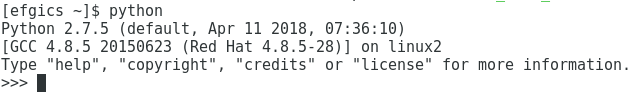
**Python Crash Course**

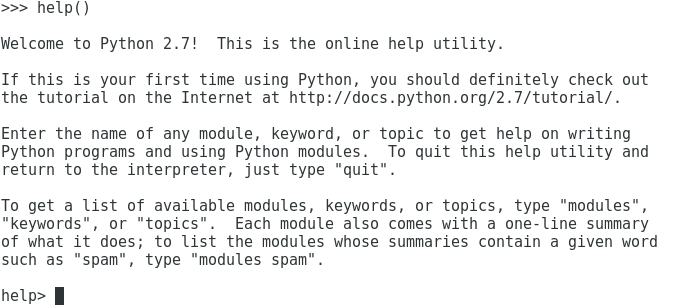
For this crash course, you’ll be learning Python 2.7, as opposed to Python 3.5. There are some subtle and some not so subtle differences between the two, but once you learn one, it’s easy to transition to the other. Feel free to use either version of Python for your assignments and/or project.

**Python Help – Your Guide to Keywords, Syntax, and More**

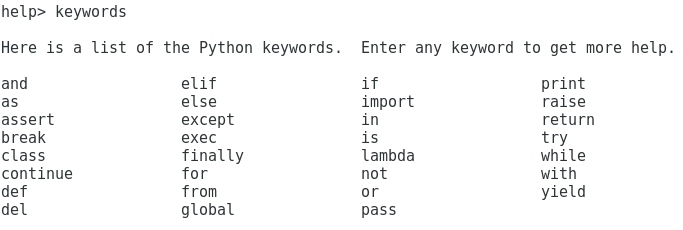
Python can be operated in a couple different ways; either in “interactive” mode or by executing a script from the command line. We’re going to operate exclusively in interactive mode. To enter Python’s interactive mode, just type **python** on the command line.



To enter the help system, type “**help()**”.

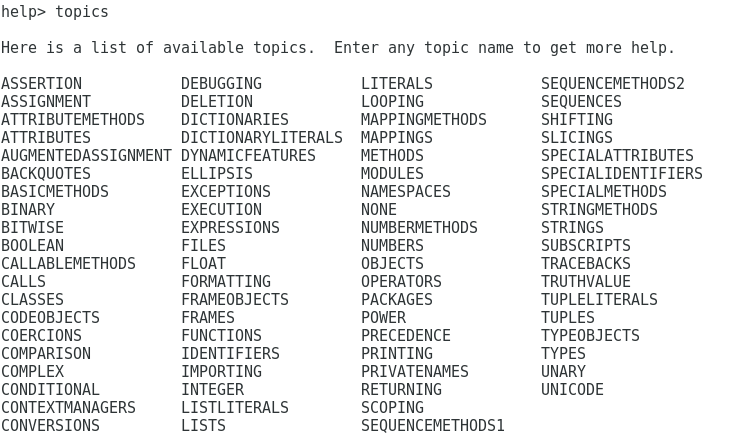


First, we want to see the list of Python keywords, which are reserved words that can only be used for their intended purpose (not as variable names or function names).



Type the name of a keyword at the help prompt to learn more about it. You’ll see a “manual” style page when you do so. Press Q to exit a help page. Try looking at the help page for pass; it’s a really useful command for instances where you’re creating code stubs that require statements in order for the interpreter to not throw an error.

Besides listing keywords and their functionality, the help system also includes help on specific topics that you might want more information on. Type **topics** to see what’s available.



We’ll get to loops soon enough, but if you absolutely can’t wait to learn more about loops in Python, just type LOOPING to get more information. Many of these topic pages are quite verbose and give more information that is probably desired, but they can point you in the right direction of where to look next for help.

To exit the help system, type quit and we’ll start looking at variables.

**Variables**

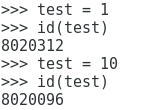
It’s important to note that everything in Python is case sensitive. There are also very specific guidelines for naming identifiers, such as variable and function names.

* Identifiers may include lowercase and uppercase letters, digits, and underscores. That’s it! No special characters are allowed!
* Identifiers may **not** begin with a digit.
* As mentioned above, keywords cannot be used as identifiers.
* You’ll know if an identifier is invalid when the Python interpreter complains!

