Structure of Python programs; how assignment and function calls work: scope of names

Up till now we've studied programs just by giving some examples and asking you to imitate them, and haven't bothered with precise definitions. Here will fill in some of those technical details. We won't fill them *all* in, because you would need to be a fairly experienced programmer already to understand it. In fact, you can get quite far by imitating examples and experimenting, and you don't need to understand all the details discussed below, at least not at the outset.

Expressions

We've already seen *expressions*. These are built from constants, like 14, -2.63, 'hello' , which are the simplest expressions, and by applying operators (+,-,*,/,//,%,**) and functions. So the following are all expressions:

```
4+37/2
6*(x+11)
'cat'+('dog'*3)
print(x,y)
27*v+math.sqrt(wind_chill(4,y+3)-z)
```

Statements

We've seen several different kinds of *statements*, and we will see several more later in the course: Expressions are statements. So are *import statement*s, which have the form

import *name_of_module*

and *assignment statements*, which have the form

variable=*expression*

and *function definitions*, which have the form

def *name_of_function*(**sequence of variables, separated by commas**):
    sequence of statements, one per line, indented.

And *return statements*, which have the form

return *expression*

Return statements cannot occur outside of function definitions. You can also use the return statement without an expression followingthe word return (see the discussion of function execution below).

Variables and function names are *identifiers*: These are sequences of letters, digits, and underscores,that do not begin with a digit and are not identical to any of Python's keywords. Names of imported modules might also contain periods.

Program syntax

A program, or Python script, is just a sequence of statements, each of them beginning in the first column of a line., contained in a file with the extension .py. A *syntax error* is a violation of these rules of formation: for example, bad indentation, ill-formed expressions (for instance, mismatched parentheses in an expression), illegal identifiers, etc. You can check if a Python script is syntactically correct by running **Check Module** in IDLE.

The rules here are very relaxed. The nonsense 'program' shown below is syntactically correct, in spite of a large number of errors that will keep it from running.

```
import stupid_library_I_made_up

'cat'*'dog'
print(u(5))
#weird place for function definition,
#but it's syntactically legal
def u(thing):
    return thang
```

Figure 1. A nonsense program filled with errors, but nonetheless *syntactically* correct.

On the other hand, if I left off any of the parentheses, or changed the name 'thang' to 2thang, or to the keyword def, I would get a syntax error.

Program execution

When the script is run, each statement is executed in turn. Executing an import statement executes all the statements in the imported module. In practice, usually this means making the functions in the imported module available to be called in our program—this is what happens when we import the `math` module, so that we can call functions like `math.sqrt, math.sin`, etc.

Executing a function definition does not execute the function itself, but instead makes the function available to be called later in the program.

Executing an expression causes the expression to be evaluated, but unless there is some visible side effect to doing this (e.g., if the expression contains a call to the print function), it will look as though nothing happened. For instance,

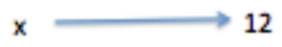
4*3

is a legal statement in a program, and it would execute properly, but you won't see any result.. As we've already discussed, when an expression is evaluated we determine not only the value of the expression but its type as well.

When an assignment statement is executed, first the expression on the right-hand side is evaluated, and the value and type are stored in the computer's memory. The result is called an **object**. Then, the variable is set to refer to that object. We often show this in a diagram, with an arrow from the variable name to the object. For instance, the result of executing

x=4*3/2

is shown in the diagram below.



A *run-time error* occurs when a statement cannot be successfully executed. In the nonsense example above, there are a slew of conditions that will trigger run-time errors. Let's go down the list. First of all, there is no module stored on my computer called 'stupid_library_I_just_made_up', so this will produce a ModuleNotFoundError. If I eliminate this useless statement, you will get a TypeError, because you can't multiply two strings. Python does not determine the type of an expression until it tries to evaluate the expression, not when it is checking syntax.

We can get rid of this error by changing 'cat'*'dog' to 'cat'+'dog', although the resulting statement doesn't produce anything that we can see when we run the program. Now, though, the next line produces a NameError, because we tried to call the function u before its definition.

Let's move the statement `print(u(5))` to the end.. Now we get another NameError, because the name `thang` is undefined. If you change this to thing, the program runs, and prints the value 5. (Although it's still nonsense.)

