Readings for Week 5

Licensing Information

Portions of these readings were modified or copied verbatim from the very nice book *Think Python 2e* by Allen Downey.

Table of Contents

1) Introduction

Last week, we introduced a very powerful means of abstraction in Python: *functions*, which allowed us to abstract away the details of a particular computation so that it could be computed multiple times on different inputs. We spent some time then on the details of how Python interpreted functions. This week's readings will, first, revisit those details with two more examples. In particular, we'll focus on the the issue of *scoping* (deciding how and where Python looks up variable names). Then, we'll discuss the "first-class" nature of Python functions. Finally, we'll introduce some snazzy new function syntax.

2) More Examples

To begin, we will step through two complex function examples with environment diagrams. These both build upon the things we learned in last week's reading, so you may wish to review those now.

2.1) Calling Functions From Within Functions, Shadowing Globals

First we walk through the following piece of (admittedly silly) code:

```
def f(x):
    x = x + y
    print(x)
    return x
```

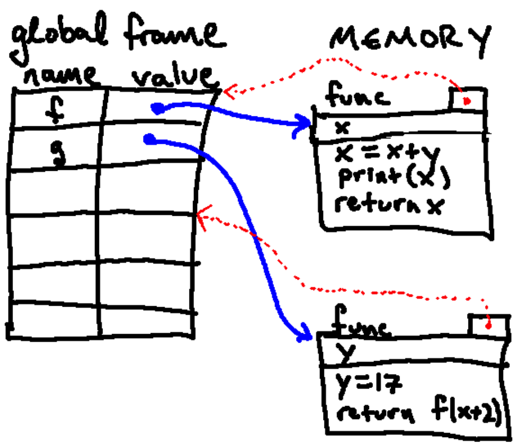
```
def g(y):
    y = 17
    return f(x+2)

x = 3
y = 4
z = f(6)

a = g(y)

print(z)
print(a)
print(x)
print(y)
```

Try to use an environment diagram to predict what values will be printed to the screen as this program runs. You can step through our explanation of how this code runs using the buttons below:



<< First Step | < Previous Step | Next Step > | Last Step >>

STEP 1

The two definition statements create two new functions and bind them to `f` and `g`, respectively, resulting in the diagram above.

At this point, no values have been printed.

2.2) Defining a Function Within a Function

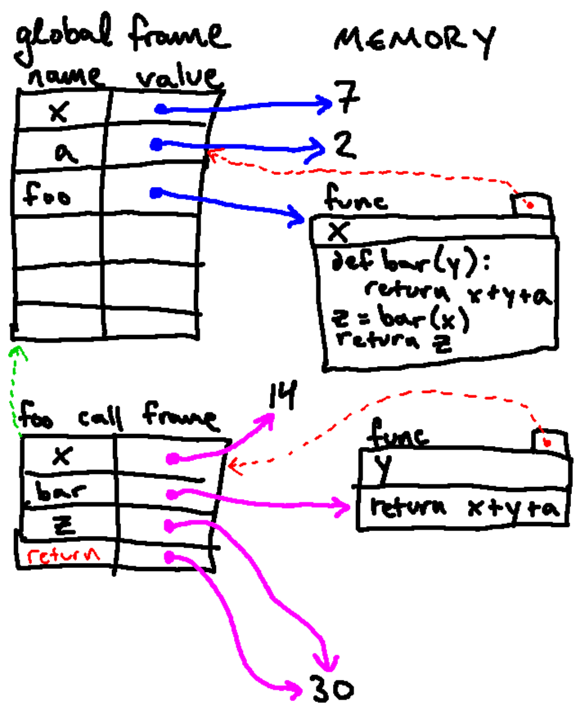
As another example, let's walk through the following piece of code. This piece of code demonstrates a new idea: because function bodies can contain arbitrary code, they can also include *other function definitions*! Consider the following code:

```python
x = 7
a = 2

def foo(x):
    def bar(y):
        return x + y + a
    z = bar(x)
    return z

print(foo(14))
print(foo(27))
```

Try to use an environment diagram to predict what values will be printed to the screen as this program runs. You can step through our explanation of how this code runs using the buttons below:



<< First Step | < Previous Step | Next Step > | Last Step >>

STEP 8

Next, we hit the return statement. The call to `foo` will return this same `30`. This value will then be printed (because of the `print` statement in the main program), and the frame will be cleaned up.

