

Listen. Strange women lying in ponds distributing swords is no basis for a system of government.

Supreme executive power derives from a mandate from the masses,
not from some farcical aquatic ceremony.

Variable Scope

Now that we have written some programs we should describe how Python keeps track of all the variables in our programs — what values they have, and which variable are being talked about at a specific point in our programs. This is an issue in every programming language and is typically managed by using **environments** (a data structure) to maintain information about the variables and values that can be accessed in a particular part of a program (**program scope**) such as a function body. The **scope of a variable** describes where in a program a particular variable can be used.

Python uses dictionaries for its environments — each dictionary maintains the mapping from variable names to values. Python uses the term **namespace** to refer to these dictionaries. There is one **global namespace** that keeps information about everything at the 'global' level. When each function is called a new **local namespace** is constructed to keep track of variables inside the function. Python comes with two functions that extract the global and (current) local namespaces: `globals()` and `locals()`. Below is some code and some results of running the code that gives an insight into how Python keeps track of variables.

```
>>> a = 10
>>> b = 11
>>> def foo(b) :
    print('Global namespace = ', globals())
    print('Local namespace = ', locals())
    return a + b

>>> foo(3)
Global namespace = { '__loader__': <class '_frozen_importlib.BuiltinImporter'>,
'foo': <function foo at 0x7f2cb35f58c8>, '__builtins__': <module 'builtins' (built-in)>,
'__spec__': None, '__package__': None, '__doc__': None,
'__name__': '__main__', 'b': 11, 'a': 10}
Local namespace = {'b': 3}
13
>>>
```

We can see that variables `a` and `b` are defined in the global namespace and `b` is also defined in the local namespace of `foo`. So, why is the value of `foo(3)` equal to 13? Python first looks in the local namespace to see if a given variable is defined there. If so, it uses the corresponding value. If not it looks in the global namespace for a value. In the case above, `b` is defined in the local namespace and its value (3) is used. The variable `a` is not defined in the local namespace so Python looks in the global namespace, getting the value 10.

Compare the above example with the following.

```

>>> a = 10
>>> b = 11
>>> def foo(b) :
>>>     a = a + 3
>>>     return a + b

>>> foo(3)

Traceback (most recent call last):
  File "<pyshell#58>", line 1, in <module>
    foo(3)
  File "<pyshell#57>", line 2, in foo
    a = a + 3
UnboundLocalError: local variable 'a' referenced before assignment
>>>

```

In this case we are trying to modify a global variable (i.e. one not in the local scope). Python sees this assignment as follows — on the left hand side is `a` and so Python treats this as a local variable. On the right hand side is an occurrence of `a`, but it has not been given a value in the local scope and so our favorite error is produced.

It is possible to modify global variables as the following example shows.

```

>>> a = 10
>>> b = 11
>>> def foo(b):
>>>     global a
>>>     a = a + 3
>>>     return a + b

>>> foo(3)
16
>>> foo(3)
19
>>>

```

global and why not

The `global` declaration tells Python to treat `a` as a global variable. This is a **very dangerous** thing to do and should be avoided where possible. Using global variables makes it difficult to understand the logic of programs — in the previous example, we call `foo` twice with the same argument but get different results.