Bootcamp: Python - Module 1

John Rajadayakaran Edison, Sara Miner More,
Eli Sherman, Musad Haque, Soumyajit Ray

Last Edited: September 2023

Contents

1 Introduction to Python

Welcome to the Bootcamp: Python course! This document is the reading material for Module 1 of the course. It is far from being a comprehensive guide to programming in Python. Since you have already acquired a lot of experience and expertise in programming from one of the Gateway Computing courses, we have included just the bare minimum of topics that will be required to start using Python as your primary programming language. We have included a few exercise problems in each section and links in the document to resources online which discuss topics in more detail.

Aside from the fact that this course might be a requirement for other core courses in your department, are there any benefits to learning a new programming language? And why learn Python among the many languages out there, you might ask. Why not LOLCODE or Whitespace or ArnoldC? Well, here are two main reasons we find Python useful.

1. Python is designed to enable the programmer to write code that is highly readable and intuitive to understand. Python code very closely resembles pseudocode and therefore is easy to (i) learn, (ii) code, (iii) test, and (iv) maintain.

2. A vast collection of open-source Python packages are available for a wide variety of scientific computing needs. With very little effort these packages can be integrated into your code, enabling rapid development of software. Want to run a molecular simulation? do symbolic math? analyze large datasets? build a computer game? There's always a package for that.

So let's get going! We'll have you begin by installing Python on your machine. Please go to https://www.anaconda.com/products/individual to install the ANACONDA scientific platform. The ANACONDA distribution includes a collection of open source software, including the Python interpreter, the Spyder Integrated Developer Environment (IDE) and the open-source collection of packages known as SciPy. In this course, you will be using the Spyder IDE to edit, debug and run code. Instructions for launching Spyder can be found at https://docs.anaconda.com/anaconda/user-guide/tasks/integration/spyder/. If you have trouble running the latest version of ANACONDA or Spyder on your machine, you can go to https://repo.anaconda.com/archive/ to find older versions of the distribution.

Once Spyder is launched, you will notice that the window has three main sections, as shown in Figure 1. Most important to us are the editor on the left and an IPython console on the bottom right. The IPython console can evaluate any valid Python statements, and we often use it to test out the value of a particular expression. Right now, go ahead and take part in the standard *I'm-learning-a-new-programming-language* ritual of writing your first "Hello, World!" program in Python. Simply type the statement `print("Hello, World!")` in the IPython console (top right side) and hit enter. Alternatively, use the editor window (left side) and then hit F5 or click on the menu item Run → Run. No matter which way you execute the line, the output `Hello, World!` should be visible in the Python console. Note also that any line that begins with a # symbol in Python is treated as a comment line. So we can also type in a two-line script like the one below, using a comment to explain

what is happening. (Here, and in other code listings in this document, we'll prepend line numbers for ease of reference.)

```
1 # The line below will output a greeting to the console
2 print("Hello, World!");
```

The implementation of Python that comes with the Anaconda distribution was written using the C Programming language and is also called CPython. There are also other implementations of Python e.g. JPython (Java Python). Python is an interpreted language, so when you run Python code, an interpreter reads the code line by line and turns it into something called a bytecode which is executed on the Python Virtual Machine (PVM). The PVM is responsible for translating the bytecode into instructions that can run on your hardware. Since the code runs on a virtual machine, it is portable and the programmer need not worry about developing code for specific hardware setups or operating systems.

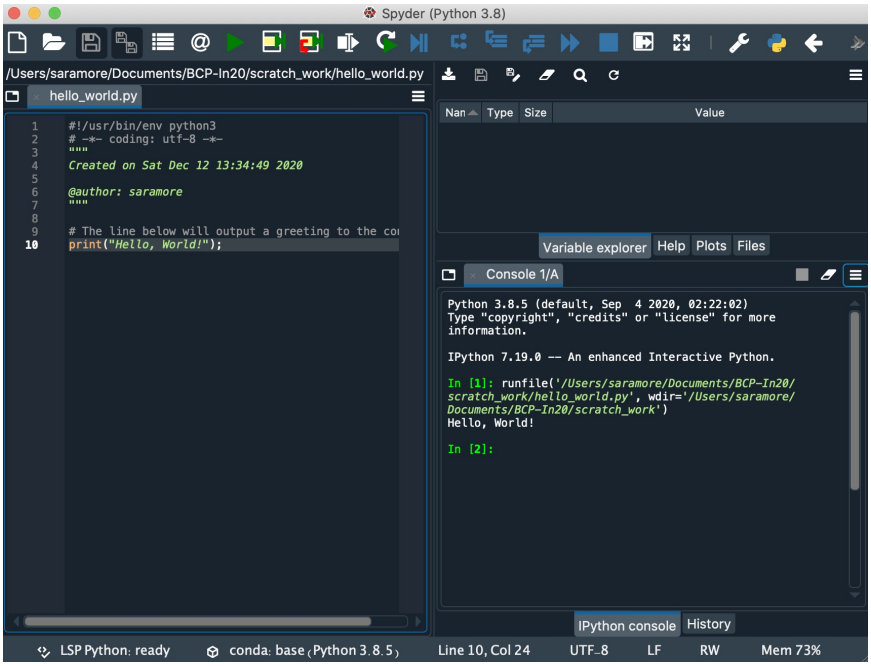

Figure 1: Snapshot of the Spyder IDE. The region within the thin blue outlined box on the left is the editor. The console is at the bottom right.

