**Python Scripting 1**

**Command line interpreters** for Linux come in many different flavours but the one thing they have in common is that they are **scriptable**. By this we mean that **commands can be added into a text or script file**. These stored commands can then be played back by running the script file. Each command in the file is then executed as though it was being typed at the prompt.

[Scripting](http://en.wikipedia.org/wiki/Scripting) is like programming. Like a program, commands are aggregated together in a file. All of the commands can then be run by running the script.

## Python

Python is an **interpreted** language, which means that written code gets passed to an interpreter which converts it line by-line. Python first needs to load it’s runtime engine i.e. the software the PC needs to convert the language from high-level to low-level, then read in the text files and then run the program

No compilation process – write code, save it, run it.

## Programming or Scripting?

If Python allows you to store commands in a single file and execute them as one then surely it is just another programming language? Kind of.

Programming languages are, in general, **more powerful** and, if compiled, certainly **faster** than scripting languages.

Most programming languages take a set of commands (source code) and compile it into machine readable language (object code). Scripting languages instead take each command in turn and execute it (called **interpretation**). This means that the script is slower but that it is not machine or CPU specific, allowing the code to be portable.

**Getting Started**

As scripts are text files you will need to be comfortable editing text files. Unix/Linux has spawned many text editing programs like ***vi***and ***nano*** but they can be hard to use (although very powerful).

## Hello World

No introduction to any sort of programming language would be complete without a program that displays the words "Hello World!" on your screen. The code looks like this:

**# this is a comment**

**print “Hello World!”**

The first line is a comment and will therefore be ignored by the interpreter. The second line used the object ***print*** to display text on the screen.

When you've entered the text save the file. We'll call it ***hello*** and, to help us tell that it is a Python file we'll add a ***.py*** extension. The complete filename is therefore ***hello.py.*** Once the file has been saved, exit from the editor. As this is a script file designed to be run we need to make it executable.

**chmod 700 hello.py**

Finally we're ready to run the script by calling the interpreter and passing it the Python script. (You may need to update your environment PATH in order to locate the Python interpreter)

**python hello.py**

You should see

**Hello World!**

Congratulations! You've just written your first script. All of the others will be done in the same way: write the text, save the file, make it executable and run it. All of our scripts will work in the same way as this example. Commands will be placed in a file and they will then be run when the file is executed. All the commands we have already covered, and many more besides, can be used in scripts.

## Variables

Variables are "boxes" used to store values (**numeric** or **text)** , which can be manipulated in exactly the same way as the values themselves could be. For example in this script two variables are assigned numbers and then some simple arithmetic operations are carried out on them and the results displayed.

**# my first variables**

**firstNum=7**

**secondNum=5**

**answer=firstNum + secondNum**

**print “Adding %d and %d = %d” % (firstNum, secondNum, answer)**

Extend the script to multiply and divide?

Now let’s extend this script to ask the user to enter two numbers they want to add:

**# my first variables**

**firstNum=raw\_input("Please enter the first number to be added")**

**secondNum=raw\_input("Please enter the second number to be added")**

**answer=int(firstNum) + int(secondNum)**

**print "Adding %s and %s = %d" % (firstNum, secondNum, answer)**

Notice that input is read in as a string, therefore we needed to convert the values to integers before performing the integer arithmetic.

Here is a final arithmetic example where we ask the user for a price and then we display that price + VAT:

**# my first variables**

**beforeVAT=raw\_input("Add VAT to this: ")**

**afterVAT =float(beforeVAT) \* 1.2**

**print (afterVAT)**

## Executing Linux Commands

Python’s **subprocess** module is used for executing external commands and this is what we will require in order to run our Linux shell commands from within the Python script. It allows us to spawn processes, connect to their input/output/error pipes and obtain their return codes.

The following is an example of a short script which will use the “ls” command to list the contents of the current working directory:

**import subprocess**

**subprocess.call([‘clear’])**

**ls\_output = subprocess.check\_output([‘ls’])**

**print (ls\_output)**

**sub\_process** comes with 3 main functions:

**call** - we can run the command line with the arguments passed as a list of strings e.g.

**subprocess.call([‘ls’, ‘-l’])** or we can set the shell argument to TRUE and list the entire command e.g. subprocess.call(‘ls –l’, shell=True) which prints back to the parents stdout/stderr

**subprocess.check\_call –** run command with arguments, wait for it to complete. If return code was zero then return, otherwise raise *CalledProcessError* with the *returncode*

**subprocess.check\_output –** run command with arguments and return it’s output. If return code was non-zero then raise *CalledProcessError* with the *returncode*

The following script demonstrates the use of these functions:

**import subprocess**

**subprocess.call([‘clear’])**

**ls\_output = subprocess.check\_output([‘ls’, ‘-l’])**

**second\_output = subprocess.check\_output(‘ls –l | wc –l’, shell=True)**

**print (ls\_output)**

**print “The count is %s” %(second\_output)**

Sometimes we need to pass variables to the external commands we wish to run. For example if we ask the user for the name of a directory and then we issue the command to list the contents of that directory. In this situation we use the %s to indicate to the **ls** command that there is a string value to follow, then after the quotes we list the variablename following the % symbol:

**import subprocess**

**subprocess.call([‘clear’])**

**directory = raw\_input(“Enter a directory: ”)**

**result = subprocess.check­\_output([‘ls %s’ % directory], shell=True)**

**print (result)**

Some external commands require the use of quotes, for example when specifying formatting for the date command you would issue the following command at the command line:

date ‘+%d/%m/%y’

When we use the subprocess module to issue this command we need to “escape” the quotes belonging to the command by proceeding them with a backslash (\) character. This means that subprocess will ignore the meaning of the quotes and retain them when it spawns the process to execute the command.

**date = subprocess.check\_output([‘date \’+%d/%m/%y\’’], shell=True)**