## CSSE 120 – Introduction to Software Development

# Concept: *Namespaces* and variables' *Scope*

Today's programs might have millions of variables. If we had to think of a different name for each one, we would be in trouble! For that reason, variables are ***local*** to the functions or class instances in which they are defined. That is, **each function *call* has its own *namespace***, which means:

With functions:

- When a function is ***called***, a ***namespace*** for its variables is created. The function's parameters and any variables defined inside the function are placed into the function call's namespace.
- Variables in one namespace have NOTHING to do with variables of the same name in another namespace. The namespaces are completely independent.
- When a function returns to its caller, its namespace (and all the variables defined in it) is "forgotten" and no longer available.

With instances of a class:

- When a class is ***instantiated***, a ***namespace*** for the variables of the class instance is created. The class methods and any variables given to the instance are placed into the class instance's namespace.
- Variables in one class instance have NOTHING to do with variables of the same name in another class instance because they are in different namespaces. Variables in one namespace have NOTHING to do with variables of the same name in another namespace. The namespaces are completely independent.

The next page shows the creation of namespaces, from a textbook by Ljubomir Perkovic. The pages after that present a concrete example using functions. You will see similar effects with different instances of a class in the future when we use classes more.

We use the following module as our running example:

Module: stack.py

```
def h(n):
    print('Start h')
    print(1/n)
    print(n)

def g(n):
    print('Start g')
    h(n-1)
    print(n)

def f(n):
    print('Start f')
    g(n-1)
    print(n)
```

After we run the module, we make the function call `f(4)` from the shell:

```
>>> f(4)
Start f
Start g
Start h
0.5
2
3
4
```

Figure 7.2 illustrates the execution of `f(4)`. Explicitly shown in the figure are the three different namespaces and the different value that `n` has in each. To understand how these

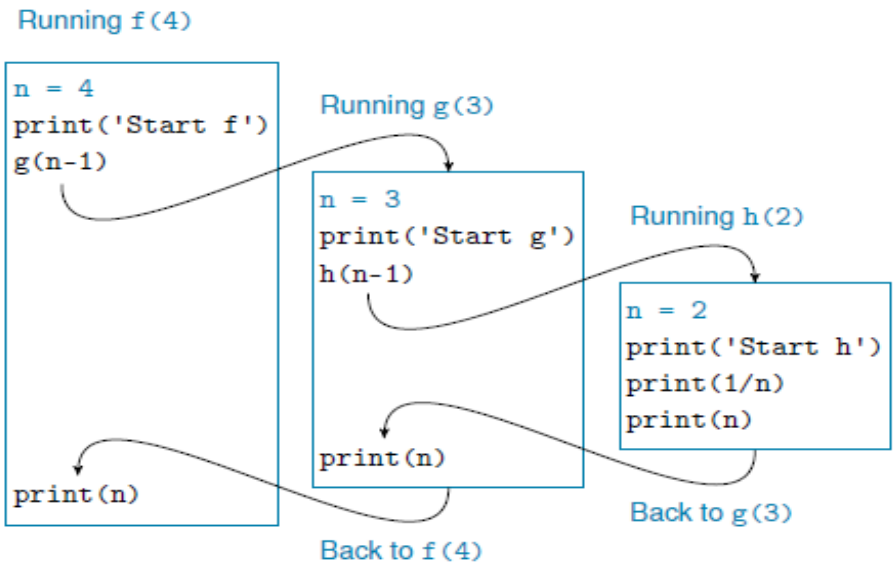


**Figure 7.2 Execution of `f(4)`.** The execution starts in the namespace of function call `f(4)`, where `n` is 4. Function call `g(3)` creates a new namespace in which `n` is 3; function `g()` executes using that value of `n`. Function call `h(2)` creates another namespace in which `n` is 2; function `h()` uses that value of `n`. When the execution of `h(2)` terminates, the execution of `g(3)` and its corresponding namespace, in which `n` is 3, is restored. When `g(3)` terminates, the execution of `f(4)` is restored.

Here is a concrete example to show that ***variables are local to their function***.

In the code shown to the right:

```
def main():
    x = 10
    y = foo(x)
    x = 'hello'

    y = foo(y)

def foo(m):
    x = 89
    return (m * x)
```

- The three `x`'s in `main` have ***nothing to do*** with the `x` in `foo` – changing the `x` in `foo` does not affect the `x`'s in `main`, and vice versa.
- The three `x`'s in `main` refer to the ***same*** place in memory – changing any of them changes the others.
- The two `y`'s in `main` refer the ***same*** place in memory -- changing either of them changes the other.
- The `x` and `y` in `main` have ***nothing to do*** with the `m` (or any other variable) in `foo` – these three variables each have their own space in memory.

  However:

  - Because a function call assigns values to the parameter of the function, there is a relationship:
    - When `foo(x)` in `main` runs, the variable `m` in `foo` is assigned the value of `x` in `main` at that point. (That value is `10` in this example.)
    - When `foo(y)` in `main` runs, the variable `m` in `foo` is assigned the value of `y` in `main` at that point. (That value is `890` in this example).
  - Also, recall that a ***`return`*** sends a value back to the caller. So when the statement

    ```
    y = foo(x)
    ```

    in `main` runs:
    - First, the right-hand-side is computed. That is, `foo(x)` is evaluated. This means that the `m` in `foo` is set to the value of `x` in `main` (which is 10 at this point). Then the code in `foo` runs and then the `return` statement in `foo` executes, sending `(10 * 89)`, which is `890`, back to `main`.
    - Second, the left-hand-side of the `y = foo(x)` statement is executed, so `y` in `main` is set to `890`.

**Try tracing the execution of the code (starting in `main`) by hand.** That is, write down line-by-line what variable(s) are changed to what values, being sure to distinguish variables in one function from variables by the same name in the other function.

The answer is shown on the next page, but **try it yourself first!**

Here is a "trace" of the execution of the code (starting in **main**). The code is repeated to the right for your convenience.

```
def main():
    x = 10
    y = foo(x)
    x = 'hello'

    y = foo(y)

def foo(m):
    x = 89
    return (m * x)
```

- **x** in **main** is set to **10**.
- **foo(x)** is called, which means:
  - **m** in **foo** is set to **10**.
  - **x** in **foo** is set to **89**.
  - **(m * x)** is computed to be **890**, and **890** is returned to the caller.
- The rest of **y = foo(x)** in **main** runs, so **y** in **main** is set to **890**.
- **x** in **main** is changed to **'hello'**.
- **foo(y)** is called, which means:
  - **m** in **foo** is set to **890**.
  - **x** in **foo** is set to **89**.
  - **(m * x)** is computed to be **890 * 89**, which is **79,210**, and **79,210** is returned to the caller.
- The rest of **y = foo(y)** in **main** runs, so **y** in **main** is changed to **79,210**.