Python Crash Course

For this course, you'll be learning Python 3 since Python 2.7 is now deprecated. Today, we'll look at how to use Python's impressive help system, variables, how to format basic statements, and look at some of the many data types that the language includes.

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 today. To enter Python's interactive mode, just type **python** on the command line.

```
[student@localhost ~]$ python3
Python 3.6.8 (default, Aug  7 2019, 17:28:10)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-39)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 
```

To enter the help system, type "**help()**".

```
>>> help()

Welcome to Python 3.6's help utility!

If this is your first time using Python, you should definitely check out
the tutorial on the Internet at https://docs.python.org/3.6/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules.  To quit this help utility and
return to the interpreter, just type "quit".

To get a list of available modules, keywords, symbols, or topics, type
"modules", "keywords", "symbols", or "topics".  Each module also comes
with a one-line summary of what it does; to list the modules whose name
or summary contain a given string such as "spam", type "modules spam".

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).

```
help> keywords

Here is a list of the Python keywords.  Enter any keyword to get more help.

and                 elif                if                  print
as                  else                import              raise
assert              except              in                  return
break               exec                is                  try
class               finally             lambda              while
continue            for                 not                 with
def                 from                or                  yield
del                 global              pass
```

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.

```
help> topics

Here is a list of available topics.  Enter any topic name to get more help.

ASSERTION           DEBUGGING           LITERALS            SEQUENCEMETHODS2
ASSIGNMENT          DELETION            LOOPING             SEQUENCES
ATTRIBUTEMETHODS    DICTIONARIES        MAPPINGMETHODS      SHIFTING
ATTRIBUTES          DICTIONARYLITERALS  MAPPINGS            SLICINGS
AUGMENTEDASSIGNMENT DYNAMICFEATURES     METHODS             SPECIALATTRIBUTES
BACKQUOTES          ELLIPSIS            MODULES             SPECIALIDENTIFIERS
BASICMETHODS        EXCEPTIONS          NAMESPACES          SPECIALMETHODS
BINARY              EXECUTION           NONE                STRINGMETHODS
BITWISE             EXPRESSIONS         NUMBERMETHODS       STRINGS
BOOLEAN             FILES               NUMBERS             SUBSCRIPTS
CALLABLEMETHODS     FLOAT               OBJECTS             TRACEBACKS
CALLS               FORMATTING          OPERATORS           TRUTHVALUE
CLASSES             FRAMEOBJECTS        PACKAGES            TUPLELITERALS
CODEOBJECTS         FRAMES              POWER               TUPLES
COERCIONS           FUNCTIONS           PRECEDENCE          TYPEOBJECTS
COMPARISON          IDENTIFIERS         PRINTING            TYPES
COMPLEX             IMPORTING           PRIVATENAMES        UNARY
CONDITIONAL         INTEGER             RETURNING           UNICODE
CONTEXTMANAGERS     LISTLITERALS        SCOPING
CONVERSIONS         LISTS               SEQUENCEMETHODS1
```

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 than 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.

```
>>> test = 1
>>> id(test)
8020312
>>> test = 10
>>> id(test)
8020096
```

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.

```
>>> test = 10
>>> type(test)
<class 'int'>
>>> test = 1.234
>>> type(test)
<class 'float'>
```

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.

```
>>> 10 * 7
70
>>> ( 10 * 7 ) + 16
86
>>> 2 ** 10
1024
```

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.

```
>>> var = 2
>>> power = 8
>>> val = 2 ** power
>>> print(val)
256
>>> id(val)
140658175149728
>>> new_val = val
>>> id(new_val)
140658175149728
```

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 then raised 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 (140658175149728) 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.