Variables require no specific declaration with a type; just assign a value to the variable and Python will do its best to interpret the value’s type. Try creating a variable by typing “test = 1”.

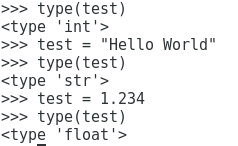
Whether you need to know this or not, everything in Python is an object! When you assign the value 1 to the “test” variable, Python actually creates an object to represent the value 1, creates the test variable if it didn’t already exist, and associates the test variable with the object 1. That is, the variable test is a reference to the object 1.

If you were to then set test to 10 using “test = 10”, you’d create a new object for the value 10 and the object for 1 will get collected by the garbage collector. Let’s prove this using the **id** function.



What the above screenshot shows is the id (memory location) of the value assigned to test. Notice that the memory location for test changes when the value of test changes from 1 to 10. This is completely opposite behavior from a typical object-oriented language like Java or C++ where the memory location is assigned to the variable and the value is simply stored in that memory location.

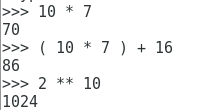
Objects in Python actually hold three things: the value of the object, the object type, and a reference counter that indicates whether the object should be garbage collected or not.



The type function tells us the type of the object, not the type of the variable since variables do not have types. Notice that after we assigned 10 to test, the type of the object 10 was int (integer) and when test was set to the string “Hello World”, this new string object was of type string, as expected. Finally, when 1.234 was assigned to test, that object was a float type.

**Statements**

A statement in Python is an instruction that the Python interpreter can read and execute. When using the command line version of Python, the tendency is to “try out” statements in the form of an **expression** where a result is output to the console, but is not assigned to a variable. However, when writing a script, most statements (outside of control structures and function definitions) are **assignment** statements.

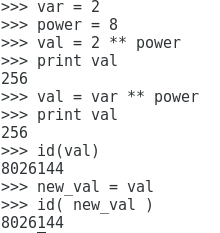


The expressions above are evaluated by the Python interpreter on the command line and their result printed to the console. The “\*\*” operator means to raise the value on the lefthand side to a power on the righthand side. Notice that Python is not picky about spacing in an expression and it doesn’t require any odd structuring in place to know how to evaluate a mathematical expression.

We’ve already seen a few assignment statements in the last section, but to formally state it, assignment statements take the form:

**variable = expression**

As you would expect, assignment statements can have expressions on the righthand side including only literals (integers, floats, strings), only variables, or a mix of literals and variables.



In the third expression, we raise 2 to the 8th power where the variable power is set to 8, while the fifth expression shows var (set to 2) is them raise to power. Same result, but using variables only. What’s interesting is when new\_val is assigned to val, they both share the same memory location (8026144) since they point to the same *object* (the integer 256).

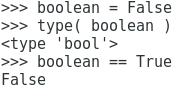
What is the type of val and new\_val? Should they be of the same type?

**Basic Data Types**

As you’ve seen, Python is a dynamically typed language since variables merely point to objects and may be assigned to point to different objects at any time.

***Booleans***

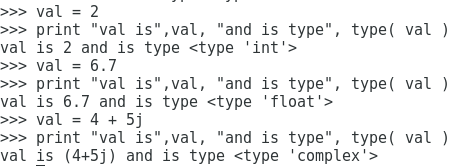
Boolean variables are mostly used as part of conditions in conditionals and loops. In Python, a Boolean may be assigned the keywords True and False.



The variable “boolean” is first assigned to False and we can see that the object type is “bool”, for Boolean. The comparison operator in Python is == and works on types like bool, int, float, and even string!

***Numbers***

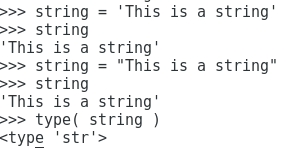
Python numbers come in the form of int, float, and complex.



We’ve seen int and float sprinkled in throughout this crash course, but one special type you haven’t seen yet is complex, which allows the user to perform calculations on complex numbers. Complex numbers contain a real and imaginary part. In this example, the real part is 4 and the imaginary part is 5j where j indicates that the value next to it is imaginary. We don’t have a particular application for complex numbers in this course, but they’re worth noting nonetheless.

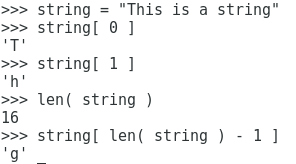
***Strings***

Strings are often the bane of a programmer’s existence in languages like C, C++, and Java, but Python is much more user friendly with strings. Strings may be declared using single or double quotes.



