Programming Concepts, Arrays, File Handling

UNIT: 8.1, 8.2 & 8.3

Topic Objectives:

8.1 Programming Concepts

- Declare and use variables and constants
- Understand and use the basic data types
- Understand and use input and output
- Understand and use the concept of sequence
- Understand and use the concept of selection
- Understand and use the concept of iteration
- Understand and use the concepts of totalling and counting
- Understand and use the concept of string handling
- Understand and use arithmetic, logical and Boolean operators
- Understand and use nested statements
- Understand what is meant by procedures, functions and parameters
- Define and use procedures and functions, with or without parameters
- Understand and use local and global variables
- Understand and use library routines
- Understand how to create a maintainable program

8. 2 Arrays

- Declare and use one-dimensional (1D) and two-dimensional (2D) arrays
- Understand the use of arrays
- Write values into and read values from an array using iteration

8. 3 File Handling

- Understand the purpose of storing data in a file to be used by a program
- Open, close and use a file for reading and writing

BASIC DATA TYPES (Tells what type of data is stored or returned)		
Datatype	**Description**	**Example**
INTEGER **(int in python)**	➢ To represent a whole number. ➢ Includes positive and negative numbers ➢ Can be used for calculations	**75** **-120**
REAL or FLOAT **(float in python)**	➢ Number with fractional part. ➢ Includes positive and negative numbers ➢ Can be used for calculations	**12.56** **-9.235**
CHAR **(char in python)**	➢ A single character ➢ Cannot be used in calculation	**'+', 'B', '4' etc.**
STRING **(str in python)**	➢ A sequence of zero or more characters ➢ Cannot be used in calculation	**"Hello World", "Sum@123", " " (Empty String)**
BOOLEAN **(bool in python)**	The logical values TRUE and FALSE. A datatype with two values either TRUE or FALSE	**TRUE** **FALSE**

IDENTIFIERS

- **Identifiers are the names given by a user or a programmer to store values during program development.**
- **Identifiers include variables, constants, arrays, functions and procedures.**

IDENTIFIER NAMING RULES

- Can only contain letters (A–Z, a–z) and digits (0–9).
- Must start with a capital letter and not a digit.
- Accented letters and other characters, including the underscore, should not be used.
- Single letters may be used where these are conventional (such as i and j when dealing with array indices, or X and Y when dealing with coordinates) as these are made clear by the convention.
- Keywords should never be used as identifier names.
- Identifiers should be considered case sensitive, for example, Countdown and CountDown should not be used as separate variables

	1. Variable	2. Constants
Definition	A named data store that contains a value that **may change** during the execution of a program.	A named data store that contains a value that **does not change** during the execution of a program
Pseudocode for declaration	**DECLARE variablename:DATATYPE**	**CONSTANT constantname ← value**
Flowchart Representation	**DECLARE variablename:DATATYPE**	**CONSTANT constantname ← value**
Example (Pseudocode)	**DECLARE Counter: INTEGER** ***(Counter is the variable name and is used to store only integers)***	**CONSTANT PI ← 3.14** ***(PI is the constant name and 3.14 is the value stored in it)***

Python does not require any separate declarations and does not differentiate between constants and variables.

3. ARRAYS – A data structure to store a group of values of same datatype under a single name

- *The **dimension** is the number of indexes required to access an element.*
- *The **index** is the position of the element in an array.*
- *For example A[25] is the 25th element of a one-dimensional array.*
- *For example Mark[3,7] is the element in the 3rd row and 7th column.*

	One Dimensional Array (1D Array)	Two Dimensional Array (2D Array)
Definition	A collection of items of the same data type. Array elements are accessed using array indices/index	A collection of data elements arranged in a grid-like structure with rows and columns. Each element in the array is referred to as a cell and can be accessed by its row and column indices/indexes.
Pseudocode for declaration	`DECLARE arrayname: ARRAY[l:u] OF DATATYPE` ***(where l is the lower and u is the upper index)***	`DECLARE arrayname: ARRAY[l1:u1, l2:u2] OF DATATYPE` ***(where l1 is the lower and u1 is the upper index of row and l2 is the lower and u2 is the upper index of column)***
Flowchart Representation	`DECLARE arrayname: ARRAY[l:u] OF DATATYPE`	`DECLARE arrayname: ARRAY[l1:u1, l2:u2] OF DATATYPE`
Example (Pseudocode)	`DECLARE StudentNames: ARRAY[1:30] OF STRING` ***(The array StudentNames can store 30 names at a time of type string)***	DECLARE NoughtsAndCrosses : ARRAY[1:3, 1:3] OF CHAR ***(The array NoughtsAndCrosses have 3 rows and 3 columns and can store a total of 9 characters)***

A **subroutine** is a self-contained piece of code that has an identifier (name), and it can be called from anywhere in the main program. Subroutine calls are represented using subroutine symbol 

Purpose of subroutine

Subroutines are useful because **they reduce code**. You write the subroutine once, and then you can call it as many times as you need to, instead of having to re-write it every time. Each time you re-write it there is a chance of an error, so this **reduces the chances of error**.

