The Ultimate Python Notebook

~ By SAGAR BISWAS

PREFACE

Welcome to the "Ultimate Python Programming Handbook," your comprehensive guide to mastering Python programming. This handbook is designed for beginners and anyone looking to strengthen their foundational knowledge of Python, a versatile and user-friendly programming language.

This handbook aims to make programming accessible and enjoyable for everyone. Whether you're a student new to coding, a professional seeking to enhance your skills, or an enthusiast exploring Python, this handbook will definitely be helpful. Python's simplicity and readability make it an ideal starting point for anyone interested in programming.

☑ WHY PYTHON?

Python is known for its simplicity and readability, making it perfect for beginners. It is a high-level, interpreted language with a broad range of libraries and frameworks, supporting applications in web development, data analysis, AI, and more. Python's versatility and ease of use make it a valuable tool for both novice and experienced programmers.

PYTHON PROGRAMMING NOTE

☑ WHAT IS PROGRAMMING?

Just like we use Bangla or English to communicate with each other, we use a programming language like Python to communicate with the computer. Programming is a way to instruct the computer to perform various tasks.

☑ WHAT IS PYTHON?

Python is a simple and easy to understand language which feels like reading simple English. This Pseudo code nature is easy to learn and understandable by beginners.

☑ FEATURES OF PYTHON

- Easy to understand = Less development time
- Free and open source
- High level language
- Portable: Works on Linux / Windows / Mac.
- Fun to work with!

P.T.O

Part 1 – MODULES, COMMENTS & PIP

☑ MODULES:

A module is a file containing code written by somebody else (usually) which can be imported and used in our programs.

☑ PIP:

Pip is the package manager for python. You can use pip to install a module on your system.

Ex: `pip install flask # Installs Flask Module`

Example(module_pyjokes): https://pyjok.es/

`"""`
`For this pyjokes module,`
 	
`we should Install with pip using:`
 	
`        pip install pyjokes`
`"""`
	`import pyjokes`
	
	`print("Printing Jokes: ")`
	
	`# function for random jokes.`
	`joke = pyjokes.get_joke()`
	`print(joke)`

☑ TYPES OF MODULES

- There are two types of modules in Python.

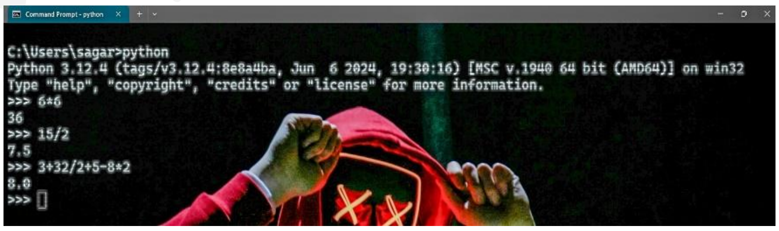
  1) Built in Modules (Preinstalled in Python)
     ```
     import os, random
     ```

  2) External Modules (Need to install using pip)
     ```
     import pyjokes, tensorflow, flask
     ```

☑ USING PYTHON AS A CALCULATOR

We can use python as a calculator by typing "python" + ↵ on the terminal. This opens REPL or Read Evaluate Print Loop.

Demo:



```
C:\Users\sagar>python
Python 3.12.4 (tags/v3.12.4:8e8a4ba, Jun  6 2024, 19:30:16) [MSC v.1940 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> 6*6
36
>>> 15/2
7.5
>>> 3+32/2+5-8*2
8.0
>>> 
```

☑ COMMENTS

Comments are used to write something which the programmer does not want to execute. This can be used to mark author name, date etc.

☑ TYPES OF COMMENTS:

- There are two types of comments in python

1. Single Line Comments: To write a single line comment just add a ‘#’ at the start of the line.

# This is a Single-Line Comment ----- Select lines then ----- **(Ctrl+/)** ……….

2. Multiline Comments: To write multi-line comments you can use ‘#’ at each line or you can use the multiline string (""" """)

"""Hi, this is Sagar Biswas.

Want to be an It Expert.

A proud citizen of Bangladesh!"""

Part 2 – VARIABLES AND DATATYPE

☑ Variable / Identifiers:

A variable is the name given to a memory location in a program. For example.

```
a_ = 1          # a is an integer <int>
b12sd1 = 5.22   # b is a floating <float>
cX1x = "Sagar"  # c is a String <str>
d1 = False      # d is a boolean <bool>
_e = None       # e is a none type variable (Nothing is stored in this variable)
```

☑ DATA TYPES:

Primarily these are the following data types in Python:

1. Integers
2. Floating point numbers
3. Strings
4. Booleans
5. None

Python is a fantastic language that automatically identifies the type of data for us.

☑ RULES FOR CHOOSING AN IDENTIFIER

• A variable name can contain alphabets, digits, and underscores.

• A variable name can only start with an alphabet and underscores.

• A variable name can’t start with a digit.

• No while space is allowed to be used inside a variable name.

Examples:

☠ 9Sagar = "Sagar" #invalid due to start with digits.
☠ @Sagar = 56 #invalid due to @symbol.

☠ Sag@r = 78.32 #invalid due to @symbol.

☑ <u>OPERATORS IN PYTHON: (4 types)</u>

1. Arithmetic operators:	"+, -, *, / etc."	Ex: x + y, x*y, x/y
2. Assignment operators:	"=, +=, -= etc."	Ex: x+=1, x=9
3. Comparison operators:	"==, >, >=, <, != etc".	Ex: x < y, x=!y, y==y,
4. Logical operators:	"and, or, not"	Ex: x and y, x not(!) y, x or y

☠ <u>Assigning Value with Expression:</u>

```
Number1 = 1 if (2>1) else 0
Number2 = 1 if (2>3) else 0

print(Number1, Number2) # Output: 1 0
```

☑ Note: square(x^y): not working in python will work: -- square(x**y) also valid multi(x*y)

```
result_xor(x^y): # This will be 1 (binary 10 XOR 11)
```

• <u>Logical Operators:</u>

# OR: 	
 	
y = None
if (True or False and True or True and False or True):   # if one side is correct then it returns true.
 	
    y = True or False
    print(y)
	# AND:
	
	z = None
	if (True and False and True and True and False and True): # if both side must correct then it returns true.
	
	    z = True or False
	    print(z)
		# NOT:
		
		print(not(False))
		print(not(True))

☑ <u>TYPE() FUNCTION AND TYPECASTING.</u>

☠ type() function is used to <u>find the data type</u> of a given variable in python.

```
a = 31
type(a) # class <int>
b = "31"
type (b) # class <str>
```

There are many <u>functions</u> to convert one data type into another.

```
str_num = str(31)            # Integer to String Conversion
num = int("32")              # String to Integer Conversion
float_num = float(32)        # Integer to Float Conversion
num = int(32.5)              # Float to Integer Conversion
float_num = float("32.5")    # String to Float Conversion
str_num = str(32.5)          # Float to String Conversion
```

☑ INPUT () FUNCTION

☠ This function allows the user to take input from the keyboard as a string.

```
A = input ("enter name") # if a is "sagar", the user entered sagar.
```

```
# a = input("\nEnter number for sum: " ) # By default the input() takes input as a string.
a = float(input("Enter number for sum: ")) # casting the string as a float
b = int(input("Enter number for sum: "))
print(type(a), type(b)) # Output <class 'float'> <class 'int'>
print("..:: The Sum is: ", a + b)
```

Part 3 – STRINGS

- ☠ string is a data type in python.
- ☠ string is a sequence of characters enclosed in quotes.
- ☠ string, tuples can't be changed (faster due to immutability)
- ☠ In Python, both single quotes (') and double quotes (") can be used to create string literals.