When on the command line, to see the contents of a string variable (or any variable), simply type the name of the variable and press ENTER. Of course, in a script, you’ll want to use a print statement.

What makes strings so friendly is ease of access to elements within the string via “string slicing”, which uses a bracket ([idx1[:idx2]]) notation. Notice that you can request individual elements of a string by just using idx1 in the bracket or a range of elements (substring) by specifying “:idx2”.



Strings are indexed beginning at index 0. Therefore, to get the first element of a string, you would need to access element 0, the second element of a string is element 1, and so on. The **len** function gives you the length of a string. If you don’t know the length of a string, it’s easy to access the last element of the string by using the expression “len( string ) – 1” in the brackets.

**Operators**

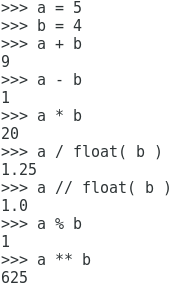
You saw some basic arithmetic operators last time and you’ll see a couple more today in addition to comparison, logical, and membership operators.

***Arithmetic Operators***

Enter Python’s interactive mode by typing **python** on the command line. Here is a list of arithmetic operators that you will find useful as we get deeper into Python.

* **Addition (+)**
* **Subtraction (-)**
* **Multiplication (\*)**
* **Integer or Floating-Point Division (/) – at least one operand must be of type float to get floating point division to occur**
* **Floor Division (//) – rounds down to the nearest whole number**
* **Modulus (%)**
* **Exponent (\*\*) – raise a number to a power, for example x\*\*y, raises x to the y power**

Try these for yourself to see what happens.



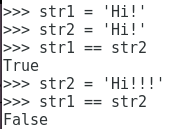
Notice that to get a floating point result for / and //, that the object stored in b was cast as a float. What happens without the cast? An integer division would occur in both cases and would produce the same result: an integer object whose value is 1. Besides casting to float with **float(var)**, you can cast an object to int with **int(var)**,and string with **str()**.

***Comparison Operators***

These operators allow comparisons of object values and will be used in conditionals and loops, among others, and are quite intuitive.

* **Equal to (==)**
* **Not equal to (!=)**
* **Greater than (>)**
* **Greater than or equal to (>=)**
* **Less than (<)**
* **Less than or equal to (<=)**

Comparison operators may be used to compare int, float, and str type objects to each other as well as *between* types, where it makes sense to do so. For example, an object of type int may be compared with an object of type float. As a bonus, string equality (and inequality) can be done with comparison operators, rather than having to use function/method calls like in Java/C/C++!



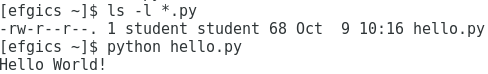
Notice that these statements result in Booleans; True or False. Let’s start writing learning how to write Python scripts, rather than executing singular commands. To exit Python’s interactive mode, type “**quit()**” and press ENTER.

**Time to Script with Hello World!**

The first script you’ll write in Python is the Hello World script. Open up your favorite text editor (I recommend Notepad++) and type in the following.



Click the Save button, name the file “hello.py”, and save it to a convenient location. Note that once you save the file as a Python file, the text editor will colorize the code for you. To run the script, just type “python hello.py” at the command line.



**Conditionals**

Python uses indentation for defining the extent of code blocks in conditionals and loops, rather than the opening and closing indicators/clauses that you see in high level languages like Java and C/C++. As we go through this crash course, you’ll see several mentions of “Indented Code Block”. When we say indented, we mean **four spaces** of indentation. Having to follow indentation can be a bit maddening, but on the bright side, it does force you to write code that’s potentially more readable than usual!

Today, we’ll look at if, if-else, and if-elif-else. Any and all of these conditionals may be nested. You may be asking yourself if there’s a switch statement in Python and the answer is a resounding no, although you could write code that “looks like” a switch statement if you really wanted to.

***If Statements***

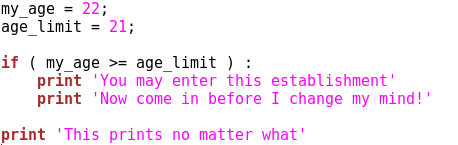
The if statement format is as follows.