2.3) A Reminder

If you find these diagrams tedious, we get it... In the end, there's a reason we want computers to be the one doing this, after all; they're much better at these operations than we are, and much faster! So in the short term, this *is* tedious. But the long-term benefits are really great! This kind of practice is helpful in building up a mental model of Python's behavior, which is important so that when you encounter unexpected behavior, you can come back to the model. With practice, this kind of thinking will become second nature, and you won't have to draw these diagrams out in such detail.

3) Functions Are First-Class

We now shift gears to learn about a powerful feature of Python: that it treats functions as first-class objects, which means that functions in Python can be manipulated in many of the same ways that other objects can be (specifically, they can be passed as arguments to other functions, defined inside of other functions, returned from other functions, and assigned to variables). In this section, we will explore how we can make use of this feature in our programs.

3.1) Functions as Arguments

Imagine that you wanted to make plots of several different functions. To do that, you would need to figure out which "y" values correspond to each of a number of "x" values. The following code computes these "y" values for different functions:

```
import math

def sine_response(lo, hi, step):
    out = [] # list of "y" values
    i = lo
    while i <= hi:
        out.append(math.sin(i)) # compute "y" value
        i += step # move to next "x" value
    return out

def cosine_response(lo, hi, step):
    out = []
    i = lo
    while i <= hi:
        out.append(math.cos(i))
        i += step
    return out

def double(x):
    return 2*x

def double_response(lo, hi, step):
    out = []
    i = lo
    while i <= hi:
        out.append(double(i))
        i += step
    return out

def square_response(lo, hi, step):
    out = []
    i = lo
    while i <= hi:
        out.append(i**2)
        i += step
    return out
```

Now imagine that you wanted to change the way that you were making the response list (or, change anything at all about the functions' behaviors, really). As it stands now, this would be a pain, because you would have to manually change *each* of the above functions. However, we can fix this by making a general function called `response`, which takes a function `f` as input and returns the list of `f`'s outputs over the specified range:

```
def response(f, lo, hi, step):
    out = []
    i = lo
    while i <= hi:
        out.append(f(i)) # here, we apply the provided function to i
        i += step
    return out
```

Notice that, inside of the definition of `response`, we call `f`, the function that was passed in as an argument. Using `response`, we could compute the response of our `double` function from earlier:

```
# These two compute the same response!
out = double_reponse(0, 1, 0.1)
out = response(double, 0, 1, 0.1)
```

When we pass in `double` as an argument, we do not put parentheses after it. This is because we want to refer to the function itself (which is called `double`), and not to any particular output of the function (which we'd get by calling it, such as in `double(7)`).

Note that we could compute responses for all of the functions described above using this new `response` function:

```
sine_out = response(math.sin, 0, 1, 0.1)
cosine_out = response(math.cos, 0, 1, 0.1)
double_out = response(double, 0, 1, 0.1)
def square(x):
    return x**2
square_out = response(square, 0, 1, 0.1)
```

3.2) Function-ception and Returning Functions

Another useful feature is that functions can not only be passed in as arguments to functions, they can also be returned as the result of calling other functions! Imagine that we had the following functions, each designed to add a different number to its input:

```
def add1(x):
    return x+1

def add2(x):
    return x+2
```

If we wanted to make a whole lot of these kinds of functions (`add3`, `add4`, `add5`, ...), it would be nice to have an automated way of making them, rather than defining each new function by hand. We can do this in Python with:

```
def add_n(n):
    def inner(x):
```

```
        return x + n
    return inner
```

This may be a little difficult to understand at first, but what is happening is this: when `add_n` is called, it will make a *new function* (here, called `inner`) using the `def` keyword, and it will then return this function.

Here is an example of the use of this function (including using it to recreate `add1` and `add2` from above:

```
add1 = add_n(1)
add2 = add_n(2)

print(add2(3))  # prints 5
print(add1(7))  # prints 8
print(add_n(8)(9))  # prints 17
```

Try Now:

What type is each of the following values?

