CE151

Introduction to Programming

Part 1

Introduction

Module Schedule

There are two lectures (Monday 11.00-11.50 and 13.00-13.50) and two 2-hour labs (on Tuesdays and Thursdays) each week in the autumn term. In addition there are two extra lectures in weeks 10 and 11 to allow time for revision.

Students have been split into two groups for the labs so you will be expected to attend one lab per week; your personal timetable will indicate which lab session you should attend.

Assessment

- One two-hour examination in week 15 (January) (50% of the module credit)
- One multiple-choice progress test on Monday of week 6 (10%)
- Two programming assignments to be submitted by lunchtime on Thursday of week 7 and Monday of week 11 (20% each)

  You will be required to demonstrate your submitted programs in your labs in weeks 8 and 11

Recommended Reading

The main recommended text for this module is

Python 3 for Absolute Beginners, Tim Hall and J-P Stacey (Apress, 2009)

An alternative (although expensive) text is

Starting out with Python, Tony Gadiss, 2nd edition (Addison Wesley, 2012)

Other texts on Python are available – if you choose to obtain something different you should ensure that it uses Python 3.

Computing

The word “computing” has a variety of meanings depending on the context in which it is used and the person you’re talking to. For our purposes, we will take it to mean performing computations automatically, that is, without human intervention. This involves two fundamental activities

- the design and construction of machines that are capable of performing any desired computation
- the programming of these machines to perform the particular computation that we require

During this module we shall concentrate on the second activity.

Historical Background 1

In order to provide some background, we start by taking a very brief look at the development of machines and programming languages, and the relationship between them.

In the 1830s Charles Babbage designed two computing machines. The first, the ***difference engine***, was intended to perform a particular kind of computation (finding the values of polynomial functions). The second, the ***analytical engine***, was intended to be a general purpose automatic computing device. It is this second machine that is of interest to computer scientists, and it is here that the history of automatic computing really begins.

Historical Background 2

The analytical engine is considered to be significant because it contained the essential components of every computer that has ever been built. It had a "store" (the equivalent of a modern computer's memory) and a "mill" (the equivalent of a modern computer's arithmetic unit). Most importantly it incorporated a system of control, based on the technology that had been developed to control weaving looms, which enabled the user to describe any desired computation in coded form on punched cards. In other words, the analytical engine could be ***programmed***.

Historical Background 3

Although neither machine was ever built in Babbage's lifetime, his co-worker Ada Lovelace did design some programs for the analytical engine, most notably one for calculating the Bernoulli numbers. This program included repetitive loops and conditional branches, which still form the basis of control in most modern programming languages. As well as being the world's first programmer, Ada Lovelace was also clearly aware of the limitations of computing machines. She wrote in her paper

" ... the Analytical Engine has no pretensions whatever to originate anything. It can do whatever we know how to order it."

Historical Background 4

Over one hundred years elapsed before any serious advances were made on Babbage's design. Towards the end of this period, developments in electrical circuitry, new theories of computation and the advent of the Second World War provided the means, the understanding and the motivation to construct a more powerful electronic version of the analytical engine.