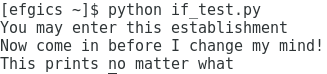
**if <logical expression> :**

**Indented Code Block**

The “logical expression” in an if statement may be a single condition, often using comparison operators (==, !=, >, >=, <, <=), or multiple conditions that are separated by logical operators (and, or, not). The colon (:) is required in the if statement and the extent of the indented code block is defined by when statement indentation ends.

Create a file called if\_test.py and enter the following code, then run it.





The if statement does not require parentheses around the logical expression, nor does it require the space between the right paren and the colon, but it makes your code more readable and less error prone to use these conventions. The extent of the if’s code block in this example is the first two print statements. Prove this to yourself by changing my\_age to 20 and running the script again.

***If-Else Statements***

The if-else statement format is as follows.

**if <logical expression> :**

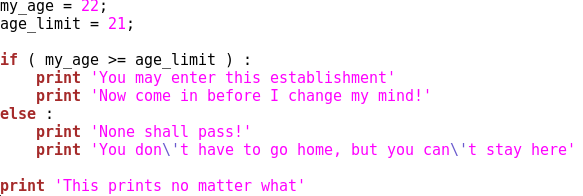
**Indented Code Block**

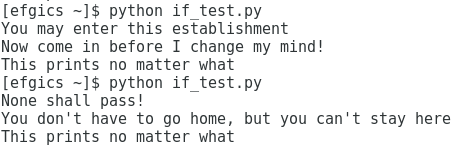
**else :**

**Indented Code Block**

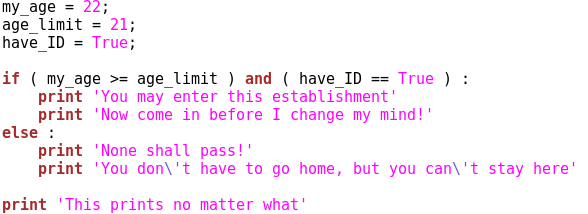
As you would expect, any code that you want to run if the logical expression evaluates to True should be indented and placed in between the if and else clauses, while code that should run if the logical expression that evaluates to False should be indented and placed below the else clause.

Let’s add on to the previous example in if\_test.py. Run this script twice; once with my\_age = 22 and once with my\_age = 20.

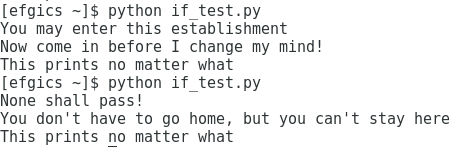




Now let’s add a second condition to the if-else statement to see how that works in practice.



This logical statement checks to see if the person also has an ID with them and only allows them in to the establishment if they meet or exceed the age limit **and** have an ID with them. Try running this script when have\_ID is True and then again when it’s set to False. The output should be as follows.

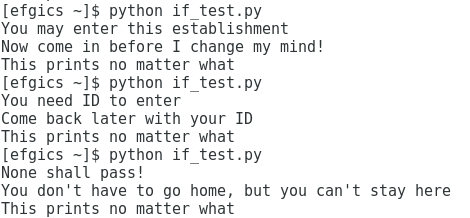


**Exercise**

Move the have\_ID == True condition inside the if clause to create a nested if. Now, when have\_ID is True, execute the same two print statements that were previously executed. However, when have\_ID is False, you should print two messages; “You need ID to enter” and “Come back later with your ID”.

Try running this script with the following three input combinations:

1. my\_age = 22 and have\_ID = True
2. my\_age = 22 and have\_ID = False
3. my\_age = 20



The above output is what you should get.

**Looping**

Python has while loops and for loops. While loops are just like you would expect, but for loops are quite different than Java/C/C++.

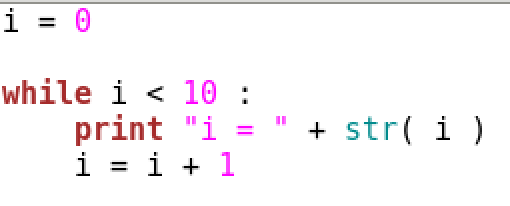
***While Loop***

The format of the while loop is as follows.

