IGCSE Computer Science 2024~2026

G1 SEMESTER ONE EXAM

Guide to Programming and Logic Gates

Eason Qin
Siddharth Harish
Karthik Sankar

The (OFS) IGCSE Computer Science G1 Exam Guide to Programming and Logic Gates

Eason Qin Luojia (eason@ezntek.com), **Siddharth Harish** (sid.falcon9@gmail.com) and **Karthik Sankar** (karthik@hackclub.com)

Second Revision

November 6th, 2024

Table of Contents

Introduction

What is this guide?

You are looking at the IGCSE Computer Science @ OFS G1 Semester Exam **Guide to Programming and Logic Gates.** This is **an abridged version** of the Computer Science Revision Guide (or just CSRG) made for the G2 examination. As there is so much overlap between programming and logic gates, I have abridged the version for my grade so that you can use it.

Please note that this guide does not cover binary, data representation and systems lifecycles! Please also use your own notes!

This guide aims to cover all about Logic Gates and Programming for the 2024~2026 batch of IGCSE CS students at OFS[1]. It aims to deliver the content in an informative form, but with enough explanation to let you understand all the concepts. This is a text that you should highlight and annotate to your heart's content. It is meant to be informative and explaining; if highlighting helps you, you should do it.

This is revision Two of the guide.

1. Initial Version.
2. Fixed an error about AND gates in the logic gates section (1 AND 1 = 0 to 1 AND 1 = 1)

Note: All references to "I", "Me", "Myself", and similar refer to the main author, Eason Qin.

For a quick tutorial as to how to use this guide, read appendix 5 at the very end. Alternatively, follow this hyperlink.

[1] Sorry if you're from an earlier batch, but this guide is not for you! Content can, however, be extrapolated; it will be done by myself at a later date. E-mail me (Eason) at eason@ezntek.com if you want to know what's happening.

License Notice

[2] Copy my work and expand upon or modify it.

Important Information

This is mostly targeted to people who are less careful with reading.

1. **I expect you to have read all the information below before asking me any questions.** If you ask me a question where the answer is contained in this section, I will only point you to this section.
2. I **expect** you to know that this text, according to the title, ***is a reference guide***. You ***do not have to read the whole text,*** this document is ***for you to refer to***. If you don't want to read, look in the table of contents for what you need, then jump to the section.
3. I **expect** you to know that this guide **may not** contain important points of information for your exam. As mentioned before, **this is a reference guide, *PLEASE still use your own notes and class resources by your teacher to revise*** and only use my guide as a supplement. **This also does not cover any binary or other topics in Chapter One, only prorgramming and logic gates!** The guide was written to the best of our abilities, but since this is not a textbook and is not audited for a long time, expect there to be mistakes and cracks.
4. I **expect** you to find answers to simpler questions in the text itself. The text was written to the best of our abilities, and despite the possibility of cracks, please attempt to read the text before asking. Before asking, also do consult your class resources and the textbook before asking me a question. I may not answer it if you did not do any of the above.
5. I **expect** you to not be upset or angered, irritated, agitated, enraged or feel any negative feelings when I release new revisions of the guide. It is not my fault for updating it to the best of my ability. I am not responsible, and making revisions is not my fault. If you have received an earlier physical copy, please ask me nicely for a new one, or simply use the digital copy. There will be at least 2 revisions; by spotting mistakes for me or reminding me of points that I did not include, you are helping not only yourself but also all other people who have the guide to have the most updated information.
6. If this guide has any references to G2 material or content, please do not be shocked or surprised; simply ignore it if you are informed, and if it doesn't make sense to you as you have never heard of it in your life, also do not be shocked. This is just an abridged version of the G1 guide anyway.

   Email eason@ezntek.com for any concerns on this matter.
7. **You still have to know the Systems Lifecycle and Binary** and other things for your exam!

Notes

1. This Revision Text is authored by Eason Qin Luojia with contributions from Siddharth Harish, and was edited by Karthik Sankar and Eason Qin.
2. The exercises for the Chapter Ten content on Logic Gates may be pulled from the textbook, but some are also produced by myself.
3. **Formal Citations and a bibliography is not provided**, as this is not an academic research document, but a reference booklet produced either from directly pulling examples from the textbook or already synthesized information. If you require citations, please ask me personally.

   If there is information that *must* be cited or would make sense if it is cited, like extremely detailed data points that have not been previously synthesized, the source will **be provided as a footnote. In no case will MLA, APA, Harvard or any other form of formal academic references to be used** as this is not an academic document.

   Legally, all licenses will be followed; i.e. if the document has a license that requires attribution, it will be provided, etc.
4. If an underlined portion of text, like the below:

   Chatbots have mostly been replaced by LLMs, see the AI section below.

   Is seen, and you have the **printed copy**, simply go to the section it says; refer to the table of contents.

   If you have the **digital copy**, you may notice that it is blue. Press on that blue text; it is a *hyperlink* that you can click.

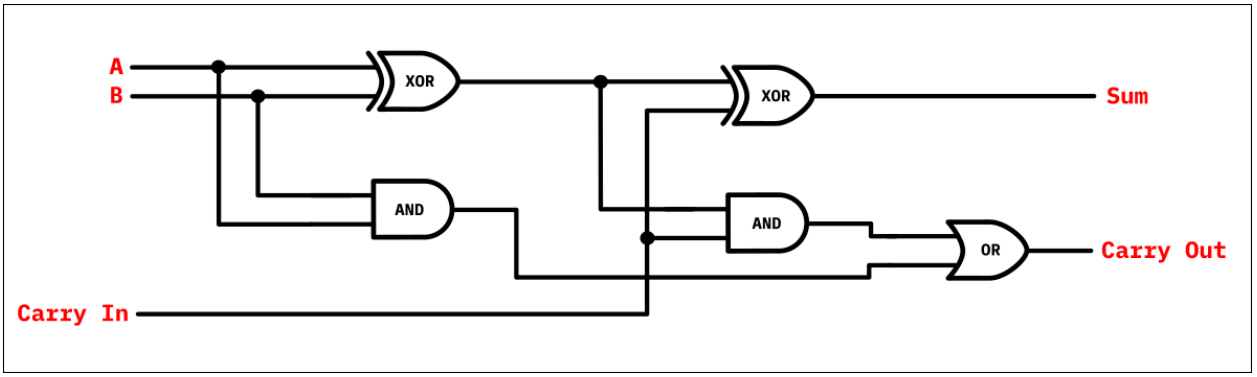
Chapter Ten — Boolean Logic

Logic gates

Think of Logic Gates as part of a circuit[3], however instead of thinking wires as carriers of EMF and current[4], think of them **as either carrying a current or not carrying a current**, i.e. being **on or off,** or **1** and **0**, with **1** being the **presence of current,** and **0** being the **absence of current**.

They are then put together into a **logic circuit**, which is like an electrical circuit, but **instead of having a power source and the beginning and end of the circuit being connected,** logic circuits take a number of **1**s and **0**s as input, and produce a number of **1**s and **0**s as output.

Data always flows from left to right in a logic circuit.



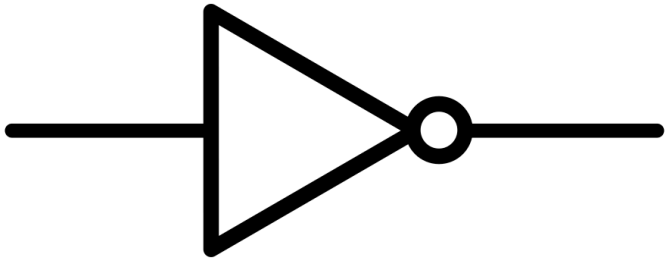
Above: A logic circuit for a full adder.

[3] Technically, they are components of a logic circuit.

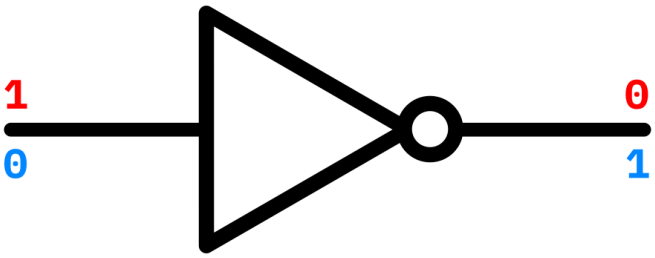
[4] This is not physics, this is computer science. This side of computer science is more seen in computer engineering, where people play with components like Resistors and Capacitors to interact with circuits and chips.

Types of Logic Gates[5]

NOT Gates



This is a **NOT Gate**. What does it do?

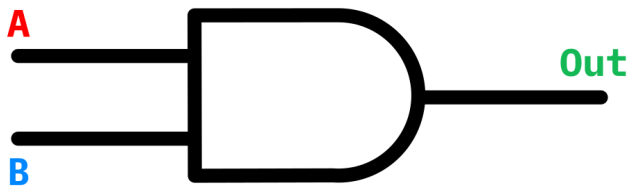


Note that Red and Blue represent different combinations.

From this we can tell that **not gates, when data flows into them, reverses** the value given into it. If a **1** is given to a **NOT** gate, a **0** comes out!

[5] Note that XNOR gates will not be covered as it is out-of-syllabus, but just know that XNOR means exclusive NOR.

AND Gates



This is an **AND Gate**. What does it do?

- **Think of it like a good relationship**. Given two partners, given that **both partner A and partner B** are happy (**1**), they are in a **healthy relationship** (**1**).
- If any of them is **not happy**, it is not a healthy relationship (**0**).

I will represent **happy** as **1**, and **unhappy** as **0**.

		B	
A		**0**	**1**
	0	0	0
	1	0	1

That was a **truth table**. It can also be written like this:

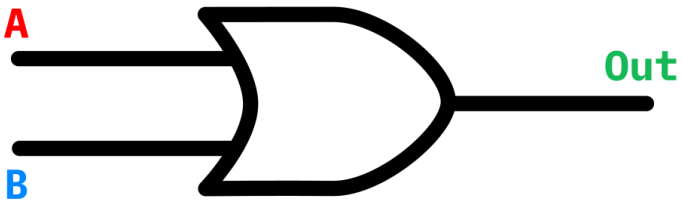
A	B	Out
0	0	0
0	1	0
1	0	0
1	1	1

Truth tables show **all combinations** of inputs for a logic gate or circuit, and shows all the possible outputs for that combination. For single gates, the first type may be used, but for the purposes of the exam and the guide, the second type will be used.

This means that:

- If both person A **and** B are unhappy, the relationship is unhealthy. (0 **AND** 0 = 0)
- If person A is happy but B is unhappy, the relationship is unhealthy. (1 **AND** 0 = 0)
- If person A is unhappy but B is happy, the relationship is unhealthy (0 **AND** 1 = 0)
- If both person A **and** B are happy, the relationship is healthy (1 **AND** 1 = 0)

OR Gates



This is an **OR Gate**. What does it do?

- Think of it like **making a cup of coffee**. Let's assume that a good cup of coffee consists of **creamer** and **sugar.**
- If I put either creamer or sugar, it will taste alright, but if I drink it black, it won't taste as good.[6] **Therefore,** if I put creamer **or** sugar, it will taste good.

