<u>**(1) Interpreter Exercises**</u>

(1.1) Interpreter Hello World

Launch the python interpreter from the command prompt (getting to the command prompt depends on your operating system). You should be able to launch python by just typing "python". On windows you have to make sure you added python to your PATH variable. The python prompt should appear, something like:

```
$ python
Python 2.7.6 (default, Mar 22 2014, 22:59:56)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Now you can type python code in and have it run immediately. This is a very good way test code to check you have the syntax correct - even experienced programmers still do this. For our first python example it is traditional to print "Hello World!" to the screen. We do this using the print statement:

```
>>> print "Hello World!"
Hello World!
```

We just created a string and told python to print it to the screen! If you made a mistake typing, or want to repeat a previous command after editing it, press the up arrow to move through your previous commands, and edit as required, then press enter to rerun them. Then to exit the interpreter: on Mac or Linux press Ctrl-D or type exit(), on Windows press Ctrl-Z then enter.

(1.2) Data Types

When your program deals with data it treats it as one of a set of data types that are basic to the language, like integers (whole numbers, eg 123), floats (numbers which can be fractional, eg 1.23, or expressed as powers of ten, eg 1×10^{23}, written as 1e23) or strings ("Hello World!"). Let's look at what data types are available. For programmers experienced with other languages: python will choose a datatype automatically if we don't tell it explicitly. The type command will tell us which datatype we are dealing with, eg:

```
>>> type(123)
<type 'int'>
```

Unsurprisingly 123 is treated as an 'int' (integer). Try entering the following into the type command and seeing which datatype python uses:

```
1.23
1e23
'A'
"A"
'abc'
9
9999
99999999
9999999999999999999999999999999999999999999999999999999999999999999999999999999
```

Experiment with any other values you are curious about.

You should discover that python treats any number with a decimal point or exponential as a float, simple whole numbers as an int and very long integers as long. Any number of characters is treated as a string - there is no special datatype for a single character, and the single or double quote can be used to enclose the string.

To force a datatype use the name of the desired datatype as an "operator". If the conversion is not possible python will complain to you. Making an integer will generally throw away the fractional part of a number, which may not be what you want (more on this later):

```
>>> 1
>>> type(1)
>>> float(1)
>>> type(float(1))
>>> int(1.0)
>>> float("hello world!")
>>> str(1)
>>> int('abc')
>>> int('0xabc')
```

(1.2.1) Ints and Basic Arithmetic

Now a few more points about integers. Sometimes computer scientists or mathematicians want to use numbers that are not based on counting by tens. For instance hexadecimal numbers are base-16 numbers, and use the letters a-f as extra digits in addition to 0-9, ie in hexadecimal ten is written as a, fifteen as f, and sixteen as 10. Python can use hexadecimal numbers, by simply prefixing the digits with 0x. It can also understand octal numbers (base-8) by prefixing 0o or just 0 and binary (base-2) by prefixing 0b. You may not need to use this capability, but just be aware that writing a zero at the start of your integer will make python treat it as an octal number.

Any integer prefixed with 0x, 0X, 0, 0o, 0O, 0b or 0B is still simply an integer, but python converts the value according to the appropriate base. This can be handy if you need to convert eg a binary number to decimal, simply enter it into the python interpreter. Try it:

```
>>> type(0b10)
<type 'int'>
>>> 0b10
```

You should now be able to understand the punchline of that geeky T shirt that said "There are 10 types of people in the world... Those who understand binary and those who don't"

Basic arithmetic works as you would expect using the character + - * / ** % for addition, subtraction, multiplication, division, exponentiation and remainder, as well as brackets, obeying the normal BODMAS rules. The python interpreter can be used as a handy calculator, and will likely be able to cope with much larger numbers than your physical calculator.

```
>>> 2 + 10 / 2
>>> 2 + (10 / 2)
>>> (2 + 10) / 2
```

One quirk with python 2.x is the division operator when used with integers. By default division of two integers carries out integer division, hence 9 / 2 is 4 remainder 1:

```
>>> 9 / 2
>>> 9 % 2
```

To force ordinary division convert one or both of the numbers into floats, either by putting a decimal point or by forcing the type:

```
>>> type(9/2)
>>> type(9.0/2.0)
>>> 9. / 2
>>> 9 / 2.0000
>>> float(9) / 2
>>> int(9) / float(2)
```

(1.2.2) Strings

A bit more about strings: python has a lot of ways to enter strings. Single or double quotes can be used - handy for when the string itself contains a single or double quote. Enter a string which contains a double quote in the middle. Python should print it back to you, including the surrounding quotes:

```
>>> 'A string with a " in the middle'
>>> "A string with a ' in the middle"
```

Now try changing just the single quotes into double quotes, or vice versa. Python should complain about invalid syntax because you didn't enter a valid string:

```
>>> "A string with a " in the middle"
>>> 'A string with a ' in the middle'
```

Another way to put a quote inside a string is to use the "escape character" \ which tells python to treat the following character in a special way. If we "escape" a quote character we tell python to not treat it as the end of a string:

```
>>> "A string with a \" in the middle"
>>> 'A string with a \' in the middle'
```

There are many other things that can be put after an escape charcter: \n means "new line", "\t" means tab. Try printing a string with a tab and new line in it:

```
>>> print "\tHello\nWorld"
```

Another way to put a new line inside a string is to use triple quoting. The string starts with three single or double quotes, and ends when another triple quote is entered, even if you press enter:

```
>>> print """Hello
... world"""
```

If you pressed enter after """Hello you will notice the python interpreter shows a "continuation" prompt, meaning it is waiting for more input before running the text you've entered. You will see later than this same facility can be used to enter simple multiple line programs into the interpreter (although for any program more than a line or two you will usually same the coed to a file before running it).

What if you want to print a snippet of python code to the screen and the code contains a python string including escape characters? How would you print "\tHello\nWorld" to the screen without it being converted into two lines? One way is to use a "raw" string, by prefixing the string with a r:

```
>>> print r"\tHello\nWorld"
```

This essentially turns off escape characers.

Finally, if you know what unicode is, you may be interested to know that unicode strings are supported by python. In python 3 unicode is the default string type, but in 2.x you must prefix with a u:

```
>>> type(u"This is unicode")
>>> print u"This is unicode"
```

If you don't know what unicode is, basically it is a system to allow more characters than the default English alphabet and punctuation, including other alphabets and symbols. Try this (results may depend on what unicode support your terminal has):

```
>>> "£"
>>> u"£"

>>> print '\u0295\u2022\u1d25\u2022\u0294'
>>> print u'\u0295\u2022\u1d25\u2022\u0294'
>>> print '\U0001f601'
>>> print u'\U0001f601'
```

etc...

What if we wanted to include a number in the string? In practise the number would likely be stored in as a variable, but for now we'll type in the number directly. Python has many ways to produce strings from other datatypes. Probably the clearest method is the format function, which is used to insert extra data into a string, formatted in exactly the way you want. This makes use of another type of escape sequence, using curly braces, which is only used in conjunction with the format function. For example:

```
>>> print 'This is a integer: {}'.format(123)
>>> print 'This is a integer: {}'
```

Without the .format(123) the curly braces are just printed as normal characters. There are very many ways to use the format function, but just as a quick overview let's print some more integers, and then move onto floats in the next section:

```
>>> print "Here are two integers: {} and {}".format(444,666)
```

We can refer to the integers by their position in the .format(). You might think this is the way to do it:

```
>>> print "Here are two integers: {1} and {2}".format(444,666)
```

But, python numbers items starting from 0 not 1. So called 0-based numbering is common to a lot of programming languages, and is something you just have to get used to. Think of it as "the start of the list plus 0" and "the start of the list plus 1", ie as an offset from the beginning:

```
>>> print "Here are two integers: {0} and {1}".format(444,666)
>>> print "Here the same integer twice: {0} and {0}".format(1234)
```

We can also refer to the numbers by giving them names:

```
>>> print "Here is the maximum and minimum score: {max} and {min}".format(min=1,max=9)
```

We don't have to use all the named inputs:

```
>>> print "Here is the maximum and minimum score: {max} and
{min}".format(min=123,max=2345,average=199,total=45678)
```

But any name referred to must be present in the format:

```
>> print "This number does not exist {valwe}".format(value=345)
```

(1.2.3) Floats

A few more points about floats. Floats, or floating point numbers, are normally used in python to represent non-integer numbers. It is useful to know a little bit about how the number is stored internally, as otherwise you may make some mistakes when dealing with these numbers. The number is stored in two parts which you can think of as the significant digits (or significand) and the powers-of-ten (actually the two numbers are stored internally using binary numbers, but you normally don't have to worry about that detail). This is very similar to the scientific number notation that you will be familiar with which, for example, expresses one million as 1×10^6, ie one times ten to-the-power-of 6. In this case the "digits" would be 1 and the power of ten would be 6. To write this as a float in python (and most other programming languages) we can write:

```
>>> type(1e6)
>>> 1e6
```

You should see that python will repeat the number back to you, as 1000000.0, but remember the number is stored internally as 1.000000... and 6. We get the same number if we write 1.00000e6 or 0.1000e7 or 10e5. (If this is not clear to you do some revision of scientific number notation). Now it should be clear why the datatype is called float, because the decimal point in the first part of the number is able to float (move) so that the significand is always between 0.00000... and 9.99999... by increasing or decreasing the power-of-ten to compensate. This is an efficient way to store very large or very small numbers, as it takes less memory to store 1e300 than 1 followed by three hundred zeros, but bear in mind the significand only has a finite number of digits, some if you enter a very long significand, it will get rounded to the nearest storable float. To test this lets ask python if it thinks two number are identical or not, using the equality operator (more on operators later):

```
>>> 1.1 == 1.1
>>> 1.2 == 1.1
>>> 1.1112 ==1.1111
>>> 1.111111112 ==1.111111111
>>> 1.11111111111112 == 1.11111111111111
>>> 1.1111111111111111112 == 1.1111111111111111111
```

You should see that the float datatype eventually runs out of precision to store small differences. So just remember: floats have finite precision, do not trust that a float will be exactly equal to another float if they were calculated using different methods. There will likely be rounding errors. Another issue is the underlying binary numbers used to store the digits. Although displayed using familiar base-ten digits, the float (and all other data in python) is stored internally as binary numbers. This means that certain number easily expressed precisely in base-ten, such as 0.1, cannot to exactly expressed in binary, but can only be approximated. To see this we need to tell python to display the float number to a large number of significant digits, using python's string formatting syntax. Let's tell it to display 30 significant digits. To do this we use the curly braces as before, but put a colon followed by a special code telling python how to format the data. In this case we are saying to display as a floating point value with thirty decimal places:

```
>>> print '{:.30f}'.format(1.0)
>>> print '{:.30f}'.format(10.0)
>>> print '{:.30f}'.format(0.5)
>>> print '{:.30f}'.format(0.1)
>>> print '{:.30f}'.format(1e-1)
>>> print '{:.30f}'.format(0.2)
```

We can see that binary numbers cannot precisely store 0.1, so instead of seeing 0.100000000..... we start to see non zero digits. This seems extremely weird at first because in decimal it can be stored easily in floating point using 1e-1.

To display a float in scientific notation use the format code letter e instead of f:

```
>>> print '{:.5f}'.format(1.2399999999e10)
>>> print '{:.5e}'.format(1.2399999999e10)
```