- We can primarily write a string in these three ways.

```
a ='sagar biswas' # Single quoted string  # for multiple character/words. ('s', 'a', 'g', 'a', 'r')
b = "sagar biswas" # Double quoted string # same as single quoted.
c = '''I love Bangladesh.
but I will fly away.
will come back for the old...'''   # Triple quoted string # for multiple line
```

☑ STRING SLICING

A string in python can be sliced for getting a part of the strings. Consider the following string:

Name =	"S	a	g	a	r		B	i	s	w	a	s"
Index: (+)	0	1	2	3	4	5	6	7	8	9	10	11
Index: (-)	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

The index in a sting starts from 0 and end to (length -1) in Python. In order to slice a string, we use the following

Negative Indices: -1 = (length - 1) index, -2 = (length – 2) index.

☑ SLICING WITH SKIP VALUE

We can provide a skip value as a part of our slice like this:

```
word = "amazing"
# word[start, end, skip]
```

```python
print(word[1: 6: 2]) # "mzn"
```

Other advanced slicing techniques:

```python
print(word[-1])  # 'g'
print(word[3])   # 'z'
print(word [:7]) # word [0:7] – 'amazing'
print(word [0:]) # word [0:7] – 'amazing'
```

☑ STRING FUNCTIONS

Some of the commonly used functions to perform operations on or manipulate strings are as follows:

```python
# Define a sample string
text = "  hello world!  "

# 1. str.upper()
print(text.upper())
# Output: "  HELLO WORLD!  "
# Converts all characters to uppercase.

# 2. str.lower()
print(text.lower())
# Output: "  hello world!  "
# Converts all characters to lowercase.

# 3. str.capitalize()
print(text.capitalize())
# Output: "  hello world!  "
# Capitalizes the first character of the
string.

# 4. str.title()
print(text.title())
# Output: "  Hello World!  "
# Capitalizes the first character of each
word.

# 5. str.strip()
print(text.strip())
# Output: "hello world!"
# Removes leading and trailing whitespace.

# 6. str.lstrip()
print(text.lstrip())
# Output: "hello world!  "
# Removes leading whitespace. front spaces

# 7. str.rstrip()
print(text.rstrip())
# Output: "  hello world!"
# Removes trailing whitespace. back spaces

text1 = "hello world, world"
# 8. str.find(sub)
print(text1.find("world"))
# Output: 6
```

```python
# 13. str.split(sep)
print(text.split())
# Output: ['hello', 'world!']
# Splits the string at whitespace (default
separator).

# 14. str.rsplit(sep, maxsplit)
# Define the string
full_name = "John Jacob Jingleheimer Schmidt"
# Split the string at the last two spaces
first_part, middle_name, last_name =
full_name.rsplit(' ', 2) # split starts from the
last speace

# Print the results
print("First Part :", first_part) # John Jacob
print("Middle Name :", middle_name) # Jingleheimer
print("Last Name :", last_name) # Schmidt

# 15. str.splitlines()
multi_line_text = """line1
line2
line3"""
print(multi_line_text.splitlines())
# Output: ['line1', 'line2', 'line3']
# Splits the string at line breaks.

# 16. str.join(iterable)
words = ["hello", "world"]
print(" ".join(words))
# Output: "hello_world"
# Joins elements of the list into a single string
with a space as separator.

# 17. str.isdigit()
print("123".isdigit())
# Output: True
# Returns True if all characters are digits.

# 18. str.isalpha()
print("hello".isalpha())
# Output: True
# Returns True if all characters are alphabetic.
# "world" first occurs at index 6.

# 9. str.rfind(sub)
print(text1.rfind("world"))
# Output: 13
# "world" last occurs at index 13.

# 10. str.index(sub)
print(text1.index("world"))
# Output: 6
# Similar to find(), but raises ValueError
if "world" is not found.

# 11. str.rindex(sub)
print(text1.rindex("world"))
# Output: 13
# Similar to rfind(), but raises
ValueError if "world" is not found.

# 12. str.replace(old, new)
print(text.replace("world", "Python"))
# Output: "  hello Python!  "
# Replaces occurrences of "world" with
"Python".

# String.endswith("gar") - This function_
tells whether the variable string ends
str = "sagar"
print(str.endswith("gar")) # Output: True
```

```
# 19. str.islower()
print("hello".islower())
# Output: True
# Returns True if all characters are lowercase.

# 20. str.isupper()
print("HELLO".isupper())
# Output: True
# Returns True if all characters are uppercase.

# 21. str.isspace()
print("   ".isspace())
# Output: True
# Returns True if all characters are whitespace.

# 22. str.zfill(width)
print("42".zfill(5))
# Output: "00042"
# Pads the string with zeros on the left to make it
of length `width`.

### string.count("r") - counts the total number of
occurrences of any character.
str = "sagar"
count = str.count("r")
print(count) # Output: 1
```

☠ pop() works on list, dictionary, set. not work with string, tuple.

☑ ESCAPE SEQUENCE CHARACTERS

Sequence of characters after backslash "\" → Escape Sequence characters Escape Sequence characters comprise of more than one character but represent one character when used within the strings.

#. Escape Sequence:

ITEM	Meaning
\b	backspace
\t	tab
\n	newline
\r	carriage return
\”	double quote
\\	backslash
\’	single quote

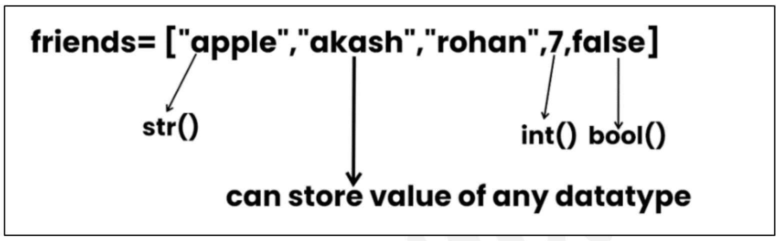
Part 4 – LISTS AND TUPLES

- Python lists are containers to store a set of values of any data type.