This is the truth table:

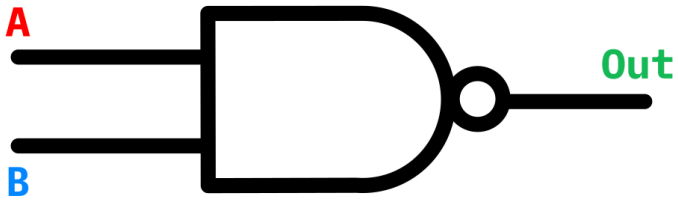
A	B	Out
0	0	0
0	1	1
1	0	1
1	1	1

This means that:

- If you do not put creamer nor sugar, it will **not taste good** (0 **OR** 0 = 0)
- If you put creamer but not sugar, it will **taste good** (0 **OR** 1 = 1)
- If you put sugar but not creamer, it will **taste good** (1 **OR** 0 = 1)
- If you put both creamer and sugar, it will **taste good** (1 **OR** 1 = 1)

[6] Please do not e-mail me arguing about how I should drink my coffee. This is an analogy :)

NAND Gates



This is a **NAND Gate**. What does it do?

- The N, and therefore the circle in front of the gate, means that it is **flipped**.
- This is similar to the AND gate

The truth table is as follows:

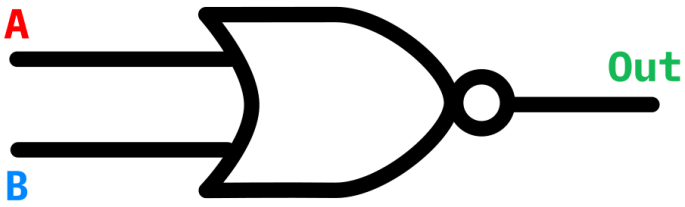
A	B	Out
0	0	1
0	1	1
1	0	1
1	1	0

The outputs are the reverse of the AND gate, the inputs must be **anything but both 1**.

Fun Fact!

You can build every other logic gate with a NAND gate, including NOT, AND and OR, which can then be used to build NOR and XOR. Look it up online!

NOR Gates



This is a **NOR Gate**. What does it do?

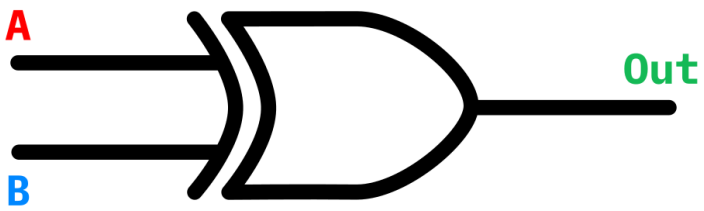
- From the name, one can simply infer what it does. **It is the inverse of the OR gate.**

The truth table is as follows:

A	B	Out
0	0	1
0	1	0
1	0	0
1	1	0

The outputs are the reverse of the OR gate, **none of the inputs can be 1.**

XOR Gates



This is an **XOR Gate** (or an EOR Gate). What does it do?.

- The X stands for **Exclusive**, the full name of the gate is **exclusive or**.
- **Exclusive** here denotes that **only one exclusive input can be 1**, i.e. **either A and B** can be 1, but **not both**.

The truth table is as follows:

A	B	Out
0	0	0
0	1	1
1	0	1
1	1	0

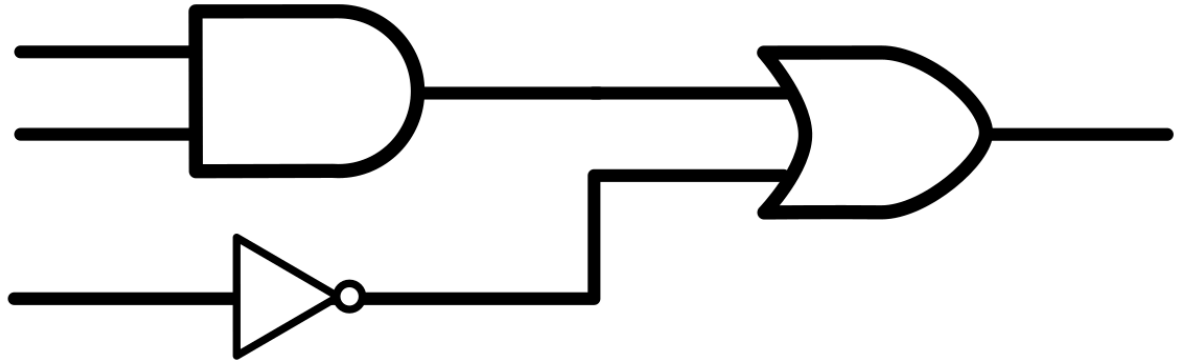
Think of it like **seasoning.** If I have Rock Salt (A) and Sea Salt (B);

- If I don't season my food at all, it will taste bad (0 **XOR** 0 = 0)
- If I season my food with Rock Salt, it will taste good (1 **XOR** 0 = 1)
- If I season my food with Sea Salt, it will state good (0 **XOR** 1 = 1)
- If I season my food with **both rock and sea salt**, my food will be **too salty** and will not taste good (1 **XOR** 1 = 0)

Logic Expressions

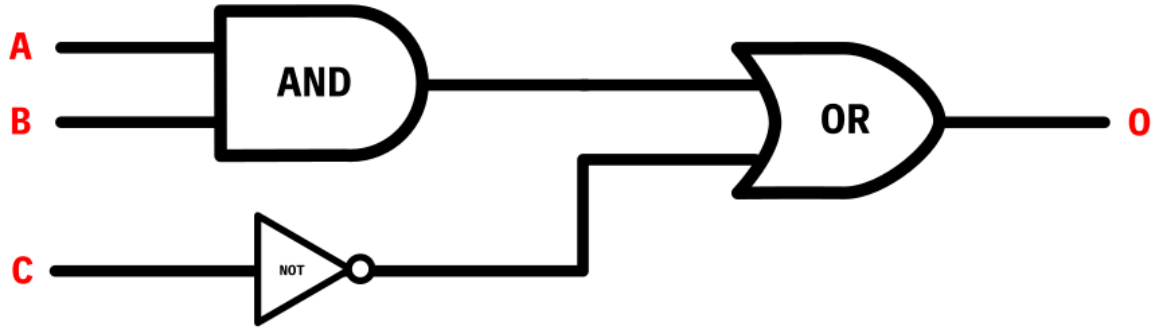
Logic Expressions are ways to express logical circuits in a compact form, like math expressions.

Take the following logical circuit as an example:



We see 3 gates from left to right.

Let us fully label it:



Now, how do we build a logical expression?

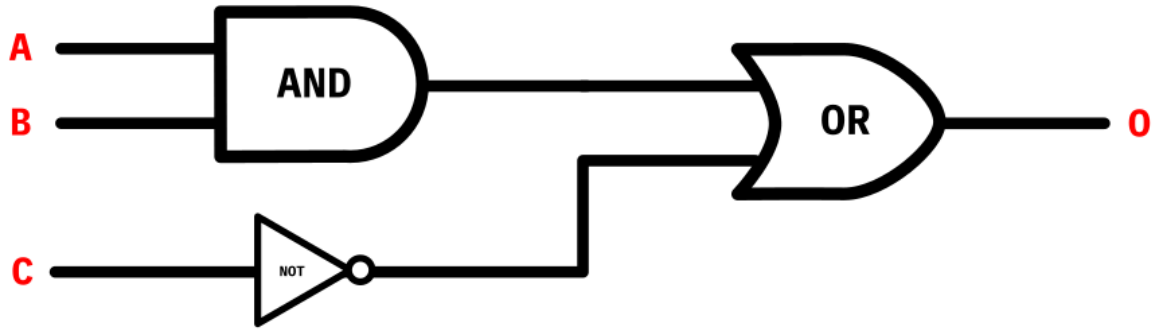
For your reference, the logical expression for this is `(A AND B) OR (NOT C)`.
How do we deduct this?

How to deduct a logical expression from a circuit

Let's follow some rules:

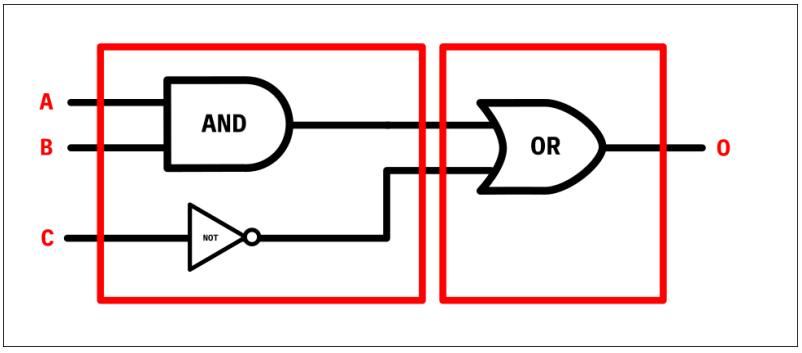
1. Work left to right, or in computer science, **bottom-up**.[7]
2. Divide the diagram into comprehensible sections.
3. When we see a gate, put it in our head.
4. Put its inputs beside it.
5. Put it in brackets.
6. Go to the next gate of the section and jump to step 3 until there are none left.[8]
7. Go to the next section, and jump to step 3 until there are no more sections left. The inputs may be an existing logical expression that you have built, like (A AND B), or just a letter, like B.

Deducting a Logical Expression (Example)



Let us follow the rules.

- Work Bottom-up.
- Divide the diagram into sections.



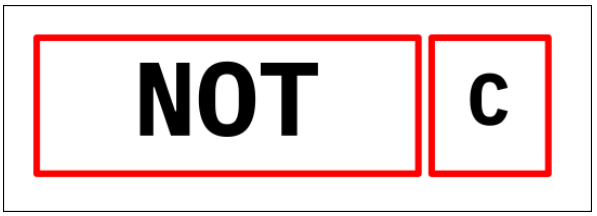
[7] This is assuming that this is a binary tree, except for **NOT** gates. Trees in computer science have the trunk at the top and the most branches at the bottom.

[8] Extra question for you, I just described a loop. Is this a pre-conditioned or post conditioned loop?

- We see **AND**, we will put it in our head.

- Put the inputs beside it.

