

PYTHON PROGRAMMING

Absolute Beginners Tutorial

JON JAMES BOND

Python Programming

Absolute Beginners Tutorial

Contents

[0. Introduction](#)

[0.0 About this Tutorial](#)

[0.1 What is coding?](#)

[0.2 How do we start?](#)

[0.3 Supported Reading!](#)

[1. Variables](#)

[1.1 Arithmetic](#)

[1.2 Basic Arithmetic Operators](#)

[1.3 Arithmetic Operator Precedence](#)

[1.4 More about Floating Point Numbers](#)

[2. Text](#)

[2.1 Mixing Strings with Numbers](#)

[2.2 Single Characters](#)

[2.3 Input](#)

[2.4 Length](#)

[2.5 Contents of Strings?](#)

[3. Logic & Decision Making](#)

[3.1 Equals](#)

[3.2 Less Than](#)

[3.3 Less Than or Equal To](#)

[3.4 Greater Than](#)

[3.5 Greater Than or Equal To](#)

[3.6 Not](#)

[3.7 Boolean Theory](#)

[3.8 If Statements](#)

[3.8.1 If-Else statement](#)

[3.8.2 Else-If statement](#)

[3.8.3 Is](#)

[3.9 Combining Logical Tests \(AND & OR\)](#)

[3.9.1 Normalizing inputs as Booleans](#)

[3.10 Not](#)

[3.11 Empty values with None](#)

[3.12 Not Equals](#)

[4. Functions](#)

- [4.1 Defining Functions](#)
- [4.2 Returning Results from Functions](#)
- [4.3 Naming functions](#)
- [4.4 Passing Parameters to Functions](#)
- [4.5 Nesting Functions](#)
- [4.6 Global Variables](#)
- [4.7 Pass](#)

[5.0 Loops](#)

- [5.1 While Loop](#)
- [5.2 Break Statement](#)
- [5.3 For Loops](#)
- [5.4 Ranges](#)
- [5.5 Continue](#)

[6.0 Lists](#)

- [6.1 Empty Lists, Append & Mixed Item Types](#)
- [6.2 Updating Values](#)
- [6.3 Inserting to Lists](#)
- [6.4 Adding Lists Together \(extend\)](#)
- [6.5 Removing from Lists](#)
- [6.5.1 Pop](#)
- [6.6 Sorting Lists](#)
- [6.6.1 Creating Sorted Lists](#)
- [6.6.2 Reversing Lists](#)
- [6.7 Counting Occurrences](#)
- [6.8 List Parsing](#)
- [6.9 Lists within Lists](#)
- [6.10 Advanced Sorting](#)
- [6.11 Tuples](#)
- [6.12 Sets & Dictionaries](#)

[7.0 Classes](#)

- [7.1 Naming Conventions](#)
- [7.2 Our First Classes](#)
- [7.3 Structure & Best Practices with Classes](#)
- [7.4 Multiple Members, Parameters & Functions](#)
- [7.5 Classes Containing Lists](#)

[8.0 Importing & Libraries](#)

- [8.1 Importing](#)
- [8.2 Import Errors](#)
- [8.3 Useful Library Examples](#)
- [8.3.1 Queues](#)

[8.3.2 Date Time](#)

[8.3.3 Platform](#)

[9.0 Handling Errors](#)

[9.1 Specific Errors](#)

[9.2 Finally](#)

[9.3 Raising an Exception](#)

[9.4 Import Errors](#)

[9.5 System Exit & Quit](#)

[10.0 Basic File Handling](#)

[10.1 Reading Text Files](#)

[10.2 With & Files](#)

[10.3 For-Else](#)

[10.4 Handling End of File](#)

[10.5 Writing Text Files](#)

[10.6 Binary](#)

[10.7 Read, Write and Seek](#)

[Appendices](#)

[Installing & Using Python](#)

[Using Python](#)

Introduction

0.0 About this Tutorial

This text is intended to present a friendly voice as you take your first steps into the ever expanding and technical world of computer programming. Step by step, no nonsense, example lead learning which aims to explain all the jargon and make you feel empowered with your new skills.

Whether you have never attempted to learn a programming language before, or always found other tutorials too intense, this text is intended to help you overcome your fears and familiarise you with the most basic concepts about general programming with the Python Programming language.

Example code is given in highlighted blocks, items of special interest are emboldened, though no code example is more than a few lines to keep your focus on the idea rather than the worry of what you are learning.

Any screen shots are captures from real machines.

0.1 What is coding?

Code is essentially a series of lines of text, that the computer can follow to perform tasks for you, control of the order is from top to bottom as you write the lines of code, but it can make the order of actions change with commands within the code itself, to decide to do one thing or another, perform loops or even just quit!

Coding, programming and scripting are all terms for the activity of writing these lines of code. Hacking is similarly one of these terms, however we will only think about hacking in the true, original, sense of the term of inquisitive exploration of technology and our code; we are not considering destructive intrusions.

0.2 How do we start?

If you are not sure how to install or use Python you will benefit from reading the two Appendices before continuing into the main body of the book.

Once you are happy and have Python installed, you will see the chapters are presented with code examples which can be typed directly into the interpreter.

As we progress, definitely by chapter 8, we will be writing code directly into separate files and running those files through Python, real programming painlessly introduced to you.

0.3 Supported Reading!

For support, articles and pointers with your learning you can find my personal blog is available online: <http://megalomaniacbore.blogspot.co.uk>

You can also find my YouTube channel full of information & learning examples: www.youtube.com/user/LordXelous

Chapter 1: Variables

The simplest thing we can do with a Python program is place a numerical value into a named location for use elsewhere. To place a number into some value we simply write a name for the item and state that it equals the numeric value, thus:

```
somewhere = 10
```

To show ourselves the value stored inside the name "somewhere", we can send this name to the print function, like this:

```
print(somewhere)
```

This location we've used is called "somewhere", it could have been called anything we wanted so long as the name we picked isn't in use by some part of Python (known as a reserved word), and it is known as a variable because we can change the value after setting up the initial value, like this:

```
somewhere = 42
```

Go ahead and print the variable again and you will find it contains the new value. We've seen that the '=' symbol assigns values to a named variable and that the 'print' function will echo the value to the screen as plain text and also that '()' brackets are used to enclose variables you pass to a function!

1.1 Arithmetic

When we have assigned a value to a variable, we might want to change the value later, we've seen this when assigning with the '=' symbol. This is called an operator. There are several other operators, the most important are known as the arithmetic operators, because they allow us to carry out simple mathematics. For example:

```
value = 1 + 2  
print(value)
```

We should see the value "3" printed out! Clearly this is the addition operator.

Performing subtraction is very similar, we just need change to use the '-' symbol...

```
value = 10 - 3  
print(value)
```


Negative numbers are handled for us also, so if we perform:

```
value = 3 - 10
```

We can now see the value is minus 7, and displayed as -7.

The number is lead by the minus symbol, which is the same as the subtract operator symbol, only the spacing tells Python what we intend to do.

So to assign a value of minus five, we do this:

```
value = -5
```

Subtracting from a negative number will make it more negative, as will adding one negative number to another number.

You will notice how important the spaces are here to tell python when you intend to subtract or when you intend to indicate a negative number with the '-' symbol, common to both actions.

Try this code out, make sure you are comfortable with it before you continue:

```
value = -5  
print (value)  
  
value = value - 5  
print (value)  
  
value = value + -7  
print (value)
```

The next arithmetic operator to look at is multiplication....

```
another = 12 * 2  
print (another)
```

We get the result 24! Spend sometime writing out multiplications, you can multiply negative numbers too...

Division is the next arithmetic operator, it is represented by yet another symbol, the '/' or forward slash and we use it in just the same way as all the previous operators...

```
something = 10 / 2  
print (something)
```

We now see the result of '5'. At this point we need to introduce something a little more complex, this is when we have part of a whole number left after a division... Lets consider the following code:

```
value = 1 / 2
```

The result is part of a whole number, known as a fraction, so you might expect the result of a half or $\frac{1}{2}$, however computers don't work with fractions they work with numbers called "floating point". A half is represented as 0.5 and this is the result we see from our code!

Floating point, its problems and limitations, are not strictly within the scope of this tutorial; however, we'd have to recommend you spend sometime exploring these code examples and trying out some divisions now.

You can also directly assign new floating point numbers to variable names, like so...

```
value = 0.25  
print (value)
```

From this we can then learn that floating point numbers can be made to add, or in this case multiply back, into whole numbers....

```
value = 0.25  
value = value * 4  
print (value)
```

However, you will notice the result, though we know it is '1' is still shown with a floating point indication, making it '1.0'. You can cast this whole number back to an integer with "int"...

```
value = 0.25  
value = value * 4  
print (value)  
whole = int(value)  
print (whole)
```

If the floating point had a non-whole remainder, it would be lost by this code...

```
value = 2.75  
print (value)  
whole = int(value)  
print (whole)
```

The result we should see is two, the 0.75 being lost in the conversion. We will cover using

"int" further in chapter 2.

1.2 Basic Arithmetic Operators

The four operators we have covered so far are the most basic mathematical operations we can carry out, it is vital you understand how they work, lets recap...

Arithmetic Operators

Name	Symbol	Example
Addition	+	value = 1 + 2
Subtraction	-	value = 2 - 1
Multiplication	*	value = 2 * 4
Division	/	value = 4 / 2

1.3 Arithmetic Operator Precedence

The mathematicians amongst us will already have been wondering about the precedence of the operations, that is the order in which python handles the math functions as it processes a line of code, when that line contains more than one operation...

```
value = 10 + 2 * 4
```

You might expect value to print out as 48, that is the code looks like it should "add 10 and 2 to give twelve then multiply by four". However, what we actually get the result 18, how did this happen?

Because the code performed the multiply first! It acted on the "two times four" then "added ten", giving eighteen.

This is an example of the order of precedence within the arithmetic operators, simply put multiplication outranks addition and so it performed first. Therefore, we need to know the order here of these precedence, so lets take a look at the table from 1.2 but this time we'll rearrange the rows into their order of precedence... Multiplication is performed first and so on...

Arithmetic Operators

Name	Symbol	Example
Multiplication	*	value = 2 * 4
Division	/	value = 4 / 2
Addition	+	value = 1 + 2
Subtraction	-	value = 2 - 1

Try out some code, see what values you expect and what you actually get...

```
value = 10 - 2 * 3
print (value)

value = 90 + 10 * 2 / 2
print (value)

value = 2 * 4 - 2
print (value)
```

If you break your code into individual mathematic operations, one per line, you are free to ignore these rules, one instruction per line simply performs the operations in the order you lay them out, from the last example above, this would be...

```
value = 2 * 4
value = value - 2
print (value)
```

However, there is a way to override the default order of precedence and perform your maths in the intended order all on one line, this is done by enclosing parts of the various calculations in the round brackets '()', also called parenthesis.

```
value = (10 - 2) * 3
print (value)
```

Python will now inspect the line of code, spot the bracketed operation and perform it first.

When we have very complex calculations, we can put one set of parenthesis inside another, further controlling how the calculation is performed. Lets place sets of brackets inside one another...

```
value = ( (10 + 2) / 2 ) * 4
```

Now we see the addition is done first, then the division before finally the multiplication! Take some time to get used to this ordering and using the brackets in this manner.

1.4 More about Floating Point Numbers

The numbers we used at the very start of this chapter, which were whole numbers like "1" and "7" were whole because they had no fraction or floating part to them, these are also known as "integers".

Floating point, or fraction, or decimal, numbers are those which are can have a whole part

and then also contain a part of a whole, or fraction... We have touched upon this, so won't labour the point, instead lets jump to some code...

```
floatingNumber = 3.14  
print (floatingNumber)
```

We use the full-stop, or period, symbol to indicate the number is a floating point number. We can use lots more than just two points of accuracy. The python interpreter actually stores your number with 53 bits of accuracy, this means the binary 0 and 1's in the computer memory used to represent your floating point number has 53 bits with which represent the value, this is an approximation of the actual value.

If the floating point number is attempting to be too precise it will loose accuracy as there are simply not enough binary bits to store the true value, this problem can make subtle problems arise in your code, lets take a look at a simple example...

```
sum = 0.1  
sum = sum + 0.1  
sum = sum + 0.1  
print (sum)
```

What would we expect the printed result to be?... 0.3, surely....

