NPA Data Science:

Computer Programming 1

michaelferrie@edinburghcollege.ac.uk

Part 1: Introductory Concepts

Computer science has been around for less than a century and so in comparison to subjects like mathematics or art, it is in its infancy. Computer Science is actually quite a bad name for the subject as it's not really a science - there are no test tubes or measuring instruments, it's also not really about computers.

As a computer scientist you can use a computer to write programs but you do not need to know how the computer was built. When writing programs it is important to try and not confuse what you are trying to do with the instrument you are using. Much like biology isn't really about microscopes, computer science is not about computers, but about how you can make a computer follow a set of instructions.

1.1 This leads to the question - what can a computer do? At the simplest level a computer can only perform two tasks:

1) Perform a calculation
2) Store the result

This seems very simple but it is true, that's all a computer can do. In this course it is important that we establish the correct methods of talking about what we are doing, once we can give the correct names to things they become much easier to discuss.

There are two types of knowledge that we are concerned with:

1) Imperative Knowledge
2) Declarative knowledge

Imperative knowledge is knowledge of how-to methods and **declarative knowledge** is knowledge of fact. Consider this example - you have been asked to solve a problem (an easy one) to make a cup of tea, here is a recipe for the task. When writing an algorithm we do not tend to number our steps, since we often need to revise our algorithms and numbers can get in the way of this.

> Put water into the kettle
>
> Once boiled pour water into cup
>
> Add 1 tea bag
>
> Remove tea bag

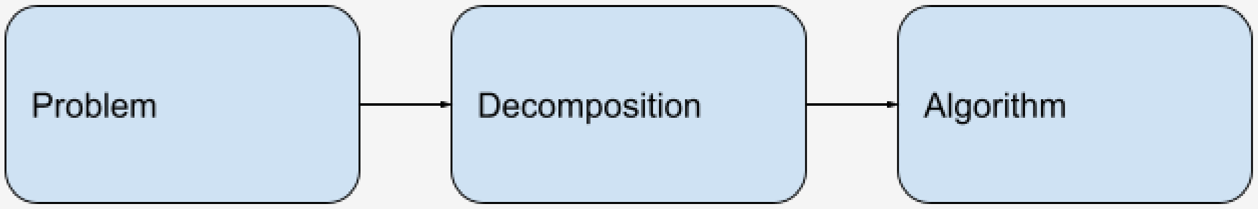
If you had never made a cup of tea before could you use those steps to complete the task?

Dividing this up into two different types of knowledge, there are ‘how-to’ steps which are imperative and facts which are declarative.

For example the water, kettle, tea bag, sugar - we do not have to explain what these are, we can take these as fact - however things like pressing a button, and pouring water and removing tea bags are imperative steps, they are how-to knowledge.

1.2 An algorithm can be thought of as a recipe, it is a list of steps to follow to achieve a result, just like the making tea example above.

Breaking a problem down into steps is called problem **decomposition**. The decomposed set of steps creates an algorithm.



Stepwise Refinement

When a decomposition is broken down into steps this is one of the oldest methods still used today in computer programming and is often called **Stepwise refinement**. Stepwise refinement refers to the progressive refinement in small steps of a program specification into a program. Sometimes, it is called top-down design. The term stepwise refinement was used first in the paper titled Program Development by Stepwise Refinement by Niklaus Wirth, the author of the programming language Pascal and other major contributions to software design and software engineering, in the Communications of the ACM, Vol. 14 (4), 1971, pp. 221-227.

Wirth said, "It is here considered as a sequence of design decisions concerning the decomposition of tasks into subtasks and of data into data structures." We use the term to describe the development of a method from its specification. We will see small steps, like breaking a high-level statement into a sequence of statements.

1.3 Let’s revisit the making tea algorithm and refine the steps into a more detailed decomposition. Our first line uses a ‘>’ symbol, which shows that the first line is a heading. When we refine a step into a more detailed decomposition we can use >>.

```
> Put water into the kettle
        >> Turn power on by pressing power button
        >> Wait for boiling to complete
> Once boiled pour water into cup
> Add 1 tea bag
        >> Add milk or sugar as required
        >> stri tea
> Remove tea bag
        >> leave for certain time to increase or decrease strength of tea
```

Guess and check algorithm

Consider this next algorithm for calculating the square root of a number, this is called a **guess and check** method to try and find the square root of a number, in this example we will try to find the square root of 25, we know this is 5 because 5*5=25 but let's take a guess that it is to work it out.

