PYTHON

HANDWRITTEN NOTES

Chip Sized

chipsized

Chip Sized

170+ PAGES

PREFACE

Hello friends! I present to you my personal handwritten notes of Python which consists of over 170+ pages involving all important topics. It took about 25 hours to complete the notes.

I request you to support my work by rating this book 5 out of 5. You can get this book for free on Gumroad. But getting it for a small amount would be of a great help if you are willing to support me and my channel.

I would constantly work to provide you with better and useful content. You can follow and subscribe to my LinkedIn, Instagram and Youtube channel.

PAYPAL DONATION

paypal.me/chipsized

PYTHON

CONTENTS

PYTHON

- SETS
- BOOLEANS
- NONE TYPE
- MUTABILITY OF DATA TYPES
- IMPLICIT TYPE CONVERSION
- EXPLICIT TYPE CONVERSION

GETTING USER INPUT

OPERATORS

- INTRODUCTION
- ARITHMETIC OPERATORS
- COMPARISON OPERATORS
- LOGICAL OPERATORS
- BITWISE OPERATORS
- ASSIGNMENT OPERATORS
- IDENTITY OPERATORS
- MEMBERSHIP OPERATORS
- OPERATOR PRECEDENCE
- OPERATOR ASSOCIATIVITY

BLOCKS AND INDENTATION

- INTRODUCTION
- IMPORTANT POINTS

CONDITIONAL STATEMENTS

- INTRODUCTION
- THE IF STATEMENT
- THE IF-ELSE STATEMENT
- THE IF-ELIF-ELSE LADDER
- ORDER OF CONDITIONS
- THE NESTED-IF STATEMENT
- SHORT HAND IF
- SHORT HAND IF-ELSE
- THE PASS STATEMENT

PYTHON

PYTHON

MATCH CASE

- INTRODUCTION
- USING MULTIPLE PATTERN VALUES
- USING A GUARD

LOOPS

- INTRODUCTION
- FLOWCHART - LOOPS
- WHILE LOOP
- INFINITE LOOP
- FOR LOOP
- THE RANGE FUNCTION
- NESTED LOOPS
- BREAK AND CONTINUE STATEMENTS
- THE ELSE STATEMENT
- THE PASS STATEMENT

STRINGS

- INTRODUCTION
- INDEXING IN STRINGS
- STRING SLICING
- STRING IMMUTABILITY
- MULTILINE STRINGS
- STRING CONCAT AND REPEAT
- STRING LENGTH AND MEMBERSHIP
- STRING METHODS
- STRING FORMATTING
- ESCAPE CHARACTERS

LISTS

- INTRODUCTION
- LIST SLICING
- CHANGING LIST VALUES
- INSERTING LIST ITEMS

- ADDING LIST ITEMS
- REMOVING LIST ITEMS
- COPYING LISTS
- SORTING LISTS
- REVERSING LISTS
- LIST CONCAT AND REPEAT
- LOOPING THROUGH A LIST
- LIST COMPREHENSION

TUPLES

- INTRODUCTION
- TUPLE ASSIGNMENT
- UNPACKING TUPLES
- UPDATING TUPLES
- DELETING TUPLES
- TUPLE CONCAT AND REPEAT
- TUPLE METHODS

SETS

- INTRODUCTION
- ACCESS AND CHANGE SET ITEMS
- ADD SET ITEMS
- REMOVE SET ITEMS
- SET OPERATIONS

DICTIONARIES

- INTRODUCTION
- ACCESS ITEMS
- CHANGE VALUES
- ADD DICTIONARY ITEMS
- REMOVE ITEMS
- LOOP THROUGH DICTIONARIES
- COPY A DICTIONARY

PYTHON

PYTHON

FUNCTIONS

- INTRODUCTION
- IMPORTANT POINTS
- FUNCTION CALL
- FUNCTION - EXPLANATION DIAGRAM
- PARAMETERS AND ARGUMENTS
- KEYWORD ARGUMENTS
- ARBITRARY ARGUMENTS
- ARBITRARY KEYWORD ARGUMENTS
- DEFAULT PARAMETERS
- CHANGES WHEN PASSING VALUES
- RETURN VALUES
- USING PASS IN FUNCTIONS
- TYPES OF FUNCTIONS

VARIABLE SCOPE

- GLOBAL, LOCAL AND NON-LOCAL VARIABLES
- OVERSHADOWING OF VARIABLES
- MODIFICATION IN DIFFERENT SCOPES

EXCEPTION HANDLING

- INTRODUCTION
- TRY AND EXCEPT BLOCK
- MULTIPLE EXCEPT BLOCKS
- ELSE AND FINALLY BLOCKS
- RAISING EXCEPTIONS

MODULES

- INTRODUCTION
- CREATING A MODULE
- IMPORT A MODULE
- USING THE FROM STATEMENT
- RENAMING THE MODULE
- BUILT-IN MODULES

PYTHON

__/__/____

PYTHON HANDWRITTEN NOTES

YOUTUBE - Chip Sized
LINKED IN - Chip Sized
INSTAGRAM - chipsized

INTRODUCTION TO PYTHON:

DEFINITION:

⊛ Python is an interpreted, high-level, and general-purpose programming language.

High level - Programs are easily understood by humans.

Interpreted - Uses an interpreter to execute programs.

General Purpose - Used in a variety of applications like web, desktop, ML, AI etc.

HISTORY:

Founder - Guido van Rossum

Year found - 1989

Purpose - Better Readability.

Versions - Python 2 and 3 (Currently Python 3.10)

__/__/____

FEATURES OF PYTHON :

1) Easy to write, read, and learn.
2) Free and open-source.
3) Interpreted
4) Supports modularity.
5) Extensible
6) Dynamic type system.
7) Automatic memory management.
8) Supports third-party packages.
9) Object-oriented.

APPLICATIONS OF PYTHON :

1) Web Development
2) Game Development
3) Machine Learning and AI.
4) Data Science and Visualization.
5) Desktop GUI
6) Web Scraping Apps.
7) Business Applications (E-Commerce)
8) Audio and Video Applications
9) CAD Applications
10) Embedded Applications.

⭐ Top companies such as Google, Facebook, Instagram, Quora, Dropbox, Udemy and IBM uses Python.

⭐ It is number one in popularity among languages.

INSTALLING PYTHON :

Note : (To check whether Python is already installed, open command prompt (CMD), and type the command "python --version". If it shows a version, you have it already installed. If not you must download the Python Installer.)

STEPS :

1) Open your browser and search python.
2) Click the link of python.org (search).
3) Download the latest version.
4) Open the Installer.
5) Check the two boxes below (Install for all users, add Python to PATH.)
6) Click on Install Now. Wait until it's complete and then close.
7) Again open CMD. Type the command "python --version", and it would display the version.

Note : (There may be different versions of Python and are constantly updated. At the time of writing, the version was Python 3.10.7.

Download the latest version anyway as there may be very little difference among different version.)

__/__/____

INTERACTIVE AND SCRIPT MODE :

⊛ In Python, there are two ways to run our code.

1) Interactive Mode
2) Script Mode.

INTERACTIVE MODE	SCRIPT MODE
1) Python statements are written in CMD and we get the result of each line instantly.	1) Python program is written in a file. The Python interpreter executes the complete file and displays output in CMD.
2) Better suited for writing very short programs.	2) More suitable for long programs.
3) Code can be edited, but it is hard to do.	3) Editing of code is easily done.
4) Code cannot be saved and used in future.	4) Code can be saved and used in future.
5) Suitable for practice and understanding code.	5) Suitable for writing programs and projects.

__/__/___

THE PRINT FUNCTION :

⊛ The print() function is used to display the output in the console.

SYNTAX :
print (<value>)

EX :	O/P :
print ('Hello World')	Hello World

MULTIPLE PRINT FUNCTIONS :

⊛ We can use multiple print functions. It prints each value in a newline.

EX :	O/P :
print (20)	20
print (3.5b)	3.5b
print ('Hello')	Hello
print ("Python")	Python

⊛ Text (called strings) in Python must be given within single or double quotes.

PRINT MULTIPLE VALUES :

⊛ Multiple values can be printed using a single print function and separating values with a comma.

SYNTAX :
print (<val 1>, <val 2>, ..., <val n>)

Ex:
```
print (10, 'Hello', "Python", 3.87)
```

O/P:
```
10 Hello Python 3.87
```

* The values are printed in a single line with a white space in between.

USING END ARGUMENT:

* Multiple values can be printed using different print functions and still make them display in a new single line.

* The end argument is used for that within the print function to print a string after all values are printed.

* By default, it holds a '\n' (newline) character and that is why, a new print function starts printing in the newline.

Ex:
```
print ("Hello", end = '#')
print ("Morning")
```

O/P:
```
Hello # Morning
```

__/__/____

USING SEP ARGUMENT:

⊛ The sep argument can be used within the print function to print a string between all values when multiple values are used.

⊛ By default, the sep argument holds a whitespace (" ") and that's why, values printed in the same line leave a whitespace.

EX:

```
print ("Hello", 10, 20.45, sep = " * ")
```

O/P:

```
Hello * 10 * 20.45
```

__/__/____

COMMENTS:

* Comments are used for explaining code in plain English.

* The Python interpreter ignores comments and they are not executed.

* A comment starts with a # symbol and is terminated by the end of line.

* Inline comments are written beside a Python statement.

```
Ex:
# This is a single-line comment
print ("My Code") # This prints My Code
```

```
O/p:
My Code
```

* Here, the first comment is a single line comment and the second comment is an inline comment.

VARIABLES:

⊛ Variables are names assigned to memory locations. The values assigned to variables can be used within the program.

SYNTAX:

```
<variable-name> = <value>
```

Note: No need to declare a variable or specify the data type. Python is dynamically typed (data type automatically detected during runtime).

EX:	O/P:
a = 10 print (a)	10

⊛ Here, 'a' is the variable name for the value 10. When you refer to 'a' anywhere in the program, that means you are referring to 10.

RULES FOR NAMING VARIABLES:

1) Variable names must start with a letter or an underscore.

```
    x = 20      # valid
_chip = "sized" # valid
  $a = 13.94    # invalid
 3b = 44        # invalid
```

2) Variable names must start with letters or underscores, but digits can be used except the start. Special symbols can't be used.

```
a_var = "chocolate"        #valid
num1 = 3                   #valid.
```

3) Names are case-sensitive

a = 10 print (A)	O/P: Error

4) Python keywords are not allowed. There are 35 keywords (as of version 3.10).

The list of keywords can be seen using this code.

```
Import keyword
print ( keyword . kwlist)
```

ASSIGNING ONE VARIABLE TO ANOTHER:

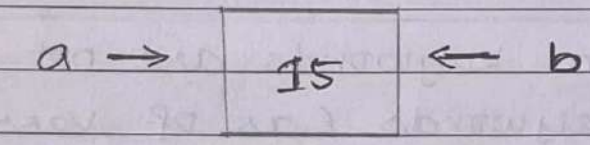
⊛ A variable containing a value can be assigned to another variable.

```
EX:
a = 20
b = a   # Assigning a (which has value 20)
        # to b
print (b)
print (a)
```

O/P:
20
20

⭐ Note: When different variables have the same value, it means that the values has two names. This can be visually displayed.

When a = 15 , b = 15



UPDATING VARIABLES:

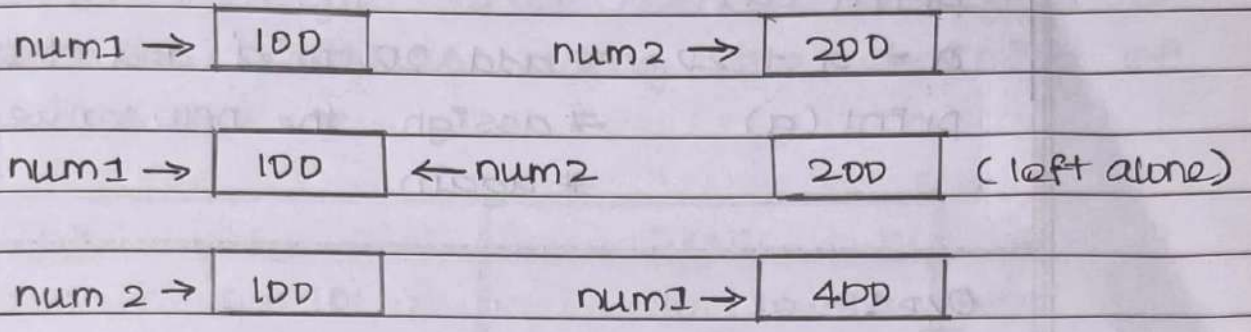
⭐ Variables can be updated to different values within a program.

```
EX:
num1 = 100
num2 = 200
print (num1 , num2)
num2 = num1          # assign num1 → num2
num1 = 400           # update num1
print (num1 , num2)
```

O/P:
100 200
400 100

__/__/____

VISUAL REPRESENTATION:



⊛ The updated values can be of different data types.

```
Ex:
val1 = 10  # an integer
print (val1)
val1 = "Good"  # a string
print (val1)
```

```
O/P:
10
Good
```

⊛ You can assign an expression to a variable and it would automatically convert it to a value.

```
Ex:
  a = 2+4 # automatically added
  print (ex)
```

```
O/P: 6
```

EX :

```
a = 10
print (a)
a = a + 20   # add 20 to 'a' and then
print (a)    # assign the new value to 'a'
             # again.
```

O/P :

```
10
30
```

ASSIGN MULTIPLE VALUES TO MULTIPLE VARIABLES :

* You can assign multiple values to multiple variables in one line. Make sure that the number of variables are equal to the number of values.

EX :

```
a, b, c = 20, 12, 34
print (a)
print (b)
print (c)
```

O/P :

```
20
12
34
```

* If variables or values become more or less it results in error.

EX :

```
a, b = "Morning", "Night", "Evening"  # Error
a, b, c = 1, 2  # Error
```

__/__/___

ASSIGN A SINGLE VALUE TO MULTIPLE VARIABLES:

* A single value can be assigned to multiple variables using the chaining of assignments.

EX:	O/P:
a = b = c = 10 print (a, b, c)	10 10 10

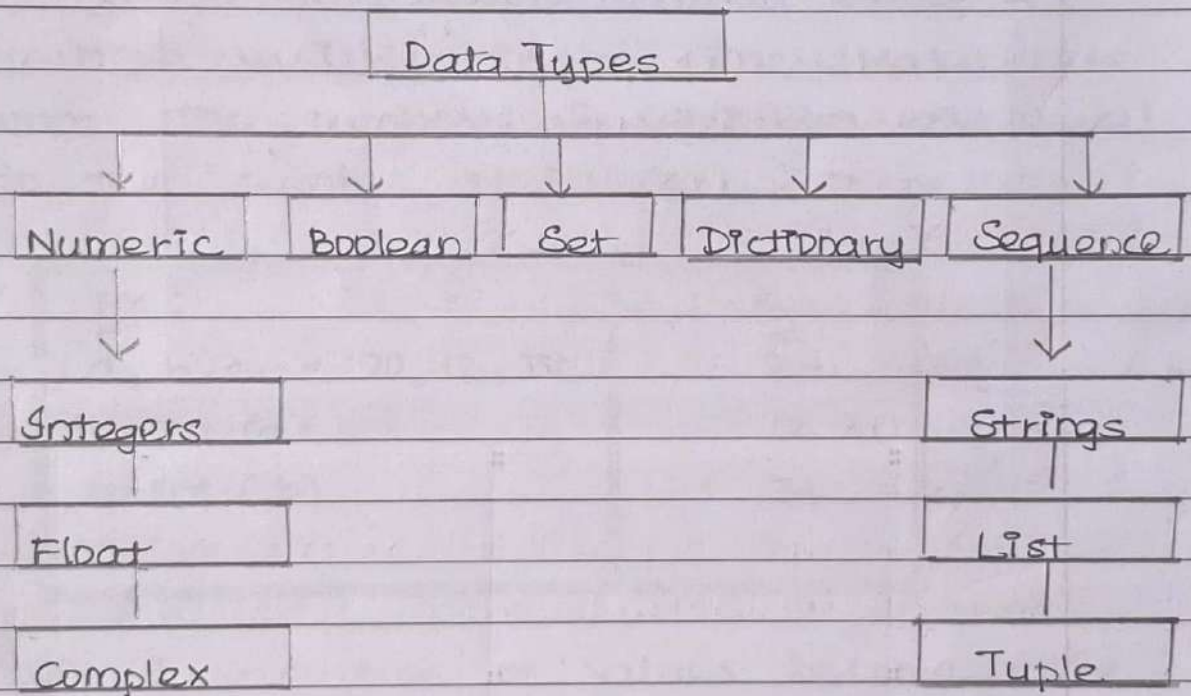
__/__/____

DATA TYPES:

Note: We would discuss about datatypes like string, list, tuple, set and dict in details in upcoming sections.

