CSSE 120 – Introduction to Software Development (Robotics section)

Concept: *Functions with Parameters, and Namespaces*

Defining functions

A function is a chunk of code that has a name. Here (to the right) is a portion of an example of the notation for ***defining*** a function.

```
def convert_and_return(celsius):
    fahrenheit = ((9 / 5) * celsius) + 32
    return fahrenheit
```

The ***name*** of the function follows the keyword `def`. The variables in the parentheses after the name of the function are called ***parameters***. This function ***returns*** a value. (Functions that have no ***return*** statement return the special value *None*.)

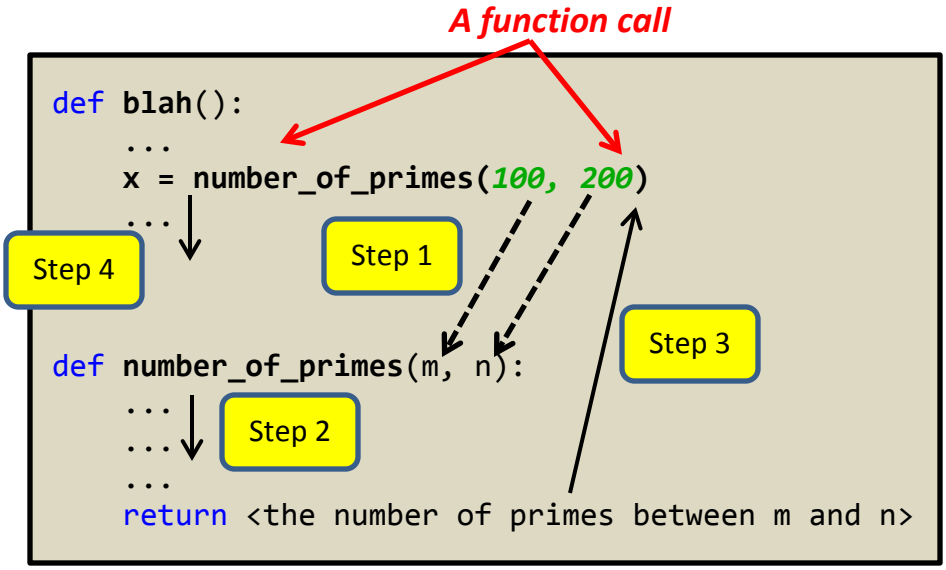
Why have functions?

Functions are powerful for 2 reasons:

- They help ***organize a program into logical chunks***. That makes it easier to:
  - Modify the program (by focusing your interest on the chunks of interest).
  - Write correct code (by understanding the organization of the program).
  - ***Encapsulate*** (enclose and hide) the behavior of a function inside its definition, thus separating the ***specification*** (*what* the function accomplishes) of the function from its ***implementation*** (*how* it accomplishes its specification).
  - Test the program by testing the chunks. (This is called ***unit testing***.)
- You can ***re-use functions***. That is, you can call them over and over again, with different values for the parameters to achieve different results.

Calling functions

You ***call*** (aka ***invoke***) a function by writing its ***name followed by parentheses***, with the ***actual arguments*** placed inside the parentheses.



When you call a function:

1. The actual ***arguments*** of the function call (the values in the parentheses) are sent into the formal ***parameters*** of the function definition.
2. ***Execution continues*** at the beginning of the definition of the called function.
3. When the function's ***return*** statement is executed, the returned value is sent back to the calling function. Or, if the end of the function is reached without a return statement, the special value *None* is sent back to the calling function.
4. ***Execution continues*** from the place where the function call appeared, with the returned value replacing the function call.

Namespaces

Today's programs might have thousands of functions. If we had to think up new variables for each function, we would be in trouble! For that reason, variables are ***local*** to the functions in which they are defined. That is, **each function call has its own *namespace*.**

From a textbook by Ljubomir Perkovic:

We use the following module as our running example:

Module: stack.py

```
1  def h(n):
2      print('Start h')
3      print(1/n)
4      print(n)
5
6  def g(n):
7      print('Start g')
8      h(n-1)
9      print(n)
10
11 def f(n):
12     print('Start f')
13     g(n-1)
14     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

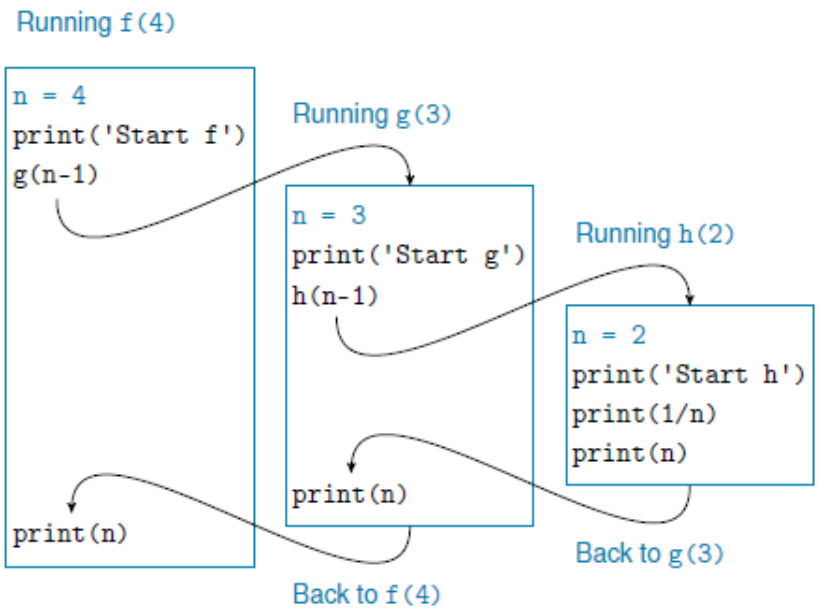


Figure 7.2 Execution of `f(4)`. The execution starts in the namespace of function call `f(4)`, where `n` is 4. Function call `g(3)` creates a new namespace in which `n` is 3; function `g()` executes using that value of `n`. Function call `h(2)` creates another namespace in which `n` is 2; function `h()`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