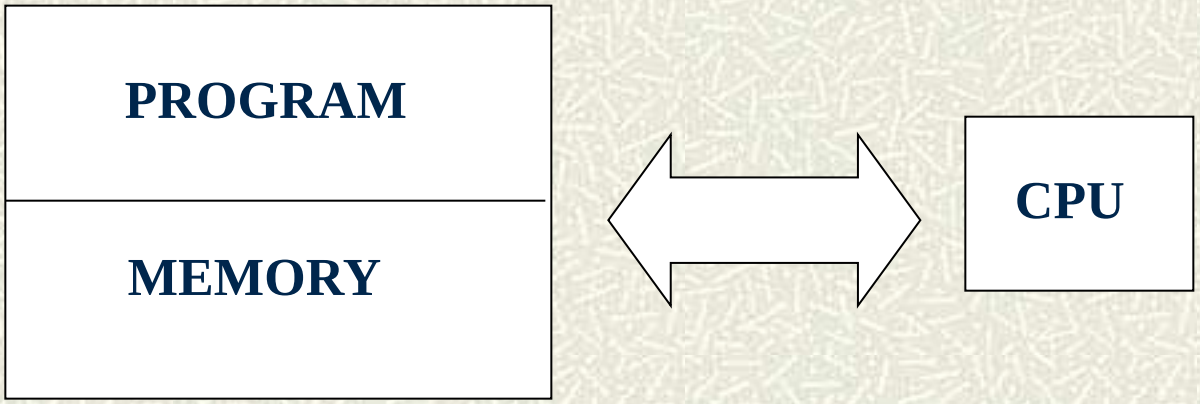
Historical Background 5

In this country, Alan Turing and others developed the Colossus machine at Bletchley Park as part of a wartime effort to decipher coded messages. Shortly after the war, at Princeton University, John von Neumann and his co-workers created what are usually considered to be the world's first stored-program computers, the ***ENIAC*** and the ***EDVAC***.

Their design included the store (now called the ***memory***) and the mill (now called the ***central processing unit*** or ***CPU***) that were present in Babbage's design, and also incorporated a control unit and input and output devices. The design was so fundamental that all subsequent variations on it are known as ***von Neumann architectures***.

Historical Background 6

The essential components of a von Neumann architecture (minus the I/O and control units) are usually depicted as is shown below.



Historical Background 7

In a conventional von Neumann architecture, the executable program, which consists of a sequence of numbered instructions, is held in the store. A ***program counter*** contained in the control unit holds the number of the next instruction to be executed. The control unit is responsible for fetching instructions from the store and executing them one at a time, in a process known as the ***fetch/execute cycle***.

Historical Background 8

In general, each instruction requires at least one of the following actions to be carried out

- read data from the memory into the processing unit
- perform some arithmetic/logical operation on the data
- write data from the processing unit to the memory
- change the content of the program counter

From these basic operations, it is possible to compose any possible computation

Historical Background 9

This design has been improved upon and made more efficient in many different ways, and the speed of electronic circuitry has increased by several orders of magnitude in the years that have elapsed since the building of the first electronic computers. We now have access to machines that can execute several thousand million instructions every second - but at the core of practically all these machines lies the von Neumann architecture. Not only has it been the major influence on the design of machines, but also on the design of programming languages.

The Multi-Level View 1

The last 50 years have seen enormous technological advances in the design and construction of computers – faster processors, denser storage media, improved peripheral devices, and so on. As the machines have become more powerful, they have become more complex and more difficult to program. At each stage of technological development it has been both possible and necessary to divert some of the gain in computing capability away from the end user, and use it to enhance the machine itself. The process works as described on the next slide.