*A **parameter** is a value that is **sent from the main program or calling program to the subroutine** (procedure or function). Parameters are declared inside the brackets after the subroutines name*

There are two types of subroutine: **procedures** and **functions.**

4. Functions	5. Procedures
➢ Functions are self-contained blocks of program statements that perform a specific task and **returns a value** to the calling program or routine. ➢ *Function definition* tells the task or action to be performed by the function and *function call* is the invoking of function definition by the main program or a routine. ➢ Function is defined once only but function calls are done many times. ➢ **Function calls are not standalone** and instead can be made on the right-hand side of an expression. ➢ Function can be with or without parameters and can take one or more values as parameter	➢ Procedures are self-contained blocks of program statements that perform a specific task and **do not return a value** to the calling program or routine. ➢ *Procedure definition* tells the task or action to be performed by the procedure *procedure call* is the invoking of procedure definition by the main program or a routine. ➢ Procedure is defined once only but procedure calls are done many times. ➢ **Procedure calls are single standalone** statements ➢ Procedure can be with or without parameters and can take one or more values as parameter
STRUCTURE OF FUNCTION	**STRUCTURE OF PROCEDURE**

Function without parameter
A function with no parameters is defined as follows:
FUNCTION <identifier> RETURNS <data type>
<statements>
RETURN value
ENDFUNCTION
The keyword RETURN is used as one of the statements within the body of the function to specify the value to be returned. Normally, this will be the last statement in the function definition.
When **the function is called it returns a value,** so something needs to happen with this value.
Functions should be called as follows:
`OUTPUT "……", FunctionName()`
OR
It could be saved in a variable, e.g.
`VariableName ← FunctionName()`
	Procedure without parameter
	A procedure with no parameters is defined as follows:
	PROCEDURE <identifier>
	<statements>
	ENDPROCEDURE
	Procedures should be called as follows:
	CALL <identifier>

Example

Write a function to ask the user to enter two values, add them together and return the value:

```
//Function Definition
FUNCTION Multiply() RETURNS INTEGER
    DECLARE Num1, Num2, Ans: INTEGER
    OUTPUT "Enter a number"
    INPUT Num1
    OUTPUT "Enter another number"
    INPUT Num2
    Ans ← Num1 * Num2
    RETURN Ans //Returning the value
// RETURN statement can be given as an
//expression. For eg. RETURN Num1 *
//Num2
ENDFUNCTION
```

To output the return value the main program can use:

```
Result ← Multiply() // function call
```

OR

OUTPUT "The product of 2 integers=", Multiply()

Function with parameter

A function with parameters is defined as follows:

FUNCTION <identifier>(<param1>:<datatype>, <param2>:<datatype>...) RETURNS <data type>
<statements>
ENDFUNCTION

The keyword CALL should not be used when calling a function.

Functions should only be called as part of an expression. When the RETURN statement is executed, the value returned replaces the function call in the expression and the expression is then evaluated.

Example

A procedure to output the numbers 1 to 10:

```
//Procedure definition
PROCEDURE Output1To10
    DECLARE Count:INTEGER // local variable
    FOR Count ←1 to 10
        OUTPUT Count
    NEXT Count
ENDPROCEDURE
```

The main program can then call the procedure with the code:

```
//Procedure Call
CALL Output1To10
```

Procedure with parameter

A procedure with parameters is defined as follows:

PROCEDURE <identifier>(<param1>:<datatype>, <param2>:<datatype>...)
<statements>
ENDPROCEDURE

The <identifier> is the identifier used to call the procedure. Where used, param1, param2, etc. are identifiers for the parameters of the procedure. These will be used as variables in the statements of the procedure.

Procedures should be called as follows:

CALL <identifier>(Value1,Value2...)

When parameters are used, Value1, Value2... must be of the correct data type as in the definition of the procedure.

When the procedure is called

- Control is passed to the procedure.
- If there are any parameters, these are substituted by their values, and the statements in the procedure are executed.
- Control is then returned to the line that follows the procedure call.