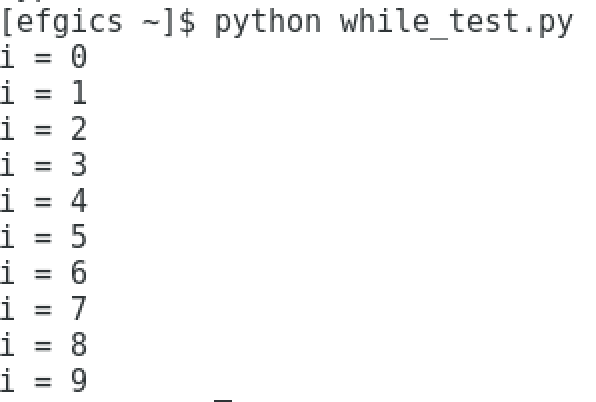
**while <logical expression> :**

**Indented Code Block**

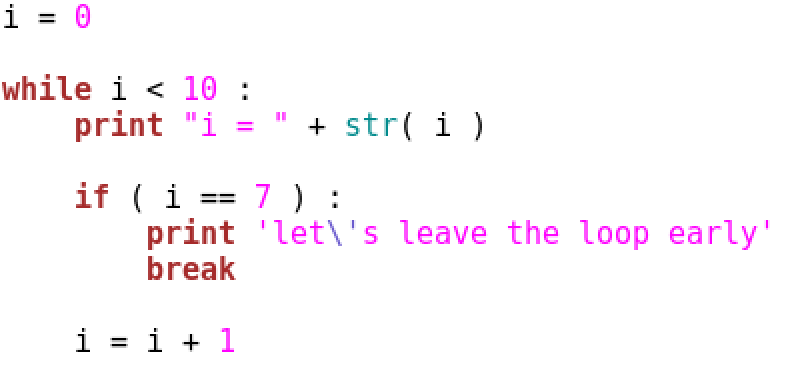
Create a new file called while\_test.py and open it in the text editor.



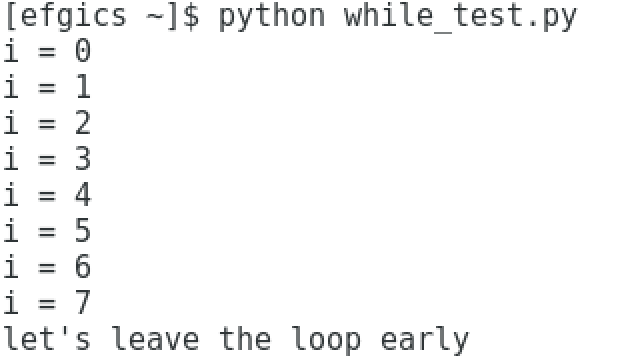
Notice the print statement uses concatenation (+) and converts the object stored in i from an int to a string so that it can be appended to the string “i = “. See the output below.



Python also includes built in loop control with the **break** and **continue** statements to allow the control of the script to leave the loop or go to the next iteration, respectively.



The break statement will cause the while loop to be exited immediately.



Notice that the script abruptly ends after the break statement is hit.

**Exercise**

Change the break statement in the above code to a continue and remove the print statement in the if. How many times does the loop run now? Is that what you expected to happen? Modify the loop so that it’s no longer infinite.

***For Loops***

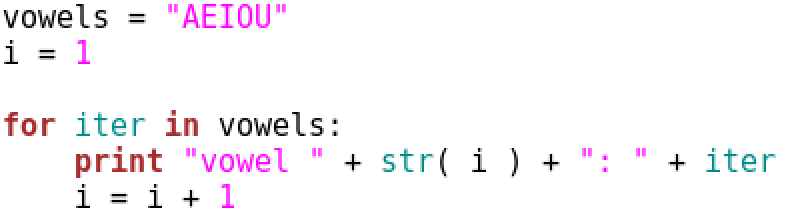
The format of a for loop is as follows.

**for iter in sequence :**

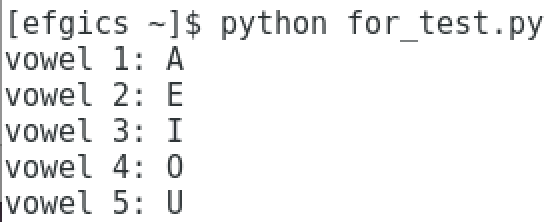
**Indented Code Block (iter)**

The for loop has an iterator variable that is assigned to each object in the sequence during each respective loop iteration. The sequence is a collection such as a string or list. During each loop iteration, the code in the loop has access to the current value of iter.

