OCR GCSE Computer Science (J277)

Revision Guide

Covering the 9 – 1 specification

Use this revision guide to help you revise the theory you learn in lessons, and practice the exam questions at the end of each topic

Paper 2

Contents:

"I'm your guide to revision! I'll be pointing out key things to remember across different parts of each topic. Look out for me!"

All technical terms and definitions can be found at the start of each section. All technical terms within each topic are in red

Look out for opportunities to answer exam style questions on specific aspects of each topic

2.1 Algorithms

In this section you will revise the following:

2.1.1 Computational Thinking

- Principles of computational thinking:
  - Abstraction
  - Decomposition
  - Algorithmic Thinking

2.1.2 Designing, Creating and Refining Algorithms

- Identify the inputs, processes, and outputs for a problem
- Structure diagrams
- Create, interpret, correct, complete, and refine algorithms using:
  - Pseudocode
  - Flowcharts
  - Reference language/high-level programming language
- Identify common errors
- Trace tables

2.1.3 Searching and Sorting Algorithms

- Standard Searching Algorithms:
  - Binary Search
  - Linear Search
- Standard Sorting Algorithms:
  - Bubble Sort
  - Merge Sort
  - Insertion Sort

Technical Terms

Technical Term	Definition
Algorithm	A set of instructions that are all in order to complete a task.
Computational Thinking	Thinking critically and logically when solving a problem. Being able to analytically solve a problem.
Abstraction	Removing the unnecessary parts of a problem so that you only focus on the necessary/important parts.
Decomposition	Breaking a complex problem down into smaller problems so that it is more manageable/easier to solve.
Algorithmic Thinking	The process of building a solution to a problem. Creating a set of instruction in order to solve a problem.
Searching Algorithm	A type of algorithm used to search through a data set to find a specific piece of data.
Binary Search	A type of searching algorithm. In order to use a binary search the data set must be in order.
Linear Search	A type of searching algorithm. The data set does not need to be in order when using a linear search.
Sorting Algorithm	A type of algorithm used to sort a data set into a specific order.
Bubble Sort	A type of algorithm used to sort a data set into a specific order. The data set is passed through, and two pieces of data are looked at in turn. The process does not stop until a pass is completed without moving any data.
Merge Sort	A type of algorithm used to sort a data set into a specific order. The data set it broken up into pairs. Each pair is reorganised in turn. Pairs are then merged together and reorganised. This process is repeated until the whole data set is merged and reorganised.
Insertion Sort	A type of algorithm used to sort a data set into a specific order. A new, temporary list is created, and each piece of data is placed into the correct place in the new list.
Pseudocode	A midpoint between programming syntax and written language. It is not syntax specific therefore can easily be converted to programming code in any language.

Flow Diagrams	A way of representing an algorithm using shapes. All shapes must always connect up creating a flow throughout the diagram.
Terminator (Flow Diagrams)	Used to start and stop a flow diagram.
Process (Flow Diagrams)	Used to give an instruction in a flow diagram.
Decision (Flow Diagrams)	Used to ask a question in a flow diagram. It provides two possible choices (Yes/No).
Data (Flow Diagrams)	Used to show input/output within a flow diagram.
Sub-Program (Flow Diagrams)	Used to call a sub-program within a flow diagram. The sub-program should have been created as a separate flow diagram, which is then called in the main flow diagram.
Trace Table	Used to test an algorithm for logic errors by simulating the flow of execution.

What is an Algorithm?

Before we go into this topic discussing various aspects and elements of an algorithm, it is important we first understand what an algorithm is.

An algorithm is a set of instructions, all placed in order, to complete a task. We often think of just computing when we think of algorithms, but we use algorithms in our everyday lives!

For example, a recipe is an example of an algorithm. In a recipe you are given a series of tasks to complete (e.g. Step 1 – get together the ingredients. Step 2 – mix the flour and butter together etc.) Imagine you weren't given these instructions? Imagine the recipe simply said 'bake a cake'. We wouldn't know what to do and when! This is where our algorithm helps us. It gives us lots of step by step instructions that allow us to complete the task (in this case, bake a cake).

However, of course we need to be thinking of algorithms in terms of computing. We can represent an algorithm in three different ways (we will discuss each of these more later on):

- Plain English
- Pseudocode
- Flow Diagram

Writing an algorithm in computing allows us to produce a list of instructions that could then be converted into a computer program. It lets us plan out how the program will work, how it will respond to different inputs and outputs, and how the user will interact with it.

"In the exam, it's really important to read the algorithm questions carefully. If it asks you to write an algorithm using pseudocode for example, you must write pseudocode. If it asks you to write an algorithm, then you are able to pick whichever way you want to represent it!"

Computational Thinking

Computational thinking is all about being able to think logically and critically. We should look at each problem we face, analyse it, and produce the most efficient and effective solution. When thinking computationally about a problem, we go through three thought processes. These are:

- Abstraction
- Decomposition
- Algorithmic Thinking

Abstraction:

Abstraction is the process of removing any unnecessary elements to a problem, so you only focus on what is necessary and important. This makes a system more efficient as it removes data that is not needed.

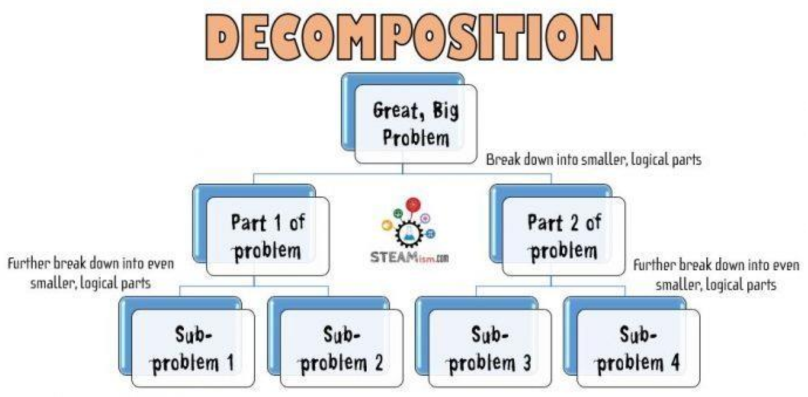
For example, a school system storing data on staff and students would use abstraction. Let's take the teachers and the catering staff for instance. The type of data teachers would need to see on a student could be their name, grades, effort and behaviour levels, and if they require any additional support in the classroom. Teachers would not need to know other pieces of data such as dietary requirements or how much money each student has on their catering account.

Now let's take the catering staff. They would be pretty much the opposite! Catering staff wouldn't need to know what grades a student has, or their effort or behavioural levels. They would need to know if a student has any dietary requirements, or any allergies. They would also need to know how much money a student has left on their catering account, to ensure the student can pay for what they order.

As you can see above, all this data is stored on the one system, but the system has been abstracted to suit different people. Reducing the amount of data the catering staff see for example makes it easier for them to understand, and also reduced load time. It means they aren't looking through various pieces of unnecessary data to find what they need.

Decomposition:

Decomposition is the process of breaking a large problem down into lots of smaller problems. This makes the process of creating a solution easier to manage. Often when we mention the work decomposition in Computing people think of the meaning in Science. It is relatively similar, expect in this case something is not rotting!


DECOMPOSITION
Great, Big Problem
Break down into smaller, logical parts
Part 1 of problem
STEAMism.com
Part 2 of problem
Further break down into even smaller, logical parts
Further break down into even smaller, logical parts
Sub-problem 1
Sub-problem 2
Sub-problem 3
Sub-problem 4

Let's go back to our recipe example again, and baking a cake. Imagine someone came up to you and simply said to you "bake me a cake". Where would you start? What would you do first? This task could be overwhelming!

What we would therefore do is decompose the task. We would take our big task of baking a cake, and break it down into smaller, more manageable tasks. For example, we would need to get together our ingredients. We would need to make a sponge for the cake. We would need to make a filling for the cake. We might even make the icing for the cake too. Now, rather than having a big task to complete, we have four smaller tasks to complete, making it more manageable for us to cope with.

This idea would be the same in computing too. Someone would approach a programmer and say what they want their program to do. The programmer would then decompose the task so that it was a series of smaller tasks, before going ahead and tackling the problem.

Algorithmic Thinking:

Algorithmic Thinking is the process of building a solution to solve a problem, where each of the stages/instructions are all in order. It's really important at this stage our instructions go in order, otherwise our solution will not work the way we want it to!

Let's go back again to baking our cake. Now we have decomposed the task and have smaller, more manageable tasks to solve, we can now begin to build our instructions to bake the cake. For example, the task of making a sponge. For this we might say:

Step 1 – Beat the sugar and butter together until fluffy

Step 2 – Whisk in the eggs one at a time
Step 3 – Fold in the flour

And so on! Here we now have a list of instructions, all in order, to complete this task.

Again, this would be the same in computing. Once the programmer had a series of smaller, more manageable tasks to complete, they would begin creating a series of steps for each task. For example, one small task might be to allow a user to log into a system. For this they might do the following:

Step 1 – Request user to input username
Step 2 – Request user to input password
Step 3 – Is username and password correct?
Step 4 – If yes, output message 'Log in successful'
Step 5 – If no, output message 'Invalid credentials'

"In the exam it is common for you to be asked for a definition for one or more of these. Make sure you learn them, as they're easy marks to pick up!"

Standard Searching Algorithms

In this section we are going to focus on searching algorithms.

A searching algorithm is used to look for and find a specific piece of data within a data set (multiple pieces of data). There are two searching algorithms we need to know:

- Binary Search
- Linear Search

It is imperative that you show your working out when answering questions about searching (and sorting) algorithms in the exam. Simply writing out the answer will not access any marks.

Binary Search:

In a binary search we locate the midpoint within our data set, removing irrelevant halves of our data set until we find out piece of data.

Very importantly, when undertaking a binary search, the data set **MUST** be in order!

Example 1:

Using a binary search locate the number 26 in the below data set. You must show your working:

5	10	26	56	68	75	97

Firstly we are going to write under each number its index in the data set. Remember, this data set is effectively a list, so the first index must be 0:

5	10	26	56	68	75	97
0	1	2	3	4	5	6

Secondly we are going to work out the midpoint of the list. This will let decide which side the data we are looking for is on, and remove the unnecessary half of the list. To work out the midpoint we add together the first index in the list and the last index in the list. We then divide that answer by 2:

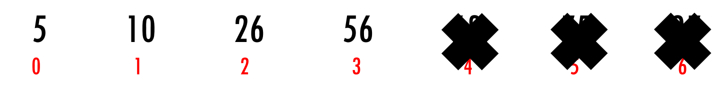
5	10	26	56	68	75	97
0	1	2	3	4	5	6
↑						↑

In this case it is:

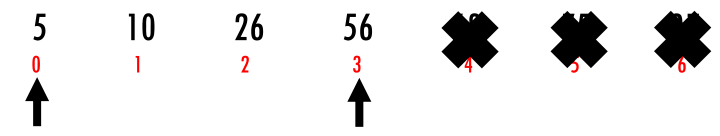
0 + 6 = 6
6 / 2 = 3

We now look at the data inside that index in the list. We firstly see if that is the data we are looking for. In this case we have found 56, which is not the number we are looking for.

We therefore now decide which side of 56 the data we are looking for (26) must be on. As we know the list is going from smallest to largest, 26 must be to the left of 56 as it is smaller. This means we know all the pieces of data to the right of 56 must be unnecessary and therefore we can remove them:



Thirdly, we now repeat the process again. We firstly find the midpoint of the new, shorter list:

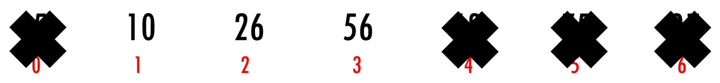


In this case it is:

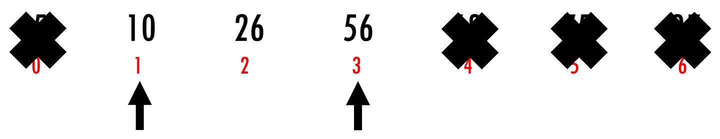
0 + 3 = 3
3 / 2 = 1.5

The above has now left us looking in the index 1.5 for some data. There is no index with 1.5 as we don't have indices with decimal places. Therefore, we need to get a whole number. We therefore are going to round down. This is because if we round up we would never be able to locate data if it was in index 0. We therefore look at the data inside index 1. In this case we have found 10, which is not the number we are looking for.

We therefore now decide which side of 10 the data we are looking for (26) must be on. As we know the list is going from smallest to largest, 26 must be to the right of 10 as it is bigger. This means we know all the pieces of data to the left of 10 must be unnecessary and therefore we can remove them:



Fourthly, we again repeat the process now with our even smaller list. Again, let's find the midpoint:



In this case it is:

1 + 3 = 4
4 / 2 = 2

Again we now look at the data inside index 2 to see if that is the data we are looking for. In this case it is, and we have found 26.

It took us, in total, 3 searches before we found the data we were looking for.

And that is it! As long as you show that working out in the exam of working out the midpoints, removing unnecessary parts of the data set, and repeating the process until you find the specific piece of data you are looking for, then you will access all the marks!

<u>Example 2:</u>

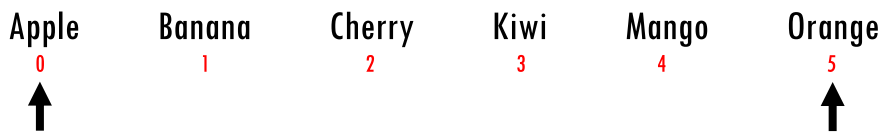
Using a binary search locate the work Mango in the below data set. You must show your working:

Apple	Banana	Cherry	Kiwi	Mango	Orange

Firstly we are going to write under each word its index in the data set. Remember, this data set is effectively a list, so the first index must be 0:

Apple	Banana	Cherry	Kiwi	Mango	Orange
0	1	2	3	4	5

Secondly we are going to work out the midpoint of the list. This will let decide which side the data we are looking for is on, and remove the unnecessary half of the list. To work out the midpoint we add together the first index in the list and the last index in the list. We then divide that answer by 2:

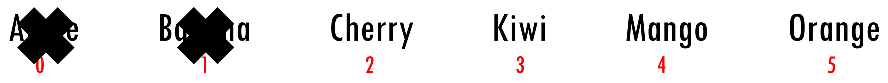


In this case it is:

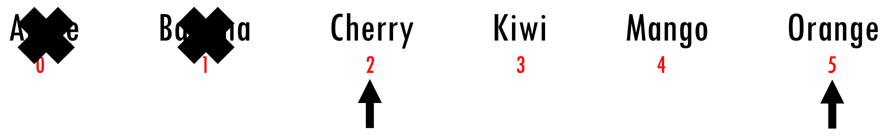
0 + 5 = 5
5 / 2 = 2.5

Remember, the index 2.5 does not exist so we round down to 2. We now look at the data inside that index in the list. We firstly see if that is the data we are looking for. In this case we have found Cherry, which is not the word we are looking for.

We therefore now decide which side of Cherry the data we are looking for (Mango) must be on. As we know the list is in alphabetical order, Mango must be to the right of Cherry as it comes later in the alphabet. This means we know all the pieces of data to the left of Cherry must be unnecessary and therefore we can remove them:



Thirdly, we now repeat the process again. We firstly find the midpoint of the new, shorter list:

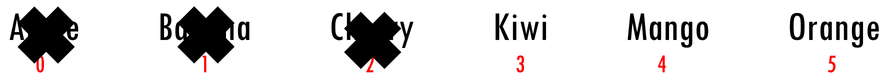


In this case it is:

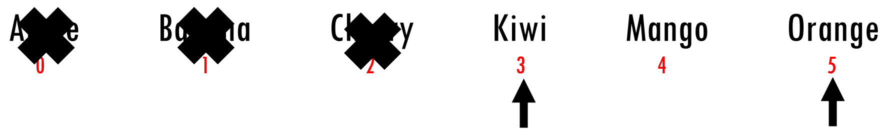
2 + 5 = 7
7 / 2 = 3.5

Again, we can't use an index of 3.5, so we round down to 3. We firstly see if that is the data we are looking for. In this case we have found Kiwi, which is not the word we are looking for.

We therefore now decide which side of Kiwi the data we are looking for (Mango) must be on. As we know the list is in alphabetical order, Mango must be to the right of Kiwi as it comes later in the alphabet. This means we know all the pieces of data to the left of Kiwi must be unnecessary and therefore we can remove them:



Fourthly, we again repeat the process now with our even smaller list. Again, let's find the midpoint:



In this case it is:

3 + 5 = 8
8 / 2 = 4

Again we now look at the data inside index 4 to see if that is the data we are looking for. In this case it is, and we have found Mango.

It took us, in total, 3 searches before we found the data we were looking for. Job done!

It's your turn!

Using a binary search, locate the number 54 in the below list. You must show your working. [3]

3	10	24	44	54	86	93

Describe why a binary search could not be used on the below data set: [2]

10	54	3	93	86	44	24

Linear Search

In a linear search we simply go down the list checking each piece of data in turn until we find the data we are looking for. If we check a piece of data and it is not the one we are looking for, we move onto the next one.

Very importantly, when undertaking a linear search, the data set does not need to be in order! It doesn't matter if it is in order or it isn't!

Example 1:

Using a binary search locate the number 26 in the below data set. You must show your working:

75	10	26	97	68	5	56

Firstly we will check the first piece of data:

75	10	26	97	68	5	56
↑						

This piece of data is not the one we are looking for (26) so we move onto the next piece of data:

75	10	26	97	68	5	56
	↑					

Again, this is not the piece of data we are looking for (26), so we move onto the next piece of data:

75	10	26	97	68	5	56
		↑				

We have found it! Now we have found the piece of data we are looking for we can stop looking. In this case, it took us 3 searches before we found the piece of data we were looking for.

Again, you would need to show the above process in the exam of checking each piece of data in turn, to show the examiner you understand in a linear search you would your way down the data set looking at each piece of data in turn.

Example 2:

Using a linear search locate the work Mango in the below data set. You must show your working:

Apple	Banana	Cherry	Kiwi	Mango	Orange

Firstly we will check the first piece of data:

Apple	Banana	Cherry	Kiwi	Mango	Orange
↑					

This piece of data is not the one we are looking for (Mango) so we move onto the next piece of data:

Apple	Banana	Cherry	Kiwi	Mango	Orange
	↑				

Again, this is not the piece of data we are looking for (Mango), so we move onto the next piece of data:

Apple	Banana	Cherry	Kiwi	Mango	Orange
		↑			

Again, this is not the piece of data we are looking for (Mango), so we move onto the next piece of data:

Apple	Banana	Cherry	Kiwi	Mango	Orange
			↑		

Again, this is not the piece of data we are looking for (Mango), so we move onto the next piece of data:

Apple	Banana	Cherry	Kiwi	Mango	Orange
				↑	

We have found it! Now we have found the piece of data we are looking for we can stop looking. In this case, it took us 5 searches before we found the piece of data we were looking for.

It's your turn!

Using a linear search, locate the number 54 in the below list. You must show your working. [3]

3	10	24	44	54	86	93

Describe one difference between a linear search and a binary search. [2]

The Pros and Cons:

Now we have practiced how to perform both searching algorithms, we can now look at the advantages and disadvantages of the two.

Advantages of a binary search:

- Better for searching through large data sets as it is quicker in locating data (as it reduces the size of the list down)
- It is a fairly simple and straightforward searching algorithm to use

Disadvantages of a binary search:

- Is not effective for searching through smaller data sets, as it over complicates the problem and therefore would take longer
- It only works on sorted lists. A binary search cannot be used on a list that is unsorted

Advantages of a linear search:

- Can be used on any list, whether it is sorted or unsorted
- Better for searching through small data sets

Disadvantages of a linear search:

- Ineffective when used on a large data set, as checking each piece of data on large data sets can be time consuming

Standard Sorting Algorithms

In this section we are going to focus on sorting algorithms.

A sorting algorithm is used to rearrange/reorder the data within a data set into a specific order (e.g. smallest to largest, alphabetical order etc.) There are three sorting algorithms we need to know:

- Bubble Sort
- Merge Sort
- Insertion Sort

As with the searching algorithms, you must show your working out when answering these questions in the exam, as you must be able to demonstrate clearly you know how the sorting algorithm works. Simply writing out the list in order with no working out will not access you any marks!

Bubble Sort:

In a bubble sort we look at two pieces of data in turn. We rearrange those two pieces of data (if we need to) before moving onto the next two pieces of data. Each time we go from the start of the data set to the end we call it a pass. We cannot stop passing through the data set until we pass through it and move no pieces of data.

Example 1:

Using a bubble sort rearrange the below data set so it is in order from smallest to largest. You must show your working:

75 10 26 97 68

Firstly we will look at 75 and 10:

$\underbrace{75 \quad 10} \quad 26 \quad 97 \quad 68$

As we are rearranging from smallest to largest, we need to swap these two numbers around:

10 75 26 97 68

Now we look at 75 and 26:

$10 \quad \underbrace{75 \quad 26} \quad 97 \quad 68$

26 is smaller than 75, so we swap them over again:

10 26 75 97 68

Now we look at 75 and 97:

$10 \quad 26 \quad \underbrace{75 \quad 97} \quad 68$

75 is smaller than 97, so we do not swap them over:

10 26 75 97 68

Now we look at 97 and 68:

$10 \quad 26 \quad 75 \quad \underbrace{97 \quad 68}$

68 is smaller than 97, so we swap them over:

10 26 75 68 97

We have now reached the end of our list, and therefore the end of our pass. Because we moved data in that pass, we must now repeat the process again, and perform another pass.

In this new pass (our second pass) we firstly look at 10 and 26:

$\underbrace{10 \quad 26} \quad 75 \quad 68 \quad 97$

10 is smaller than 26, so we do not swap them over:

10 26 75 68 97

Now we look at 26 and 75:

$$10 \quad \underbrace{26 \quad 75} \quad 68 \quad 97$$

26 is smaller than 75, so we do not swap them over:

10 26 75 68 97

Now we look at 75 and 68:

$$10 \quad 26 \quad \underbrace{75 \quad 68} \quad 97$$

68 is smaller than 75, so we swap them over:

10 26 68 75 97

Finally we look at 75 and 97:

$$10 \quad 26 \quad 68 \quad \underbrace{75 \quad 97}$$

75 is smaller than 97, so we do not swap them over:

10 26 68 75 97

We have now reached the end of our list, and therefore the end of our second pass. Because we moved data in that pass, we must now repeat the process again, and perform another pass.

In this new pass (our third pass) we firstly look at 10 and 26:

$$\underbrace{10 \quad 26} \quad 68 \quad 75 \quad 97$$

10 is smaller than 26, so we do not swap them over:

$$10 \quad 26 \quad 68 \quad 75 \quad 97$$

Now we look at 26 and 68:

$$10 \quad \underbrace{26 \quad 68} \quad 75 \quad 97$$

26 is smaller than 68, so we do not swap them over:

$$10 \quad 26 \quad 68 \quad 75 \quad 97$$

Now we look at 68 and 75:

$$10 \quad 26 \quad \underbrace{68 \quad 75} \quad 97$$

68 is smaller than 75, so we do not swap them over:

$$10 \quad 26 \quad 68 \quad 75 \quad 97$$

Now we look at 75 and 97:

$$10 \quad 26 \quad 68 \quad \underbrace{75 \quad 97}$$

75 is smaller than 97, so we do not swap them over:

$$10 \quad 26 \quad 68 \quad 75 \quad 97$$

We have now reached the end of our list, and therefore the end of our third pass. We did not move any data in that pass, which means our data set must be in order. We can therefore end our sorting process. In this case, it took us three passes to get the data set in order!

Example 2:

Using a bubble sort rearrange the below data set so it is in alphabetical order (A-Z). You must show your working:

Apple Kiwi Banana Orange Mango

Firstly we will look at Apple and Kiwi:

Apple Kiwi Banana Orange Mango

Apple comes before Kiwi in the alphabet, so we do not swap them over:

Apple Kiwi Banana Orange Mango

Now we look at Kiwi and Banana:

Apple Kiwi Banana Orange Mango

Banana comes before Kiwi in the alphabet, so we swap them over:

Apple Banana Kiwi Orange Mango

Now we look at Kiwi and Orange:

Apple Banana Kiwi Orange Mango

Kiwi comes before Orange in the alphabet, so we do not swap them over:

Apple Banana Kiwi Orange Mango

Now we look at Orange and Mango:

Apple Banana Kiwi Orange Mango

Mango comes before Orange in the alphabet, so we swap them over:

Apple Banana Kiwi Mango Orange

We have now reached the end of our list, and therefore the end of our pass. Because we moved data in that pass, we must now repeat the process again, and perform another pass.

In this new pass (our second pass) we firstly look at Apple and Banana:

Apple Banana Kiwi Mango Orange

Apple comes before Banana in the alphabet, so we do not swap them over:

Apple Banana Kiwi Mango Orange

Now we look at Banana and Kiwi:

Apple Banana Kiwi Mango Orange

Banana comes before Kiwi in the alphabet, so we do not swap them over:

Apple Banana Kiwi Mango Orange

Now we look at Kiwi and Mango:

Apple Banana Kiwi Mango Orange

Kiwi comes before Mango in the alphabet, so we do not swap them over:

Apple Banana Kiwi Mango Orange

Now we look at Mango and Orange:

Apple Banana Kiwi Mango Orange

Mango comes before Orange in the alphabet, so we do not swap them over:

Apple	Banana	Kiwi	Mango	Orange

We have now reached the end of our list, and therefore the end of our second pass. We did not move any data in that pass, which means our data set must be in order. We can therefore end our sorting process. In this case, it took us two passes to get the data set in order!

It's your turn!

Using a bubble sort, rearrange the data set below from largest to smallest. You must show your working: [4]

45	**88**	**12**	**34**	**3**

<u>Merge Sort:</u>

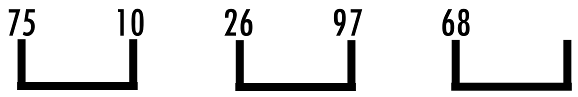
In a merge sort we break our data set up into pairs to start off with. We rearrange those pairs, and then begin to merge those pairs together. We repeat this process until we merge the whole data set back together and rearrange that.

<u>Example 1:</u>

Using a merge sort rearrange the below data set so it is in order from smallest to largest. You must show your working:

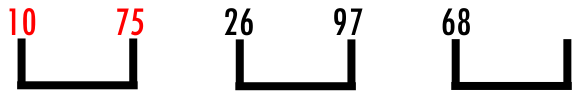
75 10 26 97 68

Firstly we will break the list up into pairs:

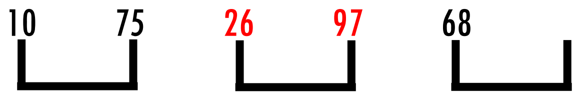


As you can see, 68 is on its own at the end. This is fine. We do not put a new number into the list for 68 to be in a pair, we simply leave it on its own.