EXAMPLE: FUNCTION WITH PARAMETER

To find largest of two numbers

```
//Function Definition
FUNCTION LargestTwo(N1:INTEGER, N2:INTEGER)
RETURNS INTEGER
  IF N1>N2
    THEN
      RETURN N1
  ELSE
      RETURN N2
  ENDIF
ENDFUNCTION

//In the calling program or routine
DECLARE FirstNum, SecondNum:INTEGER
OUTPUT "Enter the smallest number"
INPUT FirstNum
OUTPUT "Enter the largest number"
INPUT SecondNum
Large←LargestTwo(FirstNum,SecondNum)//Fn
//call
OUTPUT "Largest of 2 integers = ", Large
```

EXAMPLE: PROCEDURE WITH PARAMETER

A procedure takes 2 values and outputs all the numbers between the first number to the second:

```
//Procedure definition
PROCEDURE OutputNumbers(Num1:INTEGER,
Num2:INTEGER)
      DECLARE Count:INTEGER
      FOR Count ← Num1 TO Num2
            OUTPUT Count
      NEXT Count
ENDPROCEDURE
```

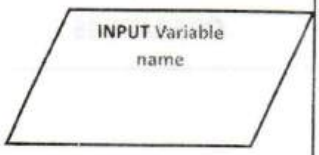
The main program takes two values from the user.

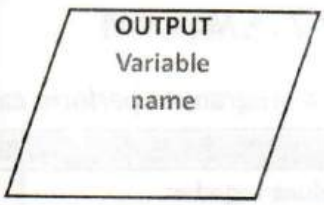
```
DECLARE FirstNum, SecondNum:INTEGER
OUTPUT "Enter the smallest number"
INPUT FirstNum
OUTPUT "Enter the largest number"
INPUT SecondNum
CALL OutputNumbers(FirstNum, SecondNum)
```

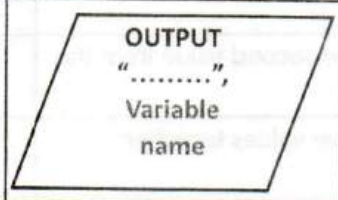
SCOPE

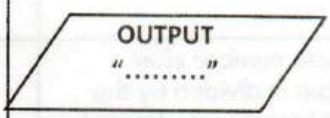
The **scope** of a variable is the area within a program that it can be accessed.
There are two scopes: **global** and **local.**

Global	Local
The variable or constant can be accessed from any part of the program. A global variable can be used by any part of a program - its scope covers the whole program. **Drawback** - *Restrict the use of global variables, because their memory is taken for the whole of the program and nothing else can use that memory space.*	The variable or constant can only be accessed in the subroutine it is declared within. Local variable can only be used by the part of the program it has been declared in - its scope is restricted to that part of the program. *When that part of the program finishes the memory location is freed if it's a local variable.*

INPUT STATEMENTS		
Pseudocode	**Flowchart Representation**	**Python Code**
INPUT Variable name Ex: INPUT Counter INPUT Name *(Here Counter and Name are variables which is used to store the value given by the user from the terminal)*		Syntax: Variable name=datatype(input()) Ex: Counter = int(input()) *(The variable Number stores an integer data)* Syntax: Variable name=input() Ex: Name=input() *(The variable Name stores a string)* The default datatype is string in python.
INPUT ".....", Variable name Ex: INPUT "Enter the integer", Counter		Syntax: Variable name=datatype(input("......")) Ex: Counter=int(input("Enter the integer")) *(Used in python program)*
OUTPUT STATEMENTS		
Pseudocode	**Flowchart Representation**	**Python Code**

OUTPUT Variable name
or
PRINT Variable name
 	
Ex: OUTPUT Area
//Displays the value stored in the
//variable Area
		Syntax:
		print(Variable name)
		
		Ex:
		print(Area)
		(The value in the variable Area is displayed on the screen)

OUTPUT ".....", Variable name
Ex:
OUTPUT "Area of rectangle", Area
//Displays the sentence Area of
//rectangle along with the value
//stored in the variable Area
		Syntax:
		print(".....", Variable name)
		
		Ex:
		print("Area of rectangle",Area)

OUTPUT "....."
Ex:
OUTPUT "Enter an integer"
//Displays the sentence Area of
//rectangle along with the value
//stored in the variable Area
		Syntax:
		print(".....")
		
		Ex:
		print("Enter an integer")

OPERATORS

ASSIGNMENT OPERATOR (←)

- The assignment operator is ←
- Assignments should be made in the following format:

  <identifier> ← <value>
- The identifier must refer to a variable (this can be an individual element in a data structure such as an array or an abstract data type).
- The value may be any expression that evaluates to a value of the same data type as the variable.

Examples

```
Counter ← 0  //Assigns the value 0 to the variable Counter
Counter ← Counter + 1 //Assigns the new updated value to variable Counter, after adding a 1 to the
//previous Counter value
TotalToPay ← NumberOfHours * HourlyRate //Assigns the product of two values into the variable
//TotalToPay
```

ARITHMETIC OPERATOR (+, - , * ,/ , ^, MOD, DIV)

Arithmetic operators instruct a program to perform calculations.

