Introduction to Python

Python is such a simple language to learn that we can throw away the manual and start with an example. Traditionally, the first program to write in any programming language is "Hello World".

1 "Hello World" in Python

One of the best things about Python (reminiscent of our glory days with Commodore 64 BASIC) is the ability to type a program directly into the interactive interpreter and have it run as you type. The instant feedback is one of the many reasons why Python is so great to learn. Let's start by typing our "Hello World" program into the IDLE Python shell (you type the **bold** bits and Python will do the rest):

```
Python 2.5.2 (r252:60911, Feb 21 2008, 13:11:45) [MSC v.1310 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> print "Hello World"
Hello World
>>>
```

That's it! your first Python program, now wasn't that easy?

The >>> (the *prompt*) indicates the interpreter is waiting for the next *statement*. Typing `print "Hello World"` and pressing Enter causes the interpreter to perform (we say *execute*) the statement. The result is `Hello World` *printed* on the next line. Python then waits (with the prompt) for the next statement to execute.

Hint

Mac OS X and Unix users may have different Python versions. Anything >= 2.2 is fine for the Challenge.

2 Python the Calculator

Python can function as a powerful calculator. To calculate (or *evaluate*) the number of seconds in leap year:

```
>>> (365 + 1)*24*60*60
31622400
```

The brackets mean that `365 + 1` is calculated first, just like in maths, and `*` (the asterisk) means multiplication.

It can also evaluate functions like absolute value (`abs`), and cosine (`math.cos`). Evaluating a function (with zero or more values) is called *calling* the function. A function is said to *return* the result of the calculation.

```
>>> abs(-3)
3
>>> import math
>>> math.cos(0)
1.0
```

Here `abs(-3)` returns the value `3`, which we can use in other calculations, like this:

```
>>> abs(-3)*5
15
```


Get into the habit of experimenting using the Python interpreter. It is often the fastest way of testing an idea, finding a bug or remembering something you already know. It also has a handy help function built in:

```
>>> help('math')
```

Try this to find what standard maths functions Python supports.

3 Saving Programs in IDLE

Most of the time you should save Python programs (in files ending in **.py**) to avoid retyping them every time you make a mistake or run them.

In IDLE, New Window (Ctrl-N) gives you a window to type your program into. Once you save this as **hello.py** notice that the fonts and colours change to match our examples in these notes. This is called *syntax highlighting* and indicates IDLE knows this text file is a Python program. It can now be executed using Run Module (F5).

Hint

IDLE doesn't add the **.py** extension when saving. You must manually type it on the end of the filename.

4 Python Syntax

Programming languages must be deliberately restricted to a small number of ways of saying things. Therefore, Python only recognises a few types of instructions. The complete set of recognised instructions is called the *syntax* or grammar of the language. Python is easy to learn because it has a very simple and clean syntax.

A *syntax error* indicates Python doesn't understand your statement because the grammar is incorrect – like saying *this my is cat* in English. Unfortunately, there are many ways to accidentally do this. Here are two examples:

```
>>> write 'Hi There'
SyntaxError: invalid syntax
>>> print "Hello
SyntaxError: EOL while scanning single-quoted string
>>>
```

In the first example we have accidentally replaced **print** with **write**, which Python doesn't recognise. In the second, we forgot to end "Hello" with double quotes. Try these two examples in IDLE. You will notice it attempts to highlight the error but *it doesn't always get it right* (it highlights the end single quote in the first example above). If the highlighted bit doesn't seem like an error try looking earlier in the line or on the previous line.

Python programs three different types of text for:

instructions that tell the interpreter what to do, including keywords like **print** and names like **abs**.

comments to help the programmer remember and others understand how their program works.

strings of letters, digits, punctuation and whitespace that the program manipulates e.g. **"Hello World"**.

Any text following a **#** is considered a *comment* in Python and is ignored by the interpreter, regardless of whether the **#** is at the start of the line or not. Another typical use of comments is to temporarily ignore (or *comment out*) a piece of code during development or debugging.

5 Strings

The quoted bits of text like **"Hello World"** are called *strings* (from *string of characters*). Single and double quoted (**'** or **"**) strings cannot extend over multiple lines (the syntax error above). Strings with triple quotation marks may extend over multiple lines.

The last statement below shows that you can add strings together, called *concatenation*, using the **+** *operator* just like for numbers. *Note, no space is added between the strings.*


```
print 'Hello World'      # single quoted string
print """Hello
World"""                 # multiline string
print "Hello" + "World"  # string concatenation
```

The output of this program looks like this:

```
Hello World
Hello
World
HelloWorld
```

Given that `"ab" + "ab"` is `"abab"`, what would you expect `"ab"*5` to be? What about `5*"ab"` and why?

```
>>> "ab"*5
'abababababab'
```

We can also find out *how long* a string is using the `len` function:

```
>>> len('Hello World')
11
>>> len("abcdefghijklmnopqrstuvwxyz")
26
```

We will see a lot more of what you can do with strings shortly.