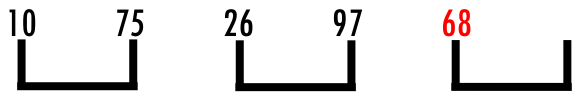
Now we rearrange the pairs. Let's look at the first pair. 10 is smaller than 75 so we swap them over:



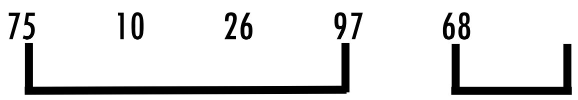
Next we look at 26 and 97. 26 is smaller than 97 so we do not swap them over:



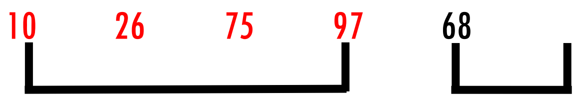
Finally we look at 68. 68 is on its own, so we do not swap it over (because there's nothing to swap it with!):



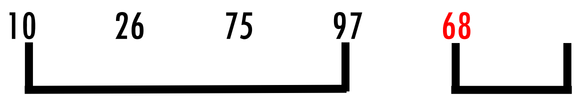
Now we have got to the end of our list, we can start to merge some pairs together. We always start from the left, and merge the nearest two pairs together. In this case, the pair containing 10 and 75, and the pair containing 26 and 97, will merge together to become a mini list. 68 has nothing to merge with so will remain on its own:



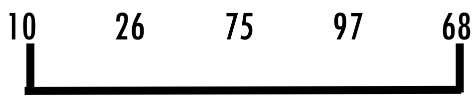
We now rearrange the new mini lists. In the case of the first mini list, 10 is the smallest number, followed by 26, followed by 75, and finally 97:



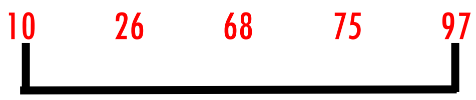
We now look at 68. 68 is on its own, so we do not swap it over (because there's nothing to swap it with!):



Now we have got to the end of our list, we can start to merge some mini lists/pairs together. Again, we always start from the left, and merge the nearest two mini lists/pairs together. In this case, we are going to be merging the mini list containing 10, 26, 75, and 97 with 68. This effectively now merges our whole list back together again:



Now we rearrange this new list. In this case, 10 is the smallest, followed by 26, then 68, followed by 75, and finally 97:



Our list has now been fully merged back together again and placed in order. The process is complete!

As long as you show that process of breaking the original list up into pairs and mini lists (or sub lists) and reorganising and then merging, you will access all the marks!

Example 2:

Using a merge sort rearrange the below data set so it is in alphabetical order (A-Z). You must show your working:

Apple Kiwi Banana Orange Mango

Firstly we will break the list up into pairs:

Apple Kiwi Banana Orange Mango

As you can see, Mango is on its own at the end. This is fine. We do not put a new word into the list for Mango to be in a pair, we simply leave it on its own.

Now we rearrange the pairs. Let's look at the first pair. Apple comes before Kiwi in the alphabet, so we do not swap them over:

Apple Kiwi Banana Orange Mango

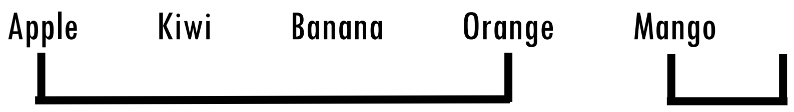
We now look at Banana and Orange. Banana comes before Orange in the alphabet, so we do not swap them over:

Apple Kiwi Banana Orange Mango

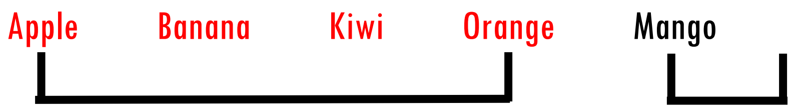
We now look at Mango. Mango is on its own, so we do not swap it over (because there's nothing to swap it with!):

Apple Kiwi Banana Orange Mango

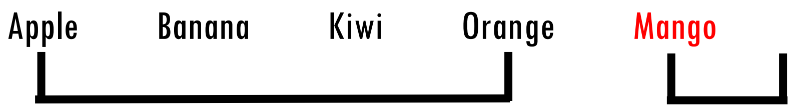
Now we have got to the end of our list, we can start to merge some pairs together. We always start from the left, and merge the nearest two pairs together. In this case, the pair containing Apple and Kiwi, and the pair containing Banana and Orange, will merge together to become a mini list. Mango has nothing to merge with so will remain on its own:



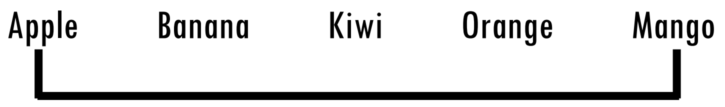
Now we rearrange the mini lists. Let's look at the first mini list. In this case Apple will come first, followed by Banana, then Kiwi, and finally Orange:



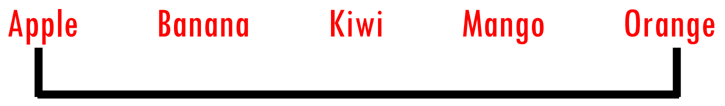
We now look at Mango. Mango is on its own, so we do not swap it over (because there's nothing to swap it with!):



Now we have got to the end of our list, we can start to merge some mini lists/pairs together. Again, we always start from the left, and merge the nearest two mini lists/pairs together. In this case, we are going to be merging the mini list containing Apple, Banana, Kiwi, and Orange with Mango. This effectively now merges our whole list back together again:



Now we rearrange this new list. In this case, Apple comes first, followed by Banana, then Kiwi, then Mango, and finally Orange:



Our list has now been fully merged back together again and placed in order. The process is complete!

It's your turn!

Using a merge sort, rearrange the data set below from largest to smallest. You must show your working: [4]

45 88 12 34 3

"Make sure you read the question in the exam really carefully, and ensure you put the data set into the order it is requesting. For example, it is asks for largest to smallest, make sure that is what you do! It's easy when under exam stress to put it in the wrong order!"

<u>Insertion Sort:</u>

In an insertion sort we create a temporary list. We then work our way through our data set and place each piece of data into the correct location in the temporary list.

<u>Example 1:</u>

Using an insertion sort rearrange the below data set so it is in order from smallest to largest. You must show your working:

```
75    10    26    97    68
```

Firstly we are going to create our temporary list, which will start off empty:

```
75    10    26    97    68
_
```

We now look at 75. As 75 is the first number we are looking at, it simply goes straight into our new list at the beginning:

```
75    10    26    97    68
75
```

We now look at 10. 10 is smaller than 75, so it goes before 75 to the start of the list, and 75 moves one place down:

```
75    10    26    97    68
75
10    75
```

We now look at 26. 26 is smaller than 75 and bigger than 10, so it slots in between them:

```
75    10    26    97    68
75
10    75
10    26    75
```

Now we look at 97. 97 is bigger than 75 so it goes at the end of the temporary list:

75	10	26	97	68
75				
10	75			
10	26	75		
10	26	75	97	

Finally we look at 68. 68 is smaller than 75 and bigger than 26, so it slots in between them:

75	10	26	97	68
75				
10	75			
10	26	75		
10	26	75	97	
10	26	68	75	97

And that's it! Our list has now been reordered into the below:

10	26	68	75	97

As long as you show the creation of a temporary list and that you are focusing on one piece of data at a time, inserting it into the correct place in the temporary list, you will access all the marks! You must make sure you show each insertion though, as you can see above. You cannot just write the list out in order once!

<u>Example 2:</u>

Using an insertion sort rearrange the below data set so it is in alphabetical order (A-Z). You must show your working:

Apple	Kiwi	Banana	Orange	Mango

Firstly we are going to create our temporary list, which will start off empty:

Apple	Kiwi	Banana	Orange	Mango
_				

We now look at Apple. As Apple is the first word we are looking at, it simply goes straight into our new list at the beginning:

Apple	Kiwi	Banana	Orange	Mango
Apple				

Now we look at Kiwi. Kiwi comes after Apple in the alphabet, so it goes after Apple:

Apple	Kiwi	Banana	Orange	Mango
Apple				
Apple	Kiwi			

Now we look at Banana. Banana comes after Apple and before Kiwi in the alphabet, so it slots in between them:

Apple	Kiwi	Banana	Orange	Mango
Apple				
Apple	Kiwi			
Apple	Banana	Kiwi		

Now we look at Orange. Orange comes after Kiwi in the alphabet, so goes at the end:

Apple	Kiwi	Banana	Orange	Mango
Apple				
Apple	Kiwi			
Apple	Banana	Kiwi		
Apple	Banana	Kiwi	Orange	

Now we finally look at Mango. Mango comes after Kiwi and before Orange in the alphabet, so it slots in between them:

Apple	Kiwi	Banana	Orange	Mango
Apple				
Apple	Kiwi			
Apple	Banana	Kiwi		
Apple	Banana	Kiwi	Orange	
Apple	Banana	Kiwi	Mango	Orange

And that is it! Our list has now been reordered into the below:

Apple	Banana	Kiwi	Mango	Orange

It's your turn!

Using an insertion sort, rearrange the data set below from largest to smallest. You must show your working: [4]

45	**88**	**12**	**34**	**3**

How to produce an algorithm using pseudocode

When writing pseudocode, we follow a general set of commands to ensure consistency amongst algorithms. Below are the different programming techniques we might use, and how we would write them in pseudocode:

Comments - //

Variables – variablename = 3	Used to store an integer
variablename = "string"	Used to store a string
CONST variablename = 3	Used to create a constant
Casting – str(20)	Used to cast an integer to a string
int("20")	Used to cast a string to an integer
Input – variablename = input()	Used to get input from the user
Output – print("Hello")	Used to output a string
print(5)	Used to output an integer
print(variablename)	Used to output the contents of a variable

Selection – if … then
elseif … then
else
endif
	Used to create selection

While Loop – while … endwhile	Used to create a while loop
For Loop – for … endfor	Used to create a for loop
Logical operators – AND OR NOT	Used for logical operations

Comparison operators	- ==	Equal to
	- !=	Not equal to
	- <	Less than
	- <=	Less than or equal to
	- >	Greater than
	- >=	Greater than or equal to
Arithmetic operators	- +	Addition
	- -	Subtraction
	- *	Multiplication
	- /	Division
	- MOD	Modulus (remainder)
	- DIV	Division (whole number)
	- ^	Exponentiation (power of)
String handling	– variable.LENGTH	Gather the length of a variable
	- variable.UPPER	Convert contents to uppercase
	- variable.LOWER	Convert contents to lowercase

Sub-routines	– function name(parameters)
	return value
	endfunction
		Create a function and pass parameters

Arrays	– arrayname[lengthofarray] - arrayname[1] = “Bob”	Use of an array and an index
File Handling	– OpenWrite(“filename.txt”)	Used to open a text file in write mode
	- file.ReadLine()	Used to read contents of a text file
	- file.WriteLine(variable)	Used to write to a text file
	- file.Close()	Used to close a text file

Writing algorithms in the exam can be daunting, especially when you see an algorithm question pop up that you are not sure how to complete! However, if you break the question down into three key parts, not only does this help you structure your algorithm, but it allows you can ensure you access some of the marks!

There are three parts to every algorithm:

Input – Here the user will need to input some data

Process – Here you will need to perform some form of process on your data (whether this is perform a calculation on your data, or perform some form of condition or check on your data)

Output – Here you will need to output something to your user

Let's look at the algorithm below, used to calculate the price of a product with a 10% discount.

```
price = input("Please enter the price of the product: ")

new_price = price * 0.9

print(new_price)
```

As you can see in red we have our input, in this case we are getting the user to input the price of the product.

In green we are then calculating the new price of the product by taking the original price and multiplying it by 0.9 (therefore taking off 10%).

Finally in purple we are outputting the new price.

In this section we are going to write some algorithms using this structure, gradually getting more and more complicated.

This revision guide does not cover lots of algorithms. It is therefore imperative that you get some algorithm questions from your teacher or a question bank online and practice! Writing algorithms is like learning a new language, you won't get it all in one attempt, it requires practice and more practice!
Q: A school wants to create a simple program to calculate the area of a triangle. The area of a triangle can be calculated by multiplying the base by the height, and then dividing by two.

Write an algorithm that:

- Allows the user to input the base and height of the triangle
- Calculates the area of the triangle
- Outputs the area of the triangle

As we can see in the above question, we have our input, our process, and our output. Let's tackle each bullet point in turn.

Firstly we need to write part of our algorithm so that the user can input their height and base of the triangle.

```
height = input("Please enter the height of the triangle: ")
base = input("Please enter the base of the triangle: ")
```

We've now done the first part of the question. Now let's move onto the second bullet point and calculate the area.

```
height = input("Please enter the height of the triangle: ")
base = input("Please enter the base of the triangle: ")

area = (base * height) / 2
```

Here we have taken the base and height and multiplied them together, before taking that answer and dividing it by two. Whatever is inside the brackets will be executed first!

Finally, we now need to output our area.

```
height = input("Please enter the height of the triangle: ")
base = input("Please enter the base of the triangle: ")

area = (base * height) / 2

print(area)
```

There we have it, a finished algorithm!
Q: A company wants to create a program that alerts the user to the freeing and boiling points of water. The boiling point of water is 100 degrees Celsius, and the freezing point of water is 0 degrees Celsius.

Write an algorithm that:

- Allows the user to input the temperature of the water
- Calculates whether the water is boiling, freezing, or neither
- Outputs appropriate messages to the user

Firstly, let's complete the first bullet point and get our input.

```
water = input("Input the temperature of the water: ")
```

Now, to determine whether the water is freezing, boiling, or neither, we will need to use an if statement. Remember, pseudocode is not syntax specific, therefore we must write it so a programmer can pick this algorithm up and create the program in any language. Therefore, we will write the if statement like this:

```
water = input("Input the temperature of the water: ")

if water >= 100 then

elseif water <=0 then

else

endif
```

Notice how we have used the word then instead of the : we would use in Python. Also notice how we have ended our if statement with the word endif, showing the programmer this is where the if statement ends.

Now, we just need to add our outputs in

```
water = input("Input the temperature of the water: ")

if water >= 100 then
        print("Water is at boiling point!")
elseif water <=0 then
        print("Water is at freezing point!")
else
        print("Water is not boiling or freezing")
endif
```

Notice how we have still indented like we would if we were programming. This is very important! Finally, if we want to, we can add a comment in to explain what the algorithm does. To do this in pseudocode we use //

```
//A program to calculate if water is at freezing or boiling point

water = input("Input the temperature of the water: ")

if water >= 100 then
        print("Water is at boiling point!")
elseif water <=0 then
        print("Water is at freezing point!")
else
        print("Water is not boiling or freezing")
endif
```

Q: A parcel tracking company want to create a program to check whether an inputted parcel number is the required 8 characters or not. If a parcel number is 8 characters it should output the message ‘VALID’, otherwise it should output the message ‘INVALID’.

Write an algorithm that:

- Allows the user to input the parcel number
- Calculates whether the parcel number is 8 characters or not
- Outputs appropriate messages to the user

Firstly let’s do our input.

```
parcel = input(“Please enter the parcel number: “)
```

Now, we need to our process. In this case we need to work out the length of the parcel number. To do this we are going to use .length, and see if it is 8 characters or not

```
parcel = input(“Please enter the parcel number: “)

if parcel.length == 8 then

else

endif
```

Now we need to finally put our outputs in.

```
parcel = input(“Please enter the parcel number: “)

if parcel.length == 8 then
        print(“VALID”)
else
        print(“INVALID”)
endif
```

Q: A student wants to create a program using a function. The aim of the program is to return the day of week based on its number e.g. Monday = 1, Tuesday = 2 etc. up to Sunday = 7.

Write an algorithm that:

- Creates a function named DayOfWeek and takes the number of the day as a parameter
- Calculates what day is outputted for each number
- Returns the day of the week

Firstly we are going to create the function and pass the number as a parameter. We will do this in red, although it isn't input. This is so we can see the start of the function.

```
function DayOfWeek (number)

endfunction
```

As you can see above, just like when using a condition or loop, we must signify to the programmer where the function ends, using endfunction.

Next week will calculate what day is outputted for each number.

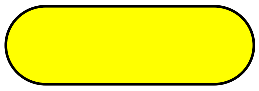
```
function DayOfWeek (number)
    if number == 1 then
    elseif number == 2 then
    elseif number == 3 then
    elseif number == 4 then
    elseif number == 5 then
    elseif number == 6 then
    elseif number == 7 then
    else
    endif
endfunction
```

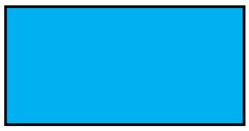
Finally, all we need to do now is add our days of the week in, and return our value (remember, a function must always return a value!)

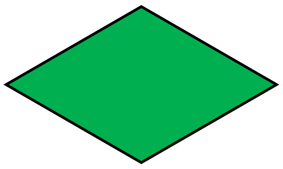
```
function DayOfWeek (number)

        if number == 1 then
                day = "Monday"
        elseif number == 2 then
                day = "Tuesday"
        elseif number == 3 then
                day = "Wednesday"
        elseif number == 4 then
                day = "Thursday"
        elseif number == 5 then
                day = "Friday"
        elseif number == 6 then
                day = "Saturday"
        elseif number == 7 then
                day = "Sunday"
        else
                print("Error")
        endif

        return day

endfunction
```

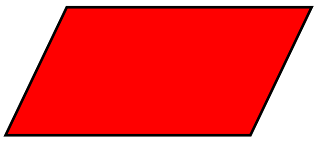
How to produce an algorithm using flowcharts

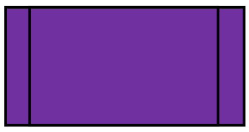
When creating a flowchart there are some key shapes that will be used:

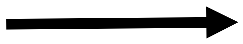

Terminal – Used to start and end the flowchart


Process – Used to show a process (something being calculated)


Decision – Used to represent a decision (yes or no)


Input/Output – Used to represent input and output of data


Sub-Routine – Used to represent the call to a sub-routine. The sub-routine would have its own flowchart, but this symbol would be used to call the sub-routine


Arrow – Used to join up the different symbols in the flowchart

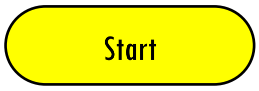
Let's have a go at creating a flowchart for a couple of our algorithms above.

Q: A school wants to create a simple program to calculate the area of a triangle. The area of a triangle can be calculated by multiplying the base by the height, and then dividing by two.

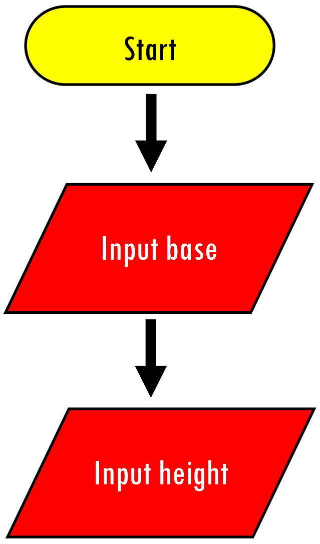
Create a flowchart that:

- Allows the user to input the base and height of the triangle
- Calculates the area of the triangle
- Outputs the area of the triangle

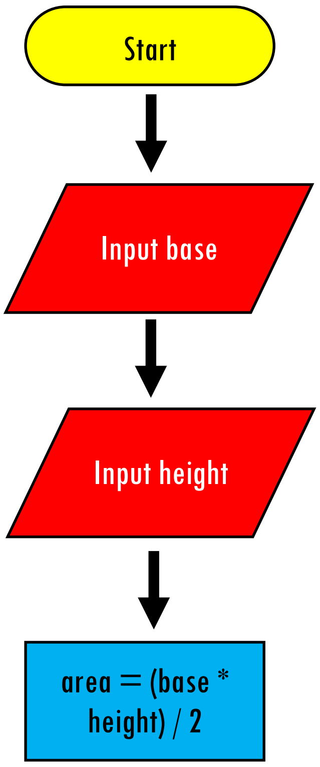
Firstly we must start with our terminal.



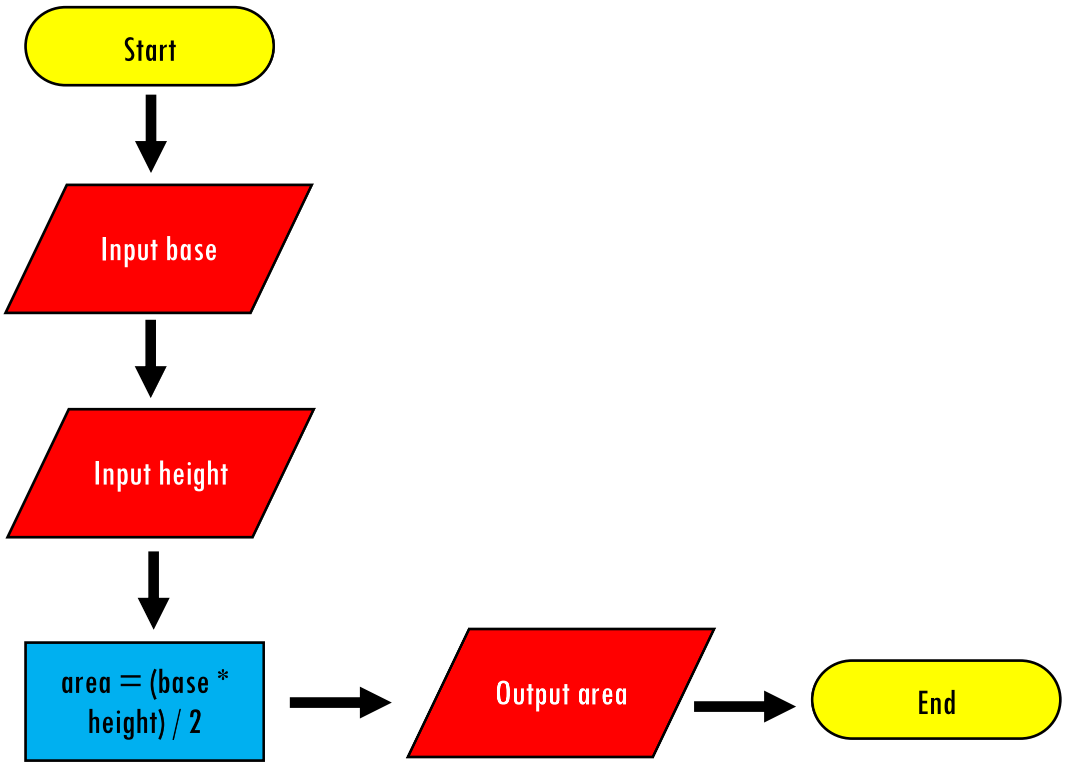
Next we will add in our inputs using our input/output shape, and join these with the arrow to denote which way the flowchart will flow.



Next we will add our process in where we calculate the area of the triangle.



Finally, let's add our output in to output the area using the input/output shape, and finish our flowchart with another terminal.



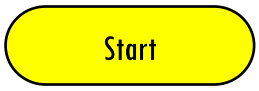
Let’s have a go at another flowchart, this time involving the decision shape.

Q: A company wants to create a program that alerts the user to the freeing and boiling points of water. The boiling point of water is 100 degrees Celsius, and the freezing point of water is 0 degrees Celsius.

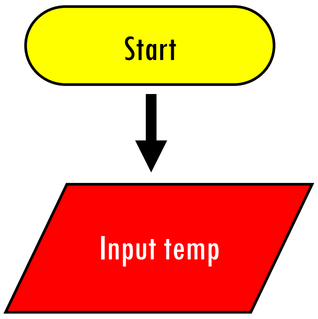
Write an algorithm that:

- Allows the user to input the temperature of the water
- Calculates whether the water is boiling, freezing, or neither
- Outputs appropriate messages to the user

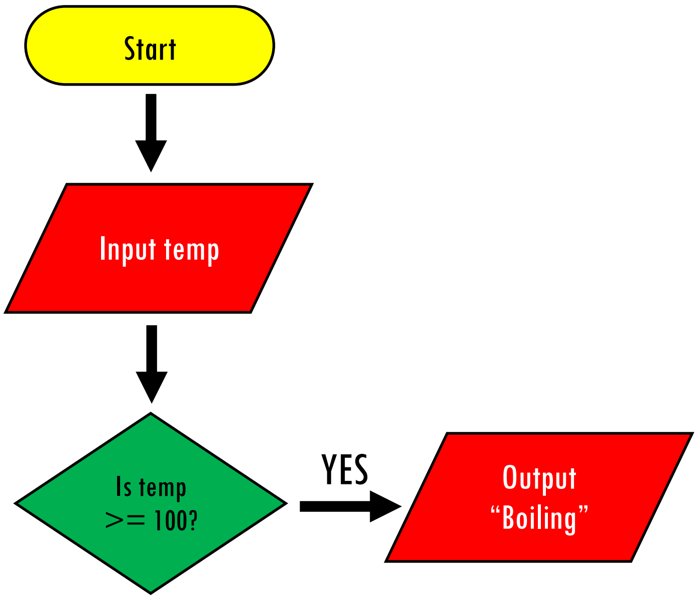
Firstly we must start with our terminal.



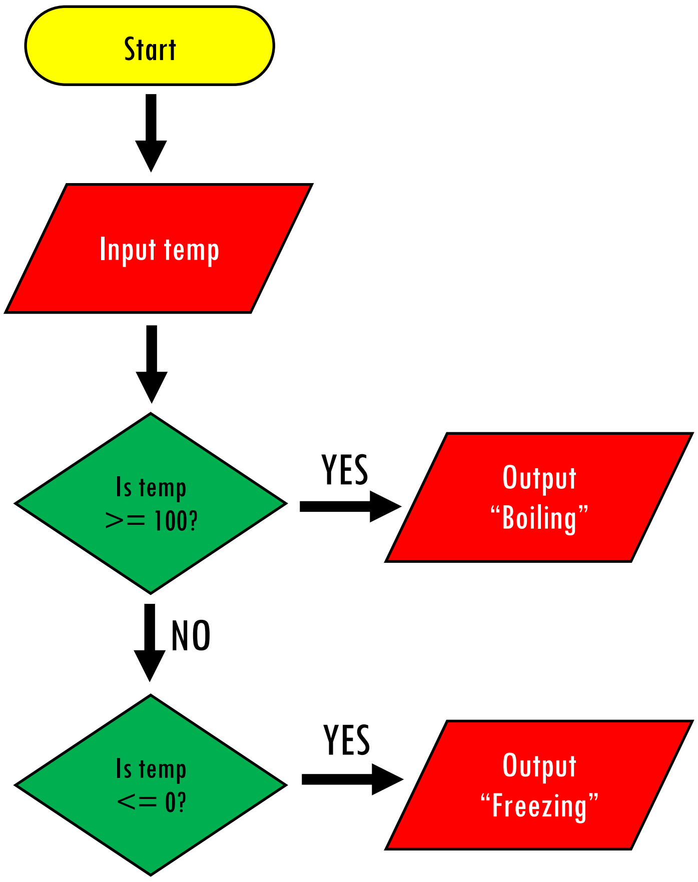
Next we will add in our input using our input/output shape, and join these with the arrow to denote which way the flowchart will flow.