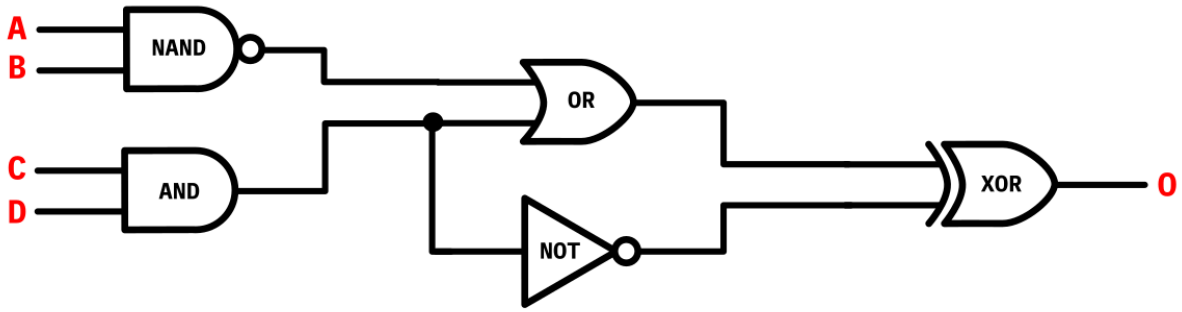
- Put it in brackets.
  `(`**`A`** `AND` **`B`**`)`
- Repeat for NOT:



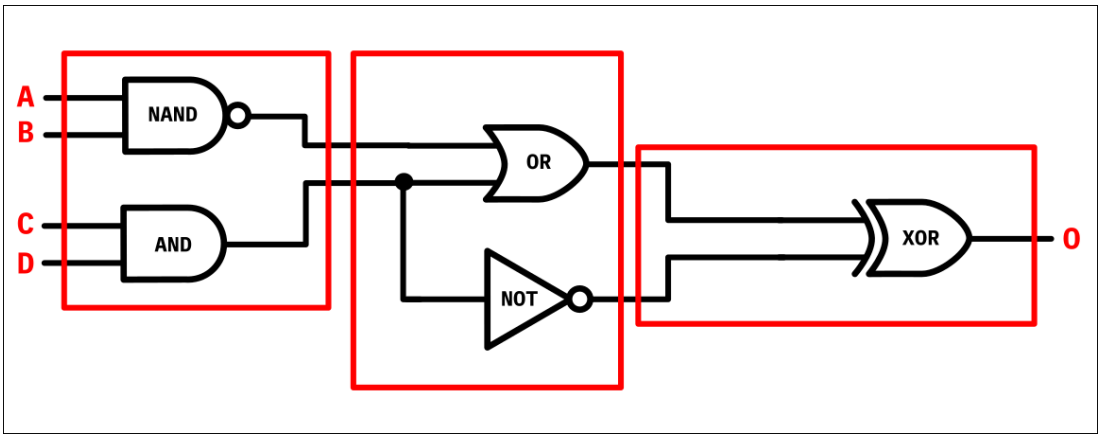
- Go to the next section:

Done! The logical expression is `(`**`A`** `AND` **`B`**`) OR (NOT` **`C`**`).`

Let's try it again with a much harder example:



- Divide it into sections

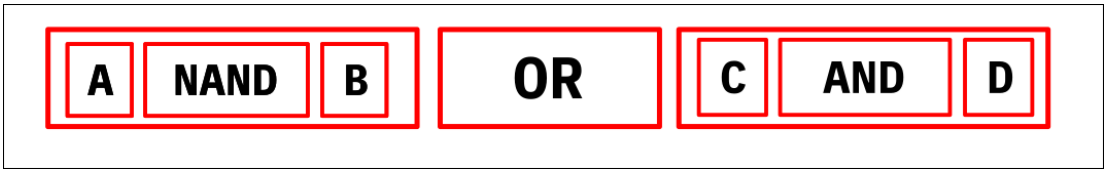


- We see NAND, so we put it in our head.

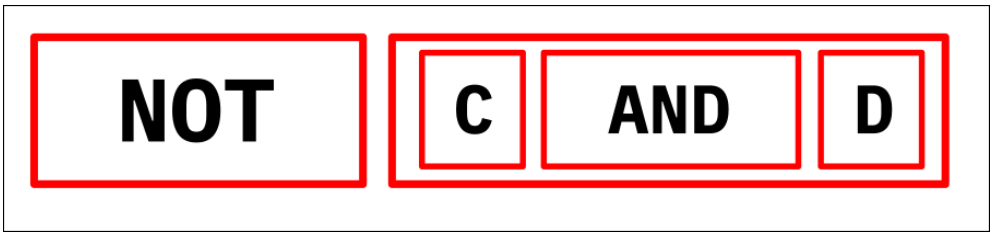
- Fill in the sides:

- Repeat:

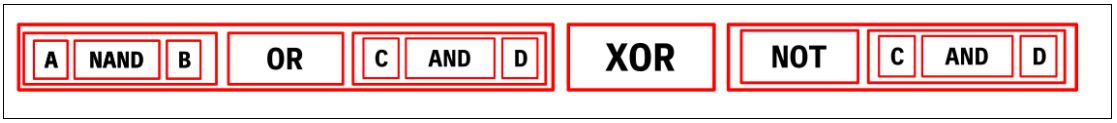
- Since A NAND B serves as the first input of OR, and C AND D serves as the second, we can fit it like so:



- For the NOT gate, C AND D serves as the input, so we can fit it like so:



- Moving onto the XOR gate, the left side would be `(A NAND B) OR (C AND D)`, and the right would be `(NOT (C AND D))`, so the outcome would be:



The result would be

`((A NAND B) OR (C AND D)) XOR (NOT (C AND D))`

This is a much more complicated example, and may not even come on your test.

Boolean Algebra Notation

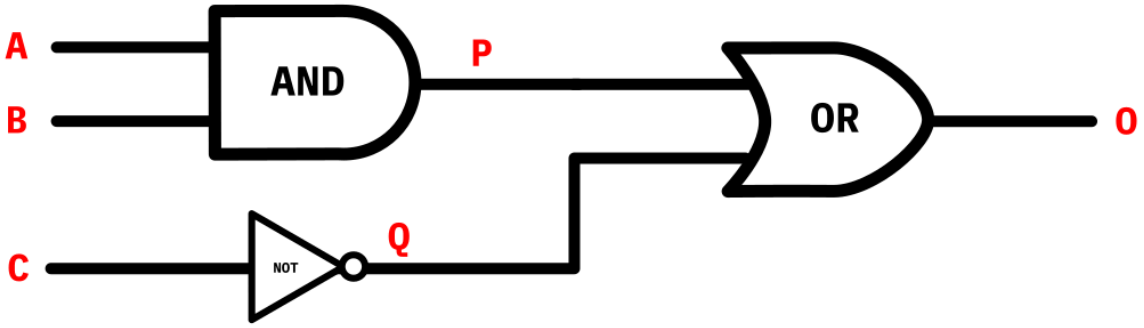
Boolean algebra is a prominent field in mathematics as well. The mathematical portion of boolean algebra you do not need to know for the final IGCSE exam nor for IB SL/HL computer science[9], but may need to know the boolean algebra **notation** for describing logic gates.

Normal	Boolean Algebra
A AND **B**	$A.B$
A OR **B**	$A + B$
NOT **A**	$\overline{A}$
A NAND **B**	$\overline{(A.B)}$
A NOR **B**	$\overline{(A + B)}$
A XOR **B**	$(\overline{A}.B) + (A.\overline{B})$
(**A** AND **B**) OR (NOT **C**)	$(A.B) + \overline{C}$
(**A** NAND **B**) AND (**C** OR **D**)	$\overline{(A.B)}.(C + D)$

[9] Subject to change! A new syllabus is currently being drafted, this may change.

Creating Truth Tables

On the exam, you may be given a logic diagram like so[10]:



And be asked to create a truth table from this[11]. How?

Here are the steps:

1. Draw a table with all letters:

A	B	C	P	Q	O

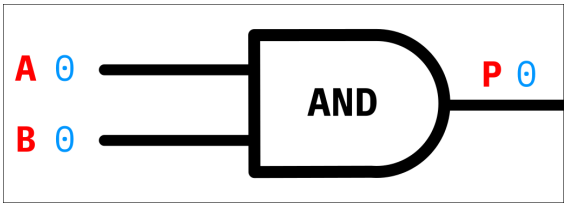
2. Write all the combinations (permutations) A B and C can be in terms of binary values; to do that, you can simply count in Binary. Note that for n inputs, there are 2^n combinations of values.

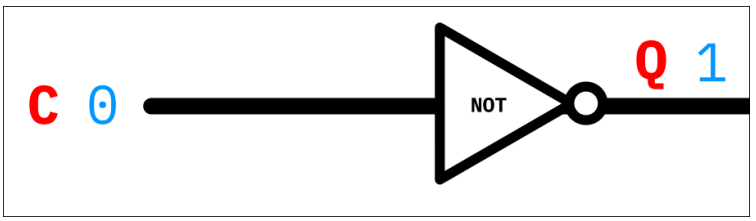
[10] Note that the intermediate steps P and Q might not be on the final IGCSE test but will be on the semester exam.

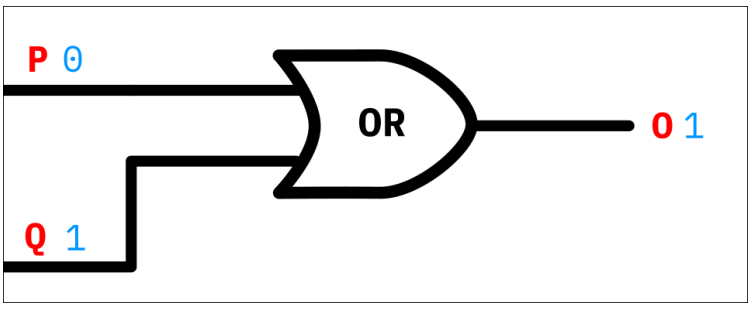
[11] For already advanced students, **you must show all your working** (at least for the G2 2025 examination batch).

A	B	C	P	Q	O
0	0	0			
0	0	1			
0	1	0			
0	1	1			
1	0	0			
1	0	1			
1	1	0			
1	1	1			

3. Begin evaluating each logic gate in the diagram, beginning on the first row. Just like deducting logical expressions, work bottom up, top-down:







4. With these results, simply fill them out:

A	B	C	P	Q	O
0	0	0	**0**	**1**	**1**
0	0	1			
0	1	0			
0	1	1			
1	0	0			
1	0	1			
1	1	0			
1	1	1			

5. Repeat until you are done:

A	B	C	P	Q	O
0	0	0	**0**	**1**	**1**
0	0	1	**0**	**0**	**0**
0	1	0	**0**	**1**	**1**
0	1	1	**0**	**0**	**0**
1	0	0	**0**	**1**	**1**
1	0	1	**0**	**0**	**0**
1	1	0	**1**	**1**	**1**
1	1	1	**1**	**0**	**1**

Programming

Notes

1. Some points and code examples may be pulled from the Pocket IGCSE Pseudocode to Python Reference Guide, which you can read by clicking the Hyperlink or following the following link:

   https://ezntek.com/revision/pseudocode_reference.html

   For your examination (2025 batch), you may write program code[12] or pseudocode for the last question, a question that you should spend 30 minutes on and is like a case study which you must write code for.
2. **This does not aim to be a how to program in pseudocode and Python book.** This is only a short guide/refresher as to how to program in pseudocode; if you want to learn programming you should read Beej's guide to Python Programming, the URL may be found here:

   https://beej.us/guide/bgpython/pdf/bgpython_a4_c_2.pdf

   And you may then translate it to pseudocode when you feel comfortable programming in Python.
3. All values in angle brackets, like so:

   ```
   <variable name>
   <type>
   <value>
   ```

   represent *meta-variables* or *meta-values*, which should wholly, i.e. including the beginning angle bracket, <, to the ending angle bracket, >, be replaced with an actual value that is described within the brackets. You may find examples in the below sections.

[12] The terminology specified by the syllabus refers to code written in a real programming language with a standardized runtime, like Python; as opposed to pseudocode, which is only used to teach computer science.

4. Note that if something spills onto a new line, like:

```
FOR <counter> ← <begin> TO <end> STEP
<step>
    <statement>…
NEXT <counter>
```

Count it as:

```
FOR <counter> ← <begin> TO <end> STEP <step>
    <statement>…
NEXT <counter>
```

Comments

Commenting is like annotating your program[13], and telling readers of your code which part of your code does what. In your IGCSE examinations, you will get marks for commenting your code thoroughly, and even in the real world it is good to comment your code.