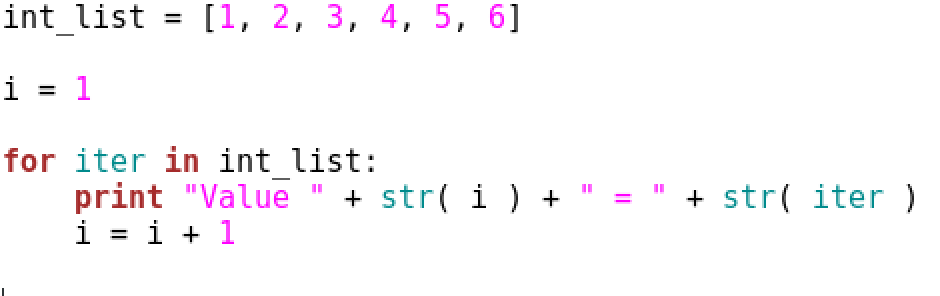
Create a new file called for\_test.py and open it in the text editor.



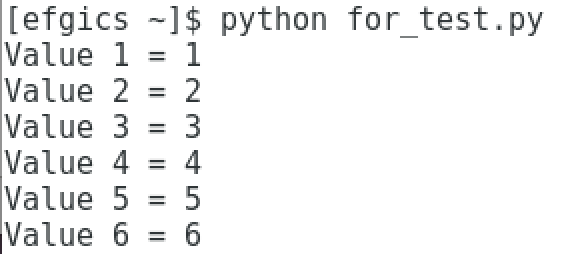
In this example, the “sequence” is a string containing vowels (yes, I know sometimes Y). In each loop iteration, iter will be assigned to a different vowel.



Let’s try using a collection called a List as a sequence in a for loop.



The List int\_list contains six integer objects and is iterated over in the for loop.



**Exercise**

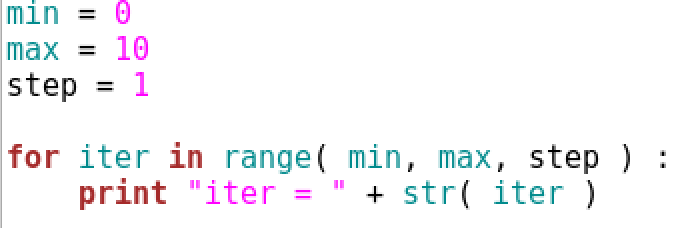
Modify the code above to compute the average of the values in int\_list. Hint: use the built in function len() to determine the number of elements in int\_list, rather than hard coding it or using i. The average should be 3.5. To get that answer, you’ll need to cast one of the objects in the computation of the average to a float.

***For Loops with a Range***

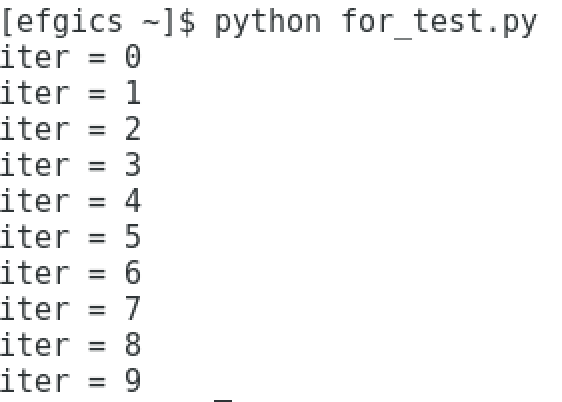
Using a range in Python is quite simple. The **range( start, stop [, step] )** function produces a list of integers beginning at start and ending before stop using the optional step to increment by. For example, to create a range from 0 to 9 by a step size of 1, you would use range(0, 10).



The above is output from interactive Python to confirm that range does produce a List.



Notice that it’s quite simple to use variables as part of the range.



What happens if you change the step size to 2 and then 3?