Operator	Description	Example
+	Adds two values together.	10 + 2 gives 12 11.3 + 9 gives 20.3
-	Subtracts the second value from the first.	10 - 2 gives 8 11.3 - 9 gives 2.3
*	Multiplies two values together.	10 * 2 gives 20 11.3 * 9 gives 101.7
/	Divides the first number by the second.	10 / 2 gives 5 11.3 / 9 gives 1.256
DIV	Gives the whole number after the first number is divided by the second, i.e. it ignores any decimals.	DIV(10, 2) gives 5 DIV(11, 9) gives 1
MOD	Gives the remainder after the first number is divided by the second, i.e. how many are left.	MOD(10, 2) gives 0 MOD(11, 9) gives 2
^	Power of.	2 ^ 3 = 8 3 ^ 2 = 9

LOGICAL OPERATOR (=, <>, >, >=, <, <=)

- A symbol that performs a comparison resulting in True or False.
- Conditions need **logical operators**
- The result of these operations is always of data type BOOLEAN.
- In complex expressions, it is advisable to use parentheses to make the order of operations explicit.

Logical operator	Description	Example
= or ==	Equals to	10 = 10? would give TRUE. 10 is equal to 10. 10 = 2? would give FALSE. 10 is not equal to 2.
<> or !=	Not equal to	10 <> 10? would give FALSE. 10 is not, not equal to 10. 10 <> 2? would give TRUE. 10 is not equal to 2.
<	Less than	10 < 11? would give TRUE. 10 is less than 11. 10 < 10? would give FALSE. 10 is not less than 10. 11 < 10? would give FALSE. 11 is not less than 10.

Logical operator	Description	Example
<=	Less than or equal to	10 <= 11? would give TRUE. 10 is less than or equal to 10. 10 <= 10? would give TRUE. 10 is less than or equal to 10. 11 <= 10? would give FALSE. 11 is not less than or equal to 10.
>	Greater than	10 > 11? would give FALSE. 10 is not greater than 11. 10 > 10? would give FALSE. 10 is not greater than 10. 11 > 10? would give TRUE. 11 is greater than 10.
>=	Greater than or equal to	10 >= 11? would give FALSE. 10 is not greater than or equal to 11. 10 >= 10? would give TRUE. 10 is greater than or equal to 10. 11 >= 10? would give TRUE. 11 is greater than or equal to 10.

BOOLEAN OPERATOR (AND, OR, NOT)

- Use to join conditions
- The only Boolean operators used are AND, OR and NOT. The operands and results of these operations are always of data type BOOLEAN.
- In complex expressions, it is advisable to use parentheses to make the order of operations explicit.

Boolean operator	Description	Example

AND	If both conditions are true, the result is true.
	If one or both conditions are false, the result is false.
		`IF 1 = 1 AND 2 = 2`
		This will return TRUE. The left of the AND is true, and the right of the AND is true.
		`IF 1 = 1 AND 1 > 2`
		This will return FALSE. The left of AND is true, but the right of AND is false.
		`IF 1 < -2 AND 0 < -1`
		This will return FALSE. Both comparisons are false, so the result is false.

OR	If one, or both, conditions are true, the result is true.
	If both conditions are false, the result is false.
		`IF 1 = 1 OR 2 = 2`
		This will return TRUE. The left of the OR is true, and the right of the OR is true.
		`IF 1 = 1 OR 1 > 2`
		This will return TRUE. The left of OR is true, but the right of OR is false.
		`IF 1 < -2 OR 0 < -1`
		This will return FALSE. Both comparisons are false, so the result is false.

NOT	Reverse the condition. If the condition is True it becomes False.	`IF NOT(1 = 1)`
		The brackets equal to TRUE, 1 equals 1. The NOT makes it FALSE, so it becomes 1 does not equal 1.
		`IF NOT (End of File)`
		This is used with file handing. End of File will return TRUE if there is no data left in the file. The NOT turns this to false. So while not at the end of the file.

LIBRARY ROUTINES

Library routines are pre-written and pretested subroutines that can be called from within a program.

MOD(Dividend, Divisor)	DIV(Dividend, Divisor)	ROUND (Real Number, No of decimal places)	RANDOM(Value1, Value2)

❖ Returns remainder of an integer division.
 	
❖ Take **2 integers** as parameters and returns an integer.
 	
EXAMPLE
Rem ← MOD(10,2)
OUTPUT
"Reminder=",Rem
 	
Output
Reminder=0
	❖ Returns the quotient (i.e. the whole number part) of an integer division.
	
	❖ Take **2 integers** as parameters and returns an integer.
	