⊛ A data type is the type of data a value holds. Common data types include Integers, floats, and strings.

⊛ But there are a lot of data types available in Python.



Note: There is also a type called None used for denoting an absent value.

__/__/____

NUMERIC TYPES:

⊛ Numeric data type represents data which contains numeric values (numbers).

⊛ The three types of numeric data types are integers (int), floating point numbers (float) and complex numbers (complex).

INTEGERS:

⊛ Integers contain positive or negative whole numbers (without fraction or decimal).

⊛ In Python, there is no limit to how long an integer value can be. (It just depends on the capacity of your system's memory).

⊛ It is represented as int in Python.

Ex: 12, 100, -9784, 49657485

```
Ex:
    value = 102   # int type
    print (value)
```

```
O/p:
    102
```

FLOATS :

* Floats contain real numbers with floating point representation (specified by a decimal point).

* The maximum value of a floating point number is 1.8×10^{308}. Any number greater than this will be indicated as Inf. (Infinity).

* It is represented as float in Python.

* Floats have 16 digits in precision (the maximum). They can also represent scientific notations using the characters E or e.

EX : 8.35, 4.04, 20.89457.

EX : 1.3E4 (means 1.3×10^4)
2.86e3 (2.86×10^3)
4.5E-5 (4.5×10^{-5})

EX :	O/P :
f = 9.101 print (f) f = 4.3E2 print (f)	9.101 430.0

__/__/____

COMPLEX :

⊛ Complex numbers are represented as complex in Python and are rarely used.

⊛ They have a real and imaginary part and are specified as (real) + (imaginary)j.

EX : 7 + 8j , 2 + 5j , 3j (means 0 + 3j).

EX :	O/P :
c = 2+3j print (c)	2+3j

SEQUENCE TYPES :

⊛ Sequence data type contains an ordered collection of similar or different data types.

⊛ They allow us to store multiple values in an organized and efficient manner.

⊛ The 3 types of sequences are string (str), lists (list), and tuples (tuple).

STRINGS :

⊛ A string is a collection of one or more characters, but in a single quote (or) double quote.

⊛ It is represented as str in Python.

⊛ A string can be defined as arrays of bytes representing Unicode characters.

⊛ Some examples of a string are, 'Hello', "Python", 'A', "40 cookies", "800", '94.86', "$89" etc.

Ex : s = "python $$$ 3.10" print(s)
O/P : Python $$$ 3.10

Note :

⊛ The quotes just mark the beginning and end of a string. They do not get printed in the output.

⊛ A numeric value when represented within quotes becomes a string. Normal arithmetic calculations can't be done with those strings.

Ex :	O/P :
num1 = '10' num2 = '30' print (num1 + num2)	1030

__/__/___

⊛ They get concatenated (joined) instead of getting added. When '+' is used between strings, they are concatenated.

LISTS:

⊛ Lists are an ordered collection of data which can contain multiple values of different data types.

⊛ A list can be created by enclosing values, separated by commas, in square brackets.

⊛ The elements of a list can be modified (mutable) and it allows duplicates.

⊛ It is represented as list in Python.

EX:

[1, 2, 3], ["Hello", 3.45, 8, 100], [] (empty list)

⊛ A list can also contain another list.

EX:

[30, 40, [50, 60, 70], 80, 90] (A list with 4 integers and a list).

[["HI", 908.0], [50, 40]] (A list with 2 lists)

EX :	O/P :
lt = [10, 20, 30, 40, 50] print (lt)	[10, 20, 30, 40, 50]

⭐ A list can contain values of any data type.

TUPLES :

⭐ Tuples are also similar to lists and store multiple values of different types, but the elements within a tuple cannot be modified (immutable).

⭐ A tuple can be created by enclosing values, separated by commas, in parantheses.

⭐ Tuples can be used for small collection of values, separated by commas, in which the values need not to change.

⭐ It is represented as tuple in Python.

EX :

```
(30, 40, 50)
('Hello', 45, 0.4789)
(20, (30, 40), [50, 60], 10)  # a tuple with
                              # 2 int, a tuple
                              # and a list.
(3,)    # tuple with a single value
        # (must have a comma after
        # the value).
```

EX :	O/P :
t = (10, 20, 30) print (t)	(10, 20, 30)

* () is considered as an empty tuple.

DICTIONARIES :

* Dictionaries are a collection of key-value pairs surrounded by curly braces.

* Each pair is separated by a comma and the key and values are separated by a colon.

* It is represented as dict in Python.

* The values of a dict can be modified. (mutable).

Ex :

{ 1 : "Red", 2 : "Blue", 3 : "Green" }

↑

Here, 1, 2 and 3 are keys. Red, Blue, Green are values.

{'India' : 'New Delhi', 'China' : 'Beijing'}

{'Name' : 'Jack', 'ID' : 201846, 'Marks' : [48, 94, 77] }

{ } # empty dict.

__/__/____

Ex :
```
d = {1:10, 2:20, 3:30}
print(d)
```

O/P :
```
{1:10, 2:20, 3:30}
```

* Keys must be Immutable (doesn't allow list, set or dict itself).

SETS :

* Set is an unordered collection of elements that can be modified and has no duplicate elements.

* A set can be created by enclosing values, separated by commas in curly braces.

* A set is mutable and can be modified. It is represented as set in Python.

* Sets are used in situations where a group of values are needed, but their order is not important. They are faster when compared to lists.

__/__/____

EX:
{'Adam', 'Jack', 'Patrick'}
{10, 20, "Hello", 40.45, (1,2)}

EX:
```
s = {'Good','Hi','Morning'}
print(s)
```

O/P:
Hi Good Morning

Note: A set doesn't allow values of mutable types such as list, dict and set itself.

Note: Lists, Tuples, Dict, Set - all are collections that look similar, but they have different properties which make them useful at different situations.

BOOLEANS:

⊛ A boolean consists of the two built-in values, True and False.

⊛ Boolean values equal to True are truthy (true), and those equal to False are falsy (false).

⊛ It is represented as bool in Python.

__/__/____

⊛ The bool type is a subclass of int type. When True and False are used in arithmetic operations, they take one and zero (1 and 0) as their values respectively.

Ex :

True , False

Ex : b = True print (b)	O/P : True

Ex : b = True + True # Arithmetic. print (b)
O/P : 2

NONE :

⊛ None is a value used to denote that a value is absent. It just denotes nothing.

⊛ None is always less than any numbers.

Ex : a = None print (a)	O/P : None

__/__/____

MUTABLE AND IMMUTABLE DATA TYPES:

⊛ An object is called mutable if it can be changed. An immutable object cannot be modified.

⊛ Examples of mutable data types include list, set, and dict.

⊛ Examples of immutable data types include int, float, complex, str and Tuple.

TESTING THE TYPE OF VARIABLES:

⊛ In Python, we can check the data type of an object using the built-in function type().

EX:	O/P:
a = 10 print(type(a))	<class 'int'>
EX:	**O/P:**
a = '247' print(type(a))	<class 'str'>

⊛ Similarly, you can check the data type of any variable or value.

__/__/____

⊛ The isinstance() function checks whether a variable or a value is of the given type. If yes, it returns True, else False.

Ex:

```
c = '123'
print (isinstance (c, str))
print (isinstance ('20', int))
```

O/P:

```
True
False
```

TYPE CONVERSION:

⊛ Type conversion is the process of converting one data type to another.

⊛ There are two types of type conversion in Python.

1) Implicit Type Conversion

2) Explicit Type Conversion

⊛ Let us see both these types in detail.

__/__/____

IMPLICIT TYPE CONVERSION:

⊛ In Implicit type conversion, the Python Interpreter automatically converts one data type to another without any user involvement.

⊛ This is also called type promotion.

EX:

```
num1 = 3   # Integer
num2 = 4.5 # float
num3 = num1 + num2
print (type (num3))
```

O/P:

```
<class 'float'>
```

⊛ Here, though we add Int and float numbers, it finally converts the value to a float (wider sized data type) automatically to avoid loss of data.

EXPLICIT TYPE CONVERSION:

⊛ In explicit type conversion, the data type is manually changed by the user as per requirement.

⊛ With explicit conversion, there is a risk of data loss, since we manually force an expression to be converted to another data type.

__/__/____

⊛ There are a lot of built-in functions available in the name of datatypes which can be used to convert a value to its datatype (int(), str(), float(), list() etc.).

Ex:

```
s = "80"
a = 40
b = a + int(s)  # converts s to int
print(b)
```

O/P: 120

Ex:

```
age = 24
print("My age is " + str(age))
```

O/P:

My age is 24

⊛ In the above program, the variable age is converted to a string before concatenation because a string and an integer can't be concatenated.

Ex :

```
s = 'Hello'
print (list(s))
```

O/P :

```
['H', 'e', 'l', 'l', 'o']
```

* Converting a string to a list separates all the characters and stores it in a list.

* Explicit type conversion from one type to another takes place only with valid values. Trying to convert invalid values would result in an error.

Ex :

```
s = 'Hello'
print (int(s))
```

O/P :

```
Value Error : Invalid literal for int().
```

* Similarly, only certain types can be converted to another. Trying the wrong type would result in error.

__/__/____

GETTING USER INPUT:

⊛ Developers often have a need to interact with users to get certain details.

⊛ To get an input from the user, Python uses a built-in function called input().

⊛ The input() function, when called, stops the program execution and waits for the user's input. When the user presses Enter, the program resumes and returns what the user typed.

⊛ The input typed is returned as a string. If you need a value of any other type, you must explicitly convert it.

Ex:

```
name = input()
print("Good Morning", name)
```

O/P:

```
Chep → (input by user)
Good Morning Chep
```

⊛ Asking for an input without a prompt would result in the user getting confused. Thus the input() function can have a prompt displayed.

__/__/__

Ex :

```
name = input("Enter a word:")
print("Hello", word)
```

O/P :

```
Enter a word : Chip
Hello Chip
```

Ex :

```
num1 = int(input("Enter a no. :"))
num2 = int(input("Enter an other no.:"))
print(num1 + num2)
```

O/P :

```
Enter a no. : 20
Enter an other no. : 30
50
```

⭐ The above program converts the input to integers before assigning it to the variables.

__/__/____

OPERATORS:

⊛ Operators are symbols used to perform operations on values and variables.

⊛ Operands are the values on which the operator is applied. (In a + b, a and b are operands).

⊛ There are different types of operators in Python.

ARITHMETIC OPERATORS:

⊛ Arithmetic operators are used to perform mathematical operations like addition, subtraction etc.

⊛ The operators are:

Operator	Definition	Example
+	Addition	x + y
-	Subtraction	x - y
*	Multiplication	x * y
/	Float Division	x / y
//	Floor Division	x // y
%	Modulus (Remainder)	x % y
**	Power	x ** y

__/__/____

Note: The float division divides and returns the result as a float value. The floor division returns the result as an integer after a floor rounding.

Ex:	O/P:
p = 7	9
q = 2	5
print (p+q)	14
print (p-q)	3.5
print (p* q)	3
print (p/q)	1
print (p // q)	49
print (p %. q)	
print (p** q)	

COMPARISON OPERATORS:

⭐ Comparison operators compares values and returns either True or False as a result based on the condition.

⭐ The operators are as follows:

Operators	Definition	Examples
>	Greater than	x > y
<	Less than	x < y
==	Equal to	x == y

__/__/___

!=	Not Equal To	x != y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

Note: = is used for assigning values.
== is used for checking if two values are equal.

Ex:	O/p:
k = 9	False
l = 14	True
print (k > l)	False
print (k < l)	True
print (k == l)	False
print (k != l)	True
print (k >= l)	
print (k <= l)	

* The comparison operators are also called as the relational operators.

LOGICAL OPERATORS:

⊛ Logical operators perform logical AND, logical OR, and logical NOT operations.

⊛ They are mostly used in combining two or more relational expressions.

⊛ The operators are as follows:

Operators	Definition	Example
and	True if both operands are True	x and y
or	True if either of the operands are True.	x or y
not	True if operand is False	not x

EX:

(3==3) and (7!=7) # True

(2>3) or (5==5) # True

(not (8==8)) # False

⊛ These operators work as per the conditions given.

__/__/____

BITWISE OPERATORS:

⊛ Bitwise operators act on bits and perform bit-by-bit operations. They operate on binary numbers.

⊛ These are mostly used when working with device drivers, low-level graphics, cryptography, and I/O communications.

⊛ The operators are as follows:

Operator	Definition	Example
&	Bitwise AND	x & y
|	Bitwise OR	x | y
~	Bitwise NOT	~ x
^	Bitwise XOR	x ^ y
>>	Bitwise right shift	x >>
<<	Bitwise left shift	x <<

⊛ Bitwise operators are rarely used in common applications and programs.

__/__/____

EX :

10 & 4 (Binary equivalent - 1010 & 0100)

1010
0100 = 0 (Perform & from units place).
0000

* Similarly, all operations take place at bit-level.

ASSIGNMENT OPERATORS :

* Assignment operators are used to assign values to the variables. The operators are as follows:

Operator	Definition	Example
=	Assign value of right side to variable in the left	x = y
+=	Add right-side operand to left side and assign the final value to left.	x += y (x = x+y)

* There are a lot of assignment operators in similar fashion. They are,

-=, *=, /=, %=, //=, **=, &=, |=, ^=, >>=, <<=

__/__/____

Ex:

```
a = 10      # Assignment
b = a       # Assignment
print(b)
b += a      # Add and assign (b = b+a)
print(b)
b *= a      # Multiply and assign (b = b*a)
print(b)
```

O/P:

```
10
20
200
```

IDENTITY OPERATORS:

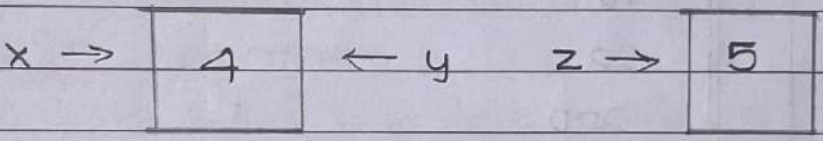
⭐ Identity operators are used to check if two variables or values share the same memory location, and are of same data type.

⭐ The operators are as follows:

Operator	Definition	Example
is	Evaluates to True if both values are same.	x is y
is not	Evaluates to True if both values are not same.	x is not y

Ex:	O/P:
x = 4 y = 4 z = 5 print (x is y) print (x is not z) print (y is z)	True True False

VISUAL REPRESENTATION:



(x and y are same)

MEMBERSHIP OPERATORS:

★ Membership operators are used to check or validate the membership of a value.

★ It tests the membership in a sequence, such as a string, list or tuple. The operators are as follows:

Operator	Definition	Example
in	Evaluates to True if an element exists in a sequence.	x in y
not in	Evaluates to True if an element does not exist in a sequence.	x not in y

__/__/____

Ex :	O/P:
s = "hello"	True
lt = [1, 2, 3, 4]	True
print ('h' in s)	True
print (3 in lt)	False
print (7 not in lt)	
print ('e' not in s)	

OPERATOR PRECEDENCE :

⊛ The combination of values, variables, operators and function calls is called an expression.

⊛ There can be more than one operator in an expression. The operators are evaluated based on the precedence table.

Ex :

2 + 7 * 3

→ 2 + 21 and not → 9 * 3

→ 23 (result) → 27 (wrong)

⊛ This is because * has more precedence than +. If you want + to be executed first, use a paranthesis as it has the highest precedence.

Ex :

(2+7) * 3

→ 9 * 3

→ 27 (result)

__/__/____

PRECEDENCE TABLE :

Operators	Meaning
()	Parantheses
**	Exponent (Power)
+x, -x, ~x	Unary plus, Unary minus, Bitwise NOT.
*, /, //, %.	Multiplication, Division, Floor division, Modulus.
+, -	Addition, Subtraction.
<<, >>	Bitwise shift operators
&	Bitwise AND
^	Bitwise XOR
|	Bitwise OR
==, !=, >, >=, <, <=, is, is not, in, not in	Comparison, Identity and Membership operators.
not	Logical NOT
and	Logical AND
or	Logical OR

__/__/____

* The precedence table is written in the order of highest to lowest precedence.