No, we see "0.30000000000000004".... This is a very odd value, this additional "00000000000000004" is very simply caused by the accuracy of the floating point number not being good enough!

Just as with the integers earlier, we can have negative floating point numbers and we can perform operations which mix integer numbers with floating point. Though once we involve a floating point number in an operation the result will always be a floating point number.

```
value = (4 / 2) + 3.1  
print (value)
```

"value" is now always a floating point number, unless we use "int" to change it back into a whole again, though we risk cutting off any floating value from the end!

We can also cast values to a float, with the "float" keyword...

```
a = 3  
print ("This is an Integer " + str(a) )  
  
b = float(a)  
print ("This is a float " + str(b) )
```

You can see we still print our floats out after using "str" to convert them into strings. There are more uses of the "float" keyword in the next chapter.

Chapter 2: Text

In chapter 1, only worked with numbers, this is a large part of what computers do for us. However, when we come to present our results we need to print more than just numbers to the screen... We need text....

For labels, titles, for names and places, to give meaning to the raw numbers inside the computer.

This is what we will learn about throughout this chapter, lets start with just assigning a string of characters to a variable, in the same way we did a number... It a string of characters is conventionally just called a "string"...

```
name = "Xelous"  
print (name)
```

The only difference is the enclosing of the string with quotes, to tell Python where the string begins and ends. In Python you can use the double quote, or the single quote for this purpose....

```
name = 'Xelous'  
print (name)
```

Just as we performed with numbers, in Python, we can add strings together...

```
passage = "To be" + " or not to be"  
print (passage)
```

The swapping of double and single quotes lets us insert a mix of their use within the output...

```
passage = "To be" + ' or not ' + "to be"  
print (passage)
```

So we can put a quote into the middle of our output... Like this...

```
print ("My Name is 'Xelous' Oh yes it is")
```

The outer string itself has double quotes, the inner string as a single quote, but this gets output as one message on screen with the name "Xelous" enclosed in single quotes when output.

So long as you end the string of characters with the same quote mark type as you started it Python will know what you intended.

One reason for this interchangeability of strings in Python is to ease your learning of how to emplace quotes into the actual string of text you want to display, let's consider this:

```
quote = "Xelous said "Hello""
```

Running this code things go very wrong, and we see Python report "SyntaxError". The reason being it can't understand what you intended here, the solution? Simply start and stop your string with one set of quotes, and use the other style to represent internal quotes, like this:

```
quote = "Xelous said 'Hello'"
print(quote)
```

This works perfectly now, however, it's not quite as we intend, in a book or other text we'd expect double quotes around Hello here, so we need to swap the quotes over to suit our needs:

```
quote = 'Xelous said "Hello"'
print(quote)
```

2.1 Mixing Strings with Numbers

When we come to write numerical and textual data within a single line, we need to beware of how we present the different underlying data to Python, your numbers are of the "integer" or "floating point" types we have already spoken about whilst the text is in the "string" data form. You can not mix the two.

However, you can convert numbers into text... so the number 2 can become the string "2".

This allows us to convert the numbers into text on the fly, and print out the result we expect, let's take a look at how this problem arises, then solve it...

```
age = 12
print("I am " + age + " years old")
```

We see an error here, that we can not print the number against a string. The solution is to use the "str" function to turn the number variable into a string like so:

```
age = 12
print("I am " + str(age) + " years old")
```

We can store the string conversion of a number just like any other text string...like this:

```
age = 12
ageString = str(age)
```



```
print ("I am " + ageString + " years old")
```

We can use this variable called "ageString", but we must update its content to keep it relevant... If we just changed the "age" number we would not see "ageString" itself change. Because there is no link between the two...

```
age = 12
ageString = str(age)
print ("I am " + ageString + " years old")
age = 13
print ("I am now " + ageString + " years old")
```

To update the "ageString" we need to make sure we convert the new value in the integer into a string to place the new value as a string before the re-printing.

```
age = 12
ageString = str(age)
print ("I am " + ageString + " years old")
age = 13
ageString = str(age)
print ("I am now " + ageString + " years old")
```

We can reverse the process of turning a number into a string, which is very useful when we have say read a number from a text file or gotten it from the user via the keyboard. Except instead of using "str" we use the "int" or "float" functions, like this:

```
textValue = "12345"
value = int(textValue)
print (value)
value = value + 43210
print (value)

floatNumber = float("123.456")
print (floatNumber)
```

A float number conversion needs to contain a single dot (or period) symbol to denote it's type.

Sometimes programs opt to store all their numbers as text, for ease of writing to files or serving over the internet as text, and only convert the string representations back into a usable numerical value when arithmetic is to be carried out upon them, like this:

```
balance = "400"
balance = int(balance)
```

```
balance = balance + 10
balance = str(balance)
print (balance)
```

2.2 Single Characters

Many programming languages make the distinction between a single character and a string of characters, making them two different types. Python does away with this convention and you can represent a character just as you have a whole string:

```
char1 = "a"
print (char1)
char2 = 'b'
print (char2)
added = char1 + char2
print ("Both Chars are: " + added)
```

2.3 Input

So far we have learned how to output with "print", how about asking the user a question? Well we have to use a function called "input", we want to call this function without any parameters and assign the value the user gives to a variable, like this:

```
print ("What is your name?")
name = input()
```

Note: if you are using Python v2 please use "raw_input" wherever we mention "input" in the examples, this is a difference between the two versions of Python, and nothing to worry about.

Python will output the question, and then wait for the user to press at least the 'Enter' key, so they can enter a blank reply if they so wish and we can output their reply:

```
print ("What is your name?")
name = input()
print ("Hello " + name + " this is a Python program!")
```

All of the variables we ask the user for with input arrive as text, so if we ask the user for a number we must remember to cast or convert it if we want to perform any calculations with them:

```
print ("How old are you?")>
age = input()
age = int(age)
nextAge = age + 1
```

```
print ("And next year you will be " + str(nextAge))
```

You can of course also use "float" on this age, perhaps if you are 13.75 years of age?

```
print ("How old are you?">  
age = input()  
print (age)
```

We should see the code execute, the user is prompted to enter their age... If we entered "13.75" the variable "age" will show back to us as "13.75", however it is not a floating point number, it is a piece of text. You can check this by trying to perform arithmetic with it...

```
age = age + 1.0
```

This code throws up an error "TypeError: Can't convert 'float' object to str implicitly", this means the value was actually text and we would not add the "1.0" to it. So as we did with integers we need to use a keyword to convert this text into a floating point number we can use, that keyword is "float".

```
print ("How old are you?">  
age = input()  
floatAge = float(age)  
floatAge = floatAge + 1.0
```

With the value turned into a number, we can then successfully perform arithmetic, it is a number not text!

2.4 Length

One other very common function to use with a string is to find it's length, you do this with the "len" function. "len" maybe used with more than just strings, however we'll cover those cases later.

```
print ("Enter a string of text!")  
enteredString = input()  
  
print ("You entered " + len(enteredString) + " characters")
```

2.5 Contents of Strings?

An extremely useful function for a string is the "contains" function... Which is one of a set of functions which include "startswith" and "endswith", that allow you to check the contents of a string... Lets take a look at some examples:

```
text = "To be or not to be"
check1 = text.startswith("To")
check2 = text.startswith("to")
print (check1)
print (check2)
```

We should see the output, on two lines of "True" then "False". These are boolean values, logic, something we will cover in Chapter 3.

Likewise "endswith"...

```
text = "To be or not to be"
check1 = text.endswith("To")
check2 = text.endswith("be")
print (check1)
print (check2)
```

Again we get two "True" or "False" results.

These boolean results are a special type, and to print them with a string you also need to use "str" on them...

```
text = "To be or not to be"
check1 = text.startswith("To")
check2 = text.endswith("be")
print ("Did it start with 'To'? " + str(check1))
print ("Did it end with 'be'? " + str(check2))
```

These two functions are contained within the strings we are working on, we see this as they are called with the variable name then "." and the function name.

"length", "startswith" and "endswith" are all of this type, they are functions within the string, however there are other functions which help us determine the contents of a string but which are not called in the same way. A good example of this is the functional operator "in". Which is called in the same way as the arithmetic operators were in Chapter 1.

```
text = "Mary had a little lamb!"
check = "little" in text
print (check)
```

Again we get a result of either "True" or "False", however now we looked through all the text string, literally to see if the first string provided was within!

In the next chapter we will look at using these boolean values to perform logical checks on

values and perform actions.

Chapter 3: Logic & Decision Making

This is one of the more complex chapters in this text, however, it is an extremely important topic which we will need to master and feel comfortable with. If you are already comfortable with the logical operators, you can jump to section 3.8 to see how they are applied within Python, however the first seven sections of this chapter serve as a common introduction to logical programming and decision making in general terms.

When we write a program, we need to think about making decisions within our code, controlling what our program does, when it does it and to decide between various options. These points are collectively referred to as flow control, to literally control the flow instructions through the program as it progresses.

The main tools at our disposal to do this are statements which compare one value with another performing an action or not. There are also statements such as loops, to repeat or coordinate processing over multiple different items within a data set.

These are called the Logical Operators, and are very similar to the arithmetic operators, but exclusively result in "True" or "False"... They look like this...

Logical Operators

Name	Symbol	Example	Value
Equals	==	value = 1 == 2	False
Equals	==	value = 2 == 2	True
Less Than	<	value = 2 < 1	False
Less Than	<	value = 1 < 2	True
Less Than	<	value = 1 < 1	False
Less Than or Equals to	<=	value = 2 <= 1	False
Less Than or Equals to	<	value = 1 <= 2	True
Less Than or Equals to	<	value = 1 <= 1	True
Greater Than	>	value = 2 > 1	True
Greater Than	>	value = 1 > 2	False
Greater Than	<	value = 1 > 1	False
Greater Than or Equal to	>=	value = 2 >= 1	True
Greater Than or Equal to	>=	value = 1 >= 2	False

Greater Than or Equal to	<=	value = 1 >= 1	True
--------------------------	----	----------------	------

The results are keywords in Python, so you can not have a variable, function or class with the name "True" or "False".

All these logical operators are used to control the program, to check bounds, to check equivalence and you can explore their use with both numbers and text... Lets take a look at some examples....

```
>>> x = 1
>>> y = 2
>>> print (x == y)
False
>>> z = 3
>>> print (z == (x + y))
True
>>> print (x < y)
True
>>> print (y > z)
False
>>> print (z >= x)
True
>>> print (x <= y)
True
>>>
```

Lets look at using them...

3.1 Equals

You will immediately notice that equals is a double symbol, we already have learned a single "=" symbol means assign, so we have to use a different symbol to tell Python we mean for it to evaluate equality between the items on either side.

Equals, represented with a double '==' symbol, to differentiate it from assignment being a single '=', will check either side of itself are the same returning true, otherwise returning false. Like this:

Equals Operator

Test	A	B
(A == B) is True	0	0
(A == B) is False	1	0
(A == B) is False	0	1
(A == B) is True	1	1

The last test we see in this table is important, if we compare a number variable to a variable of another type, such as a string or character, then they are not equal. So "0" will not equal the number 0.

The "==" symbol can also be represented as "is", see 3.8.3 below.

3.2 Less Than

Used almost exclusively with numeric values, we determine if the left side is less than the right.

With all the chevron based comparators the larger value we're logically expecting should be against the open end of the chevron in our test...

Less Than Operator

Test	A	B
(A < B) is False	0	0
(A < B) is False	1	0
(A < B) is True	0	1
(A < B) is False	1	1

3.3 Less Than or Equal To

To doing two tests, which might not be easy to maintain, most all languages provide "less than or equal to" as a single operation, like its cousin "less than" we can define it thus:

Less Than Or Equals Operator

Test	A	B
(A <= B) is True	0	0
(A <= B) is False	1	0
(A <= B) is True	0	1
(A <= B) is True	1	1

3.4 Greater Than

I'm sure we can speed up now, and define just the tables to explain these operators:

Greater Than Operator

Test	A	B
(A > B) is False	0	0
(A > B) is True	1	0
(A > B) is False	0	1
(A > B) is False	1	1

3.5 Greater Than or Equal To

Greater Than or Equals To Operator

Test	A	B
(A >= B) is True	0	0
(A >= B) is True	1	0
(A >= B) is False	0	1
(A >= B) is True	1	1

Note with the greater than, that the larger item is considered to be at the open end of the chevron.