6 `print` Statement

The `print` statement is used to display values to the user. `print` writes values to the terminal and then writes a *newline* (that is, shifts the cursor to the beginning of the next line).

This is quite different to the interactive interpreter displaying a value in Python format for the programmer's convenience.

```
>>> 'Hi There 1'
'Hi There 1'
>>> print 'Hi There 2'
Hi There 2
```

Notice the `print` statement does not result in any quotation marks around the output. To further test this, save the two statements into a `.py` file and run it. You will only see `Hi There 2` in the output.

7 Hello, Who are you? and `raw_input`

The previous "Hello World" program isn't very interesting because it doesn't interact with the user and so runs the same way every time. We want to accept information from the user and manipulate it in some way.

As a first step, let's ask the user for their name and then greet them.

```
name = raw_input('Enter your name? ') # prompt and read user's name
print 'Hello ' + name
```

Running this gives the following output (remember to answer the question otherwise it will just sit there):

```
Enter your name? James
Hello James
```

The simplest approach to reading a string from the user is to call the `raw_input` function. `raw_input` waits for the user to type in a string and press Enter, and then returns the string. `raw_input` may *optionally* be given a string which it prompts the user with (we used `"Enter your name? "`).

Think of the `raw_input` function like `abs`, taking some information (a prompt rather than a number) and giving back a result (the string from the user rather than the absolute value of the number).

8 Variables

Once we have retrieved the name from the user (which is returned to us in a string), we need to store it somewhere so that we can manipulate it. This "somewhere" is called a *variable*.

A variable is like a mailbox (or pigeon hole). You can store one value in the mailbox and check the contents later. Putting a new value in the mailbox *replaces* the existing value. Computer memory is really a very large collection of such boxes numbered from 0 up to a very big number. For instance, 256MB of RAM has about 67 million boxes for storing numbers.

Python (and almost all languages) make it easier for us to remember what each box holds by giving it a name, called an *identifier*. It is much easier to remember the variable with identifier **name** rather than **4168443356** is storing the name the user entered.

Python variables are created by *assignment*, using the **=** *operator*, which is the process of setting the *value* of a variable (or putting a value in the mailbox). Any other mention of the variable results in the value being retrieved.

```
>>> x = 10
>>> print x
10
>>> print x*3
30
```

Here, variable **x** is assigned the integer **10**. The third statement retrieves the value (still **10**) multiplies it by **30** and prints out the result (30).

```
>>> name = raw_input('Enter your name? ')
Enter your name? James
>>> print name
James
```

In `raw_input` example above the string returned from the `raw_input` function call is stored in the variable **name**. In the next line **name** is used to retrieve the value so it can be printed.

Hint

Be careful not to confuse assignment with equality (from maths). Always think about assignment as taking the *value calculated on the right* and placing into the *variable on the left*.

9 Choosing Identifiers

An identifier may start with either an uppercase or lowercase letter or an underscore (_). After the first character, identifiers can also contain digits but the first character must not be a digit. Neither punctuation, other symbols nor whitespace (spaces or tabs) can be included in the identifier.

Python keywords like `print` (and about thirty others) that go orange in IDLE when you type them) cannot be used as identifiers — that would get too confusing. Also, it pays not to create identifiers that are already used for built-in functions (e.g. `abs` or `len`). The built-in functions go purple in IDLE when you type them in.

Basically, you want to choose a name that describes what the identifier represents without being too long and cumbersome. We will talk more about choosing good identifiers in later weeks.

10 Numbers are not like strings!

Let's try to ask the user for two numbers and then add them together:

```
>>> a = raw_input('Enter a number: ')
Enter a number: 5
>>> b = raw_input('Enter another number: ')
Enter another number: 6
>>> a + b
'56'
```

Crazy! It doesn't do what we want at all, the answer should be **11**. The reason is that numbers and strings must be treated differently in Python, because they are different *types* of information. This is a problem here because **raw_input** always gives us back a string regardless of what the user enters.

If you want to use it as a number you must first convert it to an *integer* (a whole number) with **int**:

```
>>> s = raw_input('Please enter a number: ')
Please enter a number: 50
>>> s
'50'
>>> n = int(s)
>>> n
50
```

Note that after calling **int** the value in **n** doesn't have quotes around it, so it is no longer a string, it is an integer.

We will talk a lot more about these issues in the coming weeks. But for now, if you want to read a string use:

```
>>> s = raw_input('Enter your name ')
```

but if you are reading in a number use:

```
>>> n = int(raw_input('Enter a number '))
```

If you put **int(...)** around the **raw_input** for **a** and **b** above, it will now work the way we want.

11 Control Structures

So far the programs we have written aren't very interesting. They have done nothing more than follow a list of statements from beginning to end. Because of this, they will run in exactly the same way each time, which is not very interesting.

Most languages provide some form of *control structures* which allow the program to change the order in which the statements are executed or whether they are executed at all. These statements either make a decision or repeat a process (called a *loop*) or do both. They are called *control structures* because they control which parts of the program will get executed and in what order.

