Programmatic thinking

Algorithms and operators

What is programmatic thinking?

How do we *actually* solve real-world **problems** involving data?

We **break down** the problem or task into smaller more manageable parts that we can **solve** in a **systematic** and **logical** way.

This is called **PROGRAMMATIC THINKING.**

What is programmatic thinking?

> Programmatic thinking helps us to **logically organise information** into a **series of instructions** that a **computer can execute.**

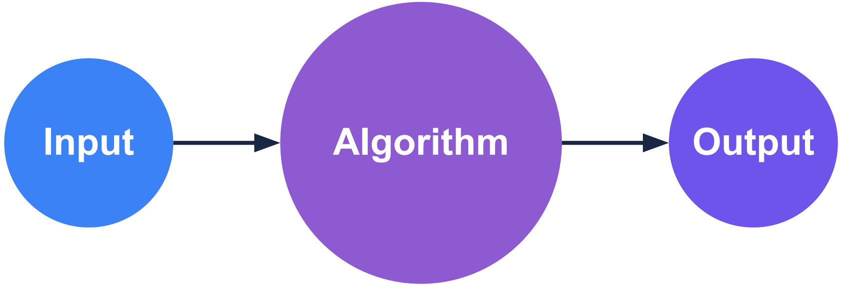
We need certain **"tools"** in order to apply programmatic thinking:

Algorithms	Operators	Flowcharts	Pseudocode	Conditional statements
The **sequence of steps** required to solve the problem.	Used to **compare** numbers, strings, and statements.	**Visual representations** of the flow of control of conditional statements.	**Descriptions** of the sequence of steps and actions in **plain natural language**.	Representation of decision-making through a set of **specific conditions**.

Algorithms

> An algorithm is a **specific procedure** to solve a problem in terms of the **required actions or steps** and the **order** in which these actions are executed.

An algorithm can be as simple as calculating the sum of a list of numbers to searching for the occurrence of a specific value in a list of numbers.



Set of **actions** to obtain the expected output from the given input

Algorithms allow us to create **repeatable** solutions to problems.

They are designed to produce a correct result every time they are used, as long as they are given an appropriate input.

Algorithms

> Algorithms can be categorised into different techniques, often to solve the same problem using **different approaches**. The algorithm efficiency will depend on the approach and the specific problem.

Brute force

The simplest algorithm where every possible alternative solution to a problem is tried until the correct solution is found.

Backtracking

Different possible solutions are tried systematically until the correct one is found, most often starting from an initial guess to the solution.

Divide and conquer

Divides the problem into smaller subproblems, each subproblem is solved, and the solutions are combined to solve the original problem.

Greedy

Makes optimal choices at each step in the algorithm considering only the current step and situation.

Recursive

Calls itself on smaller subproblems that are similar to the original problem until the problem becomes small enough to be solved directly.

Dynamic programming

Breaks down complex problems into smaller subproblems, storing solutions to the subproblems, and uses the stored solutions to solve the original problem.

Algorithms

An example of an algorithm to search for the occurrence of a specific value (x) in a list of numbers:

1. Sort the list of numbers in ascending order (an algorithm in itself).
2. Find the midpoint of the sorted list.
3. Compare the midpoint value to the value of interest, x.
4. If the midpoint value is larger than the value of interest, x, repeat steps 2 to 4 only on the list of numbers smaller than the midpoint value.
5. If the midpoint is smaller than the value of interest, x, repeat steps 2 to 5 only on the list of numbers that are larger than the midpoint value.
6. Repeat steps 2 to 5 until the midpoint value is equal to the value of interest, x.

Algorithms

An example of an algorithm to search for the occurrence of a specific value (x) in a list of numbers:

1. Sort the list of numbers in ascending order (an algorithm in itself).
2. Find the midpoint of the sorted list.
3. Compare the midpoint value to the value of interest, x.
4. If the midpoint value is larger than the value of interest, x, repeat steps 2 to 4 only on the list of numbers larger than the midpoint value.
5. If the midpoint is smaller than the value of interest, x, repeat steps 2 to 5 only on the list of numbers that are smaller than the midpoint value.
6. Repeat steps 2 to 5 until the midpoint value is equal to the value of interest, x.

This is called the **binary search algorithm**. It is both a divide and conquer and recursive algorithm.

We notice words related to **comparisons**.

But how do we “explain” to a computer how to perform these comparisons?

We use
COMPARISON OPERATORS
(==, !=, >, >=, <, <=)

Programmatic thinking

Algorithms

Consider the simple example of an algorithm that determines whether a number is between 0 and 10 (inclusive):

1. Set the value of interest as x.
2. If x is greater than or equal to zero and x is less than or equal to 10, then the statement is True.
3. Otherwise, the statement is False.

This algorithm includes **COMPARISON OPERATORS.**

We also see a **BOOLEAN OPERATOR*** that combines the two statements.

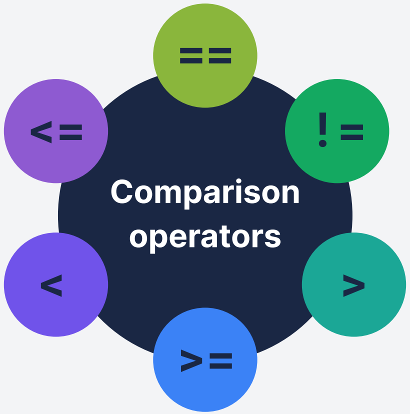
We see that **operators are an important tool** that algorithms use to make decisions.

