An Introduction to Programming With Python

Josh Borrow

August 14, 2014

1 Introduction

This little book should be your guide during this course. We will begin with our first program - a simple 'hello world'. This program will aim to teach you how to output text to the terminal. Hopefully, you have already read the previous section on setting up python on your system, and now when you type 'python' in the terminal you are presented with three greater than signs, like this:

```
>>>
```

1.1 Strings and Variables

Now, the first thing that we want to be able to do in a programming language is declare variables (well, that's up for debate, but for me it is). We declare a variable in python by using the '=' character. This means something different than when it is used in mathematics - it is an *assignment* operator rather than an *equality*.

Because python is a very 'loosely typed' language, this means we can set our variables to any type - be it an integer, a floating point number (decimal) or a 'string'. A 'string' is a set of characters, such as 'Hello World'. The characters are enclosed in a set of quotes or inverted commas.

So if I want to give a variable called 'hi' a value of "Hello World!":

```
>>> hi = 'Hello World!'
```

The next thing we want to be able to do is print some output to the screen. In python, we do this by using 'print'. In python 2.x, this is a pseudofunction, meaning it doesn't need brackets - we will learn more about this later. To print the value of hi to the screen, we simply need to write:

```
>>> print hi
'Hello World!'
```

1.2 Types of Variable

Now we've got you hooked after writing your first computer program, we have to cover some boring stuff. If you want to, you can skip this section and come back if some of the words I start to use in the later sections confuse you. We'll start by talking about the four basic variable types.

1.2.1 Integers

Integers are exactly what you would expect them to be. They can have almost any value you want to give them. However, the interesting thing here is division.

When you divide two integers in python, it returns an integer too. This is accomplished by simply discarding the remainder. For example:

```
>>> 5/2
2
>>> 20/4
5
>>> 121/120
1
```

This process is called 'integer division', rather than decimal or 'float division'.

Integer division can cause some problems, and was changed in python 3 (however we're using python 2) so that the division automatically converts the values to decimals.

To get the remainder from this division, we use the 'modulo' operator. This operator is represented by the percentage sign, %. For example:

```
>>> 5%2
1
>>> 20%4
0
>>> 121%120
1
```

1.2.2 Floating-point Numbers

Floating point numbers are what mathematicians call 'real' numbers. They can have almost any decimal value, and their division works in the normal way.

You can force a variable to be 'cast' as a float rather than as a integer by simply putting a full stop after it, or by casting it through the function 'float()':

```
>>> a = 2
>>> b = 3.
>>> c = float(4)
>>> d = 1035.2342345234
>>> type(a)
<type 'int'>
>>> type(b)
<type 'float'>
>>> type(c)
<type 'float'>
>>> type(d)
<type 'float'>
```

1.2.3 Strings

Strings, as explained above, are a 'string' of characters. They can be used to put 'words' into your program, or more excitingly, we can use the program to operate on strings. For example, we can go through each character in the string, and move it up two in the alphabet to 'encrypt' it!

Examples like these will be covered later in the course - so get excited!

1.3 Lists

Lists are just that - lists of variables. We declare a list in the following way:

```
>>> list = [1,2,3,4,5,11]
```

In a list, each variable is separated by a comma. We can fill lists with any type of object, be it a list, a float, even other lists!

1.3.1 Indexing

Lists are indexed to allow us to find values in them easily. The first value in the list has an index of 0, the second 1, the third 2, and so on. Lists are also negatively indexed, meaning we can find the last object in the list with an index of -1, second last -2, and so on.

We access the values in the list by putting the index in square brackets following the list. It's best illustrated using an example.

```
>>> list[0]
1
>>> list[3]
4
>>> list[-1]
11
>>> list[-3]
4
```

Strings are also indexed. For example:

```
>>> hi = 'Hello World'
>>> hi[0]
'H'
>>> hi[-1]
'd'
```

This allows us to do interesting things, for example we can check if the extension of a file have the name of is '.png', etc.

1.4 Definitions

In this section we will define what some of the terms mean that are scattered throughout the book.