3.6 Not

Not, or sometimes called "inverse", is the logical opposite of any of the tests we've discussed so far, so if we just define a true value:

```
a = True
```

Now we want to use the inverse, we can do this when we call a function, like print!

```
print(not a)
```

Our output will be "False". "not" is a keyword in python, and it can be placed before any boolean logical test you perform, or before any variable which contains a boolean (True or False) value.

3.7 Boolean Theory

A boolean is a variable type which can contain either the value True or the value False.

The boolean is named for the mathematician George Boole, who derived the theory of "Boolean algebra", all the logical operators, and the "not" keyword, which we've covered thus far are all part of the boolean logic system within computer science, where every evaluation is returned as a True or False result.

This is very easy for a computer to emulate, as the Binary nature of the computers memory can easily represent a True as a one and a False as a zero.

3.8 If Statements

The point of all this, somewhat difficult to grasp logic, is our wanting to control the flow of our programs, the first and most simple way we can control the flow of the code is with the "if statement", this is a piece of code using the "if" keyword followed by any of the logical tests we've just discussed, it decides if we are going to run the following piece of code or not. This is our most basic test:

```
a = 10
max = 100

if a == max:
    print("They are Equal")
```

Lets digest this code, we define two numeric variables for our comparison. Any of the tests we've discussed so far could go in place of "==", including the word "is", see section 3.8.3.

The last item on the "if" statement declaration line is a colon, to tell Python that we're done with our tests and want to run the following code, each line of the code block we intend to run as part of the success of the statement **is indented with a single tab** (it could just be one or any number of spaces, but convention generally sticks to a tab).

Code which is not indented, following this statement is considered to be back in the regular program, like this:

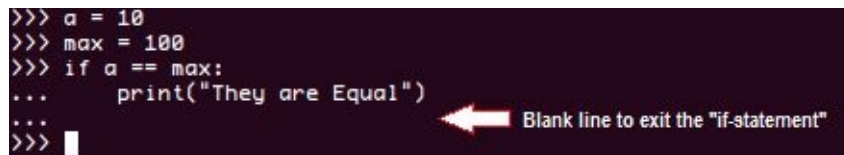
```
a = 10
max = 100

if a == max:
    print("They are Equal")

print("I run no matter what the result of the 'if'")
```

The indentation, or white space, is how Python differentiates between code inside a statement, loop or function and code which is not, blocks of code tabbed together like this will become more and more common as we progress.

If you are typing the code directly into the python interpreter, when you start your "if-statement" you will see the lines show as beginning with "...", you indent your lines with a tab here, to stop lines being within the "if" you simply enter a blank line...



```
>>> a = 10
>>> max = 100
>>> if a == max:
...     print("They are Equal")
...
>>> 
```

A red arrow points to the blank line after the indented code, with the text "Blank line to exit the 'if-statement'" next to it.

Entering this code however, we don't see the string "They are Equal", so we change the value of "a" to get a match...

```
a = 100
max = 100

if a == max:
```

```
print ("They are Equal")

print ("I run no matter what the result of the 'if'")
```

Its recommended to try a few of the other logical operators before you go any further.

You can of course follow one "if" with another...

```
a = 100
max = 100

if a == max:
    print ("They are Equal")

if now a == max:
    print ("They are NOT equal")

print ("I run no matter what the result of the 'if'")
```

3.8.1 If-Else statement

At the end of the previous section we saw two "if" statements, being the opposite result to one another, this is a little inefficient as the code needs to perform two tests in order to work out which statement to enter, this is slower than using a new idea the "else".

```
a = 42
max = 100

if a == max:
    print ("They are equal")
else:
    print ("They are not equal!")
```

This code performs a single test, when that test is "True" the first block of code is run, otherwise the second. It is easier to read and quicker than writing separate "if" statements one after another.

We are still able to apply any test into the first test too...

```
a = 42
max = 100
if not (a == max):
    print ("They are NOT equal")
else:
```

```
print ("They are equal!")
```

The brackets around "a == max" are not necessary, but they help make this code a little more readable.

3.8.2 Else-If statement

When we have more than just one test and it's opposite to handle, we can include more and more tests together...

```
name = "Xelous"

if name == "Peter":
    print ("Hello to Peter!")
elif name == "Paul":
    print ("Hi Paul!")
else:
    print ("Well, hello, whoever you are!")
```

This is not the same as writing lots of "if" statements in a series, as if two tests can result in a "True" result, only the first of those tests will move into a block of code...

```
a = 1
b = 2

if a == b:
    print ("Equal")
elif a < b:
    print ("a is smaller")
elif b > a:
    print ("also means a is smaller!")
else:
    print ("unknown")
```

We can agree that the tests "a < b" & "b > a" are the same result.

However, only the first string is output as that test was found to be positive ending the if-else-if cascading down to the other tests in the series.

This is known as "mutual exclusion", one test being successful stops all the rest occurring.

There is no limit to how many "else if" statements you can add to the list, you also do not necessarily have to have a final "else" to handle when none of the tests you perform activate.

3.8.3 Is

The keyword "is" represents another method of testing for equality, it is used exactly the same way as the "==" operator...

```
a = 1
b = 2
if a is b:
    print("They match")
else:
    print("They are different")
```

3.9 Combining Logical Tests (AND & OR)

We can perform more than one logical test at a time, lets say we want to check a pair of values before doing some code:

```
a = 1
b = 2
name = "Xelous"

if a == b and name == "Xelous":
    print("Match!")
else:
    print("Failed")
```

We get a fail when we run this code, we received a "False" from `a == b` this automatically failed the "and" because "and" needs both sides of it to evaluate as "True"...

"and" Operator

A	B	Test
False	False	(A and B) is False
True	False	(A and B) is False
False	True	(A and B) is False
True	True	(A and B) is True

The "or" test is written in the same way, however it allows the test to complete like this:

"or" Operator

A	B	Test
False	False	(A or B) is False
True	False	(A or B) is True

False	True	(A or B) is True
True	True	(A or B) is True

Some example code might be:

```
a = 1
b = 2

if a == b or a < b:
    print("Good")
else:
    print("Bad")
```

We should get "Good" as our output, because a is less than b, and that's enough to satisfy an "or" check.

Again, just to aid reading of our code we can use brackets...

```
a = 1
b = 2

if (a == b) or (a < b):
    print("Good")
else:
    print("Bad")
```

Finally, we can combine both "and" & "or" statements:

```
a = 1
b = 2
c = 3

if ( (a == b) or (a < b) ) and (c > 0):
    print("Good")
else:
    print("Bad")
```

So what this does is "and" the result of an "or" with another logical test.

It is important when programming to think about whether a compound of many logical tests like this is better for maintaining and understanding your code than perhaps just using a set of individual "if-else" statements.

3.9.1 Normalizing inputs as Booleans

So far we've used all the raw logical tests, what we have not done is actually used the "bool" function. A little as "str" gave us strings from numbers and "int" gave us whole numbers from strings the function "bool" converts the given variable to a pure "True" or "False".

```
a = 1
b = 2
x = bool(a == b)

print(x)
```

Using the "bool" we can get the result of the logical check and assign it to the variable named "x". We could then use this in any statement:

```
if x:
    print("Good")
else:
    print("bad")
```

Changing a test into a boolean value with "bool" is called normalizing.

3.10 Not

Through all the logical tests we have covered there another important item to use, it is not a test but influences the tests we are performing.

It is called "not" and is a keyword, it inverts a test result of "True" to "False" and vice versa.

```
a = True
b = False
x = not a == b
print(x)
```

Here "a" did not equal "b", yet "x" prints as "True" because we used "not". We could have used a part of parenthesis to make this code more readable, like so...

```
a = True
b = False
x = not (a == b)
print(x)
```

We can also perform "not" directly in code, without the need to assign them a result, like so...

```
FirstValue = 10
```

```
SecondValue = 30
if not FirstValue > SecondValue:
    print ("First is Less than Second!")
else:
    print ("Second was larger than First!")
```

This example is of course a little silly, we could just use the "less than", however it is good to understand that you can inverse any result.

Such as with text...

```
name = "Mary"
print ("Enter a name...")
altName = input()

if not name == altName:
    print ("You are not 'Mary'!")
```

3.11 Empty values with None

The keyword "None" is a special value in Python, it literally means "nothing is here" or "no value within".

It can be assigned to any variable name to indicate it has no value, in combination with the "not" we've already learned about we can therefore determine when a value has been set...

```
someInformation = None

someInformation = CreateInformation()

if not someInformation == None:
    print ("Data Exists!")
else:
    print ("Error Data Missing")
```

3.12 Not Equals

The inverse of an equals is known as a "not equals", we've seen this used in sections 3.10 and 3.11, a much simpler way to achieve this exact same check. By using the "!=" operator symbol...

"!=" Not Equals Operator

A	B	Test
False	False	(A != B) is False

True	False	(A != B) is True
False	True	(A != B) is True
True	True	(A != B) is False

A simple code example is to have two different values, running "!=" results in a positive result...

```
a = True  
b = False  
x = a != b  
print (x)
```

```
>>> a = True  
>>> b = False  
>>> x = a != b  
>>> print (x)  
True  
>>>
```

Chapter 4: Functions

We have briefly touched on using functions in our learning... "print", "input", "int", "str" and "len" to name a few of them. There are however hundreds more within python and the libraries which are supplied with it to help you achieve your goals.

Functions we define ourselves are also possible and extremely useful, allowing us to move sections of code we want to reuse, saving us from repeating chunks of code when we write a program. Functions also allow us to break problems we're solving into more manageable pieces.

Instead of writing hundreds of lines of code in one section, we could break them down into individual functions.

4.1 Defining Functions

To define our own function we use the keyword "def" and follow this with a name, like this:

```
def FirstFunction():
```

Notice the colon at the end, this tells python we have finished defining the name of the function, just as it told python we had finished defining a check within an "if" statement.

So, with this function named "FirstFunction" we're not yet taking any parameters in; so our brackets are empty. Lets have the function just print a message so we know we called it...

```
def FirstFunction():  
    print("Hello from our function!")
```

The block of code within the function is indented, as it was with the other statements we've covered.

Whenever we want to call this code we can use the name "FirstFunction()". Lets take a look...

```
def FirstFunction():  
    print("Hello from our function!")  
  
print("We are about to call our function....")  
FirstFunction()  
print("We called it...")
```

We can only call the function after it has been defined, calling the name before the "def" will fail...

```
print ("We are about to call our function....")  
FirstFunction()
```

```
def FirstFunction():  
    print ("Hello from our function!")
```

```
>>> print <"We are about to call our function...">  
We are about to call our function...  
>>> FirstFunction()  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
NameError: name 'FirstFunction' is not defined  
>>>
```

Python fails, because the name "FirstFunction" doesn't belong to a function yet. So you may only call a function after it is known to Python.

4.2 Returning Results from Functions

Aside from letting you put blocks of code together, as a kind of management facility, functions also allow you to mould a function to perform a specific task and return a result or perform an activity for us. This is exemplified in pretty much all the functions we have seen so far, so lets take a look at how we might do the same.

To return a value from a function we can use the keyword "return", this is placed at any location within the function, especially at the end, to make the function return a value like this:

```
def TheMeaningOfLife():  
    print ("The meaning of life is...")  
    return 42
```

You could combine the logic operators and if statements do perform a task for us, and return a value from different points:

```
def AskMeaning():  
    print ("What is the meaning of life? ")  
    textInput = str(input())  
    valueOfInput = int(textInput)  
    if valueOfInput == 42:  
        return True  
    else:  
        return False
```

This function asks for an input and checks if the value is correct returning True, otherwise False. With two return statements there are therefore two locations from which the function ends... We could, and some programmers prefer to return once, to allow easy debugging or

maintenance of code later. That code would look like this:

```
def AskMeaning():
    result = False
    print ("What is the meaning of life? ")
    textInput = str(input())
    valueOfInput = int(textInput)
    if valueOfInput == 42:
        result = True
    return result
```

Once you return from the function, or a function block just ends the program returns from where you made the call into the function... Lets use "print" to see this in action...

```
def Message():
    print ("This is the function")

print ("Main Program...")
Message()
print ("This is the main program again")
```

The result of this code are the lines output in order. Before, during and then after the function.