OPERATOR ASSOCIATIVITY:

* When two operators of the same precedence occur associativity determines whether to evaluate from left to right or from right to left.

* Almost all operators have left to right associativity.

```
EX:
print (4 * 2 // 3)  # left to right.

→ 8 // 3
→ 2 (Output)
```

* Here, * and // have same precedence, and are executed from left to right.

* If you want // to execute first, use a bracket.

```
4 * (2 // 3)

→ 4 * 0
→ 0 (Output)
```

* The exponent operator has right-to-left associativity in Python.

__/__/____

Ex:

```
Print (2**3**2)   # Right to left

→ 2**9
→ 512
```

* Use brackets to change the order.

__/__/____

BLOCKS AND INDENTATION:

⊛ A block of code is often used in Python along with concepts like control structures, functions and classes.

⊛ Before knowing about these concepts, it is important to know about blocks and indentation.

Ex:

```
if (a > b):
    print ("a is greater")
    print ("Thank you")
```

→ Block of code which is part of the if statement.

Note: Do not try to understand the if statement. We would talk about them soon. Just notice the block of code.

⊛ Languages like C, C++ use curly braces ({}) to define a block of code.

⊛ But Python uses colon (:) to specify the start of a block and indentation (4 spaces recommended) to specify the statements in a block.

⊛ Anything written with the same indentation of the block's header (i.e, if) is considered to be out of the block.

C - Style :

```
if (a > b)
{                          → start of block
   statement 1;
   statement 2;            → Block of code.
}                          → End of block
```

Python Style :

```
if (a > b):                → start of block
    statement 1
    statement 2            → Block of code
    =
statement                  → out of block
```

⊛ Note the indentation (leaving 4 spaces) to write the statements within the if block.

IMPORTANT POINTS ABOUT INDENTATION :

⊛ It is not mandatory to have 4 spaces as the indentation, but is highly recommended.

⊛ You can use any number of spaces you want, but it must be same throughout the block. Else it shows indentation error.

Ex :

```
if (a > b):
    stmt 1      # block with 4 spaces
    stmt 2      # indentation

if (c > d):
    stmt 1      # block with 2 spaces
    stmt 2      # indentation.
```

* In the above example, two different blocks have two different indent spaces.

Ex :

```
if (p >= q):
    statement 1      # same block has
        statement 2  # statements with
                     # diff. indentation.
```

* The above example shows an error.

As said before, though there are options to use different indentations, it is still best to stick to 4 spaces throughout the program.

Note : Statements like if-elif-else, match-case, for, while, functions and classes use the concept of blocks and indentation.

__/__/____

CONDITIONAL STATEMENTS :

⊛ In programming, there are situations where we need to make some decisions whether to execute a block of code or not.

⊛ The decision is made based on the condition given which results to either True or False.

⊛ These decision making statements decide the direction of the flow of program execution.

⊛ In Python, if-elif-else is used for decision making.

THE IF STATEMENT :

⊛ The if statement is the most simple decision-making statement. A block of statements would be executed if the condition given is True. If False, the block is skipped.

SYNTAX :

```
if condition:
    # This block is executed
    # if condition is true.
```

EX :

```
a = int (input ("Enter a number :"))
if a>5 :
    print (" You entered a no. > 5")
print ("Thank You") # normal statement.
```

O/P :

```
Enter a number : 10
You entered a no. > 5
Thank You
```

Alternate O/P :

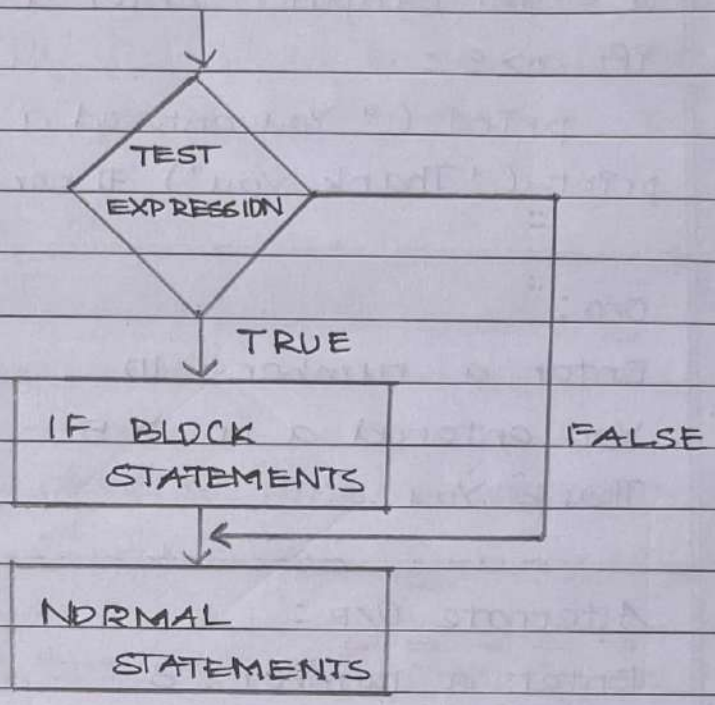
```
Enter a number : 3
Thank You
```

⊛ The above program is executed with two different inputs. In the first case, the condition becomes true and the if block is executed. But in the second case, the condition becomes false and the block is skipped.

Note : As already said, Python uses indentation to define a block of code. Also, the condition used in if can be written with or without paranthesis.

⊛ Let us now look at the flowchart of a if statement.

FLOWCHART OF IF STATEMENT:



THE 'IF-ELSE' STATEMENT:

⊛ The if block only executes when the condition is true. But if we need to execute certain statements when the condition is false, we use the else block.

⊛ The else block is an alternate to the if block which executes only when the condition of the if block becomes false.

⊛ The else block doesn't have any condition.

__/__/____

SYNTAX :

```
if condition :
    statements
else :
    statements
```

EX :

```
age = int ( input ("Enter your age:"))
if (age >= 18):
    print ("Eligible to drive")
    print ("Apply for license")
else:
    print ("Ineligible to drive")
    print ("Try after turning 18")
print ("Thank you")
```

O/P :

```
Enter your age: 16
Ineligible to drive
Try after turning 18
Thank You
```

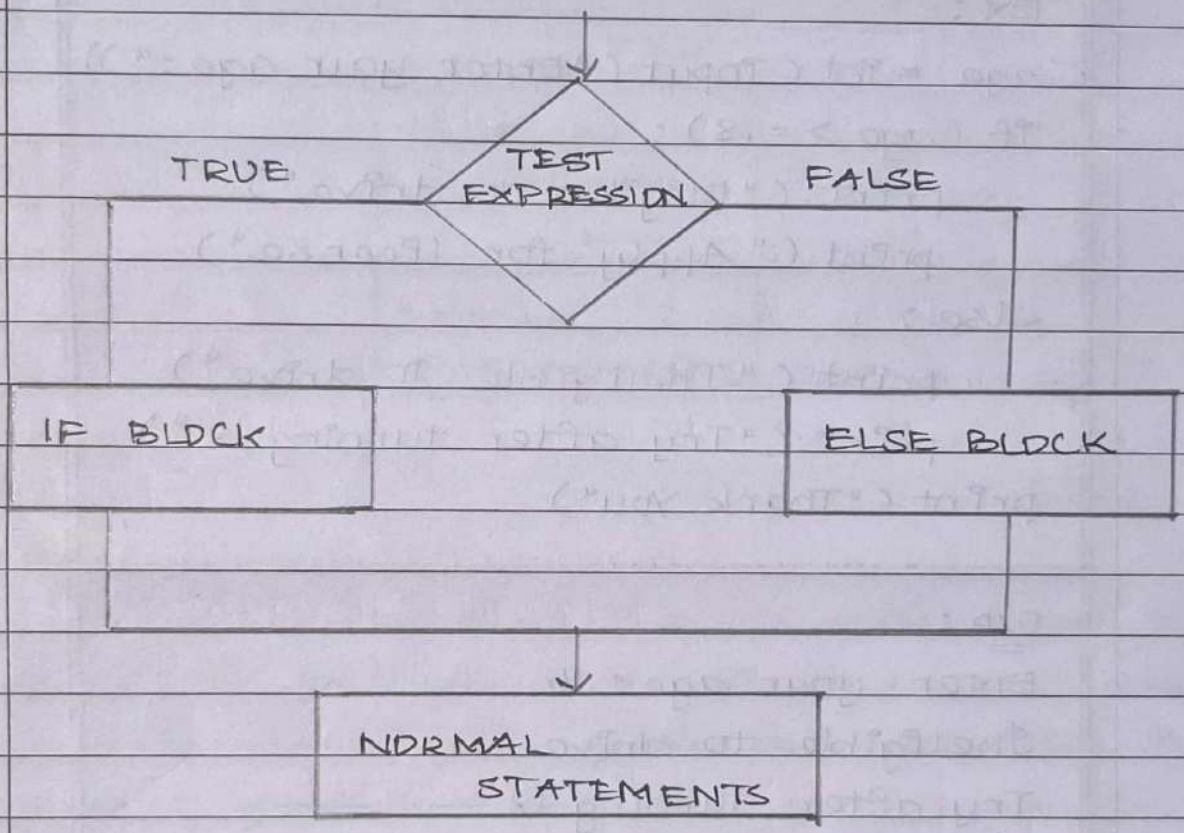
Alternate O/P :

```
Enter your age : 23
Eligible to drive
Apply for license
Thank You
```

⊛ From the above program, you can see that the output changes based on the given condition.

* In the first case, the else block executes as the condition is false. In the second case, the if block is executed as the condition is true.

IF-ELSE FLOWCHART:



THE IF-ELIF-ELSE LADDER:

* The if-else block is used in alternate scenarios. If there are more than two alternates, the if-elif-else block is used.

* The ladder starts with an if block, then we write as many elif blocks needed as per requirement. Finally, the else block is written.

__/__/____

⊛ The elif block also has a condition. If the 'if' block condition becomes false, it checks the condition of the next elif block. If it is true, the block is executed, and all other blocks are skipped.

⊛ If neither of the conditions are true, then the else block is executed.

SYNTAX:

```
if condition:
    # if block
elif condition:
    # elif block
elif condition:
    # another elif
else:
    # else block
```

⊛ The if-elif-else can also be written without an else block.

EX:

```
per = float(input("Enter your percentage:"))
if (per > 90):
    print("Grade A")
elif (per > 60):
    print("Grade B")
elif (per > 40):
    print("Grade C")
else:
    print("Better luck next time")
```

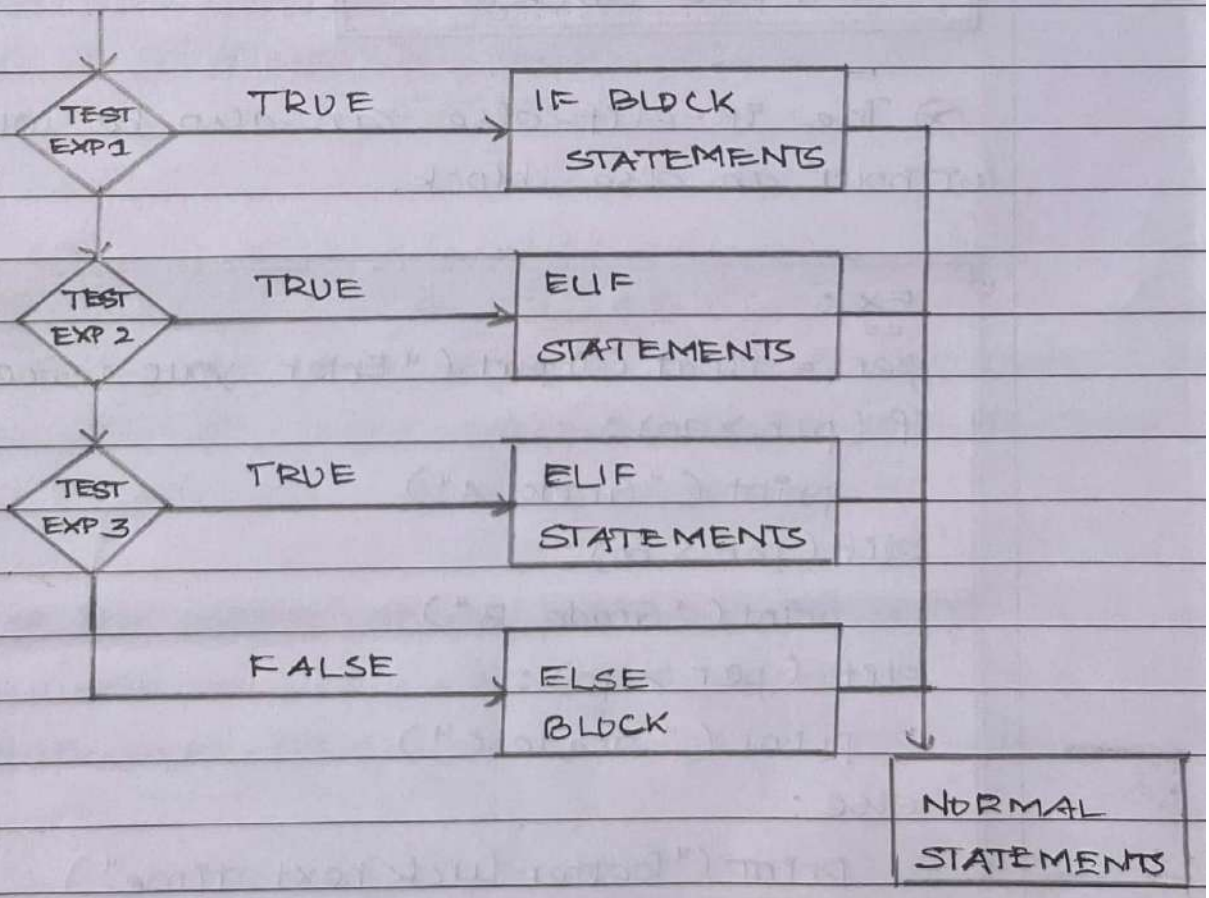
O/P:
Enter your percentage : 90.3
Grade A

Alternate O/P:
Enter your percentage : 46
Grade C

⊛ In the first scenario, the if block is executed and thus all other blocks are skipped.

⊛ In the second case, the first two conditions become false, and the second elif block is executed. The else block is skipped.

FLOWCHART - IF ELIF ELSE:



__/__/____

* The elif and else blocks do not exist without an if block.

PROBLEM IN ORDER OF CONDITIONS:

* We must make sure that the order of conditions in the if-elif-else ladder are correct.

* At times, we may not get the intended output due to change in order of conditions.

* Consider the if-elif-else example where we get the grade based on the percentage. Let me change the order of conditions.

```
per = float(input("Enter your percentage:"))
if (per > 40):
    print("Grade C")
elif (per > 60):
    print("Grade B")
elif (per > 90):
    print("Grade A")
else:
    print("Better luck next time")
```

O/P:

```
Enter your percentage : 83
Grade C
```

__/__/____

⊛ We expected Grade B, but got Grade C. That's because, the if condition becomes true for the input. Though other conditions are also true, the first block is executed based on the order.

⊛ To solve this problem, a better option would be to provide proper conditions.

Ex:
Instead of,
per > 90, we give, (per > 90 and per <= 100)

Similarly,
(per > 80 and per <= 90) etc.

⊛ Try the same program with these conditions and you will get the output, no matter whatever the order is.

THE NESTED IF STATEMENT:

⊛ An if statement can be nested. i.e) An if statement can contain another if statement. Not only if, but also elif and else statement can have nested if-elif-else.

⊛ This can be used in situations where there are multiple conditions to check.

__/__/____

SYNTAX :

```
if condition :
    statements
    if condition :
        statements
        =
    =
```

→ INNER IF BLOCK

→ OUTER IF BLOCK

Ex :

Program to choose if a restraunt is AC/Non AC, and Veg / Non-veg.

```
ac = Input ("Enter AC / Non-AC")
food = Input ("Enter Veg / Non-Veg")
if (ac == "AC") :
    if (food == "Veg")
        print ("AC Veg")
    else :
        print ("AC Non-veg")
else :
    if (food == "Veg")
        print ("Non-AC Veg")
    else :
        print ("Non-AC Non-Veg")
```

O/P :

```
Enter AC / Non-AC : Non-AC
Enter Veg / Non-Veg : Non-Veg
Non-AC Non-Veg
```

__/__/____

⊛ In this program, when the outer if is false, it skips the whole block and executes the else block.

⊛ Within the else, the condition of if is checked, and when it is also false, the else block within the outer else gets executed.

