> *What's all this then, Amen!*
>
> — *Monty Python*

Introduction

What is Software Engineering?

Software Engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software (*IEEE std 610.12-1990)*. A software engineer must have a good understanding of tools and techniques for requirements gathering, specification, design, implementation, testing and maintenance.

These days, software systems are often very large and many contain key safety or mission-critical components. The complexity of modern software systems require the application of good software engineering principles. Indeed, many agencies mandate the use of particular tools and techniques to achieve very high quality and reliable software.

This course introduces some of the key principles of software engineering and gives you the chance to put them into practice by using the programming language Python. While you'll learn lots of Python in this course, it is equally important that you learn some common programming principles and techniques so that learning the next language is not as daunting.

What is Python?

Python is a powerful, yet simple to use high-level programming language. Python was designed and written by Guido van Rossum and named in honour of Monty Python's Flying Circus. The idea being that programming in Python should be fun! The quotes in these notes all come from Monty Python and it has become somewhat of a tradition to reference Monty Python when writing Python tutorials.

What is Software?

Software (a computer program) is a set of instructions that tell the computer (hardware) what to do. All computer programs are written in some kind of **programming language**, which allows us to precisely tell the computer what we want to do with it.

Features of a Programming Language

There are many programming languages but they all have some things in common.

Syntax and Semantics

Firstly, each program language has a well-defined **syntax**. The syntax of a language describes the well-formed 'sentences' of the language. Syntax tells us how to write the instructions: what is the order of the words, how our program must be structured, etc. It is a bit like grammar and sentence rules when we write in English.

When we write a sentence in English, we may mess up the grammar and still be understood by the reader (or, the reader may ask us what we mean). On the other hand, if we mess up the syntax when we write a computer program, the computer is not as forgiving – it will usually respond by throwing up an error message and exit. Therefore, it is fundamentally important that we understand the syntax of a programming language.

Secondly, each self-contained piece of valid syntax has well defined **semantics** (meaning). For programming languages, it is vitally important that there are no ambiguities – i.e. a valid piece of syntax has only one meaning.

High- and Low- Level Languages

Programming languages are typically divided into **high-level** and **low-level** languages.

Low-level languages (like assembler) are 'close' to the hardware in the sense that little translation is required to execute programs on the hardware. Programs written in such languages are very verbose; making it difficult for humans to write and equally importantly, difficult to read and understand!

On the other hand, high-level languages are much more human-friendly. The code is more compact and easier to understand. The downside is that more translation is required in order to execute the program.

Compiled and Interpreted Languages

Before we can run our program, we need to type out the list of instructions (called our **source code** or just **code**) that outlines what our program does when we want to run it later. So how does our source code then turn into a running program? This depends on whether the programming language we have written the source code in is either a **Compiled** or an **Interpreted** language.

Compiled languages come with a special program, called a **compiler**, which takes source code written by a user and translates it into **object code**. Object code is a sequence of very low-level instructions that are executed when the program runs. Once a user's program has been compiled, it can be run repeatedly, simply by executing the object code. Examples of compiled languages include Java, Visual Basic and C.

Interpreted languages come with a special program called an **interpreter** which takes source code written by the user, determines the semantics of the source code (interprets it) and executes the semantics. This is typically done in a step-by-step fashion, repeatedly taking the next semantic unit and executing it. Examples of interpreted languages include Python, Ruby and Lisp.

Both compilers and interpreters need to understand the semantics of the language - the compiler uses the semantics for the translation to object code; the interpreter uses the semantics for direct execution. Consequently, a program in a compiled language executes faster than the equivalent program in an interpreted language but has the overhead of the compilation phase (i.e. we need to compile our program first before we can run it, which can take a while if the program is very big). If we make a change in our code with a compiled language, we need to recompile the entire program before we can test out our new changes.

One advantage for an interpreted language is the relatively quick turn-around time for program development. Individual components of a large program can be written, tested and debugged without the overhead of compiling a complete program. Interpreters also encourage experimentation, particularly when starting out – just enter an expression into the interpreter and see what happens.

Python is an interpreted language – it has an interpreter. The python interpreter is a typical **read-eval-print loop interpreter**. In other words, it repeatedly reads expressions input by the user, evaluates the expressions and prints out the result.

Data Types

Another important issue when considering programming languages is the way the language deals with **types**. Types are used by programming languages (and users) to distinguish between "apples and oranges". At the lowest level, all data stored in the computer are simply sequences of 1's and 0's. What is the meaning of a given sequence of 1's and 0's? Does it represent an "apple" or an "orange"? In order to determine the intended meaning of such a sequence, it has a *type* associated it. The type is used to determine the meaning of the 1's and 0's *and* to determine what operations are valid for that data. Programming languages come with built-in types such as integers (whole numbers) and strings (sequences of characters). They also allow users to define their own types.

Programming languages implement **type checking** in order to ensure the consistency of types in our code. This stops us from doing silly things like trying to add a number to a word, or trying to store an "apple" in a memory location that should only contain an "orange".

Programming languages are typically either **statically typed** or **dynamically typed** languages. When using a statically typed language, checks for the consistency of types are done 'up-front', typically by the compiler and any inconsistencies are reported to the user as type errors at **compile time** (i.e. while the compiler is compiling the program). When using a dynamically typed language, checks for type errors are carried out at **run time** (i.e. while the user is running the program).

There is a connection between whether the language is compiled or interpreted and whether the language is statically or dynamically typed; many statically typed languages are compiled, and many dynamically typed languages are interpreted. Statically typed languages are usually preferred for large-scale development because there is better control over one source of 'bugs': type errors. (However remember, just because the program has no type errors does not make it correct! This is just one kind of bug that our program must not have to run properly.) On the other hand, dynamically typed languages tend to provide a gentler introduction to types and programs tend to be simpler. Python is dynamically typed.

Notes Formatting

In the remainder of the notes we will use different boxes to indicate different parts of the content. These may appear in the readings or separately on Blackboard. Below are examples.

Information

Detailed information will appear in boxes similar to this one, such as the syntax and semantics of Python code presented in the notes, and summaries of the content. Understanding the concepts presented here will assist you in writing programs.

Aside

Further information will appear in these boxes. These asides go beyond the course content, but you may find them interesting. You can safely ignore them, but they will often demonstrate several powerful features of Python, and they may be a useful challenge for some students.

Extra Examples

In the remainder of the notes, we sometimes give more detailed examples. These extra examples are delimited from the main text by these boxes. You may find these examples useful.

Visualisations

These boxes appear on Blackboard, below the readings, and contain visualisations of Python code. You might find that these visualisations aid in your understanding of how Python works. You can visualise your own code by going to the Python Tutor Visualisation Tool at http://pythontutor.com/visualize.html. The home page for Python Tutor is at http://www.pythontutor.com/.