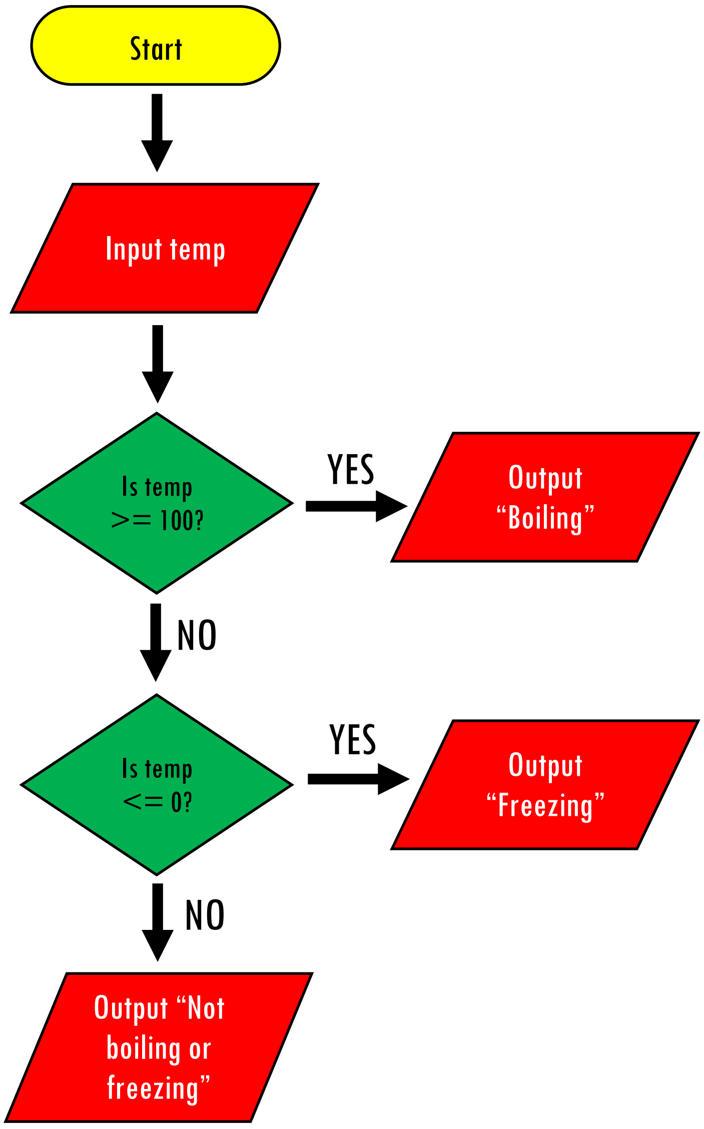
Next we will use the decision shape to calculate if the temp is greater than or equal to 100, and if it is we will output the message “Boiling”.



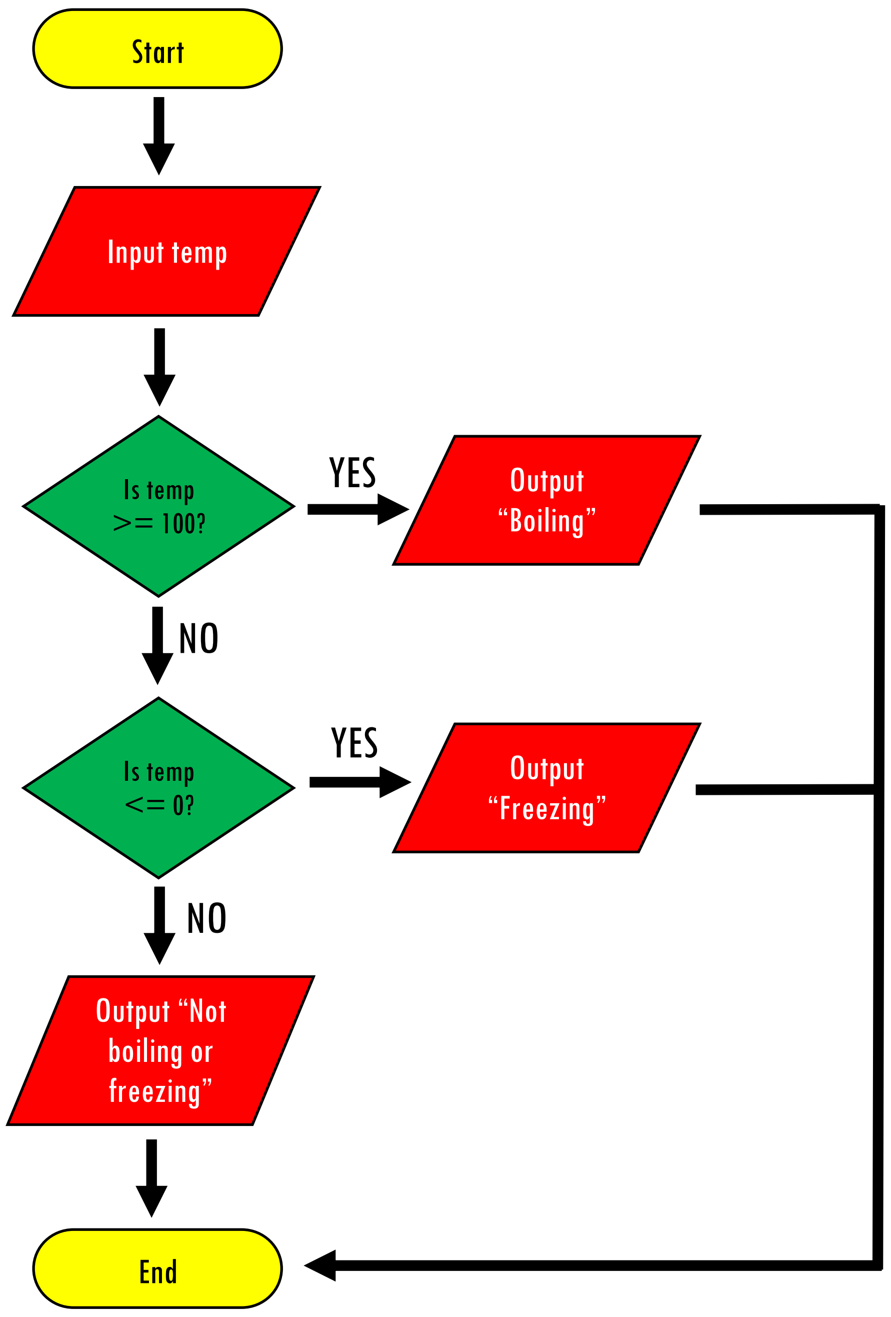
Notice how we have added in the word YES above the arrow pointing towards the output boiling. This is to signify if the condition is met (the temp is greater than or equal to 100) then this is the way the program should go. Now we need to do the NO option. In this case if the answer is NO to the question in the decision shape, we ask another question to see if the temp is less than or 0 zero, and if it is then the output should be “Freezing”.



Now we need to add our final output for if the temp is not greater than or equal to 100, or less than or equal to 0, which should just say "Not boiling or freezing"



Finally we need to add our final terminal in to signify the end of the flowchart. All shapes must connect to the next shape. As you can see we have three outputs that have nowhere to go once we get to them. Therefore, all must connect to the end terminal.



That’s it, our completed flowchart using decisions, input/outputs, and terminals.

Trace Tables

A trace table is used to simulate the execution of an algorithm, looking for any logic errors that may be hiding in the algorithm.

Remember, a logic error is an error that causes the program to work incorrectly. In other words, not the way it should. Common logic errors might be that we have used the wrong Boolean operator (e.g. AND instead of OR) or the wrong comparison operator (e.g. < instead of >).

Let's have a look at a trace table for the below algorithm:

```
number = int(input("Please input a number: "))

count = 1

result = 1

for count to number:

   result = result + count
   print(result)

next count
```

Our simple algorithm above is going to ask our user to input a number. We are then going to create a loop (count-controlled) and iterate through an addition until our count is the same as the number the user inputted (in other words, if the user inputs 5 as their number our loop will iterate 5 times).

Here is our empty trace table for the above algorithm:

count	result	number	output

Let's discuss firstly what the table consists of. As you can see, each variable in our program has been assigned a column. We have three variables in our program (count, result, number). What we are going to do is run through our program ourselves and fill in our trace table with the values of each variable, until the program stops.

We have also added an output column on the end to show any output that will take place. If no output takes place, we will simply draw a line through the box.

Let's run through the algorithm now and complete our trace table. For the example here we are going to imagine our user inputs 3 as their number.

Cycle one:

We start on the first line of code: number = int(input("Please input a number: "))

As stated above we are going to imagine our user inputs 3. Therefore, at the end of this line of code number is now 3, so we put that into the first row in our column under number.

count	result	number	output
		3	

Now we move onto the next line of code: count = 1

This one is easy! Simply write 1 under count.

count	result	number	output
1		3	

Now we move onto the next line of code: result = 1

This one is easy! Simply write 1 under result.

count	result	number	output
1	1	3	

Now we enter our loop. The next line of code is this: result = result + count

This now means result will change. We simply just update our trace table with the new value, rather than starting a new row. In this case result will now be its old value (1) added to the value of count (1) meaning the new value of result is 2.

count	result	number	output
1	2	3	

Finally, we have our output line of code: result = print(result)

In our output column we now write the value outputted, which in this case will be 2.

count	result	number	output
1	2	3	2

Above now is the completed trace table for the first cycle of our algorithm. Now we will return to the start of our loop and repeat this process until the loop is no longer repeating.

Cycle two:

count	result	number	output
1	2	3	2
2	4	3	4

Cycle three:

count	result	number	output
1	2	3	2
2	4	3	4
3	7	3	7

At the end of cycle three we have completed our loop and, in this case, the program ends. The trace table above is now the completed trace table for this algorithm.

Completing this trace table means we can spot any logic errors before we create the actual program. The more errors we can identify and eradicate before programming the better!

Past Exam Questions

Answer the questions below, to help you revise what has been covered in 2.1 Algorithms.

1. A doctor's surgery uses a computer system to manage their patients' details and records. All staff use the same system.

   The system uses abstraction.

a. Define what is meant by abstraction. [2]

b. Describe why the system would use abstraction. You should reference the doctors and the receptionists in your answer. [4]

2. Using a binary search, locate the word 'Boat' in the below list. You must show your working: [3]

Alpine	Boat	Delivery	Goat	Meteor	Zen

3. Describe why the above list is suitable for a binary search. [2]

4. Identify another standard searching algorithm. [1]

5. Describe one advantage to using a binary search over the searching algorithm you identified in question 4. [2]

6. Using a merge sort, organise the below data set so that it is in alphabetical order (A-Z). You must show your working. [4]

Boat	Zen	Meteor	Alpine	Delivery	Goat

7. Explain one advantage to using a merge sort over a bubble sort. [2]

8. Identify a third standard sorting algorithm, other than a merge sort and a bubble sort. [1]

2.2 Programming Fundamentals

In this section you will revise the following:

2.2.1 Programming Fundamentals

- The use of variables, constants, operators, inputs, outputs and assignments
- The use of the three basic programming constructs used to control the flow of a program:
    - Sequence
    - Selection
    - Iteration
- The common arithmetic operators
- The common Boolean operators AND, OR and NOT

2.2.2 Data Types

- The use of data types:
    - Integer
    - Real
    - Boolean
    - Character and String
    - Casting

2.2.3 Additional Programming Techniques

- The use of basic string manipulation
- The use of basic file handling operations:
  - Open
  - Read
  - Write
  - Close
- The use of records to store data
- The use of SQL to search for data
- The use of arrays (or equivalent) when solving problems, including both one and two dimensional arrays
- How to use sub programs (functions and procedures) to produce structured code
- Random number generation

Technical Terms

Technical Term	Definition
Variable	Used within a computer program to store a single piece of data. The contents can be changed
Constant	Used within a computer program to store a single piece of data. The data does not change within the program (it remains constant/the same)
Operator	Used within a program to represent an action e.g. + which represents addition
Input	Data which is entered into the computer program
Output	Data which is sent out by the computer program
Assignment	This is when a value is set/given to a variable
Sequence	One of the three basic programming constructs. This is when the code/instructions are all in order, so that one piece of code executes, followed by the next etc.
Selection	One of the three basic programming constructs. This is when a choice/decision is used within a program, allowing for multiple outcomes dependent on the data that is inputted
Iteration	One of the three basic programming constructs. This is when a section of code is repeated.
Condition Controlled Iteration	This is a type of iteration, where the code is repeated until a certain condition is met. This means the code could execute any number of times, as it relies on the condition being met before stopping.
Count Controlled Iteration	This is a type of iteration, where the code is repeated a specific number of times. This means the code will only execute this number of times before stopping.
String	A data type which is used when storing alphanumeric characters and symbols. E.g. letters, numbers, #?!- etc.
String Manipulation	Refers to when a string is altered or edited to suit the means of the program.
String Concatenation	Refers to when strings are brought together and combined
String Splitting	Refers to when a string is split at specific points, separating the single string into multiple strings

Text File	Used to store data beyond whilst the program is running
Database	Used to store large amounts of data beyond whilst the program is running. Offers a more secure way of storing data
Record	Refers to all data stored about a single data subject within a table. Can often be seen as a single row within a table
Field	Refers to the various categories for which data is collected. Can often be seen as a single column within a table
SQL (Structured Query Language)	The language used to manage, maintain, and query a database.
SQL Query	Written to extract data from within a database
SELECT	An element of an SQL query. Used to determine which fields will be extracted
FROM	An element of an SQL query. Used to determine which table the data will be extracted from
WHERE	An element of an SQL query. Used to set the criteria for the query. This element is not a necessity when writing an SQL query
List	Used within a program. Allows you to store multiple pieces of data in the same location
One Dimensional List	Stores one piece of data per data subject, identified using a single index
Two Dimensional List	Stores multiple pieces of data per data subject, identified using two indices
Sub-Program	Used within a computer program. It is a section of code that is 'called upon' within a program when being executed. It allows for the same piece of code to be used without having to repeatedly write it out
Argument	Data that is passed into a sub-program to be used within the sub-program
Parameter	The variables that are passed into a sub-program to be used within the sub-program
Function	A type of sub-program. A function is a sequence of instructions which return a value
Procedure	A type of sub-program. A procedure is a sequence of instructions which do not return a value

Data Type	Refers to the way data is stored within a program. The data type determines what type of operations can take place on the data. There are 5 main data types (string, integer, Boolean, real/float, character)
Integer	A data type, used to store whole numbers
Boolean	A data type, used to store a TRUE/FALSE value
Real/Float	A data type, used to store numbers that require decimal places
Character	A data type, used to store a single character
Casting	Refers to the process of altering a variables data type within a program
Arithmetic Operator	Refers to the various operators used to perform arithmetic calculations within a program (+, -, *, /)
DIV	Is short for Quotient, and is used to find the whole number of a division
MOD	Is short for Modulus, and is used to find the remainder of a division
Boolean Operator	Refers to the various operators used to perform comparisons on data within a program which provides more focused results (AND, OR, NOT)
Random	A function built into python which allows for random choices to be made, used during random number generation

****PLEASE NOTE****

The following section (2.2 Programming Techniques) will use Python based examples to support the above content where suitable.

Please be aware that other programming languages are available, and the use of the above techniques can differ between different languages.

This revision guide should also not be used as a substitute for quality teaching of programming, but as a revision resource once programming has been taught.

Variables, Constants, Operators, Inputs, Outputs, Assignments

In this topic we are going to cover the difference between a variable and a constant, as well as what an operator is. We will also discuss what input and output is and how it is used within a computer program. Finally, we will discuss what is meant by assignment.

Variables and Constants

Both variables and constants are used to store single pieces of data within a program. However, there is a difference between the two.

A variable is used to store data, where the data can change at any time within the program.

A constant is used to also store data; however, the data does not change at any time within the program. It remains constant (the same) throughout the program.

Below is an example of a program which contains variables and constants. In this program we are simply storing the correct username and password for a user, and asking them to enter their username and password:

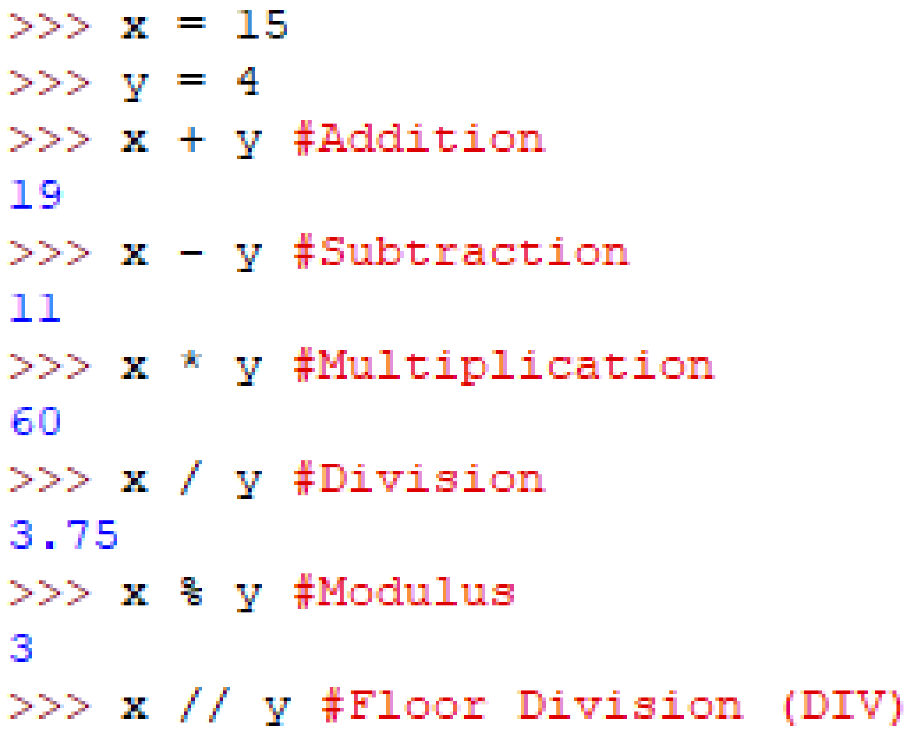
```
{ correct_username = "Username1"
  correct_password = "Password1"

{ inputted_username = ""
  inputted_password = ""

  inputted_username = input("Please enter your username: ")
  inputted_password = input("Please enter your password: ")
```

{ As you can see here the data within these constants will not change throughout the program. In this case it is because later on in the program we would likely check to see if what they have inputted as their username and password is correct. The data inside these constants will be what we use to check against what they have inputted.

{ As you can see here the data within these variables will change within the program. In this case we have started them off as empty strings, however once the user inputs their username and password this data will be stored within these variables. This means the data will change (from an empty string to whatever the user inputs).

Operators

An operator is simply something used within a program to perform an action. The following are operators used within Python programming:



```
>>> x = 15
>>> y = 4
>>> x + y #Addition
19
>>> x - y #Subtraction
11
>>> x * y #Multiplication
60
>>> x / y #Division
3.75
>>> x % y #Modulus
3
>>> x // y #Floor Division (DIV)
```

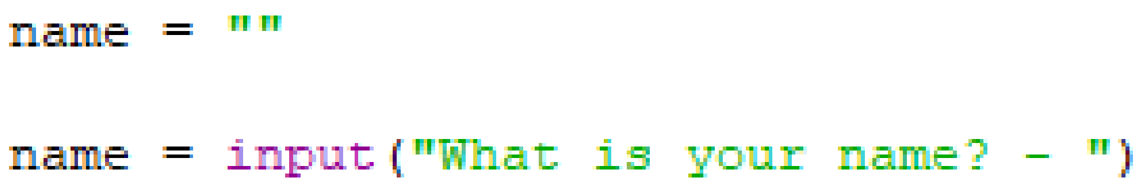
As you can see from the above example, addition and subtraction are the same as what we use in every day maths. The multiplication is changed from an x to a *, and division is changed to a /.

We will come onto % (MOD or Modulus) and // (DIV or Quotient) in a later topic!

<u>Input and Output</u>

We use input within a computer program when we want our user to enter some data. Often, we will ask them a question or request they type something in, and give them a space to enter this data. This data must then be stored (in our examples we will be storing this within a variable).

Below is an example of us using input within a program in Python. Here we are simply asking the user to enter their name, and then storing this within a variable called name:

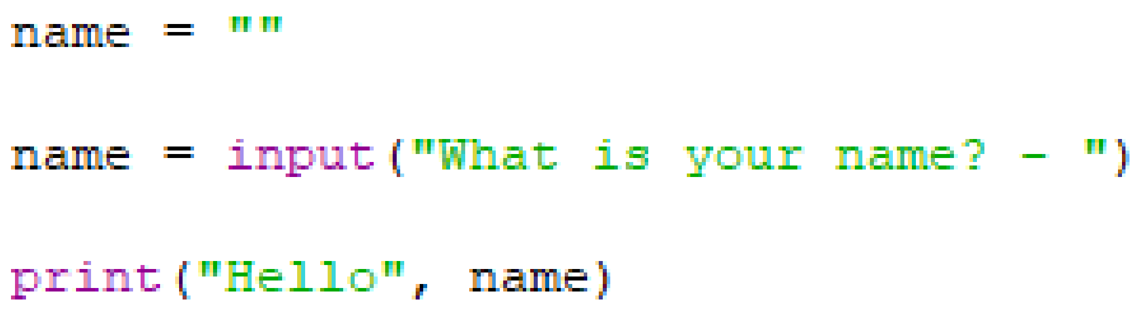


```
name = ""

name = input("What is your name? - ")
```

As you can see, within Python we use the command input which informs our program that the user must input some data here before proceeding.

We use output within a computer program when we want our program to present some data to our user on the screen.

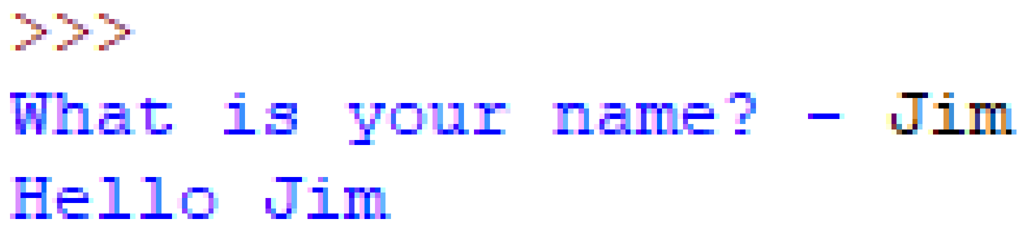
Below is an example of us using output within a program in Python. Here we are simply following on from the above program, outputting a hello message to the user:



```
name = ""

name = input("What is your name? - ")

print("Hello", name)
```

As you can see, within Python we use the command print which informs our program that we want to output something to the user. In this case, we output the message “Hello” followed by the users name.

Below is an example of how the program looks once executed:

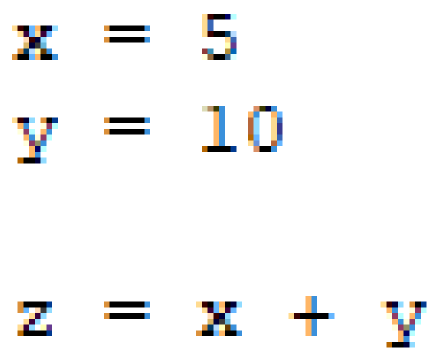


```
>>>
What is your name? - Jim
Hello Jim
```

<u>Assignment</u>

Assignment refers to when we give a variable a value. In Python we use an = symbol to do this.

Below is an example of assignment. In the below example we are assigning z the value of whatever x + y is:



```
x = 5
y = 10

z = x + y
```

Basic Programming Constructs

When producing a computer program, there are three main constructs that are used. The word construct means to make or build something. When we program, we use one or more of these constructs to create a program.

Sequencing:

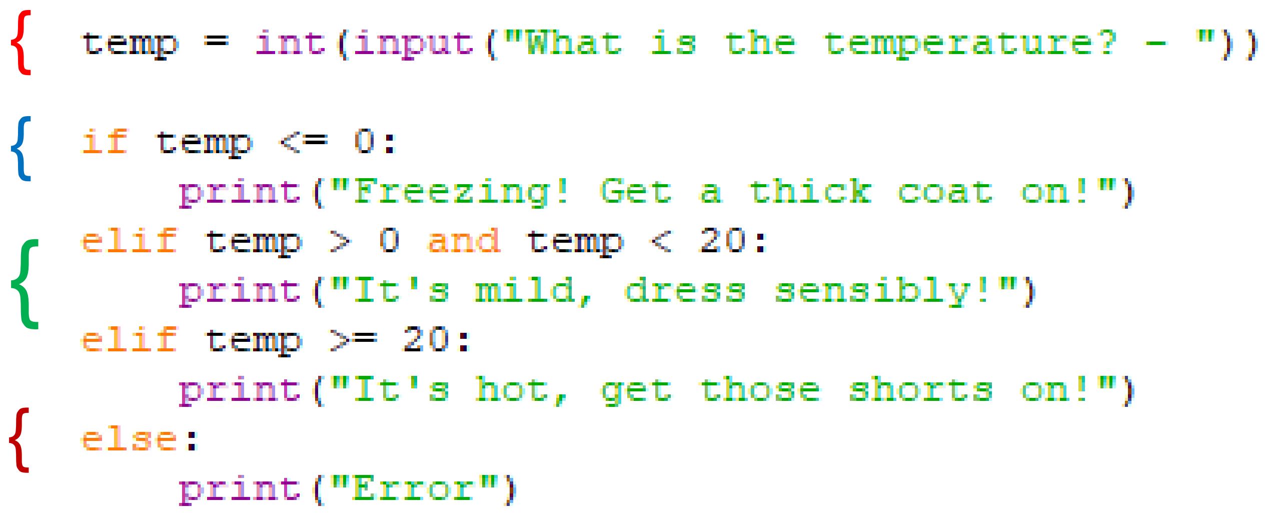
Sequencing is the process of ensuring all our instructions/pieces of code are in order, so they execute one after the other. It is important that our instructions are in order, otherwise the program will either fail, or fail to execute the way we want it to.

Selection:

Selection is the process of providing an option (or set of options) within the program, all of which allow for different outcomes depending on which option is selected.

For example, imagine your program asked your user to type in the temperature outside. We could use selection to provide different outcomes depending on what temperature they type in e.g. if the user types in a number less than 0 the program could say “Freezing! Get a thick coat on!”. However, if the user types in a number bigger than 20 it could say “It’s hot, get those shorts on!”.

In Python we use if statements to utilise selection. Below is an example of selection, using the example from above about the temperature:



```
temp = int(input("What is the temperature? - "))

if temp <= 0:
    print("Freezing! Get a thick coat on!")
elif temp > 0 and temp < 20:
    print("It's mild, dress sensibly!")
elif temp >= 20:
    print("It's hot, get those shorts on!")
else:
    print("Error")
```

{ In this section we are getting our input from our user. We have introduced a new bit of code here however, utilising int. We will discuss why in a later topic!