Control structures control a particular chunk of code called a *block*. *In Python, a statement is indented to indicate it belongs to a block.* The indentation must be to the same column for every statement in the block and must be more than the statement controlling the block. The block is often called the *body* of the control structure.

Hint

Every control structure in Python has a colon followed by an indented block!

12 if statements

Making yes/no and multi-choice decisions are handled by *if statements*. These decisions involve evaluating a *conditional expression*, that is, a question which can be evaluated to either true or false (yes or no). If the conditional expression is true, then the block controlled by the if statement (its body) will be executed.

The example below makes two decisions based on the value in **x** which we've set to **3**. The two questions are: is x less than or equal to 3? and is x greater than or equal to 3? If the answer is **True**, then the **print** will be run.

```
x = 3
if x <= 3:
  print "x is less than or equal to three"
if x >= 3:
  print "x is greater than three"
```

Try modifying the contents of **x** to change how the program executes and then rerun the program.

The first **if** statement controls the execution of the first **print** statement because it is indented under it.

However, the first **if** statement does not control the second **if** statement, because the second if statement is not indented to the same level as the body of the first **if** statement. Each **if** statement controls a block which is one statement long.

The possible comparisons are:

equal to	==	not equal to	!=
less than	<	greater than	>
less than or equal to	<=	greater than or equal to	>=

You can test conditionals in the interpreter:

```
>>> x = 3  # create a variable x not test for equality
>>> x == 3 # check for equality
True
>>> x < 3
False
>>> x != 5
True
```

13 Nested **if** statements

Try indenting the second **if** statement (and its **print** statement) further, like in the example below and run the program for various values for **x**.

```
x = 3
if x <= 3:
  print "x is less than or equal to three"
  if x >= 3:
    print "x is greater than or equal to three"
```

Now the first **if** statement now controls a block which is two statements (but three lines) long — the **print** statement and the **if** statement. The second **if** statement is now said to be *nested* inside the first **if** statement. Inside the first **if** statement, there is one **print** statement *and* one **if** statement.

14 **else** and **elif**

All of these examples have done something when the condition is **True**, but nothing when the condition is **False**. The **else** clause can be added to an **if** statement to execute a block when the conditional expression is **False**.

Here, either the first or second **print** statement will be executed but not both. In these kinds of cases the two **if** statements can be replaced by one **if** statement with an **else** clause.

```
x = 3
# only one or the other will get executed
if x <= 3:
  print "x is less than or equal to three"
if x > 3:
  print "x is greater than three"

# a simpler version using else
if x <= 3:
  print "x is less than or equal to three"
else:
  print "x is greater than three"
```

Apart from being neater, the `else` statement is more efficient because it does not have to perform another conditional evaluation. If the value of **x** is not less than or equal to 3, then it must be greater.

If you want to consider each case separately, we need to test more conditions, and so you will need to use an `elif` clause. `elif` is an abbreviation for `else` and `if` together:

```
# separating out the cases (less, equal or greater than 3)
if x < 3:
  print "x is less than three"
else:
  if x == 3:
    print "x is equal to three"
  else:
    print "x is greater than three"

# a less nested version using elif
if x < 3:
  print "x is less than three"
elif x == 3:
  print "x is equal to three"
else:
  print "x is greater than three"
```

The `elif` clause makes the code easier to read because it does not need to be as nested. It can be used to execute different blocks of code for any number of different options. Each conditional is evaluated in order. Only the body of the *first* conditional that evaluates to true will be executed. Where `elif` clauses appear, the `else` body will be executed only if none of the preceding `if` or `elif` conditionals have evaluated to true.

15 Floats and Integers

Using Python as a calculator can sometimes give unexpected results:

```
>>> 1/3
0
```

Huh?? Surely the answer is meant to be 1/3, that is, 0.333333... Let's try something different:

```
>>> 1/3.0
0.33333333333333331
>>> 1.0/3
0.33333333333333331
```

That's better. The problem seems to be that Python treats whole numbers (*integers*) differently to decimals (called *floats*). Like strings and integers, floats are a different *type* of value, and they have different behaviour.

In fact, most programming languages do this, because they are processed very differently by the underlying computer hardware (but that's a story for another day).

Basically, all you need to know for now is that *integer division* is different to *float division*. Integer division is like the *quotient* in long division, returning the number of whole times the denominator goes into the numerator:

```
>>> 10/3
3
```

So, is there a *remainder* like in long division? And the answer is yes, it is called the *modulus* (or *mod* for short):

```
>>> 10%3
1
```

If you want float division, you need to make sure at least one of the values is a float first using **float**:

```
>>> float(10)
10.0
>>> float(10)/3
3.3333333333333335
>>> 10/float(3)
3.3333333333333335
```

This is just like converting strings to integers using **int**. In fact, **float** can convert strings directly to floats:

```
>>> float('10.5')
10.5
>>> float('1')
1.0
```

Notice that the string **'1'** has been converted to a float (**1.0**) not an integer (**1**) by the **float** function.

16 That's all folks...

Tune in for the next exciting instalment and good luck with the Challenge problems!