A Short Introduction to Python Programming

July 12, 2012

1 Introduction

The goal of this session is to get started with Python through the development of a simple but effective tool for calculating average marks from student results. Along the way we will introduce concepts such *variables*, *user input*, *conditional statements*, and *loops*. We will also mention a few more advanced concepts such as types, methods and classes.

If these terms are unfamiliar to you then it might seem a lot to get to grips with right away - remember we're here to help you at every step!

2 Preliminaries

Before we begin getting to grips with any programming, you should take a moment to familiarise yourself with the environment that you'll be using. In order to write programs in Python you will need to be familiar with:

- **The Terminal** - You will be using this to issue commands, and to run the Python programs you develop.
- **Text Editor** - You will be using a text editor to write your Python programs. Individuals often have a preference for a particular editor, so if you are familiar with one we have installed then feel free to use it. Many students recommend KWrite, though this is only a suggestion.

The lab demonstrators will be able to say a few words about terminals and the text editors so feel free to ask any questions.

3 Using Python Interactively

To begin an interactive Python session you can type `python` into a terminal, and press `Enter`. Having done this you will notice that the prompt in the terminal has changed from `bash-3.2$` to `>>>`.

Listing 1: Starting an interactive shell

```
bash-3.2$ python
>>>
```

This new prompt is associated with the Python interpreter. You can think of the interpreter as a mechanism for evaluating expressions you enter. For example, if you enter `2+2` (then press `Enter`) you'll see that it evaluates this to `4`, as shown below.

Listing 2: Simple arithmetic

```
>>> 2+2
4
```

You should experiment with expressions and operators, including arithmetic operators such as +, -, * and /. Some examples shown below include the use of variables, which you can think of as storage for some value. The = symbol is used to assign a value to a variable. There is support for floating point numbers - operators applied to mixed type operands convert the integer operand to floating point.
N.B. you may get an answer similar to `6.400000004` when you calculate the last expression, this is to do with how the computer stores numbers in memory (ask a demonstrator for details!).

Listing 3: Complex arithmetic

```
>>> (60*9-4)/2
268
>>> 7/3
2
>>> width = 20
>>> length = 100/4
>>> width * length
500
>>> surface = depth = 4
>>> surface * depth / 2.5
6.4
```

Python is also capable of manipulating strings, which may be enclosed in single or double quotes. Conventionally, a string should be enclosed in double quotes if the string contains a single quote and no double quotes.

Listing 4: String manipulation

```
>>> 'Hello there'
'Hello there'
>>> a = 'Hello there'
>>> a
'Hello there'
```

```
>>> 'You can\'t'
"You can't"
>>> '"Yes," he said.'
'"Yes," he said.'
>>> "\"Hello\" she shouted."
'"Hello" she shouted.'
```

We've now seen expressions containing a few different data types, including int (whole numbers), float (decimals) and str (strings of characters). Whilst in-depth coverage of the Python type system is beyond the scope of this introduction, it is important to mention compound data types, which allow us to group values. The *list* is an example of such a data type. A list can be written as comma-separated values (items) between square brackets. The items of a list do not need to have the same type.

Listing 5: Lists

```
>>> a = ['hello', 'there', 10, 17553]
>>> a
['hello', 'there', 10, 17553]
```

The indices of a list start at 0. Many operations are possible on lists, including element access, list slicing (taking part of a list), and list concatenation. When slicing a list we use a range of the form `a:b`, where `a` and `b` are the indices we want to start and end our slice at. If one of the values is missing, for example `a[:3]`, then that implicitly means the start of the list.