Function execution

When a statement contains a call to a function, as in x=wind_chill(10,3+z)

there's a lot going on. Let's illustrate it all with the example shown below. (Again, the program is deliberately kind of stupid.)

```
def v(a,b):
    c=(a-b)**3
    return c

x=5
y=x
x=x/2
z=v(y+4,x)
print(z)
```

Figure 2. A dumb but instructive example

Pay close attention to the order in which things are executed when we run the program:

The first thing to be executed is the *definition* of the function v. But, to repeat, that does not mean that the statements in the function are themselves executed. It just means that the function is now available to be called in statements in the main program (or any functions whose definition follows this one).

Next the assignment x=5 is executed. This creates an object of type int and value 5, and the name x is set to refer to that object.

Next, in the assignment statement y=x, y is set to point to the same object.

In the assignment x=x/2, first the expression x/2 is evaluated. This creates an object of type float and value 2.5; then it changes the reference associated with the name x so that x now refers to this new object.

Now the complicated part: The next statement involves a call to the function v. The *arguments* in this function call are the expressions y+4 and x. We first evaluate y+4 and obtain an object of type int and value 9; we already have the object referred to by x. Then the function *parameters* a and b are set, respectively, to refer to these objects.

We now execute the code in the function v: We evaluate the expression (a-b)**3 and get an object with value 274.625 of type float, and then set c to refer to it. The return statement communicates a reference to this object back to the main program:

This object is now the result of evaluating the expression v(y+4,x). Finally, the variable z is set to refer to this object, and in the last statement, the object's value is printed.
(It is also possible to use the return statement without an expression following the word return. This just causes control to return from the function back to the caller. So, for example, if a function contained the lines

```
return
print(7)
```

then the print statement would never be executed.)

The whole dance is illustrated in the figure below:

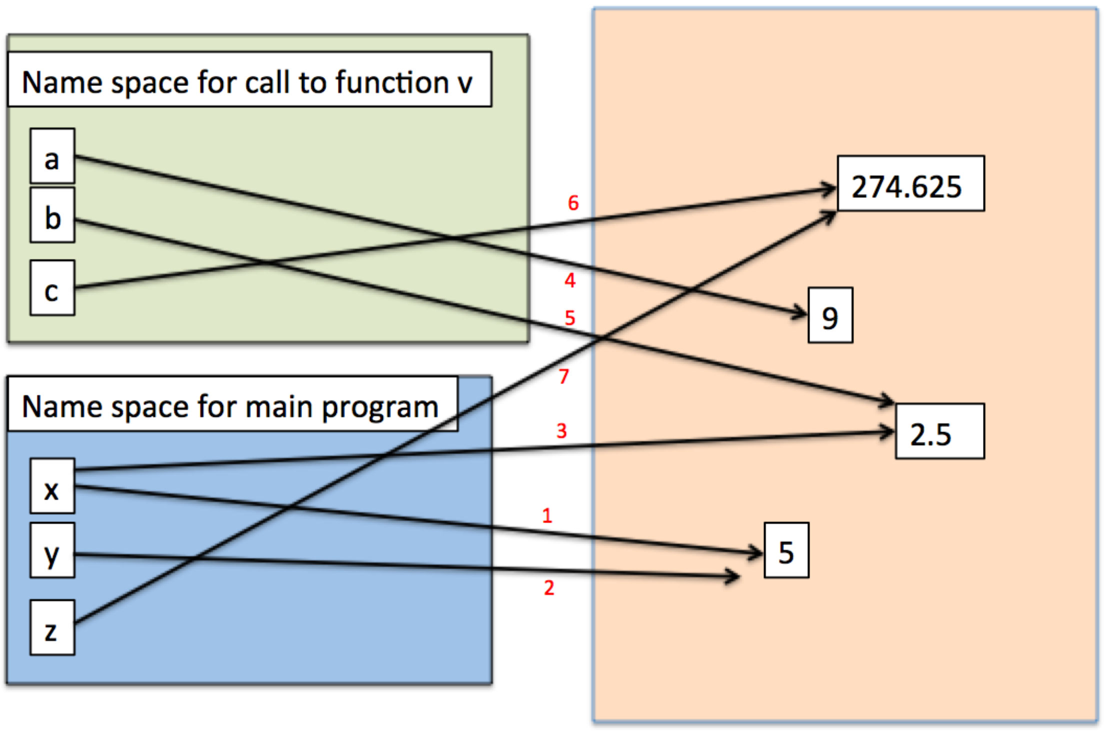

Name space for call to function v
a
b
c
Name space for main program
x
y
z
274.625
9
2.5
5
1
2
3
4
5
6
7

Figure 3. Order in which references are created during a run of the program shown above.