	EXAMPLE
	Quo ← DIV(10,2)
	OUTPUT
	"Quotient=",Quo
	
	Output
	Quotient=5
		❖ Returns a value rounded to a given number of decimal places
		
		❖ Take 2 parameters where **first parameter is a real number** and the **second is an integer**
		
		EXAMPLE
		//Rounding to 2 decimal places
		Value2Dp ← ROUND(4.829,2)
		OUTPUT"Value to two decimal places", Value2Dp
		
		//Rounding to 1 decimal place
		Value1Dp ← ROUND(4.829,1)
		OUTPUT"Value to one decimal place", Value1Dp
		
		//Rounding to nearest whole number
		WhleNum ← ROUND((4.829+0.5),0)
		OUTPUT"Rounded to nearest whole number", WhleNum
		
		Output
		Value to two decimal places 4.82
		Value to one decimal place 4.8
		Rounded to nearest whole number 5.0
			❖ Returns a random number between 2 set of values.
			
			❖ Take **2 integers** as parameters and returns an integer.
			
			EXAMPLE
			RANDOM (10,20) will return a number between 10 and 20.

STRING HANDLING ROUTINES

A string is a piece of text. This could be made up of characters, numbers and/or symbols.

STRING OPERATIONS	EXAMPLE
LENGTH(<identifier>) ➢ Returns the **integer value** representing the **length of string**. ➢ The identifier should be of data type string. ➢ This command will return the number of characters in a string.	LENGTH ("hi ") would return 2 LENGTH ("0123") would return 4. Sentence1 ← "Computer science is fun" NoOfChars ← LENGTH(Sentence1) OUTPUT "Number of characters in the sentence is", NoOfChars **Output** Number of characters in the sentence is 23
LCASE(<identifier>) ➢ Returns the string/character with all characters in lower case. ➢ The identifier should be of data type string or char.	Word ← "HELLO" Word ← LCASE(Word) OUTPUT "In lower case is:", Word **Output** In lower case is hello
UCASE(<identifier>) ➢ Returns the string/character with all characters in upper case. ➢ The identifier should be of data type string or char.	Word ← "hello" Word ← UCASE(Word) OUTPUT "In upper case is:", Word **Output** In upper case is HELLO

SUBSTRING(<identifier>, <start>, <length>)

- Returns a string of length **length** starting at position start.
- The identifier should be of data type string, length and start should be positive and data type integer.
- Depending on your language, the first character could be in position 0 or position 1.

Examples using SUBSTRING

Position starts at 0 in the following examples

Example 1

Using substring:
SUBSTRING ("Hello", 0, 1). This will start at character 0 and take 1 character. It will return **"H"**.

Example 2

Using substring:
SUBSTRING ("Goodbye" , 4 , 3) . This will start at character 4 and take 3 characters. It will return "bye".

Example 3

Output the first 4 characters in a string:

```
StringData ← "Goodbye"
NewMessage ← SUBSTRING(StringData, 0, 4)
OUTPUT NewMessage
```

This will give "Good" as output

Example 4

Output each letter of a string one character at a time. Depending on your language, the stopping condition might be the length, or the length -1 depending on whether the first character is 0 or 1.

```
//Position starting at 0
OUTPUT "Enter a message"
INPUT Stringinput
FOR Count ← 0 TO LENGTH(Stringinput) - 1
    SingleCharacter ← SUBSTRING(Stringinput, Count, 1)
    OUTPUT SingleCharacter
NEXT Count
```

```
//Position starting at 1
OUTPUT "Enter a message"
INPUT Stringinput
FOR Count ← 1 TO LENGTH(Stringinput)
    SingleCharacter ← SUBSTRING(Stringinput, Count, 1)
    OUTPUT SingleCharacter
NEXT Count
```

Generally, a start position of 1 is the first character in the string.

Example – string operations

```
LENGTH("Happy Days") will return 10
LCASE('W') will return 'w'
UCASE("Happy") will return "HAPPY"
SUBSTRING("Happy Days", 1, 5) will return "Happy"
```

CONTROL STATEMENTS are statements which control the flow of execution of a program.

1. **SEQUENCE STATEMENTS** – The statements are executed in sequential order and all the statements are executed without skipping any part of the pseudocode or program.

2. **SELECTION/BRANCHING/CONDITIONAL/ DECISION MAKING STATEMENTS**

   Used to select a part of a program or pseudocode to be executed based on certain conditions or criteria. Allows different routes through a program.