{ Here we have our first condition. The condition here is that the value inside temp (whatever the user typed in as their temperature) must be less than or equal to 0. If it is, then the code that is directly underneath it and indented will execute. If the temp is not less than or equal to 0, the program will simply skip to the next condition, and the indented code will not execute.

{ Here we have our second and third conditions. These follow the exact same rule as the one above. The code that is indented will only execute if the condition is met. Let's take the first elif condition. If the temp is greater than 0 and less than 20, then the program will say "It's mild, dress sensibly!". Both these conditions must be met here for this bit of code to execute. Again, we will talk more about the use of and later on! The second condition is checking whether the temp is greater than or equal to 20. Again, if it is, then the code directly underneath and indented will execute.

{ Finally we have our else clause. If the program gets to this part of the code then none of the above conditions have been met. It is always useful to use this as an error/invalid statement.

"When using a condition, you must always have 1 if (no more, no less), and 1 else (no more, no less). You can then have multiple elif's. The if signals the start of the condition, and the else signals the end. You cannot start a new condition without ending the previous one (unless it is indented)."

Below you can see what the program looks like in the shell once run:

```
>>>
What is the temperature? - -3
Freezing! Get a thick coat on!
>>> ==============================
>>>
What is the temperature? - 5
It's mild, dress sensibly!
>>> ==============================
>>>
What is the temperature? - 20
It's hot, get those shorts on!
```

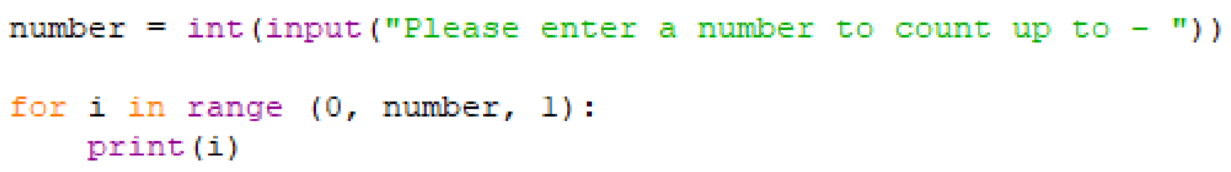
Iteration:

Iteration is the process of repeating a section of code. There are two types of iteration:

- Count Controlled Iteration – This is when the code is repeated a specific number of times, regardless of any condition. This means once the code has repeated this number of times it will stop.
- Condition Controlled Iteration – This is when the code is repeated until a condition is met. This means there is no specific number of times the code will repeated (or if it will even execute at all). The code being repeated is dependent on the condition being met.

Count Controlled Iteration:

In Python we use for loops for count controlled iteration. Below is an example of count controlled iteration. In the example we are asking the user to enter a number to count from 0 up to. The program will then count up to that number, and stop at that number. This means the code within the for loop will execute this exact number of times:

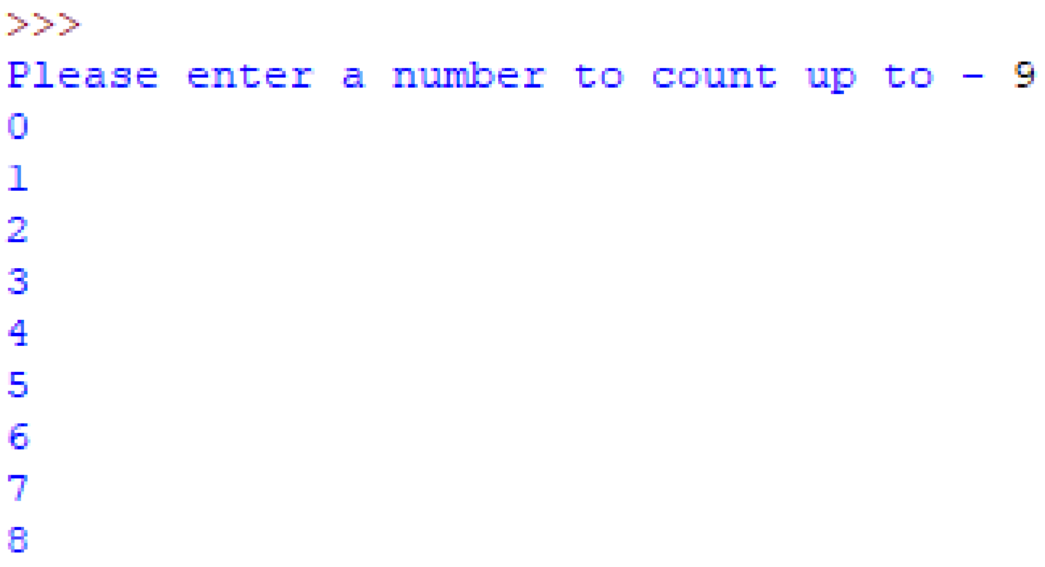


```
number = int(input("Please enter a number to count up to - "))

for i in range (0, number, 1):
    print(i)
```

Here the user in inputting their number, as explained in previous examples.

Here the program is using a for loop. It is going to output all the numbers, starting from 0, and going up to the users number, increasing by 1 each time. If we didn't use a for loop to do this, we would have to write out the same code over and over again which would waste time and not be efficient. Using the for loop allows us to condense the amount of code we have to write into 3 lines.

Below is what the program looks like once run:

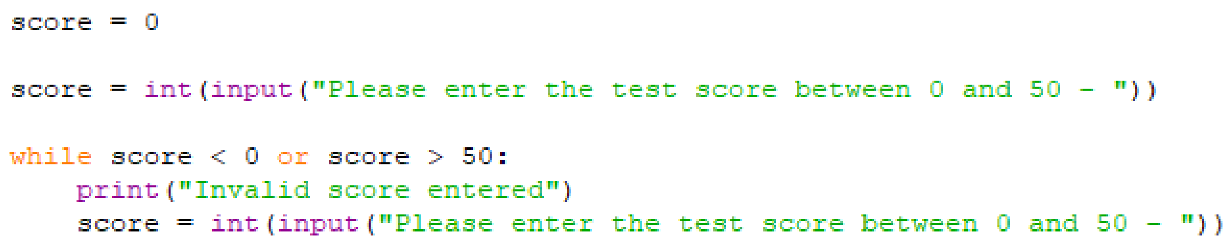


```
>>>
Please enter a number to count up to - 9
0
1
2
3
4
5
6
7
8
```

As you can see, the program stops when it gets to the number the user inputted. It does not print out that number. An important point to remember!

Condition Controlled Iteration:

In Python we use while loops for condition controlled iteration. Below is an example of condition controlled iteration. In the example we are asking the user to enter a test score between 0 and 50 for a test. If the user enters anything outside of this range (e.g. -4, 51 etc.) then the program will repeat the code, requesting the user re-enter the score. This code will repeat until the score entered meets the criteria of being between 0 and 50:



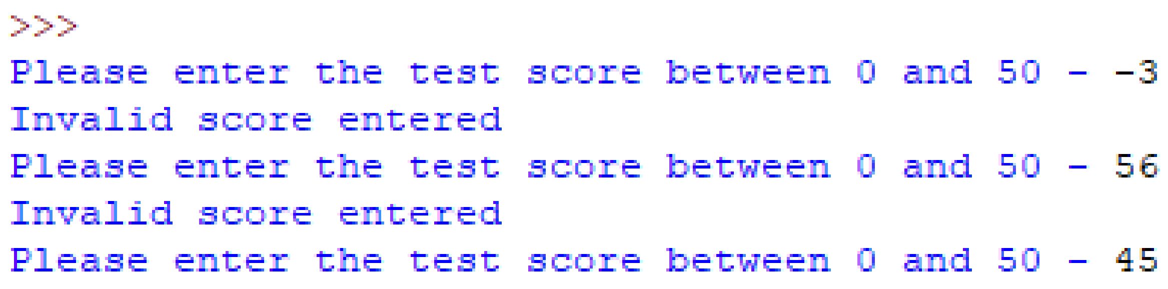
```
score = 0

score = int(input("Please enter the test score between 0 and 50 - "))

while score < 0 or score > 50:
    print("Invalid score entered")
    score = int(input("Please enter the test score between 0 and 50 - "))
```

Here, as in previous examples, we are getting our user to input some data. In this case, they are going to input the test score.

In this section is where we have our iteration taking place. The condition for the iteration taking place is that the score must be less than 0, or greater than 50. This means the user did not enter a score that is within our range. If this condition is met (e.g. they enter 57) then the code that is indented directly underneath will execute. In this case it will tell the user their score is invalid, and request they input a new test score. Once the user has done this the program will then go back to the condition and check it again. Again, if the user enters an invalid test score (e.g. this time they enter -3) then the indented code will repeat again. This will happen over and over until the user enters a test score that is valid.

As you can see, this is why it is called condition controlled iteration. The code that is indented is dependent on the condition above it, and will only repeat if that condition is met. If the user inputted a valid test score first time, the code would never execute. Alternatively, if the user inputted 100 invalid test scores one after the other, the code would continue to repeat.

Below is what the program looks like once run:



```
>>>
Please enter the test score between 0 and 50 - -3
Invalid score entered
Please enter the test score between 0 and 50 - 56
Invalid score entered
Please enter the test score between 0 and 50 - 45
```

Basic String Manipulation

As mentioned earlier, a string refers to alphanumeric characters being stored. In Python, strings are stored inside speech marks (“ “) and appear green, as can be seen below:

```
string1 = "This is a string"
string2 = "S0 i5 thls 3ven th0ugh lt ha5 numb3rs"
string3 = "You can store symbols in strings too, see ... #!£()*$"

not_a_string1 = 56
not_a_string2 = Hello
```

As you can see the data inside the variables string1, string2, and string3 are all stored inside speech marks and have turned green, indicating they are strings.

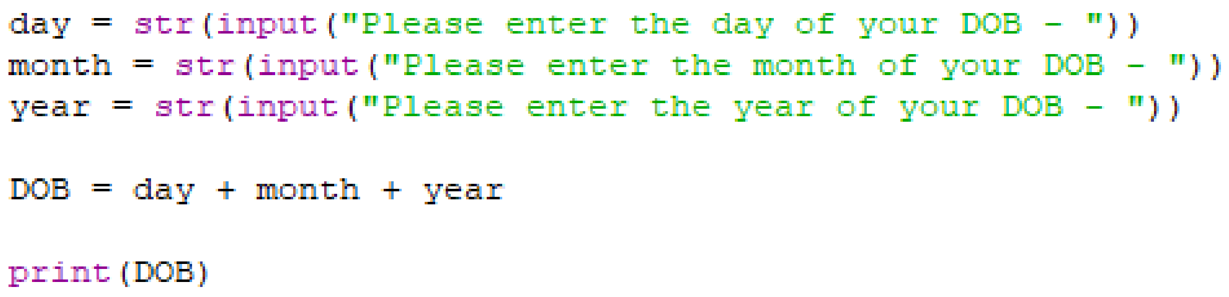
The data inside the variables not_a_string1 and not_a_string2 however are not green and have not been stored inside speech marks. These are therefore not strings. The data inside not_a_string1 would be stored as an integer (we will talk more about this later), and the data inside not_a_string2 would cause a syntax error.

String Manipulation refers to the process of being able to adapt or edit a string. We might do this in a program so that we can ensure what the user has to enter is both easy for the user to understand and is also then in the format we require it to be in.

We are now going to cover string concatenation and splitting strings. In the splitting strings section we will be talking about lists, so it might be worth you going and revising that section first if you are unsure what a list is!

String Concatenation

String concatenation simply refers to the process of bringing together (concatenating) two or more strings. In Python this is done using a +. In the example below we are requesting the user input their date of birth. First they input their date, then month, then year. We are storing each of these as strings (we will see what happens if we store them as integers and try to concatenate them after):



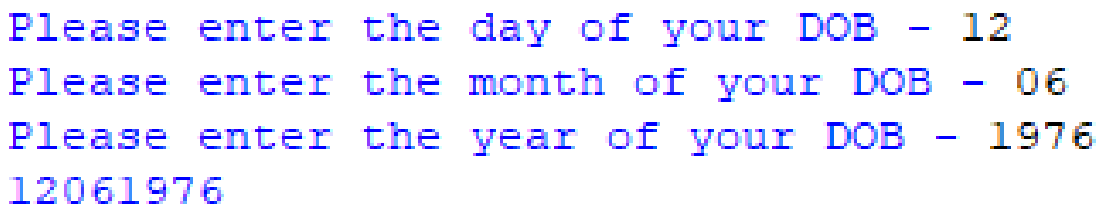
```python
day = str(input("Please enter the day of your DOB - "))
month = str(input("Please enter the month of your DOB - "))
year = str(input("Please enter the year of your DOB - "))

DOB = day + month + year

print(DOB)
```

Here we are getting the user to input their day, month, and year of their date of birth, and storing each of these in separate variables, each as strings.

In this line of code we are now concatenating the three strings together into one single string using the +. The new, concatenated string will now be stored inside the variable called DOB.

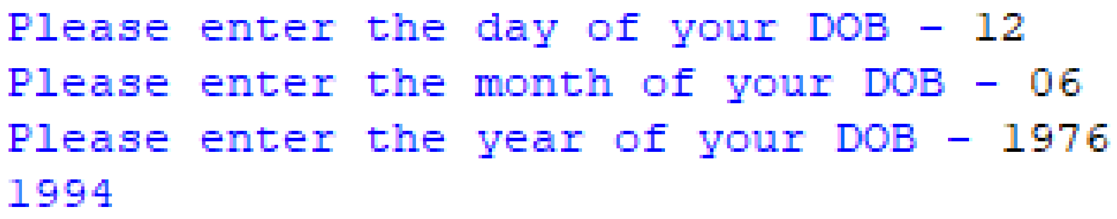
Finally, we are outputting the new string so the user can see this.

Below is what the program looks like once it is run:



```
Please enter the day of your DOB - 12
Please enter the month of your DOB - 06
Please enter the year of your DOB - 1976
12061976
```

If you attempt to concatenate integers together, it will simply add them up, like below. We have now stored each value inputted as an integer and attempted to concatenate them:



```
Please enter the day of your DOB - 12
Please enter the month of your DOB - 06
Please enter the year of your DOB - 1976
1994
```

Splitting Strings

Sometimes we want to be able to split up a string into multiple, separate strings. It might be easier for the user to input all their details in one entry, with our program then separating the various pieces of data up and storing them separately.

In this section we will cover how to split strings in the following ways:

- At specific characters
- At every character

We will also cover how to store these new strings in either a list, or separate variables.

Splitting at Specific Characters:

In Python, when splitting a string, we use the .split function, which is built into Python. However, what we must specify where we want to split the string at.

Let's go back to our date of birth example. Most often, when people enter their date of birth, they enter it by separating the different parts with a forward slash (/). In the example below we are requesting the user enter their date of birth, separating each part with a /. We are then splitting this string up at each point where the user enters a /, and in this case storing each new string in a new variable:

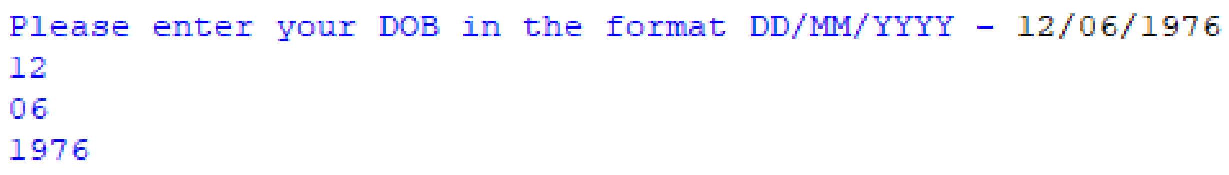
```
DOB = input("Please enter your DOB in the format DD/MM/YYYY - ")

day, month, year = DOB.split("/")

print(day)
print(month)
print(year)
```

Here we are requesting the user enter their DOB in the specified format. You will notice we haven't put str before the input. If we don't specify what data type we want to use when getting input, Python will automatically store it as a string.

Here we are splitting the string entered by the user. Firstly we have created three new variables for each of the different parts of the DOB. We are then taking the value inside the variable DOB and performing a split on it, splitting it wherever there is a /. It's a bit like slicing a cake. The first slice will be stored in the variable day, the second piece will be stored in the variable month, and the final piece will be stored in the variable year.

Finally, we are printing out the values of each variable so the user can see what is stored inside each variable.

Below is what the program looks like once it is run:

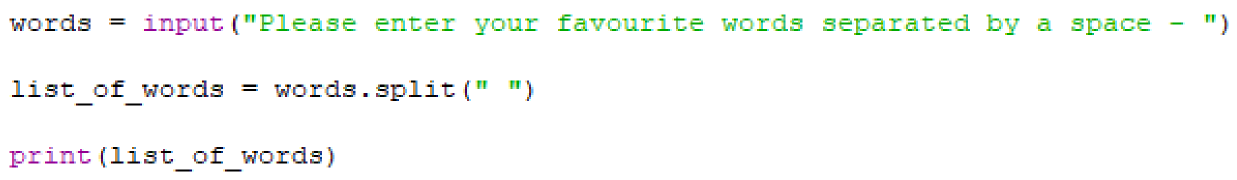


```
Please enter your DOB in the format DD/MM/YYYY - 12/06/1976
12
06
1976
```

As you can see, the input from the user (which was one whole string) has been split at each point where there is a /, and the new mini strings have been stored inside new variables.

However, it is not always possible to know how many variables you may need. In the example above we knew we needed three variables. If you don't know how many variables you need though, then you can't store your new strings in variables. This is because when splitting the string you must specify the variable names.

Therefore, we can store the new split strings inside a list. This means we have an open amount to store, and it doesn't matter how much the user inputs. In the example below we are asking the user to enter their favourite words, separated by a space. In this case we don't know how many words they might input. They could input 1, 5, 205! We don't know. Therefore, in this case, it is best to store the split strings in a list:



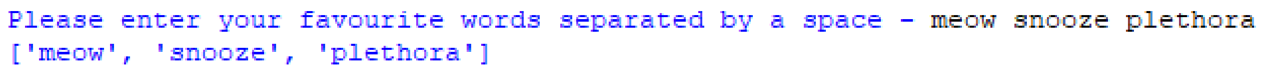
```
words = input("Please enter your favourite words separated by a space - ")

list_of_words = words.split(" ")

print(list_of_words)
```

Here we are getting our input from the user, specifying we want the words separated by a space.

Here we are splitting the words up. Again, we are using the .split function, and have specified that we want to split at each point where there is a space (shown by the “ “. Whatever we write in between the “” is where we want to split. In this case we have just put a space in). However, in this example we have created a new list, and each string will be stored in its own index within the list.

Finally we are outputting the contents of the list to the user so they can see what words are in the list.

Below is what the program looks like once it is run:



```
Please enter your favourite words separated by a space - meow snooze plethora
['meow', 'snooze', 'plethora']
```

As you can see the user inputted their favourite words. The words have then been split and stored in their own place in the list. The user inputted one whole string, and now we have three strings!

Splitting at Every Character:

Sometimes we don't want to split at a specific character, but more so at every character. To do this, rather than using .split, we use the list function. This will create a list from the split string. Let's take the example of creating a password. In the example below we are asking the user to enter a word that is memorable to them. We are then going to split it at each individual character, and store each character in its own place in a list:

```
word = input("Please enter your memorable word - ")

list_of_letters = list(word)

print(list_of_letters)
```

Here we are getting the user to input their memorable word.

Here we have created a new, empty list. Our program will then take the string stored inside the variable word, and split it at each character, storing each character into the new list called list_of_letters.

Finally we are outputting the contents of the list so the user can see that each letter has been stored in its own place in the new list.

Below is what the program looks like once it is run:

```
Please enter your memorable word - Programming
['P', 'r', 'o', 'g', 'r', 'a', 'm', 'm', 'i', 'n', 'g']
```

As you can see our word has been split at every character and each character is now its own string in a list.

What we could do then is create a unique password from this, combining random letters from within the list e.g. combining indices 1, 3, 6, and 5. We could also use it when a user logs in. Rather than entering their whole password, they enter specific characters such as the first, third, and eighth letters.

Basic File Handling Operations

When creating a computer program, we often need to store data for outside of the program being run. We can't do this with a variable or a list. Any data stored inside these whilst the program is run (e.g. a users name) will be lost when the program is closed.

A text file allows us to store data outside of the program being run. This means when you close the program down the data will still be stored within the text file.

In this section we are going to cover how to open, read, write to, and close a text file.

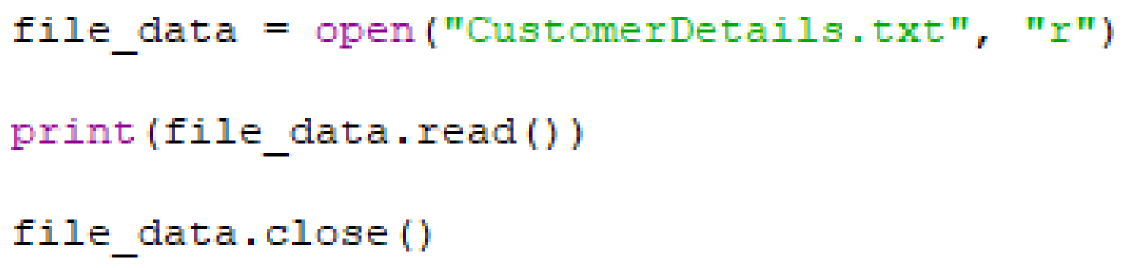
"It's important to note here that whatever you do to a text file (whether you read from it, write to it etc. you must ALWAYS open it first, and close it last. Otherwise your program won't know firstly what text file you are using, and won't save the changes to the text file."

Opening a Text File:

When opening a text file in Python we use the open command. This tells our program that we want to open (or create if it doesn't already exist) a text file with the name we give it. We also must tell the program what mode we want to open the text file in:

r = Read mode. Opening a text file in this mode means you cannot make any changes to the text file, only read it.
w = Write mode. Opening a text file in this mode means you can make changes to the text file, however the current contents will be erased and overwritten.
a = Append mode. Opening a text file in this mode means you can make changes to the text file, and add new data to the end of the text file, without erasing what is already there.

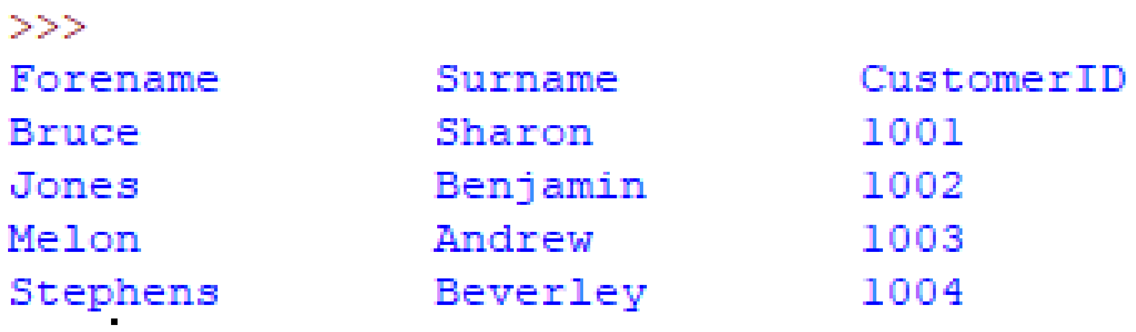
In the example below we are opening a text file called 'CustomerDetails' in read mode. For the purpose of this example the text file has already been created and populated with some data. We have then outputted the contents of the text file and closed the text file (however, we will discuss those in the following sections. For this section just focus on the top line of code):



```
file_data = open("CustomerDetails.txt", "r")

print(file_data.read())

file_data.close()
```

Here we have used the open function to open up a text file called CustomerDetails.txt. It is important this part is exactly how the text files name appears, with the .txt at the end. If you create a new text file here with spaces in, be aware Python will fill those spaces in with underscores (_)! We have then opened the text file up in read mode. This means we just want to see the contents of the text file and not make any changes. Finally, all this data has to go somewhere in our program. We have created a variable now to store this data in called file_data. This is because we don't directly interact with the text file within the program, but with the variable where the data is stored.

Below is what the program looks like once it is run:



```
>>>
Forename          Surname           CustomerID
Bruce             Sharon            1001
Jones             Benjamin          1002
Melon             Andrew            1003
Stephens          Beverley          1004
```

"The process of opening a text file is the same, no matter what mode you open it up in. You always need your open function, the name of the text file, the mode, and a variable to use to interact with/store the text files data in"

Reading a Text File:

If you want to just see the entire contents of a text file without making any changes, you would use the .read() function. This function would be used to output the entire contents of the text file. In the example below we are using the same code from opening the text file, however we will now focus on the second line of code:

```
file_data = open("CustomerDetails.txt", "r")

print(file_data.read())

file_data.close()
```

Here, after opening the text file, we are reading the contents of the text file using the .read() function. What is happening here is the entire contents of the text file is being read, and then using the print command we are outputting this to the user.

Below is what the program looks like once it is run:

```
>>>
Forename          Surname           CustomerID
Bruce             Sharon            1001
Jones             Benjamin          1002
Melon             Andrew            1003
Stephens          Beverley          1004
```

As you can see the entire contents of the text file has been outputted.

However, sometimes you might want to output the contents of the text file line by line. You can use the .readline() function to do this. This will take each line of the text file as a separate string, compared to the program above which outputs the whole text file as one string. Below is an example of a program which is outputting the first 2 lines of the text file CustomerDetails:

```
file_data = open("CustomerDetails.txt", "r")

{ print(file_data.readline())
  print(file_data.readline())

file_data.close()
```

{ Here in this section we have used the .readline() function twice, which means the program will read the first two lines of the text file, and output them using the print command.

Below is what the program looks like once it is run:

```
>>>
Forename          Surname          CustomerID

Bruce             Sharon           1001
```

As you can see the first two lines of the text file have been outputted, both as separate strings. If we wanted to print out multiple lines, we could of course make our program more efficient by using a for loop!

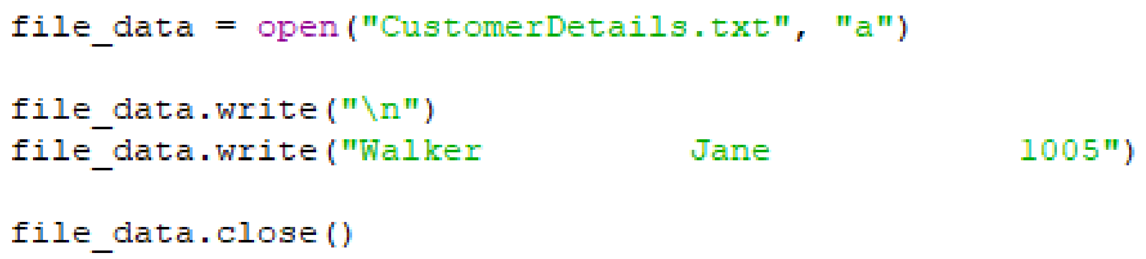
"You might be wondering what the empty brackets at the end of the .readline() is for. If you want to specify how many bytes (e.g. characters) you want to read then you would put a number inside here. In our examples we have left it blank as we wanted to read the whole line"

<u>Writing to a Text File:</u>

When writing to a text file we have two choices for the mode:

- Using the 'w' mode – This will remove all data that is currently stored inside the text file. Only use this mode if you are writing to the text file for the first time, or want the text files complete contents wiped
- Using the 'a' mode – This will add to the end of your text file. Use this mode if you want to add additional data to the text file without removing what you have previously written.