2 Output using the `print` function

In examples below, we will be using Python's built-in `print` function. This function can be used to output to the console the result of an expression or the value in a variable. The `print` function can accept any number of arguments to output, and will, by default,

separate them with a single space. Also by default, a newline character is printed to the output after each `print` statement. The `print` function has two optional keyword-based arguments `end` and `sep` which are used to format the output. The example below illustrates how these arguments can be used.

```
# Declare and initialize three integer variables
x=1; y=2; z=7
# Output their values and some literal strings too
print("hello",x,y,z,"goodbye")

# The sep argument can be used to output a specified string
# between the values being printed
print("Ratio",x,y,z,sep=":")

# The end argument is used to select a different string in place
# of the newline character to end one statement's output
print("x =",x, end = " ")
print("y =",y, end = " ")
print("z =",z, end = " ")
```

Below is the output generated in the console by the example script above.

```
hello 1 2 7 goodbye
Ratio:1:2:7
x = 1 y = 2 z = 7
```

Try it yourself:

Don't just read the output shown above and take our word for it that this is how things work! Try typing in and running the code yourself. Play around with the `print` function options for a minute or two to get comfortable with them.

3 Variables and Datatypes

A first step before processing information using computer code is often to store data values using variables. A programming language typically classifies data into different *types* and if you have learned C or Java before you might be familiar with types like integers, strings and floats. Python offers several built-in datatypes. Below we will learn about two numeric data types `int` and `float` and a sequence datatype (`list`). In the next module, we will learn a few more advanced data structures.

3.1 Integers and Floats

The `int` datatype is used to store whole numbers of any size. In most programming languages, the integer datatype comes in several flavors which differ by the amount of memory used. For example, C allows for 2 byte, 4 byte and 8 byte integers each with its own upper and lower bound (e.g. 4 byte integers in C can take values between -2,147,483,648 to 2,147,483,647). Python, however, offers only a single type of integer. There is no upper or lower bound in value for a Python integer. In your IPython console, try describing some mathematical expressions that involve larger integers (e.g. type `3**121`, which will compute the value 3 raised to the 121st power).

The `float` datatype is used to store real numbers. In Python, `float` values are stored in double precision by default; they have 16 digits (53 bits) of precision.

3.1.1 Initialization

How do we assign a value to a variable? Let's say we need to store the number of goals scored by Bayern Munich against Barcelona in a variable titled `goals_scored`. This is achieved using the assignment statement `goals_scored=8`. Note that we do not mention anything about the *type* of data (integer) stored in the variable `goals_scored`. In other words, we do not declare the datatype of the variable `goals_scored`. Python is a *dynamically-typed* language, which means that it infers datatypes during runtime. Below is an example that demonstrates this. We will use two built-in functions `type` and `id`. The first gives the datatype of a variable and the second gives its location in memory.

```
1 goals_scored = 8
2 print(type(goals_scored))
3 print(id(goals_scored))
```

```
<class 'int'>
4305263200
```

Upon reading the statement in line number 1 the interpreter i) stores the value `8` in memory ii) attaches a label `goals_scored` to this location in memory. Here are a few different ways of initializing variables in Python.

`x=y=5`	The number 5 is stored in the memory and is referred to by two variables (labels) x and y. These variables are aliases; they do not hold separate copies of 5!
`x,y=5,7.0`	Multiple assignments are possible within a single line; x gets the integer 5 and y gets the float 7.0

3.1.2 A few rules and conventions to remember:

- Variable names in Python are case-sensitive, i.e `goals_scored` and `Goals_scored` are not the same variable. They cannot include whitespace and they cannot be Python keywords.
- Only letters, numbers and underscores can be used in variable names, with the first character of each variable restricted to being a letter or an underscore. It is recommended to i) use descriptive names for variables ii) use lower case and separate words using an underscore ("snake case"). As we will see later in this course, variables that begin with an underscore are treated differently.
- All variables in Python store references to objects. (All data types in Python are treated as classes, including the primitive types.) A Python variable itself does not have a type and can therefore store a reference to any type of object.
- Note that variables can be reassigned data values of a different data-type anywhere within the program.
- Once initialized, a variable in a Python is visible everywhere in the script. In other words, Python variables have *global scope*.
- Python does not support constant variables, in that there is no way to specify that a variable's value may never be changed. However, by convention, Python programmers commonly use all capital letters to specify variables that will play the role of named constants.

The run-time inference of data types offers great flexibility. However it comes at a cost. A frequently-occurring mistake is shown below.

```
1 test_passed = False
2 score = 70
3 if (score>50):
4     test_pased = True
5 print("Did I pass the test?", test_passed)
```

Here is the output of the script shown above:

```
Did I pass the test? False
```

Notice the typo in line 4 - a misspelling of the variable name. Compilers in statically-typed languages typically catch such errors. In such languages, for example, C or C++ for example we would have declared the variable `test_passed` to be a boolean variable `bool test_passed;`. Later in the code when the compiler looks at line number 4, it

does not recognize the variable `test_pased` and reports an error. However, Python finds the above piece of code syntactically correct. When the interpreter looks at line 6, it simply creates a new variable `test_pased` and therefore the reported results are erroneous.

Try it yourself:

Define two variables a=5 and b=7. Write a one-line Python expression to swap the values of variables a and b.