The numbers on the arrows indicate the order in which these references are created.

Locality of Names

Don't worry if you don't understand all of that right away. But there is one important take-away: You'll notice the different regions in the diagram---the right-hand region is supposed to represent a vast computer memory, where objects are created and stored. The left-hand side holds the names of the variables that refer to these objects. There are two different regions, labeled 'name space'. The main program, and each call to the function, have separate name spaces, and the *name space for the function call is valid only while the function is executing*. When the control returns to the main program, that name space is gone.

What this means in practice is that while the function v is executing, these names have nothing to do with names in the main program. You could use x,y,z instead of a,b,c for the variables in the function v, and nothing would change. The *scope* of the names x,y,z in the function is restricted to that function call.

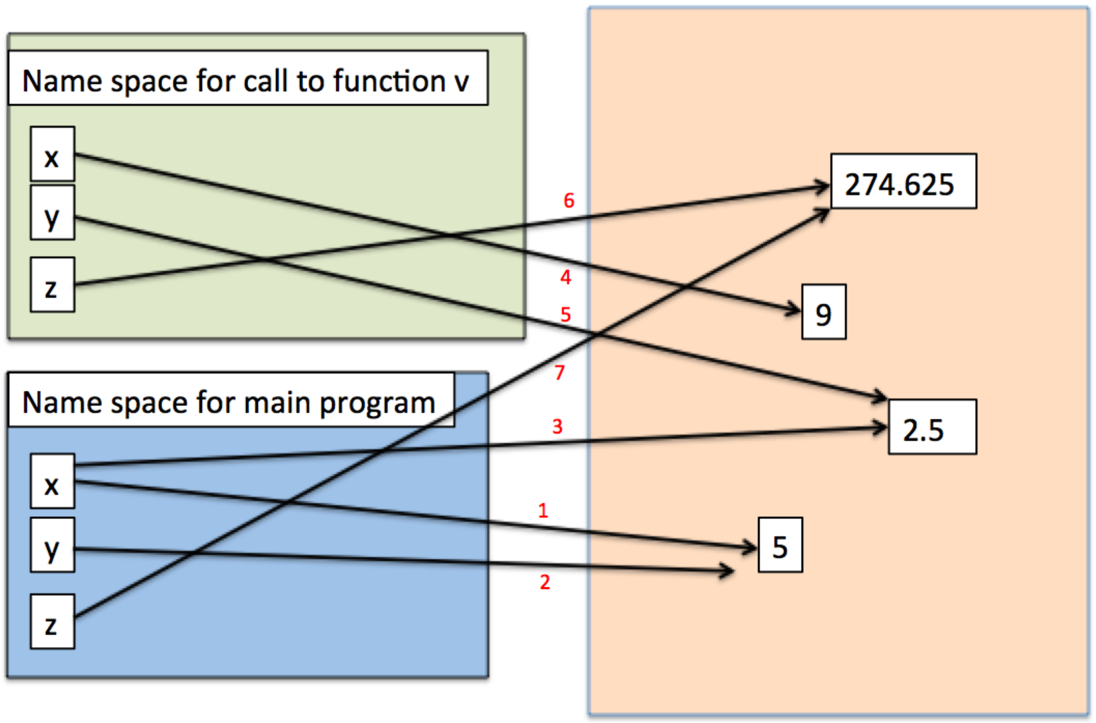

Name space for call to function v
x
y
z
Name space for main program
x
y
z
274.625
9
2.5
5
1
2
3
4
5
6
7

Figure 4.Locality of names: Effect of changing the names of the variables in v to x,y,z. Everything is the same!

If you think about this in terms of a built-in function like math.sqrt, this makes sense. Somebody wrote that function, which means that somebody had to choose names for the parameter of this function and whatever variables were used in the function. But when we call math.sqrt, we have no idea what those names are, and we shouldn't have to worry about using different names or the same names in our program.