☑ LIST INDEXING

♠ A list can be indexed just like a string.

```
frinds = ["sakib", "Asif", 5, 3.24, False, "Nazmul"]

marks = [["Sagar Biswas", 100],["Nazmul", 80]] ### List of Lists

print(marks)

print(frinds[0])
frinds[0] = "Xavi"

print(frinds[0])
print(frinds[1:4])
```


friends= ["apple","akash","rohan",7,false]
str()
int()
bool()
can store value of any datatype

- list can be changed
- If you need in ORDERED, then you must use list.
- len() work on string, set , string, list, dictionary.

☑ LIST METHODS:

```
# Create a list of numbers
numbers = [10, 5, 20, 15]

# Find the maximum value
max_value = max(numbers)
print("Maximum value:", max_value)
# Find the minimum value
min_value = min(numbers)
print("Minimum value:", min_value)

# Get the total length of the list
num_elements = len(numbers)
print("Number of elements:", num_elements)

# Append an element to the list
numbers.append(25)
print("Updated list after appending:", numbers)

# Extend the list with more elements
more_numbers = [30, 35]
numbers.extend(more_numbers)
print("Updated list after extending:", numbers)

# Sort the list in ascending order
numbers.sort()
print("Sorted list:", numbers)

# Reverse the order of elements in the list
numbers.reverse()
print("Reversed list:", numbers)

# Insert a value (1111) at a specific index (3)
numbers.insert(3, 1111)
print("List after inserting 1111 at index 3:",
numbers)

# Remove the element at index 3 using pop()
removed_value = numbers.pop(3)
print("Removed value at index 3:",
removed_value)
print("Updated list after pop:", numbers)
# Remove the value 10 from the list using
remove()
numbers.remove(10)
print("Updated list after removing 10:",
numbers)
```

Output:

```
Maximum value: 20
Minimum value: 5
Number of elements: 4
Updated list after appending: [10, 5, 20, 15, 25]
Updated list after extending: [10, 5, 20, 15, 25, 30, 35]
Sorted list: [5, 10, 15, 20, 25, 30, 35]
Reversed list: [35, 30, 25, 20, 15, 10, 5]
List after inserting 1111 at index 3: [35, 30, 25, 1111,
20, 15, 10, 5]
Removed value at index 3: 1111
Updated list after pop: [35, 30, 25, 20, 15, 10, 5]
Updated list after removing 10: [35, 30, 25, 20, 15, 5]
```

TUPLES IN PYTHON:

- A tuple is an immutable data type in python.
- string, tuples can’t be changed (faster due to immutability)

```
a = ()                    # empty tuple

a = (1,)
# tuple with only one element needs a comma, without comma “a” act like a single int.

a = (1,7,2)               # tuple with more than one element

a = ()                    # Empty tuple
print(a)                  # output: ()

a = (1)
print(type(a))            # output: <class 'int'>

a = (1,)
print(type(a))            # output: <class 'tuple'>
```

TUPLE METHODS:

```
# Creating a tuple
my_tuple = (1, 2, 3, 4, 5)

# Indexing
print("Indexing:")
print(my_tuple[0])          # Output: 1
print(my_tuple[1:3])        # Output: (2, 3)

# Counting occurrences of a value
my_tuple = (1, 2, 3, 2, 2, 4)
print("\nCount:")
print(my_tuple.count(2))    # Output: 3

# Finding the first index of a value
my_tuple = (1, 2, 3, 2, 4)
print("\nIndex:")
print(my_tuple.index(2))    # Output: 1

# Getting the length of the tuple
my_tuple = (1, 2, 3)
print("\nLength:")
print(len(my_tuple))        # Output: 3

# Getting the minimum value in the tuple
print("\nMinimum value:")
print(min(my_tuple))        # Output: 1
```

```
# Getting the maximum value in the tuple
print("\nMaximum value:")
print(max(my_tuple))        # Output: 3

# Getting the sum of all items in the tuple
print("\nSum:")
print(sum(my_tuple))        # Output: 6

# Checking if an element exists in the tuple
print("\nElement existence check:")
print(2 in my_tuple)        # Output: True
print(4 in my_tuple)        # Output: False

# Concatenating two tuples
tuple1 = (1, 2, 3)
tuple2 = (4, 5, 6)
combined = tuple1 + tuple2
print("\nConcatenation:")
print(combined)        # Output: (1, 2, 3, 4, 5, 6)

# Repeating the elements of a tuple
repeated = my_tuple * 3
print("\nRepetition:")
print(repeated)             # Output: (1, 2, 3, 1,
2, 3, 1, 2, 3)
```

PART 5 – DICTIONARY & SETS

☠ Dictionary is a collection of keys-value pairs.

Code:

```
marks = {
    "Sagar": 100, # Sagar, Asifm karim are the key of the dictionary.
    "Asif": 56,
    "Karim": 35,
    "List": [1,2,3], # storing list in a dictionary
    0: "Shisher",
    0.1: "Nazmul"
}

emptyDic = {} # Empty Dictionary.
print(type(emptyDic))
print(f"Empty Dictionary: {emptyDic}")


print (marks, type(marks))
# print(marks[0]) ### NOT WORKING AS LIKE THE (LIST, TUPLE AND STRINGS)
# [USE KAY NOT THE INDEX]
print(marks["Sagar"])
print(marks["List"])
```

☑ PROPERTIES OF PYTHON DICTIONARIES:

1. It is unordered.
2. It is mutable.
3. It is indexed (By keyValue).
4. Cannot contain **duplicate** keys.

☑ DICTIONARY METHODS:

```
# Creating a dictionary
d = {'a': 1, 'b': 2, 'c': 3}

print("\nLenth of the dictionary value: ")
print(len(d))

# Getting all keys
print("\nKeys:")
print(d.keys())  # Output: dict_keys(['a',
'b', 'c'])
print(list(d.keys()))  # Output: ['a', 'b',
'c']

# Getting all values
print("\nValues:")
```

```
# Popping a specific key
print("\nPop key 'c':")
value = d.pop('c')
print(value)  # Output: 3
print(d)  # Output: {'a': 1, 'b': 2, 'd': 4, 'e': 5}

# Popping an item from the last. (key-value pair)
print("\nPop an item:")
item = d.popitem()
print(item)  # Output: ('e', 5)
print(d)  # Output: {'a': 1, 'b': 2, 'd': 4}

# Clearing the dictionary
print("\nClear dictionary:")
d.clear()
print(d)  # Output: {}
```

```python
print(d.values())  # Output: dict_values([1, 2, 3])
print(list(d.values()))  # Output: [1, 2, 3]

# Getting all items (key-value pairs)
print("\nItems:")
print(d.items())  # Output: dict_items([('a', 1), ('b', 2), ('c', 3)])
print(list(d.items()))  # Output: [('a', 1), ('b', 2), ('c', 3)]

# Getting the value for a specific key
print("\nGet value for key 'a':")
print(d.get('a', 'Not Found'))  # Output: 1
print(d.get('z', 'Not Found'))  # Output: Not Found

# Setting a default value for a key
print("\nSet default value for key 'd':")
print(d.setdefault('d', 4))  # Output: 4
print(d)  # Output: {'a': 1, 'b': 2, 'c': 3, 'd': 4}

# Updating the dictionary with another dictionary
print("\nUpdate dictionary:")
d.update({'e': 5})
print(d)  # Output: {'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5}
```

```python
# Copying the dictionary
print("\nCopy dictionary:")
d = {'a': 1, 'b': 2, 'c': 3}
d2 = d.copy()
print(d2)  # Output: {'a': 1, 'b': 2, 'c': 3}

# Creating a dictionary from keys
print("\nCreate dictionary from keys:")
new_dict = dict.fromkeys(['a', 'b', 'c'], 0)
print(new_dict)  # Output: {'a': 0, 'b': 0, 'c': 0}

# why are we using list()? to remove dict_values, dict_items???
:-->
# the list() function is used to convert the view objects
# (dict_keys, dict_values, dict_items) into lists, which makes them easier to read and understand when printed.
```