Note: It is better to write the same program with if-elif-else ladder than using nested if so that the code looks neat.

```
Ex:
if (ac == "AC" and food == "Veg"):
    print ("AC Veg")
elif (ac == "AC" and food == "Non-Veg"):
    print ("AC Non-Veg")
elif (ac == "Non-AC" and food == "Veg"):
    print ("Non-Ac Veg")
elif (ac == "Non-AC" and food == "Non-Veg"):
    print ("Non-AC Non-Veg")
```

SHORT-HAND IF STATEMENT:

⊛ If there is only one statement within the if block, it can be written in the same line as the if statement.

```
SYNTAX:
if condition : statement
```

__/__/____

EX:	
a = 10	a = 10
If a>3:	If a>3: print(a)
print (a)	↑
print ("Done")	SHORT HAND IF

O/P:
10

⊛ When there are more than one statements, this method won't work.

SHORT-HAND IF-ELSE STATEMENT:

⊛ If there is only one statement in both if and else block, they can be combined in one single line.

⊛ It is also called the ternary operator in Python.

SYNTAX:
statement if condition else statement

EX: (NORMAL)

```
drink = input("Coffee/Tea")
if (drink == "Coffee"):
    print ("Coffee")
else:
    print ("Tea")
```

(This can be written in short hand)

EX : (SHORT HAND)

```
drink = input ("Coffee / Tea")
print ("Coffee") if drink == "Coffee" else
                                print ("Tea")
```

O/P :

Coffee / Tea : Tea

Tea

THE PASS STATEMENT :

⊛ A block in Python must definitely contain at least one statement. If not it would show an error.

⊛ There may be scenarios where you would start a block and define it after sometime. To avoid an empty block, you can use the pass statement.

⊛ The pass statement does absolutely nothing and is just used in situations like this.

EX :

WITHOUT PASS

```
If (a>b):
print ("Done")
```

O/P : Error

WITH PASS

```
If (a>b):
    pass
print ("Done")
```

O/P : Thanks (Done)

__/__/____

MATCH CASE:

⊛ Another alternative for If-else was introduced in Python 3.10 called the match case.

⊛ In a match-case, the match statement will compare a given variable's value to a pattern until a given pattern matches and the block executes.

⊛ Using match-case makes the code more readable and manageable.

SYNTAX:

```
match variable_name:
    case <pattern 1>:
        statements
    case <pattern 2>:
        statements
    case <pattern n>:
        statements
```

Ex:

```
num = int(input("Enter 1,2 or 3:"))
match num:
    case 1:
        print("You entered 1")
    case 2:
        print("You entered 2")
    case 3:
        print("You entered 3")
    case _:
        print("Please enter 1,2 or 3")
```

O/P:
Enter 1, 2 or 3 : 1
You entered 1

Note: The underscore case is called the wildcard pattern. When none of the patterns are matched the wildcard case gets executed.

USING MULTIPLE PATTERN VALUES:

⊛ Match case can use multiple pattern values in a single case using the OR operator (|).

EX:

```
sample = Input ("Enter good or bad")
match sample:
    case ("good" | "bad"):
        print ("You entered good or bad")
    case _:
        print ("You entered something else")
```

O/P:
Enter good or bad : bad
You entered good or bad

USING A GUARD:

⊛ We can also add an if statement to a pattern within match case. That condition is known as the guard.

⊛ The expression for a given pattern would be evaluated only if the guard is True.

__/__/____

Ex:

```
n = int (input ("Enter a number:"))
match n:
    case n if n<0:
        print ("Number is -ve")
    case n if n==0:
        print ("Number is 0")
    case n if n>0:
        print ("Number is +ve")
```

O/P:

```
Enter a number : 0
Number is 0
```

Note:

* The match-case statement doesn't work upto Python version 3.9.

* Match case is computationally faster than if else. When the number of conditions are large it is best to use match case.

__/__/____

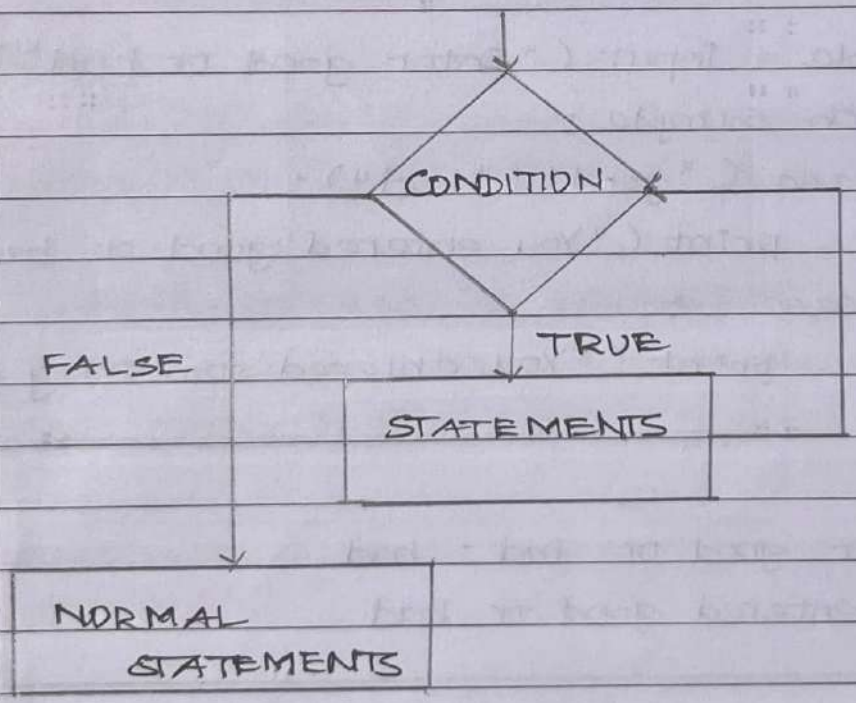
LOOPS:

⊛ There may arise situations where a block of code needs to repeated a number of times or until a condition is false. These are called loops.

⊛ Looping (or) Iterating can be done in Python using while and for statements.

⊛ Conditional and looping statements are highly used in writing logical programs.

FLOWCHART:



__/__/____

WHILE LOOP:

⊛ The while loop can execute a set of statements as long as the condition is true.

⊛ Make sure to initialize the required variables before using them in the condition and also increment them within the loop to avoid an infinite loop.

SYNTAX:

```
while condition:
    statements
    =
normal statements
```

EX:

```
a = 0
while a < 5:    # print from 0 to 4
    print(a)
    a = a+1
```

O/P:

```
0
1
2
3
4
```

⊛ From the above program, you get to know that the loop executes until 'a' is less than 5. Once the condition becomes false, it gets out of the loop and starts executing the other

statements present, if any.

INFINITE LOOP :

⊛ There may be situations where the while statement may loop infinite times due to the condition being always true.

⊛ Let me take the example above and remove the increment statement.

```
a = 0
while a < 5:
    print(a)
```

O/P : (Infinite Loop)

```
0
0
0
≡
```

⊛ The above program when executed doesn't stop and keeps on printing zero as it is never incremented and the condition never becomes false.

⊛ If you run this program and want to stop manually, use Ctrl + C to pass a keyboard interrupt and end the program.

__/__/____

FOR LOOP :

⊛ A for loop in Python is somewhat different compared to other programming languages.

⊛ A for loop acts as an iterator which iterates or loops over a sequence of values. (String, list, tuple, dict or set).

⊛ A for loop does not require a variable to be initialized before hand or updated within the block like in the while loop.

SYNTAX :

```
for variable in sequence:
        statement(s)
        ≡
normal statements
```

EX:	O/P:
S = "Cake"	C
for i in S:	a
print (i)	k
print ("Done")	e
	Done

ANOTHER EX:	O/P:
li = [1, 2, 'Hi', 10.5]	1
for ele in li:	2
print (ele)	Hi
	10.5

__/__/____

⊛ Here, the variable name holds one value at a time from the collection, and it automatically updates to the next value in the next loop.

THE RANGE() FUNCTION:

⊛ The range() function is often used along with the for loop to iterate through a block a specified number of times.

⊛ The range() function returns a sequence of numbers starting from zero by default, and increments by 1 (by default), and ends at the specified number.

⊛ We use the range function instead of the collections used.

SYNTAX:
range(start, stop, step)

Note: You can only mention the stop value and the other two are optional.

EX:	O/P:
for i in range(5): print(i)	0 1 2 3 4

__/__/____

* In the above example, we have specified the end value within the range function. It generates values from 0 to 4 and not from 0 to 5. (end value is not counted).

USING START AND END:

```
for I in range (3,7):
    print (I)
```

O/P:

```
3
4
5
6
```

USING START, END AND STEP:

```
for I in range (0,10,2):
    print (I)
```

O/P:

```
0
2
4
6
8
```

NESTED LOOPS:

⊛ A nested loop is a loop inside a loop. The inner loop will be executed completely for every iteration of the outer loop.

```
Ex:
a = [0, 1, 2]
b = [3, 4, 5]
for i in a:
    for j in b:
        print(i, j)
```

```
O/P:
0 3
0 4
0 5
1 3
1 4
1 5
2 3
2 4
2 5
```

LOOP CONTROL STATEMENTS:

⊛ Loop control statements are used to control the flow of loop in Python.

⊛ These are nothing but the break and continue statements. These statements can stop or change the flow of the loop.

__/__/____

⊛ They can be used with both for and while statements.

THE BREAK STATEMENT:

⊛ The break statement is used to stop and exit the loop even when the condition is true.

EX:	O/P:
a = 1	1
while a<=5:	2
print (a)	
a = a+1	
if a == 3:	
break	

⊛ In the above example, though the loop is true until a=5, it prints only till 2.

⊛ Then the break statement is encountered once the value becomes 3 and the if statement becomes true. Thus it exits the loop.

THE CONTINUE STATEMENT:

⊛ The continue statement is used to stop the current iteration and continue with the next.

Ex:	O/P:
for i in range(6):	0
if i == 3:	1
continue	2
print(i)	4
	5

⊛ Note that the value 3 is skipped and all the other values are printed.

THE ELSE STATEMENT:

⊛ An else statement can be used with the while loop and for loop as well.

⊛ The else block would be executed once, only when the condition is no longer true.

⊛ If the loop stops due to a break statem-ent, the else block doesn't execute.

Ex:	O/P:
li = [2,4,6,8]	2
for i in li:	4
print(i)	6
else:	8
print("Done")	Done

⊛ You can see that it loops through the list completely and once it completes, the else block executes.

__/__/____

AN EXAMPLE WITH BREAK:	O/P:

li = [2, 4, 6, 8]
for i in li:
  if i==6:
  break
  print(i)
else:
  print("Done")
	2
	4

* Note that the else block is not executed as the for loop has encountered break.

THE PASS STATEMENT:

* As already said, a pass statement can be used when a block is needed, but you do nothing with it.

* The pass statement can be used with a while as well as for loop.

Ex:	Ex:

for i in range(10):
  pass
	while a>5:
	  pass

* You can use these when you would write a block of code afterwards and just need to write the start of a block.

__/__/____

STRINGS:

⊛ Strings are a sequence of characters surrounded by single or double quotes.

⊛ Ex : "hello" , 'hi'

⊛ Each character in a string can be represented with its index.

Ex : The string "Hello World" can be represented with its index. as memory blocks.

0	1	2	3	4	5	6	7	8	9	10
H	e	l	l	o		W	o	r	l	d
-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

⊛ Note that each character is given an index starting from zero.

⊛ The character in a string can also be represented with a -ve index starting from the end with -1.

⊛ A space is also given an index here. Any character (an alphabet, number, symbol or white space) has an index and can be accessed.

__/__/____

INDEXING:

⊛ A particular character from a string can be accessed using the index. That is called Indexing.

Ex:	O/P:
s = "Hello World"	Hello World
print (s)	H
print (s[0])	W
print (s[6])	

⊛ Negative Indexing can also be used in accessing characters from a string.

Ex:	O/P:
s = "Python"	n
c = s[-1]	
print (c)	

⊛ The Index must be within the range. Using an index greater than the maximum Index would result in an Index error.

Ex:	O/P:
s = "Python"	IndexError
print (s[6])	

⊛ Here, the Index 6 is not in existence. There is only index number 5 which is the last character.

__/__/____

SLICING:

⊛ A range of characters (substring) can be accessed using slicing.

⊛ We just specify the start and end index. of the substring we want.

SYNTAX:

string [start : end]

⊛ The start index is inclusive and the end index is exclusive (not included) while slicing a string.

EX:

Consider the string 'programming'.

0	1	2	3	4	5	6	7	8	9	10
p	r	o	g	r	a	m	m	i	n	g
-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

EX:	O/P:
s = 'programming' print(s) print(s[2:6])	programming ogra

Note: It considers only the string from index 2 to 5 and doesn't take 6.

⊛ The start value must always be less than the end value.

__/__/____

⊛ If you leave the start index, it would start from the first character (index 0) by default.

⊛ When you don't specify the end index, it would end at the last character by default.

Ex:	O/P:
print (s[:4])	prog
print (s[7:])	ming
print (s[:])	programming

⊛ You can also use -ve indexing in slicing.

Ex:	O/P:
print (s[-7:-4])	ram

⊛ A third optional value can be given as the step value. By default the step value is 1.

Ex:	O/P:
print (s[1:7:2])	rga
print (s[::-1])	gnimmargorp
print (s[6:1:-2])	mro

⊛ Note that the second and third slice has a -ve step value which slices the string in reverse. Make sure that start value is bigger than the end value in this case.

__/__/____

IMMUTABILITY OF STRINGS :

⊛ Strings are immutable which means, that the sequence of characters in it cannot be modified.

For ex :

```
s = "Hello"
s[0] = "J"   # error
```

⊛ Here, we try to modify the character H with J. But that doesn't happen. But reassignment of a string is possible.

Ex :

```
s = "Hello"
print(s)
s = "Jello"   # reassigning variable s
print(s)
```

O/P :

```
Hello
Jello
```

⊛ Here, Hello is not replaced by Jello, but it is a completely new string in a different memory location.

⊛ This proves that strings are immutable.

__/__/____

MULTILINE STRINGS:

⊛ Writing a multiline string is possible with the help of triple quotes (''' or """)

Ex :	O/P :
String = """ Hello! This is a Python string.""""	Hello! This is a Python string.

⊛ Multiline strings are printed along with the line breaks and spaces that we provide.

⊛ They are mostly used as docstrings (used for documentation of code) at the beginning of a function.

STRING CONCATENATION AND REPETITION:

⊛ A string can be concatenated (joined) and repeated using the + and * operators.

Ex :	O/P :
s1 = 'Hello' s2 = 'World' print (s1 + s2)	HelloWorld
Ex : s = "Apple" print (s*3)	O/P : AppleAppleApple

__/__/____

LENGTH OF A STRING:

⊛ The length of a string can be found using the built-in function len().

EX:	O/P:
s = "Python" print (len(s))	6

Note: The len() function can also be used to find the length of a list, tuple, set and dictionary.

STRING MEMBERSHIP:

⊛ To check if a character(s) is present or not in a string, we can use the 'in' and 'not in' operators.

```
EX:
s = "Windows"
print ("W" in s)      # True
print ("do" in s)     # True
print ("r" in s)      # False
print ("r" not in s)  # True
```

STRING METHODS:

⊛ There are a lot of methods (methods are similar to functions) available to use with a string.

⊛ All string methods return new values and do not modify or change the original string.

__/__/__

⊛ Some important and mostly used methods are demonstrated here:

METHODS	DEFINITION
count()	Returns the number of times a specified value occurs in a string.
format()	Formats specified values in a string.
index()	Searches the string for a specified value and returns the position of where it was found. (only first occurence by default).
join()	Joins the elements of an iterable to the end of the string.
replace()	Returns a string where a specified value is replaced with another value.
split()	Splits the string at the specified separator and returns a list.
strip()	Returns a trimmed version of the string. (removes whitespaces from start and end.)
upper()	Converts a string into uppercase.
lower()	Converts a string into lowercase.

__/__/____

Ex:

```
s = "Coding"
print(s.upper())
print(s.lower())
s = "This is a program."
print(s.count("i"))
print(s.index("s"))
print(s.replace('is','was'))
print(s.split(" "))
s = 'Hello'
print(s.strip())
li = ['Python','is','awesome']
print(' '.join(li))
```

O/P:

```
CODING
coding
2
3
Thwas was a program.
['This','is','a','program']
Hello
Python is awesome
```

Note: The methods are used only with a string using the (.) dot operator.

__/__/____

STRING FORMATTING:

⊛ String formatting is the process of adding and combining things into a string dynamically.