First the algorithm written as a decomposition:

```
> take a number n
> take a guess g
> multiply g and g
> check result
```

Now refine the decomposition into :

```
    > take a number n
    > take a guess g
        >> is g* g = stop, you have the solution
    > if g * g is not = n update guess to a new guess
    > multiply new guess by new guess again to check if it equals n
        >> return to previous step until g*g = n
```

Now run through the algorithm and try to find the square root of 25.

1. Guess the square root of 25 - guess is 3
2. Multiply 3 by 3 and see if this is 25
3. Answer is 9 - too low, take a higher guess
4. Guessing that the answer is 4
5. Multiply 4 by 4 and see if this is 25
6. Answer is too low take a higher guess… getting fed up now let's go higher - guess 6
7. Multiply 6 by 6 and the answer is 36 - that's too high take another guess
8. Guessing 5
9. Multiply 5 by 5 and the answer is 25 - thats it
10. The square root of 25 is 5

That's great - now try with the calculator on your phone, use the guess and check method to calculate the square root of 1600? - remember to start with a guess, then follow the steps until you get the right answer.

Example Algorithm: Peano Addition

Created by Italian mathematician Giuseppe Peano (1858–1932). The sum of two numbers x and y, if x is zero then y is the answer. Otherwise the answer is the sum of the decrement of the first number and the increment of the second.

Here is the algorithm written out in decomposition notation, as you can see it is very simple:

```
> take two numbers x and y
> if x is zero then y is the answer
    >> decrement x by 1 and increment y by 1 until x is zero
```

Here is an example using 3 as x and 4 as y:

3+4 (3 is not zero so 4 is not the answer)

```
3-1 = 2   and  4+1 = 5

2-1 = 1   and  5+1 = 6

1-1 = 0   and  6+1 = 7
```

7 is the answer because the x is now zero.

If and for

A couple of keywords we can use when describing an algorithm are if and for, we can use for if we want to talk about a collection of data or some object and we can use if to introduce a condition.

Here is an example of an algorithm to count steps, this could be used in a phone pedometer or a similar device.

```
> step counter starts at zero for the day
> for each step add 1
```

Now introduce a conditional statement to the algorithm:

```
> step counter starts at zero for the day
> for each step add 1
```

>> if counter reaches 10000

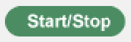
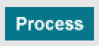
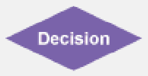
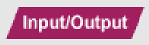
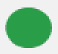
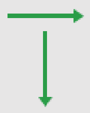
>> restaurant message to user saying they have reached their step goal

Part 2: Flowcharts

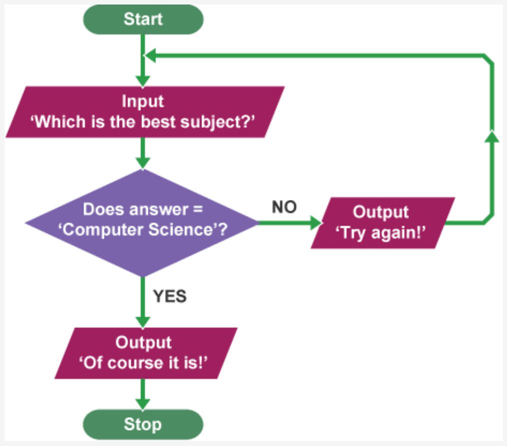
One way of representing algorithms is to use flow charts, also called flow diagrams. They are a useful way of planning how a computer program might work, and show others your thinking. A flow chart shows the key points in an algorithm:

1. the start and end
2. the order in which the sequences of instructions are performed
3. the points where inputs and outputs occur
4. the points where decisions are made about what to do next
5. A sequence of many instructions that does not involve any of these key points may be represented as a single rectangular box.