- `add_n`
- `add_n(7)`
- `add_n(9)(2)`
- `add_n(0.2)(3)`
- `add_n(0.8)(2)`

Show/Hide

- `add_n` is a `function`, as with other examples we saw before.
- `add_n(7)` is the result of calling `add_n` with `7` as its argument, which will also be a `function`.
- `add_n(9)(2)` calls `add_n` with an argument of `9`, and then it calls the *result* with an argument of `2`. This yields `11`, an `int`.
- `add_n(0.2)(3)` yields `3.2`, a `float`.
- `add_n(0.8)(2)` yields `2.8`, a `float`.

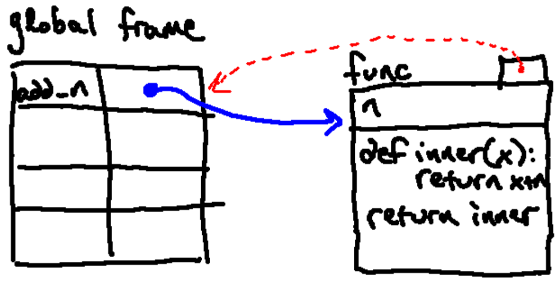
3.3) More Environment Diagrams

The examples above may be a little bit surprising, but we can understand them by working through them using an environment diagram. Even if they aren't surprising, it's important to know exactly what Python is doing under the hood. Here, we'll look at simulating a piece of the above code using an environment diagram:

```
def add_n(n):
    def inner(x):
        return x + n
    return inner
```

```
add1 = add_n(1)
add2 = add_n(2)

print(add2(3))
print(add1(7))
```



<< First Step | < Previous Step | Next Step > | Last Step >>

STEP 1

The first definition statement creates a function `add_n`, as shown in the diagram above. Note that, as before, defining the function *does not* cause the body of the function to be executed. As such, we have not yet hit the definition of `inner` (indeed, we will not hit it until we *call* `add_n`).

4) Defining Functions

To close, we introduce a convenient new way to define functions.

The most common way to define functions in Python, which we've already seen, is via the `def` keyword. For example, earlier we made a function that doubled its input, like so:

```
def double(x):
    return 2*x
```

Recall that this will make a new function object in memory, and associate the name `double` with that object.

Python has another way of defining functions: the `lambda` keyword[1]. The below expression is also a function that doubles its input:

```
lambda x: 2*x
```

The variable name(s) before the colon, here just `x`, are the names of the arguments. The expression after the colon is what the function will return.

This function is almost exactly the same as `double`, except that it does not have a name.

We could have used a `lambda` instead of a `def` when creating the response for `double` from above:

```
double_out = response(lambda x: 2*x, 0, 1, 0.1)
```

If we did not care about being able to access `double` outside of computing its response, it might make sense to do this. This is the same as passing a *function* in as the first argument to `response`; the function is just being defined with `lambda` instead of with `def`.

We could do the same to get the response for `square`:

```
square_out = response(lambda x: x**2, 0, 1, 0.1)
```

And we could have have defined `add_n` as follows:

```
def add_n(n):
    return lambda x: x+n
```

You can also define functions of more than one argument using `lambda`s. Both of the below pieces of code define `multiply` to be a function which returns the multiplication of its two inputs, for example:

```
def multiply(a, b):
    return a*b
```

```
multiply = lambda a, b: a*b
```

You should know that `lambda`, while sometimes a nice convenience, is never necessary—you can **always** use `def` instead! Similar to the comprehension syntax from last week, the `lambda` syntax is less explicit about what Python is doing than its `def` counterpart, and it's harder to debug since it cannot include `print` statements. As such, use it sparingly, and only when the function body is simple (e.g. a one-line return statement).

5) Summary

In this set of readings, we revisited the details of how Python invokes functions. We also learned the ways in which Python functions are *first-class objects*. They can be treated just like any other objects in Python: among other things, they can be passed as arguments to functions and can be returned as the result of other functions! And we saw the `lambda` keyword.

In next week's readings, we'll investigate one way to *use* functions: recursion. And we'll talk about strategies for designing large programs.

Footnotes

[1] This may seem like a bizarre name, but it comes from a mathematical system for expressing computation, called the Lambda calculus.