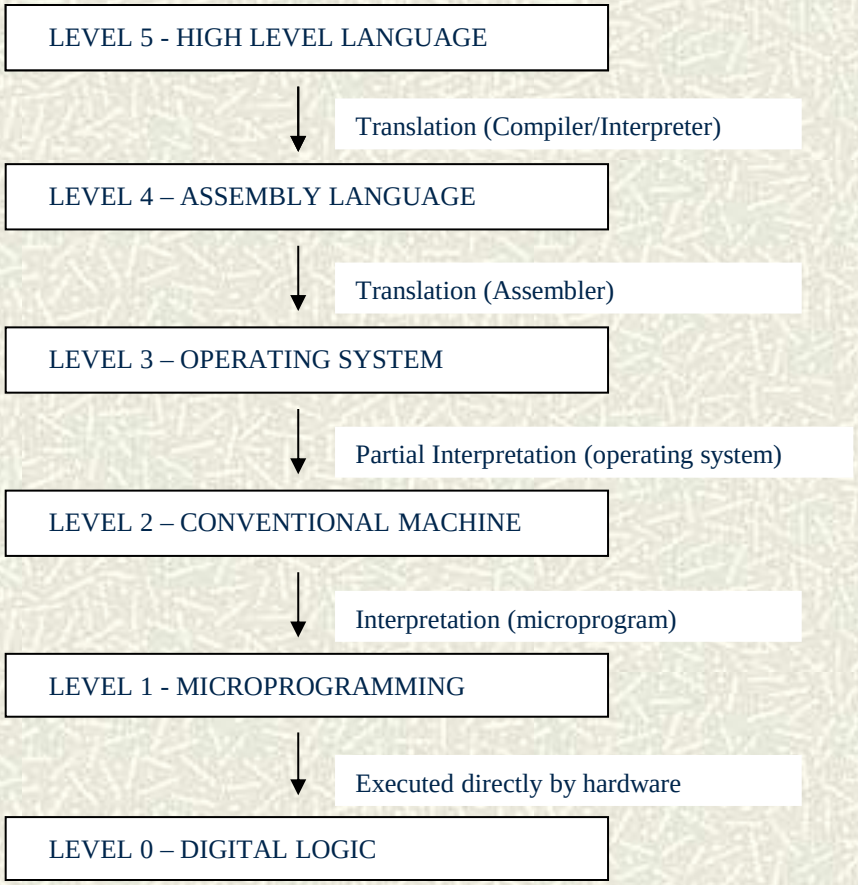
The Multi-Level View 2

Suppose we have a machine called Mark I which has a reasonable processing capability but is a nightmare to program. We then make a technological breakthrough that would enable us to make Mark I go 10 times faster. Instead of presenting the world with a faster machine that is still difficult to program, we keep back some of the increase in power and use it to make the machine easier to program by incorporating an improved set of possible instructions. Result - the world gets a Mark II that is only 8 times faster than Mark I, but which can be programmed more quickly and reliably.

The Multi-Level View 3

What we have done is effectively wrap the rather unfriendly Mark I within a user-friendly shell. The modern computer can be viewed as a collection of many such concentric shells with the von Neumann architecture at its heart. In the diagram on the next slide (taken from "*Structured Computer Organisation*" by A.S.Tanenbaum) this layered view of the computer is depicted as a series of levels, each one built upon the one below.

The Multi-Level View 4



The Multi-Level View 5

The diagram is necessarily a simplification of the real picture. For example, the operating system does not exist in a neat layer on its own, but is present in various levels of the machine from device drivers to user interface. From our point of view the internal organisation of the computer is not important. We will be concerned only with how to program the level 5 machine, but there are a couple of important points that arise from the diagram.

Firstly, each level of the machine has its own language – the language in which it can be instructed. In between the levels is an indication of how instructions to a particular level of machine are translated into instructions for the machine at the next level down. We shall examine the translation process in more detail shortly.

The Multi-Level View 6

Secondly, at every level apart from the lowest, the "machine" is a complex mixture of hardware and software components. However, when we are considering the level 5 machine, we are only concerned with what the machine can do and how we can program it. It doesn't matter to us exactly how that functionality is made possible. In Tanenbaum's words

"Hardware and Software are logically equivalent"

This is an example of ***abstraction*** - the process whereby we can clarify our thinking about something (which could be a concept, a physical system, a software requirement, etc.) by separating the relevant external characteristics of that thing from its internal details.

High-Level Languages 1

There are literally hundreds of different high level languages, and there are a variety of reasons for their existence. Sometimes new languages have been developed for particular application areas. For example, two of the earliest languages, ***COBOL*** (Common Business Oriented Language) and ***FORTRAN*** (Formula Translation), were designed respectively for data processing and scientific applications. More recent examples include ***Prolog*** (Programming in Logic), which was developed specifically for programming Artificial Intelligence solutions, and ***Java***, which is oriented towards Web applications.