⊛ It can be done using the format() method or by using f-strings.

THE FORMAT() METHOD:

Ex:

```
name = "Jake"
age = 27
result = "The name is {} and age is {}."
print(result.format(name, age))
```

O/P:

```
The name is Jake and age is 27.
```

⊛ Here, the {} acts as a placeholder for the variables used. With this you can easily inflese the variables instead of concatenating the strings and variables.

F-STRINGS:

⊛ The f-strings are more easier to format and was introduced from Python 3.6.

⊛ We just prefix 'f' before the string to change it to an f-string.

__/__/____

Ex :

```
name = "Ahmed"
age = 23
print ( f'Name is {name} and age is {age}.')
```

O/P :

```
Name is Ahmed and age is 23.
```

⊛ You can also directly use methods within the placeholders.

```
name = "JACK"
print ( f'My name is {name.upper()}')
```

O/P :

```
My name is JACK
```

ESCAPE CHARACTERS :

⊛ Some characters have a special meaning in Python and can't be printed using a string. (Ex : ', ", \ etc.)

⊛ These characters can be printed using a prefix of backslash (\).

Ex :

```
print ("He asked me, \"How are you?\" ")
```

O/P :

```
He asked me, "How are you?"
```

__/__/____

Ex :
```
print("The path is C:\\Desktop\\Files")
```

O/P :
```
The path is C:\Desktop\Files
```

⊛ Also, we can use special characters like \n (for newline) and \t (for tabs).

Ex :
```
print("Name \t Age \t Class")
```

O/P :
```
Name    Age    Class
```

Ex :
```
print("Hello \n World")
```

O/P :
```
Hello
World
```

__/__/____

LISTS:

⊛ A list is a collection used to store multiple values of different data types.

⊛ They are created using square brackets. Each element can be accessed using its index.

Ex:

```
lt = [1, 2, 4.9, 'Hello', "Python", [2, 4, 6]]
print (lt[0])
print (lt[3])
print (lt[5])
```

O/P:

```
1
Hello
[2, 4, 6]
```

0	1	2	3	4	5
1	2	4.9	Hello	Python	[2, 4, 6]
-6	-5	-4	-3	-2	-1

⊛ Here, the strings and list stored within, itself are collections. To take a particular element or character from it, use a second square bracket.

Ex:	O/P:
print (lt[3][1])	e
print (lt[5][2])	6

__/__/__

LIST SLICING:

⭐ List slicing is similar to string slicing and has the same syntax. So, let me not explain this. You can just look at the examples.

Ex:

```
li = [10, 20, 30, 'Hi', 'Good']
print(li[1:4])
print(li[:2])
print(li[::2])
print(li[::-1])
```

O/P:

```
[20, 30, 'Hi']
[10, 20]
[10, 30, 'Good']
['Good', 'Hi', 30, 20, 10]
```

CHANGING LIST VALUES:

⭐ Lists are mutable which means, the elements of a list can be modified. A new element can be added and removed in the same memory location.

⭐ To change the value of a specific item, refer to its index number.

Ex:

```
l = [1, 2, 3, 4, 5]
l[3] = 10
print(l)
```

O/P:

```
[1, 2, 3, 10, 5]
```

__/__/___

⊛ A range of elements can be changed using the slicing syntax:

Ex:	O/P:
l = [1,2,3,4,5] l[1:3] = [20,30] print(l)	[1, 20, 30, 4, 5]

⊛ Inserting more items than the given range would insert given items starting from the given position and remaining items will move accordingly.

Ex:	O/P:
l = [1,2,3,4,5] l[1:3] = [20,30,40,50] print(l)	[1, 20, 30, 40, 50, 4, 5]

⊛ Inserting less items than the given range would insert given items into the range and given other items will move accordingly.

Ex:	O/P:
l = [1,2,3,4,5] l[1:3] = [20] print(l)	[1, 20, 4, 5]

⊛ Note that the range given was for two elements. But only one value was replaced (both 2 and 3 gone). The values 4 and 5 moved accordingly.

__/__/____

INSERTING LIST ITEMS:

⊛ To insert an item into a list without replacing the existing values, we use the insert() method.

SYNTAX:

```
list_name.insert(index, item)
```

Ex:	O/P:
`lf = [10, 20, 30]`	[10, 20, 25, 30]
`lf.insert(2, 25)`	
`print(lf)`	

ADDING LIST ITEMS:

⊛ To add an item to the end of a list, use the append() method.

SYNTAX:

```
list_name.append(item)
```

Ex:	O/P:
`lf = [10, 20, 30]`	[10, 20, 30, 40]
`lf.append(40)`	
`print(lf)`	

⊛ To add multiple values to the end of a list, use the extend() method.

SYNTAX:

```
list_name.extend([list_of_items])
```

Ex:	O/P:
lt = [10, 20, 30] lt.extend ([40, 50, 60]) print (lt)	[10, 20, 30, 40, 50, 60]

⊛ The argument within the extend() method can not only be a list, but any iterable like strings, tuples, sets or dictionaries.

REMOVING LIST ITEMS:

⊛ To remove a specified item, use the remove() method.

Ex:	O/P:
lt = [10, 20, 30, 40] lt.remove (20) print (lt)	[10, 30, 40]

Note: The remove() method displays an error if the specified item is not in the list.

⊛ To remove an item from the specified index, use the pop() method.

Ex:	O/P:
lt = [20, 40, 60] lt.pop(1) print (lt)	[20, 60]

__/__/____

⭐ The pop() method when used without an index, removes the last item.

Ex:	O/P:
li = [20, 40, 60] li.pop() print(li)	[20, 40]

⭐ The del keyword is also used to remove values from a specific index.

Ex:	O/P:
li = [100, 200, 300] del li[0] print(li)	[200, 300]

⭐ The del keyword can delete the whole list when specified without index.

Ex:	O/P:
li = [10, 20, 30] del li print(li)	NameError

⭐ The elements of a list can be cleared without deleting the list. The final result is an empty list.

Ex:	O/P:
li = [1, 2, 3] li.clear() print(li)	[]

__/__/____

COPYING LISTS:

⊛ A list cannot be copied by directly assigning it to another variable.

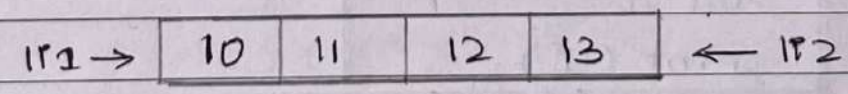
⊛ Instead, we must use the copy() method or the built-in function list().

⊛ You can also copy a list using slicing.

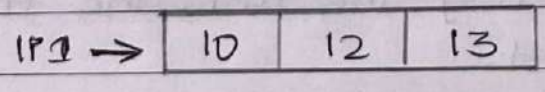
Ex:	O/P:
l1 = [10, 11, 12, 13]	[10, 12, 13]
l2 = l1	[10, 12, 13]
l2.remove(11)	
print(l1)	
print(l2)	

⊛ Here, removing a value in l2 also removes the value in l1 because l1 and l2 are considered names of the same list.

Before removing 11,



After removing 11,



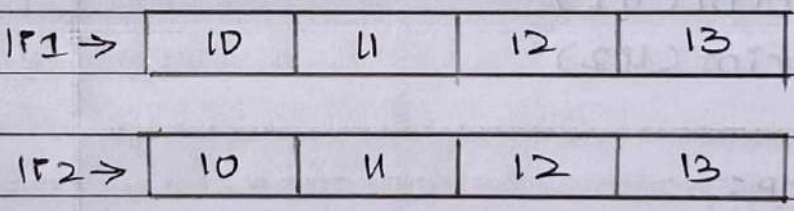
__/__/____

⊛ Use the copy() method,

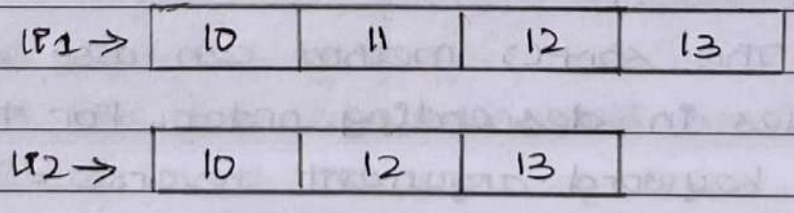
Ex:	O/P:
l1 = [10,11,12,13]	[10,11,12,13]
l2 = l1.copy()	[10,12,13]
l2.remove(11)	
print(l1)	
print(l2)	

⊛ You can note here that l2 is just a copy of the list l1 and is stored in a different memory location.

Before removing 11,



After removing 11,



Note: The same concept applies for mutable types like dict and set as well.

⊛ Copying can also be done using the list() method and list slicing. It has the same method effect of copy().

__/__/____

Ex :

```
li1 = [10, 11, 12, 13]
li2 = list (li1)    # using list() method
li3 = li1 [:]       # using list slicing
```

SORTING LISTS :

⊛ A list can be sorted using the sort() method.

Ex :

```
li1 = [20, 40, 10, 50, 30]
li2 = ["Hi", "Good", "Hello"]
li1.sort()
li2.sort()
print (li1)
print (li2)
```

O/P :

```
[10, 20, 30, 40, 50]
['Good', 'Hello', 'Hi']
```

⊛ The sort() method can also sort the values in descending order. For that, use the keyword argument reverse = True within the sort() method.

Ex :

```
li = [20, 40, 10, 50, 30]
li.sort(reverse = True)
print (li)
```

O/P : [50, 40, 30, 20, 10]

__/__/____

REVERSING LISTS:

⊛ A list can be reversed using the reverse() method.

Ex:	O/P:
l1 = [20, 40, 10, 50, 30]	[30, 50, 10, 40, 20]
l1.reverse()	
print(l1)	

LIST CONCATENATION AND REPETITION:

⊛ Lists can be concatenated (joined) using the '+' operator and repeated using the '*' operator.

Ex:	O/P:
l1 = [10, 20]	[10, 20, 30, 40]
l2 = [30, 40]	[10, 20, 10, 20, 10, 20]
l3 = l1 + l2	
print(l3)	
print(l1 * 3)	

LOOPING THROUGH A LIST:

⊛ A for loop can be used to iterate through the values of a list, one by one.

Ex:	O/P:
l = [10, 20, 30, 40, 50]	10
for i in l:	20
print(i)	30
	40
	50

__/__/__

⊛ You can also loop through its values using the index number. For that, we use the len() and range() functions.

Ex:	O/P:
li = [10, 20, 30, 40, 50]	10
for i in range (len(li)):	20
print (li[i])	30
	40
	50

⊛ The above method, eventhough produces the same result, is mostly used when solving logical programs.

LIST COMPREHENSION:

⊛ List comprehension is used to generate list values in an elegant way.

⊛ It reduces a long code into just one line of code.

For ex,

⊛ If you want to generate square numbers from 1 to 10,

WITHOUT LIST COMPREHENSION:

```
li = []
for x in range (1,11):
    li.append (x * x)
print (li)
```

__/__/____

WITH LIST COMPREHENSION:

```
lr = [x * x for x in range (1, 11)]
print (lr)
```

COMMON O/P:

[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

⭐ Both programs get the same output, but the code using list comprehension is simple.

⭐ A list comprehension can also use an if statement to filter values before adding them to the list.

EX:

```
lr = [x * x for x in range (1, 11) if x % 2 == 0]
print (lr)
```

O/P:

[4, 16, 36, 64, 100]

⭐ Here, the list generates square numbers only for even numbers from 1 to 10.

__/__/____

TUPLES:

⊛ As already discussed in the data types topic, tuples are similar to lists which store multiple values.

⊛ But tuples are immutable and cannot be modified. Tuples can be indexed and sliced similar to strings and lists.

Ex:	O/P:
tup = (10, 20, 30, 40) print (tup[1]) print (tup[:3])	20 (10, 20, 30)

TUPLE ASSIGNMENT:

⊛ Tuples can be assigned with or without paranthesis. The comma is what makes it a tuple.

Ex:	O/P:
tup1 = (10, 20, 30) tup2 = 10, 20, 30 print (type (tup1)) print (type (tup2))	<class 'tuple'> <class 'tuple'>

⊛ But an empty paranthesis is considered a tuple.

Ex:	O/P:
tup = () print (type (tup))	<class 'tuple'>

__/__/___

UNPACK TUPLES :

⊛ The values of a tuple can be assigned to multiple variables and is called unpacking.

EX :

```
tup = ('Apple', 'Ball', 'Cat')
a1, a2, a3 = tup   # unpacking
print (a1)
print (a2)
print (a3)
```

O/P :

```
Apple
Ball
Cat
```

⊛ The number of variables must match the number of values in the tuple.

⊛ If not, you must use an asterisk to collect the remaining values as a list.

USING ASTERISK (*) :

EX :

```
tup = (10, 20, 30, 40, 50, 60)
a, b, *c = tup
print (a, b, c)
```

O/P :

```
10  20  [30, 40, 50, 60]
```

__/__/____

⊛ The asterisk variable can be anywhere in the order, and the unpacking would be done accordingly.

Ex :	O/P :
tup = (10, 20, 30, 40, 50, 60) a, *b, c = tup print (a, b, c)	10 [20, 30, 40, 50] 60

UPDATING TUPLES :

⊛ As tuples are immutable, they can't be updated. But there is a small trick in which

(i) We would convert the tuple to a list
(ii) Then add, remove, or modify elements.
(iii) And again convert it back to a tuple.

```
Ex :
tup = (10, 20, 30)
li = list(tup)
li.append(40)      # add element
li.remove(20)      # remove
li[0] = 80         # change
tup = tuple(li)
print(tup)
```

O/P :
(80, 30, 40)

__/__/____

DELETING TUPLES:

⊛ A tuple can be completely deleted using the del keyword.

Ex:	O/P:
tup = (10, 20, 30) del tup print (tup)	NameError

TUPLE CONCATENATION AND REPETITION:

⊛ Tuples can be concatenated and repeated using '+' and '*' respectively.

Ex:	O/P:
tup1 = (2, 4, 6) tup2 = (8, 10) print (tup1 + tup2) print (tup2 * 3)	(2, 4, 6, 8, 10) (8, 10, 8, 10, 8, 10)

TUPLE METHODS:

⊛ There are two built-in methods that can be used with tuples. They are count() and index().

METHOD	DEFINITION
count()	returns number of times a specified value occurs in a tuple.
index()	Searches the tuple for a specified value and returns the position of where it was found.

__/__/____

Ex:

```
tup1 = (10, 20, 10, 30, 10)
print (tup1.count(10))
print (tup1.index(20))
```

O/P:

```
3
1
```

__/__/____

SETS :

⊛ We saw about sets in the datatypes TOPIC. We would look at it in detail here.

⊛ A set contains multiple values enclosed within {} (braces). It can also be created using the built-in function set().

```
EX :
st = set (('Hi', 'Hello', 'Good'))
print (st)
```

```
O/P :
{'Good', 'Hi', 'Hello'}
```

Note :

⊛ A set is unordered and the order of the value changes.

⊛ The set() function takes an iterable as a single argument.

⊛ A set doesn't allow duplicates. The same value, if appears two times gets printed only once.

ACCESS AND CHANGE SET ITEMS :

⊛ A set item cannot be accessed with an index as it is unordered. However, all elements can be looped using the for statement in random order.

__/__/____

Ex:

```
st = {"Apple", "Ball", "Cat"}
for x in st:
    print(x)
```

O/P:

```
Ball
Apple
Cat
```

⊛ A set item cannot be changed, but you can add or remove items.

ADD SET ITEMS:

⊛ You can add one item to a set, using the add() method. A collection of values can be added using the update() method.

Ex:

```
st = {'A', 'B', 'C'}
st.add('D')
print(st)
st.update(['E', 'F'])
print(st)
```

O/P:

```
{'A', 'C', 'D', 'B'}
{'F', 'E', 'D', 'A', 'C', 'B'}
```

⊛ Here, I have passed a list in the update() method. You can use any iterable you want.

__/__/____

REMOVE SET ITEMS :

⊛ To remove an item in a set, we can use the remove() or discard() method.

⊛ The remove() method raises an error if the item specified doesn't exist.

⊛ The discard() method does not show an error if the item doesn't exist.

Ex :	O/P :
st = {'A', 'B', 'C'}	{'A'}
st.remove('B')	
st.discard('C')	
print(st)	

⊛ You can also use the pop() method to remove an item, but this would remove the last item. As a set is unordered, you never know which item would get removed.

⊛ The pop() method returns the value removed.