☑ SETS IN PYTHON

- ☠ set is a collection of non-repetitive elements.

```python
# Creating a set with unique elements
      set1 = {1, 5, "Sagar", 4.56, 5.34, 32, 54, 2}
      print(set1)  # Output: {32, 1, 2, 5, 54}

set1= {1, 3, 4, 2, 5,} # Output {1, 2, 3, 4, 5} // Automatic ordered.. but this is not guaranteed.
print(set1)

# Creating an empty set
emptySet = set()
print(emptySet)  # Output: set()

# Creating a set with repeating elements
# Repeating elements are automatically removed in a set
s1 = {3, 2, 1, 2, 3, 1, 2, 3, 2, 2, 1, 1, 3, 3}
print(s1)  # Output will be {1, 2, 3} removed duplicate values

s1 = {'d', 'b', 'c', 'a', 'b', 'f', 'e', 'a'}
print(s1) # {'a', 'e', 'f', 'c', 'd', 'b'} --> not ordered and auto removed duplicate values
```

If you are new to programming and not familiar with mathematical operations on sets, you can think of sets in Python as collections of unique values, meaning each value appears only once in the set.

☑ PROPERTIES OF SETS:

1. Sets are unordered => Element’s order doesn’t matter
2. Sets are unindexed => Cannot access elements by index
3. There is no way to change items in sets.
4. Sets cannot contain **duplicate** values.

☑ SET METHODS:

```
# Creating sets
set1 = {5, 3, 1, 4, 2, 10}
print("Original set1:")
print(set1)  # Output may vary, not
necessarily sorted

# lenth of the set
print("\nLenth of the set value: ")
print(len(set1))

# Adding an element
set1.add(6)
print("\nAfter adding 6:")
print(set1)  # Output will include 6

# Removing a specific element
set1.remove(3) # Raises a KeyError if the
element is not present in the set.
print("\nAfter removing 3:")
print(set1)  # Output will not include 3
# Discarding an element
set1.discard(10)  # 10 is in the set!
print("\nAfter discarding 10:")
print(set1)  # Output: {5, 3, 1, 4, 2}

# Discarding an element (does not raise an
error if the element is not present)
set1.discard(11)  # 11 is not in the set, so
no error!
print("\nAfter discarding 11:")
print(set1)  # Output remains the same

# Popping an arbitrary element
popped_elem = set1.pop() # Raises a KeyError
if the set is empty. The element removed is
arbitrary, meaning there is no guarantee
which element will be removed.
print("\nPopped element:")
print(popped_elem)  # Output will be an
arbitrary element, Output: 1
print("Set after popping an element:")
print(set1)  # Output without the popped
element, output: {2, 4, 5, 6}

# Clearing all elements
set1.clear()
```

```
# Intersection of sets
intersection_set = set1.intersection(set2)
print("\nIntersection of set1 and set2:")
print(intersection_set)  # Output: {3}

# Difference of sets
set11 = {1, 2, 3}
set22 = {2, 3, 4}
difference_set = set11.difference(set22) # Find
what’s uncommon in first set "set11" but not in
another set.
print("\nDifference of set11 and set22:")
print(difference_set)  # Output: {1}
set11 = {1, 2, 3}
set22 = {2, 3, 4}
difference_set = set22.difference(set11) # Find
what’s uncommon in first set "set22"  but not in
another set.
print("\nDifference of set22 and set11:")
print(difference_set)  # Output: {4}
# Symmetric difference of sets
symmetric_difference_set =
set11.symmetric_difference(set22) # Find what’s
uncommon in both sets.
print("\nSymmetric difference of set11 and set22:")
print(symmetric_difference_set)  # Output: {1, 4}

# Checking subset
s1 = {1, 2}
print("\nIs set1 a subset of set2?")
print(set1.issubset(set2))  # Output: False
print("\nIs s1 a subset of set11?")
print(s1.issubset(set11))  # Output: true

# Checking superset
print("\nIs set1 a superset of the intersection
set?")
set1 = {1, 2, 3}
intersection_set = {3}
print(set1.issuperset(intersection_set))  # Output:
True

# Checking disjoint sets
# disjoint set refers to two sets that have no
elements in common
```

```python
print("\nAfter clearing the set:")
print(set1)  # Output: set()

# Copying the set
set1 = {1, 2, 3}
set1_copy = set1.copy()
print("\nCopy of set1:")
print(set1_copy)  # Output: {1, 2, 3}

# Union of sets
set2 = {3, 4, 5}
union_set = set1.union(set2)
print("\nUnion of set1 and set2:")
print(union_set)  # Output: {1, 2, 3, 4, 5}
```

```python
set3 = {6, 7}
print("\nAre set1 and set3 disjoint?")
print(set1.isdisjoint(set3))  # Output: True
print("\nAre set1 and set2 disjoint?")
print(set1.isdisjoint(set2))  # Output: false


# Operator in set

  s1 = {1, 2}
  s2 = {1, 2, 3}
  x = s1-s2
  y = s2-s1

print("set operator: ")
print (x) # Output: set()
print (y) # Output: {3}
```

Explanation:

```
add()                : Adds an element to the set.
remove()             : Removes a specific element (raises KeyError if not present).
discard()            : Removes a specific element (does nothing if not present).
pop()                : Removes and returns an arbitrary element.
clear()              : Removes all elements.
copy()               : Creates a copy of the set.
union()              : Returns a new set with elements from both sets.
intersection()       : Returns a new set with elements common to both sets.

difference() : Returns a new set with elements in the first set but not in the second set.
               symmetric_difference(): Returns a new set with elements in either set but not
               in both.

issubset()           : Checks if all elements of the set are in another set.
issuperset()         : Checks if all elements of another set are in the set.
isdisjoint()         : Checks if the set has no elements in common with another set.
```

Part 6 – CONDITIONAL EXPRESSION

Sometimes we want to play PUBG on our phone if the day is Sunday.