3.1.3 Immutability of variables

In Python, integers, floats, and strings are immutable. That is, once an object is created, its value cannot be changed. This appears to present a major inconvenience. If integers and floats are immutable, how can one change a value? Let's look at an example where we modify the value of a variable `x` that stores an integer.

```
1 x = 4234
2 y = x
3 print("Address of x :",id(x))
4 print("Address of y :",id(y))
5 x = x + 1
6 y = "Hi"
7 print("Address of x post increment:",id(x))
8 print("Address of y post edit :",id(y))
```

```
Address of x : 4349653840
Address of y : 4349653840
Address of x post increment: 4317580912
Address of y post edit : 4348113448
```

Here is how the Python virtual machine deals with statements 1,2,5 and 6. First the value 4234 is stored in memory, and the label `x` is assigned to it. In statement 2 another label `y` is assigned to the same memory location. In line 5, `x` is incremented. Now a new value 4235 is stored in memory and the label `x` is now attached to it. Label `y` continues to be attached to the memory location that stores 4234. Once `y` is assigned to the string `"Hi"`, the location where value 4234 was stored is no longer associated with a label, so it is treated as garbage and cleared from memory.

3.2 Lists

Often, we have a need to store and process a collection of multiple items rather than working with, say, one string or a single number at a time. Python offers several built-in data types

for this general purpose, each with its own unique features. In a later module, you'll learn about `set`, `tuple`, and `dictionary` data types, but in this module, we start with the Python `list` type.

A Python `list` is a data type used to store a collection of items. It can be created by specifying a set of items enclosed within square brackets and separated by commas. The objects that make up a `list` can belong to more than one data type. A `list` can also be a collection of `list` objects. Finally, we note that a `list` is a mutable collection of objects, meaning that unlike `int`, `float` or `string` objects in Python, a `list` object *can* be modified at runtime.

3.2.1 Intializing lists

Here are some simple examples that show how to initialize a list.

```
1
2  # Initialize an empty list
3  my_empty_list = []
4
5  # Create a homogeneous list of integers
6  my_test_scores = [20, 40, 40, 60]
7
8  # Create a list of length 12 containing 3 copies of values in the list
9  # my_test_scores, e.g. [20, 40, 40, 60, 20, 40, 40, 60, 20, 40, 40, 60]
10 my_repeated_scores = my_test_scores*3
11
12 # Create a heterogeneous list
13 car_info = ["Honda", "Civic", 2015]
14
15
```

We will learn more elegant ways of creating lists towards the end of this module.

3.2.2 Accessing list features and list items

The size of a list can be obtained using the built-in function `len`. Each item in a list is assigned an index value. The first item is assigned an index value of 0. The index increments by 1 for every additional element. The table below shows how to access and modify individual items of a sample list `my_list = [2, 6, 10, 11]`.

`n = len(my_list)`	This function `len` on the right returns the number of items in the list named `my_list` which is 4 and stores the result in the variable `n`.
`a = my_list[1]`	Read the item in index position 1 and store it in variable `a`. In this example, a will refer to value 6. Note that Python uses zero-based indexing.
`b = my_list[-2]`	Items of a list can also be accessed using negative indices. Index position -1 corresponds to the last element of the list and index `-len(my_list)` gives the first element of the list. In this example, b will refer to value 10.
`my_list[1] = 18`	Assign value 18 at index position 1, replacing 6.

3.2.3 Slicing

So far, we have only looked at cases where a single element of a list was accessed or modified. It is also possible to access or modify a subset or a slice of a list in a single statement. We can select a slice by specifying three parameters within square brackets, with parameter values separated by a colon (`:`). The parameters are `start`, `stop` and `step`. The syntax is the name of the list followed by [start:stop:step]. All three parameters are optional, and can be left unspecified. By default the start value is 0, stop is given by the length of the list and step equals one. Note that the element in position stop is not included in the slice. Let's look at a few slices of the list `my_list = [8,9,1,3,7,2,5]`

```
# Initialize a list of seven integer values; indices go from 0 to 6.
my_list = [8,9,1,3,7,2,5]
# List containing all values starting from index 3, namely [3, 7, 2, 5].
print(my_list[3:])
# List containing values at indices 1, 2 and 3, namely [9, 1, 3].
print(my_list[1:4])
# List containing values at indices 1 and 4 only, namely [9, 7].
print(my_list[1:6:3])
```

Output:

```
[3, 7, 2, 5]
[9, 1, 3]
[9, 7]
```

Here is one more example that shows how a slice can also be modified. It's also a great opportunity to point out the top-right portion of the Spyder window, where we can monitor the current value and type of each variable we've initialized.

Try it yourself

In the top-right portion of your Spyder window, click on the tab named 'Variable explorer', and watch what gets created as you type in and execute the following script. The Variable explorer is a valuable tool for debugging in Python.

```
# Initialize a list
my_list = [8,9,1,3,7,2,5]
# Use * to append together three copies of a one-element list
# containing 0, and store this in a new variable
three_zeroes = [0]*3
# Now, change every other element of my_list to zero
my_list[1::2] = three_zeroes
print(my_list)
```

Output:

```
[8, 0, 1, 0, 7, 0, 5]
```

Try it yourself

- Let's assume we have a list `list_1 = [1,5]`. What would you guess is the value of the expression `list_1*4`? Type it in and see.
- Further, assume we have `list_2 = [5,3]`. What would you guess is the value of `list_1 + list_2`? Check.
- Explore how the built-in functions `sum`, `min` and `max` can be used on lists. See https://docs.python.org/3/library/functions.html if you get stuck.