IGCSE Pseudocode	Python

`// This is a comment.`
`// To comment, simply put two`
`// slashes (//) in front of your text.`
	`# This is a comment.`
	`# To comment, simply put one`
	`# hashtag (#) in front of your text.`

Values

Values, otherwise known as **literals,** are direct, raw values. These are used to represent things like whole numbers and negative numbers (integers), decimal numbers (floats), portions of text (like "good morning, computer!") (strings), True or False values (booleans), and single characters like 'y' or 'n'.

IGCSE Pseudocode	Python

`// These are all INTEGER's, or whole`
`// numbers`
`42`
`-2043`
	`# These are all int's, or whole`
	`# numbers`
	`42`
	`-2043`

`// These are all REAL's, or decimal`
`// numbers`
`3.14159`
`56.52`
	`# these are all float's, or decimal`
	`# numbers`
	`3.14159`
	`56.52`

`// These are STRING's, or "text"`
`// (enclosed in only "):`
`"Good morning, user!"`
`"Thomas"`
`"Jason Lee"`
	`# These are str's, or "text"`
	`# (enclosed in both " and ')`
	`"Good morning, user!"`
	`'Thomas'`
	`'Jason Lee'`

`// These are BOOLEAN's, either TRUE or`
`// FALSE`
`TRUE`
`FALSE`
	`# These are bool's, either TRUE or`
	`# FALSE`
	`True`
	`False`

`// These are CHAR's, or singular`
`// characters (enclosed only in '):`
`'c'`
`'F'`
`'b'`
	`# there is no CHAR in Python, just use a`
	`str.`

[13] A sequence of instructions that reaches the objective/problem you are trying to solve.

Variables & Types

We have learned of values of different *types*. In order to store them somewhere so that we can keep track of values throughout the life of our program, we use **variables**[14].

These are like storage boxes, cans; anything you can imagine that stores something. Variables can hold a value, like a number that represents the user's age, a message to broadcast to computers over the network, or simply a counter to keep track of a loop. They have a name, or **identifier**, a **type**[15], and a **value**. Under the hood, these are kept in the computer's **RAM**, or **memory**[16].

Above: a (bad) visualization of what a variable is

In order to use a variable, you must **make it clear to the computer** that the variable will exist and be used in the first place; almost like allocating space in a warehouse. To do this:

IGCSE Pseudocode	Python
`DECLARE <variable name>: <type>`	`<variable name>: <type>`

`// e.g.`
`DECLARE Name: STRING`
`DECLARE TotalScore: INTEGER`
	`# e.g.`
	`name: str`
	`total_score: int`

Now we have *declared* our variables, now what do we do?

[14] Not math variables! In math, variables represent either unknowns or predefined values. We do not deal with unknowns in programming, but pre-defining a value and saving it somewhere is common, the place you save it to is called a **constant**.
[15] Optional in Python, but very much recommended.
[16] Refresher: this is short-term memory used by all programs to store temporary data and is fully erased upon restarting your computer.

We can give them values by **assigning** a value to a variable. Note that in order to use a variable, you must set it to something first. Declared variables will have the value None in Python[17].

IGCSE Pseudocode	Python

`<variable name> ← <expression>`
`// NOTE: you may write it like <- in`
`// your computer.`
 	
`// e.g.`
`Name ← "Thomas"`
`TotalScore ← 84`
`Name ← FirstName`
	`<variable name> = <expression>`
	
	`# e.g.`
	`name = "Thomas"`
	`total_score = 84`
	`name = first_name`

This will then put “Thomas” inside the variable named Name, put 84 in TotalScore, etc. You can even assign variables to other variables to copy them!

Constants

These are similar to variables, but they are values that do not change throughout the program, like a grade boundary in a program that determines the achievements of students.

IGCSE Pseudocode	Python

`CONSTANT <variable name> <- <value>`
 	
`// e.g.`
`CONSTANT Teacher <- "Hubbard"`
`CONSTANT PassMark <- 70`
	`<VARIABLE_NAME> = <value>`
	
	`# e.g.`
	`TEACHER = "Hubbard"`
	`PASS_MARK = 70`

[17] Or it is simply undefined in many languages.

Variable Casing Conventions

This is an often overlooked part of programming; **naming your variables is quite important!** This section of the guide will not tell you what to call your variables exactly, you can find some guidelines as to how to *name* variables **in the appendix**.

For context, **you cannot have spaces in variable names when you write code**[18]. This is why you must follow a convention to name variables that would otherwise have spaces. Here's how you would name variables without spaces:

PascalCase (IGCSE Pseudocode)	**snake_case** (Python)
`FirstName`	`first_name`
`GoodMorningAmerica`	`good_morning_america`
`OutsideTemperature`	`outside_temperature`
`ClassroomStutdents`	`classroom_students`
`StudentMarks`	`student_maks`

PascalCase was introduced back when the **Pascal Programming Language**[19] was conceived, it requires one to:

- Have the first letter of every word that follows capitalized
- Words do not need to be separated

snake_case was around for a long time, and was first mentioned in the **C Programming Language** book by Brian Kernighan and Dennis Ritchie, but was popularized by Python. It specifies the following rules:

- All letters must be lowercase
- Words must be separated by underscores

NOTE: When using Constants in Python, make sure that you use **SCREAMING_SNAKE _CASE,** which is capitalized snake_case. In this guide, all Pseudocode variable and function/procedure names will be in PascalCase, and for the Python Equivalents I will rewrite them in snake_case. Consider doing the same.

[18] Using spaces makes the code extremely hard to parse (process your code in order to execute it)

[19] Fun fact! Pascal was created as a **compiler for pseudocode**, so that people could write simple and human-readable code. It was then adopted by many academic institutions and companies at the same time; however, it was phased out.

Input and Output

To make the program useful for a user, the program must be able to take some input or values from the user somehow, and also output things for the user to see. In these languages, outputs and inputs are all on the console.

IGCSE Pseudocode	Python

`OUTPUT <expression>`
`OUTPUT <expression>, ...`
`// Print however many things you`
`// require.`
 	
`INPUT <expression>`
 	
`// e.g.`
`OUTPUT "What is your name"`
`OUTPUT "Welcome", Name`
`OUTPUT "What is your Social Security Number?"`
`INPUT SocialSecurityNumber`
`OUTPUT "What is your ID?"`
`INPUT ID`
	`print(<expression>)`
	`print(<expression>, ...)`
	`# Print however many things you`
	`# require.`
	
	`<variable name> = input(<prompt>)`
	
	`# e.g.`
	`print("What is your name")`
	`print("Welcome", name)`
	
	`# Note that if you need to input`
	`# something into an integer, you must`
	`# wrap input in int, or separate them`
	`# like so:`
	`social_security_number = int(input("What is your Social Security Number?"))`
	
	`id = input("What is your ID?")`
	`id = int(id)`

We have learned enough to be able to **write a program to take the user's name, and say "Hello, <name>!"**

Advanced learners, you may choose to do so alone, but the answers are below:

IGCSE Pseudocode	Python

`DECLARE Name: STRING`
`OUTPUT "What is your name"`
`INPUT Name`
`OUTPUT "Hello ", Name`
	`name: str`
	`name = input("What is your name")`
	`print("Hello ", name)`

Note that declaring the variable in Python is mostly redundant; even when using a ***type checker***[20]*. You may simply write* `name: str = input("What is your name")` *as well.*

[20] Python does support types, but they are not **enforced** (your code will work without them or if you assign a string to an int, let's say), but there are programs that watch your code as you write it, that warn you of type *violations* and *effectively enforces types upon your program* (mypy, pyright). Programmers like myself enjoy it as it catches type-related errors much earlier on in the programming cycle.

Math

Math in programming is quite simple. Here's how you do calculations on variables and values:

IGCSE Pseudocode	Python

`<expr> <operator> <expr>`
 	
`// e.g.`
`2 + 5`
`3 - 1`
`4 * 4 // multiplication`
`26 / 2 // division`
 	
`// here are the special operators`
`13 DIV 2`
`5 MOD 2`
 	
`// you can group them together`
`(3 * X) + 1`
 	
`// you can combine it with an`
`// assignment, like so:`
`NextTerm ← X + 1`
	`<expr> <operator> <expr>`
	
	`# e.g.`
	`2 + 5`
	`3 - 1`
	`4 * 4 # multiplication`
	`26 / 2 # division`
	
	`# here are the special operators`
	`13 // 2`
	`5 % 2`
	
	`(3 * x) + 1`
	
	`# you can combine it with an`
	`# assignment, like so:`
	`next_term = x + 1`

Note that in programming, there are two special operators that do not exist in normal math:

- Floor Division (DIV): This means doing normal division, but removing all decimals, if any.
- Modulus (MOD): This means doing division, but giving the **remainder** of the division instead.

You can also have **arithmetic assignments**, like so:

IGCSE Pseudocode	Python

`// They DO NOT exist in pseudocode,`
`// but may be substituted with:`
 	
`<ident> ← <ident> <operator> <expr>`
 	
`// e.g.`
`Age ← Age + 1`
`Temperature ← Temperature - 5`
	`<ident> <operator>= <expr>`
	
	`# e.g.`
	`age += 1`
	`temperature -= 5`

They are language **constructs**[21] that allow you to do math on a variable and immediately assign it back, you can see the behind-the-scenes on the left side.

You may increment one to your age on your birthday, or you may increment one to the score counter for a team in a basketball video game. This would be the way that you would do it.

Random Numbers and Rounding

You can also generate random values and round numbers like so:

IGCSE Pseudocode	Python

`ROUND(<ident>, <decimal places>)`
`RANDOM() // gives a number between 0`
`// and 1, with decimals`
 	
`// Round a number to 1dp`
`ROUND(5.23, 1)`
`// Generate an integer between 0 and 10`
`ROUND(RANDOM() * 10, 0)`
	`round(<expr>, <decimal places>)`
	
	`# you must put this at the beginning of`
	`# the file for this to work!`[22]
	`import random`
	
	`random.random()`
	`random.randint(<lower>, <upper>)`
	
	`# Round a number to 1dp`
	`round(5.23, 1)`
	`# Generate an integer between 0 and 10`
	`round(random.random() * 10, 0)`
	`# or`
	`random.randint(0, 10)`

Comparison Expressions[23] (Boolean Expressions)

These expressions allow you to compare two values based on these things:

- Equality (Are they the same)
- Inequality (Are they **not** the same?)
- Greater than
- Lesser than
- Greater than or equal to
- Less than or equal to

[21] A part of a programming language that makes up for the whole programming language. Output and Input are constructs of pseudocode, as it is a component of pseudocode; and its syntax sets it apart.

[22] I.e. **importing**. Since random is not part of the built-in function set as it is not as commonly used. It is isolated into a **library** which is a collection of reusable code that you can "borrow" anytime

[23] This is a **boolean expression**, meaning that it has two values (hence the bi- prefix)

They all evaluate to **booleans**, meaning that you can substitute this for a direct boolean value and therefore be able to use it in things such as and not limited to while loops, if statements, C-style for loops if you choose to learn Java in the future, etc.