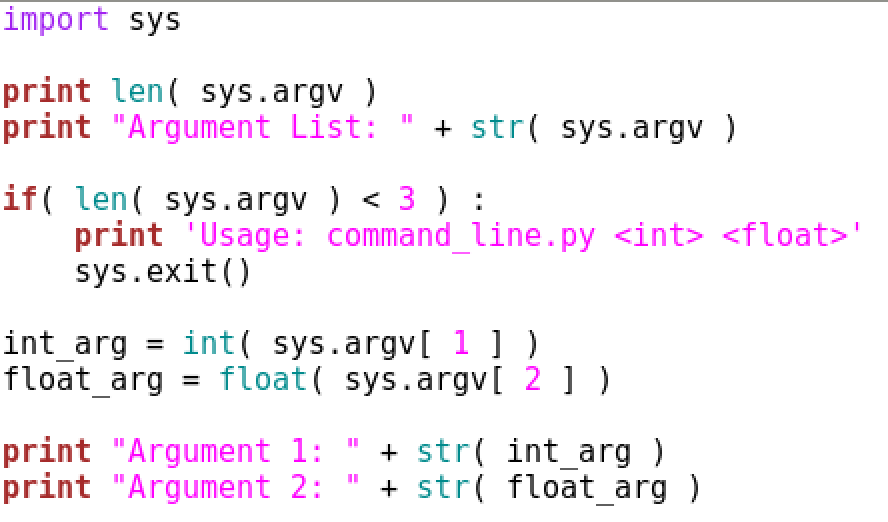
**Modules**

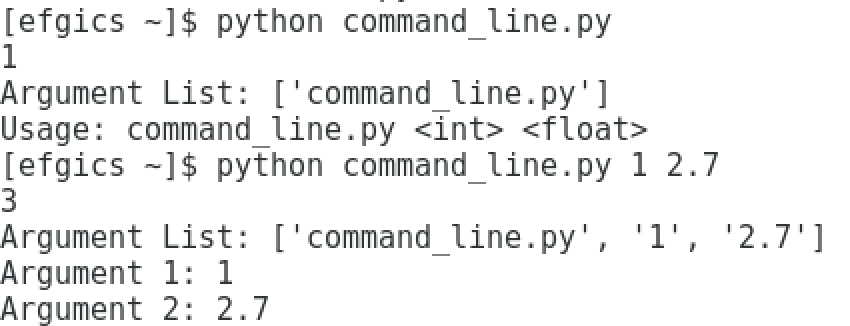
You’ll be seeing many modules related to plotting data, storing data in special formats, and if you choose, to perform data mining. Let’s start out with a really simple module example, the sys module.

***Using the sys Module to Read Command Line Arguments***

To read command line arguments, a module called **sys** must be imported into your Python script. Python modules are the equivalent of libraries, many of which are included with a standard Python installation. In order to use modules that are currently installed, the statement **import <module>** must be used at the top of the script.

Create a file called command\_line.py and open it with your favorite text editor (Notepad++ perhaps). We’ll be demonstrating how to read command line arguments using the **argv** object in the **sys** module. To see more objects and functions in the **sys** module, type “help(‘sys’)” in interactive Python.





The script begins by importing the sys library and prints out the length of argv, which indicates the number of command line arguments. Notice that when the script is run with no arguments, the length of argv is 1. This is because “command\_line.py” is considered the “first” command line argument. All additional command line arguments beyond the first are what the user provides. For the purposes of this script, the “usage” includes an int and a float.

The second and third arguments, argv[1] and argv[2], respectively, are cast to their intended types and printed to the console. If not enough command line arguments are provided, the script exits using a function call to the sys module called **exit()**.

**Functions**

The general format of a function is as follows:

**def fun( arg1, arg2, … ) :**

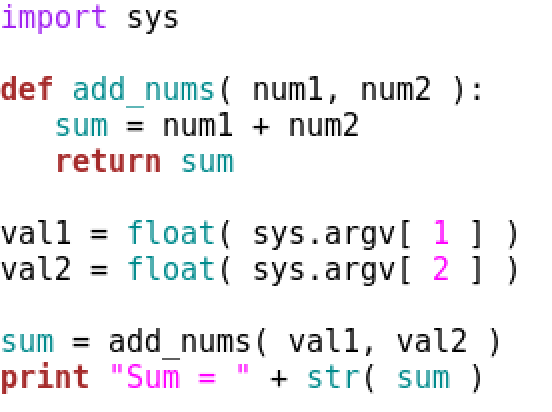
**Indented Code Block**

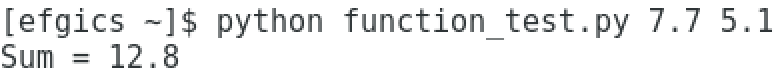
Functions should be defined as close to the top of the Python script as possible (below any imports). A function prototype includes the keyword **def**, followed by the name of the function and any arguments. Arguments are *not* required. The function “ends” at the end of the indented code block.

An argument that is “mutable”, such as a List, is passed by “pointer”, while “immutable” objects, such as int, float, and String, are passed by value. This means that changes to mutable objects made by a function are reflected in the calling function, while changes to immutable objects are not.