Sometimes we order Ice Cream online if the day is sunny.

Sometimes we go hiking if our parents allow.

All these are decisions which depend on a condition being met.

In python programming too, we must be able to execute instructions on a condition(s) being met.

~This is what conditionals are for!

☑ IF ELSE AND ELIF IN PYTHON

- If else and elif statements are a multiway decision taken by our program due to certain conditions in our code.

Quick Quiz: Write a program to print yes when the age entered by the user is greater than or equal to 18.

☑ RELATIONAL OPERATORS:

- Relational Operators are used to evaluate conditions inside the if statements. Some

  examples of relational operators are:

    - == equals.
    - >= greater than/ equal to.
    - <= lesser than/ equal to.

LOGICAL OPERATORS:

In python logical operators operate on conditional statements. For Example:

- and – true if both operands are true else false.
- or – true if at least one operand is true or else false.
- not – inverts true to false & false to true.

☑ ELIF CLAUSE:

- elif in python means [else if]. An if statements can be chained together with a lot of these elif statements followed by an else statement.

☑ IMPORTANT NOTES:

1. There can be any number of elif statements.

2. Last else is executed only if all the conditions inside elifs fail.

Example: (if, elif and else)

```
fn = int(input("Enter the number: "))
ln = int(input("Enter the number: "))

condition1 = ln < fn
condition2 = ln > fn

if (condition1):     # if condition1 is True
    print ("Frist entered number is greater")
elif(condition2):   # if condition2 is True
    print("Last entered number is greater")
else:                # otherwise
    print("Frist and Last entered numbers both are equal")
```

```
a=22

if(a>9):
print("greater")

else:
print("lesser")
```

Part 7 – LOOPS IN PYTHON

Sometimes we want to repeat a set of statements in our program. For instance: Print 1 to 1000.

☠ Loops make it easy for a programmer to tell the computer which set of instructions to repeat and how!

☑ TYPES OF LOOPS IN PYTHON

💣 Primarily there are two types of loops in python.

- while loops
- for loops

We will investigate these one by one.

Syntax:

```
    while (condition): # the while loop continues to execute the block of code as long as the condition is True.
            # Body of the loop
```

☠ In while loops, the condition is checked first. If it evaluates to true, the body of the loop is executed otherwise not!

☠ If the loop is entered, the process of [condition check & execution] is continued until the condition becomes False.

Quick Quiz: Write a program to print 1 to 50 using a while loop

Example:

```
    i = 0
    while i < 5: # print "Sagar" – 5 times!
        print("Sagar")
        i = i + 1 # or i +=1a
```

☒ Note: If the condition never becomes false, the loop keeps getting executed.

Quick Quiz: Write a program to print the content of a list using while loops.

☑ FOR LOOP

☠ A for loop is used to iterate through a sequence like list, tuple, or string [iterables]

Syntax:

```
    l = [1, 7, 8]
    for item in l:
        print(item) # prints 1, 7 and 8
```

☑ RANGE FUNCTION IN PYTHON

The range() function in python is used to generate a sequence of number. We can also specify the start, stop and step-size as follows:

```
range(start, stop, step_size)
# step_size is usually not used with range()
```

☑ AN EXAMPLE DEMONSTRATING RANGE () FUNCTION.

```
for i in range(0,7): # range(7) can also be used.
    print(i) # prints 0 to 6
```

☑ FOR LOOP WITH ELSE

The else block is executed when the loop completes normally (i.e., without encountering a break statement).

Example:

`l= [1,7,8]`
`for item in l:`
`print(item)`
`else:`
`print("done") # this is printed when the loop exhausts!`
	Output:
	1
	7
	8
	done

☑ FOR LOOP Codes:

```
#.........................................
# simple for loop
for i in range(11):
    print(i)
# i += 1 # this line will auto prossesed in
for loop
#.........................................

# we can print numbers in one line by using
the end parameter of the print function:
for i in range(11):
    print(i, end=' ')
#.........................................

# range:
for i in range(100, 120): # i = 100 to 119
    print(i, end=' ') # Output: 100 to 119

# range(start, stop, step_size)
# step_size is usually not used with
range()
for i in range(0, 50, 4):
    print(i, end=' ') # Output: 0 4 8 12 16
20 24 28 32 36 40 44 48
#.........................................

# for loop (iterate)
list = [1,3,5,7,9,11] # list
for item in list:
    print(item, end = " ")
# for loop (iterate)
```

```
# Display with string.
for i in range(11):
    print(f"{i}. Sagar Biswas")
#.........................................

### for loop with else: (MOST UNCOMMON)-- Important fo
rinterview
l = [1,3,5,7,9,11] # list
for i in l:
    print(i, end = " ")
else:
    print("Done..")
#.........................................

# break statement...
l = [1,3,5,7,9,11] # list
for i in l:
    print(i, end = " ")
    if(i == 7):
        break
else: # don't go to else because i==7 matched. then
breaked...
    print("Done..")
#................................................
# continue statement
l = [1,3,5,7,9,11] # list
for i in l:
    if(i == 7):
        continue # continue skip this itertion...
    print(i, end = " ")
#.........................................
```

```
t = (1,3,5,7,9,11) # tuple
for item in t:
    print(item, end = " ")

# for loop (iterate)
s = "Sagar Biswas" # string
for item in s:
    print(item, end = " ")
#..........................................
```

```
# pass statement(pass = null statement, instructs to
"do nothing")
for i in range(500):
    pass # will go to next code (pass will skip this
loop to finish it letter)
i = 1 # start
# range(stop, step_size)
for i in range(11, 2):
    print(i)
#..........................................
```

☑ THE BREAK STATEMENT

- It instructs the program to –exit the loop now.

Example:

```
for i in range (0,80):
      print(i) # this will print 0,1,2 and 3
      if i==3:
            break
```

☑ THE CONTINUE STATEMENT

‘continue’ is used to stop the current iteration of the loop and continue with the next one. It instructs the Program to “skip this iteration”.

Example:

```
for i in range(4):
      print("printing") # This line will print 4 times. Because it stays on the upper
side of continue keyword
      if i == 2: # if i is 2, the iteration will be skipped
            continue
            print(i) # 0,1,3
```

☑ PASS STATEMENT

- pass is a null statement in python.
- It instructs to “do nothing”.

```
l = [1,7,8]
for item in l:
      pass # without pass, the program will throw an error.
```

Part 8 – FUNCTIONS & RECURSIONS

- A function is a group of statements performing a specific task.

- When a program gets bigger in size and its complexity grows, it gets difficult for a program to keep track on which piece of code is doing what!
- A function can be reused by the programmer in a given program.