IGCSE Pseudocode	Python
`// Equality` `Age = 18`	`# Equality` `age == 18`

`// Greater than, less than`
`Age > 18`
`Age < 18`
	`# Greater than, less than`
	`age > 18`
	`age < 18`

`// Greater than or equal to, less`
`// than or equal to`
`Age >= 18`
`Age <= 18`
	`# Greater than or equal to, less`
	`# than or equal to`
	`age >= 18`
	`age <= 18`

`// Not equal to` `Age <> 18`	`# Not equal to` `age != 18`

Logical Expressions[24]

Remember logic gates? Yeah, you can have them in programming too! But instead of them being in a part of a *circuit* and *evaluating* to **1**'s and **0**'s, they take in booleans[25] on each side and evaluates to a boolean, similar to a logical **expression**.

IGCSE Pseudocode	Python
`// is one condition TRUE AND the` `// other one true?`	`# is one condition TRUE AND the` `# other one true?`
`ConditionOne AND ConditionTwo`	`condition_one and condition_two`
`// is one condition TRUE OR the` `// other one true?`	`# is one condition TRUE OR the` `# other one true?`
`ConditionOne OR ConditionTwo`	`condition_one or condition_two`
`// is the condition NOT true?`	`# is the condition NOT true?`
`NOT Condition`	`not condition`

On the next page are some examples to make it clearer:

[24] This is **also a boolean expression**.

[25] Just as a reminder, booleans are True or False, they represent binary values and can only be either true or false, like `IsWaterHot` or `IsPlayerAlive`.

IGCSE Pseudocode	Python

`// is the coffee warm and sweetened?`
 	
`CoffeeWarm AND CoffeeSweet`
 	
`// Does the player have more than 10`
`// points OR is alive?`
 	
`IsPlayerAlive OR PlayerPoints > 10`
 	
`// Has the exit button not been`
`// pressed?`
 	
`NOT ExitButtonPressed`
	`# is the coffee warm AND sweetened?`
	
	`coffee_warm and coffee_sweet`
	
	`# does the player have more than 10`
	`# points OR is alive?`
	
	`is_player_alive or player_points > 10`
	
	`# Has the exit button not been pressed?`
	
	`not exit_button_pressed`

Expressions

I must introduce the concept of **expressions** so that you can **effectively use a programming language**, to easily funnel your thoughts and logical thought process into a program efficiently by chaining expressions.

So far, we have covered a handful of expressions:

- Math Expressions
- Logical Expressions
- Comparison Expressions

All of the above, as per the name, are **expressions**. These are chunks of code that evaluate or expand to values when the program is run, that can then be used as a part of statements or other expressions. What I mean is:

IGCSE Pseudocode	Python

`// Step one: Original`
`// Assume score is 5 and boundary is`
`//  7, and IsAlive is True`
 	
`(Score + 1 < Boundary) AND IsAlive`
 	
`// Step Two`
 	
`(6 < 7) AND IsAlive`
 	
`// Step Three`
 	
`FALSE AND TRUE`
 	
`// Step Four`
 	
`FALSE`
	`# Step one: Original`
	`# Assume score is 5 and boundary is`
	`# 7, and is_alive is True`
	
	`(score + 1) < boundary and is_alive`
	
	`# step two`
	
	`(6 < 7) and is_alive`
	
	`# step three`
	
	`False and True`
	
	`# step four`
	
	`False`

That was a simulation of what happens under the hood. All expressions are able to magically mutate and evaluate to things. Since variables may be included, the outcomes may also change.

Here's an exercise; What if `Boundary` were 7 and score were 7? Would the outcome be True or False?

Here is a non-exhaustive list of things that can be expressions in both Python and Pseudocode:

- Math Operations
- Comparison Operations
- Logical Operations
- Function Calls (does not include functions that return values).
- Array Indexes
- String Indexes
- String Slices (substrings in pseudocode)
- Array Slices **(Python Only)**

For advanced learners, visit **section four** of the appendix to find an actually exhaustive list of expressions.

Conditional Branching (Selection)

Now we have covered some basic statements so that you can do math, how do we **branch the flow of execution**? How do we do something if something holds true, and what do we do otherwise?

We use if statements for that, these are like making decisions or asking the computer questions, and doing stuff if the answer is yes or no (True or False). Here's an example:

IGCSE Pseudocode	Python

`// either:`
`IF <condition>`
`  THEN  // PRESS SPACE TWICE!`
`  <code>  // PRESS SPACE TWICE!`
`ELSE`
`  <code>  // PRESS SPACE TWICE!`
`ENDIF`
	`if <condition>:`
	`  <code>  # PRESS SPACE 4 TIMES!`
	`else:`
	`  <code>`

```
// or:
IF <condition>
  THEN
    <code>
ENDIF

// e.g.
IF Age > 18
  THEN
    OUTPUT "you can drink!"
ELSE
  OUTPUT "you cannot drink..."
ENDIF
```

```
# or
if <condition>:
    <code>

# e.g.
if age > 18:
    print("you can drink!")
else:
    print("you cannot drink...")
```

Let's immediately try an exercise. *Write a program that asks the user for a temperature, and decides if a glass of water at that temperature is too hot to drink. Assume all temperatures are in Celsius, and that "too hot" is 45°C.*

Answers are here:

IGCSE Pseudocode

```
DECLARE WaterTemp: INTEGER
OUTPUT "How hot is your water?"
INPUT WaterTemp

IF WaterTemp > 45
  THEN
    OUTPUT "The water is too hot!"
ELSE
  OUTPUT "You can drink the water..."
ENDIF
```

Python

```
water_temp: int
water_temp = int(input("How hot is your
water?"))

if water_temp > 45:
    print("The water is too hot!")
else:
    print("You can drink the water...")
```

Note that in Python, input only returns **str**, meaning that you must manually convert the type, whereas in pseudocode, INPUT returns whatever type you declared the variable with. But that's it!

The program:

- Declares WaterTemp as an integer
- Prints out "How hot is your water?", and grabs the user's input
- If the water temperature is greater than 45, it outputs that the water is too hot, otherwise it tells the user that you can drink the water.
    - Hence, the program **branches its execution** depending on the condition, which is `WaterTemperature > 45`.

You can also chain if statements by using **elif** in Python. This means that when you are done checking the first condition, you move onto the next one.

IGCSE Pseudocode

```
// This does not exist in pseudocode,
but can be emulated in the following
way:

IF <condition>
  THEN
    <code>
ELSE
  IF <condition>
    THEN
      <code>
  ELSE
    <code>
ENDIF

// the IF statement inside the
// larger ELSE statement can be
repeated
// as many times as needed.

IF Age > 18
  THEN
    OUTPUT "You can drink!"
ELSE
  IF Age > 16
    THEN
      OUTPUT "You can almost drink!"
  ELSE
    OUTPUT "You can't drink..."
ENDIF
```

Python

```
if <condition>:
    <code>
elif <condition>:
    <code>
else:
    <code>

# e.g.
if age > 18:
    print("you can drink!")
elif age > 16:
    print("you can almost drink!")
else:
    print("you can't drink...")
```

BEWARE: In pseudocode, you must follow the exact indentation of **THEN** being on a new line, two spaces indented, and the code two spaces more. The code after **ELSE** must only be indented by two spaces. This syntax is enforced in the examination, and is considered in the mark scheme. **All other code blocks are indented by 4 spaces**.

With the previous example, **can you modify your water temperature program to print the water is too cold if it is less than 5 degrees?**

Answers are on the next page!

IGCSE Pseudocode	Python

`DECLARE WaterTemp: INTEGER`
`OUTPUT "How hot is your water?"`
`INPUT WaterTemp`
 	
`IF WaterTemperature > 45`
`  THEN`
`    OUTPUT "The water is too hot!"`
`ELSE`
`  IF WaterTemperature < 5`
`    THEN`
`      OUTPUT "The water is too cold!"`
`  ELSE`
`    OUTPUT "You can drink the water..."`
`  ENDIF`
`ENDIF`
	`water_temp: int`
	`water_temp = int(input("you can drink!"))`
	
	`if water_temp > 45:`
	`    print("The water is too hot!")`
	`elif water_temp > 5:`
	`    print("The water is too cold!")`
	`else:`
	`    print("You can drink the water...")`

By the Pseudocode example, we can see that elif simply means asking many questions at one time; in Python, elif is simply a shortcut for the pseudocode example on the left.

Else-if chaining vs writing many If statements

Given this code that calculates grade boundaries,

IGCSE Pseudocode	Python

`DECLARE Grade: INTEGER`
`OUTPUT "Enter your grade”`
`INPUT Grade`
 	
`IF Grade > 85`
`  THEN`
`    OUTPUT "Distinction"`
`ELSE`
`  OUTPUT "Fail"`
`ENDIF`
	`grade: int`
	`grade = int(input("Enter your grade"))`
	
	`if grade > 85:`
	`    print("Distinction")`
	`else:`
	`    print("Fail")`

What happens if you wanted to add another condition to see if the user got 70 marks and above and therefore a regular pass? The naïve solution is on the next page:

IGCSE Pseudocode	Python

`DECLARE Grade: INTEGER`
`OUTPUT "Enter your grade"`
`INPUT Grade`
 	
`IF Grade > 85`
`  THEN`
`    OUTPUT "Distinction"`
`ENDIF`
 	
`IF Grade > 70`
`  THEN`
`    OUTPUT "Pass"`
`ELSE`
`  OUTPUT "Fail"`
`ENDIF`
	`grade: int`
	`grade = int(input("Enter your grade"))`
	
	`if grade > 85:`
	`    print("Distinction")`
	
	`if grade > 70:`
	`    print("Pass")`
	`else:`
	`    print("Fail")`

However, what happens if the data is 90? **The first condition will cause the program to print** "Distinction", **but as there is another if statement** (asking a different question), and 90 is greater than 70, **the program will also print** "Pass", which is not the intended behavior.

However by using else-if chaining:

IGCSE Pseudocode	Python

`DECLARE Grade: INTEGER`
`OUTPUT "Enter your grade"`
`INPUT Grade`
 	
`IF Grade > 85`
`  THEN`
`    OUTPUT "Distinction"`
`ELSE`
`  IF Grade > 70`
`    THEN`
`      OUTPUT "Pass"`
`  ELSE`
`    OUTPUT "Fail"`
`ENDIF`
	`grade: int`
	`grade = int(input("Enter your grade"))`
	
	`if grade > 85:`
	`    print("Distinction")`
	`elif grade > 70:`
	`    print("Pass")`
	`else:`
	`    print("Fail")`

The program will only print distinction, as the first condition is met, and it will jump outside the if statement.

Pattern Matching

Pattern matching also allows you to branch the flow of your program's execution, but it works a little bit differently than If statements.

Consider the following program written by a programmer very biased towards Apple:

IGCSE Pseudocode

```
DECLARE PhoneBrand: STRING
OUTPUT "Who made your phone?"
INPUT PhoneBrand

IF PhoneBrand = "Apple"
  THEN
    OUTPUT "You are the best!"
ELSE
  IF PhoneBrand = "Google"
    THEN
      OUTPUT "Your phone is trash!"
  ELSE
    OUTPUT "Your phone is so trash
that I don't even know who made it!"
  ENDIF
ENDIF
```

Python

```
phone_brand: str
phone_brand = input("Who made your
phone?")

if phone_brand == "Apple":
    print("You are the best!")
elif phone_brand == "Google":
    print("Your phone is trash!")
else:
    print("Your phone is so trash that I
don't even know who made it!")
```

That's quite inefficient! Now imagine if the programmer would like to insult Xiaomi, Samsung and Huawei users. We would need to repeat it 5 times in total! Not efficient, right?