When we return values from functions, like the number 42 previously, we can use these values just as we use other variables, assigning the values to variables or just performing functions directly on the result... Like this...

```
def TheMeaningOfLife():
    return 42

print ("The value is " + str(TheMeaningOfLife()))

theValue = TheMeaningOfLife()
print ("The Meaning of Life value is: " + str(theValue) )
```

We could of course return the string directly, like this...

```
def TheMeaningOfLife():
    return str(42)

print ("The value is " + TheMeaningOfLife())

theValue = TheMeaningOfLife()
print ("The Meaning of Life value is: " + theValue )
```

Or like this...

```
def TheMeaningOfLife():  
    return "42"  
  
print ("The value is " + TheMeaningOfLife())  
  
theValue = TheMeaningOfLife()  
print ("The Meaning of Life value is: " + theValue )
```

But of course the strings can't be used in a calculation; at least not until one uses "int" on them again.

You can operate mathematically on returns from different functions as well, either directly returning them or by assigning them to a variable first... Like this...

```
def A():  
    return 1  
  
def B():  
    return 2  
  
result = A() + B()  
  
print (result)
```

Or this...

```
def A():  
    return 1  
  
def B():  
    return 2  
  
a = A()  
b = B()  
result = a + b  
  
print (result)
```

4.3 Naming functions

When you come to name functions you should really practice naming them something

meaningful, related to their operation perhaps, but always descriptive. The use of capitals or not is entirely up to your own liking in Python, the language has no requirement to name a function "Hello" or "hello" or "HELLO". However, other developers using your code might like some semblance of order, so once you have elected to name functions sensibly and in some discernible manner, stick to it.

Once you define a function, that name can no longer be used elsewhere, so a function called "A" can not contain a variable named "A", doing this will result in a "SyntaxError".

Similarly, if we define a function like "A" and don't use the brackets to tell Python "this is a function call", like this:

```
def A():  
    return 1  
  
print (A)
```

Then we get a "NameError", with the information "name 'A' is not defined".

This is simply python not knowing of a variable named "A" because of the missed brackets to tell it to call the function.

Consistent naming of your functions from the moment you start to create them will benefit you in the long term, make your code understandable by others, easy to document (if it even needs documenting at all) and allows you to identify where and remove errors.

Good practice in naming functions will then only leave us with the risk of accidentally naming a function with a "keyword" or a name already in use.

4.4 Passing Parameters to Functions

When we called "print" previously we were passing it a parameter, sometimes called an argument, each time. The print function therefore would have been defined something like this:

```
def Message(OurText):  
    print (OutText)
```

Our definition changes by adding the parameter between the brackets after the name, so the name of our parameter is "OurText".

We can call this new function now:

```
def Message(OurText):  
    print (OutText)
```

```
Message("Hello World")
```

Print is a bit of a special case, we know it can take a string or number without changing, and we don't define what type our parameter is, so by coincidence our "Message" function here will like-wise take a number as well, without needing to change:

```
def Message(OurText):  
    print (OurText)
```

```
Message(54321)
```

If we pass a parameter within a function, which is used in a strange or incompatible manner (just as with mixing str and int) then things will go wrong!

When you name your parameters just like the functions before then, they should have meaningful names and the names used should not have already been used outside the function... The names of a parameter are only unique within the function itself, so in this code, we can have the names like this....

```
text = "Hello"
```

```
print (text)
```

```
def Message(text):  
    print (str(len(text)) + ":" + text)
```

```
Message("Mary had a little lamb")
```

The "text" variable with just the word "Hello" is nothing to do with the function, it is something known as a global in that it is in the main body of the program. And by sheer, intentional, coincidence the parameter to the "Message" function is also called "text", but that parameters name is ONLY used within the function!

So the "text" within the function is the string passed when the function is called, and it contains "Mary had a little lamb", not "Hello".

We can pass multiple parameters to a function by separating the various parameter names with a comma:

```
def Sum (val1, val2, val3):  
    print ("Summing...")  
    print (val1)  
    print (val2)
```

```
print (val3)
temp = val1 + val2 + val3
print ("Result " + str(temp) )
return temp

x = 1
y = 2
z = 3
r = Sum(x, y, z)
```

And the parameters don't need to all be the same type:

```
def Person(name, age):
    print ("Person, named: " + name)
    print ("Aged " + str(age) + " years")

ageValue = 12
nameValue = "Xelous;
Person(nameValue, ageValue)
```

They do need to be the same type for adding together to print though, hence the "str" use for the numerical age passed in.

4.5 Nesting Functions

We can of course define multiple functions and call them from one another, like this:

```
def square(value):
    return value * value

def sum(first, second):
    return first + second

def Hypot(length, height):
    t1 = square(length)
    t2 = square(height)
    return sum( t1, t2 )

print ("Triangle is: ")
tLen = 10
tHei = 12
print (tLen)
print (tHei)
```



```
tHyp = Hypot (tLen, tHei)  
print (tHypo)
```

Now pretend we only want a user of our code to see the "Hypot" function, we want to hide the "square" and "sum" functions from them... We do this with nesting, like so...

```
def Hypot(length, height):  
    def square(value):  
        return value * value  
  
    def sum(first, second):  
        return first + second  
  
    t1 = square(length)  
    t2 = square(height)  
    return sum( t1, t2 )  
  
print ("Triangle is: ")  
tLen = 10  
tHei = 12  
print (tLen)  
print (tHei)  
  
tHyp = Hypot (tLen, tHei)  
print (tHypo)
```

You can now clearly see that the "sum" and "square" functions are tabbed in one step, and defined after the "Hypot" define. This means that only the Hypot functions content can see them!

The content of "Hypot" follows, it is tabbed in once and performs the same calls as before and the same return.

You only need remember to double tab indent the blocks within the two nested functions, so Python can tell which function the various lines belong to.

However, what if we have this code...

```
def sum(x, y, z):  
    return x + y + z  
  
def Hypot(length, height):  
    def square(value):  
        return value * value
```

```
def sum(first, second):
    return first + second

t1 = square(length)
t2 = square(height)
return sum( t1, t2 )

print ("Triangle is: ")
tLen = 10
tHei = 12
print (tLen)
print (tHei)

tHyp = Hypot (tLen, tHei)
print (tHypo)
```

Now we have a "sum" inside the main body of the program, and another nested within Hypot... The code still works though, "Hypot" will not use the copy of the function from the outside which takes three parameters, because it is a different scope.

Inside the function is one scope, and outside the function is another, when Python is within the function and looking for "sum()" it looks in it's local scope first. Only if it fails to find the function will it move up a level of indents. Again if its not found it goes up another level, and so on until it finds the function or causes an error.

The "sum" which takes three parameters is in what is called the global scope, there is a special reserved word to let functions access things from outside themselves without complaining... You guessed it, it's the keyword "global".

4.6 Global Variables

```
Count = 0

def Increment():
    Count = Count + 1
    print (Count)

print (Count) Increment()
print (Count)
```

This is a clearer example than the "sum" function from the previous section, we clearly have a "Count" inside the function and one outside the function, and when we called "Increment" the outside "Count" does not change.

Because we changed the instance inside the function. If we just try to use the one outside...

We get a horrible error: "UnboundLocalError"

```
>>>
>>> Count = 0
>>>
>>> def Increment():
...     Count = Count + 1
...     print (Count)
...
>>> print (Count)
0
>>> Increment()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in Increment
UnboundLocalError: local variable 'Count' referenced before assignment
>>>
```

The function is looking in itself for something named "Count" and cannot find it! We just start to try and modify it.

The solution is to tell the function that "Count" is in the global scope...

```
Count = 0

def Increment():
    global Count
    Count = Count + 1
    print (Count)

print (Count) Increment()
print (Count)
```

Now the function has been told "when you see something called Count, it is in the global space", so it looks for it, finds the variable from the first line of our code and it increments that value!

This is different to passing variables and returning results from a function, and though it is beyond the scope of this tutorial when you start to use advanced techniques like threading (running more than two activities at a time) then you need to take heed of which global is being accessed by what and when, otherwise errors can occur.

4.7 Pass

Pass is another keyword, which we will introduce now, though it is not solely related to functions.

it is used to skip out of the current block of code, a little like "return" did, however it returns no value and Python drops all references and requirements of the code, it is literally passed over.

```
def Foo():  
    pass  
  
def Baa(message):  
    foo()  
    print ("This is Bar")  
  
Baa()
```

Here the call to "Foo" does nothing, it is passed over, only "Bar" results in any activity.

You can skip any block formed statement, such as an "if" statement...

```
x = 1  
y = 2  
if x < y:  
    pass
```

You can continue to code without worrying that you have not fleshed out all the detail before proceeding.

Chapter 5: Loops

We've discussed packaging code within functions, however we've not thought about any situation where we might want to perform repeated calls to the same function over and over.

Making consecutive high-speed calls to repeat automated tasks is something Computers are extremely good at, it pretty much defined why they were invented and how they were initially employed in business, removing the tedium, most errors and human factor in performing complex large scale calculation.

History lesson over, lets take a look at how not to do repetitive tasks....

```
def Increment(value):  
    return value + 2  
  
start = 0  
  
start = Increment(start)  
start = Increment(start)  
start = Increment(start)  
start = Increment(start)  
start = Increment(start)  
  
print (start)
```

We can see this function counts up by two each call, we use it over and over to print a final result... Pretty clumsy, lets look at better ways to do this same task, with various loops.

5.1 While Loop

The first loop we'll use is a "while loop", to perform the same task as we've seen, counting by two...

```
def Increment(value):  
    return value + 2  
  
start = 0  
  
while start < 10:  
    start = Increment(start)  
  
print (start)
```

You will notice the use of the "less than" operator, to check the value we have accrued thus far... This check returns it's "True" or "False" value into the "while" keyword which checks whether to continue into the loop or end... "True" must be passed into the "while". Some therefore refer to the "while loop" as a "while true loop".

Any of the logical operators can be used to indicate the while should continue into it's block of code... Or one can simply set it to use a variable name, which is set to "True" and then set that same variable to "False" when one wants to exit, thus...

```
RunningFlag = True
start = 0
while RunningFlag:    start = start + 10
    if start > 100:
        RunningFlag = False

print ("Done!")
```

5.2 Break Statement

Sometimes there may be special need to immediately exit a loop, perhaps when an error is encountered, or some condition is met, to do this we use the keyword "break". Take this code for example...

```
start = 0

while True:
    start = Increment(start)

print ("Done!")
```

There is no exit from this loop, it would literally run forever!

To break out of the loop we can employ the "break" keyword, like this....

```
start = 0

while True:
    print ("Started Loop")    start = Increment(start)
    if start > 10:
        print ("Breaking loop!")    break
    print ("Ending Loop, going back to the while")
print ("Done!")
```

You will see here that we print "Started Loop" and then "Ending Loop, going back to the

while" for all the sets until the last, where we immediately see "Breaking loop!", the "Ending" print on that cycle is skipped the loop immediately exited!

The "break" keyword can be used in any loop in this chapter.

5.3 For Loops

Creating your own count or using a break is not always ideal, a good example is the "for-loop", which can loop through all the entries in something else... Like a string...

```
text = "this is the start of something"

for char in text:
    print(char)
```

This is a typical for loop, and contains the two keywords "for" and "in". In this instance "in" is not seeking for anything, as we saw previously, instead it is telling Python to break the right hand item, the variable "text", into individual parts. For a string those parts are individual characters.

The result is our code prints each character its own line of output.

We can now easily imagine how the "len" function works with such a loop within it...

```
def NewLen(text):
    count = 0
    for char in text:
        count = count + 1
    return count

print(NewLen("Mary had a little lamb"))
```

That might essentially be one way in which the "len" function could be written, it certainly works!

Any type in Python which can be broken down into individual pieces, we will learn about a few more as we progress, can be processed through just such a "for-loop".

5.4 Ranges

Another way to move over a set of numbers in a "for-loop" is with yet another function called a "range", which given a number gives us every number from zero to the inputted parameter value... So "range(3)" gives the return list of 1, 2, and 3.

