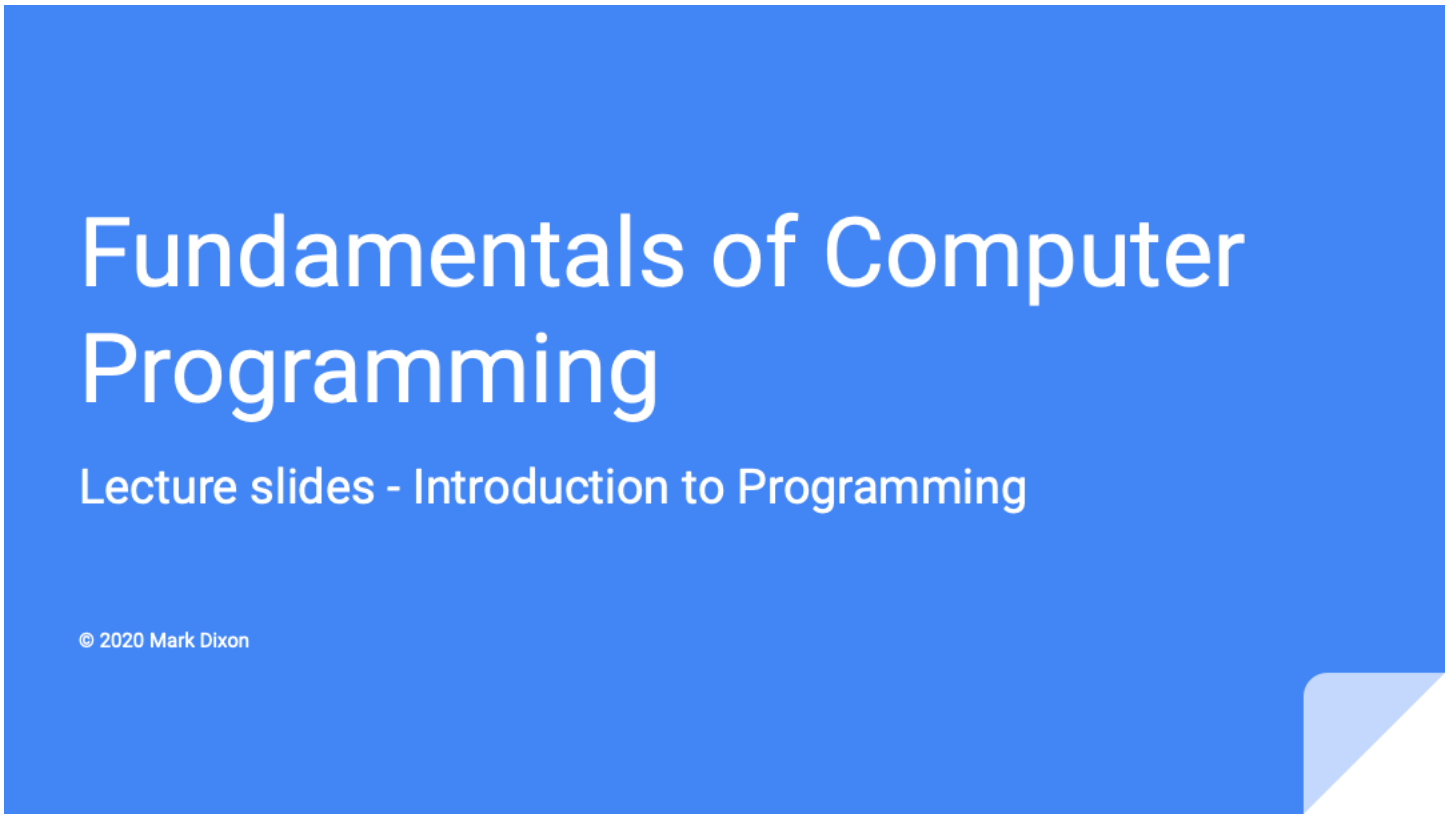
Lecture 1 - Introduction to Programming

Wednesday, 27 September 2023 10:22

8512341



Intro

- This lesson covers
  - What is a program?
  - Types of programming language
  - Overview of Python
  - The development tools
  - Expressions
  - Errors

Overview of the Module

- This module aims to introduce the art of computer programming
- We will be using the **Python** programming language, although the aim is to teach and understand the concepts applicable to all programming
- Python is one of the world's most widely used programming languages, and relatively easy to understand
- The lecture notes are accompanied by a set of step-by-step tutorials, and a number of exercises for self-testing purposes
- The tutorial, and many of the exercises will be delivered using a command line program called the **python interpreter**

What is a Computer Program?

→ useful definitions slide

- A computer program is a sequence of instructions that are executed in order to carry out a specific set of operations
- Programs take *input*, perform *processing*, and generate *output*
- Input and output (I/O) can include many sources (not just human via a keyboard or mouse, and screen), e.g. I/O can be via files only
- Programmers write the computer programs, which implement *algorithms* designed to solve a problem
- *Programming languages* allow us to write code that is easy to understand, this is then translated into a form that can be executed by a computer

↳ Designed for humans as the ideal way to implement algorithms as a sequence of instructions.

