CAIE Computer Science IGCSE
8 - Programming
Advanced Notes

8.1 Programming concepts

Variables and constants are ways of storing data – however, the data stored in a variable may change throughout the execution of a program, whilst the data stored in a constant must stay the same.

Statements	Description	Examples
Variable Declaration	Creates a variable to store data.	Example: `name = "Alex"`
Constant Declaration	A value that does not change while the program runs. Often given fully uppercase identifiers.	Example: `PI = 3.14`
Assignment	Setting or updating a value in a variable.	Example: `score = score + 10`

What Are Data Types?

In programming, a **data type** defines the kind of data a variable or constant can hold. It tells the program how the data will be **stored, processed, and displayed**.

Common Data Types

Term	Description	Examples
Integer (int)	Whole numbers only, no decimals	`5`, `-20`, `0`
Real (float)	Numbers that include a fractional/decimal part. Also called float in some languages	`3.14`, `-0.5`, `99.99`
Boolean (bool)	Often used for conditions and logic, only has two states	`True` or `False`
Character (char)	A single symbol or letter, enclosed in single quotes for most programming languages	`'A'`, `'a'`, `'#'`
String (str)	A sequence of characters, enclosed in double quotes for most programming languages	`"Hello"`, `"123"`, `"£$%"`

What Is Input/Output in Programming?

Input/Output (I/O) refers to how a program interacts with the outside world — specifically how it:

- Receives data from the user (input)
- Displays data or information to the user (output)

Input (Getting Data from the User)

Used to collect data that a program requires to process. Typically stored in a variable after being entered.

Example (Pseudocode):

```
pgsql

name ← INPUT("Enter your name: ")
```

Example (Python):

```
python

name = input("Enter your name: ")
```

Output (Displaying Data)

Used to show messages, results, or prompts to the user.

Example (Pseudocode):

```scss
OUTPUT("Your score is " + score)
```

Example (Python):

```python
print("Your score is", score)
```

Notes:

- Outputs can display text, numbers, or variable values.
- Inputs are usually strings by default and may need type conversion (e.g., `int(input(...))` in Python).

Input/output operations are often used with selection and iteration, such as branching depending on the data entered or prompting repeatedly until valid data is entered.

What Is Structured Programming?

Structured programming is a method of writing clear, modular, and easy-to-understand code using three core principles:

1. Sequence – instructions are executed in the order they were written
2. Selection – decisions (`IF-ELSE, CASE`)
3. Iteration – repetition (`WHILE`, `FOR`)

Selection Statements

Selection statements are used to determine program flow, i.e. making decisions in a program, usually done using if–else statements or case statements.

- **If–Else Statement:**

```
height = 120

if height >= 150:
      print(“You can ride the rollercoaster”)
else:
      print(“You are too short to ride”)
```

- **Case Statement:**

```
day = "Monday"

match day:
    case "Monday":
        print("Start of the week")
    case "Friday":
        print("End of the week")
    case _:
        print("A regular day")
```

Note: case_ : is used to represent anything other than the previously stated cases. In this example, it represents any input other than Monday or Friday.

Iteration Statements

Iteration is used to repeat a block of instructions within a program, this can either be:

- **Count-controlled** (for loop) to repeat a certain number of times:

  ```
  for i in range(10):
        print("Hello World")
  ```

- **Pre-condition** (while loop) to repeat until a condition is met by checking the condition before the loop

  ```
  num = 0

  while num <= 5:
        print(num)
        num = num + 1
  ```

- **Post-condition** (do…while loop) to repeat until a condition is met by checking the condition at the end of the loop

  ```
  num = 0

  while True:
        print(num)
        num = num + 1
        if number > 5: # Condition is checked at the end
              break
  ```

Totalling and Counting

Totalling means to keep a running total (a sum) of numbers as they are inputted or processed. This is usually done using iteration to add numbers as you go along.

Example:

```
sum = 0
for i in range(5):
      sum = sum + 2

print(“The total is: “ + sum)
```

Counting is keeping track of how many times an event has occurred, such as counting how many times a specific item is found in an array.

Example:

```
colours = [“blue”, “red”, “red”, “yellow”, “red”]
count = 0

for i in range(len(colours)):
      if colours[i] == “red”: # checking if red is in the array
            count = count + 1

print(count)
```

What Is String Handling?

