Notes on functions

From the textbook: Read Chapter 3 on functions, and the first 3 sections of Chapter 6 on 'fruitful functions' (a term apparently devised by the textbook's author). We've used a bunch of functions already: For example,

**sqrt** in the math library takes an int or a float as an argument, and returns a float

**int** takes as an argument a float, an int, or a string, and returns an int, provided there is a meaningful integer representation of the argument

**float** takes as an argument an a string or an int and returns a float, with the same proviso.

**str** takes as an argument any object and returns a string representation of the object's value.

**type** takes as an argument any object and returns a type object.

**round** takes two arguments, a floating point number and an int and returns a float.

**raw_input** takes as an argument a string (the prompt string) and returns a string (whatever you type at the keyboard)

**create_oval** can take four arguments of type int (there are ways to supply additional arguments) and does not return anything

The textbook author calls functions that return a value 'fruitful functions'. All but the last function on the list below are fruitful functions. **create_oval** as described here is treated as a void function---it doesn't return a value. (Full disclosure: **create_oval** actually does return a value, but we just aren't using it, so let's pretend for now...)

**Now we learn how to write functions.**

The syntax for creating new functions is illustrated in the files bart_simpson.py, and in windchill.py. The model is:

```python
def name_of_function(list of parameters, separated by commas):
    sequence of statements, indented 4 spaces
```

The four spaces for the indentation are a suggestion for 'best practice', rather than a rule. However all the statements must be indented the same amount. IDLE will handle the indentation for you automatically, once it sees the colon at the end of the first line. The rules for naming the function are the same as for naming variables. The function can have 1 parameter, none, or many.
What's the point?

A good summary is given in the brief section 'Why Functions?' in the textbook. Observe that windchill.py splits the program into two separate modules, a main program section that gets the user input and displays the result, and the function that performs the wind chill calculation. Each piece stands on its own: we could use the windchill function as it is with a different main program, one that might call it many different times with different parameter values. We could also change the windchill function, perhaps coding the computation differently, without having to alter anything in the main program.

This last point underscores a crucial principle: The caller of a function (in this case the main program) communicates with the function ONLY through the values of the arguments and the return value. A very common student error is to include print statements in functions when none are needed. (The print statements that appear in the textbook’s development of the distance function in Chapter 6 are only there for the testing and development phase, and are removed from the final version.)

What is happening? (What's in a name?)

From the standpoint of the program that calls the function, the function is a black box: The program hands to the function the values of the arguments and gets back the return value. But the program cannot see inside the function, and does not know how the function does what it does, nor the names of the parameters and variables that the function uses.

Here is a somewhat more technical explanation of what happens when a function is called and executed, as well as some of the rules for names in functions and the programs that call them.

Look at the program scoperules.py. When the first line of the main program is executed, an object of type int with value 3 is created, and the name z is associated with it. This name z lives in the 'name space' for the main program.
In the next statement, the function f is called, with z as the argument. When f is launched, its own name space is created, and the name x in the function is associated to the same object as z.

![Diagram](image1)

Then the first statement of the function f is executed. This creates a new object of type int and associates the name z to it. The two z’s are treated as different names, and as long as we are in the function f, it is this second z that matters. This is what we mean when we say that the names of variables and parameters in f are local to f.

![Diagram](image2)

The function terminates with the return statement, and control passes back to where we left off in the main program. A new name y is created in the main
program’s name space and associated to the result of the function. The name space for the function \( f \) disappears.

The value of \( y \) is handed to the print statement and 7 is printed. Then the value of \( z \) (the only \( z \) that is still around) is printed. If you tried to print the value of \( x \), you would get an error message. There is nothing named \( x \).