3.2.4 A few useful methods to modify lists

You can a description of a set of methods that can be used to process lists at https://docs.python.org/3/tutorial/datastructures.html. Below are a few examples that demonstrate the use of some of these methods.

```
# Initialize a list
my_list = [8,9,1,3,7,2,5]

#Add an element to the end of the current list
my_list.append(4)
```

```
print("Append 4 to the end: ",my_list)

#Remove the first occurrence of element 2 from the current list
my_list.remove(2)
print("Remove the value 2: ", my_list)

# Delete the item at the 1st index position
del my_list[1]
print("Delete item 1: ",my_list)

# Reverse the items of a list
my_list.reverse()
print("List post reversal: ",my_list)

# Sort the items of a list
my_list.sort()
print("List after sorting: ",my_list)

```

```
Append 4 to the end:  [8, 9, 1, 3, 7, 2, 5, 4]
Remove the value 2:  [8, 9, 1, 3, 7, 5, 4]
Delete item 1:  [8, 1, 3, 7, 5, 4]
List post reversal:  [4, 5, 7, 3, 1, 8]
List after sorting:  [1, 3, 4, 5, 7, 8]
```

Try it yourself:

Can you reverse a list without using the `reverse` method? Hint: Use slicing.

3.2.5 Copying a list

As we mentioned earlier, lists are mutable objects in Python. Let's attempt to copy the contents of one list over to another.

```
list_one = [2, 4, 6]
list_two = list_one
list_one.append(5)
print("List 1 : ", list_one)
print("List 2 : ", list_two)
print("List 1 address : ", id(list_one))
print("List 2 address : ", id(list_two))

```

```
List 1 :  [2, 4, 6, 5]
List 2 :  [2, 4, 6, 5]
List 1 address :  4332897160
List 2 address :  4332897160
```

In the above example line number 2 assigns `list_one` to `list_two`. Since list variables hold references to objects, what this statement effectively does is add a second label `list_two` to the memory location that contains the data. As we can see from the results of the `id` function calls, both `list_one` and `list_two` refer to the same memory location. Therefore, modifying `list_one` automatically modifies `list_two`. If we want to avoid this situation and make a separate copy of the values, we can specifically ask for this behavior using the slicing operator but omit the operands, as shown below:

```
1 list_one = [2, 4, 6]
2 list_two = list_one[:]
3 list_one.append(5)
4 print("List 1 : ", list_one)
5 print("List 2 : ", list_two)
6 print("List 1 address : ", id(list_one))
7 print("List 2 address : ", id(list_two))
8
```

```
List 1 :  [2, 4, 6, 5]
List 2 :  [2, 4, 6]
List 1 address :  4331848584
List 2 address :  4332168328
```

On line 2, the command `list_one[:]` returns the "slice" of `list_one` that matches the original exactly. As an alternative, one could instead use the `copy()` method: `list_two = list_one.copy()`.

What if a list was composed of mutable objects? Let us assume we have a list of lists. How could we copy the contents of such a list to another list? Let's try it out.

```
1 list_i = [2, 4, 6]
2 list_j = [1, 6, 9]
3 list_one = [list_i, list_j]
4 list_two = list_one[:]
5 print("Lists copied")
6 print(list_one)
7 print(list_two)
8 list_i.append(2)
```

```
print("One of the member lists is modified")
print(list_one)
print(list_two)

```

```
Lists copied
[[2, 4, 6], [1, 6, 9]]
[[2, 4, 6], [1, 6, 9]]
One of the member lists is modified
[[2, 4, 6, 2], [1, 6, 9]]
[[2, 4, 6, 2], [1, 6, 9]]
```

We might have guessed that in the above example using `[:]` would copy all of the contents to a new location, implying that modifying one of the sublists of `list_one` would not affect `list_two`. However, that does not seem to be the case.

Try it yourself:

Learn about the usage of the `deepcopy` method and use it on the example above and compare the difference. Stuck? Check out https://docs.python.org/3/library/copy.html or other Internet resources on Python `deepcopy`.

3.3 Strings

The Python `str` type allows storage of text or strings, as in:
`my_name="Sir Lancelot the Brave"`
Happily, just like in lists, individual elements or subsets of a string can be accessed by slicing. Additionally, the Python library provides numerous methods to modify strings (see a long list at https://docs.python.org/3.8/library/stdtypes.html#text-sequence-type-str). We have demonstrated a few examples below, but strongly encourage you to familiarize yourself with additional methods listed in the above reference under the header "String Methods".

```

#Capitalize string
print("guido van rossum".capitalize())

username = "guido van rossum"
#All upper case string
print(username.upper())
# Check if a string is upper case
print(username.isupper())
```

```
# Find and replace a character
print(username.replace("o","a"))

```

```
Guido van rossum
GUIDO VAN ROSSUM
False
guida van rassum
```

3.4 Obtaining user input and type casting

3.4.1 input

Before we learn to build expressions in Python, let's learn about a very useful built-in function: `input`. The `input()` function is used for obtaining data from the user. The typical argument to an `input` function is a prompt string which offers information to the user on the input required and is printed to the console. The function then reads the input provided by the user and return this value as a string. Here is an example: `username = input("Enter your user name:")`. The variable `username` will refer to the string input by the user.