We can use it like this...

```
sum = 0

for x in range(10):
    sum = sum + x

print (sum)
```

The number we pass to "range" can be anything, such as the length of another string...

```
text = "Mary had a little lamb"
sum = 0

for x in range(len(text)):
    sum = sum + x

print (sum)
```

5.5 Continue

We've covered finishing a loop normally, flagging with a boolean or logic operation and even just breaking out midway through a loop. However there is one more statement to make you aware of, the "continue" statement.

Which is a little like the "pass" statement we've discussed, except instead of passing over the block out and doing nothing, it jumps us back to the top of the loop.

```
count = 0

for x in range(100):
    count = count + 1
    if x < 95:
        continue
    print ("Over fifty!")

print ("Total Counts: " + str(count) )
```

With this code, we only see the message printed for the last few counts, however the total counted loop passes performed is 100, just like the range!

Each time it reached the "continue" it moved back to the top of the loop, never printing, but counting each time.

Imagine using this with the "input" commands, to check the input is valid in a loop, using continue to jump back to the top for new input until it is valid.

A continue can be used in either a "for" or a "while" loop.

Chapter 6: Lists

We spoke about types we could step through within loops, one of the most common of these is a list. Lists are represented by a name in just the same way as other variables, but instead of their containing one value, the single name contains many values, each placed on after the other.

We must think of lists as containing many cells, you can assign a new value or read the current value out of each cell using the square brackets "[" operation, lets take a look at a list into which we place three numbers...

```
numbers = [ 1, 2, 3 ]
```

We can print the whole list in one call to "print (numbers)", or you could print each element like this...

```
print (numbers[0])  
print (numbers[1])  
print (numbers[2])
```

You will notice that the first cell is NOT addressed as one, it is zero... If you were to do a loop counting through the list, it would look like this, using the "len" function to tell you how many items are in the list, just as you did with a string.

```
numbers = [ 1, 2, 3 ]  
count = 0  
max = len(numbers)  
while count < max:  
    print (numbers[count])  
    count = count + 1
```

Or as a for-loop, you might use the range of the "len" to count through them as well, but you can use "in" as well...

```
numbers = [ 1, 2, 3 ]  
for CurrentNumber in numbers:  
    print (CurrentNumber)
```

We can place any type into a cell of a list, a list of strings might be the days of the week...

```
DaysOfTheWeek = ["Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday",  
"Saturday"]
```

You will notice that each element is simply separated from the previous item with a comma. If we are entering strings which contain comma's, then Python knows to look for the string as it is bordered with either style of quotes.

And again we can print such strings out...

```
for day in DaysOfTheWeek:  
    print (day)
```

We can use a list in a function, we'll use the "DaysOfTheWeek" above and create a function to return the day of the week from a value 1 to 7... This is a human input index value, whilst we know the list index starts from zero, so we need to take that into account within our code...

```
DaysOfTheWeek = ["Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday",  
"Saturday"]
```

```
def GetDayOfWeek(Index):  
    result = None  
    if Index > 0 and Index < 8:  
        global DaysOfTheWeek  
        result = DaysOfTheWeek[Index - 1]  
    return result
```

```
print (GetDayOfWeek(1))  
print (GetDayOfWeek(7))  
print (GetDayOfWeek(3))
```

In terms of complexity this example is somewhat more complex than those before it, we are re-using many of our previous elements, such as "global", using a list "[]" accessor, altering the parameter "Index" with an arithmetic, we also have one "return" point, and so the "result" we are setting up has a default value of "None". Take sometime to go through this example.

Of course we could document this function to any user, explaining to the range of valid input is 1 to 7 inclusive and that upon any problem a "None" type is returned, otherwise a string is returned with the correct day name starting from Sunday.

However, this code is self-documenting, it is simple enough to understand without the need for additional commentary.

As could add a comment to our code though, like so...

```
#-----  
# The global string list used for the  
# days of the week functionality
```

```
#-----
DaysOfTheWeek = ["Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday",
"Saturday"]

#-----
# Returns the "Day Of the Week" as a string
# when presented with a day index 1 to 7,
# Sunday to Saturday, else it returns a
# None type value upon an index or range
# error
#-----
def GetDayOfWeek(Index):
    result = None
    if Index > 0 and Index < 8:
        global DaysOfTheWeek
        result = DaysOfTheWeek[Index - 1]
    return result

print (GetDayOfWeek(1))
print (GetDayOfWeek(7))
print (GetDayOfWeek(3))
```

Any line of code starting with the hash symbol "#" is not creating executable code, it is simply text for display.

6.1 Empty Lists, Append & Mixed Item Types

You do not need to always create a list with contents in it, you can start with an empty list like so...

```
numbers = []
```

This is not a "None", it is a list, calling "len" will yield a length of zero...

```
numbers = []
print ("Length: " + str(len(numbers)))
```

The list has no idea what kind of items you have inserted into it, indeed you can insert different items at different places in the list... So, unlike when adding strings up to print out, you do not need to ensure everything was a string with the "str" function...

```
items = []
items.append("hello")
items.append(123)
```

```
for i in items:  
    print (i)
```

It is perfectly valid, to append a string and then append a number to a list.

"append" is the first important function we will come across with a list, and I'm sure you just understood what it did, it adds a new cell to the end of the list and populates that cell with an element value you passed in.

If you address an item beyond the end of a list, you will get an error...

```
>>> items = []  
>>> items.append("Hello")  
>>> items.append(1)  
>>> for i in items:  
...     print (i)  
...  
Hello  
1  
>>> items[2] = "Hello"  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
IndexError: list assignment index out of range  
>>>
```

You see, we performed the previous code, but then tried to assign position 2... Only positions 0 and 1 had been initialised, so the call to put "Hello" into position 2 fails.

You must therefore use "append".

6.2 Updating Values

You can update the value inserted into an element with the square brackets "[]" in the same way you read them back out for printing...

```
numbers = [ 1, 2, 100, 4, 5 ]
```

```
for n in numbers:  
    print (n)
```

```
numbers[2] = 3
```

```
for n in numbers:  
    print (n)
```

You can assign any value back into a cell in a list, so long as the list address you are trying to use is in range (use "len" to tell you the maximum address)...

```
numbers = [ 1, 2, 3, 4, 5 ]
```

```
for n in numbers:
    print (n)

length = len(numbers)
print ("The Number of items in the list is : " + str(length) )

max = length - 1
print ("The highest address is therefore : " + str(max) )

numbers[max] = 42

for n in numbers:
    print (n)
```

6.3 Inserting to Lists

Appending an element always puts the new item at the end of the list, if we wanted to add a new element somewhere else we need to use the "insert" function:

```
Months = [ "January", "March", "April" ]

Months.insert(1, "February")

print (Months)
```

The first parameter of insert is the index at which you want to place the new item, we are starting from 0 once again, so "January" is at zero, we insert at 1, shuffling all the other elements along to make space, then place the second parameter, which is "February" into the new location.

You can again insert any new item of any type, the intended index passed must be less than the maximum address already created; otherwise you may as well have used "append".

6.4 Adding Lists Together (extend)

We also have a function which lets us add one list to another, it is called "extend", lets make two sequences:

```
Start = [ 1, 2, 3, 4, 5 ]
End = [ 6, 7, 8, 9, 10 ]

print (Start)
print (End)
```

Start.extend(End)

```
print (Start)
```

The second printing of "Start" will show us the whole sequence and just as with other inserts or edits you can have any mix of types inside the list, so you can add lists of anything to any other things in this way.

The "End" list in this example remains unchanged.

6.5 Removing from Lists

If we have a list, lets go with "DaysOfTheWeek" again, and we want to remove an element we use the function "del", passing to it the element by index.

However, "del" is not a function within the list, we use it a little as we do "len". Lets delete Tuesday, I've never liked Tuesdays... In our list this was index 2.... Our delete call looks like this...

```
del (DaysOfTheWeek[2])
```

6.5.1 Pop

"pop" is a special kind of "removing", it removes the last element from a list but also returns it to you, unlike "del" which just obliterates the element...

```
numbers = [ 1, 2, 3, 4, 5 ]

while len(numbers) > 0:
    currentItem = numbers.pop()
    print (currentItem)
```

This is useful to make your list act like a "stack", which is one of the most common data structures in general computing & programming.

6.6 Sorting Lists

We can define a list out of order, lets try this with numbers:

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

Then when needed we can sort the list into an order... Numerical or Alphabetical order are the default...

```
nums.sort()
```

The numbers are placed in ascending order by default and the same applied if we sort strings:

```
days = [ "Mon", "Tue", "Wed", "Thurs", "Fri" ]  
days.sort()  
print (days)
```

The new order of "Fri", "Mon", "Thurs", "Tue" and "Wed", of course makes no real-world sense but it is alphabetically correct!

6.6.1 Creating Sorted Lists

If we want to make a copy of an unsorted list into a sorted form, Python provides a special function, called "sorted". You pass this function any list and it returns the same elements in a new list but sorted, letting you retain the original.

```
oldlist = [ 4, 5, 1, 3, 2 ]  
  
newlist = sorted(oldlist)  
  
print (oldlist)  
print (newlist)
```

You can pass the unsorted list directly to the "sorted" function too...

```
newlist = sorted([ "b", "d", "a", "c" ])  
  
print (newlist)
```

6.6.2 Reversing Lists

Another function built into lists is to simply reverse their order:

```
numbers = [ 1, 2, 3, 4, 5 ]  
print (numbers)  
  
numbers.reverse()  
print (numbers)
```

You can reverse the order once, and reverse it again. Often times developers use the reverse function in order to achieve appending to the front of a list, simply reverse the list you hold append and reverse again.

This is known as a queue, but there are better ways to implement this.

We can also create reverse sorted lists straight away with "sort" function, but we give the sort a parameter called "reverse", lets take a look:

```
numbers = [ 2, 5, 4, 3, 1 ]  
  
nums.sort(reverse=True)  
  
print (nums)
```

Instead of two lines of code, "sort" then "reverse", we shortened our code. However some developers consider this to be more confusing and harder to maintain for little benefit, the two separate lines did exactly the same action, here you may miss; at a glance; that there are two actions being carried out. Making your code harder to maintain or document.

The problem is compounded further if we use the "sorted" function in the same manner.

```
numbers = [ 2, 5, 4, 3, 1 ]  
  
result = sorted(numbers, reverse=True)  
  
print (result)
```

6.7 Counting Occurrences

Lets say we have a list of values, like a series of numbers and we want to count how many occurrences of a certain number we have... We use the "count" function and pass it the item we want to find...

```
numbers = [ 1, 2, 3, 4, 5, 1, 4, 3, 2, 5, 1, 2]  
count = numbers.count(2)  
print ("We have " + str(count) + " instances of the number two!")
```

We could use this to count how may vowels we have in some input...

```
print ("Please enter some text..")  
inputString = input()  
  
characters = list(inputString)  
  
CountOfA = characters.count("A") + characters.count("a")  
CountOfE = characters.count("E") + characters.count("e")  
CountOfI = characters.count("I") + characters.count("i")
```

```
CountOfO = characters.count("O") + characters.count("o")
CountOfU = characters.count("U") + characters.count("u")

print ("We have " + str(CountOfA) + " A's")
print ("We have " + str(CountOfE) + " E's")
print ("We have " + str(CountOfI) + " I's")
print ("We have " + str(CountOfO) + " O's")
print ("We have " + str(CountOfU) + " U's")
```

6.8 List Parsing

In the previous example we slipped in a new keyword "list". This is another special function, to create lists is the function "list", this takes an input and turns it into individual elements, just as we saw used in the "for" loop previously.

If you provide it with a string, it would turn the string into individual characters:

```
Name = "Xelous"
NameList = list(Name)

print (NameList)
```

If you attempt to turn something which has no sub-elements into a list, i.e. a whole number, you will get a "TypeError" with the follow up that the type "object is not iterable".

6.9 Lists within Lists

We've already seen you can mix the type of items in the cells of a list, we could therefore present all the information about something in one list... Lets take a student record of "StudentId", "Name", "Course" and "Year of Entry"...

```
studentA = [ 1, "Xelous", "Geography", 2016 ]
studentB = [ 2, "Marvin", "Robotics", 2016 ]
studentC = [ 3, "Zaphod", "Politics", 2016 ]
```

On their own each of these lists is hard to collate into anything, say the class register list... However, we can do this simply, and in the same way, putting the existing individual student lists into one master register list...

```
classRegister = [ studentA, studentB, studentC ]

print (classRegister)
```

If we wanted to access the second students Course, we could have to use two sets of square

brackets, like so...

```
print (classRegister[1][2])
```

The first square brackets takes us to the student in the "classRegister", and the second square brackets take us to the third item of that student.