High-Level Languages 2

The motivation behind the development of some other languages (e.g. the functional languages) has been the perceived need to have a sounder mathematical basis for the specification and verification of software.

Object-oriented languages (e.g. ***C++*** and Java) arose from highly developed notions of modularity in software design, whereas the visual languages (e.g. ***Visual Basic***) are recent innovations whose event-driven control structures and associated libraries are closely linked to modern windows-based user interfaces.

High-Level Languages 3

This great diversity makes it difficult to classify languages. Any particular language may be classified in a number of ways, depending on criteria such as the underlying computational framework, data and procedural abstraction capabilities, control of program execution, and so on. C++ and ***Python*** could be considered to be either procedural or object-oriented or both. (The latter also has some of the features of functional languages.) ***Visual C++*** could be called an event-driven variant of C++, but it would probably be more accurate to say that it is an entire programming environment that incorporates the C++ language.

High-Level Languages 4

High-level languages can be split into two main categories – ***declarative*** and ***imperative*** languages. These differ in three main aspects – the way in which programs are written, the way in which they are executed and the underlying computational framework that they employ.

Programs written in a declarative language are written as a collection of statements of ***what*** is true, together with an expression to be evaluated in the context of those statements. Programs written in an imperative language, on the other hand, consist of a sequence of instructions as to ***how*** to perform the required computation.

Declarative Languages 1

Most declarative languages can be categorised as either logic languages (e.g. Prolog) or functional languages (e.g. ***Lisp***, ***Miranda***, ***Haskell***).

Logic languages are based on theories of logic. The most commonly used logic language, Prolog, is based on a theory called the ***first order predicate calculus*** (FOPC), and the statements in a Prolog program are actually expressions in FOPC presented in a simpler syntactic form. These statements can be either facts (i.e. known truths) or rules.

Declarative Languages 2

For example, a program concerning family relationships may contain the fact that Mary is the parent of Joe, the fact that Joe is the parent of Anne, and the rule that a person **X** is the grandparent of person **Z** if **X** is the parent of some person **Y** and **Y** is the parent of **Z**. These statements would appear as follows.

```
parent(Mary, Joe).
parent(Joe, Anne).
grandparent(X,Z) :- parent(X,Y), parent(Y,Z).
```

Declarative Languages 3

The rules are right-to-left implications. This means that if the right hand side is true then the left hand side is also true. The comma on the right hand side means "and". So the grandparent rule reads "**if** `X` is the parent of `Y` **and** `Y` is the parent of `Z`, **then** `X` is the grandparent of `Z`". `X`, `Y` and `Z` are ***variables*** – that is they are not actual entities like Mary and Anne, but are placeholders for such entities.

Having established this context of definitions, we can then add an expression, and ask the Prolog system to evaluate it - is it true or is it false? For example, if we wanted to determine whether Mary is the grandparent of Anne, we would add the expression

```
grandparent(Mary, Anne) ?
```

Declarative Languages 4

The `?` indicates that this is the expression to be evaluated. The Prolog system first searches its collection of facts to see if it contains the fact that Mary is the grandparent of Anne. In this case, there is no such fact, but there is a rule about grandparents. The Prolog system then attempts to find suitable substitutions for the variables on the right hand side of the rule. In this case, it finds that if it substitutes Mary for `X`, Joe for `Y` and Anne for `Z`, then the right hand side of the rule is true according to the Parent facts. It can therefore conclude that, with these substitutions, the left hand side is also true – in other words Mary is the grandparent of Anne.

Declarative Languages 5

Functional languages are based on a system called the lambda calculus. This is a mathematical framework within which we can define functions and evaluate expressions containing those functions. The statements in a functional language program are functions definitions. For example, a program written in a functional language may contain the following ***pattern-directed*** definition of the factorial function

```
fac 0 = 1
fac (n+1) = n+1 * fac(n)
```

which says that the factorial of 0 is 1, and the factorial of any number `n+1` is `n+1` multiplied by the factorial of `n`. (This form of definition is known as ***recursive*** - the function `fac` has been defined in terms of itself.)