3.4.2 Type casting

Suppose we need an integer as input from the user. Since the built-in `input` function always returns a string, we need to cast or convert data from one type to another. Functions `int()`, `float()` and `str()`, as their names imply, are used to convert data to integer, float and string respectively. Here are some examples:

```

# Convert an integer and concatenate with a string
print ("Zip code is " + str(21211))
# Convert an integer to a float value
print ("Taxable income ", float(23000))
# Convert a string to an integer
print ("Tax return ", 3*int("400"))

```

```
Zip code is 21211
Taxable income  23000.0
Tax return  1200
```

3.5 Datatypes as objects

In Python, all data types including int, float and strings are defined using classes. The expression `age = 8`, creates an object of type integer and stores the reference in variable `x`. As members of a class, these objects can access member functions or methods defined in the class. We have seen a few of these already e.g `append` for lists, `upper`, `join` etc. for strings.

Try it yourself:

`list()` can be used to cast any object that be iterated over, into a list. Try using this on a string, example `list("Hello")`.

4 Expressions

Now that we have learned to store a few types of values, let's construct expressions to manipulate them. For this purpose, we need to know the list of operators available in Python.

4.1 Binary operations

Here are a few standard binary operators : addition $(+)$ subtraction $(-)$ multiplication $(*)$ division $(/)$ floor division $(//)$ exponentiation $(**)$. As implied by the name, binary operators require two operands (variables or data). Since you are familiar with the usage of these operators, we are going to just mention a few points worth keeping in mind when constructing expressions with binary operators.

- Python evaluates expressions left to right. For assignment operations the right hand side is first evaluated before assigning it to the variable on the left.
- When, as frequently happens, more than one operator is used in an expression, the order of precedence by which execution occurs is: Parentheses, Exponentiation, Multiplication, Division, Addition, Subtraction (PEMDAS).
- Python allows construction of expressions between mixed numeric types (float and integer). The result always has the data type of the variable with the higher precision.
- One unique operator found in Python is the floor division operator $(//)$. This operator performs the usual division operation and then rounds the result towards negative infinity. Note the division operator $(/)$ always returns a float as the output. Some examples below demonstrate the difference between the floor division operator $(//)$ and the division operator $(/)$.
- When a number (`int` or `float`) is raised to a negative power using the exponentiation operator, the data type of the result is automatically converted to float.

```
# The result to which variable x refers will be of type float
x = ( 5.0 + 2 ) * 3
print (x, type(x))

# Usage of the // operator
print ( "5//2 = " ,5//2)
print ( "-5//2 = ",-5//2)
print ( "-5.0//2 = ", -5.0//2)
print ( "-5/2 = ", 5/2)

# Usage of the exponentation operator
x = 10**2
print("10 raised to 2 = ", x, "Type : ", type(x))

x = 10**-2
print("10 raised to -2 = ", x, "Type : ",type(x))
```

```
21.0 <class 'float'>
5//2 =  2
-5//2 =  -3
-5.0//2 =  -3.0
-5/2 =  2.5
10 raised to 2 =  100 Type :  <class 'int'>
10 raised to -2 =  0.01 Type :  <class 'float'>
```

Try it yourself

The operators listed above are not restricted to numeric types. In a later module, we will learn how to define their usage for any pair of user-defined objects. In the meantime, though, what do you expect the output of these expressions to be?

```
print("Hello"*3)
print("Hello" + "World")
```

Try them out in a Python console.

4.2 Boolean expressions

Python also provides a boolean data type that can hold one of two values `True` or `False`. We will soon see how to construct comparisons between expressions. The result of such a comparison is always a boolean value. It is also possible to convert any object to a boolean value using the built-in function `bool()`. Below are some examples of the latter.

Expression	Result	Comment
`bool(0)`	`False`	The integer 0 converts to value `False`
`bool(100)`	`True`	All integers other than 0 convert to `True`
`bool(0.0)`	`False`	Zero of any numeric type (integer, float, complex numbers) converts to `False`
`bool("")`	`False`	Empty sequences (lists, strings, sets, tuples and dictionaries) convert to `False`.
`bool("False")`	`True`	Non-empty string converts to `True`
`bool([1, 3, 5])`	`True`	Non-empty list converts to `True`
`bool([0])`	`True`	Even if the list contains just 0!

The operators that are used for making comparisons between expressions are $<$ (less than), $>$ (greater than), $<=$ (less than or equal to) $=$ (greater than or equal to), $==$ (equal to), $!=$ (not equal to). Here are a few points to remember when constructing expression that compare two variables.