6.10 Advanced Sorting

With the student list from the previous section, we can sort the list into order:

```
classRegister.sort()  
print (classRegister)
```

It will default to using the first element of the lists inside the class list, so the "StudentId" field. However, lets say we don't want to order by their student id, but by name... How do we do this?

Well, we need help from python. We'll cover this in more detail later, however for now we're going to use a new keyword pair called "from" and "import", and we're going to ask python to import a special piece of code to let us pick which field in our list to sort by:

```
from operator import itemgetter
```

We're asking for a function called "itemgetter" be imported into our program from a library called "operator", don't worry about the details of this yet, just know that this is how we unleash the power of the huge library collection within Python!

```
from operator import itemgetter
```

```
studentA = [ 1, "Xelous", "Geography", 2016 ]  
studentB = [ 2, "Marvin", "Robotics", 2016 ]  
studentC = [ 3, "Zaphod", "Politics", 2016 ]
```

```
classRegister = [ studentA, studentB, studentC ]
```

```
print (classRegister)
```

```
classRegister.sort(key=itemgetter(1))
```

```
print (classRegister)
```

Just as we passed the "reverse=True" to the sorting, we can pass the "key=" and tell it to get the second item from each inner list!

The value passed to "itemgetter" is a valid index 0 to "len" minus one of the student list within the register.

We can combine this name sorting with a reverse alphabetical order...

```
classRegister.sort(key=itemgetter(1), reverse=True)

print (classRegister)
```

You can even use the same parameter passing to create pre-sorted lists with the "sorted" function.

6.11 Tuples

Tuples, like lists before them, are a structure which lists a series of fields one after another, they are created not with the "[]" brackets we've see with lists but with "()", like this:

```
studentA = ( 4, "Dave", "Dancer", 2016 )
print (studentA)
```

The difference to a list is you can not edit a tuple, so you can not sort it, reverse it or alter the field values once it has been created.

```
studentA[1] = "David"
```

Running this code will result in an "TypeError" with the message "'tuple' does not support item assignment". Whilst trying to sort a tuple will error with "AttributeError: 'tuple' object has no attribute 'sort'".

Not being able to edit the values is important for use when you want to ensure no code, maybe code you've not written cannot alter the stored values.

6.12 Sets & Dictionaries

We are not going to cover the specifics of Dictionaries or Sets within this tutorial, they are a much deeper topic than is intended for this text. However, it should be noted they exist, and you can read about them through the official Python documentation. Though it should be kept in mind that mastering lists first is key skill.

Chapter 7: Classes

A class, also known as an object, is a piece of code which we can use to represent all the actions and details which relate to one thing, be that a representation of a real item, such as a shape or a vehicle... Or something imaginary, like a monster in a game or a mathematical construct.

The benefit of combining everything related to an object into a class is the simplicity of maintenance it lends to your project, you can come back to your code easily and know where the code is, you can model how it is to interact before you write the code and once written can more easily be traced through to discover and eliminate bugs.

We have already used some objects, the "list" was an object, the string we used however was not, the distinction being that the list contained functions such as "sort" and "reverse", where we called them within the class... "list.sort()" for example.

This packing together data of functionality & data into a class is known as encapsulation.

7.1 Naming Conventions

The rules for naming a class are that its name can only start with a letter, it may then contain letters or numbers, it may contain underscores but not contain other punctuation. This is the physical rules in regards class naming, far more important is your putting meaning into names.

A name should tell the reader what your intention is, what the class represents, the name "A7433_error" tells the reader very little, whilst the name "Error_A7433_OutOfSpace" conveys both the technical and informal meaning of the class.

7.2 Our First Classes

Our first class will just be called "Message" and we will add one function to it called "Hello"....

```
class Message:
    def Hello(self):
        print("Hello from inside!")
```

To use this class, we might create a variable called "ourClass", like so...

```
ourClass = Message()
```

And we can use the variable like so...

```
ourClass.Hello()
```

These last three sections of code show us a class which has a single function, creating a copy of the class and then using it. You can see as with previous examples we call the "Hello" function with just the variable name a full stop then the function name, it takes no parameters so the brackets are empty when we call the function. However, they are not empty when we define it, they contain this mysterious keyword "self".

"self" is the class itself, they tell Python this function is called upon a copy of the class the function is within, and Python knows this function is within the "Message" class because of the indentation.

This "Message" class has no data within it, it only contains a function, if we needed a class to contain some data then we need to create it with a special function called the "init" function, or constructor... It looks like this....

```
class Message:
    def __init__(self, inputMessage):
        self.textToPrint = inputMessage

    def Hello(self):
        print (self.textToPrint)

ourClass = Message("Hello World!")

ourClass.Hello()
```

We can see our "init" function it leads in and out with two underscore characters, it we add spaces "__init__" though the spaces are not entered with the code. This function takes the "self" but also a second parameter called "inputMessage". The two parameters are separated by a comma. The function within, like all other functions has it's definition ended with a colon.

On the next line we see indentation for both the class and then the function, and this strange assignment to "self.textToPrint" of the "inputMessage"... This is an internal or member variable called "textToPrint" being created and set!

In all places through the next function called "Hello" we see this variable pointed to with "self." telling Python to use the name "textToPrint" from within the class.

Very much in the same manner with globals, however, now we are telling Python to look inside the class.

From outside the class we can access "textToPrint" directly, like so...

```
print (ourClass.textToPrint)
```

Or...

```
ourClass.textToPrint = "Something Else!"
```

We could also set this text within the class, without the need for a parameter within "init" ... like so...

```
class Message:
    textToPrint = "Hello Another World!"

    def Hello (self):
        print (self.textToPrint)

ourClass = Message()

ourClass.Hello()
```

This has been an intensive section, and just a small part of using classes within Python, take sometime to go over these examples, try yourself to create a class with two functions within it... Once you are happy, only then continue.

7.3 Structure & Best Practices with Classes

We have seen the use of functions in the form of the "init" function and others, let us now look at a class with several functions... And expand upon the use of comments to help us read our code... Remember comments are lines starting with the hash '#' symbol and are ignored by Python, intended for us to leave comments for ourselves or other readers of our code.

```
class Student:
    #----- Member Variables -----
    DaysAttended = 0

    #----- CONSTRUCTOR FUNCTION -----
    def __init__(self, ParamName):
        self.Name = ParamName
        self.DaysAttended = 0

    #----- Present function -----
    def Present(self):
        self.DaysAttended = self.DaysAttended + 1

    #----- End Student Class -----
```

```
GoodBoy = Student("John")

print (GoodBoy.Name + " present for " + str(GoodBoy.DaysAttended))
GoodBoy.Present()

print (GoodBoy.Name + " now at " + str(GoodBoy.DaysAttended))
```

Some may argue to you that comments like this, naming the function they sit along side or mark the end of pointless, and for remembering the name they are, but for understanding the code quickly, finding where certain stanza's of code begin and end they prove themselves invaluable.

7.4 Multiple Members, Parameters & Functions

We'll continue with less explanation, the following class will represent a shape...

```
class Square:
    #----- Constructor -----
    def __init__(self, pWidth, pHeight):
        self.Width = pWidth
        self.Height = pHeight

    #----- Functions -----
    def Area(self):
        return self.Width * self.height

    #----- Print Self -----
    def Print(self):
        print ("Square [" + str(self.Width) + ", " + str(self.Height) + "]")

#----- End Square Function -----

mySquare = Square(12, 30)
area = mySquare.Area()
print ("The area is " + str(area))

mySquare.Print()
```

Two members have been added to this class, two functions, the constructor takes two parameters, you may expand upon this example yourself.

The special feature of this class however is that no-where in it's definition do we state that "pWidth" or "pHeight" and by association "self.Width" and "self.Height" have to be integers,

they could easily be used as floating point numbers...

```
floatSquare = Square(3.45, 9.87)
newArea = floatSquare.Area()
print ("The Area is " + str(newArea))
```

This is one of the powerful features of Python, it mutates your code as you change your input parameters, without the need to change the code as it expressed.

Of course if the parameters passed in were not compatible with any functionality within your class, the code would cause an error... For example, if we pass strings into the class...

```
myTextSquare = Square("alpha", "beta")
```

This is perfectly valid, one only assigns the two input strings to the internal member strings... However, when one calls the "Area" function...

```
print (myTextSquare.Area())
```

We receive a "TypeError", with the message "can't multiply sequence...". This is an example of how to run code in order to test the functionality, one part of the code was valid whilst the function invalidated the whole class. The solution? We could use a "try-except" statement and the "int" or "float" functions in order to ensure "pWidth" and "pHeight" are forced to be numbers!

7.5 Classes Containing Lists

We are able to add lists inside our classes, as we have seen in the previous example strings are just lists of characters... But more difficult for some to grasp is that we can add instances of classes into lists... A good example might be when we have a Student class, and need a register...

```
class Student:
    def __init__(self, pName)
        self.Name = pName

classRegister = [ Student("John"), Student("Paul") ]

print ("There are " + str(len(classRegister)) + " students")

print (classRegister[0].Name)
print (classRegister[1].Name)
```

The access to the list with the "[]" operator is made to select which student, then you can

access the members and functions with the "." operator.

As an example of adding lists as members to classes...

```
class DaysOfTheWeek:
    def __init__(self):
        self.DayStrings = [ "Sunday", "Monday", "Tuesday", "Wednesday", "Thursday",
"Friday", "Saturday" ]
        self.WeekendDays = [ 0, 6 ]

    def IsWeekendIndex(self, DayIndex):
        return (DayIndex in self.WeekendDays)

    def MaxDayIndex(self):
        return len(self.DayStrings)
```

This class has three functions, the initialiser and then "IsWeekendIndex" and "MaxDayIndex"... We could use it like this...

```
dow = DaysOfTheWeek()

for day in dow.DayStrings:
    print (day)

for i in range(dow.MaxDayIndex()):
    if dow.IsWeekendIndex(i):
        print (dow.DayStrings[i] + " is a weekend day!")
```

Chapter 8: Importing & Libraries

We have touched upon this topic before, but now we're going to learn how to include code from the vast collection of libraries within Python!

A library is simply a set of code, classes or functionality, which perform particular jobs, for example there is a networking library, there is a library for handling data structures and we've seen there are libraries to extend functions to other classes in the language.

We will not be going over a comprehensive list of libraries, there are simply too many, instead we are going to learn how to import and use a library; once complete you should take sometime to look through the libraries available to you for certain tasks.

8.1 Importing

We have seen ourselves import once before, bringing a function into use for our list sorting... However, now we need to show how to import between one file and another, Appendix 2 covers how to run python scripts you write from a file, rather than within the interpreter (as per all the screen shots we've seen so far).

Lets start with a file called "one.py"...

```
class Alpha:
    def Display(self):
        print ("This is Alpha")

class Beta:
    def __init__(self, p_Message):
        self.m_Message = p_Message

    def Display(self):
        print ("Message from Beta is '" + self.m_Message + "'")
```

We've written two classes called "Alpha" and "Beta" and saved this file.

In the same directory as we save that file, we create a new script file called "two.py" and enter this code...

```
from one import *

print ("This is the main program code")

a = Alpha()
b = Beta()
```

```
a.Display()
```

We can see we bring all the code from "one.py" into "two.py" and we can run the classes and code from the first file, within the second. You can only import in this way with your own "py" files when you are working in the same folder.

There are ways to tell python to load files from other locations, indeed the standard locations, however, we are working on our own user code from one directory.

In this last example, we could just import "Alpha"... Like so...

```
import Alpha from one

print ("This is the main program code")

a = Alpha()

a.Display()
```

As we only imported "Alpha", we can not use "Beta".

Breaking a project down into separate classes was one step along the journey to efficient development, working in different files is another. Dividing tasks between members of a team, or just yourself, over time it made simple with importing between files in this way.

8.2 Import Errors

If we attempted to import from a library we can't be found, or asked for a specific function (or class) which was missing; or we simply made a typo; imports throw a kind of error, known as an "ImportError".

We have seen errors previously, and will deal with how we should handle errors in the very next chapter. However, for now you should beware that if you make a mistake with an import you are notified immediately.