1.4.1 Iteratable

An Iteratable object is one where you can go through and use each object. They are objects that have indexes, so strings and lists are examples of iteratable objects. They allow us to write loops using their values, which we will learn about in more detail later.

1.4.2 Function

A function is a construct that is used to do something over and over again. An example of a function is the 'type()' function used above. This function takes a variable and then returns its type.

2 Control Structures

Control structures are how we get things done in programming, it is how the computer makes decisions. We can ask the computer if something is true, for example $7 > 4$, and then act on this information. First, we must introduce booleans.

2.1 Booleans

Booleans are another type of variable, that are best introduced alongside these structures. There are two possible states for a boolean variable, 'True' or 'False'. In python it is very important that the first letter is capitalized.

2.2 if Statements

These are the foundation of control structures. What we say to the computer is, if this condition is true, execute this bit of code. If it's not, then do this instead. This turns out to be a very powerful tool. Here's an example:

```
>>> x = 3
>>> if x < 4:
...     #if x is less than 4 execute this
...     print 'x is less than 4'
... else:
...     #otherwise execute this
...     print 'x is bigger than or equal to 4'
...
x is less than 4
```

There are a few new things that we have introduced here, aside from the if statement. We can add a comment in python by having a # anywhere on the line. If the interpreter finds a #, it ignores all other characters after that.

This allows us to create comments that aid people inheriting our code figure out what it means a lot more easily than having to pick through it line by line. Commenting your code is immensely important and is a good habit to get into. However, don't excessively comment your code as this will simply frustrate whoever has to read it. As you do more and more

programming you will learn what the correct amount of comments is - this is not something that is easily taught!

2.2.1 elif Statements

If you have more than one condition you would like to check, you can use elif statements. This is short for 'else if' and is executed if the first if statement is not true, but it is. For example:

```
>>> x = 5
>>> if x < 4:
...     #if x is less than 4 execute this
...     print 'x is less than 4'
... elif x < 10:
...     print 'x is less than 10, but bigger than 4'
... else:
...     #otherwise execute this
...     print ''x is bigger than or equal to 4"
...
x is less than 10, but bigger than 4
```

You can have as many elif statements as you would like, so you can check an unlimited number of statements.

3 Iterating

If you have a large set of data, and you want to perform operations on all of them, then you could do operations on each one by hand. However, this will take a very long time, and what if someone wants you to do it again, with different data?

This is where computers and programming comes in. We can deal with large volumes of information with ease and perform huge amounts of calculations in a very short time. The following statements will show you how to *iterate* over *iteratable* statements.

3.1 The while Loop

The while loop keeps on going until some condition is met. This includes the infamous 'while true' loop - which will keep on going forever until either the program crashes or the user tells it to stop.

Here's an example of a while loop:

```
>>> x = 0
>>> while(x < 3):
... print x
... x = x + 1
...
0
1
2
```

This is interesting, because if we hadn't initialised the variable 'x' as 0 to begin with, then no code inside the while loop would have been executed, as how can you check if 'something' is less than 3 if that thing is not a number?

It also showcases what we mean when we say that the '=' sign is an assignment, rather than an equality. We can - quite rightfully - say that 'x = x + 1', whereas if this was mathematics then that would be an impossible statement. What this really means is 'take x and set it equal to what it used to be, with one added'.

3.2 The for Loop

This is where iteratable structures come in (which you would know about if you had read the 'boring' bit that I told you could skip - if you're confused then now would be a good time to go back and read it). With the for loop, we can say take this structure, and one at a time give me something from it and let me do something to it. For example:

```
>>> l = ['a', 'b', '1', '2', '7']
>>> for item in l:
...     print item
...
a
b
1
2
7
```

We can even iterate through strings:

```
>>> hi = 'Hello World'
>>> for character in hi:
...     print character
...
H
e
l
l
o

W
o
r
l
d
```

This is because both strings and lists (like many other types in python) are *iteratble structures*.