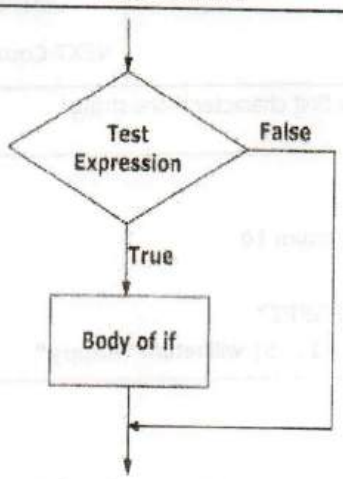
   Includes:

   - **IF Statements**
   - **CASE Statements**

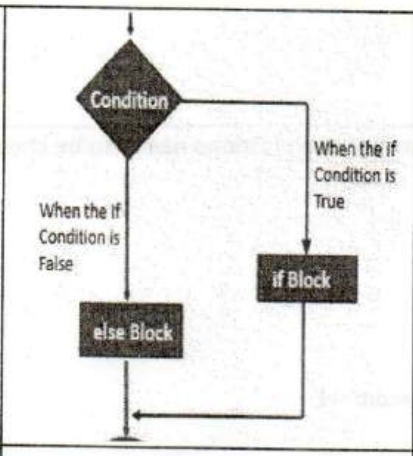
3. **LOOPING/ITERATION/ REPETITION STATEMENTS** – A series of statements that executes for a specified number of repetitions or until specified conditions are met.

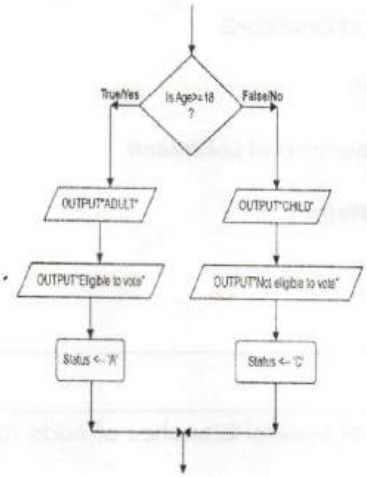
   There are three types of loop structure which perform iterations to repeat programming code.

   - **Count-controlled loops (for a set number of iterations) – FOR.....TO....NEXT**
   - **Pre-condition loops – may have no iterations - WHILE.....DO.....ENDWHILE**
   - **Post-condition loops – always has at least one iteration – REPEAT.....UNTIL....**

SELECTION/BRANCHING/DECISION/CONDITIONAL STATEMENTS		
1. IF Statements		
PSEUDOCODE	**FLOWCHART**	**PYTHON CODE**
SIMPLE IF **IF condition** **THEN** **True Statements** **ENDIF** **Ex: IF Age>=18** **THEN** **OUTPUT "ADULT"** **OUTPUT "Eligible to vote"** **Status← 'A'** **ENDIF** ***(Here a condition is checked to check if the Age is greater than or equal 18, if the condition is true, then a set of statements are executed)***	 Fig: Operation of if statement	**if condition:** **true statements** **Example:** **if Age>=18:** **print("ADULT")** **print("Eligible to vote")** **Status='A'** **#end of if**

<table>
<tr>
<td><u>IF ELSE</u>

IF condition
THEN
True Statements
ELSE
False Statements
ENDIF</td>
<td> 
</td>
<td>if condition:
true statements
else:
false statements</td>
</tr>
<tr>
<td>Ex: IF Age>=18
THEN
OUTPUT “ADULT”
OUTPUT “Eligible to vote”
Status← ‘A’
ELSE
OUTPUT “CHILD”
OUTPUT “Not eligible to vote”
Status← ‘C’
ENDIF

(Here a condition is checked to check if the Age is greater than or equal 18, if the condition is true, then a set of statements are executed, otherwise another set of statements are executed.)</td>
<td> 
</td>
<td>Ex:
if Age>=18:
print(“ADULT”)
print(“Eligible to vote”)
Status← ‘A’
else:
print(“CHILD”)
print(“Not eligible to vote”)
Status← ‘C’
#End of if else</td>
</tr>
<tr>
<td>NESTED IF

IF Condition1
THEN
IF Condition2
THEN
True Statements of condition2
ELSE
False Statements of condition2
ENDIF
ELSE
False Statements of condition1
ENDIF

(Nested if checks for another condition if first condition is true i.e. if inside another if is known as nested if.
It can take different forms based on the situation)</td>
<td></td>
<td>if condition1:
if condition2:
True statements of cond2
else:
False statements of cond2
else:
False statements of cond1</td>
</tr>
</table>

IF ELSE IF Ladder – Used when multiple conditions needs to be checked

PSEUDOCODE

```
IF Condition1
   THEN
   True Statements of Condition1
   ELSE IF Condition2
           THEN
              True Statements of Condition2
              ELSE IF Condition3
                      THEN
                      True Statements of Condition3
              .....
                      ELSE IF ConditionN
                              THEN
                               True Statements of ConditionN
                             ELSE
                               False Statement
                      ENDIF
             ENDIF
      ENDIF
ENDIF
```

2. CASE STATEMENTS

CASE statements allow one out of several branches of code to be executed, depending on the value of a variable.

CASE statement allows the program to take one variable or value, and then have lots of different options depending what it is equal to.