```
>>> boolean = False
>>> type(boolean)
<class 'bool'>
>>> boolean == True
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.

```
>>> val = 2
>>> print("val is " + str(val) + " and is type " + str(type(val)))
val is 2 and is type <class 'int'>
>>> val = 6.7
>>> print("val is " + str(val) + " and is type " + str(type(val)))
val is 6.7 and is type <class 'float'>
>>> val = 4 + 5j
>>> print("val is " + str(val) + " and is type " + str(type(val)))
val is (4+5j) and is type <class 'complex'>
```

We’ve seen int and float sprinkled in throughout this lesson, 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. Notice in the print statements that the “+” operator is used for string concatenation and any non-strings are cast as strings using the str() function. We’ll go into more depth on this in future lessons.

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.

```
>>> string = 'This is a string'
>>> string
'This is a string'
>>> type(string)
<class 'str'>
```

When in interactive mode, 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”.

```
>>> string = "This is a string"
>>> string[ 0 ]
'T'
>>> string[ 1 ]
'h'
>>> len( string )
16
>>> string[ len( string ) - 1 ]
'g' _
```

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've seen some basic arithmetic operators and you'll see a couple more, including comparison and logical 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 (*)**
- **Division (/) – performs "real division", with the result being a float**
- **Floor Division (//) – rounds down to the nearest whole number (equivalent to integer division)**
- **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.

```
>>> a = 5
>>> b = 4
>>> a + b
9
>>> a - b
1
>>> a * b
20
>>> a / b
1.25
>>> a // b
1
>>> a % b
1
>>> a ** b
625
```

Notice that to get a floating-point result for a division, the / operator is used, while the // operator returns an integer result.

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++!

```
>>> str1 = 'Hi!'
>>> str2 = 'Hi!'
>>> str1 == str2
True
>>> str2 = 'Hi!!!'
>>> str1 == str2
False
```

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.

Logical Operators

Logical operators allow you to make decisions using multiple conditions. Like comparison operators, these operators produce Boolean results of True and False.

- **(condition1) and (condition2) - evaluates to True if both conditions are true**
- **(condition1) or (condition2) - evaluates to True if one or both conditions are true**
- **not (condition) – evaluates to the opposite result of the condition**

Try these examples below for yourself by setting a to 5, b to 7, and c to 8.

```
>>> ( a < b ) and ( a < c )
True
>>> ( a < b ) or ( a < c )
True
>>> not( a < b )
False
```

You may use logical operators to evaluate as many conditions as you want and use parentheses to define the desired order of operations for evaluating a series of conditions.

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 and type in the following.

```
# This is the Hello World script for Python

print("Hello World!")
```

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.

```
[student@localhost ~]$ python3 hello.py
Hello World!
```

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.

```
my_age = 22;
age_limit = 21;

if ( my_age >= age_limit ) :
    print("You may enter this establishment")
    print("Now come in before I change my mind!")

print("This prints no matter what")

[student@localhost ~]$ python3 if_test.py
You may enter this establishment
Now come in before I change my mind!
This prints no matter what
```

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.

```
my_age = 22;
age_limit = 21;

if ( my_age >= age_limit ) :
    print("You may enter this establishment")
    print("Now come in before I change my mind!")
else :
    print("None shall pass!")
    print("You don\'t have to go home, but you can\'t stay here")

print("This prints no matter what")
```

```
[student@localhost ~]$ python3 if_test.py
You may enter this establishment
Now come in before I change my mind!
This prints no matter what
[student@localhost ~]$ python3 if_test.py
None shall pass!
You don't have to go home, but you can't stay here
This prints no matter what
```

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

```
my_age = 22;
age_limit = 21;
have_ID = True;

if ( my_age >= age_limit ) and ( have_ID == True ) :
    print("You may enter this establishment")
    print("Now come in before I change my mind!")
else :
    print("None shall pass!")
    print("You don\'t have to go home, but you can\'t stay here")

print("This prints no matter what")
```

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.

```
[student@localhost ~]$ python3 if_test.py
You may enter this establishment
Now come in before I change my mind!
This prints no matter what
[student@localhost ~]$ python3 if_test.py
None shall pass!
You don't have to go home, but you can't stay here
This prints no matter what
```

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

```
[efgics ~]$ python if_test.py
You may enter this establishment
Now come in before I change my mind!
This prints no matter what
[efgics ~]$ python if_test.py
You need ID to enter
Come back later with your ID
This prints no matter what
[efgics ~]$ python if_test.py
None shall pass!
You don't have to go home, but you can't stay here
This prints no matter what
```