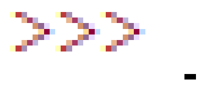
The function we then use in Python to write to a text file is .write. In the below program we are going to append to the CustomerDetails text file a new customer. We have therefore opened the text file up in the append mode using 'a'. We are then using the .write function, before closing the text file:



```
file_data = open("CustomerDetails.txt", "a")

file_data.write("\n")
file_data.write("Walker              Jane                1005")

file_data.close()
```

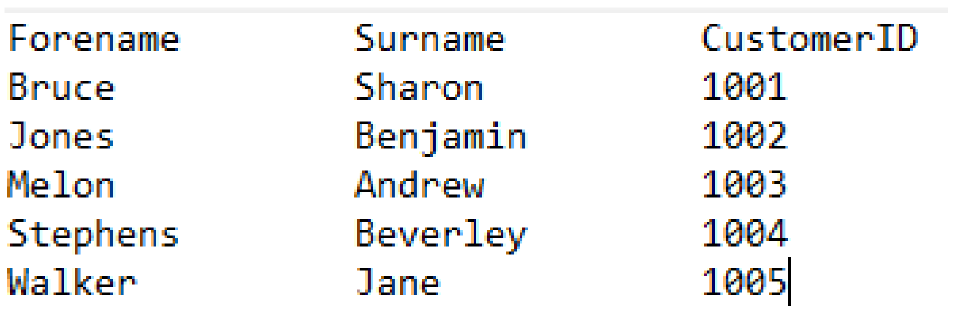
Firstly we have inserted a new line, to ensure the data being written to the text file is placed on a new line (it won't be if we don't do this). We have then written our new data into the text file.

Below is what the program looks like once it is run:



```
>>>
```

As you can see, nothing! This is because we aren't outputting anything. This doesn't mean it hasn't worked! If we open up the text file we can now see the new customer has been added:



```
Forename        Surname         CustomerID
Bruce           Sharon          1001
Jones           Benjamin        1002
Melon           Andrew          1003
Stephens        Beverley        1004
Walker          Jane            1005
```

Closing a Text File:

As mentioned earlier, no matter what you do with a text file, you must always close it! If you make any changes to the contents of a text file, closing it effectively saves the text file. To close a text file we simply use the .close() function. It couldn't be easier!

Below is the code we used earlier when opening our CustomerDetails text file, however this time we have highlighted closing the file:

```
file_data = open("CustomerDetails.txt", "r")

print(file_data.read())

file_data.close()
```

Here to close the text file we have written the name of our variable used to interact with the text file, followed by the .close() function. The text file once run will be closed at this part in our code. If we wanted to use it again we would have to reopen it again.

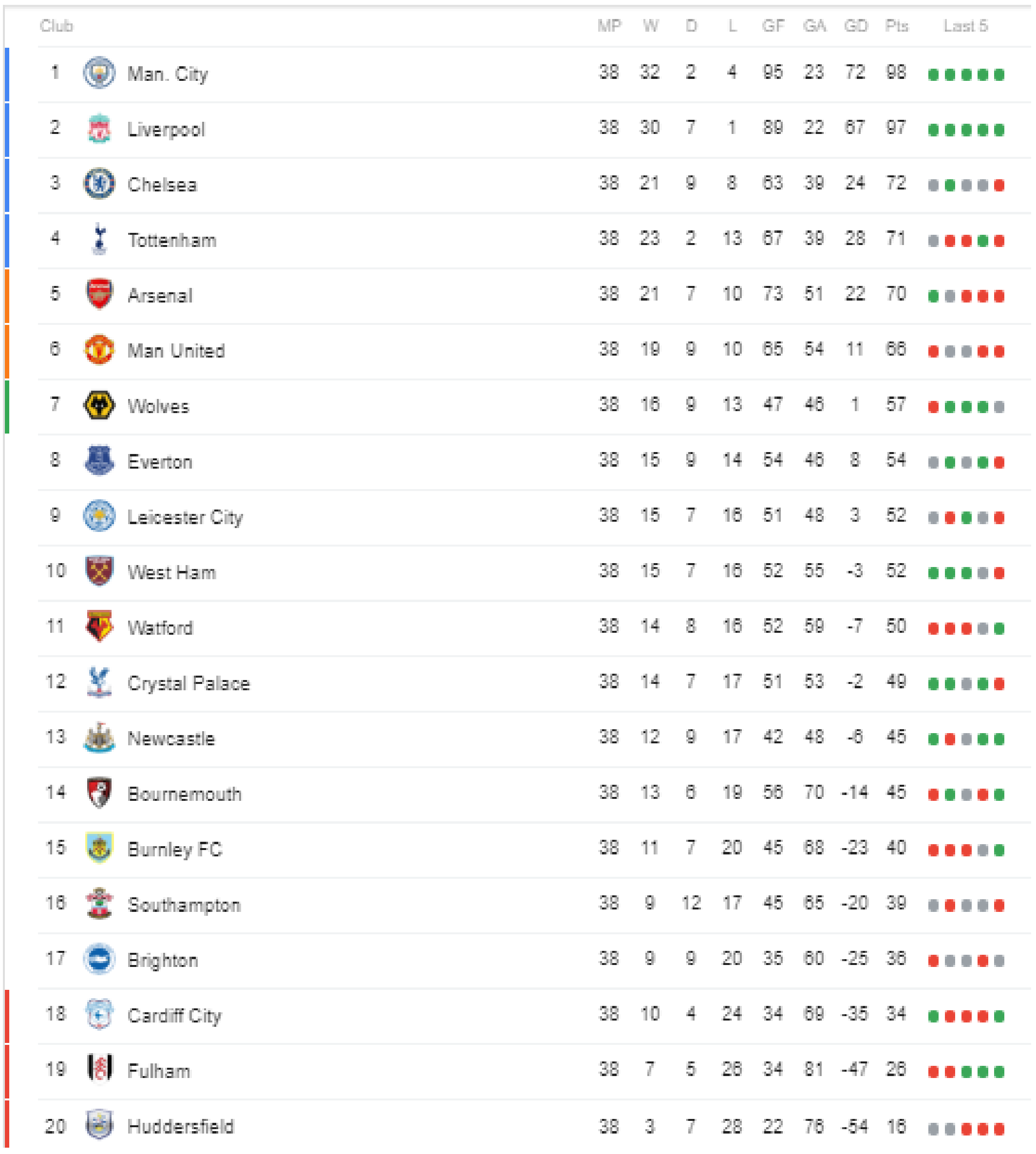
The use of records to store data

When storing data, we of course have those methods already mentioned (variables, lists, text files). However, companies don't often store their data in these ways. Variables and lists are not permanent and are simply too small, and text files do not offer the security required when storing confidential information such as that of a customer. Companies therefore often store their data inside a database.

A database is an organised way of storing data that can be made secure. Data is organised into tables, using fields to break up each record.

A record is all the data about a certain data subject. A field is a specific category which is used to collect data about a certain aspect of the data subject.

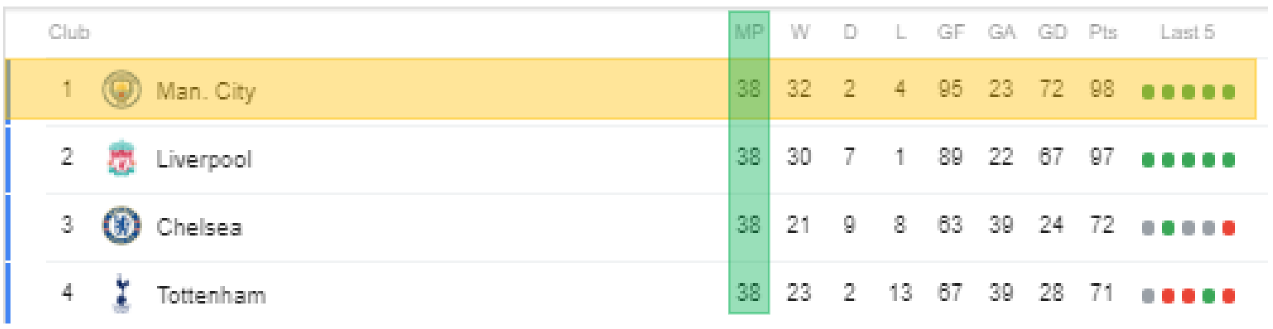
Let's look at a real life example of a table within a database. Below is the Premier League table from the football 18/19 season. The table is called **Premier**:



	Club	MP	W	D	L	GF	GA	GD	Pts	Last 5
1	Man. City	38	32	2	4	95	23	72	98	🟩🟩🟩🟩🟩
2	Liverpool	38	30	7	1	89	22	67	97	🟩🟩🟩🟩🟩
3	Chelsea	38	21	9	8	63	39	24	72	⬜🟩⬜⬜🟥
4	Tottenham	38	23	2	13	67	39	28	71	⬜🟥🟥🟩🟥
5	Arsenal	38	21	7	10	73	51	22	70	🟩⬜🟥🟥🟥
6	Man United	38	19	9	10	65	54	11	66	🟥⬜⬜🟥🟥
7	Wolves	38	16	9	13	47	46	1	57	🟥🟩🟩🟩⬜
8	Everton	38	15	9	14	54	46	8	54	⬜🟩⬜🟩🟥
9	Leicester City	38	15	7	16	51	48	3	52	⬜🟥🟩⬜🟥
10	West Ham	38	15	7	16	52	55	-3	52	🟩🟩🟩⬜🟥
11	Watford	38	14	8	16	52	59	-7	50	🟥🟥🟥⬜🟩
12	Crystal Palace	38	14	7	17	51	53	-2	49	🟩🟩⬜🟩🟥
13	Newcastle	38	12	9	17	42	48	-6	45	🟩🟥⬜🟩🟩
14	Bournemouth	38	13	6	19	56	70	-14	45	🟥🟩⬜🟥🟩
15	Burnley FC	38	11	7	20	45	68	-23	40	🟥🟥🟥⬜🟩
16	Southampton	38	9	12	17	45	65	-20	39	⬜🟥⬜⬜🟥
17	Brighton	38	9	9	20	35	60	-25	36	🟥⬜⬜🟥⬜
18	Cardiff City	38	10	4	24	34	69	-35	34	🟩🟥🟥🟥🟩
19	Fulham	38	7	5	26	34	81	-47	26	🟥🟥🟩🟩🟩
20	Huddersfield	38	3	7	28	22	76	-54	16	⬜⬜🟥🟥🟥

As you can see from the table, all the data is organised making it much easier to locate specific pieces of data such as how many wins a certain team got, or how many goals a certain team conceded.

We have now reduced the amount of data we are looking it (just for the purposes of definitions). Here are the top four teams from the table, with some areas highlighted:



	Club	MP	W	D	L	GF	GA	GD	Pts	Last 5
1	Man. City	38	32	2	4	95	23	72	98	🟩🟩🟩🟩🟩
2	Liverpool	38	30	7	1	89	22	67	97	🟩🟩🟩🟩🟩
3	Chelsea	38	21	9	8	63	39	24	72	⬜🟩⬜⬜🟥
4	Tottenham	38	23	2	13	67	39	28	71	⬜🟥🟥🟩🟥

🟨 Here we have a record within the table. As you can see, all the data stored about the data subject (in this case Manchester City) is the record.

🟩 Here we have a field within the table. In this case we have selected the MP field (matches played). This means all the data stored within this column is only regarding the number of matches each team played.

As we move into queries it is important to remember the difference between a record and a field!

The use of SQL to search for data

SQL (Structured Query Language) is a language used to manage, maintain, and query data stored within a database.

When writing an SQL query there are three parts:

- SELECT – This part of the query determines what fields you want to select, and therefore show in the final output. You can either select individual fields (separated by a comma) or all fields using an Asterisk (*).
- FROM – This part of the query indicates which table you are querying
- WHERE – This is the condition you are setting. You do not need to have this part if you do not want to set a condition.

Let's do a couple of examples together of writing an SQL query. To do this we are going to use the **full** Premier League 19/20 table shown on page 65.

Q: Write the SQL to locate all the records where the wins is 14

Firstly, we need our SELECT statement. The question does not specify which fields to pull out, therefore we will pull them all out. To do this, we will use the Asterisk. Here is our SQL query so far:

```
SELECT *
```

Secondly, we now need to indicate which table we are using. As mentioned above the table we are using, it is called Premier. Here is our SQL query now with the FROM statement attached:

```
SELECT * FROM Premier
```

Finally, we need to add our condition (if we need one). In this case the question is asking us to locate the records where the wins is 14. That is our condition, that the team must have 14 wins. We firstly need to know what the name of the field is where the wins are stored. In this table it is called W. Here is our final SQL query with the condition now added:

```
SELECT * FROM Premier WHERE W = 14
```

<u>Q: Write the SQL to locate all the team name and points where the losses is less than 11.</u>

Firstly, we need our SELECT statement. In this question we have got specific fields to extract (team name and points). We first need to know the exact names of those fields. In this table the team name is the field called club, and the points is called Pts. Here is our SQL query so far:

```
SELECT Club, Pts
```

Secondly, we now need to indicate which table we are using. As mentioned above the table we are using, it is called Premier. Here is our SQL query now with the FROM statement attached:

```
SELECT Club, Pts FROM Premier
```

Finally, we need to add our condition (if we need one). In this case the question is asking us to locate the records where the number of losses is less than 11. That is our condition, that the team must have less than 11 losses. We firstly need to know what the name of the field is where the losses are stored. In this table it is called L. Here is our final SQL query with the condition now added:

```
SELECT Club, Pts FROM Premier WHERE L < 11
```

"You might be asked to write an SQL query using more than one condition e.g. where the wins is greater than 15 and losses is less than 10. If that is the case, you simply separate them with the relevant Boolean operator (AND or OR). In this case it would be:"

```
WHERE W > 15 and L < 10
```

The use of Arrays

In this section we are going to talk about lists (for the purpose of the GCSE course an array and a list are the same thing), and how to create them in Python.

Firstly, we must define what a list is. A list allows you to store multiple pieces of data in one location. This is different to a variable which only allows you to store one piece of data. There are two types of list you must be aware of:

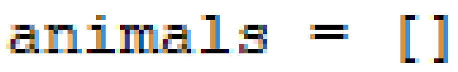
- One Dimensional List – In this list we store one piece of data per data subject
- Two Dimensional List – In this list we store more than one piece of data per data subject

You do not need to know how to create a two dimensional list, only a one dimensional list. Therefore, this is what we will now focus on.

Creating a One Dimensional List:

As mentioned above, a list allows us to store multiple pieces of data within one location. A one dimensional list is when we store one piece of data per data subject.

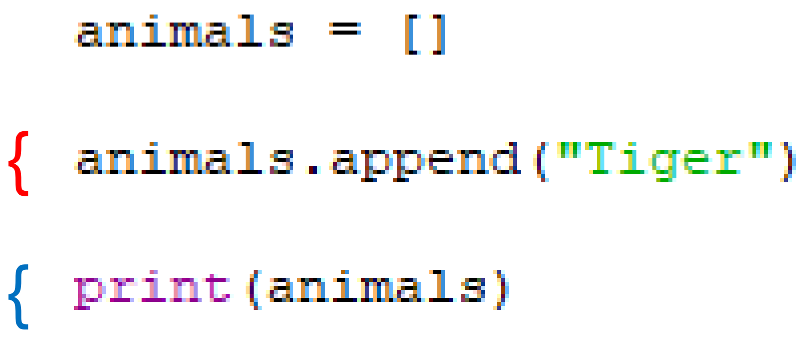
Imagine we run a zoo. We want to store all the animals located in location A of the zoo in one place, rather than separate variables. In the program below we have created a list called animals, ready to store some animals in it:



```
animals = []
```

As you can see, when creating a list you always use square brackets [ ]. This represents the creation of a list. At the moment our list is empty, so let's add some animals!

Firstly let's add a Tiger in. To do this we are going to use the .append function. This will add our new animal to the end of our list. We will then output the contents of the list so the user can see it has been added:

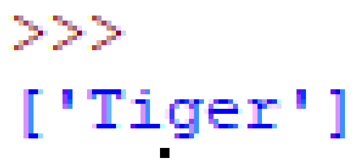


```
animals = []

{ animals.append("Tiger")

{ print(animals)
```

{ Here we have used the .append function to add the animal “Tiger” to the end of our list.

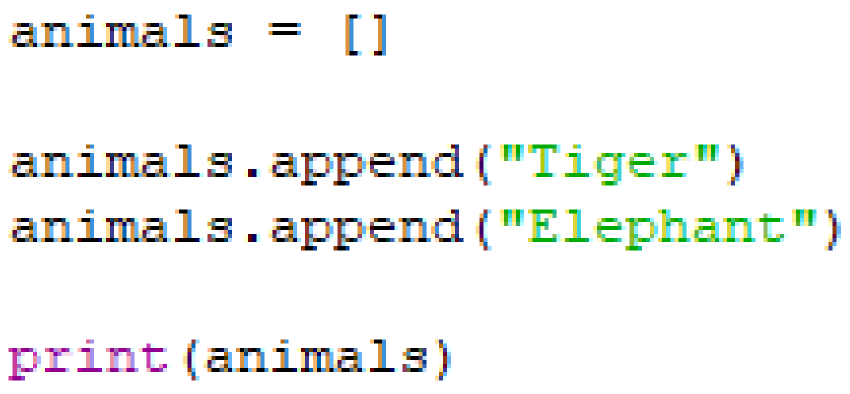
{ Here we have used the print command to output the contents of our list.

Below is what the program looks like once it is run:



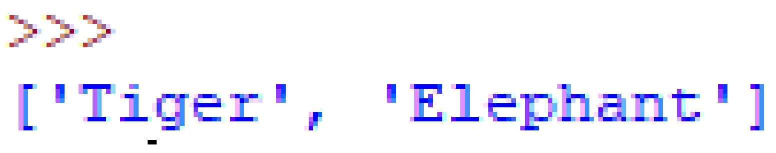
```
>>>
['Tiger']
```

Now let’s add a second animal. Again we will use the .append function. This time we will add Elephant:



```
animals = []

animals.append("Tiger")
animals.append("Elephant")

print(animals)
```

Below is what the program looks like once it is run:



```
>>>
['Tiger', 'Elephant']
```

As you can see, the animal Elephant has now been appended to the end of our list.

Don’t forget, if we were to remove the code we have added appending the answers and run it again, the list would be empty! A list is only temporary storage whilst the program is running don’t forget!

<u>Lists and Indices:</u>

We have now populated our list manually with some animals below:

```
animals = ["Tiger", "Elephant", "Rhino", "Cheetah", "Deer"]
```

You will notice each animal has its own place in the list, separated by a comma. It's a bit like a box being split up into different sections, and each section storing one piece of data.

When referring to these individual places, we call them indices. Each piece of data can be referred to in a program using its index. A list always starts at index 0, a very important point!

Let's add our indices under each data so we can understand this point easier:

```
animals = ["Tiger", "Elephant", "Rhino", "Cheetah", "Deer"]
               0         1           2         3         4
```

So, for example, the data stored inside index 2 would be "Rhino". Even though "Elephant" is stored in what looks like position 2 in the list, the first index is 0 and not 1. To prove this point, below is a program that is outputting the data inside index 2 in the list:

```
animals = ["Tiger", "Elephant", "Rhino", "Cheetah", "Deer"]

print(animals[2])
```

As usual we have used the print command. However, instead of printing out the entire contents of the list like we did before, we have now specified which index we want to output. In this case, the data inside index 2.

Below is what the program looks like once it is run:

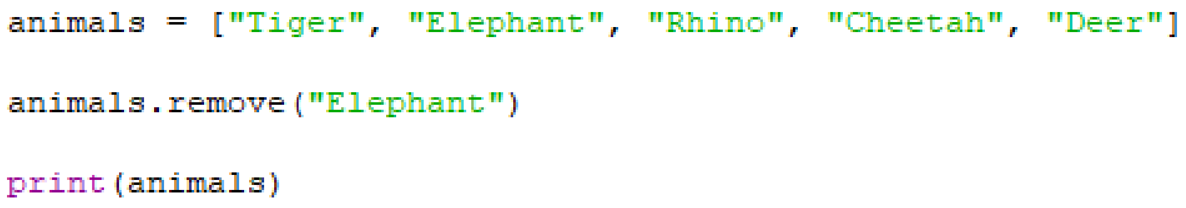
```
>>>
Rhino
```

As you can see, the data from index 2, "Rhino", has been outputted.

<u>Other Code:</u>

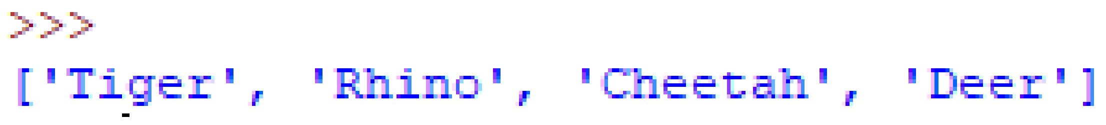
There are other things we can do with a list.

Firstly, we can remove certain pieces of data using the .remove function. Below is a program which is removing a piece of data:



```
animals = ["Tiger", "Elephant", "Rhino", "Cheetah", "Deer"]

animals.remove("Elephant")

print(animals)
```

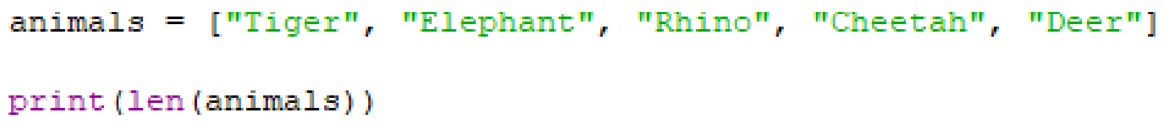
Below is what the program looks like once it is run:



```
>>>
['Tiger', 'Rhino', 'Cheetah', 'Deer']
```

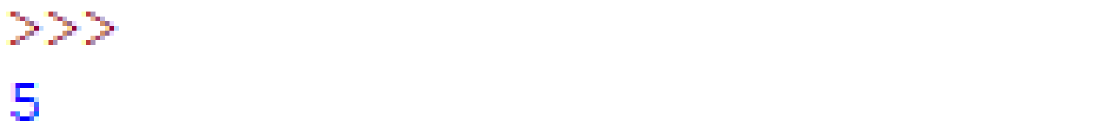
As you can see, the data "Elephant" has now been removed.

We can also find out the length of a list (how many pieces of data are inside a list) using len. Below is a program that outputs the length of the list animals:



```
animals = ["Tiger", "Elephant", "Rhino", "Cheetah", "Deer"]

print(len(animals))
```

Below is what the program looks like once it is run:



```
>>>
5
```

As you can see, the program has outputted how many pieces of data are stored within the list, in this case it is 5.

The use of Sub-Programs

As you can imagine, building computer programs can be big work! Often you can find yourself building programs with thousands of lines of code, which can quickly become cluttered and unorganised.

Using sub-programs allows us to break up our code into smaller segments/sections, which makes it much more organised and easier to manage. When creating a sub-program you still write the code you want to want the same way you would do. However, with a sub-program you effectively give it a name. Then, anytime you want to use that section of code, you simply 'call' its name, and the code will run.

Let's look at an example. In the program below we are wanting to output the messages repeatedly (forget about loops for this!). Here we have just used print commands to do this:

```
print("This is one line of code")
print("It's not my favourite line of code ...")

print("This is another line of code")
print("I prefer this line of code!")

print("This is one line of code")
print("It's not my favourite line of code ...")

print("This is another line of code")
print("I prefer this line of code!")
```

As you can see, we are repeating code so not being efficient, and also the code looks messy. If another programmer wanted to come and look at our code they would find it difficult to find what they are looking for. So, let's make it more organised and efficient!

In the program below we have now broken our code up into two sub-programs, one called 'message1' and one called 'message2'. We are then calling our sub-programs to activate and run the code inside them:

```
def message1():
    print("This is one line of code")
    print("It's not my favourite line of code ...")

def message2():
    print("This is another line of code")
    print("I prefer this line of code!")


message1()
message2()

message1()
message2()
```

Here we have created our first sub-program. We do this by defining it using 'def'. We then give it a name, in this case we have called it 'message1'. We will talk about the brackets in a short while! However, for this example we have left them empty. The code we then want to be part of the sub-program is indented. In this case we just have a couple of outputs, however this of course could be much more complex.

Here we have then created our second sub-program. This sub-program must have a different name to the first sub-program, so we have called it 'message2'. Again, the code inside this sub-program must be indented, and again we have just used some simple outputs.

Finally, here we have 'called' our sub-programs. By simply writing out the name of the sub-program and adding the brackets represents us wanting the code inside those sub-programs to execute. Python here will then locate that sub-program, and execute any code inside. To show you the sub-program can be called more than once in a program we have called each sub-program twice.

Below is what the program looks like once it is run:

```
>>>
This is one line of code
It's not my favourite line of code ...
This is another line of code
I prefer this line of code!
This is one line of code
It's not my favourite line of code ...
This is another line of code
I prefer this line of code!
```

As you can see the code within those sub-programs has been executed when called!

Parameters in Sub-Programs

When using sub-programs we can pass what we call parameters into them. A parameter is a variable that is used as an input to the sub-program. For example, we might gather some data from our user outside of the sub-program, and then pass this data into the sub-routine as a parameter so it can be used in the code of the sub-program. A sub-program is effectively blind to the rest of the code outside of itself, so if you want to use something within it then you must pass it in as a parameter.

Let's look at an example. Here we have created a simple addition program. We are asking the user to input two numbers. We are then passing these as parameters into the sub-program so they can be used within the code of the sub-program:

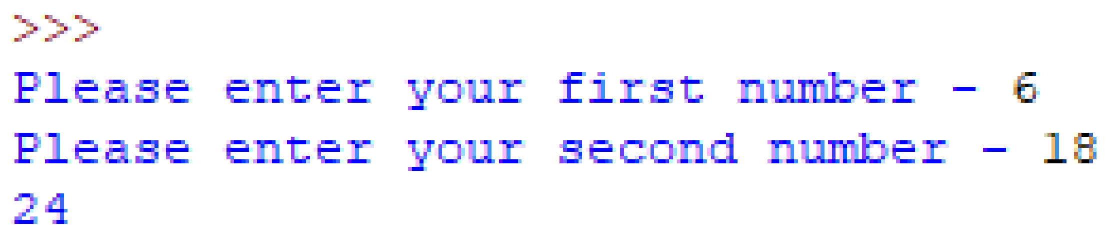
```
def addition(x, y):
    z = x + y
    print(z)

x = int(input("Please enter your first number - "))
y = int(input("Please enter your second number - "))

addition(x, y)
```

{ Here we have defined our sub-program. We have called it ‘addition’, and we have created two parameters. In essence here we are telling the sub-program that two variables with these names (‘x’ and ‘y’) will be passed into it. We have then created the mini addition program adding together x and y and outputting the result.