Ex :	O/P :
st = {'H', 'I', 'J'}	{'J', 'H'}
st.pop()	
print(st)	

__/__/____

⊛ To clear the set and make it empty, use the clear() method.

EX:	O/P:
st = {'C', 'D', 'E'}	set()
st.clear()	
print(st)	

Note: {} is an empty dictionary. An empty set is defined as set().

⊛ The del keyword deletes the set completely.

EX:	O/P:
st = {'A', 'B'}	Name Error
del st	
print(st) # Error	

SET OPERATIONS:

⊛ Set operations involve joining TWO sets based on certain conditions.

⊛ These operations involve union, intersection, difference and symmetric difference.

⊛ The union() method returns a new set containing all items from both sets.

__/__/____

Ex:	O/P:
st1 = {'A', 'B', 'C'} st2 = {'C', 'D', 'E'} st3 = st1.union(st2) print(st3)	{'A', 'E', 'B', 'C', 'D'}

Note: The elements common to both sets are taken only once.

⭐ The intersection() method returns a new set containing only the common items present in both sets.

Ex:	O/P:
st1 = {'A', 'B', 'C'} st2 = {'C', 'D', 'E'} st3 = st1.intersection(st2) print(st3)	{'C'}

⭐ The difference() method returns a new set which contains elements from the first set, but removes elements with common values in both sets.

Ex:	O/P:
st1 = {'A', 'B', 'C'} st2 = {'B', 'D', 'E'} st3 = st1.difference(st2) st4 = st2.difference(st1) print(st3) print(st4)	{'C', 'A'} {'D', 'E'}

__/__/____

Note: The element 'B' is common to both sets and is excluded while using difference.

⊛ The symmetric_difference() method returns a new set which contains all items from both sets except the common ones.

Ex:

```
st1 = {'A', 'B', 'C'}
st2 = {'C', 'D', 'E'}
st3 = st1.symmetric_difference(st2)
print(st3)
```

O/P:

{'A', 'E', 'B', 'D'}

⊛ These methods return a new set, but if you want a change in the original set for these operations, use these methods.

i) update()
ii) intersection_update()
iii) difference_update()
iv) symmetric_difference_update()

__/__/____

DICTIONARIES:

⊛ Dictionaries are used to store data as key : value pairs. A dictionary is mutable and can be modified.

⊛ A dictionary is ordered and do not allow duplicates.

⊛ The values of a dictionary can be of any type, but the keys must be of an immutable type.

EX:

```
{1 : 'Apple', 2 : 'Ball', 3 : 'Cat'}
```

ACCESS ITEMS:

⊛ You can access the items of a dictionary by referring to its key name, inside square brackets.

Ex:

```
d = {'A' : 10, 'B' : 20, 'C' : 30}
print (d['A'])
print (d['C'])
```

O/P:

```
10
30
```

__/__/____

⊛ We can also use the get() method to access values.

Ex :

```
d = {'A' : 10, 'B' : 20, 'C' : 30}
print (d.get ('B'))
```

O/P :

20

⊛ Access the keys of a dictionary using the keys() method. It returns a list of all the keys in the dictionary.

⊛ The values () method will return a list of all values in the dictionary.

⊛ The items() method will return each item in a dictionary, as tuples in a list.

Ex :

```
d = {'A' : 1, 'B' : 2, 'C' : 3, 'D' : 4}
print (d.keys())
print (d.values())
print (d.items())
```

O/P :

```
dict_keys (['A', 'B', 'C', 'D'])
dict_values ([10, 20, 30, 40])
dict_items ([('A', 10), ('B', 20), ('C', 30), ('D', 40)])
```

__/__/____

CHANGE VALUES:

⊛ You can change the value of a specific item by referring to its key name.

Ex:	O/P:
d = {'A': 10, 'B': 20} d['A'] = 30 print(d)	{'A': 30, 'B': 20}

⊛ We can also use the update() method to change the value. The argument must be a dictionary with a key value pair.

Ex:	O/P:
d = {1: 'Apple', 2: 'Ball'} d.update({2: 'cat'}) print(d)	{1: 'Apple', 2: 'cat'}

ADD ITEMS:

⊛ An item can be added to the dictionary by using a new index key and assigning a value to it.

Ex:
d = {'A': 10, 'B': 20} d['c'] = 30 print(d)
O/P: {'A': 10, 'B': 20, 'c': 30}

__/__/____

REMOVE ITEMS:

⊛ To remove an item, use the pop() or popitem() method.

⊛ The pop() method removes the item with the specified key name. The popitem() method removes the last item.

```
EX:
d = {'A': 10, 'B': 20, 'C': 30}
d.pop('A')
d.popitem()
print(d)
```

```
O/P:
{'B': 20}
```

⊛ The del keywords removes the item with the specified key name. It can also delete the complete dictionary.

```
EX:
d = {'A': 10, 'B': 20, 'C': 30}
del d['B']
print(d)
del d
print(d)
```

```
O/P:
{'A': 10, 'C': 30}
Error
```

__/__/____

⊛ The clear() method empties the dictionary.

Ex:	O/P:
d = {'A': 10, 'B': 20}	{}
d.clear()	
print(d)	

LOOP THROUGH DICTIONARIES:

⊛ A for loop can be used to loop through a dictionary. By default, the normal method returns the keys of a dictionary.

⊛ But there are methods to print the values as well.

Ex:	O/P:
d = {'A': 1, 'B': 2, 'C': 3}	A
for x in d:	B
print(x)	C

Note:

⊛ This method prints keys.

⊛ To print values, use the key indexing method.

Ex:	O/P:
for x in d:	1
print(d[x])	2
	3

__/__/____

⊛ We can use the keys(), values(), and items() method as well to loop through a dictionary.

Ex:	O/P:
for x in d.keys(): print(x)	A B C
Ex:	**O/P:**
for x in d.values(): print(x)	1 2 3
Ex:	**O/P:**
for x, y in d.items(): print(x, y)	A 1 B 2 C 3

Note: Here, two loop variables x and y are used to store both key and value returned as a tuple by the items() method.

COPY A DICTIONARY:

⊛ As a dictionary is mutable, assigning it to a variable doesn't mean copying. Instead, we must use the copy method.

⊛ We can also use the dict() built-in function to make a copy of a dictionary.

__/__/____

Ex:

```
d1 = {'A': 10, 'B': 20}
d2 = d1      # referencing
d3 = d1.copy()  # copying
d1['A'] = 40
print(d1)
print(d2)
print(d3)
```

O/P:

```
{'A': 40, 'B': 20}
{'A': 40, 'B': 20}
{'A': 10, 'B': 20}
```

Note:

⊛ We have referenced d2 to be equal to d1. Any changes made in d1 would also affect d2.

⊛ But d3 is just a copy of d1 and is stored separately. Any changes in d1 doesn't affect d3.

__/__/____

FUNCTIONS:

* In Python, a function is a group of related statements that perform a specific task.

* Functions help break our program into smaller and modular chunks. Functions help making larger code more organized and manageable.

* It avoids repetition and makes the code reusable.

SYNTAX:

```
def function_name(parameters):
    statement(s)
    =
    return value
```

IMPORTANT POINTS:

* This is what is called a function definition. Here is where we write the code to be executed when the function is called.

* The keyword def marks the start of the function header.

* The function_name is used to uniquely identify the function. It follows the rules of writing identifiers.

__/__/____

⊛ Parameters are through which we pass values to a function. They are optional.

⊛ The statement(s) present within the function body must have the same indentation (4 spaces recommended).

⊛ The return statement is used to pass back control to the function call. It can return some value if needed. Using a return statement is optional.

Ex:

```
def greet():    # function definition
    print("Good day")
    print("Have a nice day")
print("Thank you") # normal statement
```

O/P:

```
Thank you
```

⊛ Note that the greet function doesn't get executed because it has not been called. Only the normal statements out of the function block gets executed.

⊛ Here, we have just defined the function and have not called it yet.

__/__/____

FUNCTION CALL:

⊛ To execute a defined function, we must call it. A function call is nothing but to call its name along with values to be passed to the parameters if any.

Ex:

```
def greet():    # fn. definition
    print ("Good Day")
    print ("Have a nice day")

greet()     # function call
print("Thank you")
```

O/P:

```
Good day
Have a nice day
Thank you
```

Note: There are no parameters defined here. So the function call doesn't pass any values (we also call it as arguments).

⊛ A function is executed only when it is called. It can be called as many times as you want.

__/__/____

Ex:

```
def greet():
    print ("Good day all!")

greet()  # fn. call 1
print ("Thank you")
greet()  # fn. call 2
```

O/P:

```
Good day all!
Thank you
Good day all!
```

* Here, greet() has been called two times. Note the order of the output. First, the greet() function is called and executed. Then the print function prints "Thank you". Again, greet() is called and executed.

* In Python, a function definition must always be present above the function call. Or else, it would display an error.

Ex:

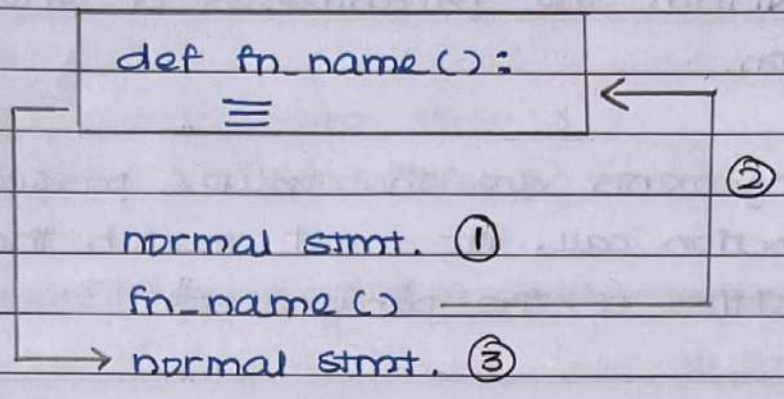
```
greet()
print ("Thank You")
def greet():
    print("Good day")
```

O/P:

```
NameError
```

__/__/___

FUNCTION - EXPLANATION DIAGRAM:



⊛ Wherever it sees a function call, the function definition gets executed.

⊛ After execution, it comes back to the next statement below the function call.

Note:

⊛ The greet() function that we defined had no parameters or a return statement as it was a very simple function to print a greeting directly.

⊛ But now, let us define the same function by passing our name as an argument and print a personalised message.

__/__/____

PARAMETERS AND ARGUMENTS:

⊛ Parameters are the names given to the values passed. Those names (or variables) are given within the parantheses of the function definition.

⊛ Arguments are the values passed from the function call. It must match the number and position of the parameters.

Ex:

```
def greet (name, time):
    print (f"Good {time}, {name}")

greet ("Ahmed", "Evening")
greet ("Raju", "Morning")
greet ("Jack", "Night")
```

O/P:

```
Good Evening, Ahmed
Good Morning, Raju
Good Night, Jack
```

⊛ Here, name and time are variables (parameters) that store the arguments passed in the function call.

⊛ We pass different arguments, each time we call the function.

__/__/____

⊛ If the function definition contains two parameters, the function call must match the number and position of the parameters.

Ex:

```
def greet (name, time):
    print (f" Good {time}, {name}")

greet ("Evening", "Ahmed")  # Wrong O/P
greet ("Morning")           # Error
```

O/P:

```
Good Ahmed, Evening
greet() missing 1 required positional argument
```

⊛ Because it requires the number and position to be retained, we call them positional arguments.

KEYWORD ARGUMENTS:

⊛ Keyword arguments allow us to associate a name along with the value to be passed.

⊛ This means, that the positions can change and only the number of arguments must be retained.

⊛ The names used must match the names of the parameters given.

__/__/____

Ex:

```
def greet(name, time):
    print(f'Good {time}, {name}')

greet(time = "Evening", name = "Ahmed")
```

O/P:

Good Evening, Ahmed

* Here, though I have interchanged the position of arguments, the names given would pass them to the correct parameters.

* Both positional and keywords arguments can be combined together. But, keyword arguments must come only after positional arguments.

Ex:

```
# (1 positional, 1 keyword) Right
greet("Ahmed", time = "Evening")

# Wrong order (keyword before positional)
greet(time = "Evening", "Ahmed")

# Wrong (both args. try going to name parameter)
greet("Evening", name = "Ahmed")
```

__/__/____

ARBITRARY ARGUMENTS:

* If you do not know, how many number of arguments would be passed exactly you can add an asterisk (*) before the parameter name.

* All arguments would be passed to that parameter as a tuple which can then be accessed using index, unpacking or a for loop.

Ex:

```
def greet (*names):
    for x in names:
        print ("Hello", x)

greet ('Jack','Raghu','Ahmed','Nisha','Emma')
```

O/P:

```
Hello Jack
Hello Raghu
Hello Ahmed
Hello Nisha
Hello Emma
```

Note: Arbitrary arguments are often shortened to *args in Python documentations.

__/__/____

ARBITRARY KEYWORD ARGUMENTS:

* If you do not know, how many keyword arguments would be passed exactly, you can add two asterisks (**) before the parameter name.

* All keyword arguments would be passed as a dictionary to the parameter and then can be accessed by the methods used to access dictionaries.

* Arbitrary keyword arguments are often shortened to **kwargs in Python documentation.

Ex:

```
def name (**names):
    print ("First Name:", names['fname'])
    print ("Last Name:", names['lname'])

name (fname = 'Abdul', lname = 'Rahman')
```

O/P:

```
First Name : Abdul
Last Name : Rahman
```

DEFAULT PARAMETERS:

* You can choose to provide a default value to the parameters in a function definition.

* If there are no arguments given for a parameter, the default value would be considered.

__/__/____

Ex:

```
def greet (name = "Buddy", time = "Morning"):
    print (f'Good {time}, {name}')

greet ("John", "Afternoon")
greet ("John")
greet ()
greet (time = "Evening")
```

O/P:

```
Good Afternoon, John
Good Morning, John
Good Morning, Buddy
Good Evening, Buddy.
```

⭐ The last three function calls would have shown an error if there were no default values.

⭐ You can also have a mix of parameters with and without defaults. But, the non-default parameters must come before the default parameters.

Ex:

```
# No error
def greet (name, time = "Morning"):
    print (f'Good {time}, {name}')

# Error
def greet (name = "Buddy", time):
    print (f'Good {time}, {name}')
```

__/__/____

CHANGES WHEN PASSING VALUES:

⊛ Values with mutable data types, when passed into a function, may be modified. If any modification occurs, the change affects the original value as well.

⊛ On the other hand, when values of immutable data types are passed, any changes occuring are only within the function. The original value remains the same.

EX: (For mutable types - list)

```
def func (li):
    li.append (6)      # change applied
    print (li)         # within function

list_1 = [1, 2, 3, 4, 5]
print (list_1)         # before fn. call
func (list_1)
print (list_1)         # after fn. call
```

O/P:

```
[1, 2, 3, 4, 5]
[1, 2, 3, 4, 5, 6]
[1, 2, 3, 4, 5, 6]
```

EX: (For immutable types - String)

```
def func (s):
    s = s.replace ('H', 'J')   # change applied
    print (s)                  # within function
```

__/__/____

```
string = 'Hello'
print (string)    # before fn. call.
func (string)
print (string)    # after fn. call
```

O/P:

Hello
Jello
Hello

⊛ You can see from the examples, that the original list has changed, but the original string has not changed. The changes of a string occurs only within the function.

RETURN VALUES:

⊛ The return statement is used to return a value back to the function call as a result.

⊛ The return value can be stored in a variable and used, or it can directly be printed if the function call happens within a print function.

⊛ A function without a return statement returns None as a value.

__/__/____

Ex:

```
def func (l, b):
    per = 2*(l+b)      # perimeter of rectangle
    return per         # return per result.

print ( func (4,3))    # prints the return value.
```

O/P:

14

ANOTHER EXAMPLE:

```
def func (l, b):
    per = 2*(l+b)
    return per

p = func ( 6,2)   # Store return value in p.
print ("The perimeter is :", p)
```

O/P:

16

ANOTHER EXAMPLE:

```
def func (l, b):
    per = 2* (l + b)   # No return statement

print ( func (6,2))
```

O/P:

None

__/__/____

⊛ In the above example, a value is not returned. If we try printing a function call, it just prints None as the return value is None for a function without a return statement.

⊛ A better option in this case would be to print the result directly within the function and just call the func() function below without printing it.

⊛ Statements below a return statement are not executed if the return statement is executed. That's because, the control goes back to the function call once a return statement is encountered.