We can use pattern matching to solve that. Instead of checking the result one by one, we can just check if the value we have in the variable matches any of our choices all at once[26]. **Note that to use Pattern Matching in Python, you must have Python 3.10 and above.** If you use Replit or have an installation of Python from Python.org, you will be fine.

The example is on the next page:

[26] To really understand why it is so much better to pattern-match, you would need to be fluent in C and know how value-based hashing functions work, and how assembly jumps work. Just know that it can magically jump to the line number with the code just by looking at the value once.

IGCSE Pseudocode	Python

`CASE OF <expr>`
`  <expr>: <statement>`
`  <expr>: <statement>`
`  ...`
`  // optionally,`
`  OTHERWISE <statement>`
`ENDCASE`
 	
`// e.g.`
`CASE OF PhoneBrand`
`  "Apple": OUTPUT "You are the best!"`
`  "Xiaomi": OUTPUT "Are you Chinese?"`
`  "Google": OUTPUT "Your phone is trash!"`
`  "Samsung": OUTPUT "Are you Korean?"`
`  OTHERWISE OUTPUT "Your phone is so trash I don't even know who made it!"`
`ENDCASE`
	`match <expr>:`
	`  case <expr>:`
	`  <code>`
	`  case <expr>:`
	`  <code>`
	`  ...`
	`  # This is equivalent to OTHERWISE`
	`  case _:`
	`  <code>`
	
	`match phone_brand:`
	`  case "Apple":`
	`  print("You are the best!")`
	`  case "Xiaomi":`
	`  print("Are you Chinese?")`
	`  case "Google":`
	`  print("Your phone is trash!")`
	`  case "Samsung":`
	`  print("Are you Korean?")`
	`  case _:`
	`  print("Your phone is so trash I don't even know who made it!")`

The code above outputs a message depending on the value of PhoneBrand, but if it doesn't match any of them, it goes to the **otherwise** section of the code, which runs the output statement there. Note that you can repeat as many of them as you need.

Pre- and Post-condition Loops

Pre-condition loops (While Loops)

One of the next fundamental pillars of programming is **looping** or **iteration**. In layman's terms, iteration is just doing something repeatedly, while a condition is met. It is better to show you how we can use code to season some fish and chips[27]:

As always, the general form goes first.

IGCSE Pseudocode	Python

`WHILE <condition> DO`
`    <code>`
`ENDWHILE`
 	
`// e.g.`
`DECLARE SaltGrams: INTEGER`
`SaltGrams <- 0`
`WHILE SaltGrams < 3 DO`
`    SaltGrams <- SaltGrams + 1`
`    OUTPUT "There is ", SaltGrams,`
`"Grams of salt on your Fish and Chips"`
`ENDWHILE`
	`while <condition>:`
	`    <code>`
	
	`# e.g.`
	`salt_grams: int = 0`
	`while salt_grams > 1:`
	`    salt_grams += 1`
	`    print("There is ", salt_grams,`
	`"Grams of salt on your Fish and Chips.")`

Here, we run the following instructions:

- Add 1 gram of salt to SaltGrams
- Output "There is <SaltGrams> grams of salt on your Fish and Chips

While there is less than 3 grams of salt[28]. Since procedures and functions were not introduced yet, you will have to view the examples in the appendix.

[27] This is targeted.

[28] Unfortunately, British food is simply too bland.

Post-condition loops (Do-While, Repeat-While, Repeat-Until Loops)

This differs from pre-conditioned loops. **In a pre-conditioned loop, the code may not even be run once,** as the condition is checked before the loop begins. **In a post-conditioned loop, the code is always run once** as the condition is checked before the loop begins. Since I cannot show you assembly language[29], I will simply show you in IGCSE pseudocode with some C syntax to show labels[30] (note that **GOTO** simply makes the code go to the label in this case)

IGCSE Pseudocode with C labels

```
DECLARE Password: STRING
OUTPUT "Enter your Password"
INPUT Password
LoopBegin:
IF Password <> "Secure Password"
  THEN
    OUTPUT "Wrong Password!"
    OUTPUT "Enter your Password again:"
    INPUT Password
    GOTO LoopBegin
ENDIF
```

If the password is equal to “Secure Password” on the first try, the program will not ask the user the second time to enter the correct password. Only if it is wrong will the code run.

Let us now see the post-condition example:

IGCSE Pseudocode with C labels

```
DECLARE Password: STRING
OUTPUT "Enter your Password"
INPUT Password
LoopBegin:
OUTPUT "Wrong Password!"
OUTPUT "Enter your Password again:"
INPUT Password

IF Password <> "Secure Password"
  THEN
    GOTO LoopBegin
ENDIF
```

Now the condition is run at the very end of the loop.

[29] One day, one day ;)
[30] Please do not write this in your exam.

But now the code does not make sense, why does it ask for your password, say it's wrong, only for you to enter it again, and only then does the check start?

These two types of loops are **not interchangeable**, as post-condition loops like this one would always run the contents of the loop, regardless of the condition being true the first try or not. You must modify the code like so:

IGCSE Pseudocode with C labels

```
DECLARE Password: STRING
LoopBegin:
OUTPUT "Enter your Password:"
INPUT Password
IF Password <> "Secure Password"
  THEN
    OUTPUT "Wrong Password!"
    GOTO LoopBegin
ENDIF
```

To make it correct again. Note that pseudocode does not have C's do-while, it has repeat-until, doing something **while the condition is not met yet.**

Python does not have post condition loops. After Python's release and rise of popularity as a language, it ideologically killed the post-condition loop as being mostly unnecessary. Modern languages like Rust, Ruby, Lua, Kotlin, JavaScript and TypeScript, etc. all do not have post-condition loops. **Technically, you could get by with while loops**, but you must understand the difference between the two for your exams. You can hack them into Python in the example below.

IGCSE Pseudocode

```
REPEAT
    <code>
UNTIL <condition>

// e.g.
REPEAT
    OUTPUT "Enter the password..."
    INPUT Password
    IF Password <> "Secret"
      THEN
        OUTPUT "Wrong..."
    ENDIF
UNTIL Password = "Secret"
```

Python

```
while True:
    password = input("Enter the
password...")
    if password != "Secret":
        print("Wrong...")
    else:
        break
```

You can also exit a loop or go back to the top of the loop with break and continue, respectively, in Python.

Arrays and Lists

Arrays (or Lists in Python[31]) are data structures that allow you to store a **sequence** or a **list** of values.

In Pseudocode, arrays are **static**, meaning that you cannot put extra things onto the list or remove them whenever you please; the length is fixed at a set value and cannot be changed. However, in Python, lists are **dynamic**, meaning that you can add or remove elements from the back whenever you please.

Consider writing a program for a small classroom that checks the attendance of each student, where there are 5 of them. You would need to store the name of each student in some sequence, how would we do that?

The code example is below; the standard form will always be first:

IGCSE Pseudocode

```
DECLARE <ident>:ARRAY[l,h] OF <type>

// Declaring an ARRAY (2-dimensional)
//
// l1 and h1 are the bounds of the
// first dimension, l2 and h2 are the
// bounds of the second dimension
DECLARE <ident>:ARRAY[l1,h1:l2,h2] OF
<type>

// e.g.
DECLARE StudentNames:ARRAY[1,5] OF
STRING

// Adapted from the IGCSE Syllabus
DECLARE TicTacToe:ARRAY[1,3:1,3] OF
CHAR

// Assign to an ARRAY (1 dimensional)
StudentNames[2] ← "Marcos"
TicTacToe[1,3] ← 'X'

// Use an ARRAY
<ident>[<index>] // 1D ARRAY
```

Python

```
# you do not have to specify bounds!
<ident>: list[<type>]

# Declaring a list (2-dimensional)
<ident>: list[list[<type>]]

# Initializing a list (1D):
<ident> = []

# Initializing a list (2D)
<ident> = [[]]

# e.g.
student_names: list[str]

# Python does not have CHAR!
tic_tac_toe: list[list[str]]

# Assign to a list
student_names[2] = "Marcos"

# You can even assign a whole list!
student_names = ["Tom", "James",
"Jimmy", "John", "Peter"]
```

[31] There are technically "Arrays" in Python, in that unlike Lists, which are dynamic and can have items added or removed dynamically with a changing length, Python does have single-type static arrays that do not have the dynamic functionality of lists in the array module, but they are extremely infrequently used due to their lack of features. Just use lists, I may refer to Python lists as arrays from this point forward.

```
// e.g.
StudentNames[3] // get 3rd student name
```

```
# Use a list
<ident>[<index1>][<index2>] # 2D list

# e.g.
student_names[3] # get 3rd student
                 # name
```

You can even store a **matrix** or a **2D array** of data; this is how you would represent a Tic-tac-toe[32] board:

IGCSE Pseudocode

```
// Adapted from the IGCSE Syllabus
DECLARE TicTacToe:ARRAY[1,3:1,3] OF
CHAR

// Assign to an ARRAY (1 dimensional)
TicTacToe[1,3] ← 'X'

// Use an ARRAY
<ident>[<index1>,<index2>] // 2D ARRAY

// e.g.
TicTacToe[2,1] // get the character at
               // 2, 1 on the Tic Tac
               // Toe board
```

Python

```
# Python does not have CHAR!
tic_tac_toe: list[list[str]]

# Assign to a list
tic_tac_toe[1][3] = "X"

# Use a list
<ident>[<index1>][<index2>] # 2D list

# e.g.
tic_tac_toe[2][1] # get the character
                  # at 2, 1 on the
                  # Tic Tac Toe board
```

[32] Noughts and crosses for those who prefer that.

Count-based Iteration (for)

Now we have the superpower of being able to store sequences of data; how do we work with them most effectively?

Puzzle piece one:

IGCSE Pseudocode	Python

`FOR <counter> ← <begin> TO <end>`
`    <code>`
`NEXT <counter>`
 	
`FOR <counter> ← <begin> TO <end> STEP <step>`
`    <code>`
`NEXT <counter>`
 	
`// e.g.`
`FOR Number ← 1 TO 30`
`    OUTPUT Number`
`NEXT Number`
 	
`FOR OddNumber ← 1 TO 30 STEP 2`
`    OUTPUT OddNumber`
`NEXT OddNumber`
	`for <counter> in range(<begin>, <end>):`
	`    <code>`
	
	`for <counter> in range(<begin>, <end>, <step>):`
	`    <code>`
	
	`# e.g.`
	`for number in range(1, 30):`
	`    print(number)`
	
	`for odd_number in range(1, 30, 2):`
	`    print(odd_number)`

For loops allow you to go through all the numbers between two values[33].

Puzzle Piece Two: In any programming language, you are provided with a way to get the number of items in a list. If my class has 5 students, and I store the names of the students in a list, I will simply get 5. Here's an example:

IGCSE Pseudocode	Python
`OUTPUT LENGTH(StudentNames)` `// gives me 5!`	`print(len(student_names))` `# gives me 5!`

Note that I use LENGTH in pseudocode as it was specified in the Strings section of pseudocode, and it is used to find the length of a sequence of characters. It also showed up in past-year papers, which is why I use it.