- Numbers of built-in numeric types (int, float, complex) generally can be compared within and across the types. A special exception is `Nan` (not a number).
- Sequences (lists, tuples or range) can sometimes be compared with other objects of their own type. For comparisons to be possible, both sequences should be of same length and the data type of every pair of objects obtained from the sequences should admit (allow) comparison.

```
print(2 == 2.0)
print([2,3] == [2.0,3.0])
print("Hello"=="hello")
print(7 <= 7.0)
print(7 < 7.0)
```

```
True
True
False
True
False
```

5 Decision making : `if ... elif ... else`

The `if` statement is used to conditionally execute a set of statements. In other words, within an `if`, the set of statements that are executed, depends on whether a specified

condition has been satisfied. The syntax of this control structure in Python begins with the `if` keyword followed by an expression that evaluates to either `True` or `False`. A colon must be included after the expression. Here is an example:

```
1 age = int(input("Enter your age :"))
2 if age>=18:
3     print('Yay ! You are eligible to vote')
4     print('You are also eligible for jury duty')
```

The indented block of statements (lines 3 and 4) below the `if` statement are executed if the condition `age>=18` evaluates to `True`. Optionally an `else` clause can be included along with its own indented block of statements which are evaluated when the condition next to the `if` statement evaluates to `False`. **The Python interpreter uses indentation levels to identify when blocks of statements begin and terminate.** To indent a statement in Spyder, use the Tab key which adds four spaces by default.

It is also possible to modify the flow of the program based on more than one criteria using the `elif` ("else if") clause. Here is an example.

```
1 number = int(input("Enter a number :"))
2 if (number>0):
3     print("Number is positive")
4 elif (number==0):
5     print("Number is zero")
6 else:
7     print("Number is negative")
8
9 print("Execution complete")
```

```
Enter a number :0
Number is zero
Execution complete
```

In the above example, suppose the user enters 0. The first condition after the `if` keyword is not met. The execution moves on to check the condition of the `elif` clause. Since the condition is satisfied, line number 5 is executed, i.e. "Number is zero" is displayed on the console. The execution then exits the conditional block and moves directly to line 9.

In writing `if .... elif ... else` blocks, the programmer should be careful to construct conditions such that no more than one can be true at any time. Additionally, the keywords `else` and `elif` should be at the same indentation level as the `if` statement.

5.1 Logical Connectives

Here is a list of Python operators/keywords which are useful to write compound Boolean expressions.

5.1.1 **and** operator

```
1 number = int(input("Enter a number :"))
2 if number%2 == 0 and number%3 == 0:
3     print("The entered number is divisible by 2 and 3")
4     print("The entered number must be divisible by 6 as well")
```

```
Enter a number:18
The entered number is divisible by 2 and 3
The entered number must be divisible by 6 as well
```

The `and` keyword can be used to combine two boolean expressions. As the name indicates it evaluates to `True` only when both the expressions evaluate to `True`. In the example above, if the value stored in variable `number` is divisible by both 2 and 3, then lines 3 and 4 are executed.

5.1.2 **or** operator

When the `or` keyword is used to combine two boolean expressions, the result is True if either one (or both) of the subexpressions is `True`.

```
1 points_scored = int(input("Enter the number of points scored"))
2 assists = int(input("Enter the number of assists"))
3
4 if points_scored>20 or assists>8:
5     print("Great game!")
```

5.1.3 **not** operator

The `not` keyword returns `True` if the expression next to it evaluates to `False` and vice versa. Though we have not formally introduced the `in` operator used in the following example, its meaning should be clear from context:

```
1 name = input("Enter your name  :"))
2 if not 'a' in name:
3     print("Your name does not have the alphabet letter a.")
```

Compound expressions that involve `and` or `or` are lazily evaluated, i.e evaluated only if necessary. For example, in an expression that uses the `and` keyword, the second expression is evaluated *only* when the first expression is `True`. Likewise when the `or` operator is used, the second expression is evaluated *only* when the first expression is `False`.

5.2 Conditional Expressions

Python allows the programmer to write one-line conditional expressions using the `if` keyword. Here is the syntax of a such an expression.

```
answer = expression_1 if boolean_expression else expression_2
```

When a statement of this form is executed, first `boolean_expression` is evaluated. If it is `True`, then `expression_1` is evaluated and the result is stored in variable `answer`. Otherwise, `answer` stores the result of `expression_2`. Here is an example:

```
number = 20
odd_or_even = "Even" if number%2==0 else "Odd"
print(odd_or_even)
```

```
Even
```

Try it yourself

Write a *one-line* conditional expression that prints "Positive" if a variable x is greater than zero, "Negative" if it is less than zero and "Zero" otherwise.

5.3 `in` operator

When used in an `if` statement the `in` keyword performs membership testing or it tests if a value or an item is present in a collection. Example : (i) test if a character is present in a string (ii) test if an item is present in a list.

```
list_of_my_cars = ["Civic", "Accent", "Camry", "Corolla"]

if "Civic" in list_of_my_cars:
    print("I own a Civic")
```

5.4 `is` operator

The `is` keyword is used to test if two variables refer to the same object stored in memory. It provides information similar to that learned when using the built-in `id()` function to compare the address of the two variables.

```
shopping_list = ["yogurt", "banana", "bread", "eggs"]
shopping_cart = shopping_list

if shopping_cart is shopping_list:
    print("All items in the list are now in the cart")

```

Try it yourself

Will the code below execute without an error every time? What mistakes can you identify in this piece of code?

```
age                 = int(input("Enter your age: "))
driving_experience = int(input("How many years have \
  you held a driver's license: "))

if age > 25 and driving_experience > 5:
    print("You are an experienced driver")
    print("We can offer you our premium insurance plan")
elif age < 25 or drvng_experience < 5:
    print("We can offer you our young driver plan")
else:
    print("We do not have an insurance plan to offer you")

