Ordering of Functions and Scope of Variables in Programs with Functions

FEB 25TH, 2013

Ordering functions in your code

- Will the following code work? Here the function is defined after the main program that is calling it.

```
print foo()
def foo():
        return "hello"
```

- Will this work? Here functions are defined before the main program. But, `foo2()` is called before it is defined by `foo1`.

```
def foo1():
        return foo2()
def foo2():
        return "hello"
print foo1()
```

How does Python process code with functions?

```
def foo1():
        return foo2()
def foo2():
        return "hello"
print foo1()
```

1. Python starts scanning the code from the beginning of the file.
2. It notes down names of functions as it encounters their *definitions*. Note that the functions are not executed at this time.
3. It reaches the first executable statement (`print foo1()`) and since `foo1` is known to Python, control is transferred to `foo1`.
4. In `foo1`, Python encounters a call to `foo2`. Function `foo2` is also known to Python and so control is transferred to `foo2`.

Moral of this example?

- Define *all* functions before the main program.
- And then don’t worry about the order in which the functions themselves are defined.

Scope of a variable

- The *scope* of a variable refers to the “where” and “when” a variable is available for use.
- Things were simple when we did not have functions.
- If we only had a main program: the scope of a variable extends from the point where the variable is first defined till the end of the program.

Scope of variables inside functions

- Parameters and variables defined inside a function are "local" to that function.

```
def foo():
    var1 = "hello"
    return var1 + var1

# main program
print foo()
if var1 == "hellohello":
    print foo()
```

var1 is a variable that is local to foo(). It comes into existence when the first line of foo() is executed and it "dies" when we exit the function.

var1 is not defined and this usage will cause an error.

Function parameters are also local

```
def foo(x):
    var1 = "hello"
    return var1 + x

# main program
print foo("bye")
if x == "hellohello":
    print foo()
```

The variable x is undefined here because the parameter x lives only for the duration of the function

How does all this work?
Mental model: version 1

1. Python creates a dictionary of variable names when it starts evaluating the main program. It uses this dictionary to insert, look up, and update variable names.
2. When the function `foo` is executed, a new dictionary of variable names, specific to `foo` is created.
3. First the parameter `x` is inserted into this dictionary. Then variable `var1` is inserted.
4. Whenever we access a variable inside `foo`, `foo`'s dictionary is looked up.
5. When the execution of `foo` is over, `foo`'s dictionary is destroyed.

Global variables

- The mental model 1.0 explains why variables defined inside a function cannot be used in the main program.
- What about variables defined in the main program? Can they be used inside a function?

```
def foo(x):
    var1 = "hello"
    return var1 + x + y

y = "good"
print foo("bye")
```

y is a *global* variable (i.e., it is defined in the main program), but can be used in the function that is called after it is defined.

Mental model: version 1.1

- Here is a “more correct” version of item (4)

  Whenever we access a variable inside `foo`, `foo`’s dictionary is looked up. If a variable is not found in `foo`’s dictionary, then Python looks up the dictionary of the main (calling) program.

- This allows a function access to “global” variables.

Local variables override global variables

```
def foo(x):
    y = "hello"
    return x + y

y = "good"
print foo("bye")
print y
```

This is a different, local y. During the function, all mention of y refers to this local y.

y is a global variable

- This mechanism also gives local variables precedence.
- In the above example, the variable `y` is found in `foo`'s dictionary and that is the variable that is accessed in `foo`.

Explicit global variables

```
def foo(x):
    global y
    y = "hello"
    return x + y

y = "good"
print foo("bye")
Print y
```

We are now explicitly declaring that the y we want to access inside foo() is the global variable y

- global is a Python keyword.
- If it were not for the global y statement, the variable y being mentioned inside foo would have been defined in foo's dictionary and would be local to foo.

Explicit global variables avoid confusion like this

```
def foo():
    if y == "hello":
        print "Hello to you as well!"

    y = "hi"
    print y

y = "hello"
foo()
```

Here y is a global variable

And here y is a local variable

- This is an error in Python because Python sees the assignment `y = "hi"` inside `foo()` and assumes that all appearances of `y` inside `foo()` refer to this local variable.
- Therefore, in the first line of `foo()` we are accessing a variable not defined yet.

WARNING!!

- I would discourage the use of global variables, both implicit and explicit.
- Communication between functions or between the main program and a function should be explicit – via parameters/arguments and returned values.