EX:

```
def func(a,b):
    c = a+b
    return c
    print("Hello") # doesn't execute

print(func(10,20))
```

O/P:

```
30
```

USING PASS IN FUNCTIONS:

⊛ The pass statement can be used within a function if you want to define a function and not do anything or define it's statements after a while.

__/__/____

Ex:

```
def my_fun():
    pass
```

TYPES OF FUNCTIONS:

⭐ There are two types of functions available in Python. They are,

1) Built-in functions
2) User-defined functions

⭐ Built-in functions are pre-built into Python and can be called directly. The function is already defined.

⭐ Ex: print(), input(), int(), str() etc.

⭐ User-defined functions are functions that we write on our own using the def keyword. All these examples that we saw before such as the greet() function are user-defined.

__/__/____

VARIABLE SCOPE :

⊛ As we have discussed about functions, it is the right time to talk about variable scope.

⊛ The lifetime of a variable in which we can access it is defined by its scope.

⊛ We have three different scopes of variables.

i) Global
ii) Local
iii) Non-local

⊛ A variable that can be accessed throughout a program is called a global variable.

⊛ Variables are defined within a function are local variables and can be accessed only within the function.

⊛ Non-local variables are used with nested functions. Variables within the outer function, but not in the inner function are considered non-local variables.

EX : (GLOBAL AND LOCAL)

```
x = 10          # global variable
def fn():
    y = 20      # local variable.
    print(y)
```

__/__/____

```
    print(x)  # global var. inside fn.
fn()
print(x)
```

O/P:
20
10
10

⊛ Here, x is a global variable and can be accessed from anywhere within the program (outside or inside function).

⊛ But, y is a local variable created within function fn(). It can be used only within the function.

EX: (NON-LOCAL)

```
def outer():
    x = 10          # non-local to inner()
    def inner():
        y = 20      # local
        print(y)
        print(x)
    inner()
    print(x)
outer()
```

O/P: 20
10
10

__/__/____

⊛ The variable x is valid within outer() function, but not available globally. It is also used within the inner() function.

⊛ The variable y is just local to inner() function and can't even be accessed by the outer() function.

OVERSHADOWING OF VARIABLES:

⊛ Variables having the same name in different scopes are overshadowed.

⊛ A variable with the same name in the global and local scope considers only the local variable within the function.

Ex:

```
x = 5   # global
def fn():
    x = 10   # local
    print(x) # prints local value

fn()
print(x)  # prints global value.
```

Op:

```
5
10
```

Note: Same applies for non-local and local variables.

__/__/____

MODIFICATION IN DIFFERENT SCOPES:

* Trying to modify global variables within a function would raise an error.

Ex:	O/P:

x = 10
def fn():
  x = x + 10 # Error
  print(x)
 	
fn()
	Error

* In the above program, we never defined x within the function. So it considers x as an undefined variable.

* To modify a global variable within a function, use the global keyword.

Ex:	O/P:

x = 10
def fn():
  global x
  x = x + 10
  print(x)
 	
fn()
	20

__/__/____

* The same concept applies to the non-local scope as well. In cases, where the non-local variables need modification within a local scope, use the non-local keyword.

Ex:

```
def outer():
    x = 10    # non-local
    def inner():
        nonlocal x  # refer x as nonlocal
        x = x + 10
        print(x)

    inner()
outer()
```

O/P:

20

__/__/____

EXCEPTION HANDLING:

* When an error occurs (we call them exceptions as well), the Python program would normally stop and generate an error message.

* To avoid such sudden stops and crashes, Python uses the try-except blocks to handle exceptions in an elegant manner.

* This means that the program would run even after an exception occurs, as the exception is well handled.

EX: (WITHOUT EXCEPTION HANDLING)

```
a = 0/0
print(a)
print("Done")
```

O/P:

```
ZeroDivisionError
```

* Zero can't be divided by zero. As the first line displays an error, the code below it is not printed.

TRY AND CATCH (EXCEPT) BLOCK:

* The try block lets you test the code for errors. If there are no errors, it runs the complete block and skips the except block.

__/__/____

⊛ If there is an error in the try block, it directly goes to the except block and executes whatever present.

⊛ After the try-except block is executed, the program runs normally without stopping half-way.

EX : (WITH EXCEPTION HANDLING)

```
try :
    a = 0/0    # exception raised
    print (a)
except :
    print ('can't divide by zero')

print ('Done')
```

O/P :

```
Can't divide by zero
Done
```

⊛ Here, the exception is handled well, and the program doesn't stop abruptly.

MULTIPLE EXCEPT BLOCKS :

⊛ It is recommended to specify the error type to be handled in the except block rather than a generic except.

⊛ There can be multiple except blocks to handle errors as well.

__/__/____

Ex:

```
try:
    print(d)
except NameError:    # Specific for NameError
    print('d is not defined')
except:    # All other errors
    print("Some generic error")
```

* Here, the last except is optional and doesn't need to be given.

* There are a lot of built-in error types such as ArithmeticError, ImportError, IndexError, NameError, SyntaxError, IndentationError, TypeError etc.

* You can use any of the error types in the except block to handle errors. If there are a lot of errors to be handled, you can use a generic except block.

ELSE AND FINALLY BLOCKS:

* You can use the else block below the try-except block to execute some code if no errors were raised.

* The finally block is executed regardless of whether an exception or error occurs or not. This can be used to clean up resources and close objects such as closing a file, database etc.

__/__/____

Ex :

```
try:
    a = 10
    print (a)
except NameError:
    print ("a not defined")
else:
    print ("No exception occurred")
finally:
    print ("Program finished")
```

O/P :

```
10
No exception occurred
Program finished
```

* As this program didn't raise an exception, the try block executes, the else and finally blocks are also executed.

* In the same example, if the line a=10 is removed or commented out, a NameError would be raised. If that was the case, the output would be,

O/P :

```
a not defined
Program finished
```

* Here, as the exception occurs, the except block would be executed and the finally block is executed. The else block

__/__/____

is avoided and not executed.

RAISING EXCEPTIONS:

⊛ If you want to raise an exception for a custom condition, use the raise keyword.

EX:

```
x = "hello"
if type(x) is not int:
    raise TypeError("Only Integers Allowed")
```

O/P:

```
TypeError: Only Integers Allowed
```

__/__/____

MODULES :

⊛ A Python module is nothing but a Python file containing Python functions, classes, and variables.

⊛ It can be imported into other programs and the functions, classes, and variables can be used in that programs.

⊛ It helps us to organize code and avoid name clashes.

CREATING A MODULE :

⊛ Create a Python file and define anything you want within that file. Let the filename be mymod.py.

Ex : (mymod.py)

```
def add (a, b) :
    return (a+b)

def sub (a, b) :
    return (a-b)
```

⊛ Here, we have defined two functions, add() and sub() within mymod.

⊛ Similarly we can include variables and classes as well.

__/__/____

IMPORT A MODULE:

⊛ A module created can be imported into another Python file. Use the import keyword.

⊛ Let us import the mymod module created above. The module name is nothing but the filename.

EX:	O/P:
import mymod x = mymod.add(5,3) y = mymod.sub(9,6) print("Sum:", x) print("Diff:", y)	Sum: 8 Diff: 3

⊛ Here, the functions within the module can't be called directly. Use the module name, then a dot and then the function name() to access a function from the module.

```
<module_name>.<function_name>
```

⊛ Also make sure that the module file is placed in the same location of Python file where the module is used.

⊛ The module can actually be placed anywhere. But, you would have to edit the PATH variable to search that location.

__/__/____

USING THE 'FROM' STATEMENT :

⊛ Python's from statement is used along with import, to import specific contents from a module without importing the module as a whole.

EX :

```
from mymod import sub
res = sub(10,6)
print(res)
```

O/P :

```
4
```

⊛ Note that the function sub() is called without mentioning the module name. This is how you call functions that are specifically loaded from a module.

RENAMING THE MODULE :

⊛ A module can be renamed before using it in our program using the as keyword.

EX :

```
import mymod as m
print(m.add(10,20))
print(m.sub(20,30))
```

O/P :

```
30
-10
```

⊛ You can also use mymod as the module name here. The name 'm' is just an alias.

__/__/____

BUILT-IN MODULES :

⊛ In addition to built-in functions, Python has a large number of pre-defined functions available within different modules.

⊛ These modules are built-in and must be imported before using its functions.

⊛ Each built-in module contains resources for specific functionalities such as OS management, disk I/O, networking, database connectivity etc.

⊛ To display the list of all available modules, use the following command in the Python console.

```
help('modules')
```

COMMONLY USED MODULES :

MODULE NAME	DESCRIPTION
os	This module has functions to perform many tasks of the operating system.
random	The random module is used for generating random numbers.

__/__/____

MODULE NAME	DESCRIPTION
math	This module contains commonly required mathematical functions.
sys	The sys module provides functions to manipulate the Python runtime environment.
collections	This module provides alternates to built-in container data types such as list, tuple and dict.
time	The time module contains many time related functions.
re	This module is used to create regular expressions and do pattern matching.
http	The http module is used for implementing web servers.

* There are a lot of built-in modules available and are used for different purposes. The above said modules are commonly used.

__/__/____

SOME EXAMPLES:

⊛ There are a lot of functions available in each of the modules. Below examples are just simple programs on how to use those modules.

Ex:

```
Import math
print (math.sqrt(100))
print(math.ceil(4.57))
print(math.floor(8.9))
```

O/P:

```
10.0
5
8
```

Ex:

```
Import os
os.mkdir('D:\\sample')
```

Note: Go to the specified location and a folder would have been created.

Ex:

```
Import time
print(time.ctime())
```

O/P:

```
Fri Oct 21 19:18:22 2022
```

⊛ Here, I have demonstrated some simple functions from math, os, and time module.

⊛ To learn about all the other functions in a module, specify the module name within the help function.

__/__/____

```
help('math')
```

⊛ To get details about a particular function within a module, specify the function name along with the module name.

```
help('time.ctime')
```

__/__/____

PACKAGES :

⊛ A package in Python consists of one or more modules. It is nothing but a folder of modules.

⊛ In addition to that, the folder must contain a special file called __init__.py which is a packaging list.

⊛ The init file has two purposes. It is necessary for the Python interpreter to recognise a folder as a package.

⊛ Secondly, it offers only specified resources from its modules to be imported.

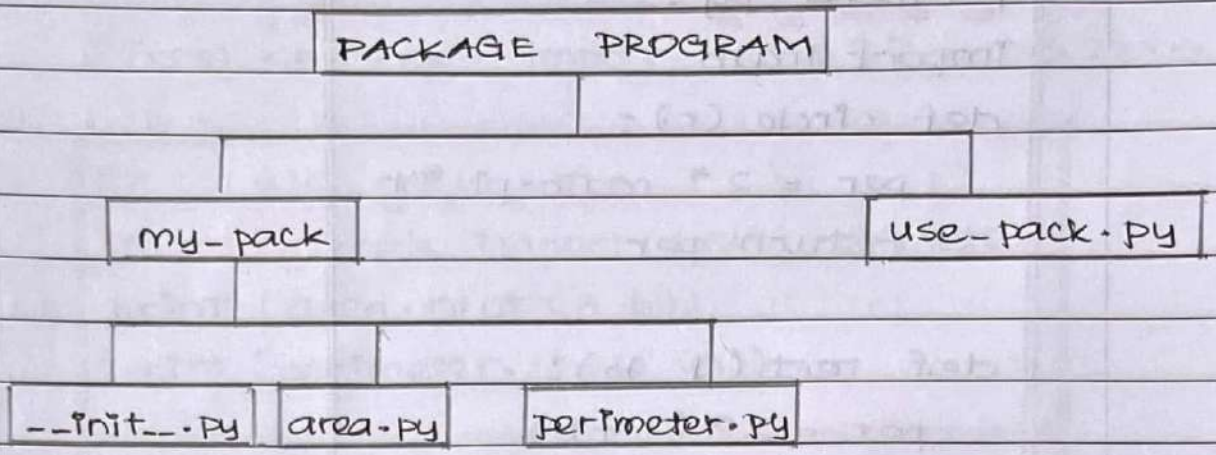
⊛ The __init__.py file can be left empty if you want, to make all the resources available for use.

⊛ A package can be used locally within a program, or deployed for a system-wide use or even made publicly importable by uploading it to PyPI repository.

CREATING A PACKAGE:

⊛ For demonstrating the creation and usage of a package let us create a folder structure.

__/__/___



⊛ Within the Package Program folder, we have the my_pack folder which is our package and use_pack.py file, where we would import and use our package.

⊛ Inside my_pack, we have two modules (python files) namely area.py and perimeter.py which contains the functions for calculating the area and perimeter of a rectangle and a circle.

⊛ There is also an empty __init__.py file which is a must for my_pack to be recognized as a package.

area.py :

```
import math
def circle(r):
    area = math.pi * r * r
    return area
def rect(l, b):
    area = l * b
    return area
```

__/__/____

perimeter.py :

```
import math
def circle (r):
    per = 2 * math.pi * r
    return per

def rect (l, b):
    per = 2 * (l+b)
    return per
```

IMPORTING A PACKAGE :

⊛ To import modules from a package, use

from <package> import <module>

Ex : (use_pack.py)

```
from my_pack import area
print (area.rect (2,3))
print (area.circle (5))
```

O/P :

```
6
78.54
```

⊛ In the above program the 'area' module has been imported from the my_pack package.

⊛ If you want to import multiple modules at once, use a comma and mention all the modules.

__/__/____

from <package> import <mod 1>, <mod 2> ...

EX: (use_pack.py)

```
from my_pack import area, perimeter
print (area.rect (3,6))
print (perimeter.circle (7))
```

O/P:

```
18
43.98
```

__/__/____

CLASSES AND OBJECTS:

⊛ Python supports object-oriented programming which involves classes and objects.

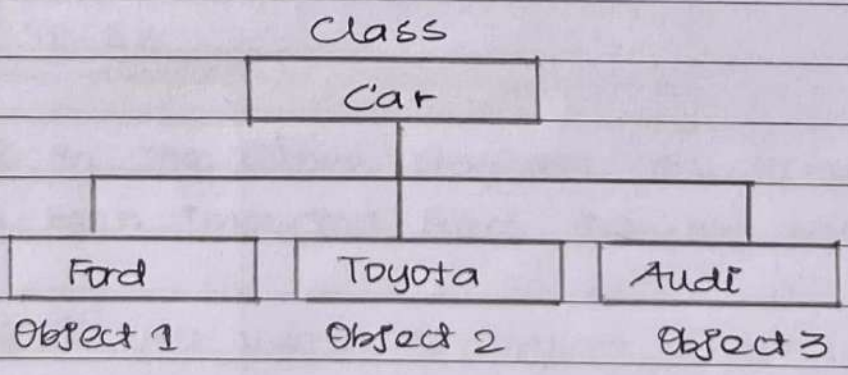
⊛ A class is a blueprint for the objects which contain some attributes (variables) and methods (functions).

⊛ An object is an instance of the class which can access the attributes and methods available within the class.

⊛ You can create as many objects as you want for a class.

⊛ The concept of classes and objects improves modularity, supports reuse of code, increases flexibility and gives effective problem solving.

EXAMPLE DIAGRAM:



__/__/____

CREATING A CLASS:

* A class in Python can be created using the keyword "class".

* It may contain some attributes and methods, along with the __init__() method.

* The __init__() method is called the constructor of a class which is invoked automatically when an object is created.

* The __init__() method is used to define and initialize attributes within a class.

Note: If you do not understand these terms, the syntax and examples below would help you.

SYNTAX:

```
class User:
    def __init__(self):
        attributes

    other methods

normal statements.
```

__/__/____

Ex :

```
class User :                          # class
    def __init__ (self) :             # constructor
        self.name = "Ahmed"
        self.age = 23                 # attributes (name,
        self.id = 1008                # age, id)

    def display (self) :              # method
        print ("Name:", self.name)
        print ("Age:", self.age)
        print ("ID :", self.id)
```

⊛ Here, we have defined a class named User. There are three attributes (name, age, and id) and one method (display).

⊛ By default, there is one parameter given, named self which would denote the current object using the class.

⊛ Every method in a class would have self as the first parameter by default.

⊛ Every attribute in a class would be written as self.attribute as well.

⊛ The self parameter doesn't need to be named self. It can be named anything. But it is common to name it as self.

Note:

* You can clearly see that attributes are nothing but variables. When present within classes, we call them attributes.

* Similarly, functions when used within classes are called methods.

* We can also call attributes as data members and methods as member functions of a class.

CREATING OBJECTS:

* So far, we know how to create classes. But to access the attributes and methods within it, we need to define objects.