Puzzle Piece Three: We know that **arrays in Pseudocode start at 1, and arrays in Python start at 0**. Since we can reliably get the start and end values for any list, can we use a FOR loop to do operations on them?
Yes! This is one of the most common uses for for loops. Take a look:

[33] I am referring to the standard range-based for loops here; For-each loops go in the appendix.

IGCSE Pseudocode	Python

`FOR Counter ← 1 TO LENGTH(StudentNames)`
`    OUTPUT "There is a student called", StudentNames[Counter], " in the class."`
`NEXT Counter`
	`for counter in range(0, len(student_names)):`
	`    print("There is a student called ", student_names[counter], "in the class.")`

On the pseudocode side, we set the range to 1 till 5 (the length of the list), and for each number between them, **including 5**, it uses the array access syntax `StudentNames[Counter]` to grab the Counterth element of the list and prints it out.

On the Python side, we set the range from 0 till 5. Note that in Python, the range generated does not include the upper bound, so the numbers that will be given to you is actually 0, 1, 2, 3 and only up to 4. It also uses the array access syntax `student_names[counter]` to grab the Counterth element of the list.

We can even have a negative step to go backwards!

IGCSE Pseudocode	Python

`FOR Counter ← 1 TO LENGTH(StudentNames) STEP 1`
`    OUTPUT "There is a student called", StudentNames[Counter], " in the class."`
`NEXT Counter`
	`for counter in range(0, len(student_names), -1):`
	`    print("There is a student called ", student_names[counter], "in the class.")`

Procedures

We have passed the great filter of programming features; learning all the control flow features. Give at pat on your own shoulder for that![34]

Now we come to **procedures.** These are **reusable sections of code** that can be invoked from anywhere you like and have a set content. These were called **subroutines** a very long time ago (back when computers such as the Sinclair ZX-80 and XZ-81, etc. and Commodore computers were around), and are still sometimes referred to as **subprograms**, as just like a program, they are meant to reach an objective or sort a problem, just one smaller than the objective of the program surrounding it.

[34] Only if you read the whole programming guide so far. If you jumped here, Give a pat on your shoulder after you have understood everything so far 🙂

Consider the following case. You are the grade-level coordinator of your grade, and you need the average grade of 3 different classes. How are you going to total each of them?

You could do it like so:

IGCSE Pseudocode

```
// assume the students' arrays are already declared

DECLARE TotalA: INTEGER
TotalA <- 0
FOR Counter <- 1 TO LENGTH(ClassAGrades)
       TotalA <- TotalA + ClassAGrades[Counter]
NEXT Counter
OUTPUT "Class A's total is, ", TotalA

DECLARE TotalB: INTEGER
TotalB <- 0
FOR Counter <- 1 TO LENGTH(ClassBGrades)
       TotalB <- TotalB + ClassBGrades[Counter]
NEXT Counter
OUTPUT "Class B's total is, ", TotalB

DECLARE TotalC: INTEGER
TotalC <- 0
FOR Counter <- 1 TO LENGTH(ClassCGrades)
       TotalC <- TotalC + ClassCGrades[Counter]
NEXT Counter
OUTPUT "Class C's total is, ", TotalC

DECLARE TotalD: INTEGER
TotalD <- 0
FOR Counter <- 1 TO LENGTH(ClassDGrades)
       TotalD <- TotalD + ClassDGrades[Counter]
NEXT Counter
OUTPUT "Class D's total is, ", TotalD

DECLARE TotalE: INTEGER
TotalE <- 0
FOR Counter <- 1 TO LENGTH(ClassEGrades)
       TotalE <- TotalE + ClassEGrades[Counter]
NEXT Counter
OUTPUT "Class E's total is, ", TotalE


// do something else
```

However, this would be quite inefficient. Since each of these chunks all solve a smaller objective (total a list and print it out) anyway, we could use **procedures** to tidy the code up.

An example section of what procedures look like first[35] (standard form goes first as always):

IGCSE Pseudocode	Python

`// declaring procedures`
`PROCEDURE <name>`
`    <code>`
`ENDPROCEDURE`
 	
`PROCEDURE <name>(<parameter name>:`
`<type>, <parameter name>:<type>, ...)`
`    <code>`
`ENDPROCEDURE`
 	
`// e.g.`
`PROCEDURE SayHello`
`    OUTPUT "Hello!"`
`ENDPROCEDURE`
 	
`PROCEDURE Line(Size:INTEGER)`
`    FOR Length ← 1 TO Size`
`        OUTPUT '-'`
`    NEXT Length`
`ENDPROCEDURE`
	`# declaring procedures`
	`def <name>():`
	`    <code>`
	
	`def <name>(<parameter name>:<type>,`
	`<parameter name>:<type>, ...):`
	`    <code>`
	
	`# e.g.`
	`def say_hello():`
	`    print("Hello!")`
	
	`def line(size: int):`
	`    for length in range(1, size):`
	`        print('-')`

`// calling procedures`
`CALL <name>`
`CALL <name>(<parameter>, <parameter>...)`
 	
`// e.g.`
`CALL SayHello`
`CALL Line(10)`
	`# calling functions`
	`<name>()`
	`<name>(<parameter>, <parameter>...)`
	
	`# e.g.`
	`say_hello()`
	`line(10)`

As you can see, procedures can also take data **in** as **arguments**, or **parameters**. These all have an assigned type[36], and you can access them *just like a variable* from within the procedure's contents (this is called the **body**).

(Flip to the next page)

[35] **Very important:** Python does NOT differentiate between procedures and functions! The right portion of the table actually just shows standard Python functions that do not return, which is what procedures are. You will not find dedicated keywords for procedures in modern languages.
[36] They are optional in Python, but they are present for clarity's sake.

Here's the first example but written with procedures.

IGCSE Pseudocode

```
// assume the students' arrays are already declared

PROCEDURE TotalClass(ClassName: CHAR, Grades: ARRAY OF INTEGER)
    DECLARE Total: INTEGER
    Total <- 0
    FOR Counter <- 1 TO LENGTH(Grades)
        Total <- Total + Grades[Counter]
    NEXT Counter
    OUTPUT "Class ", ClassName, " total is, ", Total
ENDPROCEDURE

CALL TotalClass('A', ClassAGrades)
CALL TotalClass('B', ClassBGrades)
CALL TotalClass('C', ClassCGrades)
CALL TotalClass('D', ClassDGrades)
CALL TotalClass('E', ClassEGrades)
```

Functions

Imagine procedures, but they give you something back. This makes functions more like math functions, with inputs and outputs; take a look:

IGCSE Pseudocode

```
// declaring functions
FUNCTION <name> RETURNS <type>
    <code>
    RETURN <expr> // you MUST return
                  // something!
ENDFUNCTION

FUNCTION <name>(<parameter name>:
<type>, <parameter name>:<type>, ...)
RETURNS <type>
    <code>
    RETURN <expr> // you MUST return
                  // something!
ENDFUNCTION

// e.g.
FUNCTION GimmeFive RETURNS INTEGER
    RETURN 5
ENDFUNCTION

FUNCTION AddOne(Num:INTEGER) RETURNS
INTEGER
    DECLARE Result:INTEGER
    Result ← Num + 1
    RETURN Result
ENDFUNCTION
```

Python

```
# declaring functions
def <name>() -> <type>:
    <code>
    return <expr> # you MUST return
                  # something!

def <name>(<parameter name>:<type>,
<parameter name>:<type>, ...) -> <type>:
    <code>
    return <expr> # you MUST return
                  # something!

# e.g.
def gimme_five() -> int:
    return 5

def add_one(num: int) -> int:
    result: int
    result = num + 1
    return result
```

```
// calling functions
GimmeFive()
AddOne(5)

// ...or use them as expressions
AddOne(GimmeFive())
OUTPUT GimmeFive(), "+ 1 is",
AddOne(5)
```

```
# calling functions
gimme_five()
add_one(5)

# ...or use them as expressions
add_one(gimme_five())
print(gimme_five(), "+ 1 is",
add_one(5))
```

The key difference between procedures and functions are that procedures are **fruitless** (they do not produce outputs), but functions are **fruitful** (they produce outputs).

Adapting the totaling example from above:

IGCSE Pseudocode

```
// assume the students' arrays are already declared

FUNCTION TotalClass(ClassName: CHAR, Grades: ARRAY OF INTEGER) RETURNS INTEGER
    DECLARE Total: INTEGER
    Total <- 0
    FOR Counter <- 1 TO LENGTH(Grades)
        Total <- Total + Grades[Counter]
    NEXT Counter
    RETURN Total
ENDFUNCTION

OUTPUT "Class A's total is, ",  TotalClass('A', ClassAGrades)
OUTPUT "Class B's total is, ",  TotalClass('B', ClassBGrades)
OUTPUT "Class C's total is, ",  TotalClass('C', ClassCGrades)
OUTPUT "Class D's total is, ",  TotalClass('D', ClassDGrades)
OUTPUT "Class E's total is, ",  TotalClass('E', ClassEGrades)
```

Exercise: Adapt the Pseudocode to Python. Answers are on the next page:

Python

```
def total_class(class_name: str, grades: list[int]) -> int:
    total: int = 0
    for counter in range(0, len(students)):
       total += students[counter]
    return total

print("Class A's total is, ", total_class('A', class_a_grades))
print("Class B's total is, ", total_class('B', class_b_grades))
print("Class C's total is, ", total_class('C', class_c_grades))
print("Class D's total is, ", total_class('D', class_d_grades))
print("Class E's total is, ", total_class('E', class_e_grades))
```

String Manipulation

We are mostly **done** with the fundamental programming concepts! Congratulate yourself even more if you were able to push through and understand the content.

These are strings.

IGCSE Pseudocode	Python
`"Good Morning"` `"Irene Wong"`	`"Good Morning"` `"Irene Wong"`

Useful, right? **In pseudocode, they begin at 1**, and in Python, they begin at **0**.

We can grab the length of the string just like how you would a list:

IGCSE Pseudocode	Python

`LENGTH(<string>)`
 	
`// e.g.`
`LENGTH("Hello") // gives you 5`
	`len(<string>)`
	
	`# e.g.`
	`len("Hello") # gives you 5`

To convert strings into either uppercase or lowercase:

IGCSE Pseudocode	Python

`UCASE(<string>)`
`LCASE(<string>)`
 	
`// e.g.`
`UCASE("Hello") // gives you “HELLO”`
`LCASE("ANGER") // gives you “anger”`
 	
`// Given a variable LastName,`
`UCASE(LastName)`
`// returns the uppercase of LastName`
	`<string>.upper()`
	`<string>.lower()`
	
	`# e.g.`
	`"Hello".upper() # gives you “HELLO”`
	`"ANGER".lower() # gives you “anger”`
	
	`# Given a variable last_name,`
	`last_name.upper()`
	`# returns the uppercase of last_name`

You can also slice only a portion of a string, this is called taking a **substring**. Note that in Python, strings begin at 0. The end value for a substring operation in Python **also means that the operation ends before** the position, i.e. if you substring a string between 0 and 4 (in “hello”, h is 0, e is 1, l is 2, etc.) it will only take characters 0, 1, 2, 3 (“hell”).