Values may be returned from a function using a **return** statement. You may choose to return one or more values from the function, return nothing, or not include a return statement at all. To call a function, the format used is **fun( arg1, arg2, … )**.

Create a file called function\_test.py in a text editor. We’ll be creating a single function called add\_nums that will add two numbers together and return the sum.

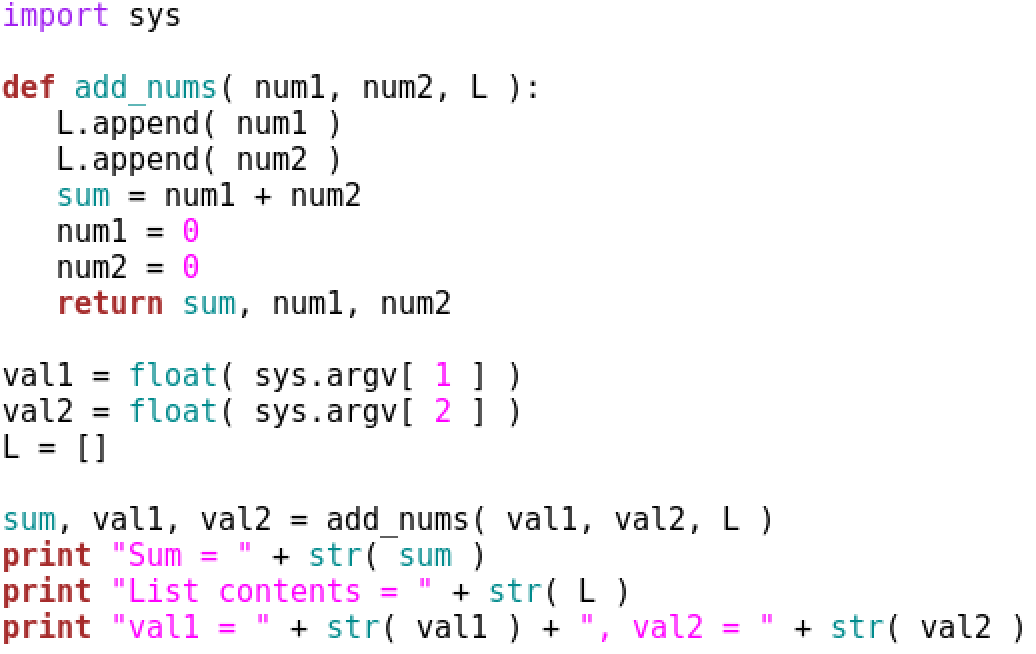


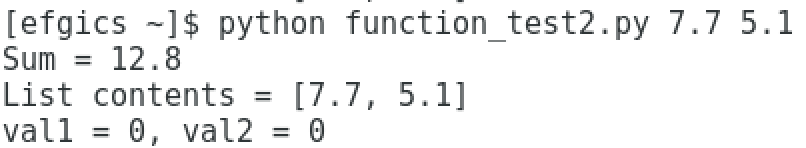


The sys module is imported so that the script can accept command line arguments, which are subsequently converted to floats and passed to the add\_nums function. Notice that add\_nums returns a float called sum. This variable is local to the function only and is different than the variable sum that is assigned to the result of calling the add\_nums function.

The contents of variables val1 and val2 are passed to the function by value. If those objects were modified by add\_nums, changes to them would not be reflected in the section of code that called the function because floats are immutable. Try to prove this to yourself by modifying add\_nums so that it sets num1 and num2 to 0 prior to the return statement and print out val1 and val2 after the function call.

Now rename function\_test.py to function\_test2.py so that we can demonstrate pass by object, how to use a return statement to return multiple values, and how to store all those returned values into variables in the calling function.





In this script, the add\_nums function accepts a third argument, a List L (you will learn more about lists in the next Python class exercise) and it returns three items; the sum as well as num1 and num2, which are both set to 0 after the sum is computed. Since floats are immutable, the only way we can reflect changes to them outside the function is to return them with a return statement.

After the command line arguments are cast to floats, an empty List L is created and is passed to add\_nums by object since a List is mutable. In add\_nums, we append the contents of num1 and num2 to the end of List L, resulting in a list containing [num1, num2]. Notice that the function did not return L. However, we can see in the print statement that outputs the list contents that the List L actually was modified by the function. At the same time, sum, val1, and val2 are set to the objects returned by the function.