3.3 A Note on Indentation

In the code examples you have seen so far, we have used strict indentation. After you have a block that requires a colon (if, else statements, while and for loops etc.) you must indent the code a block. You can do this either by using spaces or tabs - the choice is yours. You must be consistent with this though - the interpreter is very strict about this and does not like a mix of tabs and spaces.

This will initially be frustrating, but it makes it faster for the interpreter to read your code and makes your code neater, improving readability. Nobody wants to read a file where everything is not indented so you can't see which statements line up with each other - it's a mess.

4 Functions

Functions are our way of cutting down on code re-use. Let's say you want to do something several times in your code, for example add two to a number. You could just have 'x = x + 2', but that would be inefficient - what happens when you want to change it to being 'x = x + 3'? We can deal with this by using a function.

4.1 Function Structure

Functions have a name, for example 'type'. They are then followed by brackets - these must be included in *all* functions, even if you don't need to pass any arguments through them. An argument is something you 'give' to the function, and are in the brackets:

```
>>> type(10)
<type 'int'>
```

Here, we give the number '10' (an argument) to the function called 'type', which we do by enclosing it in the brackets.

We can even have functions that take more than one argument:

```
>>> range(1, 10, 2)
[1, 3, 5, 7, 9]
```

This built in function, called 'range()', takes one to three arguments. You can specify only one, and it will give you a list of that many numbers, starting at zero. For example:

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

However, when we give it three arguments, we specify start, stop and step. So the range function will give us a list of numbers starting at the first one, stopping at the second one, with a step in between of the last number. This is a very useful funciton - you will need to create many lists of numbers in your time programming and this is much faster than writing them all out yourself!

4.2 Defining Your own Functions

So, using built in functions is really useful, but something that would be even more useful is if we could write our own functions to perform our own routines.

We can do this, using the 'def' reserved word. For example, if you wanted to create a function that is called 'addtwo' with an argument of a number that we want to add two to, we would write:

```
>>> def addtwo(number):
...     return number + 2
...
>>>
```

We can now call our function on a number, and it will return two more than it! A key word that we've used here is 'return'. This is exactly like the 'print' pseudofunction, and will 'return' the argument. For example:

```
>>> x = 10
>>> newx = addtwo(x)
>>> print newx
12
```

So in the 'type()' function, we would see:

```
>>> def type(argument):
...     #some code that finds the type
...     #and stores it in a variable
...     #that is called 'typeofvar'
...     return typeofvar
...
>>>
```

5 Loading External Modules

Python can be used as a modular language, meaning that you keep different functions in different files. This allows you to use a function more than once without copying and pasting it into the new file.

Importing modules and calling functions from them is very simple. We will begin by using a module written by someone else, and then we will move to looking at ones that we have written ourselves.

5.1 Loading in a Module

There is a module called 'Numpy' that provides a lot of useful mathematical tools for python users. If we want to load the module in, we just need to add 'import numpy' at the top of our file. If we then want to use functions from it, for example the 'array()' function, we write 'numpy.array()'. For example:

```
>>> import numpy
>>> ourlist = [1,2,3,4]
>>> ourarray = numpy.array(ourlist)
>>> type(ourarray)
<type 'numpy.ndarray'>
```

This code snippet creates a list and then turns it into an array. We will learn much more about the numpy module later, in it's own section.

5.2 Writing and Using Your Own Modules

It is very simple to write your own modules. We will begin by learning to use files to store your python code.

5.2.1 Writing a python File

We have been running all of our programs interactively so far. Here, we are going to use files to run our python scripts from - which is much easier if you make a mistake (you can go back and edit it instead of having to write it all again). It's similar to moving from using a pen and paper to using a word processor.

If you haven't already chosen and downloaded an editor and become familiar with using the terminal, please go and read those sections at the beginning of the document as they will be used heavily here.

Open your text editor and write some python code! I would suggest performing a simple 'hello world' as your first program from file.

Write this:

```
hi = 'Hello World'
print hi
```

Save your file, with a '.py' extension, for example 'helloworld.py'.

Now, open your command prompt/terminal and change to the directory that the python code is stored in (cd), and type:

```
python <yourfilename>.py
```

You should now have 'Hello World' printed in your console!