CASE statements are written as follows:

```
CASE OF <identifier>
        <value 1> : <statement>
        <value 2> : <statement>
          ...
  ENDCASE
```

EXAMPLE:

```
INPUT N1, N2 //Inputting 2 numbers
OUTPUT"1. Addition, 2. Subtraction, 3. Multiplication, 4. Division"
OUTPUT "Press any number between 1 and 4"
INPUT Option //To input an option
CASE OF Option
1: Ans← N1+N2
2: Ans← N1- N2
3: Ans← N1 *N2
4: Ans← N1/N2
OTHERWISE
  OUTPUT "Invalid Option"
ENDCASE
OUTPUT "Result of calculation is", Ans
```

An OTHERWISE clause can be the last case:

```
CASE OF <identifier>
      <value 1> : <statement>
      <value 2> : <statement>
        ...
    OTHERWISE <statement>
ENDCASE
```

LOOPING/ITERATION/ REPETITION STATEMENTS

COUNT CONTROLLED LOOP (FOR...TO...NEXT)

- This type of loop uses a counter to run a set number of times.
- The most common count-controlled loop is the for loop.

This has the structure:

FOR *variable* ← *start value* TO *endvalue*
Code that runs repeatedly
NEXT *variable*

The loop will run from the start value to the end value, increasing by 1 each time. If the start value is 1 and the end value is 10, it will run 10 times (1, 2, 3, 4, 5, 6, 7, 8, 9 and 10).

An increment can be specified as follows:

FOR <identifier> ← <value1> TO <value2> STEP <increment>
<statements>
NEXT <identifier>

- The increment must be an expression that evaluates to an integer.
- In this case the identifier will be assigned the values from value1 in successive increments until it reaches value2.
- If it goes past value2, the loop terminates. The increment can be negative.

Example
To input 10 numbers
FOR Count← 1 TO 10
Output "Enter a number"
INPUT Number
NEXT Count

PRE-CONDITION LOOP (WHILE....DO...ENDWHILE)

- A pre-condition loop tests the condition before starting the loop. This means that if the condition is false, the code inside the loop will not run.
- It loops while the condition is true.
- It stops looping when the condition is false.
- A WHILE loop is a pre-condition loop.

It has the structure:

WHILE *condition* DO
Code that will run when the condition is true
ENDWHILE

Example
To input 10 numbers

Count← 1
WHILE Count<=10
Output "Enter a number"
INPUT Number
Count ← Count + 1
ENDWHILE

POST-CONDITION LOOP (REPEAT.....UNTIL.....)

- A post-condition loop runs the code inside the loop once, and then checks the condition at the end of the loop.
- This means that the code will always run once.
- A REPEAT UNTIL loop is a post-condition loop.

It has the structure:

REPEAT
Code that runs inside the loop
UNTIL *Condition*

In this case it continues until the Condition becomes True. It loops while the condition is False.

Example
To input 10 numbers
Count← 1
REPEAT
Output "Enter a number"
INPUT Number
Count ← Count + 1
UNTIL Count>10

(Note: If Count is initialized with zero, then the condition will be UNTIL Count=10)

NESTED STATEMENTS

A construct (selection or iteration) that is inside another construct.

A **nested statement** is one or more selection and/or iteration statements inside another selection/iteration statement.

This could be an **IF** statement inside an **IF** statement or a loop inside an IF statement or an **IF** statement in a loop or a loop within a loop.

Example 1

Count how many numbers entered are more than 10, and how many are equal to 10:

```
MoreThan10 <---- 0
EqualTo10 <---- 0
FOR X <---- 0 TO 99
    OUTPUT "Enter a number"
    INPUT Number
IF Number > 10 THEN
      MoreThan10 <---- MoreThan10 + 1
    ENDIF
    IF Number = 10 THEN
        EqualTo10 <---- EqualTo10 + 1
    ENDIF
NEXT X
```

This code has an IF statement nested inside a count-controlled loop.

```
REPEAT
  Swap←0 // Swap is an integer variable
  FOR Index ←1 TO Last - 1
      IF Values[Index] > Values[Index + 1]
         THEN
           Temp←Values[Index]
           Values[Index]←Values[Index + 1]
           Values[Index + 1]←Temp
           Swap  1
      ENDIF
  NEXT
  Last←Last - 1
UNTIL Swap=0 OR Last = 1
```

This code has a count-controlled loop (FOR loop) nested inside post-condition loop (REPEAT..UNTIL..)