*A boolean operator is also known as a logical operator.

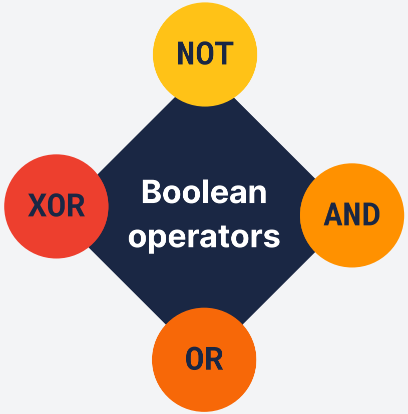
Programmatic thinking

Operators

> We use operators in conditional statements, flowcharts, and pseudocode to **compare** numbers, strings, and statements, and combine or exclude statements.



Used to **compare numbers** or **strings** to perform the evaluation within a boolean expression*.



Used as conjunctions to **combine** (or **exclude**) **statements** in a boolean expression.

*A boolean expression is a statement that results in an answer of either True or False.

Comparison operators

Used to **compare numbers** or **strings** to perform the evaluation within a boolean expression.

Equal to	`==`	Returns **`True`** when the value on the left is **strictly equal** to the value on the right.	`2 + 2 == 4 → True` `2 + 3 == 4 → False`
Not equal to	`!=`	Returns **`True`** when the value on the left is **strictly not equal to** the value on the right.	`2 + 2 != 4 → False` `2 + 3 != 4 → True`

Programmatic thinking

Comparison operators

Name	Operator	Description	Examples

Greater than	`>`	Returns **True** when the value on the left is **greater than** the value on the right.	`5 > 4 → True`
			`4 > 5 → False`
			`3 + 1 > 4 → False`

Greater than or equal to	`>=`	Returns **True** when the value on the left is either **greater than or equal to** the value on the right.	`5 >= 4 → True`
			`4 >= 5 → False`
			`3 + 1 >= 4 → True`

Less than	`<`	Returns **True** when the value on the left is **smaller than** the value on the right.	`5 < 4 → False`
			`4 < 5 → True`
			`3 + 1 < 4 → False`

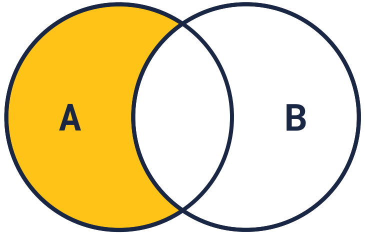
Less than or equal to	`<=`	Returns **True** when the value on the left is either **smaller than or equal to** the value on the right.	`5 <= 4 → False`
			`4 <= 5 → True`
			`3 + 1 <= 4 → True`

Programmatic thinking

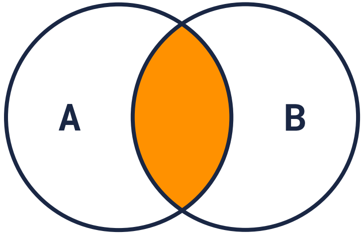
Boolean operators

Used as conjunctions to **combine** (or **exclude**) **statements** in a boolean expression.

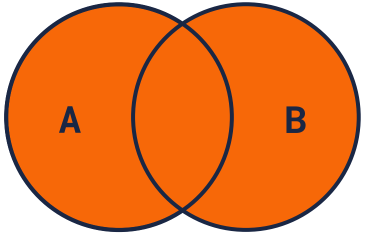
We can visually represent the logic of boolean operators:



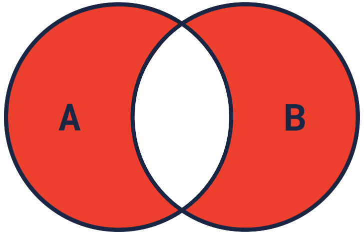
A NOT B



A AND B



A OR B



A XOR B

Programmatic thinking

Boolean operators

```
NOT False → True
NOT True → False
```

NOT

Only considers the argument **after** the operator. The algorithm considers the **opposite** of the argument.

```
2 + 2 == 4 AND 1 + 3 == 4 → True
2 + 2 == 4 AND 4 > 5 → False
4 > 5 AND 2 + 3 == 4 → False
```

AND

The two arguments are considered in conjunction.

Both arguments should be **True** for the statement to be **True**.

```
2 + 2 == 4 OR 1 + 3 == 4 → True
2 + 2 == 4 OR 4 > 5 → True
4 > 5 OR 2 + 3 == 4 → False
```

OR

If **either** of the arguments to the left or right of the operator is **True**, the statement will be **True**. If both arguments are **True**, the statement will also be **True**.

```
2 + 2 == 4 XOR 1 + 3 == 4 → False
2 + 2 == 4 XOR 4 > 5 → True
4 > 5 XOR 2 + 3 == 4 → False
```

XOR

Exclusive OR

Only if **exactly one** of the arguments to the left or right of the operator is **True** will the statement be **True**. If both arguments are **True**, the statement is **False**.