{ Here we have asked our user to input two numbers, storing each of them in our variables ‘x’ and ‘y’.

{ Finally, here we are calling our sub-program ‘addition’. However, this time we are passing data into sub-program, in this case the variables ‘x’ and ‘y’. This means this data (which is external to the sub-program) will be passed into the sub-program to be used within its code.

Below is what the program looks like once it is run:



```
>>>
Please enter your first number - 6
Please enter your second number - 18
24
```

As you can see, the users two numbers they inputted have been passed into the sub-program and used within the calculation, with the answer being outputted as 24 (6 + 18).

<u>Procedures vs Functions</u>

A common exam is the difference between a procedure and a function. Both are types of sub-program, used to break up and structure our code. However, they do have a slight difference.

A function returns a value. A procedure does not return a value. This is a general rule. So, if you use a sub-program that returns a value to the user (like our addition program above) then you are using a function. If your sub-program does not return a value (it might just be used to get some input for example) then you are using a procedure.

The use of Data Types

When gathering input or using data within a program, it is important it is in the correct format/type. If it is not in the correct format/type we may not be able to use it the way we would like to.

There are five data types you need to be aware of:

- Integer – This is used to store whole numbers. An example of some data stored using the integer data type would be 77. You might use the integer data type to store someone's age or house number.
- Real/Float – This is used to store numbers with a decimal place. An example of some data stored using the real/float data type would be 4.35. You might use the real/float data type to store a monetary value (as you have pounds and pence e.g. £2.45).
- Boolean – This is used to store TRUE/FALSE values. You might use the Boolean data type to store a value for the question "Is the individual 18+" in a voting program. The answer is either TRUE (yes, the individual is 18 or older) or FALSE (no, the individual is not 18 or older).
- Character – This is used to store a single character. An example of some data stored using the character data type would be "A". You might use the character data type to store the input for a menu within a program (e.g. select option A, B, C, or D).
- String – This is used to store alphanumeric characters. An example of some data stored using the string data type would be "Programming". You might use the string data type to store the name of an individual, or for someone's date of birth (as it often includes / to separate values).

"Be aware when using the integer value it will remove any 0's at the start of a number, just like we would. For example, if we write the number 4 we write it just like that, we don't write 04 as that extra 0 is pointless. So, if you were storing someone's phone number, this obviously starts with a 0. Therefore, you couldn't store it as an integer as the program would remove the first 0 meaning the number would be incorrect! Same with a person's bank details, there card number could start with a 0 and you cannot risk that being removed. You would therefore have to store these details as strings, to ensure the 0's were not removed"

Casting

Casting is simply the process of changing a variables data type within a program. So, for example, you might get the user to input their data as an integer to begin with. However, later on down in the program you may require this data to become a string (so you can manipulate it for example such as splitting it). You would therefore 'cast' the variable to become a string.

Below is a program that takes the users age as an integer, and then casts it into a string. Just so you can see the data types changing, we are adding 5 to the integer age, and concatenating 5 to the string age:

```
age_integer = int(input("Please enter your age - "))
print(age_integer + 5)

age_string = str(age_integer)
print(age_string + "5")
```

Here we are requesting our user input their age as an integer. We are then outputting their age add 5 to it. As you can see we have also written the 5 as an integer as it is not inside speech marks. This should therefore undertake a calculation.

Here we are now casting our users age (stored inside age_integer) into a string, and storing the new value inside the variable age_string. We are then outputting their age concatenated with the string "5". Again, you can tell this is a string as speech marks are used. This should therefore just take our users age and stick a 5 onto the end of it.

Below is what the program looks like once it is run:

```
>>>
Please enter your age - 20
25
205
```

As you can see, the user inputted their age as 20. Firstly, we have used the integer version of their age to add 5 to it, producing the integer 25. Secondly, we have used the string version of their age (once casted) to produce the string "205".

Common Arithmetic + Boolean Operators

An operator is simply a function we are performing on some data.

At the beginning of this topic we discussed the common arithmetic operators and how to use them within a program. However, we used modulus (%) and quotient (//) without an explanation. This is what they are used to do:

Modulus (MOD or %) – Used to find the remainder of a division e.g. 5MOD2 = 1 (5 divided by 2 is 2 with a remainder of 1. We take the remainder and ignore the other part).

Quotient (DIV or //) – Used to find the whole number of a division e.g. 5DIV2 = 2 (5 divided by 2 is 2.5. We take the whole number part and ignore anything after the decimal point).

Boolean operators are used to allow for comparisons to be performed on data, producing more specific results. We will see these more in a later topic covering logic gates. However, here is a brief definition for each of the ones you need to be aware of:

AND – Used to ensure both sides of a comparison/condition are met e.g. x > 5 and y < 10 (x must be greater than 5 AND y must be less than 10. If both those conditions are not met then the whole statement is FALSE).

OR – Used to allow for either one or both sides of a comparison/condition to be met e.g. x > 5 or y < 10 (x must be greater than 5 OR y must be less than 10. One or both of those conditions must be met for the outcome to be TRUE).

NOT – Used to produce the opposite of the condition e.g. not x > 5 (if x is greater than 5 the output will be FALSE, even though x is greater than 5 and would therefore usually be TRUE).

Random Number Generation

If creating a program where we want a random choice or decision to be made, we use the function random, which is built into python.

To start using the random function, we must first import it from python's library. To do this we simply write the below code:

```
import random
```

We then can use a variety of commands to produce numerous different random results. For example, we might make a random choice from a list/array, we might output a random float. In this case, we are going to output a random integer (whole number). To do this, we add the below code:

```
import random

print(random.randint(0,10))
```

Firstly, we have our print command, which will output the random value once calculated.

We then have our command random.randint. The first part (random) is calling the use of the random function. The second part (randint) is requesting a random integer. However, we now need to set our range!

This can be seen inside the brackets after randint. We have set our range from 0 to 10. This mean the program will output a random, whole number, anything from 0 to 10. Let's see what it outputs after three runs:

```
>>>
7
>>> ================================ RESTART ================================
>>>
4
>>> ================================ RESTART ================================
>>>
5
>>> |
```

As you can se above, three random numbers generated!

Let's have a look at how we would make a random choice from a list. Below is a list of fruit:

```
import random
fruits = ["apple", "banana", "orange"]
```

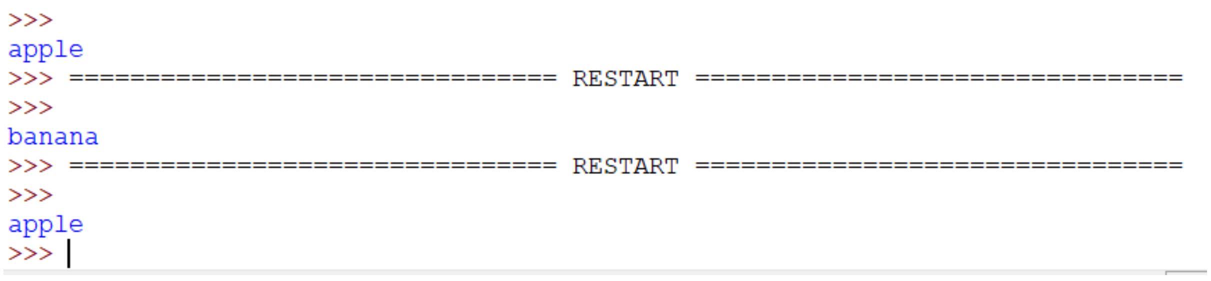
Again, we still have our import random code, so we can use the random function (this never changes no matter what random command you want to utilise).

To get our program to make a random choice from a given list we use random.choice. Let's have a look at how this code looks:

```
import random
fruits = ["apple", "banana", "orange"]

print(random.choice(fruits))
```

As seen in the last example, the first part (random) is calling the use of the random function. The second part (choice(fruits)) in this case as now requesting the program to make a choice from the list given, in this case fruits.

Let's see what it outputs after three runs:



```
>>>
apple
>>> ================================ RESTART ================================
>>>
banana
>>> ================================ RESTART ================================
>>>
apple
>>> |
```

There are numerous other commands that can be used with the random function, such as:

- random.randrange(x, y) which will select a random number within a given range
- random.shuffle(list) which will shuffle the contents of a list
- random.random() which will return a random float number between 0 and 1

Past Exam Questions

Answer the questions below, to help you revise what has been covered in 2.2 Programming Fundamentals.

1. Describe the difference between a variable and a constant. [2]

2. Describe what is meant by selection. [2]

3. Look at the table below. Tick **one** box in each row to show whether each statement is TRUE or FALSE. [5]

Statement	TRUE	FALSE
A while loop is used for count-controlled iteration		
An integer stores numbers with a decimal place		
A string would be used to store a telephone number		
A condition can have multiple if statements, but only one elif statement		
Sequencing means every instruction will execute, one after the other		

4. Mina decides to create a sub-routine.

   i. Describe one benefit to Mina using a sub-routine in a program. [2]

   ii. Mina decides to create a function.

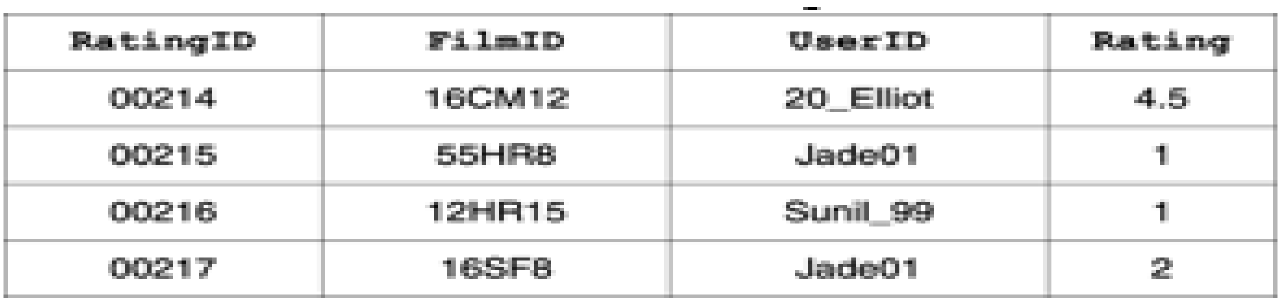
   Describe the difference between a function and a procedure. [2]

5. Mina realises data in her program requires casting.

   i. Describe what is meant by casting. [2]

6. Describe the difference between a one-dimensional array and a two-dimensional array. [2]

7. Look at an extract from a table called RATING below:



RatingID	FilmID	UserID	Rating
00214	16CM12	20_Elliot	4.5
00215	55HR8	Jade01	1
00216	12HR15	Sunil_99	1
00217	16SF8	Jade01	2

i. Write the SQL statement below that will extract all records from the table where the rating is 1. [3]

ii. Write the SQL statement below that will extract the RatingID from the table where the UserID is Jade01. [3]

2.3 Producing Robust Programs

In this section you will revise the following:

2.3.1 Defensive Design

- Defensive design considerations:
    - Anticipating misuse
    - Authentication
- Input validation
- Maintainability:
    - Use of sub-programs
    - Naming conventions
    - Indentation
    - Commenting

2.3.2 Testing

- The Purpose of testing
- Types of testing:
    - Iterative
    - Final/Terminal
- Identify syntax and logic errors
- Selecting and using suitable test data:
    - Normal
    - Boundary
    - Invalid
    - Erroneous
- Refining algorithms

Technical Terms

Technical Term	Definition
Input Sanitisation	The process of 'cleaning up' a user's input so it is in a suitable format ready for processing
Validation	The process of checking the user has inputted data in the correct format e.g. for a date it could be DD/MM/YYYY
Contingency Planning	The process of having a 'back up plan' in case the original plan fails
Misuse	When a user does not use the program the way it was designed to be used
Authentication	The process of ensuring a user has the rights/permissions to access a program e.g. by getting them to enter a username and/or password
Maintainability	This refers to ensuring the program can be easily updated, improved, and fixed
Comments	Used within a program. They help briefly identify what a section of code does
Naming Conventions	A 'rule of thumb' programmers will use when writing in a specific language. They follow certain rules when naming identifiers such as use of lowercase/uppercase characters, underscores etc.
Testing	The process of assessing whether a program works correctly and the way it should work. We aim to find 'bugs' and rectify them so the program is faultless
Iterative Testing	A type of testing. Iterative testing is carried out throughout the development of the program. Multiple 'mini tests' are carried out on sections of code
Final/Terminal Testing	A type of testing. Final/terminal testing takes place once the program has been completed. It tests the whole program at the end of its development
Syntax Error	An error within the code. This means the code has been incorrectly written, and will stop the program from working
Logic Error	An error that does not stop the program from working, but stops the program from working the way it should do
Test Data	The data that is selected to be used when testing a program
Normal Data	A type of test data. This data is correct and should be accepted by the program

Invalid Data	A type of test data. This data is of the correct data type but outside of the requirements set
Boundary Data	A type of test data. This is data that is on the edge of a range specified by the program and therefore should be accepted
Erroneous Data	A type of test data. This is data that is of the incorrect data type and therefore should be rejected

Defensive Design Considerations

When producing a computer program, it is important that we incorporate as many different strategies as possible to ensure the program does not fail. This means the program is robust, and can cope with numerous different types of data entry or misuse without failing.

There are numerous different defensive design considerations a programmer can consider:

- Input Sanitisation/Validation
- Contingency Planning
- Anticipating Misuse
- Authentication

Input Sanitisation

It seems strange to see the word 'sanitisation' come up in a computer science revision guide doesn't it! However, the meaning of the word in the sense you're thinking about is the same in this situation too!

Input sanitisation refers to the process of 'cleaning up' our users input to ensure it is in a suitable format ready for processing. For example, you might ask the user to input their name. The user may put a number of spaces at the front of their name that you obviously do not want to store. You would have designed your program so that it could cope with this, and would remove the white space so that only the necessary data was stored.

You can also edit the users input into specific case. In Python we would use .lower and .upper when gathering input to do this. For example, if you wanted to get the user to input their username, often usernames are not case specific. Therefore, you may convert the users input all into lowercase so that when you came around to using a condition to check if they entered the correct username, their input was in the correct format. Remember, when using a condition it has to be an exact match, so sanitisation comes in handy here!

Validation checks to see if what the user has inputted is in the correct format. It DOES NOT check to see if the data inputted is the correct data. For example, validation would check to see if a user has inputted their date of birth in a DD/MM/YYYY format, but would not be able to check if what the user has entered is actually their date of birth or a made up date of birth.

Contingency Planning

Contingency planning effectively means having a backup plan in case things don't go the way they should. When developing a computer program, it is important you have a contingency (or backup) plan, to ensure the program does not crash when experiencing something that is unexpected.

Thinking away from programs, printers for example have contingency plans. If you send something to print and the printer runs out of ink half way through the job, the job isn't lost. The printer doesn't simply just forget where it was up to and delete your job meaning you have to send it to print again. The printer has a contingency plan which is to store your job and where it got up to until the ink is replaced. Once this is done the job is simply resumed, saving you the job of having to resend the document to print again.

In terms of a computer program, imagine you were creating a quiz program. You might ask your user a question and give them four possible answers to pick from. A contingency plan would be what happens if they don't pick from those four options? Your program would be able to cope with this. It may throw up an error message alerting the user to the fact they did not select one of your options, or it may get them to re-enter their answer but this time select one of your options.

Anticipating Misuse

This is one area most computer programmers wish they did not have to consider, but unfortunately there will always be people in the world who want to exploit weak programs in order to access confidential information. It is important that, when designing a program, you think about how a user may misuse the program. This means you will be able to fix those weaknesses before someone more dangerous exploits them.

For example, if using an entry form linked to a database, you would need to consider the risk of SQL Injection (see paper 1 revision guide for a definition on this). As a programmer you would need to come up with a way of rejecting SQL statements that aim to extract data from a database via this method, therefore anticipating this potential misuse of the form.

Authentication:

Authentication is the process of ensuring a user has the right permission to access a computer program. There may be different levels of a program different users can access due to different permissions. It is important your program can ensure the right user accesses the right level to ensure safety.

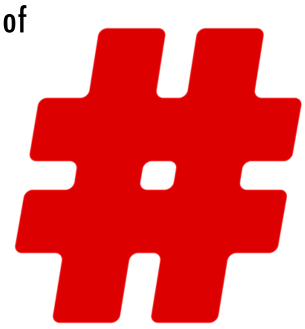
This could be achieved through a system where the user enters a username or password, and the program checks to see if the user is both a permitted user, as well as what they are permitted to access. If incorrect details are entered, the program should reject the user.

Maintainability

It is important to ensure we produce a program as we go that can be easily maintained in the future. Programs can be thousands of lines of code, and any programmer of the same language should be able to open up the code and be able to look through it to locate specific sections they need. This all comes down to the structure and layout of the code.

Comments

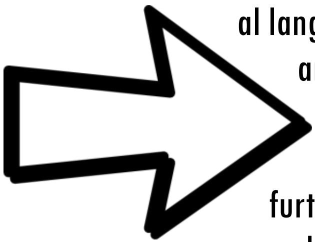
Comments are critical when creating computer programs, as they tell a user what is happening in different sections or lines of code. In Python we use a hashtag (#) to indicate a comment. Comments do not affect the code in anyway as they are bits of code that are not executed when run, they are simply there to help identify what different sections do. It's a bit like leaving lots of little post-it notes on your code saying "this bit is for this" and "that bit does that and links to that other bit".



Adding comments help maintain a program as it allows different programmers to investigate and understand the code. Imagine the code needed updating, but you had left the job and a new programmer had taken over what you had created. Without comments could mean the new programmer looking through lines and lines of code trying to find the bit they need. With comments, it allows the programmer to quickly identify what they are looking for and forget about the rest of the code, making the maintenance process much more efficient.

Indentation

In some languages, indentation is forced upon you, such as Python. Without correct indentation the program will not work and produce an error. However, it is not a necessity in al languages, and therefore is considered good practice to incorporate anyway.



Indentation refers to when we 'push in' a bit of code so that it is further away from the edge. It allows us to create a bit of a layered approach to our code, showing that bits of code that line up with each other link together. It also allows us to see what code follows on from other pieces of code, like in a condition, where if a condition is met we can see what code will execute next.

Sub-Programs

Sub-programs are a crucial way for programmers to write code in an efficient and organised way. Remember from 2.2 Programming Techniques, a sub-program is a block of code that is given an identifier (name) and then called upon whenever it is needed. Using sub-programs helps maintain a program because it makes the program easier to read and understand, as it is all broken up into neat sections. Furthermore, if an element of the program stops correctly working the programmer only needs to review and modify the code in the given sub-program that is not working, rather than having to review the whole program.

Naming Conventions

When writing code, it is often good to follow general naming conventions for the language you are writing in. A naming convention is a general rule of thumb programmers of the language follow when creating identifiers, it helps them understand the code at a quick glance. Some of these naming conventions are below:

- Separate words in identifiers using an underscore
- Write constants that will be remain constant throughout the program in capital letters
- When creating classes, start each word with a capital letter
- When creating functions use lowercase letters and separate with an underscore

The purpose of testing and the different types of testing

Testing a solution is just as important as any other stage in the creation of the solution, if not more important! As a programmer you should not submit any solution as being a final version before it has been thoroughly tested and inspected for any underlying errors or bugs.

Testing is the process of ensuring a solution works correctly and the way it should work. Testing therefore is not just about making sure the solution does not crash when run, but also that it works the way it is expected to work.

Testing allows us to find faults in our program which can then be rectified before we hand over the final solution to our client. It is likely there may still be minor bugs and glitches, as there usually is with large scale programs, however the program should generally work well and accurately and the bugs left should only be minor.

There are two different types of testing, iterative testing and terminal/final testing.

Iterative Testing

Iterative testing involves testing our solution throughout its development. In other words, every time we complete a small section of code, we would test it. This would happen over and over again throughout its development.

An advantage to iterative testing is that it allows us to test small sections of code one bit at a time. This means we are not overloaded with potential errors as the code we are testing should only be a single section, not the whole program. This means it makes it easier to spot errors and fix them, compared to looking at the whole program to locate errors. However, iterative testing can take up valuable time in the development of a solution, especially if you are creating multiple sections of code. Stopping to test repeatedly takes time, which could extend the amount of time taken to complete the project.

Terminal/Final Testing

Terminal (or final) testing involves testing the whole solution once the development is complete. In other words, you test the whole program once it is completed. This means one big, final test at the end.

An advantage of terminal testing is that it saves time during the development phase of the solution as you are not stopping repeatedly to test the solution. You also test the whole solution and how it interacts with its different sections, rather than testing each individual section on its own. However, terminal testing can make it difficult to spot errors. Imagine you have written 1000's of lines of code without any iterative testing, and then run a terminal test at the end once you have finished coding. You could end up with 100's of errors! Locating all those errors and rectifying them can be extremely difficult, especially if you are using an IDE that doesn't tell you where the errors are!

So, in summary, it is best to use both types of testing! Use iterative testing as you go through the process of developing your solution, and then use terminal testing once the solution is complete to test the whole code.

"Remember, good practice is to test your program as you go. This means you reduce the amount of errors you are faced with in one go. No matter how good the programmer is, they'll always be errors in the code. It is important to rigorously test your code so that once you hand the solution over to the client it is in the best condition it can be!"

It's your turn!

Describe the difference between terminal testing and iterative testing. **[2]**

____

____

____

____

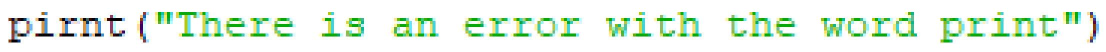
Syntax and Logic Errors

When finding an error within your code, it is important you are able to know what type of error it is. In the exam they also like to show you a piece of code and ask you to identify the syntax and logic errors, so it is important you can spot them yourself too.

Syntax Errors

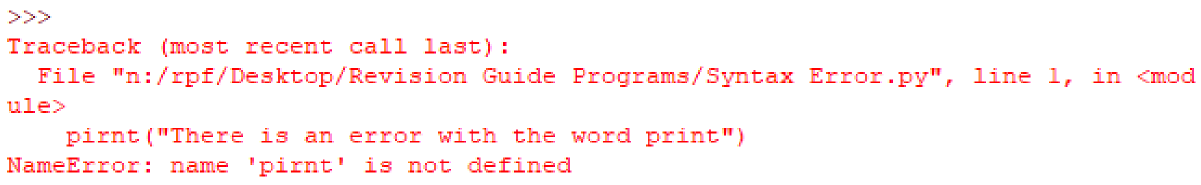
A syntax error is an error within the code. This type of error will cause the program to fail or crash. A syntax error refers to what the programmer has actually written within the syntax. For example, they may have spelt a certain word wrong that is built into the language (e.g. print in Python). The code therefore cannot be executed.

Below is an example of a syntax error in Python. In this example we have written the word 'print' wrong. This means when the code is executed Python is unable to understand what is meant by the word 'pirnt' and therefore alerts us to a syntax error:



```
pirnt("There is an error with the word print")
```

As you can see in the above code, the word 'print' is incorrectly spelt. Below is what is displayed in the shell when the program is run:



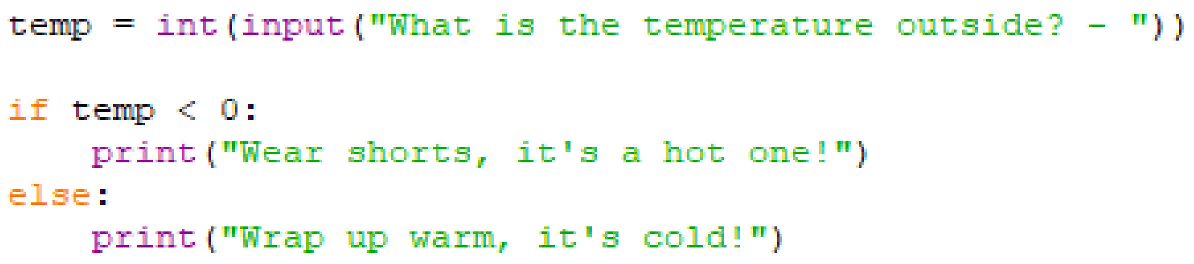
```
>>> 
Traceback (most recent call last):
  File "n:/rpf/Desktop/Revision Guide Programs/Syntax Error.py", line 1, in <mod
ule>
    pirnt("There is an error with the word print")
NameError: name 'pirnt' is not defined
```

In the above message Python is alerting us to a syntax error in the code and is telling us the word 'pirnt' is not defined in Python and therefore it does not know what to do when executing this line of code.

<u>Logic Errors</u>

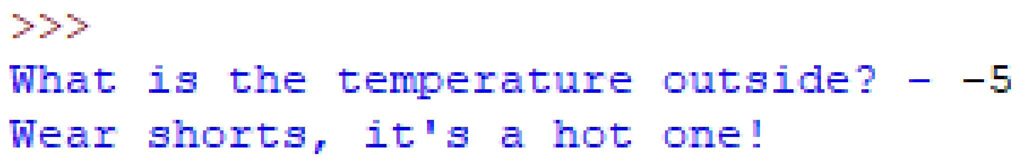
Logic errors refer to when the program does not work the way it should. Logic errors will not stop the program from running (unlike syntax errors) however the outcome of the program will not be what we expected.

Below is an example of a logic error in Python. As you can see the program has not failed when being run, however the output is not what we would have expected. The program asked the user to input the temperature, and they inputted '-5'. You would expect the program to tell the user it is cold and therefore to wrap up warm, but instead it tells them to wear shorts! The program clearly has not worked the way it should, and has produced a logic error:



```
temp = int(input("What is the temperature outside? - "))

if temp < 0:
    print("Wear shorts, it's a hot one!")
else:
    print("Wrap up warm, it's cold!")
```

Below is what is displayed once the program is run, and once the user has inputted the number '-5':



```
>>>
What is the temperature outside? - -5
Wear shorts, it's a hot one!
```

As you can see, the program has clearly not worked the one is should have, the output is not logical. In this case, we have got the sign and output mixed up and have said if the temperature is less than 0 then wear shorts. The correct code should be if the temperature is less than 0 then wrap up warm!

The above program could be improved as if the user enters 1 then it will tell them to wear shorts. We all know not to wear shorts when it is 1 degree outside! However, you can see the logic error in this program regardless of the rest of the code.

Suitable test data

When testing your program, it is important that you use suitable test data. The test data is the data that you will use when running the test. For example, if you are requested to input your age, what will you input? Will you input data that should be accepted by the program? Will you input data that should be rejected? Will you input absolutely nothing?

It is important you use different types of test data too when testing a solution so you can test different 'angles' of the code.