String handling refers to the operations you can perform on strings (text data). Such as measuring length or extracting substrings.

A string is a sequence of characters, e.g. `"hello123!"`

Key String Operations

Operation	Description	Example
`length`	Returns the number of characters in a string	`length("hello") → 5`
`substring`	Extracts a sequence of characters within a string	`substring("computer", 0, 2) → "com"`
`upper`	Converts a string to uppercase	`upper(“hello”) → “HELLO”`
`lower`	Converts a string to lowercase	`lower(“Milly”) → “milly”`

What Are Arithmetic Operations?

Arithmetic operations are the basic mathematical calculations that can be performed in a programming language. These are essential for processing numerical data in algorithms and programs.

Standard Arithmetic Operators

Operation	Symbol	Example	Result
Addition	`+`	`3 + 2`	`5`
Subtraction	`-`	`7 - 4`	`3`
Multiplication	`*`	`5 * 3`	`15`
Real Division	`/`	`10 / 4`	`2.5`
Exponentiation	`^`	`3 ^ 2`	`9`
Modulo	`%`	`10 % 3`	`1`
Floor Division	`//`	`7 // 3`	`2`

What Are Relational Operations?

Relational operations are used to compare two values. They return a Boolean value:

- `True` if the comparison is correct
- `False` if it is not

These operations are commonly used in conditions, such as `IF` statements and loops.

Relational Operators

Operation	Symbol in most languages	Example	Result
Equal to	`==`	`5 == 5`	`True`
Not equal to	`!=` or `<>`	`3 != 4`	`True`
Less than	`<`	`2 < 5`	`True`
Greater than	`>`	`6 > 7`	`False`
Less than or equal to	`<=`	`5 <= 5`	`True`
Greater than or equal to	`>=`	`7 >= 10`	`False`

What Are Logical Operators?

Logical operators use Boolean values (`True` or `False`). They are used in conditions to control the flow of programs.

Logical Operators Explained

Operator	Description	Example	Result
`NOT`	Reverses the Boolean value	`NOT True`	`False`
`AND`	Returns `True` if both input conditions are true	`True AND True`	`True`
`OR`	Returns `True` if either input condition is true	`True OR False`	`True`

Nested Statements

Selection and iteration statements can be nested (indented) within each other to provide more complex decision-making within a program. However, nested iteration does massively increase the complexity of a program.

```
age = 20
has_id = True

if age >= 18:                 # Outer if
    if has_id:                # Nested if
        print("You are allowed to enter.")
    else:
        print("You need an ID to enter.")
else:
    print("You are not old enough to enter.")
```

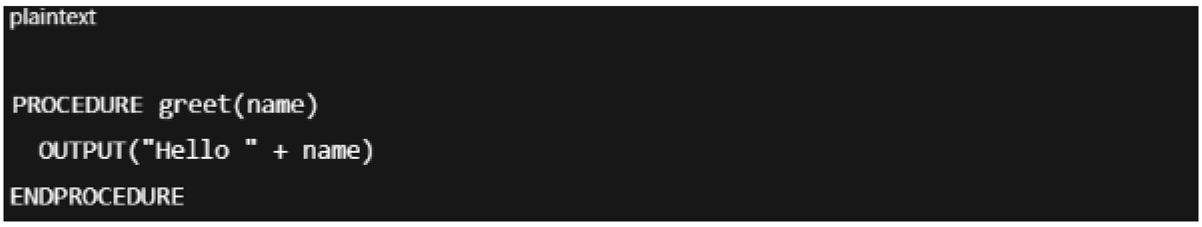
What Is a Subroutine?

A **subroutine** is a block of code that performs a **specific task** and can be **reused** by calling it by name at any point in a program.

Types:

- **Procedure**: performs an action and **does not return a value**
- **Function**: performs an action and **returns a value**

Example (Pseudocode):



```
PROCEDURE greet(name)
  OUTPUT("Hello " + name)
ENDPROCEDURE
```

Parameters and Return Values

- **Parameters** allow data to be passed into a subroutine
- **Return values** allow data to be passed back to the main program
    - The returned value should be assigned to a variable in the main program

Library Functions

Libraries can be used to provide an extensive set of tools to be used within your programs, such as creating random numbers.