Listing 6: List manipulation

```
>>> a = ['hello', 'there', 10, 17553]
>>> a[0]
'hello'
>>> a[3]
17553
>>> a[:2] + ['you', 2*3]
['hello', 'there', 'you', 6]
>>> 2*a[:3] + ['123']
['hello', 'there', 10, 'hello', 'there', 10, '123']
>>> a[:]
['hello', 'there', 10, 17553]
>>> a[2] = a[2] + 15
>>> a
['hello', 'there', 25, 17553]
>>> a = ['a', 'b', 'c', 'd']
>>> len(a)
4
```

So far we've seen how expressions can be evaluated, how variables can be used to store values of different types, and how variables can be grouped using list

structures. However, practical programming is about much more than evaluating simple expressions. Indeed, we can think about programming as being about issuing a sequence of instructions, many of which will contain expressions, that can be understood by a computer. A program is merely this sequence of instructions. That is, rather than issuing instructions one-by-one for an interpreter to evaluate, we can write a program containing these instructions - as we will see!

For now you'll need to type `exit()` in order to escape your interactive session.

4 Your Task

The task for today is to write a program, *averageMark.py*, that accepts a list of student marks, calculates their average, and prints the result to the screen. When we run the program from a terminal, we typically enter the program's name followed by a number of arguments, student marks in this case.

Listing 7: Running the averageMark program

```
bash-3.2$ ./averageMark.py 71 75 79
```

Entering the above into a terminal should produce output something like that shown below.

Listing 8: Output of the averageMark program

```
The average mark is: 75
```

Now that we have a goal, we can begin developing our program. You might not finish all the sections in this handout today, so do feel free to continue exploring Python in your own time.

4.1 Making the File

The first task is to make the *averageMark.py* file. To do this you should open a text editor. Once there you should enter the code shown below and think about what the output would be. To test your thoughts you should save the file as `averageMark.py` and run your program by executing the following two commands in the terminal: `chmod u+x averageMark.py` then `./averageMark.py`
The first command sets the file as `user executable`, meaning we can run it like a program, and only needs to be run once. The second command runs the program.
You should be able to see that the program above consists of three lines. The `#!/usr/bin/env python` line makes the Python program directly executable. This will almost always be the first line of any Python program you write. The next non-blank line also begins with a #, which means that it is a comment. Finally, the next line is a print statement that will output the associated string. You should run your program and check that the output is what you expect.

Listing 9: Basic program skeleton

```
#!/usr/bin/env python

# code goes here
print "Hello World"
```

4.2 Variables

Variables are typically one of the first concepts you will encounter when starting to develop computer programs. As the name implies, they are a mechanism for storing values that can subsequently be used in expressions.

Being able to understand the operation of a program without an executing it is a valuable skill. Indeed, once you become adept at this, you'll likely find that you can write high-quality code of your own. See if you can guess what will happen if you run the program below before you execute it.

Listing 10: Variables

```
#!/usr/bin/env python

# code goes here
x = 10
print x
y = x * 2
print y
```

So what happened? Firstly, you will notice we introduced two variables, `x` and `y`. We store the value 10 in the variable `x`, which allows us to refer to it later. We then use the value of `x` (10), multiply it by 2 and then store it in `y` (20).

You may like to experiment with the use of floating point values, as well as integers. For example, you might try calculating the average of two integers, such that the result would be a floating point value.

4.3 User Input

So now we've seen how to write a simple program that performs calculations on "hard coded" numbers and prints a result. However, for our program, we want to be able to do it for any numbers input by the user. This means we need to know how to capture user input. To do this python uses a variable list stored in `sys.argv`. The input to the script is assumed to be space delimited, with each 'word' going into a different space in the list. For example, below is

a command that executes the `averageMark.py` program with three arguments provided by the user, i.e., `firstWord`, `secondWord` and `thirdWord`. Beneath that is a table that shows how each of those arguments can be accessed from within the program.

Listing 11: Running the averageMark program

```
./averageMark.py firstWord secondWord thirdWord
```

	Value
sys.argv[0]	averageMark.py
sys.argv[1]	firstWord
sys.argv[2]	secondWord
sys.argv[3]	thirdWord

The easiest way to see this work is by executing the program show below. Note that taking arguments from the command line requires the `sys` library to be imported. After you've understood the program below you should experiment by using different numbers and types of arguments. For example, can you take four integers as input and output their sum?

