Unit 8: SCRIPTING

Shell Scripting

Firstly, a shell describes something use to interface with an operating systems services, GUIs (Graphical User Interfaces) or CLIs (Command-Line Interfaces) are examples, though in this unit we will be focusing specifically on Command-line shells. On UNIX or Linux any command-line interpreters are called shells.

A shell script is a text-file containing a set of instructions to be interpreted by a shell. This file can then be executed from a shell, which is much faster than doing all of these individual steps separately. These files can also be executed with arguments too, like regular CLI commands. Most repetitive or menial tasks that can be done on a computer can be automated with scripts. For example, if you wanted to recursively trawl your hard drive and create an index of your files and how much space they took up, you could script it! Or say you wanted to get an update one the weather or the news whe you opened your shell. You could script this too! Computers are great!

The Hashbang

In Linux or Unix, all shell scripts are expected to have a special first line, starting with what is called a hashbang (also called a she-bang), which looks like this `#!`. This line tells the shell being used which interpreter should be used to execute the file.

For example tcsh..

```
#!/bin/tcsh
echo "Hello World!"
```

Or Python..

```
#!/usr/bin/env python2.7
print "Hallo Welt!"
```

Python

Python is a interpreted programming language, which means that it doesn't need to be compiled into machine code before it is run, as oppose to compiled programming languages. One advantage of it is that it reads very similar to psuedocode (explained in module 7), which makes it fairly easy to understand at a glance. Let me show you an example, I will define two identical functions: One in C++ and then one in Python.

C++ Implemenation

```
int get_factorial(int value)
{
    int factorial=1;
    for (int i = 1; i <= value; ++i) {
        factorial *= i;
    }

    cout<< "Factorial of "<<value<<" = "<<factorial;
    return factorial;
}
```

Python Equivilent

```
def get_factorial(value):
    factorial = 1
    for i in xrange(1, value + 1):
        factorial *= i

    print("Factorial of {0} = {1}".format(value, factorial))
    return factorial
```

At least for me, this second function is much easier to make sense of. There is much less punctuation used in the formatting of python, which is something that can make C++ look quite confusing at first. One way Python achieves is by by requiring strict use of whitespace. You can see that the indentation quite consistent between the two functions, but the difference is that in Python this is required. We could remove this indentation in C++ and our function would compile and run just fine. For example, as horrible as it looks, here is a how our C++ function could have looked:

```
int get_factorial(int value){int factorial=
1;for(int i=1;i<=value;++i){factorial
*=i;}cout<<"Factorial of "<< value<<" = "<<
factorial;return factorial;}
```

This works exactly as before, whereas straying too far from the formatting of our Python function gives us an error:

```
>>> def get_factorial(value):
... factorial=1
  File "<stdin>", line 2
    factorial = 1
            ^
IndentationError: expected an indented block
```

This is possible in C++ because it uses curly braces {} to contain code within constructs such as functions and for/while loops, and semicolons ; to signify line endings (*semicolons for line endings are supported in python syntax, but not encouraged unless required.*). The advantage of this strict whitespace formatting is that it encourages clear and easy to read code. This is one example of how using Python makes for code that is quick to understand and modify.

Another characteristic is that is that you do not need to declare variables until you use them, or set their type, which makes Python a *dynamically-typed language*. Also the language is entirely object-oriented, including every variable, therefore different types are just instances of different classes.

Though there is a cost to this freedom that comes with Python coding: execution speed. Given the right task, C++ could outperform Python more than 400 times over. But this is very understandable. This analogy might help:

> Imagine having 2 guests over on a Friday evening for dinner. Guest 1's favourite food is ham/pineapple pizza, so you prepare him some. But you have never met guest 2 before, so you aren't sure what he likes, so you decide to buy a couple other pizza's that he might be into, including separate vegan and vegetarian options, just in case. Guest 1 is the C++ code, the compiler knows exactly what to expect and can prepare for it, and even perform some validation checks to help ensure everything will run smoothly and fast. Guest 2 on the other hand, is the python code. The objects being manipulated could be anything, so most validation would be useless. The interpreter relies on you writing compatible code.

It's important to note that the Python interpreter is in fact written in C, a very popular statically-typed compiled language. C is compiled directly into machine code, which is perhaps the lowest level programming language: binary, which is executed directly by the CPU. It's also worth mentioning that interpreted languages are not necessarily slower than compiled languages. ASM (assembly language) is an interpreted language that can in many cases run tasks faster than compiled C. In fact C was written in assembly language.

Though it may not be as speedy as other languages, Python is very effective in other areas. High level abstraction and it's dynamically typed nature, make it a very easy language to learn and use, even for people who might not have programmed before. It is also much faster to write than a C program would be, as you don't have to spend so much time on memory-management and many useful algorithms and functions are already defined in the standard libraries.

Python Wrapped APIs

This is a method that is used which is effective at taking advantage of the ease and versatility of Python, and the speed and efficiency of a relatively low level language. Tasks which are CPU-intensive or time critical can be executed in low level languages. Python-wrapping is the process of defining python functions and classes which mirror, and point to, those written in another language. When these functions are called from within python, they are executed in the wrapped language, and when you interact with one of these objects, you are indirectly interacting with an instance of an object from the wrapped language.

OpenMaya Python

Autodesk Maya's C++ API, OpenMaya, has it's own Python-Wrapped version packaged with it. This proves to be very handy for quickly putting together plugins and custom workflow tools, which might run to slowly in its default python `maya.cmds` API. In one test, I rewrote a MEL tool which applied a array attribute containing all vertex positions to a mesh, which I used the OpenMaya Python API for, and it ran twice as fast as the original.

From within a Python environment in Maya, this can be accessed from `maya.OpenMaya` for OpenMaya 1.0, or from `maya.api.OpenMaya` for the newer and more Pythonic OpenMaya 2.0 (in Maya 2016 or later).

Application Specific Languages

Many powerful, industry-level applications have their own proprietary languages built in. There are many advantages for having application specific languages (often called *domain specific languages*), one being that they are often much faster to learn, as they are only designed to perform what is required in the scope of the program, which means that in some cases their functions can be refined to make them as efficient as possible for their specific tasks. One example of an application specific language is Maya's MEL, which stands for Maya Embedded Language. This language contains a host of commands only relevant to 3d design and animation and only works within the program.