Declarative Languages 6

If we wanted to know the factorial of 3, then we would type the expression.

```
fac 3
```

The underlying system tries to match this expression with the given function definition and finds that it doesn't match the pattern `fac 0`, but it does match the pattern `fac (n+1)` if 2 is substituted for `n`. The expression can therefore be rewritten as

```
(2+1) * fac 2
```

which can be evaluated to

```
3 * fac 2
```

The process is now repeated, so the expression becomes

```
3 * (1+1) * fac 1
```

Declarative Languages 7

Arithmetic can now be performed, giving

```
6 * fac 1
```

The process is again repeated, so the expression becomes

```
6 * (0+1) * fac 0
```

which simplifies to

```
6 * fac 0
```

`fac 0` is defined to be 1, so the whole expression evaluates to

```
6 * 1
```

to yield the final answer, 6.

Imperative Languages

Imperative languages were not derived from a mathematical theory like FOPC or the lambda calculus. The statements in an imperative language program are commands to perform some action, or combination of actions, on the underlying von Neumann architecture. There are two fundamental concepts in an imperative language, ***destructive assignment*** and ***sequencing***, neither of which is present in the declarative languages.

Destructive assignment is the process whereby the contents of the memory are updated by overwriting the values held there. The updates are carried out in a specified order, or sequence, until the desired result is obtained. This sequence of destructive assignments takes the computation through a succession of ***states***.

Translation and Execution 1

On slide 18 we presented a multi-layered view of the modern computer, and noted that each level of the machine had its own language. In between each pair of levels, there was a comment about how programs were translated from the higher level language (known as the ***source*** language) to the lower level language (which we shall call the ***target*** language), by means of either ***compilation*** or ***interpretation***.

The difference between interpretation and compilation is very similar to the difference between the activities of a human interpreter (who translates spoken sentences one at a time) and a human translator (who translates an entire piece of text).

Translation and Execution 2

When a program is interpreted, each statement in the source language program is translated into possibly many statements in the target language by a program called the ***interpreter***. Each time a source language statement is translated, the generated target language statements are immediately executed on the target machine before the next source language statement is processed. Translation and execution are interleaved, and can be considered to be one combined activity.

Translation and Execution 3

When a program is compiled, the entire source language program is translated to the target language by a program called the ***compiler***. The result is a program in the target language which is then executed on the target machine. Translation and execution are completely separate activities.

Translation and Execution 4

The main advantages of using compilation are:

- translation only happens once, no matter how many times the program is executed
- execution of the program is quicker, since it is not interleaved with the translation task
- the translated program can be optimised, yielding further improvement in execution time
- programs can be compiled remotely, and the compiled code can be stored and loaded onto machines without revealing the source text to the user

Translation and Execution 5

However, the use of interpretation also has some advantages:

- interpreters can detect errors that occur during program execution (e.g. out of range values) and generate more informative error messages, since they can report the exact point in the source text at which the error occurred.
- interpretation is often used in situations where memory size is limited. The source program does not have to reside in the computer's memory, since individual statements can be read from another medium (e.g. disc or tape) and processed one at a time, whereas with compilation, the entire translated program must be stored in the computer before execution can begin.

Translation and Execution 6

The development of advanced compilers and “debugging” tools, and the advent of very large memories and disk caching techniques, have made the advantages listed on the previous slide less important, and the rate at which computer speeds have increased makes some of the advantages of compilation less significant.

Some languages use a combination of both techniques. When Java is being used the compiler converts the Java source code into a target language known as ***byte code*** which is then interpreted.