```

5.5 Nesting

It is possible to define an `if` statement within another `if` statement. This is called nesting. Python uses indentation levels to distinguish statements associated with each level of nesting. Here is a simple example

```
ones_string = ["zero","one","two","three","four", \
               "five","six","seven","eight","nine"]
tens_string = ["ten","twenty","thirty","forty","fifty" \
               ,"sixty","seventy","eighty","ninety"]

number = int(input("Enter a number : "))

if number>=0 and number<100:
    if (number<10):
        print("You entered "+ones_string[number])
    else:
        print("You entered "+tens_string[number//10 - 1] + \
```

```
13                 " " + ones_string[number%10])
14 else:
15     print("Enter a number between 0 to 99")
```

Clearly the code above does not require a nested `if` statement. It is often better to avoid nested `if` statements for better code readability and performance, if possible. Here is a simple fix for the above example.

```
1  ones_string = ["zero","one","two","three","four", \
2                 "five","six","seven","eight","nine"]
3  tens_string = ["ten","twenty","thirty","forty","fifty" \
4                 ,"sixty","seventy","eighty","ninety"]
5
6  number = int(input("Enter a number : "))
7
8  if number>=0 and number<10:
9      print("You entered "+ones_string[number])
10 elif number>=10 and number<100:
11     print("You entered "+tens_string[number//10 - 1] + \
12           " " + ones_string[number%10])
13 else:
14     print("Enter a number between 0 to 99")
```

6 Loops

Loops are used to used to repeat a set of statements. There are two kinds of loops in Python: `while` and `for`. The `while` statement is typically used when a set of actions has to be repeated until a *condition* is satisfied.

6.1 `while` loop

Here is an example of a `while` loop that calculates and prints the sum of the squares of the first three numbers.

```
1 i = 1
2 sum_square = 0
3 while (i<=3):
4     sum_square += (i*i)
5     i += 1
6
7 print("Sum of squares is :", sum_square)
```

```
Sum of squares is : 14
```

The above loop performs 3 iterations, i.e the indented block (lines 4 and 5) is repeated 3 times. The variables `sum_square` and `i` are incremented in each iteration until the value of the variable `i` is greater than 3. Once the condition `i<=3` is no longer satisfied, i.e evaluates to `False` the loop terminates. The `print` statement (line 7) is outside the loop as its indentation level is same as the `while` statement.

6.1.1 `break` statement

It is possible to control the flow of execution within a loop using `break` and `continue` statements. When the `break` statement is executed, the rest of the statements within the loop (indented block) are ignored, the loop is terminated, and the execution moves to the statement after the loop. Here is a small piece of code that prints the line "You are awesome!" until the user decides to quit

```
1 while True:
2
3     print("You are awesome!")
4     choice = input("Shall I repeat it y/n ?")
5
6     if (choice=="n"):
7         break
8
9 print("Goodbye")
```

Try it yourself:

Will the piece of code shown below run in a Python interpreter? If yes, why and what is the expected output? If no, why not?

```
1 while 1:
2     print("When will this end?")
```

6.1.2 `continue` statement

A `continue` statement is used when you need to conditionally skip iterations. Here is an example which computes the sum of squares and cubes of numbers, but only if the number is odd.

```
1 while True:
2
```

```
3       number = int(input("Enter a number: "))
4
5       if number % 2 == 0:
6           print("Please enter an odd number")
7           continue
8
9       print("Square of the number is ", number**2)
10      print("Cube of the number is ", number**3)
11
12      choice = input("Shall I collect another (y/n)? ")
13
14      if (choice=="n"):
15          break
16
17  print("Goodbye")
```

In the above example, if the number entered is even (divisible by 2), a warning to enter an odd number is printed on the screen. Then the continue statement is executed (line 7) which transfers the execution back to line number 1 (`while` statement), ignoring the rest of the statements in the loop. The next iteration of the loop then commences.

6.1.3 `else` statement

Often in loops with a `break` statement, the programmer may need to know if a loop ended (i) due to a `break` statement or (ii) due to the condition next to `while` not being met. For such situations the `else` clause of the `while` loops proves useful. Here is an example.

```
1   final_positions = ["Vettel", "Verstappen", "Bottas","Ricciardo", \
2                      "Albon","Hamilton","Perez","Raikkonen"]
3   driver_name = input("Enter driver name: ")
4
5   i=0
6   while (i<3):
7       if final_positions[i] == driver_name:
8           print(driver_name + " finished at position ", i+1)
9           break
10      i+= 1
11  else:
12      print("Search complete. Driver did not finish in top 3")
```

If the loop terminates via the `break` statement the block within the `else` clause is ignored. If the loop terminates due to the condition next to `while` not being met then the flow of execution moves to the `else` block. Note that the `else` clause can be used even in situations when a `break` is not present. In such cases, once the loop terminates the statements in the `else` block get executed.

6.2 `for` loop

The `for` loop in Python is more versatile than what C or MATLAB offers. It repeats a set of statements for every item in something called an *iterable*. Let's first learn what an iterable is.

6.2.1 Iterables

An iterable is an object that has a collection of members which can be processed one at a time. In other words an iterable is something you iterate over. Lists and strings which we have learned in this module are examples of iterables. Later in this course you will learn about other iterables like sets, tuples, dictionaries and numpy arrays.