Generations of Programming Languages

- Programming languages can be divided into "generations", which have developed over time to make programming easier

changed

- Languages are usually classified as a 1GL, 2GL, 3GL, 4GL, etc.

→ Does go higher but 4GL is highest commonly used

1GL

- First generation languages involved writing directly in **machine code**
- This is basically a list of numbers that instruct a CPU to perform operations

→ Binary
→ Inputted often by a punch card

- Problem: A *CPU* has a very specific *instruction set*, that is not that easy to understand by human standards
- Machine code therefore takes a long time to write, is hard to read, difficult to *debug* and varies for different types of CPU

Second Generation Programming Languages (2GL)

Assembly language

- **Assembly** is a second generation programming language

Sometimes still used
↳ relevant when writing low level software

- Removes the need to program with numbers and introduces simple *mnemonics* that are mapped directly into CPU *machine code*

↳ short word that represents a specific instruction

Most powerful Programming language
→ you can control everything in the computer
↳ including ALL hardware

- One mnemonic usually equates to one CPU instruction, e.g.

```
add r1, r1, r2   # gets mapped to a single CPU instruction
```

- The tool that converts *mnemonics* to machine code is called an **Assembler**
- Problems:
  - Different CPUs require different assembly languages (code not portable)
  - Programming is still fairly difficult and time consuming

Different CPU - Different type of instructions

Third Generation Programming Languages (3GL)

Biggest advantage - no need for different code for different CPUs - more portable

More abstract - one instruction could be multiple in assembly language

- Third generation languages are the most commonly used, they include -

- C, COBOL, Pascal (procedural) → oldest - original languages of 3GL
- C++, Objective C, C#, Java (Object Oriented) → improvement from ^ - makes it easier to write larger programs
- Python, JavaScript, PHP (multi-paradigm - scripting type). → most modern - slightly more abstract than ^

- A **Compiler** translates these into Assembly/Machine code
  - Its own program
- Languages such as C# and Java often execute on a *virtual machine*
  - A program (often written in C) that will pretend to be a CPU, which will then execute the code
- Scripting type languages are often **Interpreted**, or sometimes use Just in Time (JIT) compilation
  - Program is compiled to machine code but only as a specific bit is run.
  - Don't get converted to assembly language - when executed code is interpreted to run, rather than compiling for a specific CPU.

Fourth Generation Programming Languages (4GL)

Rather than specifying algorithms in a 4th generation, you define what is wanted to be done

- this is why it is higher level
- Fourth generation languages are more abstract from the underlying machine, and are based on defining *what* is to be done, rather than *how*
- These type of languages tend to be less general purpose, and more focussed on a particular problem domain, well known 4GLs include,
  - SQL (data manipulation)
  - R (statistics and data-science)
    - modern
  - SPSS (statistical analysis)
- Although 4GLs are seen as more advanced than 3GLs, they are not a direct replacement, especially for general purpose problems

Take what has to be done and work out how to do it.

The high level of abstraction equates to a loss of some control.

Python - A Popular 3GL

- Python is easier to learn than many other languages and is widely used in education and industry → getting increasingly more popular
- It supports the popular *procedural*, *functional* and *object-oriented* (OO) styles of programming — mainly
- It can be used to build anything from simple programs (*snippets*) to complex applications
- Although general purpose, it is popular in many specific areas such as artificial intelligence, data science, forensics, and security → Cumbe used for pretty much anything
- It is a good starting point before moving on to learn other languages

Command Line Python and IDEs

- Initially we will be using a *command-line Interpreter* for learning Python
- You will have to learn the basics of how to use the Interpreter, but what you are really trying to learn is how to program in Python
- Later on within the module you will write programs using a *text editor* then execute the program files using the interpreter
- Other environments such as *Jupyter Notebook*, along with *IPython* are often

used during teaching (and may be referred to in books etc.)

- Don't get confused between the various environments and the language

The "Python" Interpreter

- The python interpreter can be used in **interactive-mode** that allows you to write and execute Python code immediately - no need to make a file etc
- It is based on a *Read–Eval–Print Loop* (**REPL**) interaction model
  (Read = typing in; Eval = works out; Print = shows result; Loop = lets you go again)
- The command line interpreter has a number of features that make experimenting with Python easy, including -
  - An in-built help system (for both itself and Python)
  - Auto-completion while typing
  - Running and debugging
  - Command history

Python Interpreter - example screen

```
>>> print("Hello from Python")
Hello from Python
>>>
>>> 21 * 3
63
```

```
>>> 10 / 3
3.3333333333333335
>>>
```

- In the above example several commands have been *interactively* input (following the '>>>' input prompt)
- Once <return> is hit the code is ***Read***, ***Evaluated***, and output is ***Printed***
- The 'up' and 'down' arrow keys can be used to recall and allow editing of previously input instructions

Works like a command line CMD etc.

