INF1-IP

Introduction to Programming

Lecture 1 – Algorithmic Thinking

> What is programming for?

Communicating between *people* and *machines*

Can't AI do this for us?

- Increasingly, yes
- But it is still really important to understand what AI is doing in order to work with it effectively
- You learn this by doing

> What is programming for?

Computers are *binary* – they work with 0s and 1s
It is hard for humans to communicate in this way!

In 1946, Kathleen Booth invented *Assembly* – represented mac
code using short words instead of raw binary numbers

In the 50s, high-level languages like FORTRAN and LISP starte
used

```
[ High-Level Code ]  ->  [ Assembly Code ]  ->  [ Machine Code ]
  total = x + y            ADD EAX, EBX           00000011 11011000
```

> What is programming for?

Computer scientists used to argue about whether or not all computer scientists should understand Assembly

Nowadays, most computer scientists don't understand Assembly – but some still do and it's essential in some areas. Now we argue about whether people need to learn things like Python

It's unclear how AI will shift programming, but *algorithmic* and *computational* thinking will always underpin computer science. For now, we teach this through existing languages. This gives you the tools to access whatever will be used in the future.

It's not about the language, it's about the thinking.

> How does programming work?

Is it about getting good enough to stop making mistakes?

- No!!

However good you get, finding and fixing errors will always be a big part of your coding.

AI can be really helpful for this. But it's not good at doing this at a deeper level – it makes lots of errors of its own.

That's why we need to have *skilled humans* working *with* AI

Programming can be *massively frustrating.*

But it's also *massively rewarding.*

If you get stuck trying to make your program work, you're not a *bad* programmer. You're just a programmer.

If you enjoy the process of figuring out what's wrong, working to fix it and build your skills so that next time you get stuck on a harder problem, you might one day become a *good* programmer.

> Teaching coding through PRIMM

Predict: before running the code, look at it - do you understand each line? What do you think will happen?

Run: now run the code and see what output you get.

Investigate: did it go as you thought it would? If not, can you figure out why?

Modify: now change the code and once again try to predict the output. If you got it wrong the first time, did you get it right now?

Make: finally, try to write your own program from scratch using the skills you learned.

> Using algorithms

Code like print(“Hello world”) does not contain an algorithm.
There is no *logic* in it.

An algorithm is *the application of logic to solve a problem*.

To solve a problem, we:

- Create an algorithm by thinking about applying logic to solve the problem
- Translate the algorithm into Python

> Using algorithms

Good code is written so that the logical process is clear

- Anyone reading the code can figure out what the logic is
- Easy for small problems – not always easy for larger pieces of code

If we are coding with AI:

- We may be able to use AI to figure out the algorithm and write the code, but
- We should be able identify and understand the algorithm that it's using
- Working with AI is about updating and perfecting the algorithm it's created to make sure it is working effectively and correctly

> Using algorithms

Good code is written so that the logical process is clear

- Anyone reading the code can figure out what the logic is

But how do we do this?

We use **decomposition** to break a complex problem down into smaller problems. This makes the logic *much clearer*. We will work on this later in the course, when we are building more complex programs.

We should also use **comments** to explain to readers what we are doing

- In Python, comment lines start with #

> But why do we care about the logic?

If you are 100% sure your code works in all conceivable cases, and you *never want to use it again*, you can ignore the logic.

But code is almost never used in isolation. It is reused, build upon, shared, updated, altered, etc. Doing any of these things is very difficult if the logic is not clear

You will even find it hard to understand your own code if you revisit after a few weeks, if the logic is not clear and it is not well commented.

Thorough testing can demonstrate that code is correct, but if testing finds bugs (it usually does!) then finding and correcting these bugs is really hard if the logic is not clear.

> Input, Process, Output

In our small example, we took something in, we did something with it, and we printed something out.

This is the way a lot of programs work:

1. **Input**: obtain information from the user;
2. **Process:** transform or combine that information, often by implementing some kind of algorithm;
3. **Output:** communicate the result back to the user.

> Pseudocode

Human language that *looks like* code:

- It is precise
- Ordered logically

But it doesn't use the syntax of any language in particular.
Very useful for formulating and sharing ideas of an algorithm without limiting to a particular implementation.

ASK the user for their name
STORE a ticket price and quantity
CALCULATE price multiplied by quantity
DISPLAY the name and calculated cost