) who all happen to be 10 years old and all living in the same city. Only the names are changing.

To take advantage of the parameters these group of people have in common, we could create a curried function, just like we did for the multiplier:

```
def get_simple_intro(age, job, location):
    def simple_introduction(name):
        return "This is %s, a %d-year-old %s living in %s" % (name, age, job, location)
    return simple_intro
>> simple_intro = get_simple_intro(10, 'student', 'Seattle')
>> simple_intro('Maya')
'This is Maya, a 10-year-old student living in Seattle'
>> simple_intro('Alison')
'This is Alison, a 10-year-old student living in Seattle'
```

There is, however, an easier way to achieve this.

Currying with *functools.partial*

The functools module in the standard library provides utilities for working with functions:

https://docs.python.org/3.5/library/functools.html

Creating a curried function turns out to be common enough that the functools.partial function provides an optimized way to do it:

What functools.partial does is:

- Makes a new version of a function with one or more arguments already filled in.
- The new version of a function documents itself.

Let's go back one last time to the multiplier example:

```
def multiplier(x, n=3):
    return x * n
```

We will now use functools.partial to create a curried version of the multiplier function:

```
from functools import partial

def multiplier(x, n=3):
    return x * n

double_it = partial(multiplier, n=2)
quadruple_it = partial(multiplier, n=4)

>> double_it(4)
8
>> quadruple_it(4)
16
```

partial returns a curried function in which one or more parameters of the original function have been given values, so that the returned function will not “ask” for those parameters. In the example above, the *multiplier *method was defined right there and then, but it still works if the method is being imported. Let’s see it working with our *introductions* module:

```
>>> from introductions import introduce_person
>>> from functools import partial
>>> simple_intro = partial(introduce_person, age=10, job='student', location='Seattle')
>>> simple_intro('Letty')
'This is Letty, a 10-year-old student living in Seattle'
```