The Python Language 1

Python is a language that is easy to learn, yet has enough powerful features to develop large-scale applications. It can be used in a procedural style but also supports object-oriented and functional features. A large library is available to allow the writing of programs with graphical user interfaces, programs that access databases, network applications and internet applications. To become an expert in the use of all of these libraries takes some time, so in this module we concentrate primarily on the features that support procedural programming, ending up with material on simple graphical user interfaces.

The Python Language 2

Python is an interpreted language; this makes it easier to use for the beginner since it is possible to try out fragments of code without writing complete programs.

The development of the original version of the language was performed by a handful of programmers. Since version 2 was introduced in 2001 it has been managed by a non-commercial organisation called the Python Software Foundation and consequently versions for Windows, Linux and MacOS are all available free of charge.

The language has been growing in popularity during the last decade, being used by companies such as YouTube and Google.

The Python Language 3

Version 3, introduced in 2009, is not backward-compatible with version 2 – some of the features have been removed or changed; consequently many large users are still using version 2 to avoid having to rewrite existing code.

We shall be using version 3, since we don't have any existing code and it is likely that Python 3 will be more widely used when you are searching for jobs three years from now.

The Python Language 4

The programs you will need to write for the laboratory exercises and coursework for this module will not run correctly on any version of Python 2 so if you wish to work on your own machine you will need to ensure that version 3 is installed. (It does not matter which release you obtain, since we will not be using any of the advanced features that have changed since the introduction of version 3.0.)

Running Python Interactively 1

Since Python is an interpreted language we can run it interactively in a Python shell window. When such a window is opened a prompt (>>>) will appear beneath some introductory messages. We can simply type expressions or fragments of code in response to this prompt:

```
>>> 3+4
7
>>> 12/3
4.0
>>> (37.53/12.57)*(37.53/12.57)
8.914286202516504
```

(In the above examples user input is shown in light text, with machine output in bold).

Running Python Interactively 2

In the example on the previous page it was inconvenient to have to type `(37.53/12.57)` twice, and also inefficient to have to calculate this value twice in order to find its square. A better approach is to perform the division, store the result and then calculate the square of the result:

```
>>> i = 37.53/12.57
>>> i*i
8.914286202516504
```

The result of the division has been stored in a ***variable***, which we have chosen to call `i`. The storing of a value is a variable is known as ***assignment***. (Note that when performing assignment in the Python shell no result is output.)

A First Python Program 1

The interactive use of Python is fine for testing code fragments or evaluating expressions but if we wish to write programs that can be run more than once we need to write them in files.

Here is a first example

```
"""
hello.py
Simple greeting program
"""
print("Hello, CE151")
```

A First Python Program 2

In a Python program any lines at the beginning enclosed between a pair of lines containing """ are regarded as documentation. Tools can be used to display the documentation from the beginning of a file.

We have included the filename and a brief description in this documentation.

Other than the documentation our program contains just one line, an instruction to print the contents of a ***string*** of text. Strings in Python must appear in quotes – we can use either ' or " (but each string must have the same quote at the end as at the beginning).

A Second Python Program 1

We now wish to make the program interactive:

```
"""
hello2.py
Slightly less simple greeting program
"""
print("Hello, CE151 student")
name = input("What's your name?")
print("Welcome to Essex,", name)
```

This time we ask the user to supply his or her name – this is done by using a ***function*** called `input` with a string ***argument*** containing a prompt to be displayed on the screen. When the program is run one line of input will be read and stored in a variable called `name`.

A Second Python Program 2

Having obtained the user's name we then display it as part of a welcome message. Note that this time we have supplied two arguments to the `print` function, separated by a comma. When we do this the two items are displayed on a single line, separated by default by a single space. (Note that there's also a comma within the string to appear in the output.)

We could supply more than two arguments:

```
print("Welcome,", name, ",to Essex")
```

[Note that the way `print` is used is one of the main differences between versions 2 and 3 of Python so even our simplest examples will give different results using Python 2.]