* Objects are nothing but variables which contain the initialization of a class.

SYNTAX:

```
object = Classname()
```

Ex:

```
user1 = User()    # creating an
                  # object
```

* Here, user1 is the object which can access the attributes and methods of the User class.

__/__/____

Ex:

```
class User:
    def __init__(self):
        self.name = "Ahmed"
        self.age = 23
        self.id = 1008

    def display(self):
        print("Name:", self.name)
        print("Age:", self.age)
        print("ID:", self.id)

user1 = User()
user2 = User()
user1.display()
user2.display()
```

O/P:

```
Name : Ahmed
Age : 23
ID : 1008
Name : Ahmed
Age : 23
ID : 1008
```

* In this example, we have defined the User class and then created two objects (user1 and user2).

* With these two objects we can access the attributes and methods of the User class.

⊛ We have accessed the display method for both objects. However, it displays the same output. (We will customize it in the next topic).

Note :

⊛ The __init__() method was never called directly. Instead it was automatically called when we created the objects.

⊛ Also note that the self parameter doesn't need any value to be passed. It automatically passes the objects created, to the self parameter.

PASSING VALUES TO CONSTRUCTOR :

⊛ Each object can have different values for its attributes, using parameters within the __init__() method.

⊛ The values would be passed when creating the object.

Ex :

```
class User :
    def __init__(self, name, age):
        self.name = name
        self.age  = age

    def display(self):
        print("Name:", self.name)
        print("Age:", self.age)
```

__/__/____

```
user1 = User("Ahmed", 23)
user2 = User("Raja", 18)
user1.display()
user2.display()
print(user1.name) # direct access of attribute.
```

O/P:

```
Name: Ahmed
Age: 23
Name: Raja
Age: 18
Ahmed
```

Note: Here we have passed different values for each object. So each user has different details for name and age.

MODIFYING OBJECT ATTRIBUTES:

* The attribute values of an object can be changed within the program.

<Object name>.<attribute name> = value

Ex: (continuing the previous program)

```
user1.name = "Jack"
user1.age = 42
user2.age = 21
user1.display()
user2.display()
```

__/__/____

O/P:
Name : Jack
Age : 42
Name : Raja
Age : 21

⊛ Here, the name and age values of user1 and age value of user2 has been modified.

⊛ However, modifying attribute values directly outside a class is not recommended. Instead, we use getters and setters.

GETTERS AND SETTERS:

⊛ Getters and setters are nothing but methods used to access and modify attribute values and to avoid direct modification outside the class.

⊛ Getters are methods that return the value of an attribute.

⊛ Setters are methods that modifies or sets the value of an attribute by taking the new value in its parameter.

EX:

```
class User:
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

__/__/____

```
    def getname(self):    #getter
        return self.name

    def getage(self):     #getter
        return self.age

    def setname(self, name): # setter
        self.name = name

    def setage(self, age): # setter
        self.age = age

user1 = User("Ahmed", 29)
print(user1.getname())
print(user1.getage())
user1.setname("Drake")
user1.setage(24)
print(user1.getname())
print(user1.getage())
```

O/P:

```
Ahmed
29
Drake
24
```

⊛ The getters and setters are efficient methods that help us avoid direct access or modification of an attribute.

__/__/____

⊛ However, it doesn't guarantee that the code may not be able to access or modify an attribute.

⊛ For that, we must make the attributes private where needed.

PRIVATE ATTRIBUTES:

⊛ An attribute of a class can be made private by prefixing it with two underscores (__).

⊛ A private attribute is accesible only within a class and not anywhere outside the class.

⊛ With this, we can avoid direct access of an attribute outside the class.

EX:

```
class User:
    def __init__(self, name):
        self.__name = name

    def getname(self):
        return self.__name

    def setname(self, name):
        self.__name = name

user = User("Raju")
print(user.getname())
```

__/__/___

```
user.setname("Nancy")
print(user.getname())
print(user.__name)
```

O/P:
Raju
Nancy

⊛ If we try printing user.__name it will display an error as it is a private attribute.

⊛ Attributes without an underscore prefixed, are public attributes and can be accessed from anywhere within the program.

DELETING ATTRIBUTES AND OBJECTS:

⊛ A particular attribute of an object or an object itself can be completely deleted using the del keyword.

Ex:

```
class User:
    def __init__(self):
        self.name = "Hello"
        self.age = 5

    def getname(self):
        return self.name

    def getage(self):
        return self.age
```

```
user = User()
print(user.getname())      # Hello
print(user.getage())       # 5
del user.name
print(user.getname())      # Error
print(user.getage())       # 5
del user
print(user.getage())       # Error (obj. deleted)
```

⊛ In the above program, we first delete the name attribute for the user object. So the getname() method doesn't work.

⊛ The name attribute would work for other objects if created.

⊛ Secondly, we delete the user object itself. Even though age attribute is not deleted, it can't be accessed by the user object, as it is deleted.

THE PASS STATEMENT :

⊛ If a class needs to be defined afterwards, we can use a pass statement within it to avoid errors same like we do for functions and other control structures.

```
EX :
class Car :
    pass
```

__/__/____

CLASS ATTRIBUTES:

* The attributes we used till now were defined within the __init__() method and are called instance attributes (specific to each objects).

* Attributes can also be defined outside the __init__() method and are called class attributes (shared by every object).

* Class attributes can be accessed using the name of the class as well as their objects.

* These attributes are used when you need every object to have a common value.

Ex:

```
class User:
    bonus = 3000  # class attribute

    def __init__(self, name):
        self.name = name    # instance
                            # attribute

us1 = User("Jack")
us2 = User("Jill")
print(us1.bonus)
print(User.bonus) # Accessed by class name
User.bonus = 4000
print(User.bonus)
print(us2.bonus) # value changes
```

__/__/___

O/P:

```
3000
3000
4000
4000
```

* In this program, 'bonus' is a class attribute which has the value 3000. It can be accessed by all objects as well as the class name User.

* Modification is done using the class name, which would change the value for all objects.

* When the value 3000 becomes 4000, the objects us1, us2 and class User itself would hold 4000 as the bonus value.

CLASS METHODS:

* A method that is common to all objects and can be accessed directly by the class name is a class method.

* A class method can be defined by adding the '@classmethod' tag at the top of the definition.

* A class method doesn't take the self parameter. Instead it uses the cls parameter which denotes the class name.

* It is mostly used to manipulate and access the class attributes in the class.

__/__/____

Ex:

```
class User:
    bonus = 3000

    def __init__(self, name):
        self.name = name

    @classmethod
    def add_bonus(cls, amount):
        cls.bonus += amount

us1 = User("Jack")
us2 = User("Jill")
User.add_bonus(1000)
print(us1.bonus)
print(us2.bonus)
```

O/P:

```
4000
4000
```

* Here, the add_bonus() method is a class method, which takes the amount and adds it to the bonus attribute.

* Note that the class method accesses only the class attributes and not the instance attributes.

Note: The '@classmethod' is nothing but a decorator. A decorator extends the behaviour of a function without actually modifying it.

__/__/____

STATIC METHODS:

⊛ A class can also contain methods that do not take any parameters such as self or cls, and can be accessed by every object and even the class itself.

⊛ It is like a normal function, but within a class. Such a method is called a static method.

⊛ A static method is denoted by the '@staticmethod' decorator.

⊛ It is used for writing functions that need to be safe and secure in a class rather than being used directly in a program.

Ex:

```
class Calc:
    @staticmethod
    def square(x):
        return x * x

c1 = Calc()
print(c1.square(5))
print(Calc.square(7))
```

O/P:

```
25
49
```

__/__/____

⊛ In this program, we have accessed the static method square() using both object and class.

__/__/____

INHERITANCE :

⊛ In OOP, Inheritance is an important concept that allows a class to inherit all the attributes and methods from another class.

⊛ Inheritance helps us in avoiding rewriting code. Instead common attributes and methods can be inherited from a parent class.

⊛ The class being inherited from is called the parent class or the base class or the super class.

⊛ The class that inherits from another class is called the child class or the derived class or the sub class.

SYNTAX :

```
class One :
    attributes
    methods

class Two (One):      # Two inherits One
    attributes
    methods
```

EX :

```
class One :
    def hello (self):
        print ("Hello from One")
```

__/__/____

```
class TWO (One):
    def hi (self):
        print ("Hi from TWO")

o = One()
t = TWO()
o.hello()
t.hello()
t.hi()
```

O/P:

Hello from One
Hello from One
Hi from TWO

⊛ In this example, we have defined TWO classes One and TWO. Each class has one method named hello() and hi() respectively.

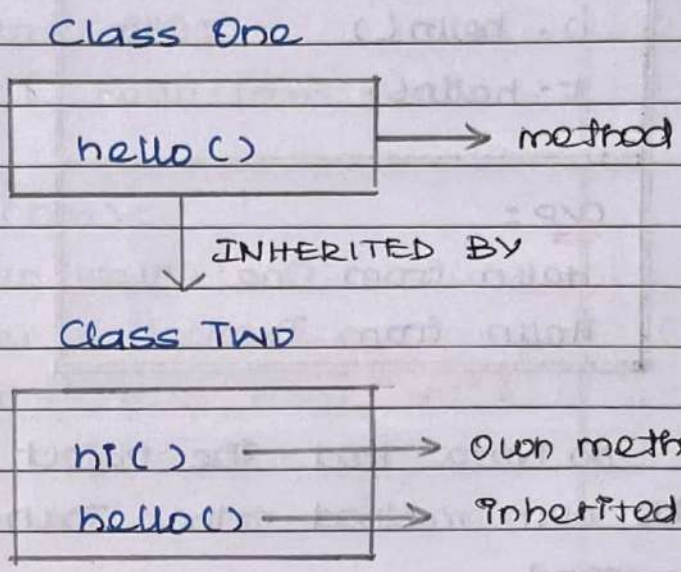
⊛ Class TWO inherits One. So, the objects of TWO can access its own method hi() as well as the method in its parent class, hello().

⊛ This is the power of Inheritance. The child class could access the method from its parent class instead of defining the method again within its class.

Note: Private members of a class do not get inherited.

__/__/____

DIAGRAM:



METHOD OVERRIDING:

⊛ When there are methods of same name in both parent and child classes, the child class object executes the method within the child class and not the parent class. This is called method overriding.

⊛ Usually, method overriding is done extend the functionality of a method.

EX:

```
class One:
    def hello(self):
        print("Hello from One")

class TWO(One):
    def hello(self):
        print("Hello from TWO")
```

__/__/____

```
o = One()
t = Two()
o.hello()
t.hello()
```

O/P:

Hello from One
Hello from Two

⊛ Note that the object of class Two executes its own method rather than its parent class method.

⊛ This often happens for the __init__() method which would be present in both classes.

⊛ To avoid these clashes and to call the method from the parent class, we use the super() function.

THE SUPER() FUNCTION:

⊛ A method or attribute in a parent class can be referred using the super() function.

⊛ It supports code reusability as there would be no need to write the entire function.

__/__/____

Ex:

```
class One:
    def hello(self):
        print("Hello from One")

class TWO(One):
    def hello(self):
        super().hello()
        print("Hello from TWO")

t = TWO()
t.hello()
```

O/P:

```
Hello from One
Hello from TWO
```

⊛ Here, in the hello() method of the derived class, we have explicitly called the hello() method of the base class using super() function and also printed our own message.

⊛ In this way, you can extend the functionality of a method in the child class and still execute the same method in the parent class using the child class object.

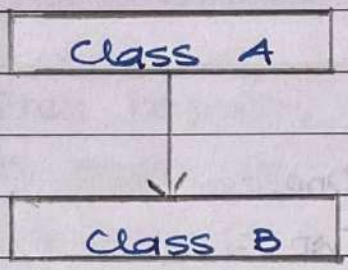
__/__/____

TYPES OF INHERITANCE:

⊛ Inheritance can be classified into different types based on how the derived class(es) inherit from the base classes. They are as follows :

SINGLE INHERITANCE:

⊛ When a child class inherits from only one parent class, it is called single inheritance.



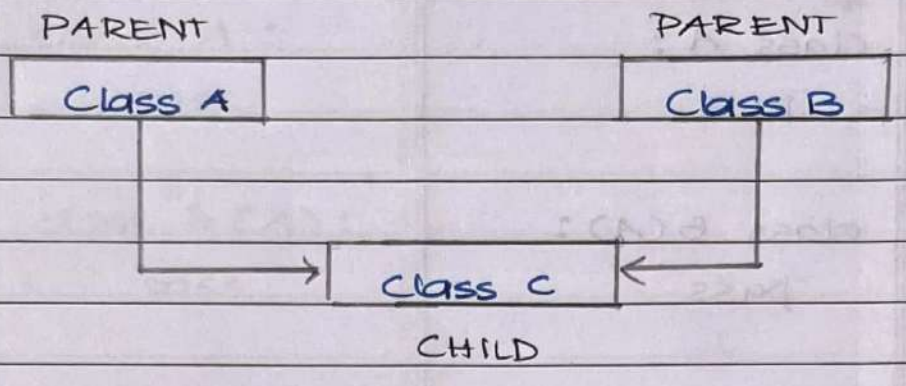
Ex :

```
class A :
    pass

class B(A):
    pass
```

MULTIPLE INHERITANCE:

⊛ When a child class inherits from multiple parent classes, it is called multiple inheritance. The base classes are separated by a comma during inheritance.

__/__/____

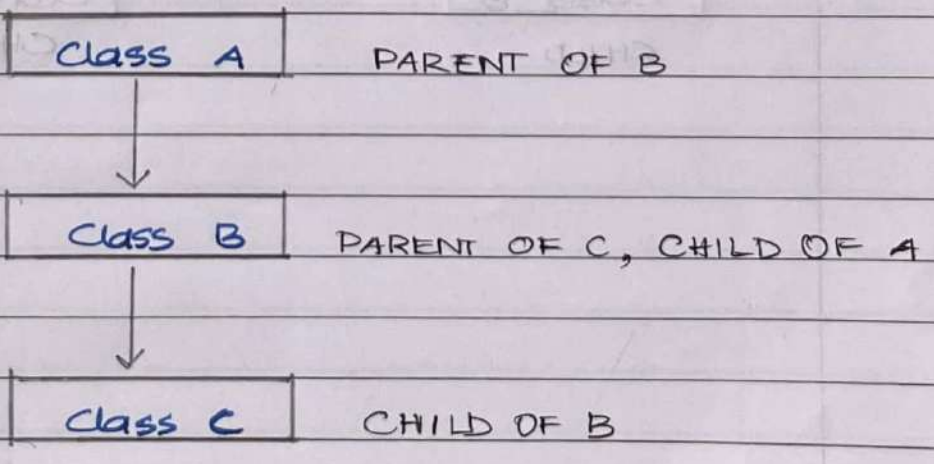


EX :

```
class A :
    pass

class B:
    pass

class C(A,B):
    pass
```

MULTILEVEL INHERITANCE :

⭐ When inheritance occurs in multiple levels where a child class becomes a parent class for another class is called multilevel inheritance.



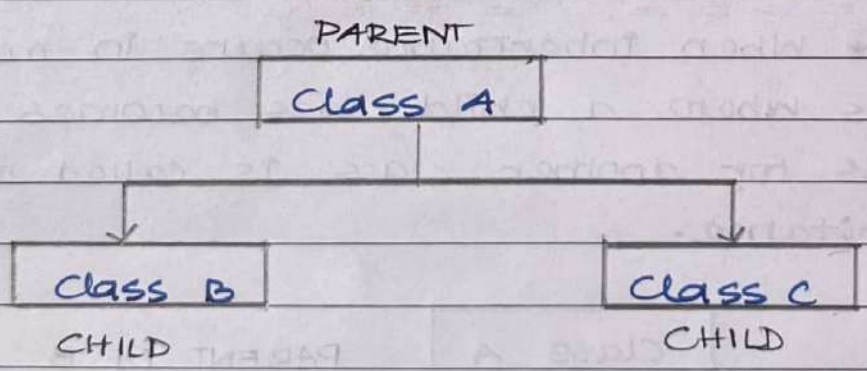
__/__/____

```
Ex :
class A :
    pass

class B(A):
    pass

class C(B):
    pass
```

⊛ As class B is connected to both class A and class C, the objects of class C can access the members of class A as well.

HIERARCHICAL INHERITANCE :

⊛ When more than one child class inherits from a single base class, it is called hierarchical inheritance.



__/__/____

Ex:

```
class A :
    pass

class B (A):
    pass

class C(A):
    pass
```