☑ EXAMPLE AND SYNTAX OF A FUNCTION

The syntax of a function looks as follows:

```
# User defined function.
# Function defination
def avg():
    a = int(input("Enter the number: "))
    b = int(input("Enter the number: "))
    c = int(input("Enter the number: "))

    average = (a+b+c)/3
    print(f"The average number: {average} \n")

# function call
avg()  # This line calls the function, so the code inside the function will run
print("Thanks for your contribution.")
avg()  # This line calls the function again
avg()  # This line calls the function a third time
```

This function can be called any number of times, anywhere in the program.

☑ FUNCTION DEFINITION

- The part containing the exact set of instructions which are executed during the function call.

Quick Quiz: Write a program to greet a user with "Good day" using functions.

☑ TYPES OF FUNCTIONS IN PYTHON

- There are two types of functions in python:
    - Built in functions (Already present in python)
    - User defined functions (Defined by the user)

Examples of built in functions includes len(), print(), range() etc.

The avg(): function we defined is an example of user defined function.

☑ FUNCTIONS WITH ARGUMENTS

- A function can accept some value it can work with. We can put these values in the parentheses.
- A function can also return value as shown below:

```
print("\n")
# single perimeter
def goodDay(name):
    print("Good Day " + name)
```

```
goodDay("Sagar")
goodDay("Mr.Been")

print("\n")
# duble perimeters
def goodDay(name, ending):
    print("Good Day " + name)
    print(ending)

goodDay("Sagar", "Thank You.")
goodDay("Mr.Been", "Thanks.")
```

```
Output:
Good Day Sagar
Good Day Mr.Been


Good Day Sagar
Thank You.
Good Day Mr.Been
Thanks.
```

☑ DEFAULT PARAMETER VALUE

- We can have a value as default as default argument in a function.

If we specify `ending="Thank Youuuuuuuuuu....")` in the line containing def, this value is used when no argument is passed.

Example:

```
print("\n")
def goodDay(name, ending="Thank Youuuuuuuuuu...."): # ending default perimeter
    print("Good Day " + name)
    print(ending)
    print("\n")

goodDay("Sagar") # this with prints with the default perimeter

goodDay("Asshole", "Thanks.") # This with prints without the default perimeter. Because the value of ending is assigned here.
```

☑ FUNCTIONS WITH RETURN VALUE:

- A function that produces and sends back a value after its execution.
- The return value of a function can be used in various ways, such as assigning it to a variable, using it in expressions, within other function calls, or in conditional statements.

```
Example 1:

# Non-return function...
print("\n")
def goodDay(name, ending):
    print("Good Day " + name)
    print(ending)
    # return "Done" # Not returning any value/(s).

a = goodDay("Sagar", "Thank You.")
print(a)   # Prints "Good Day Sagar" and "Thank You." # Prints With the return value [None] because the goodDay() is assigned with ‘a’ variable and also for the print()

goodDay("Mr.Been", "Thanks.") # prints without return value. because goodDay() is not assigned to a variable and also for not using the print()
```

```
print(goodDay("Anis", "Thanks Boss")) # Prints With the return value [None] because of print()
```

Example 2:

```
# return function
print("\n")

def goodDay(name, ending):
    print("Good Day " + name)
    print(ending)
    return "Done"  # returned value

a = goodDay("Sagar", "Thank You.")
print(a)  # Prints "Good Day Sagar" and "Thank You." from the function call, and then # Prints
With the return value [Done]. because the goodDay() is assigned with ‘a’ variable and also for
the print()

goodDay("Mr.Been", "Thanks.")  # prints without return value.

print(goodDay("Anis", "Thanks Boss")) # Prints With the return value [Done] because of print()
```

Example 3:

```
# A perfect example of a return function

        def avg():
            a = int(input("Enter the number: "))
            b = int(input("Enter the number: "))
            c = int(input("Enter the number: "))

            average = (a+b+c)/3
            return average

        # function calling…
        print(f"The average number: {avg()} \n")  # avg() returns with the average value. avg()
        will return None if there is no return value.
        print("Thanks for your contribution.")
        print("\n")
        a = avg() # avg() returns with average and storing the average value in ‘a’ variable.
        avg() will return None if there is no return value.

        print(f"The average number: {a} \n")
```

☑ RECURSION

- ☠ Recursion is a function which calls itself.
- ☠ It is used to directly use a mathematical formula as function.

Example:

```
'''
factorial(0) = 1
factorial(1) = 1
factorial(2) = 2 X 1
factorial(3) = 3 X 2 X 1
factorial(4) = 4 X 3 X 2 X 1
factorial(5) = 5 X 4 X 3 X 2 X 1
factorial(n) = n X n-1 X (n-2) X (n-3) ... X 1 # keep doing this till n=1 OR n= 0.

'''
```

Code:

```python
def factorial(n):
    if(n==1 or n==0):
        return 1
    # return n*factorial(n-1) # will work as the else statement.
    else:
        return n*factorial(n-1)

n = int(input("Enter the number: "))
print(f"The factorial of the number is: {factorial(n)}")
```

The programmer needs to be extremely careful while working with recursion to ensure that the function doesn't infinitely keep calling itself. Recursion is sometimes the most direct way to code an algorithm.

PROJECT 1: SNAKE, WATER, GUN GAME

We all have played snake, water, gun game in our childhood. If you haven't, google the rules of this game and write a python program capable of playing this game with the user.