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.

```
i = 0

while i < 10 :
    print("i = " + str( i ))

    i = i + 1
```

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.

```
[student@localhost ~]$ python3 while_test.py
i = 0
i = 1
i = 2
i = 3
i = 4
i = 5
i = 6
i = 7
i = 8
i = 9
```

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.

```
i = 0

while i < 10 :
    print("i = " + str( i ))

    i = i + 1

    if ( i == 7 ) :
        print("Let\'s leave the loop early")
        break
```

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

```
[student@localhost ~]$ python3 while_test.py
i = 0
i = 1
i = 2
i = 3
i = 4
i = 5
i = 6
Let's leave the loop early
```

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 becomes 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.

```
vowels = "AEIOU"
i = 1

for iter in vowels:
    print("vowel " + str( i ) + ": " + iter)
    i = i + 1
```

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.

```
[student@localhost ~]$ python3 for_test.py
vowel 1: A
vowel 2: E
vowel 3: I
vowel 4: O
vowel 5: U
```

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

```
int_list = [1, 2, 3, 4, 5, 6]

i = 1

for iter in int_list:
    print("Value " + str( i ) + " = " + str( iter ))
    i = i + 1
```

The List int_list contains six integer objects and is iterated over in the for loop.

```
[student@localhost ~]$ python3 for_test.py
Value 1 = 1
Value 2 = 2
Value 3 = 3
Value 4 = 4
Value 5 = 5
Value 6 = 6
```

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.

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).

```
>>> range( 0, 10 )
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

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

```
min = 0
max = 10
step = 1

for iter in range( min, max, step ) :
    print("iter = " + str( iter ))
```

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

```
[student@localhost ~]$ python3 for_test.py
iter = 0
iter = 1
iter = 2
iter = 3
iter = 4
iter = 5
iter = 6
iter = 7
iter = 8
iter = 9
```

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. 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.

```
import sys

print(len( sys.argv ))
print("Argument List: " + str( sys.argv ))

if( len( sys.argv ) < 3 ) :
    print("Usage: command_line.py <int> <float>")
    sys.exit()

int_arg = int( sys.argv[ 1 ] )
float_arg = float( sys.argv[ 2 ] )

print("Argument 1: " + str( int_arg ))
print("Argument 2: " + str( float_arg ))

[student@localhost ~]$ python3 command_line.py
1
Argument List: ['command_line.py']
Usage: command_line.py <int> <float>
[student@localhost ~]$ python3 command_line.py 1 2.7
3
Argument List: ['command_line.py', '1', '2.7']
Argument 1: 1
Argument 2: 2.7
```

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.

```
import sys

def add_nums( num1, num2 ):
   sum = num1 + num2
   num1 = 0
   num2 = 0
   return sum

val1 = float( sys.argv[ 1 ] )
val2 = float( sys.argv[ 2 ] )

sum = add_nums( val1, val2 )
print("Sum = " + str( sum ))
print("val1 = " + str( val1 ) + ", val2 = " + str( val2 ))

[student@localhost ~]$ python3 function_test.py 7.7 5.1
Sum = 12.8
val1 = 7.7, val2 = 5.1
```

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.

```
import sys

def add_nums( num1, num2, L ):
   L.append( num1 )
   L.append( num2 )
   sum = num1 + num2
   num1 = 0
   num2 = 0
   return sum, num1, num2

val1 = float( sys.argv[ 1 ] )
val2 = float( sys.argv[ 2 ] )
L = []

sum, val1, val2 = add_nums( val1, val2, L )
print("Sum = " + str( sum ))
print("List contents = " + str( L ))
print("val1 = " + str( val1 ) + ", val2 = " + str( val2 ))

[student@localhost ~]$ python3 function_test2.py 7.7 5.1
Sum = 12.8
List contents = [7.7, 5.1]
val1 = 0, val2 = 0
```

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.