ARRAYS

One Dimensional Array

To input values into a 1D array

```
//Using FOR loop
FOR Count ← 1 TO Size
      Output "Enter a number"
      INPUT ArrayName[Count]
NEXT Count

//Using WHILE loop
Count ← 1
WHILE Count<=Size
   DO
      Output "Enter a number"
      INPUT ArrayName[Count]
      Count ← Count + 1
ENDWHILE

//Using REPEAT loop
Count ← 1
REPEAT
      Output "Enter a number"
      INPUT ArrayName[Count]
      Count ← Count + 1
UNTIL Count<=Size
```

Two Dimensional Array

To input values into a 2D array

Index	1	2	3
1	30 11	12	13
2	21	22	23
3	31	32	33
4	41	42	43

```
//To input values Row-wise
FOR Row ← 1 TO RowSize
  FOR Col ←1 To ColumnSize
     OUTPUT "Enter value in row",Row,"and column",Col
     INPUT ArrayName[Row, Col]
  NEXT Col
NEXT Row

//To input values Column-wise
FOR Col ←1 To ColumnSize
  FOR Row ← 1 TO RowSize
     OUTPUT "Enter value in row",Row,"and column",Col
     INPUT ArrayName[Row, Col]
  NEXT Col
NEXT Row
```

TO CALCULATE TOTAL & AVERAGE

```
//Using FOR loop
Total ← 0
FOR Count ← 1 TO Size
      Output "Enter a number"
      INPUT ArrayName[Count]
      Total ← Total + ArrayName[Count]
NEXT Count
Avg ← Total/Size
OUTPUT "Total =", Total
OUTPUT "Average=", Avg

//Using WHILE loop
Count ← 1
Total ← 0
WHILE Count<=Size
   DO
      Output "Enter a number"
      INPUT ArrayName[Count]
      Total ← Total + ArrayName[Count]
      Count ← Count + 1
ENDWHILE
Avg ← Total/Size
OUTPUT "Total =", Total
OUTPUT "Average=", Avg

//Using REPEAT loop
```

```
//To calculate Row-wise total
FOR Row ← 1 TO RowSize
   Total ← 0
  FOR Col ←1 To ColumnSize
     OUTPUT "Enter value in row",Row,"and column",Col
     INPUT ArrayName[Row, Col]
     Total ← Total + ArrayName[Row,Col]
  NEXT Col
  Avg←Total/ColumnSize
  OUTPUT "Total =", Total
  OUTPUT "Average=", Avg
NEXT Row

//To calculate Column-wise total
FOR Col ←1 To ColumnSize
  Total ← 0
  FOR Row ← 1 TO RowSize
     OUTPUT "Enter value in row",Row,"and column",Col
     INPUT ArrayName[Row, Col]
     Total ← Total + ArrayName[Row,Col]
  NEXT Col
  Avg←Total/RowSize
  OUTPUT "Total =", Total
  OUTPUT "Average=", Avg
NEXT Row
```

```
Count ← 1
Total ← 0
REPEAT
    Output "Enter a number"
    INPUT ArrayName[Count]
    Total ← Total + ArrayName[Count]
    Count ← Count + 1
UNTIL Count<=Size
Avg ← Total/Size
OUTPUT "Total =", Total
OUTPUT "Average=", Avg
```

FILE HANDLING

PURPOSE OF FILE

When data is saved to a file

- It is stored permanently
- Can be accessed by the same program at a later date
- Can be accessed by another program, or sent to be used on another computer(s).

Open a file, stating the mode of operation, before reading from or writing to it.

This is written as follows:

OPENFILE <File identifier> FOR <File mode>

The file identifier will be the name of the file with a data type string.

The following file modes are used:

- READ for data to be read from the file
- WRITE for data to be written to the file.

A new file will be created and any existing data in the file will be lost.
A file should be opened in only one mode at a time.

Data is read from the file (after the file has been opened in READ mode) using the READFILE command as follows:

READFILE <File Identifier>, <Variable>

When the command is executed, the data item is read and assigned to the variable.

Data is written into the file after the file has been opened using the WRITEFILE command as follows:

WRITEFILE <File identifier>, <Variable>

When the command is executed, the data is written into the file.

Files should be closed when they are no longer needed using the CLOSEFILE command as follows:

CLOSEFILE <File identifier>

Example – file handling operations

This example uses the operations together, to copy a line of text from `FileA.txt` to `FileB.txt`

```
DECLARE LineOfText : STRING
OPENFILE "FileA.txt" FOR READ
OPENFILE "FileB.txt" FOR WRITE
READFILE "FileA.txt", LineOfText
WRITEFILE "FileB.txt", LineOfText
CLOSEFILE "FileA.txt"
CLOSEFILE "FileB.txt"
```

Features of maintainable program

- Use comments …
- … to explain the purpose of each section of code
- …for example, logic / syntax
- Use meaningful identifier names to …
- … clearly identify the purpose …
- … of variables, constants, arrays, procedures etc
- Use procedures and functions …
- … to avoid repeated code
- … simplify logic
- Use indentation and white space …
  … to make the program readable