main.py

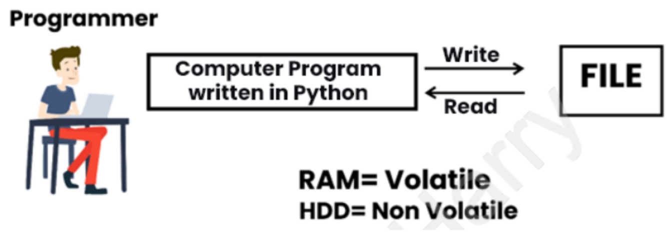
```python
'''
Remember the rules:

Snake   drinks  Water       and wins.
Water   drowns  the Gun     and wins.
Gun     shoots  the Snake   and wins.

Suppose:

    Snake = 1, Water = 0, Gun = -1

'''


import random

'''
The easier code:

From ---------------------------------------------- The advanced code's easier version.

if(you == computer):
    print("Match Draw. ")
```

```
else:
    if(you == 1 and computer == 0):  # 1+0 = 1        # 1+(-0)   = 1
        print("You Win!") # Win........

    elif(you == 1 and computer == -1): # 1+-1 = 0       # 1+(-(-1) = 2
        print("You Lose!")

    elif(you == 0 and computer == 1):  # 0+1 = 1        # 0+(-1)   = -1
        print("You Lose!")

    elif(you == 0 and computer == -1): # 0-1 = -1       # 0+(-(-1) = 1
        print("You Win!") # Win........

    elif(you == -1 and computer == 0): # -1+0 = -1      # -1+(-0)  = -1
        print("You Lose!")

    elif(you == -1 and computer == 1): # -1+1 = 0       # -1+(-1)  = -2
        print("You Win!") # Win........

    else:
        print("Something is wrong...")
To ---------------------------------------------- The advanced code's easier version.

# First approach will not work. (because 1,0,-1 all for win)
# Secound approach will work.    (because of only 1,-2 you can win)
# So, now                        (The Most Small, Complex, Time-Efficient: approach):
'''

# The advanced code

print("\n")
n = int(input("..:: The Total Point: ")) # n -- the number of loops (How my match wanna play).
print("\n")

# For keeping track of the number of draws and wins.
draw = 0
win = 0

for i in range(1, n+1): #if n=10, the loop will iterate from 1 to 10.

    yourInput = input(f"{i}. Enter your choice (S for Snake, W for Water, G for Gun): ")
    print("\n")
    yourInput = yourInput[0].lower() # converting to lower letter to avoid any error.

    if yourInput in ("s", "w", "g"):

        yourDict = {"s": 1, "w": 0, "g": -1}
        you = yourDict[yourInput]

        optionsStr = {1: "Snake", 0: "Water", -1: "Gun"}
        computer = random.choice([1, 0, -1])

        print(f"---> You Chose: {optionsStr[you]} AND The Computer Chose: 
{optionsStr[computer]}\n")
# ---------------------------------------------- The easier code's advanced version.
        if(you == computer):
            print("---> Match Draw. ")
            draw += 1
        else:
            if(you - computer == 1 or you - computer == -2):
                print("---> You Win!") # Win......  for only[1 and -2]
                win +=1
            else:
                print("---> You Lose!")
# ---------------------------------------------- The easier code’s advanced version.

        print("\n")
        lose = n-(win+draw) # for wrong input win and draw will not be increased. So, without
WIN and DRAW everything(errors & loses) will count as LOSE....


    else:
        print("Invalid input. Please choose S, W, or G.\n")

print(f"..:: Total Win/(s): {win}, Lose/(s): {lose} and Draw/(s): {draw} in {n} Points\n\n")
```

Part 9 – FILE I/O

"The random-access memory is volatile, and all its contents are lost once a program terminates. In order to persist the data forever, we use files." If a file is data stored in a storage device. A python program can talk to the file by reading content from it and writing content to it.

☑ TYPE OF FILES.



- There are 2 types of files:
  1. Text files (.txt, .c, etc)
  2. Binary files (.jpg, .dat, etc)

- Once a program finishes execution, the operating system typically frees up the memory allocated to that program.
- Python has a lot of functions for reading, updating, and deleting files.

☑ OPENING A FILE

Python has an open() function for opening files. It takes 2 parameters: filename and mode.

```
# open("filename", "mode of opening(read mode by default)")
open("this.txt", "r")
```

☑ READING A FILE IN PYTHON

```
# Open the file in read mode
f = open("this.txt", "r")
# Read its contents
text = f.read()
# Print its contents
print(text)
```

```
# Close the file
f.close()
```

☑ OTHER METHODS TO READ THE FILE.

- We can also use f.readline() function to read one full line at a time.

```
f.readline() # Read one line from the file.
```

☑ MODES OF OPENING A FILE

r	- open for reading
w	- open for writing
a	- open for appending
+	- open for updating.
‘rb’	- will open for read in binary mode.
‘rt’	- will open for read in text mode

☑ WRITE FILES IN PYTHON:

In order to write to a file, we first open it in write or append mode after which, we use the python’s f.write() method to write to the file!

```
# Open the file in write mode
f = open("P:/Codes/Practice.txt", "w")
# Write a string to the file
f.write("This is a nice note :)")
f.close()
```

FILE FUNCTIONS:

```
# Import the os module to check for file existence
import os

# Define the file path
filepath = 'P:/Codes/Chapter 9/file/myfile.txt'

print("\n")
# Open a file in write mode and write some text
file = open(filepath, 'w')                          # 'w' --> write mode
file.write('Hello, world!\n') # write
file.writelines(['First line\n', 'Second line\n']) # writelines/readlines --> list
file.close()

print("..:: File written successfully.\n")

# Check if the file exists
if os.path.exists(filepath):
    print("..:: File exists. Reading file content: \n")

    # Open the file in read mode and read its content
    file = open(filepath, 'r')                      # 'r' --> read mode
    content = file.read()
    file.close()
    print(content)
    print(f"..:: The type of the content is: {type(content)} \n") # <class 'str'>
else:
    print("..:: File does not exist")

# Open the file in append mode to add more text at the end.
file = open(filepath, 'a')                          # 'a' --> append mode
file.write('Appending a new line\n')
file.close()
print("..:: File appended successfully.\n")

# Read the file again to check the appended content
file = open(filepath, 'r')
content = file.readlines()                            # readlines --> list
file.close()
print("..:: Updated file content: \n")
print(content)
print("") # Output comes as a list.

'''    file = open(filepath)
      line1 = file.readline()
      print(line1)
      line2 = file.readline()
      print(line2)
      .
      .
      .
      print(line5 == "") # returning True as an empty string. because line5 doesn't exists.
      print()

      file.close() '''


# Same purpose but here we using while loop...
'''    #readline() --> using while loop
      file = open(filepath)
      line = file.readline()
      print("..:: The lines are: \n")
      while(line!=""):
          print(line)
          line = file.readline()   '''
```

☑ WITH STATEMENT

The best way to open and close the file **automatically** is the with statement.

```
# the same can be written using with statement like this:
with open("P:/Codes/Chapter 9/file/myfile.txt") as f:
    print(f.read())
# dont have to explicitly close the file.
```

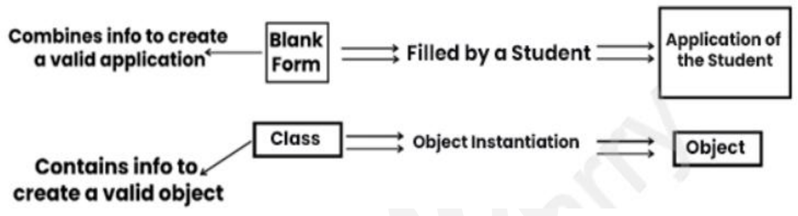
You can now use multiple context managers in a single with statement more cleanly using the parenthesised context manager

```
with (
      open('file1.txt') as f1,
      open('file2.txt') as f2
):
# Process files
```

Part 10 - OBJECT ORIENTED PROGRAMMING

Solving a problem by creating object is one of the most popular approaches in programming. This is called object-oriented programming.

This concept focuses on using reusable code (DRY Principle).