Listing 12: Command line arguments

```
#!/usr/bin/env python

# Command line arguments example
#
# sys allows us to access system specific information
import sys

print sys.argv[0]
print sys.argv[1]
print sys.argv[2]
```

4.4 Conditionals

Another important programming construct is the conditional statement. This form of statement allows us to perform an action only if a certain condition is met. For example, if we wanted to detect whether the first argument given to a program was equal to 100, we could easily determine this using a conditional statement. Conditional statements are more commonly known as if-statements. The example below should make clear the reason for this convention. Although not shown below you should remember that it is possible to nest if-statements.

Listing 13: Conditional statements

```
#!/usr/bin/env python

# If-statement example
x = 1

if x < 0:
    x=0
    print 'Negative changed to zero'
elif x == 0:
    print 'Zero'
elif x == 1:
    print 'Single'
else:
    print 'More'
```

To ensure that you're familiar with using the if-statement you should develop a program that takes an integer, representing a student exam mark, as an argument and outputs a relevant grade. You should assume that the integer is in the range [0,100], making a sensible decision about what to do if the integer is outside this range.

4.5 Loops

It may now seem that we have all the tools required to complete the task of writing a program that accepts a list of student marks, calculates their average, and prints the result to the screen. However, as we don't know how many students we'll need to deal with, we need to understand something about iteration before we can complete the task. You might argue that we could check manually by going through `sys.argv[0]`, `sys.argv[1]`, `sys.argv[2]` and so on, but how do we know when to stop?

The for-loop in Python iterates over the items of any sequence, e.g., a list or a string, in the order that they appear in the sequence. The example below shows how a list of integers can be printed in-order using a for-loop.
Once you have understood the example above you'll have a good idea about how the for-loop works. Knowing that, can you guess what be output by the second example shown below? You should try to work it through in your head or on a piece of paper before executing the program.

4.6 Can You Put It All Together?

So, we now know how to take input from the command line as a list of items, how to iterate over the items of a list, and how to take, possibly conditional,

Listing 14: Loops

```
#!/usr/bin/env python

# For-loop example

a = ['one', 'seven', 'nine']

for x in a:
        print x
```

Listing 15: Loops with conditional statements

```
#!/usr/bin/env python

# Can you see what is happening?

a = ['one', 'seven', 'nine']
b = [2, 8, 10]

for x in a:
    for y in b:
        if x == 'nine': print y
```

actions based on the values of variable. Given this knowledge it should now be possible to write a simple program to take a list of student marks, calculates their average, and print the result to the screen. Try doing this now!

If you experience any difficulties then please ask a demonstrator for help, they will be delighted to help you.

N.B. Items in the `sys.argv` list are strings, to convert them to numbers (integers), use the `int()` function, for example: `int(sys.argv[0])`

5 Additional Functionality

Now that you've completed the overarching task, you might like to provide some additional functionality. To get you started, the example below shows how functions can be defined. Functions provide us with a convenient way of doing computation without having to replicate code. You should spend time understanding this example before attempting the additional ideas listed below. To extend your program, you might like to:

- Write a function called `sum` that takes a list of integers a returns their sum, then use your new function to simply the code that calculates the

Listing 16: A simple function

```
#!/usr/bin/env python

def printme( str ):
    "This prints a passed string into this function"
    print str
    return str

printme('Hello')
```

average of the student marks.

- Write a function called `sumZeros` that takes a list of integers and returns the number of zeros found in the list.
- Write a function called `median` that takes a list of integers and returns the median of the values in the list.

If you do manage to complete all these exercises the congratulations, you've done exceptionally well if you're new to Python! That said, regardless of how far you have got with the above tasks it is important to remember that learning programming, or indeed any single programming language, takes time. Indeed, it is only with time that technically challenging concepts become second nature.

6 Resources

We hope you have enjoyed this taster session. For any one who wants to find out more or to continue learning the Python programming language, a non-exhaustive list of web resources that you might find useful is shown below.

- http://www.diveintopython.net/
- http://learnpythonthehardway.org/book/
- http://www.python.org/doc/

The LearnPythonTheHardWay book listed above comes highly recommended by our team of lab demonstrators.