Integrated Development Environment

→ Reason why they are popular
- Support most languages
- has many tools - text editor, version control, debugger
- much more popular than CMD

- IDEs are popular environments for developing software (support many other languages as well as Python)
- Typically provide a text editor (with *syntax* highlighting), debuggers, version control, help, etc.
- IDEs are much easier than using the Python interpreter when developing large applications
- IDEs can be a complex to use, given their large amount of functionality
- Used by many software houses, so a useful skill to develop in the future

Software Resources

- All of the software we use on this module is freely available
- These lecture notes (along with the associated tutorials and exercises) will be made available for download for use away from the University
- Be careful when downloading Python or looking at resources.
- Python 3.x is quite different to version 2.7 and earlier
  - ↳ popular but not used for this module
- The provided notes will assume you are using Python version 3.7 or later.
- Any text editor can be used to develop programs later in the module

Python Libraries

- Python programs rely heavily on **libraries** (which contain pre-written code)
- They often perform significant tasks with modest amounts of code, and help avoid "reinventing the wheel" → Find already done code to reuse
- The Python Standard Library itself provides lots of rich capabilities, but many additional libraries are also available
  - → Large - can do most things with it. Comes by default with every python install.
- This module will focus primarily upon the **language** rather than the **libraries**, but we will use parts of the standard library within the module
- Python itself is a fairly small language, so knowing how to find and use 3rd → often the difficult bit

Libraries are prewritten code from oth
programmers over the years. These
packaged up and made available. Th
big tasks to be done with smaller am
code.

party libraries is key to development e.g. the Visualization libraries are extremely powerful and popular with data-scientists

There will be a library for everything

Expressions within Python

- An easy way to start programming, and becoming accustomed with the tools, is to write **expressions** Calculations needed to be done with code
- Expressions are used in just about every program you will ever write
- Expressions consist of *operands* and *operators*, e.g.

```
45 + 20
```

- In this example the *operands* are the numbers **45** and **20** and the *operator* is the **+**
- Typing an expression into the python interpreter, causes the result to be immediately displayed (note: this only occurs in interactive mode)

When typing in an IDE or text editor you need to use the 'print' command to get the answer displayed

Common Python Operators

- Other common operators used within Python expressions include -

  ** (exponentiation), i.e. raise to the power

*	(multiplication)
/	(division)
//	(floor division) → Division that removes decimal place instead result
%	(remainder) → what's left over from a division
-	(subtraction)

Operator Precedence

- Expression evaluation is not always performed left to right, there is an *operator precedence* at work, e.g. multiplication occurs before addition -

```
>>> 10 + 5 * 2
20
```

- The *precedence* of the common operators we have seen so far is -

parentheses ( )
exponentiation (**)
multiplication (*), division (/), floor division (//) and remainder (%)
addition (+) and subtraction (-)

- Operators at the same level are evaluated left to right

Matches BIDMAS (ish) from school

Errors

- When writing computer programs errors are inevitable
- The ability to find and fix errors is a key skill required by any programmer
- Errors are generally categorized as **syntax errors** or **logical errors**
- Syntax errors are usually the most common, but also the most easy to fix
  - These occur when the programming language syntax has not been used correctly

Grammar / formatting is incorrect

- Logical errors occur during run-time, and can be hard to find and fix
  - These occur when a program is syntactically correct, but the underlying algorithm was incorrectly designed or poorly implemented
  - These type of errors are commonly referred to as "bugs"

Dealing with Errors

- People new to programming spend most of their time fixing **syntax errors**
- The development tools will report these (often prior to program execution), so ensure you read the error report and then fix the error
- **Logical errors** do not always result in an error being reported, and have to be identified through *testing*
- Occasionally errors will be reported at *run-time*, and displayed as a "*traceback*" message. Although they may seem cryptic these allow the problem to be identified and fixed

→ shows the part of the code the problem is with
→ seem complex but aren't bad

- A good program will be written in a way that prevents such *run-time* errors

IMPORTANT

occuring, or reports them gracefully to the user

↳ Error messages etc

ALWAYS READ ALL ERROR MESSAGES

Summary

- A **program** instructs a computer to perform specific operations
- There are many types of *programming language*, we will be using **Python** which is an interpreted 3GL
- We will be initially programming in **interactive-mode** to allow experimentation and rapid feedback of results
- **Libraries** are very useful and save a lot of time, but only start to use these once you understand the **language** itself
- Writing **expressions** and dealing with common **errors** is a good place to start - *tip:* ALWAYS READ ERROR MESSAGES

Useful Resources

- The Python web-site

  https://www.python.org/

- The Python 3.x tutorial

https://docs.python.org/3/tutorial/