The iterable can also be just a list of numbers. The built-in `range()` function allows you to iterate over a series of integers. It is typically used in a `for` loop when you know the number of times a block of statements has to be repeated. The `range` function can take three parameters (start, stop, step), e.g. `range(1, 10, 2)` returns every second number from 1 to 9. The start and step parameters are optional. By default, the start is 0 and step is 1.

Let us look at a simple example that uses a for loop. The code block below simply prints the string *Hello* three times. Since we know exactly how many times to repeat the statement, we make use of the `range` function.

```
1 for i in range(3):
2     print(i, "Hello")
```

```
0 Hello
1 Hello
2 Hello
```

The variable `i` takes on the value 0, 1 and 2 during subsequent iterations. In the last section we learned that the `in` keyword is used for membership testing. When used in a `for` loop its role is to feed one item from the iterable to the variable on its left. Note that the statement to be repeated (line number 2) is indented.

Now let's examine another simple example in which the iterable is a list of strings.

```
1 leaderboard = ["Vettel", "Verstappen","Hamilton","Raikkonen"]
2
3 for driver in leaderboard:
4     print(driver)
```

```
Vettel
Verstappen
```

Hamilton
Raikkonen

6.2.2 enumerate

enumerate is a built-in function that is useful for constructing loops. It takes an iterable as input and returns the next item in the iterable and its index position. Since it returns two values during each iteration we will need two variables next to for to store them. By default the index position starts with zero, but an optional start parameter can be included to modify it. Here is a simple example that demonstrates the use of enumerate.

```
1
2 leaderboard = ["Vettel", "Verstappen","Hamilton","Raikkonen"]
3
4 for position, driver in enumerate(leaderboard, start=1):
5     print(position, driver)
```

```
1 Vettel
2 Verstappen
3 Hamilton
4 Raikkonen
```

Note that the range function is lazily evaluated. It just evaluates one number at a time which is fed to the loop. Consequently, its memory footprint is small. The statement range(1000000000) does not allocate memory for a billion numbers in memory, it simply requires enough memory to store one number. Here is a simple demonstration that describes this. We will need the getsizeof() function which is part of the sys module. The sys module comes packaged with the Python standard library. To use the module we need the import sys statement in our script.

```
1  import sys
2
3  # Define a range of numbers
4  x_as_range = range(100000)
5  # Convert the range into a list
6  # Now memory has to be allocated for all numbers
7  x_as_list = list(x_as_range)
8
9  print("Size in bytes : range  ",sys.getsizeof(x_as_range))
10 print("Size in bytes : list(range) ", sys.getsizeof(x_as_list))
11
```

```
Size in bytes : range   48
Size in bytes : list(range)  900112
```

6.2.3 Keywords `else`, `break` and `continue`

`for` loops can also have an optional `else` clause. The statements in the `else` block are executed once all the iterations of the loop are complete. You can use the `break` and `continue` statements in a `for` loop in the same manner as you can in a `while` loop to control the flow of execution. Here is an example that shows the usage of these statements. The code checks if a vowel is present in a string input.

```
1
2  vowels = ['a','e','i','o','u']
3  name   = input("What is your name? \n")
4
5  for letter in name:
6      if (letter in vowels):
7          print("Your name has a vowel")
8          break;
9  else:
10     print("Your name has no vowel in it")
11
```

And just as with `if` statements, loops can also be nested. There is no limitation on number of nesting levels that you can employ. However for the sake of readability and performance, it is best to avoid nesting loops more than 2 or 3 levels deep, and come up with a different algorithm to solve your problem.

Try it yourself:

Write a script to investigate the Collatz conjecture

1. Pick any positive number x.
2. If x is odd replace it with 3x +1
3. If x is even replace it with x/2
4. Repeat steps 2 and 3

In mathematics, the Collatz conjecture states if the steps above are repeated enough times, x eventually reaches the value 1. Write a script to collect an input value x and then output how many steps are required for that x to converge to 1.

6.2.4 List comprehensions

List comprehensions are terse statements that are used to create lists. They use a one-line `for` loop. The examples below demonstrate their usage. Note that a one line `if` statement can also be employed in a list comprehension to conditionally filter out the items of a list.

```
# List of the squares of all numbers from 1 to 10
list_squares = [x*x for x in range(1,11)]
print("List of squares : ", list_squares)

# List of the squares of all numbers from 1 to 10
# that are divisible by 2
list_squares = [x*x for x in range(1,11) if (x%2)==0]
print("List of squares of even numbers only : ", list_squares)

# List that has squares of all numbers from 1 to 10
# and zeros if a number is odd
list_squares = [x*x if (x%2)==0 else 0 for x in range(1,11) ]
print("List of squares of even numbers : ", list_squares)

```

```
List of squares :  [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
List of squares of even numbers only :  [4, 16, 36, 64, 100]
List of squares of even numbers :  [0, 4, 0, 16, 0, 36, 0, 64, 0, 100]
```

Here's how to interpret the first list comprehension above: for every number from 1 to 10 add an entry to the list that is equal to the square of the number.

The second example above can be read as: for every number from 1 to 10 add an entry to the list that is equal to the square of the number only if the number is divisible by 2.

And the third one; for every number x from 1 to 10, add an entry to the list with value that equals the square of the number if x is divisible by two and 0 otherwise. Note that when the `else` clause is used in a list comprehension the order of the statement changes; `for` moves to the end of the statement.

Try it yourself:

In one line of code, find the sum of the square of the integers 1 to 1000.