There are six different types of test data you need to be aware of:

- Normal data – This is data that should be accepted by the program without causing any errors
- Invalid data – This is data that is the correct data type, however does not meet the requirements set and therefore should be rejected by the program (e.g. if you inputted your date of birth as a date in the future)
- Erroneous data – This is data that should be rejected by the program as it is not of the correct data type
- Boundary data – This is data that falls on the very edge of the specified range of the program (e.g. if you request the user to input a number between 1 and 100 and they input 100)

When using test data you may find you use a piece of data that falls across more than one of these areas (e.g. you might input some data that is both valid and in range).

It's your turn!

James and John are testing there program which calculates a users BMI. The max height the program can work with is 2 metres. When testing the program John enters 2 as the height. Describe what type of test data has been used. [2]

____

____

____

Past Exam Questions

Answer the questions below, to help you revise what has been covered in 2.3 Producing Robust Programs.

1. Most computer programs are built to be robust. One way a computer program can be built to be robust is by contingency planning.

   i. Explain what is meant by authentication. [2]

   ii. Identify and describe one other way a programmer can ensure their program is robust. [2]

2. Explain what comments are and how they help maintain a program. [3]

3. Alex is told he must test his computer program before releasing it to his customers.

   i. One method of testing is iterative testing. Identify another type of testing. [1]

   ____

   ii. Describe the difference between iterative testing and the testing method you have mentioned in answer 3(i). [2]

   ____

   ____

   ____

   ____

   iii. Explain **one** advantage and **one** disadvantage to using iterative testing. [4]

   ____

   ____

   ____

   ____

   ____

4. Describe what is meant by a logic error. [2]

____

____

____

5. Look at the program below:

```
1  username = inputt(“What is your username? – “)
2  password = input(“What is your password? – “)
3
4  if username != “Jones589” and password != “Password12” then
5      print(“Incorrect credentials. Failed to log in”)
6  else
7      print(“Correct credentials. Log in successful”)
```

i. Identify the below errors in the program. For each error you must state what line the error is on, what part of the code is incorrect, and then the correct code: [6]

Syntax Error – Line ______________________

Incorrect Code - ______________________

Correct Code - ______________________

Logic Error – Line ______________________

Incorrect Code - ______________________

Correct Code - ______________________

6. Identify and describe **two** types of test data. [4]

2.4 Boolean Logic

In this section you will revise the following:

2.4.1 Boolean Logic

- Simple logic diagrams using the operations AND, OR and NOT
- Truth tables
- Combining Boolean operators using AND, OR, and NOT
- Applying logical operators in appropriate truth tables to solve problems

Technical Terms

Technical Term	Definition
Binary	This is the language of the computer. Binary is used to represent all data within a computer system. It refers to transistors opening and closing allowing electricity to pass through or not, which is then represented using 1's (ON) and 0's (OFF)
Logic Diagram/Gate	A way we can represent Boolean operations. The three Boolean operations you need to be aware of are AND, OR, and NOT
AND	A type of logic gate. In this gate both inputs must be TRUE (ON or 1) for the output to be TRUE (ON or 1)
OR	A type of logic gate. In this gate one or both inputs must be TRUE (ON or 1) for the output to be TRUE (ON or 1)
NOT	A type of logic gate. In this gate the output is the opposite of the input.
Truth Table	A way we can represent a logic diagram in table form. It shows all the possible inputs and outputs for a logic diagram.
/\	Used to represent the AND operator in a truth table or logic statement.
\/	Used to represent the OR operator in a truth table or logic statement.
¬	Used to represent the NOT operator in a truth table or logic statement.

Why computer systems use binary

As you will know, computer systems do not speak human! However, they still need to be able to process what instructions they are given, and be able to communicate with their different components.

Therefore, all computer systems use binary. Binary is a number system that is called base 2. It used two numbers, 1 and 0, and all data is stored in these binary digits. That is regardless of whether it is a letter, a pixel on a picture, or a sample from a sound. Everything is stored in binary!

Data is converted to binary via transistors. As computer systems use electrical currents (and there are no values to these) there needs to be a way of converting these currents into a form where the computer can understand what it is processing. As electricity passes through these transistors, they are either open (ON, TRUE, or 1) or closed (OFF, FALSE, or 0). The process of this electricity being passed through allows us to convert these electrical currents into 1's and 0's.

We will learn more about binary in 2.6 Data Representation, including how data is converted into binary.

Simple logic diagrams

In order for data to be processed more accurately with a greater array of options, computer systems use logic gates/diagrams. These effectively take some input, process it using its 'rule' and therefore output the required data.

There are three logic gates you need to be aware of and able to calculate the inputs and outputs for:

- AND gate
- OR gate
- NOT gate

As we go through explaining these gates, you can think of each input for the gate being a light switch, and the output being a light bulb. It will make sense!

We can also represent the input or output as the following (they all mean the same thing):

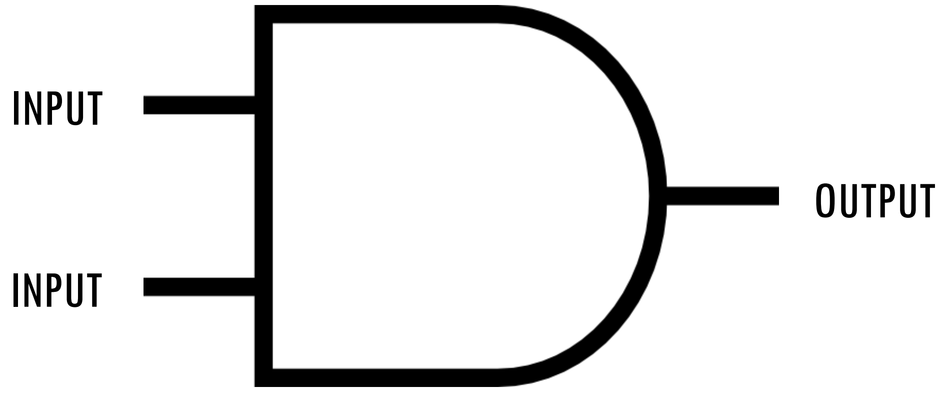
1 / ON / TRUE – The electrical current passes through the gate

0 / OFF / FALSE – The electrical current does not pass through the gate

AND Gate

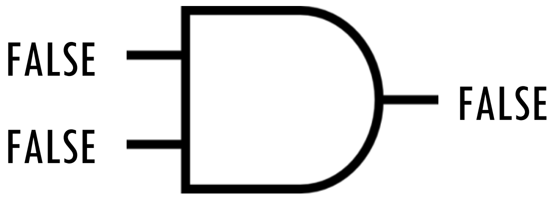
In this gate **both** inputs must be TRUE (ON or 1) for the output to be TRUE (ON or 1).

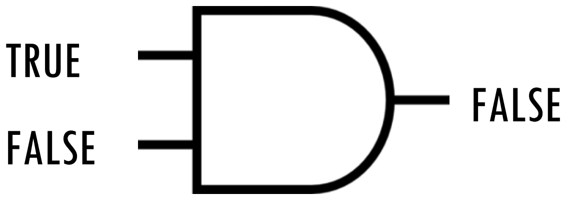
The AND gate looks like this:

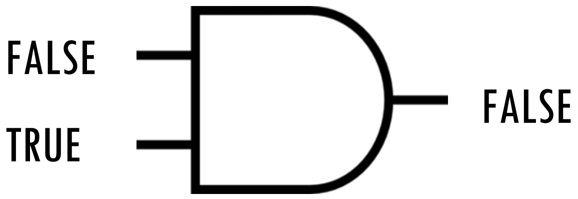


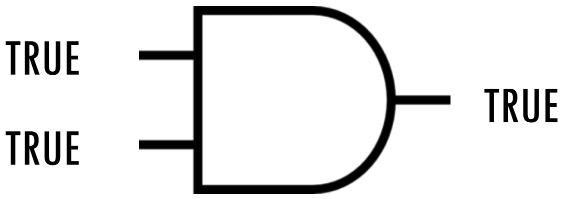
The best way to remember how the AND gate looks is that it looks like a letter D (and there is a letter D in the name AND).

As you can see, the AND gate has two inputs and one output. Now, remembering the rule that both inputs must be TRUE for the output to be TRUE, we can determine what the four possible outcomes for the AND gate are:





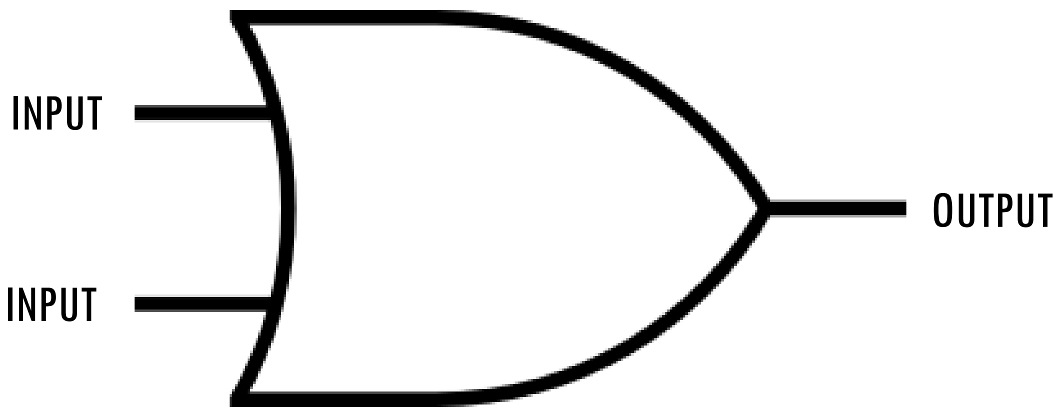




OR Gate

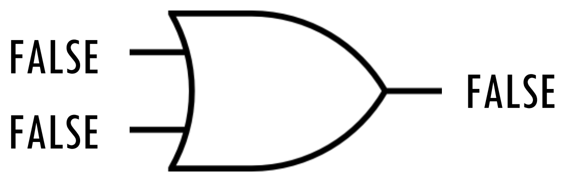
In this gate **one or both** inputs must be TRUE (ON or 1) for the output to be TRUE (ON or 1).

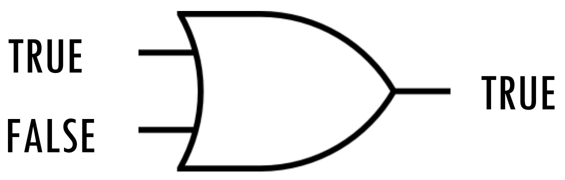
The OR gate looks like this:

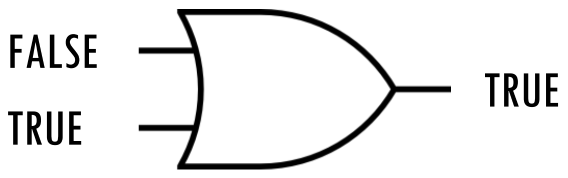


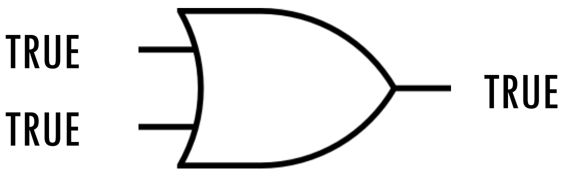
The best way to remember how the OR gate looks is if you look at the curved part, it looks like half an O, and there is an O in the word OR. It also looks a bit like a kite!

As you can see, the OR gate has two inputs and one output. Now, remembering the rule that one or both inputs must be TRUE for the output to be TRUE, we can determine what the four possible outcomes for the OR gate are:





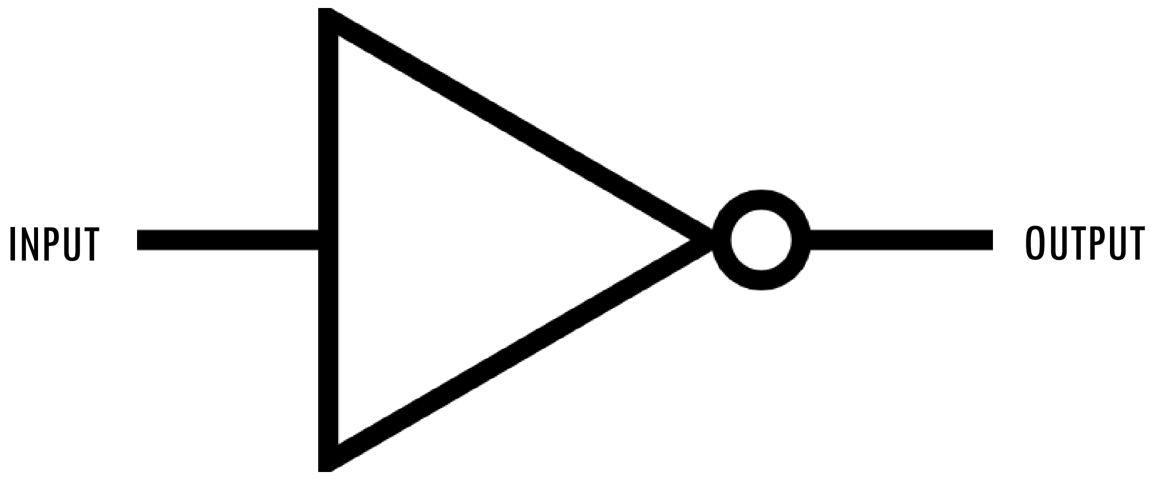




NOT Gate

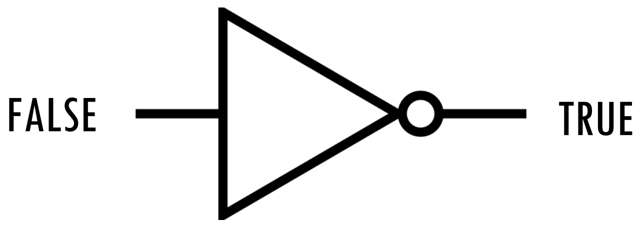
In this gate the output is the **opposite** of the input.

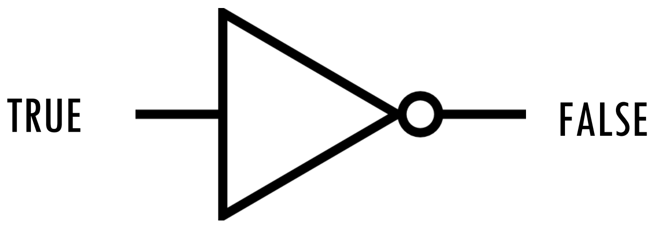
The NOT gate looks like this:



The best way to remember how the NOT gate looks is that it has the triangle with a small ball at the tip of it. It also looks quite a bit different to the AND and OR gates!

As you can see, the NOT gate has only one input, and one output. Now, remembering the rule that the output is the opposite of the input, we can determine what the two possible outcomes for the NOT gate are:

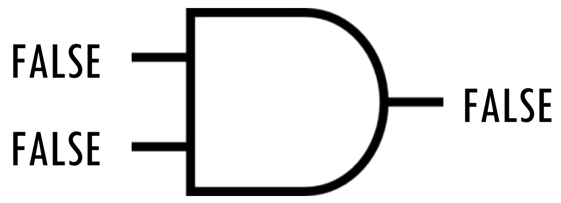


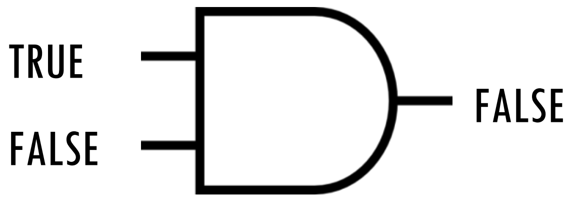


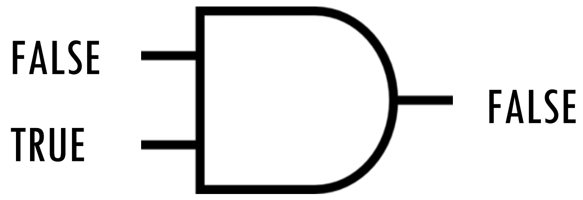
Truth Tables

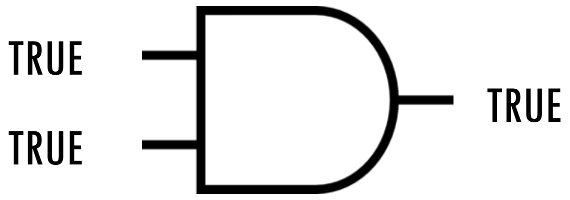
We can represent the logic gates above using truth tables as well. Effectively all a truth table is is the logic gate but without the diagram! We can use truth tables to represent logic circuits too, but we'll come on to this in the next section.

Let's take our AND logic gate first of all. Again, here are the four possible combinations:

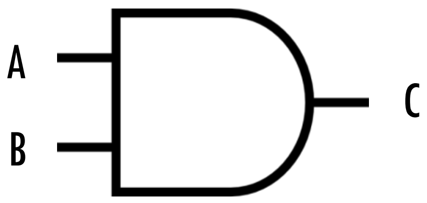








Now, to convert these into a truth table is simple! We will refer to the two inputs as A and B, and the output as C. This is what the logic gate looks like now we have A, B, and C:



Let's do the first combination of FALSE and FALSE = FALSE.

A	B	C
FALSE	FALSE	FALSE

What our table effectively says now is if input A is FALSE, and input B is FALSE, then the output (C) will be FALSE.

Let's do the next combination, FALSE and TRUE = FALSE.

A	B	C
FALSE	FALSE	FALSE
FALSE	TRUE	FALSE

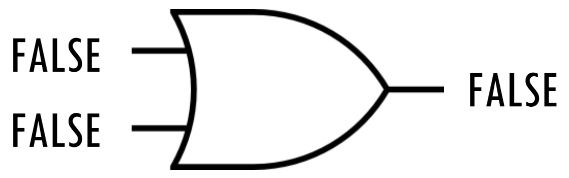
Again, what our table effectively says now is if input A is FALSE, and input B is TRUE, then the output (C) will be FALSE.

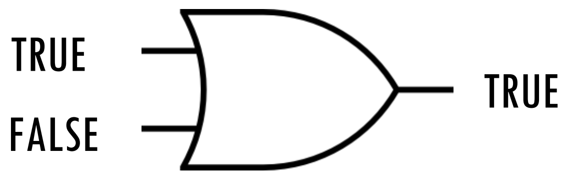
Here is the completed table:

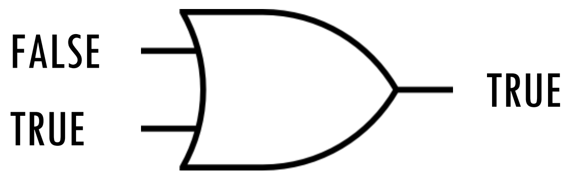
A	B	C
FALSE	FALSE	FALSE
FALSE	TRUE	FALSE
TRUE	FALSE	FALSE
TRUE	TRUE	TRUE

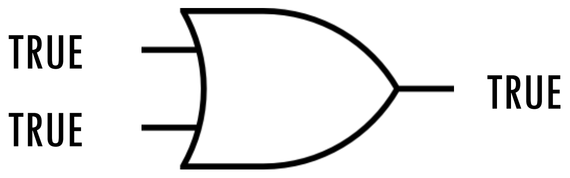
And that is a truth table! All we have done is written down all the possible combinations for A and B, and therefore calculated what the output (C) would be.

Let's look at the truth table for the OR gate now. Again, as a reminder, here are the four possible OR gate combinations:

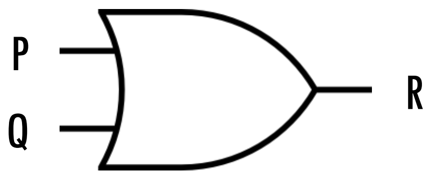








Again, let's now convert our inputs and output into letters. To mix it up, this time we'll use P and Q as our inputs, and R as our output:



Let's do the first combination of FALSE and FALSE = FALSE.

P	Q	R
FALSE	FALSE	FALSE

What our table effectively says now is if input P is FALSE, and input Q is FALSE, then the output (R) will be FALSE.

Let's do the next combination, FALSE and TRUE = FALSE.

P	Q	R
FALSE	FALSE	FALSE
FALSE	TRUE	TRUE

Again, what our table effectively says now is if input P is FALSE, and input Q is TRUE, then the output (R) will be TRUE.

Here is the completed table:

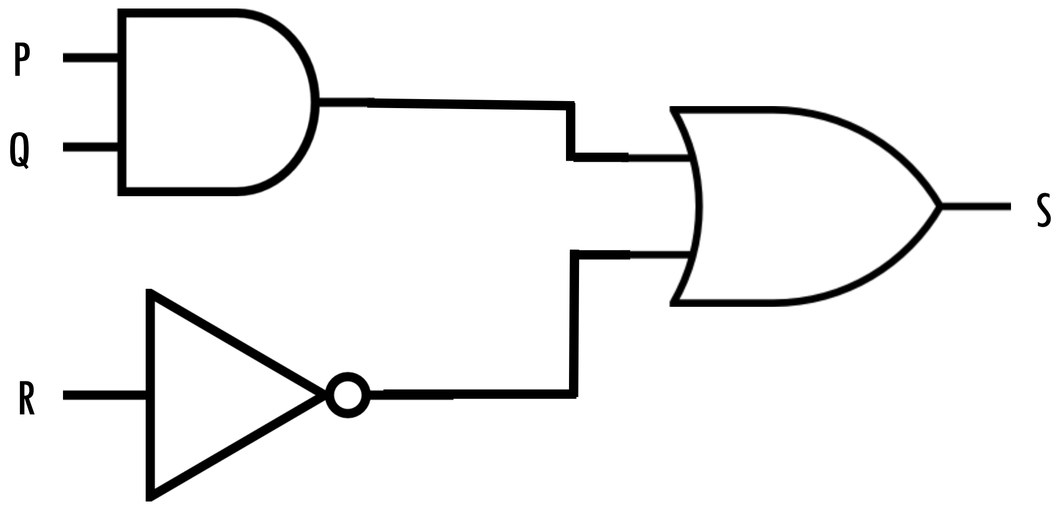
P	Q	R
FALSE	FALSE	FALSE
FALSE	TRUE	FALSE
TRUE	FALSE	TRUE
TRUE	TRUE	TRUE

And that is the truth table for the OR gate! We won't do the truth table for the NOT gate, as you can see how they work. These will get more complicated in the next section, so ensure you understand these basic truth tables before moving on!

Combining Boolean operators to two levels

Now you understand how the AND, OR, and NOT gates work to one level, we are going to begin to combine them to make them a bit more complicated. Alongside each two level logic circuit we will complete the truth table also.

All we mean when we talk about two levels is that there are more than one logic gate, and they will combine together over two levels. Let's look at an example:



Now, although this appears on the surface to look quite complicated, it really isn't! All we do is the exact same thing we did before when working with one logic gate in order to solve it and produce the truth table containing all the different outcomes.

Here is our empty truth table ready to go:

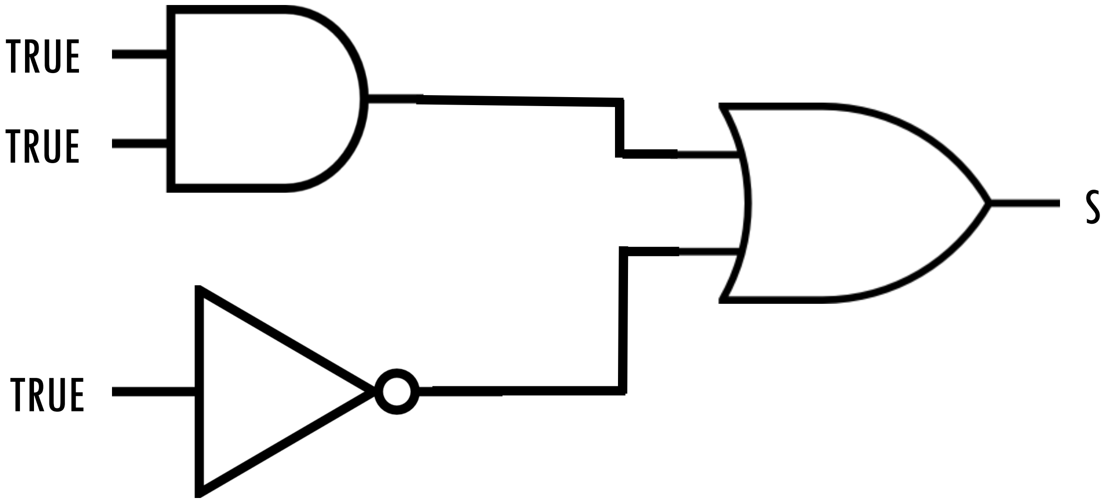
P	Q	R	S

We can tell there are going to be 8 different combinations in total as there are 4 possible combinations for the AND gate, and 2 possible combinations for the NOT gate. 4 multiplied by 2 gives us 8.