This is different to regular code in Python, where you may not be informed of the error you have made until you actually run through the code with the mistake. The reason being that imports are carried out from all files immediately that your script is loaded into memory, nothing has run yet, and so "ImportErrors" maybe the first error you ever encounter in a file.

8.3 Useful Library Examples

Python's standard library of functionality is far too big for us to cover, and the extended libraries beyond that are even larger, however we will now cherry pick a few useful parts and

explain what they do.

8.3.1 Queues

When we covered lists we talked about using a list as a queue, with some creative use of the reverse function, of course there is a better and built in way to use a queue.

These sort of classes reside in the "collections" module of Python, and we can import the class "deque"...

```
from collections import deque

queue = deque()

queue.append("A")
queue.append("B")
queue.append("C")

print ("Queue Loaded with...")
print (queue)

while len(queue) > 0:
    item = queue.popleft()
    print (item)

print ("Queue is empty...")
```

In the code we can see the variable "queue" is an instance of the "deque" class we imported, it is empty, and we append a series of items onto the end of the queue, and we process them off of the queue with a while-loop until the queue is empty.

8.3.2 Date Time

```
from datetime import datetime

timenow = datetime.now()

print (timenow)
```

The output I just got is: "2016-11-19 14:26:58.743029". Not very humanly comprehensible, and not a string, to make this a string we can use "str"...

```
stringTimeNow = str(timenow)
```

It's still not very usable by a human, yet, I'm in the UK so I like to see my dates as "days" then "months" then years... Lets code that up, using a mask string we call two things, the "now" function of "datetime" to get the actual time, and then we call a function on that datetime object itself to format the date only out of it...

```
dateNow = datetime.now().strftime("%d/%m/%y")
print (dateNow)
```

My output is now "19/11/16", you can check out the Python documentation for more examples of formatting the date and time in this manner!

8.3.3 Platform

The platform library allows you to retrieve useful information about the system your Python program is running on, including what operating system, which version of Python and what processor the machine has. This can be very useful in dealing with the differences between Python2 and Python3; which we've already covered.

In Linux (or other Unix like systems) it is easy to define which version of Python to use when we start the interpreter, with Python3...

```
xelous@xelous-PBL21:~$ python3
Python 3.5.2 (default, Jul  5 2016, 12:43:10)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> x = input()
Helo World
>>> print(x)
Helo World
>>> quit()
```

Working and using "input()" without a problem, but with Python2...

```
xelous@xelous-PBL21:~$ python2
Python 2.7.12 (default, Jul  1 2016, 15:12:24)
[GCC 5.4.0 20160609] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> x = input()
Hello World
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    File "<string>", line 1
      Hello World
      ^
SyntaxError: unexpected EOF while parsing
>>> quit()
```

Failing to work, we recommended earlier in the tutorial to swap to the call "raw_input" for Python2 users, this is because on systems like windows you will have only one version of Python installed, not both.

However, with the "platform" library, we can work around these problems without ever needing to worry the end user of our program. Lets start by defining a function to get an input message... We know we need to handle a different input function for python 2 than python 3...

```
def InputMessage():  
    if python2:  
        theMessage = raw_input()  
    else:  
        theMessage = raw_input()
```

We get the value for "python2" here from the platform library thus:

```
import platform  
  
pythonVersion = platform.python_version_tuple()  
python2 = (pythonVersion[0] == "2")  
  
def InputMessage():  
    global python2  
    if python2:  
        theMessage = raw_input()  
    else:  
        theMessage = raw_input()  
  
x = InputMessage()
```

Lets go over this code, first we import the platform library, then we have a global variable which is called "pythonVersion" which we assign the value of something called the "python_version_tuple". There are very many different functions inside platform to use, this one however gives us the tuple of strings for the version of Python and so we can access them with the "[]" operator.

The first element of the version tuple "[0]" is therefore the major version of Python in use, so it will be "3" or "2". We use a logical "==" check to get a "True" or "False" for this being "2" and assign that boolean to the global variable called "python2".

In any function in our program now we can tell the function to use the global called "python2" and use the flag we've set. You can see this inside the "InputMessage" function later on.

The very last line of our example program show us seamlessly calling the function, and it now makes no difference to our code which version of python is called, it handles both and the end user never knows there was a difference.

You could use the value directly as well, like this...

```
import platform  
  
def InputMessage():
```

```
if platform.python_version_tuple()[0] == "2":
    theMessage = raw_input()
else:
    theMessage = input()

x = InputMessage()
```

I consider this to be slower, evaluating a string, and quite verbose in what it is doing. Evaluating a value which is already a boolean is slightly quicker, and something which you can log or record as your program starts. It is also easier to remember and write the use of one variable, like "python2" above, rather than "platform.python_version_tuple()[0] == '2'" at every place. This is especially true if you think about the version of python you want to target being "3" in the future, do you want to go through the risk of editing potentially hundreds of lines of code through our many files, introducing bugs or missing one or two causing havoc?... or do you just want to change one line in one known location?... Yes, I'd prefer the latter too!

Another common use of the platform library is to determine whether you are on Windows or not, in order to use the folder structures correctly (Windows and Unix like file systems and paths are different). For example, you want to ensure you create a "temp" folder when you start your program, this might be "C:\temp" on windows, but "/temp" on Linux... The Platform library can help us tell which...

```
import platform

IsWindows=False

print ("Operating System is: " + platform.system())
if platform.system() == "Windows":
    IsWindows = True
```

We now have a global called "IsWindows" which will be "True" whenever we are on the Windows Operating System, otherwise its false...

```
def GetPathToLogFile():    global IsWindows
    if IsWindows:
        return "C:\temp"
    else:
        return "/temp"
```

You can read a whole lot more about how the [Platform library works within the python documents](#).

Chapter 9: Handling Errors

Whilst we have learned out Python code we have learned about the problems, in the form of errors, our code can run into. We've seen name errors, range errors, import errors and a few others.

All these errors are known as exceptions, and they can stop your code running on the spot. Exceptions are a variable, created by Python for us, and we can catch them when they are raised by the language, so that our code keeps executing rather than stopping with a horrible error.

Lets jump back to the very first problem we saw in our learning of chapter 1, the case where we try to add a string and an integer during a call to "print", which results in a type error...

```
print ("Hello World " + 42)
```

This code raises a "TypeError"... Some languages call this "throwing an exception", because you can raise your own errors with the "throw" keyword, however in Python the keyword is "raise", therefore we will call this "raising exceptions".

TypeError: cannot concatenate 'str' and 'int' objects

In order to handle these errors we have two keywords to learn... "try" and "except"... Many other languages call this a "try-catch" block, as they use those as keywords, Python has "except" rather than catch... In action, they look like this...

```
print ("This is the start of the program")

try:
    print ("Hello World " + 42)
except:
    print ("I had an error!")

print ("This is the regular code again")
```

You will notice the "try" statement, and it is a statement as it ends with a colon, begins the block, everything inside the block is indented and runs UNTIL the error is raised, the moment, the very line the error occurs the code jumps down to the "except" which is a statement; not indented; with its own lines of code to report the error.

Lets take a look at a useful example of this in action...

```
print ("Enter a whole number: ")
while True:
```

```
text = input()
try:
    value = int(text)
    break
except:
    print ("You made a mistake, and failed to enter a number")
print ("Program is Complete...!")
```

Take a moment to look at this code, we ask the user to enter a value, then enter a loop which can never exit on its own. We call to get the input text, then try to convert that into an integer... At this point if the text is not a valid integer number an error will jump to the "except" block and print a message... If the integer was valid then the next line is "break" which exits the loop... We just validated a number input!

9.1 Specific Errors

When we just use the "except" on its own we will receive and handle every single exception of every single type... Generally this is a bad idea, what if we have code which could raise two different types of error... We need to handle the different errors.

```
numbers = [ 1, 2, 3 ]

while True:
    print ("Enter which number by index [0 - 2]: ")
    try:
        text = input()
        index = int(text)
        number = numbers[index]
        print ("You selected Index [" + text + "]")
        print ("it has value [" + str(number) + "]")
        break
    except TypeError as te:
        print ("You didn't enter a number!")
    except:
        print ("We had some other error!")
```

So if you entered "abc" as your selection, the "int" call would fail raising a "TypeError" and we capture it in the first "except". You can see we use "as" to give this error a name of "te"... And you could use "print (te)" to show the raw error message.

However, the other potential error is an "IndexError" if you enter a valid number but one which is negative or larger than 2... You are out of range of the "numbers" list.

This second error is not a TypeError, so the first except is skipped, the second except handles

that error... Because it is still handling ALL errors, we could alter the code again to handle the "IndexError" specifically.

```
numbers = [ 1, 2, 3 ]

while True:
    print ("Enter which number by index [0 - 2]: ")
    try:
        text = input()
        index = int(text)
        number = numbers[index]
        print ("You selected Index [" + text + "]")
        print ("it has value [" + str(number) + "]")
        break
    except TypeError as te:
        print ("You didn't enter a number!")
    except IndexError as ie:
        print ("You didn't select an index in the right range!")
```

We now handle the two errors we know about, but any other error will go up and leave our code, potentially stopping the program running!

We could add the except back on the end to catch all other possible errors...

```
numbers = [ 1, 2, 3 ]

while True:
    print ("Enter which number by index [0 - 2]: ")
    try:
        text = input()
        index = int(text)
        number = numbers[index]
        print ("You selected Index [" + text + "]")
        print ("it has value [" + str(number) + "]")
        break
    except TypeError as te:
        print ("You didn't enter a number!")
    except IndexError as ie:
        print ("You didn't select an index in the right range!")
    except:
        print ("Something else went wrong")
```

There are a huge number of different exceptions which can be thrown by your application, you should beware of them and handle errors as you discover them, but once you transition to

production quality code a visit to the python documentation for a list (see [Python Docs Section 5.](#))

9.2 Finally

No, we don't mean this is the end of the tutorial... "finally" is a keyword, just like "try" and "except", but it is used to clean up after exceptions, you can "try" one block of code, catch problems at the "except" block and clean up after the failing code within the following "finally" block.

```
print ("This is the start of the program")

try:
    print ("Entered Try-Except block")
    print ("Hello World " + 42)
    print ("We never see this code")
except:
    print ("An Error Happened.. somewhere... in the try-except block!")
finally:
    print ("Clean up to Aisle 12!")

print ("This is the regular code again")
```

When you run this code you will see the start of the "try" block, then it jumps to handle the error within "except" and then performs the block within the "finally". This is of particular use when working with shared system resources, such as files.

If you opened a file, but encountered an error whilst using it, a "finally" block would allow you to always ensure you close the file. We will talk more about files in the next chapter.

9.3 Raising an Exception

All exceptions are based off of a class called "BaseException", you can derive from this to make your own exceptions, this is beyond the scope of this tutorial. But we need to learn how to raise exceptions... Like this:

```
try:
    print ("Hello")
    raise BaseException("Whoop Whoop Error Here!")

except BaseException as Ex:
    print ("Exception Occurred: ")
    print (Ex)
```

```
print ("Carry on, Carry on! Nothing to see here")
```

You can see the new keyword "raise" and the type of exception we want to send out, the "BaseException" accepts a string of text, which we see when we "print" the error out later.

We can raise an exception from any location in our code, we do not need to be in a "try" block at all, so this code would be perfectly valid...

```
def Limited (p_First, p_Second):  
    if ( p_First > p_Second):  
        raise BaseException("BEYOND LIMIT")
```

But we would need to be in a "try-except" block in order to handle the error...

```
try:  
    print ("Step A")  
    Limited(1,2)  
    print ("Step B")  
    Limited(2,1)  
except BaseException as Ex:  
    print ("Limit Exception!")  
    print (Ex)
```

9.4 Import Errors

In chapter 8 we covered the idea of importing libraries, and elsewhere in this tutorial we have talked about differences between Python2 and Python3, one area that you might need to handle this is when importing.

Lets look at an example, with the "HTMLParser" class, this class is used to turn the HTML you see for display in a browser into or out of the HTTP format we transmit over the internet, this translation from one form to the other is called parsing (go with me here).