- **MOD** – Finds the remainder after a division:

  17 MOD 5 = 2

- **DIV** – Finds only the whole number after a division:

  20 DIV 3 = 6

- **ROUND** – Rounds a decimal number to the nearest integer, or a given number of decimal places:

  ROUND(3.9) = 4

  ROUND(5.02) = 5

  ROUND(3.14159, 2) = 3.14

- **RANDOM** – Generates a random number

  RANDOM(1, 6) → any number between 1 and 6 (inclusive)

Creating Maintainable Programs

It is important to create programs in a clear and structured way so that they can be maintained and improved upon in the future. Programmers can revisit their code and understand it immediately and make any changes necessary.

- **Meaningful identifier names**

  Using clear, descriptive names for variables, constants, and subroutines helps to improve the readability and understanding of what the code does.

  Example: Use `totalMarks` instead of `x`.

- **Comments**

  Comments can help programmers understand the purpose of a section of code and give insight into the original developer’s intentions

- **Procedures / Functions (Subroutines)**

  Can be called many times without having to repeat code. This makes testing and their use easier as they only have to be debugged once.

8.2 Arrays

What is an Array?

- A **collection of similar data items** (elements) stored under a **single name**.
- Each item is accessed using an **index** (position number).

Characteristics:

- Items must be of the **same data type**.
- Indexing usually starts at **0**, but can also start at **1**.

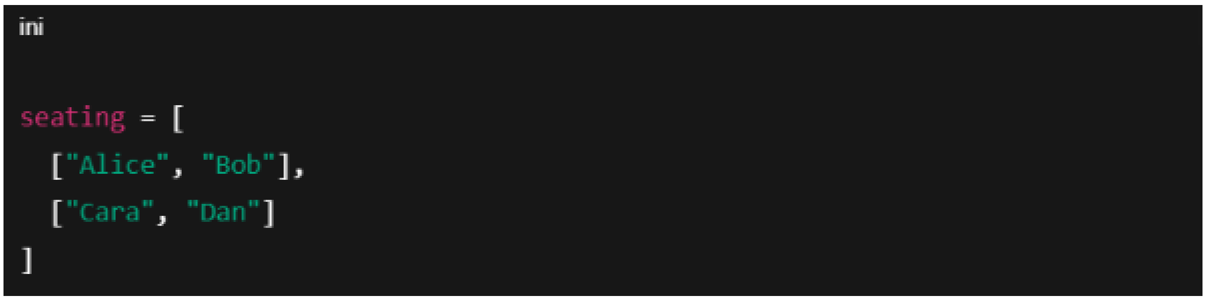
One-Dimensional Array (1D)

- A **single list** of items.
- Example:

```
scores = [10, 20, 30]
scores[1] → 20     # 0-based indexing
```

Two-Dimensional Array (2D)

- An array of arrays (like a **table or grid**).
- Example:



```ini
seating = [
  ["Alice", "Bob"],
  ["Cara", "Dan"]
]
```

```
seating[1][0] → "Cara"     # 0-based indexing
```

Iterating through Arrays

The data in an array can be read through using iteration, as well as writing values to an array using iteration.

- **Reading from an array:**

```
colours = [“blue”, “red”, “red”, “yellow”, “red”]

for item in colours:
      print(item)
```

This would print each item in order.

- **Writing to an array:**

```
evenNumbers = []

for x in range(1, 10):
      evenNumbers.append(x * 2)     # [2, 4, ...]
```

For a 2D array, it is often beneficial to use nested iteration to read from it. This is because the first level of iteration can be used to iterate through the sub-arrays; the second, nested iteration can then be used to iterate through each element of the sub-arrays.

An example using the array seating, shown above:

```
seating =  [

           [“Alice”, “Bob”],

           [“Cara”, “Dan”]

           ]

for x in range(0,1):

      for y in range(0,1):

           print(seating[x][y])
```

The code above would print Alice, Bob, Cara, Dan in the order given.

File Handling

File handling is useful as you can permanently save data used in a program, as well as read external data. A file handling variable is required to access the file itself, and the file must be opened in the desired mode, i.e. “write” mode or “read” mode.

Files should be closed after they have been accessed, this ensures that any changes made to the file are committed to memory and saved.

- **Reading a line from a file:**

```
myFile = open(“hello.txt”, “r”) # r is read mode

print(myFile.readline())

myFile.close()
```

- **Writing a line to a file:**

```
myFile = open(“hello.txt”, “w”) # w is read mode

myFile.write(“Hello World”)

myFile.close()
```