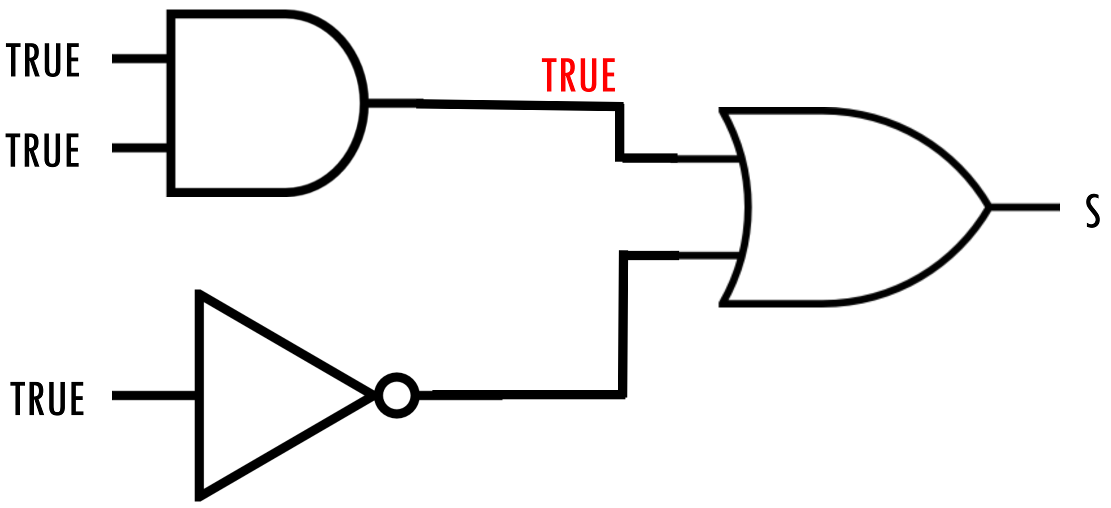
Let's do the first possible combination:

P	Q	R	S
TRUE	TRUE	TRUE	

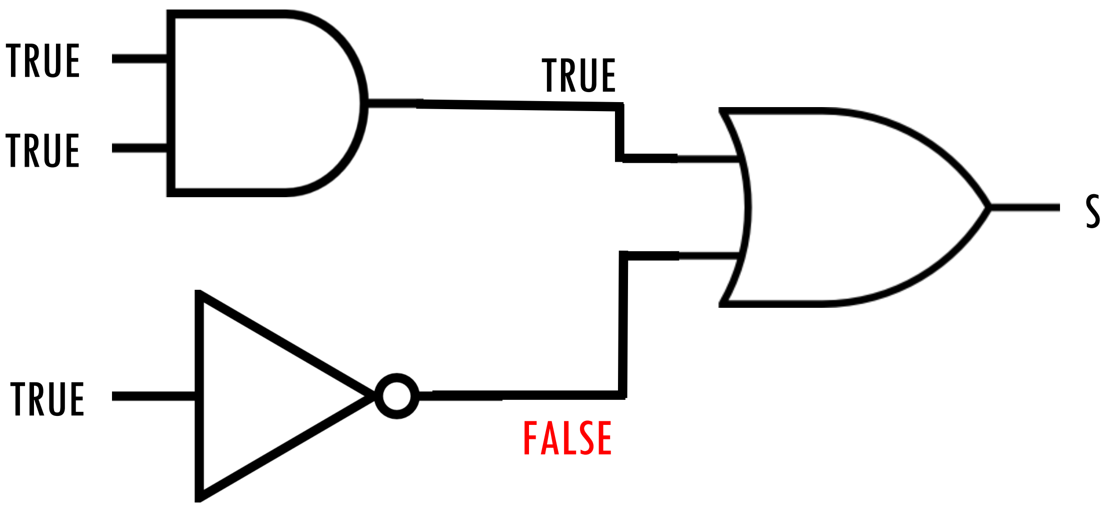
For this one let's draw the diagram again:



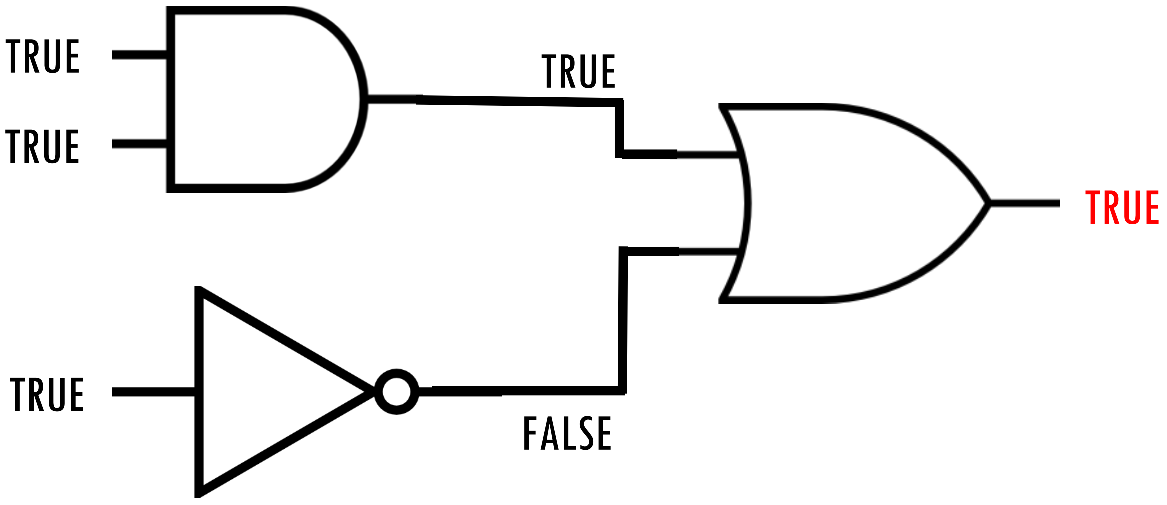
Let's focus on the AND gate first. If the two inputs are TRUE and TRUE (knowing the rule is both inputs must be TRUE for the output to be TRUE) we know the output must be TRUE. We will put the output to the end of the line which means it is going to become the input for the OR gate later on:



Now let's focus on the NOT gate. If the input is TRUE (knowing the rule is the output is the opposite of the input) then the output must be FALSE. Again, we will put the output to the end of the line which means it is going to become the second input for the OR gate:



We now have our two inputs for the OR gate (TRUE and FALSE). If the inputs are TRUE and FALSE (knowing the rule is one or both inputs must be TRUE for the output to be TRUE) then the output must be TRUE.



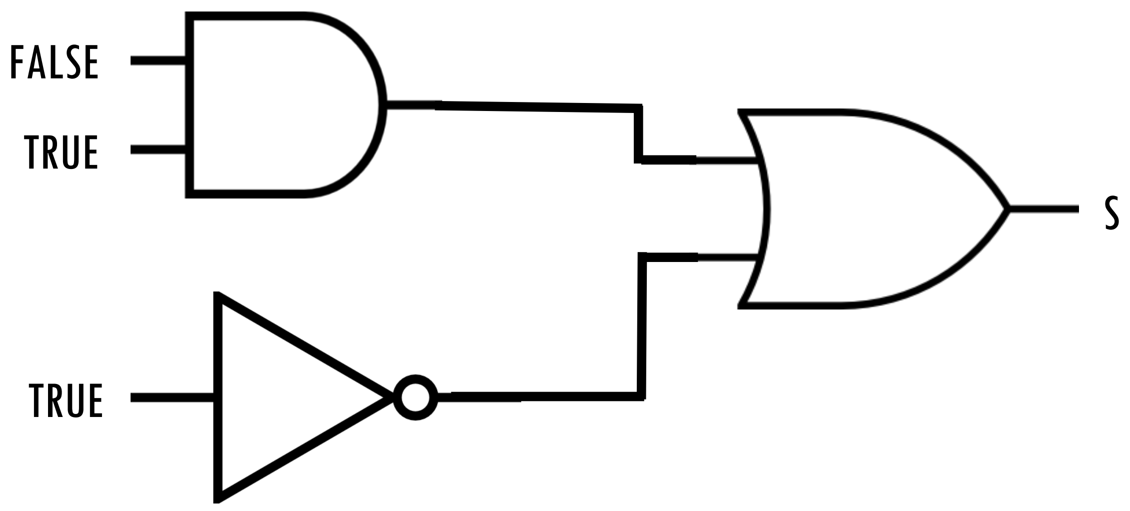
And there is the first row in our truth table completed! Here is the truth table so far:

P	Q	R	S
TRUE	TRUE	TRUE	TRUE

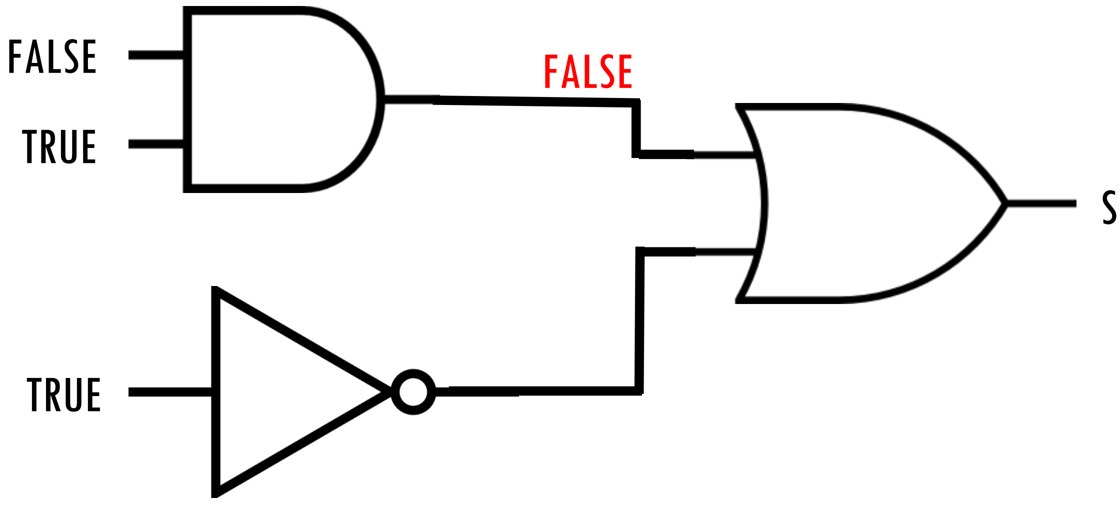
Again, let's do the next possible combination. This time let's do the below:

P	Q	R	S
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	TRUE	

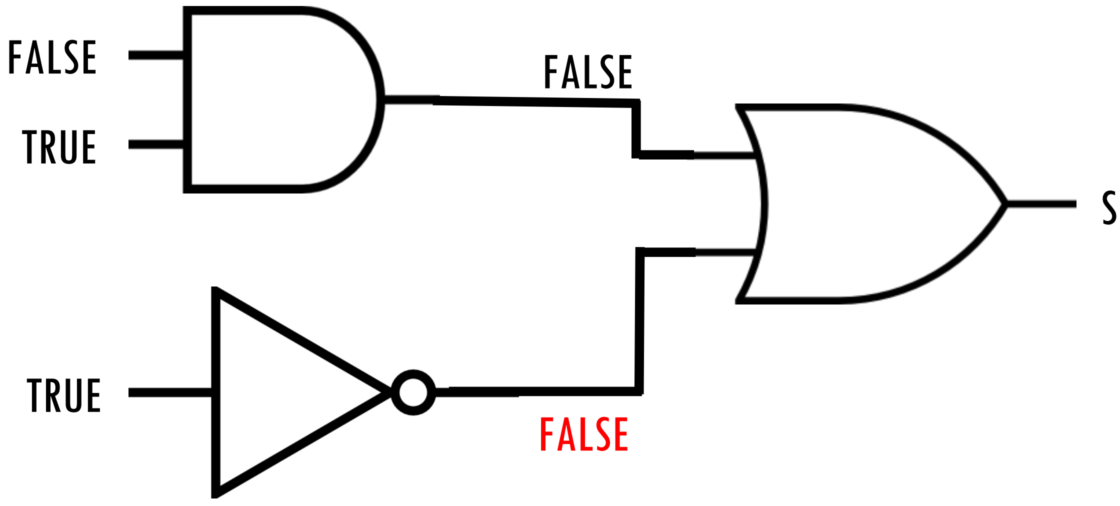
Again, let's draw out the diagram:



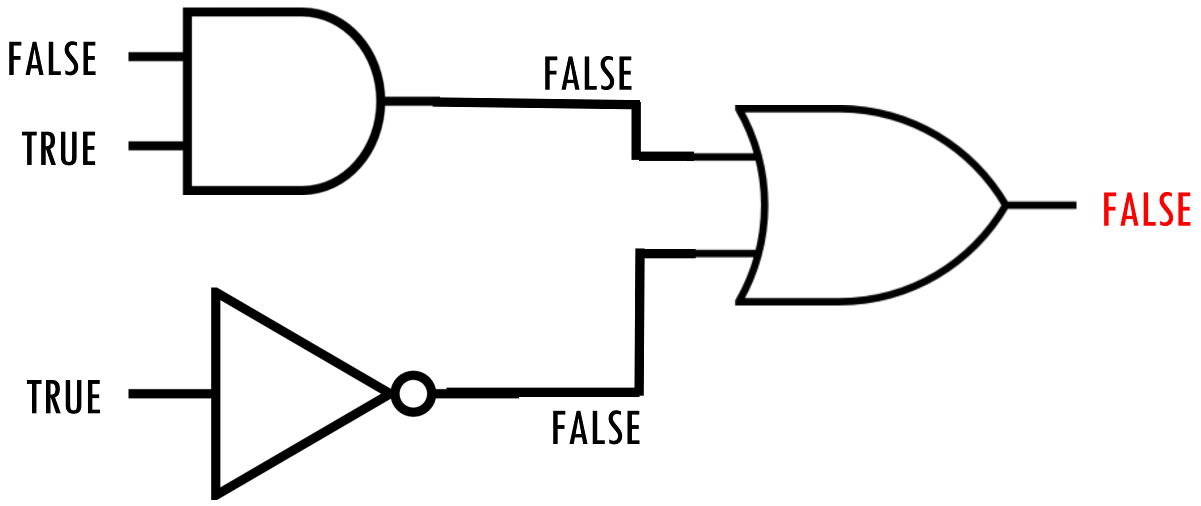
Let's focus on the AND gate first. If the two inputs are FALSE and TRUE (knowing the rule is both inputs must be TRUE for the output to be TRUE) we know the output must be FALSE. We will put the output to the end of the line which means it is going to become the input for the OR gate later on:



Now let's focus on the NOT gate. If the input is TRUE (knowing the rule is the output is the opposite of the input) then the output must be FALSE. Again, we will put the output to the end of the line which means it is going to become the second input for the OR gate:



We now have our two inputs for the OR gate (TRUE and FALSE). If the inputs are FALSE and FALSE (knowing the rule is one or both inputs must be TRUE for the output to be TRUE) then the output must be FALSE.



And there is the second row in our truth table completed! Here is the truth table so far:

P	Q	R	S
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	TRUE	FALSE

What we can then do is continue to work our way through the truth table, completing all the various different inputs and outputs.

Here is the complete truth table for the diagram shown:

P	Q	R	S
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	TRUE	FALSE
TRUE	FALSE	TRUE	FALSE
FALSE	FALSE	TRUE	FALSE
TRUE	TRUE	FALSE	TRUE
FALSE	TRUE	FALSE	TRUE
TRUE	FALSE	FALSE	TRUE
FALSE	FALSE	FALSE	TRUE

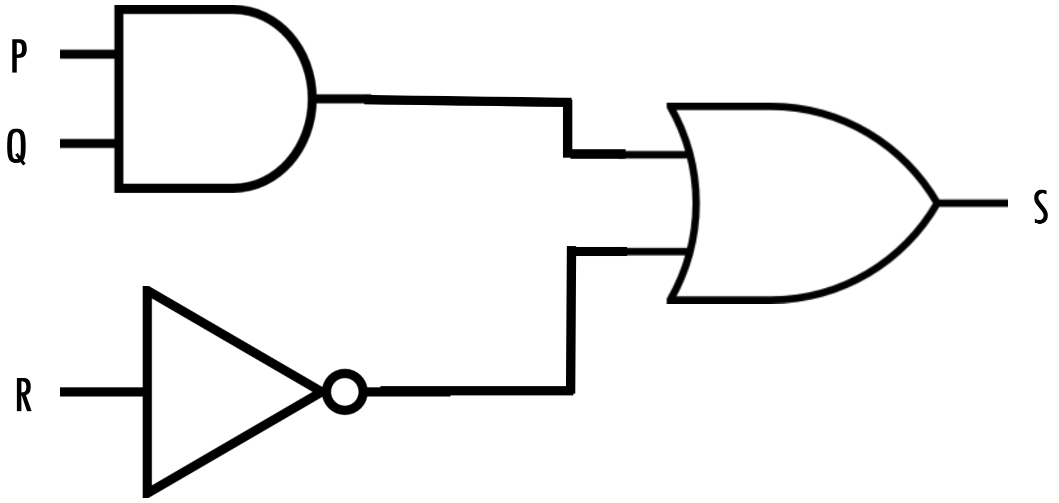
"When completing a truth table it is easy to forget what you have done or repeat a combination. The best thing to do is pick one of the gates to alternate the different combinations with. So, for example, in the table above we did all the combinations for if R was TRUE (mixing up P and Q) and then all the combinations if R was FALSE (again mixing up P and Q). This meant we didn't repeat anything accidentally!"

Expressing logic circuits and using /\, \/, and ¬

When answering questions in the exam about logic circuits and gates, it is possible you won't have the diagram drawn for you. You will either have an expression (statement), or symbols will be used.

Let's focus on expressions first.

Look at this diagram below, one we did previously:



We can write this logic circuit as an expression. Firstly, let's start with this:

S =

Now, what does S equal? How do we get to the answer for S? Let's do the AND gate first.

S = P AND Q

Now let's look at the NOT gate.

S = P AND Q NOT R

The above is incorrect as those two expressions need to be separate. Let's separate those using brackets.

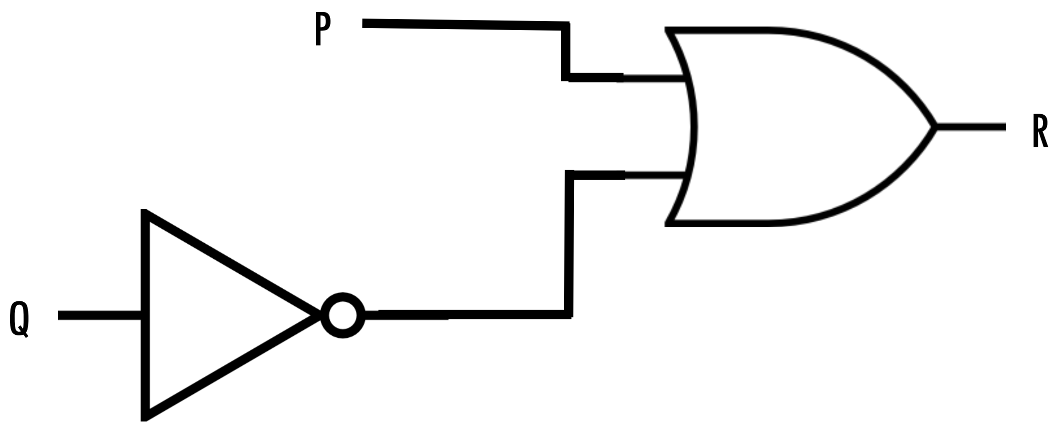
S = (P AND Q) (NOT R)

Finally, let's add in the OR gate.

S = (P AND Q) OR (NOT R)

What the above expression says now is that S is equal to the outcome of P AND Q, the outcome of NOT R, and then the outcome of those two outcomes in an OR gate.

Let's try another:



Again, we can write this logic circuit as an expression. Firstly, let's start with this:

R =

Now, what does R equal? How do we get to the answer for R? Let's do the NOT gate first.

R = NOT Q

Now let's move onto the OR gate. The inputs for the OR gate are P and whatever the output of NOT Q is.

R = (NOT Q) OR P

We can also express AND, OR, and NOT in the form of symbols:

- /\ = AND
- \/ = OR
- ¬ = NOT

So, for example, let’s take the below expression

S = (P AND Q) OR (NOT R)

This expression could also be written like this:

S = (P /\ Q) \/ (¬ R)

Here’s another example:

R = (NOT Q) OR P

This expression could also be written like this:

R = (¬ Q) \/ P

“It is simply a case of remembering these symbols. Try to remember it as the AND symbol (/\) looks a bit like a letter A, the NOT symbol (¬) looks like half a T, and the OR symbol (\/) is the left over one! It is common for examiners to use these in the exam, especially in truth tables. So, make sure you understand them!”

Past Exam Questions

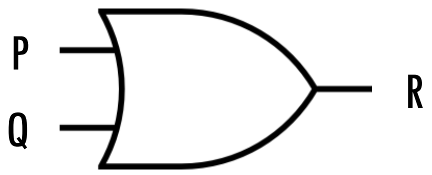
Answer the questions below, to help you revise what has been covered in 2.4 Boolean Logic.

1. Describe why computer systems use binary. [2]

2. Draw the logic gate for the below expression: [2]

   C = A NOT B

3. Complete the truth table for the below logic gate: [5]



P	Q	R
FALSE		FALSE
TRUE		TRUE
FALSE	TRUE	
		TRUE

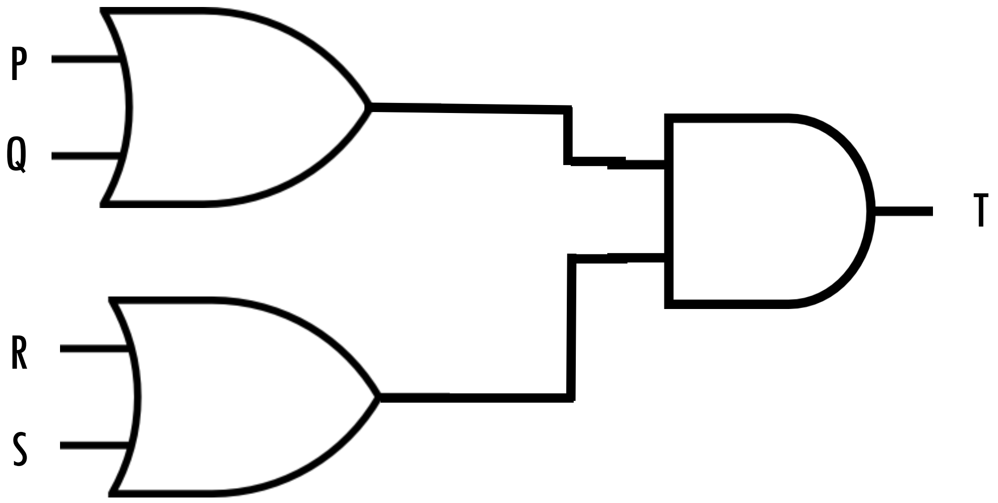
4. Draw the logic gate for the below expression: [2]

C = (NOT A) AND B

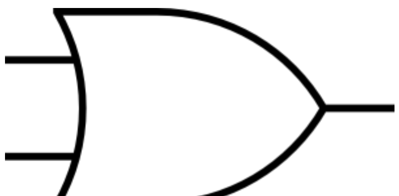
5. Complete the below truth table: [4]

A	B	(¬A) /\ B
TRUE	TRUE	
TRUE	FALSE	
FALSE	TRUE	
FALSE	FALSE	

6. Write the below logic circuit in the form of an expression. [3]



7. Complete the logic circuit. [1]

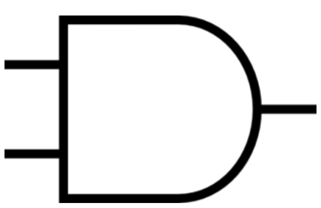


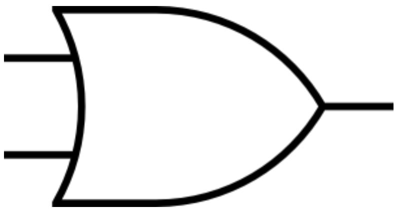
TRUE

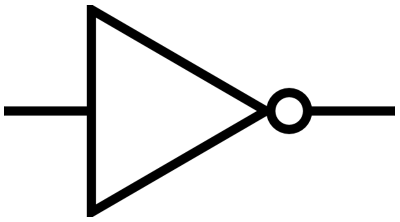
FALSE

____

8. Identify the names of each of the below logic gates. [3]







__________ __________ __________

9. Complete the truth table below. [8]

P	Q	R	S = (P /\ Q) /\ (¬R)

2.5 Programming Languages and Integrated Development Environments

In this section you will revise the following:

2.5.1 Languages

- Characteristics and purpose of different levels of programming language:
    - High-level languages
    - Low-level languages
- The purpose of translators
- The characteristics of a compiler, and an interpreter

2.5.2 The Integrated Development Environment (IDE)

- Common tools and facilities available in an Integrated Development Environment (IDE):
    - Editors
    - Error diagnostics
    - Run-time environment
    - Translators

Technical Terms

Technical Term	Definition
Low Level Programming Language	Written in 'assembly language', and more closely represents how the CPU works. Translated by an assembler when run into machine code.
High Level Programming Language	Written in a programming language such as Python. Translated by an interpreter or compiler when run into machine code.
Translator	Converts the programming language into machine code. This therefore allows the computer system to understand and process the code.
Compiler	Translates high level programming code into machine code. A compiler translates the whole program in one go.
Interpreter	Translates high level programming code into machine code. An interpreter translates the program line by line.
Integrated Development Environment (IDE)	Used to create programs and software. It provides us with a place to write the programming code and execute it.
Code Editor	A feature of an IDE. Provides a place where the user can write the programming code, often called the 'shell'.
Debugger/Error Diagnostics	A feature of an IDE. Used to detect errors in the code. It will identify where the error is, what line number the error is on, and what type of error it is.
Run-time Environment	A feature of an IDE. Allows the user to execute the program one step at a time so they can test each line.

Different levels of programming language

There are two levels of programming language you need to be aware of for your exam. These are low level programming languages, and high level programming languages.

Low level programming language

A low level programming language is written in what is called an 'assembly language'. This means it closely represents how the CPU works, rather than written in a format that is easier for humans to understand. However, because the code is written closely to how the CPU works, it executes faster.

A low level programming language is translated by an assembler. Low level programming languages are harder to write and understand for humans, which makes them less likely to be used by humans.

High level programming language

A high level programming language is written in a programming language such as Python, Javascript, or Java etc.

A high level programming language is translated by either an interpreter or a compiler. High level programming languages are easier for humans to write and understand as they closer to the human language. However, because they are written to suit humans and are not as closely written to the CPU they take longer to execute.

Translators

When writing a computer program (whether it is written using a low level programming language or a high level programming language) it must be translated in order to be executed. The process of translation is taking the programming code and translating (converting) it into machine code, which means the computer can process and understand the instructions you are giving it.

There are two types of translator you need to be aware of:

- Compiler (used to translate high level programming language)
- Interpreter (used to translate high level programming language)

Compilers and Interpreters

Both compilers and interpreters are used to translate a high level programming language into machine code.

A compiler translates the whole program in one go. This is compared to an interpreter which translates the program line by line.

As a compiler translates the whole program in one go the code is executed much quicker than an interpreter, which does it line by line. However, an interpreter will make it easier to identify errors as the code is executed line by line. This is compared to a compiler which, because it translates the whole program in one go, makes it much more difficult to identify errors.

Integrated Development Environments (IDEs)

An Integrated Development Environment (IDE) is used as a means for programmers to write and execute their code. It provides an easy to use, often simple interface for the programmer to create their programs.

IDEs offer multiple different functions that can help a programmer when they are creating their programs. There are four functions you need to be aware of:

- Code Editor – This provides a place where the programmer can write the programming code, often called the 'shell'
- Debugger/Error Diagnostics – This detects errors in the code. It will identify where the error is , what line number the error is on, and what type of error it is
- Run-time Environment – This allows the user to execute the program one step at a time so they can test each line
- Translator – This allows the code to either be interpreted or compiled, and therefore translated into machine code

It's your turn!

Sally is using an IDE to help her create her computer program.

Explain two functions of the IDE that benefit Sally when creating her program. [4]

Past Exam Questions

Answer the questions below, to help you revise what has been covered in 2.5 Programming Languages and Integrated Development Environments.

1. Paul and Fred are discussing high level programming languages and low-level programming languages. Paul says “high level programming languages and low level programming languages are the same”. Fred says “high level programming languages and low level programming languages are different”.

   State which person is correct, and explain why they are correct. [3]

2. State the purpose of a translator when executing programming code. [1]

3. Identify one translator that is used to translate high level programming languages into machine code. [1]

4. Explain **one** difference between a compiler and an interpreter. [4]

5. Fred is using an IDE to create his computer program. He is told his IDE offers him 'error diagnostics'.

   Describe what is meant by 'error diagnostics' and how it helps Fred create his program. [2]

6. Explain **two different** functions of an IDE, other than error diagnostics. [4]