In Python3 we import the HTTPParser with:

```
import html.parser
```

However, if you tried this import on python2 you would get an error, an "ImportError" to be precise... But we can handle these errors as they are exceptions... The exact code to handle this for the difference between python 2 and 3 is...

```
try:  
    import html.parser  
    print ("We are in Python 3")
```

```
except ImportError:
    import HTMLParser
    print ("We are in Python 2")
```

Of course in this code we then need to handle the difference within our code, so we might set a flag to we know which of the two blocks we imported with, and use the correct syntax for python 2 or 3, and the same script written for one version of the language will work on the other.

```
g_UsingPython3 = True

try:
    import html.parser
    print ("We are in Python 3")
except ImportError:
    g_UsingPython3 = False
    import HTMLParser
    print ("We are in Python 2")

def HTMLStringDecode(p_HTTPString):
    global g_UsingPython3
    l_parser = None
    if g_UsingPython3:
        l_parser = html.parser.HTMLParser()
    else:
        l_parser = HTMLParser.HTMLParser()
    l_HTMLString = l_parser.unescape(p_HTTPString)
    return l_HTMLString

l_Source = "<HTML><BODY><H1>Hello
World</H1></BODY></HTML>"

l_Destination = HTMLStringDecode(l_Source)

print (l_Destination)
```

9.5 System Exit & Quit

When running your script if you encounter a critical error, and wish stop your program you can do one of possible two things. The simplest is to call the function "quit()". You can do this yourself within the interpreter or on any line of your script, and Python will stop running and completely exit.

```
print ("Hello From A")
quit()
```

```
print ("Hello From B")
```

The other is to "raise" an error of the "SystemExit" type. For instance when we might want to have a log file

```
logFile = None
try:
    logFile = open("/var/log/mylog.log", "w")
    print ("All our other code runs here")
except:
    raise SystemError("Unable to open log!")
finally:
    if not logFile == None:
        logFile.close()
```

Opening our log is a requirement of this code, without it, we must stop... And we also use a finally to ensure the log file is closed before we leave all our other code.

A simpler example might be...

```
print ("Hello World")
raise SystemError("Good bye")
```

Chapter 10: Basic File Handling

The ability to load information into a program and store the results is not limited to just the keyboard and screen. We are far more likely to read data from and write data to a file on disk. To do this there are several different support libraries as well as built in functions within python.

10.1 Reading Text Files

The easiest way to open a file is to use the keyword "open"...

```
x = open ("somefile.txt", "r")
```

This tries to open a file with the name "somefile.txt" (in the same directory as your script), if the file is present it opens it in read mode; denoted by the "r". Once open in this manner we can read all the lines of text in the file like this...

```
x = open ("somefile", "r")  
  
for line in x:  
    print (line)  
x.close()
```

We are using a "for" along with the "in" keywords once again, and by default a text-file is not broken down into individual words or characters, but into lines of text.

We can of course also read letter by letter, with the "read" function...

```
x = open ("somefile", "r")  
  
character = x.read(1)  
  
print ("The first character in the file is " + x)  
x.close()
```

Or you could read a certain number of bytes at a time...

```
x = open ("somefile", "r")  
  
character = x.read(5)  
  
print ("The first character in the file is " + x)  
x.close()
```


If we try to read more bytes in this last call than remain available in the file, then we will get an "IOError", which we would need to handle with a "try-except" block, but perhaps also a "finally" to ensure the file is closed.

All of the data being read is in "text" format by default, so strings and characters. If you want to convert the data into numbers you must use the "int" and "float" commands which we've already seen in use.

10.2 With & Files

Another keyword "with" is invaluable when working with files. It works on an item you provide within the following block, but ensures it is cleaned up with once that block completes...

```
with open ("somefile", "r") as theFile:  
    for line in theFile:  
        print(line)
```

Here we see the "open" command being given the name "theFile", within the block we can use this name and know that as the block ends the file is closed automatically.

Except when there is an error, is an error is raised within the block the code jumps out of the "with" block completely, so you are best to practice wrapping the whole segment into a "try" block, like so...

```
try:  
    with open ("somefile", "r") as theFile:  
        for line in theFile:  
            print(line)  
except:  
    print ("Error in the file handling")
```

10.3 For-Else

Another structure we have not looked at, which indeed is often not used by even experienced Python programmers, but which is brilliant with files is the "for-else" statement, for example...

```
with open ("somefile", "r") as theFile:  
    for line in theFile:  
        print(line)  
else:  
    print ("The file has ended!")
```

10.4 Handling End of File

The "read" function indicates the end of the file by returning an empty string, lets read every character from a file and see this in action...

```
x = open("somefile", "r")
done = False
while not Done:
    char = x.read(1)
    if len(char) == 0:
        print ("End of File")
        done = True
    else:
        print ("Data: " + char)
x.close()
```

We've already seen that the file, when read with a "for" ends after the last line read:

```
x = open ("somefile", "r")

for line in x:
    print (line)
x.close()
```

10.5 Writing Text Files

So far we've seen our calls to "open" just the "r" for read flag, in order to write we need to use the "w" flag...

```
outFile = open("somefile", "w")
```

Just as we used "read" we can use "write"...

```
outFile.write("Hello World")
```

Or we can write some variable, so long as we ensure we make them strings; just we did when using "print"...

```
a = 100
b = "Hello World"
outFile.write(str(a))
outFile.write(b)
```

When we call "write" we get a value back, this is a number, the number of bytes written to the file...

```
data = "Hello"

outFile = open("somefile", "w")

bytesWritten = outFile.write(data)

if bytesWritten == len(data):
    print("Data Written")
else:
    print("Error, not enough data written")
    print("Disk may be full!")
```

10.6 Binary

Binary files are beyond the scope of this tutorial, however you can open for read or write with the mode "rb" or "wb", however, all data going into or out of the files must be of the types "bytearray" or "bytes"... More information, as ever, can be found in the [Python Docs](#).

10.7 Read, Write and Seek

Another file open mode we can use is "rw", which means both read and write, like this...

```
file = open("somefile", "rw")
```

Once either of these are open, calls to "read" and "write" are valid for both...

If we take this further and write a value to a file, then try to read it back, lets see what happens...

```
file = open("somefile", "rw")
bytesWritten = file.write("Hello")
data = file.read(bytesWritten)
print(data)
```

The result?... A blank line, no data.. nothing... What happened?...

Well, files work on a pointer, which indicates where we've gotten to through the file, when we open the file this pointer is at position zero... When we then write the data "Hello" it moves to position 5, as indicated by the number of bytes written. So when we then read, we get nothing, because we're already beyond the end of the data in the file.

One bad solution might be to close the file and re-open it...

```
file = open("somefile", "rw")
bytesWritten = file.write("Hello")
file.close()
file = open("somefile", "rw")
data = file.read(bytesWritten)
print (data)
```

Though this works, it is very slow and you may as well just open the file with "w" then reopen it with "r", so you're not leveraging being able to read and write with the single file open call! One solution would be to just move the file pointer back to the beginning of the data we wanted... Older reader might want to think about this like rewinding the a tape!

```
file = open("somefile", "rw")
bytesWritten = file.write("Hello")
file.seek(0)
data = file.read(bytesWritten)
print (data)
file.close()
```

The "seek" function here moves the file pointer to the given position in the file, zero here being the start of the file!

Given the file with just the word "Hello" we could seek to only get "lo" like this...

```
file = open("somefile", "r")
file.seek(3)
data = file.read(2)
print (data)
file.close()
```

This therefore opens the file, for read, then seeks to position 3 and reads two bytes... Printing out "lo"!

Appendices

Appendix 1: Installing & Using Python

Python is an interpreted language, this means each instruction is inspected by the Python application itself and acted upon. This is different to some other languages which are turned from the code instructions you enter and can read into instructions which really only the machine itself understands, these are sometimes called "compiled" languages.

Compiled languages are somewhat quicker than interpreted languages, whilst interpreted languages are more portable between different operating systems and indeed whole machines.

This is skipping over a lot of detail, however you should really just understand Python is interpreted.

When you run the Python interpreter itself you will be presented with a text based console interface, on Windows and Mac this will of course be within a window itself. The key interaction with the interpreter are three right chevrons ">>>".

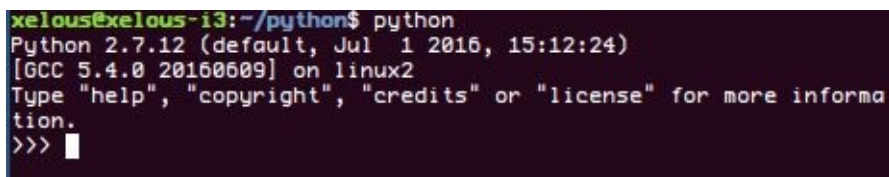
These chevrons indicate Python is ready and waiting for a new line of code to be entered, you can type the command and then press the "ENTER" key it execute the command.

Once you have gotten used to running code line by line you can then use any text editor... notepad on windows for example... To write your code into a text file, then you can call the python interpreter and pass it the file for running as a complete program.

Before we run Python we need to make sure we have it installed...

Appendix 1.1: Linux

Most Linux distributions come with Python installed, simply typing "python" at the command-line will bring the interpreter up:

A terminal window with a dark background. The prompt is 'xelous@xelous-i3:~/python\$'. The user has entered 'python'. The output shows 'Python 2.7.12 (default, Jul 1 2016, 15:12:24)' and '[GCC 5.4.0 20160609] on linux2'. It then says 'Type "help", "copyright", "credits" or "license" for more information.' and ends with the prompt '>>>' followed by a cursor.

```
xelous@xelous-i3:~/python$ python
Python 2.7.12 (default, Jul 1 2016, 15:12:24)
[GCC 5.4.0 20160609] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> █
```

This particular machine has "Python version 2.7.15", take care which version of Python you have, because the examples in this book target Python Version 3.

Python 3 has a different executable name under Linux "python3"...

```
xelous@xelous-i3:~/python$ python
Python 2.7.12 (default, Jul 1 2016, 15:12:24)
[GCC 5.4.0 20160609] on linux2
Type "help", "copyright", "credits" or "license" for more informa
tion.
>>> quit()
xelous@xelous-i3:~/python$ python3
Python 3.5.2 (default, Sep 10 2016, 08:21:44)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more informa
tion.
>>> █
```

Installing on Linux is simple through your package manager; for Debian based machines (as a sudoer) you use the line:

```
sudo apt-get install python3
```

Or for distributions using "yum", you use:

```
su root
yum install python3
```

If you have a visual package manager, or software installation centre, feel free to use this for your installation.

Appendix 1.2 Windows

From www.python.org you can download the Python 3 installer for Windows, you will require Administrator user access to start the installation. At the time of writing the latest version for windows is 3.5.4.

The installer has one key page, shown below...



We have added a red arrow to indicate where the installer wants to place the program... A green arrow where you can change your installation, we'd recommend you stick to the default options.

Then the most important item is indicated with a purple arrow, "Add Python 3.5 to PATH", this should be ticked ON.

PATH is an environment variable, part of all of Windows, once your Python installation location is added to the PATH whenever you open a new command prompt you can simply type "Python" to start the interpreter... Without this added to PATH you will need to add it yourself (from the location indicated with the red arrow) each time you open a new command prompt.

Other than the command prompt you can run the interpreter from a short cut installed for you, when running on Windows, Python appears like this...

Python 3.5 (32-bit)

```
Python 3.5.1 (v3.5.1:37a07cee5969, Dec 6 2015, 01:38:48) [MSC v.1900 32 bit <Intel>] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> _
```

Appendix 2. Using Python

With our interpreter installed we can run code in one of two ways, the first is to start the interpreter and when presented with the three chevron prompt ">>>" you can start to enter single lines of code at a time... It is recommended you cover the first few chapters in this manner, by chapter 8 you will have to enter code in a more advanced way.

This second way to run code is to type all your lines of code into a text file. The file can be called anything, however by convention we complete Python code files with the ".py" extension.

Once saved you can pass the whole file to Python in one call. Lets say you have saved your code as "one.py", you would pass this file to interpreter like so...

```
python3 one.py
```

This is with the prompt you are using sat at the directory with the script in it... If your prompt were just open, and your code file in a folder called "Scripts", then the command might look like this on Windows...

```
python3 c:\Scripts\one.py
```

Or on *nix style systems...

```
python3 /Scripts/one.py
```

Remember from Appendix 1, we are targeting our use to version 3 of python, hence we use "python3" as our command line call.