Flow charts use a variety of standard flowchart symbols to represent different elements, and arrows to show the flow or direction. These shapes are formally agreed standards (British Standard BS4058). The most commonly used symbols are:

Name	Symbol	Usage
Start or Stop		The beginning and end points in the sequence.
Process		An instruction or a command.
Decision		A decision, either yes or no. For example, a decision based on temperature that turns a central heating system on or off.
Input or output		An input is data received by a computer. An output is a signal or data sent from a computer.
Connector		A jump from one point in the sequence to another.
Direction of flow		Connects the symbols. The arrow indicates direction.

Flowcharts can be used to plan out programs. Planning a program that asks people what the best subject they take is, would look like this as a flowchart:



Making tea again - updated version

Let's return to the earlier example of making tea and this time we will take our algorithm and represent it with a flow chart:

> Put water into the kettle

>> Turn power on by pressing power button

>> Wait for boiling to complete

> Once boiled pour water into cup

> Add 1 tea bag

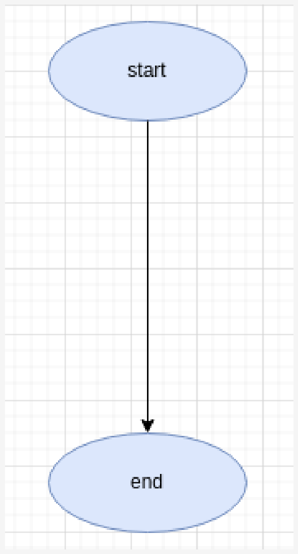
>> Add milk or sugar as required

>> stri tea

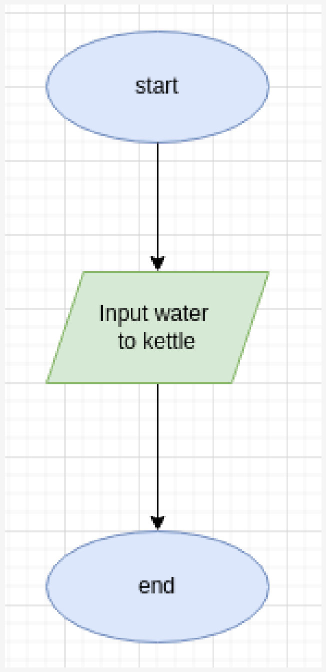
> Remove tea bag

>> leave for certain time to increase or decrease strength of tea

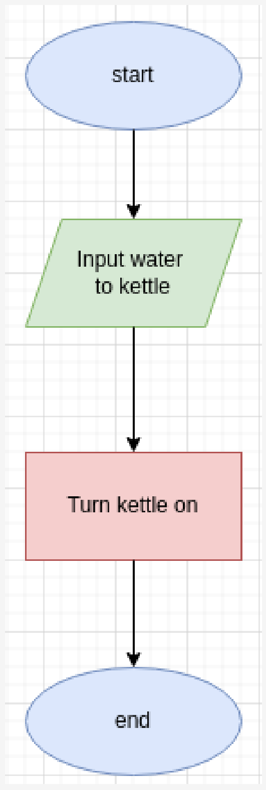
Always start and end with an oval shape called a terminator. So far not the most exciting program:



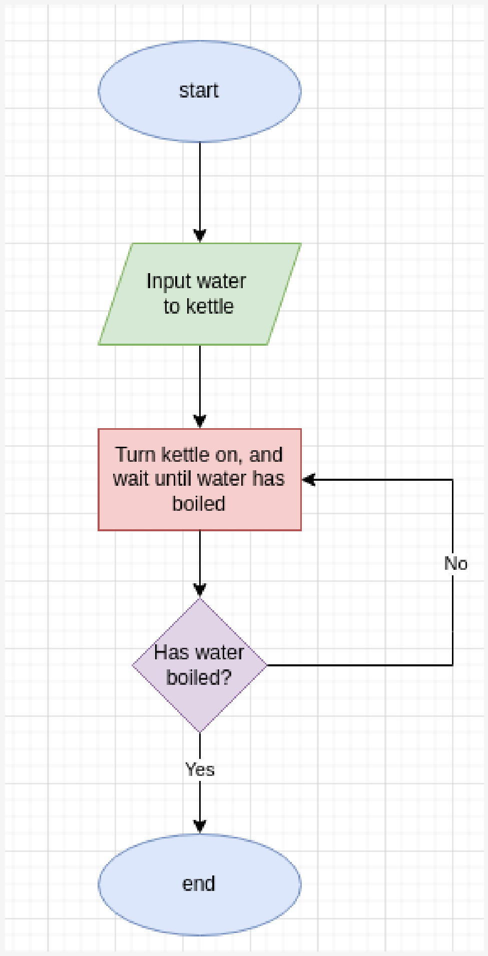
Now we need to add some input, any that is input or output from the program is represented by a parallelogram:



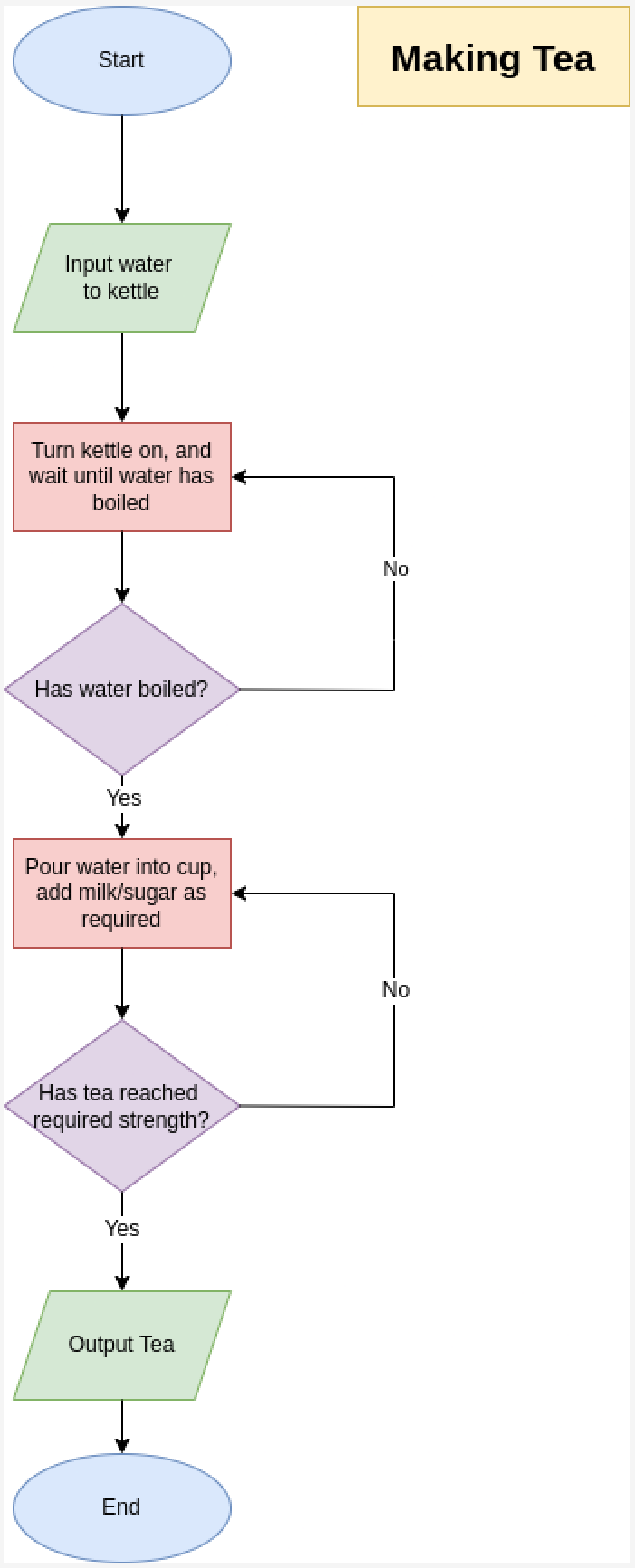
Now we want to create a process, these represented with a rectangle:



Now we need to add a decision, in a programming language this might be a conditional statement:



For brevity, we will not add every single step in this fashion but make sure you are aware of these main shapes, a final flowchart would look like this for our algorithm:


Making Tea
Start
Input water to kettle
Turn kettle on, and wait until water has boiled
Has water boiled?
No
Yes
Pour water into cup, add milk/sugar as required
Has tea reached required strength?
No
Yes
Output Tea
End