IGCSE Pseudocode	Python
`SUBSTRING(<string>, <begin>, <length>)`	`<string>[<begin>:<end>]`

`// e.g.`
`SUBSTRING("Hello, Thomas!", 1, 5)`
`// gives you “Hello”`
	`# e.g.`
	`"Hello, Thomas!"[0:4]`
	`// gives you “Hell” (wrong!)`
	`"Hello, Thomas!"[0:5]`
	`// gives you “Hello”`

File I/O

The last part, working with files. **Just as a refresher,** files are pieces of data with a name stored on your hard drive. This includes text files, Microsoft Office/Apple iWork[37]/LibreOffice[38] files, your apps, plain text files, and images.

We can create files in code as well; simple text files that we can read and write strings to.

Here’s the general process to using files in a programming language:

- You must **open** the file to tell the operating system (computer for the laymen)[39] that you want to work with the file.
    - You can ask to either read the file, write the file, or both
- You can then **write** or **read** content from the file, meaning changing it or viewing what’s inside.
    - You do this by using variables to represent the file’s content in memory.
- You must **close** the file to tell the operating system and programming language that you are done.
    - This writes any residual data that has only been partially written to the file fully.
    - If you don’t close a file, you may get resource leaks and a dangling file pointer[40].

As always, standard form goes first:

[37] Pages, Keynote, etc.
[38] Not just base!
[39] The base layer upon which all your programs are on, all your hardware is managed, and where all your file management happens, like Windows, macOS, Linux, BSD, and even Android and iOS.
[40] Basically a number that represents the location of the file on disk.

IGCSE Pseudocode	Python

`// file modes include READ and WRITE`
`//`
`// opening files`
`OPENFILE <file name> FOR <file mode>`
 	
`// reading files (read into <variable>)`
`READFILE <file name>, <variable>`
 	
`// writing files (write from <variable>)`
`WRITEFILE <file name>, <variable>`
 	
`// closing files`
`CLOSEFILE <file name>`
 	
`// e.g.`
`OPENFILE data.txt FOR READ AND WRITE`
`READFILE data.txt, Content`
`WRITEFILE data.txt, Content + "Hi!"`
`CLOSEFILE data.txt`
	`# READ corresponds to 'r'`
	`# WRITE corresponds to 'w'`
	`# READ AND WRITE corresponds to 'r+'`
	`# or 'w+'`
	`# opening files`
	`<ident> = open(<file name>, <file mode>)`
	
	`# reading files`
	`<variable> = <ident>.read()`
	
	`# writing files`
	`<ident>.write(<variable>)`
	
	`# closing files`
	`<ident>.close()`
	
	`# e.g.`
	`file = open("data.txt", "r+")`
	`content = file.read()`
	`file.write(content + "Hi!")`
	`file.close()`

Appendix

You may find extra information and content here. Note that everything but exercises are not required for your exam

Appendix One (Practice Projects)

1 — Attendance Program (Iteration, Branching)

Consider writing a program for a small classroom that checks the attendance of each student, where there are 5 of them. You would need to store the name of each student in some sequence, how would we do that?

We assume that it must store the attendance of each student in a separate list, and print each of them out at the end in the following format:

```
John is Here
Charles is not Here
Thomas is Here
Peter is Here
William is Here
```

Etc. We will also assume that an array `StudentNames` is already populated with the names of the students.

2 — Class Totaling Program (Iteration and Totaling, Procedures)

Given a procedure that totals the grades of all the students in a class,

IGCSE Pseudocode

```
// assume the students' arrays are already declared

PROCEDURE TotalClass(ClassName: CHAR, Grades: ARRAY OF INTEGER)
    DECLARE Total: INTEGER
    Total <- 0
    FOR Counter <- 1 TO LENGTH(Grades)
        Total <- Total + Grades[Counter]
    NEXT Counter
    OUTPUT "Class ", ClassName, " total is, ", Total
ENDPROCEDURE

CALL TotalClass('A', ClassAGrades)
CALL TotalClass('B', ClassBGrades)
CALL TotalClass('C', ClassCGrades)
CALL TotalClass('D', ClassDGrades)
CALL TotalClass('E', ClassEGrades)
```

Rewrite the example code above in Python. Hint, declaring the procedure looks something like this:

Python

```
def total_class(class_name: str, grades: list[int]):
   # Write your code here!
```

3 – Password Creation System (String Manipulation, Iteration, Branching)

Create a system to let users create passwords. You need to make sure that **their password is more than 10 characters long.** If the user's password does not meet this requirement, keep entering the password until it does meet the requirement. Make use of **repeat-until** loops and some **conditional branching** to notify the user if they entered a too insecure password.

Appendix Two (Sample Project Implementations)

1 — Attendance Program

IGCSE Pseudocode

```
DECLARE StudentNames: ARRAY[1:5] OF STRING
DECLARE StudentAttendance: ARRAY[1:5] OF BOOLEAN
DECLARE CurrentAttendance: STRING

FOR Counter <- 1 TO LENGTH(StudentNames)
    OUTPUT "Is ", StudentNames[Counter], " Here?"
    INPUT CurrentAttendance

    IF CurrentAttendance = "Here"
      THEN
        StudentAttendance[Counter] <- TRUE
    ELSE
      StudentAttendance[Counter] <- FALSE
    ENDIF
NEXT Counter

FOR Counter <- 1 TO LENGTH(StudentNames)
    IF StudentAttendance[Counter] = TRUE
      THEN
        OUTPUT StudentNames[Counter], " is Here"
    ELSE
      OUTPUT StudentNames[Counter], " is not Here"
    ENDIF
NEXT Counter
```

Python

```
# create an empty list of 5 bools.
# note that you cannot index into any index of the list and use it,
# you must create it manually first. I use this multiplication syntax
# as a shorthand, you only need to understand how the algorithm works.
student_names: list[str]
student_attendance: list[bool] = 5 * [bool()]
current_attendance: str = ""

for counter in range(0, len(student_names)):
    # You can only have one string as an argument, so just
    # use + for now
    current_attendance = input("Is " + student_names[counter] + " Here?")

    if current_attendance == "Here":
       student_attendance[counter] = True
    else:
       student_attendance[counter] = False

for counter in range(0, len(student_names)):
    # Python does not need you to use == True or == false if the value is already
    # a boolean. The below is equivalent to
    # student_attendance[counter] == True
    #
```

```
    # If you wanted to negate it, you could write:
    # if not student_attendance[counter] == True
    #
    if student_attendance[counter]:
        print(student_names[counter], " is here")
    else:
        print(student_names[counter], " is not here")
```

To break down the code, we:

- Declare 2 lists, the one provided to us and the list of booleans to store attendance
- Declare a variable to store the user input for the current student's attendance.
- For every index in the list, student_names,
    - Ask the user for the student's attendance
    - Save True to the student_attendance list at the same index if the user typed "Here", else save False. Saving the same index for the name and the attendance allows us to have a pair of data bound by the index.
- Loop again when the first one is done
    - Print <student name> is here if True was saved at that index, else print false.

2 — Class Totaling Program

Python

```
def total_class(class_name: str, grades: list[int]):
    total: int = 0
    for counter in range(0, len(students)):
       total += students[counter]
    print("Class ", class_name, " total is, ", total)

total_class('A', class_a_grades)
total_class('B', class_b_grades)
total_class('C', class_c_grades)
total_class('D', class_d_grades)
total_class('E', class_e_grades)
```

3 — Password Creation System

IGCSE Pseudocode

```
// declare the variable for later use
DECLARE UserPassword: STRING

// Our post-condition loop, this ensures that the user enters the password at least
// once
REPEAT
    OUTPUT "Choose a password above 10 characters"
    INPUT UserPassword

    // somehow telling the user their password is invalid is better than repeating
    // the prompt
    IF LENGTH(UserPassword) < 10
      THEN
        OUTPUT "Your password is too short!"
    ENDIF
// repeat until this condition is met, i.e. keep going while this is false
UNTIL LENGTH(UserPassword) >= 10
```

Python

```
# Note that since we are using a pre-condition loop, we must initialize the
# variable to something first to let the condition even evaluate.
#
# If you don't initialize the variable, it will simply throw an error.
user_password: str = ""

# Here, the condition will also run at least once as user_password is always going
# to be 0 characters long as we made it that way
while len(user_password) < 10:
    user_password = input("Choose a password above 10 characters")

    # somehow telling the user their password is invalid is better than repeating
    # the prompt
    if len(user_password) < 10:
        print("Your password is too short!")
```

Appendix Three (Extra Terminology)

Here are some key compiler engineering terms[41]

- Identifier (variable name)
- Literal (a value)
- Binary Operation (two values with an operator)
- Unary Operation (one value with an operator)

Appendix Four (Expressions in Python)

Here is the exhaustive list of what can be expressions in Python, note that you **definitely do not need all of them for your exam**:

- Identifiers by themselves
- Literals by themselves (includes containers, listed below)
    - Lists
    - Tuples
    - Dictionaries
    - Sets
- Math expressions
- Comparison Expressions
- Logical Expressions
- **Bitwise Operations** (You have covered these in chapters 1 and 10!)
    - Bitwise AND, OR, NOT, XOR, NOR, NAND
    - Bit shifts (left and rights)
- Membership Operators
    - In and not in; If fruit were a string, and orchid were a list of strings, you could run `fruit in orchid` instead of a linear search, but not recommended for your exam.
- Fruitful Function Calls (functions that return)
- Comprehensions (expressions that run loops to generate the following)
    - Lists
    - Tuples
    - Dictionaries
    - Sets
- Ternary Operations
    - `x if condition else y`

[41] That's what I, Eason Qin specialize in.

- Walrus Operator Results (assigns value to variable and also evaluates to an expression)
- Array Indexes and slices
- Generators
- Type Casts
- Context managers
    - `with open('file.txt') as f: f.read()`
- F-strings
- Class Constructors and method calls

Appendix Five (Using this guide effectively)

In order to make the most out of this guide and your existing revision resources, I recommend the following:

- Revisit your own notes to go over the links in your brain you have made yourself. This is actually very helpful!
- Read the revision guide from cover to cover, to deepen your understanding, think of this guide just being a large collection of detailed notes.
    - You should be able to use a highlighter on this guide!
    - You can also make notes off of this guide
    - **Note that this guide is not comprehensive, for the last time; please do not rely on this as your ONLY source of revision.**
- ***USE THE TEXTBOOK!*** It is still a great resource. Check the table of contents for the things that you must learn, make sure you use the textbook to reinforce your learning.
    - Also make sure to do as many exercises in the textbook as you need. **Unfortunately, we are learning computer science in an academic context,** so you must do things the academic way, i.e. research projects, a very static, concrete and somewhat outdated syllabus, and **exams**.
    - The textbook also has a lot of diagrams and tables, which is what this doesn't have.
- When you need help, just consult the guide; check the table of contents for what you need, and go over what you highlighted or read or took notes on.

Appendix Six (Digital Copies)

If you have the printed copy, you may find the digital copy here:

It is hosted on my website (ezntek.com) **and is 100% safe!**

Alternatively, visit this hyperlink or visit the following weblink:

https://ezntek.com/revision/csrg_g1_2024.html

To get a constantly updated version of the CSRG.

Sorry that the PDF does not have proper bookmarks, blame Google Docs! The developers are so incompetent that they can't even put that tiny bit more effort into adding bookmarks based on headings. I have no way of doing it. Sorry, but not my bad!

Please read my blog for more information, at ezntek.com.