Combines info to create a valid application
Blank Form
Filled by a Student
Application of the Student
Contains info to create a valid object
Class
Object Instantiation
Object

☑ CLASS

A class is a blueprint for creating object.
Syntax:

```
class Employee: # Class name is written in pascal case
    # Methods & Variables
```

☑ OBJECT

☠ An object is an instantiation of a class. When class is defined, a template (info) is defined. Memory is allocated only after object instantiation.

☠ Objects of a given class can invoke the methods available to it without revealing the implementation details to the user. – Abstractions & Encapsulation!

☑ MODELLING A PROBLEM IN OOPS

We identify the following in our problem.

• Noun	→ Class	→ Employee
• Adjective	→ Attributes	→ name, age, salary
• Verbs	→ Methods	→ getSalary(), increment()

☑ CLASS ATTRIBUTES

Class attributes are variables shared among all instances of a class, defined within the class but not inside any methods.
Example 1:

```
class Employee:
    company = "Google"  # Specific to Each Class (Class Attribute)
Sagar = Employee()      # Object Instatiation
Sagar.company
Employee.company = "YouTube" # Changing Class Attribute (Instance Attribute)
```

☑ INSTANCE ATTRIBUTES

Instance attributes are variables that belong to each specific object created from a class, usually defined in the __init__ method(Constructor).

Example2: (along with Example 1):

```
            Sagar.name = "sagar"
            Sagar.salary = "30k" # Adding instance attribute.
            print(Sagar.name, Sagar.company, Sagar.salary)
```

Explanation: (Ex1 & Ex2)

- ☠ Sagar is an instance of the Employee class.
- ☠ name and salary are instance attributes because they are assigned directly to the instance (Sagar) of the class, not to the class itself.
- ☠ Sagar.company is a class attribute/class variable.
- ☠ Sagar.salary is an instance attribute because it is specific to the Sagar instance.

☒ Note: Instance attributes override class attributes with the same name when accessed or assigned for a specific instance.

☒ Note: When looking up for sagar.attribute it checks for the following:

1) Is attribute present in object?

2) Is attribute present in class?

☑ INSTANCE VS CLASS ATTRIBUTE

```
 class Employee:
      name = "Sagar Biswas"    # class variable
      language = "C++"         # class variable
      salary = 100000          # class variable

    def __init__(self):   # constructor
        print ("Name: ", self.name, ", Language: ", self.language, ", Salary: ", self.salary)

Sagar = Employee()          # Constructor calls automatically
Sagar.language = "Python"   # instance variable
Sagar.name = "Bear Grylls"  # instance variable
Sagar.salary = 300000       # instance variable
print("Name: ", Sagar.name,  ", Language: ", Sagar.language, ", Salary: ", Sagar.salary)
```

☒ Note: Sagar.language, Sagar.name, and Sagar.salary are all instance variables because they are assigned directly to the instance Sagar and override the class variables for this specific instance.

Output:

```
Name:  Sagar Biswas , Language:  C++ , Salary:  100000
Name:  Bear Grylls , Language:  Python , Salary:  300000
```

☑ SELF PARAMETER

self refers to the instance of the class. It is automatically passed with a function call from an object.
The function getSalary() is defined as:

```
        class Employee:
                company = "Google"
                def getSalary(self):
                        print("Salary is not there")

        Sagar.getSalary()  # here self is Sagar # equivalent to Employee.getSalary(Sagar)
```
P.T.O

☑ STATIC METHOD

- Sometimes we need a function that does not use the self-parameter.

We can define a static method like this:

```
                @staticmethod # decorator to mark greet as a static method
                def greet():
                print("Hello user")
```

☑ __INIT__() CONSTRUCTOR

- __init__() is a special method which is first run as soon as the object is created.
  (almost always constructor runs at first when the object being created)
- __init__() method is also known as constructor.

It takes 'self' argument and can also take further arguments.

For Example:
```
                class Employee:
                        def __init__(self, name):
                                self.name=name
                        def getSalary(self):
                                ...
                sagar = Employee("sagar")
```

☒ Code: (constructor, static_method, class and instance variable):

```
class Employee:
    name = "Sagar Biswas"
    language = "C++"
    salary = 100000

    # donder methods (satrts with def __ ), only __init__ and __str__ method call
automatically when you will create an object.
    def __init__(self, name, salary, language): # constructor
        self.name = name
        self.salary = salary
        self.language = language
        print ("\nI am creating an object.")
    def display(self): # self is a parameter.
        print(f"The name is {self.name}. language is {self.language}. salary is
{self.salary}")
    def greetM(self): # self parameter automatically passed when object will be called.
        print("Good Morning")

    @staticmethod # we don't need any object's property in this method. so, we used
@staticmethod
    def greetE():
        print("Good Evening")


Sagar = Employee("Mr. Hooked", 1400000, ".bat") # Assigning class variables
Sagar.language = "Python" # Changing the value of 'language' using an instance variable

print(Sagar.name, Sagar.language, Sagar.salary)
Sagar.display() # act as Employee.display(Sagar)...
Sagar.greetM() # auto passed for self parameter..
Sagar.greetE()
# shisher = Employee() # Error! missing 3 required positional arguments.
```

Part 11 - INHERITANCE & MORE ON OOPS

☠ Inheritance is a way of creating a new class from an existing class.

Syntax:

```
class Employee: # Base class
        # Code
class Programmer(Employee): # Derived or child class
        # Code
```

♠ We can use the <u>method and attributes</u> of 'Employee' in 'Programmer' object. Also, we can <u>overwrite or add new attributes and methods</u> in 'Programmer' class.

☑ TYPES OF INHERITANCE

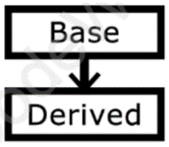
1) Single inheritance
2) Multiple inheritance
3) Multilevel inheritance

☑ <u>SINGLE INHERITANCE</u>

☠ Single inheritance occurs when child class inherits only a single parent class.

```
# Base class
class Animal:
    def speak(self):
        print("Animal speaks")
```

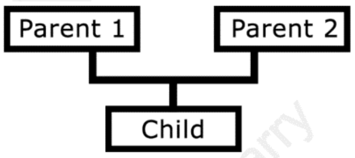
```python
# Derived class
class Dog(Animal):
    def bark(self):
        print("Dog barks")

# Create an instance/object of Dog
dog = Dog()
dog.speak()  # Inherited method
dog.bark()   # Own method
```



☑ MULTIPLE INHERITANCE

- Multiple Inheritance occurs when the child class inherits from more than one parent classes.

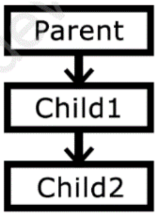
```python
# Base class 1
class Animal:
    def speak(self):
        print("Animal speaks")

# Base class 2
class Pet:
    def play(self):
        print("Pet plays")

# Derived class
class Dog(Animal, Pet):
    def bark(self):
        print("Dog barks")

# Create an instance of Dog
dog = Dog()
dog.speak()  # Inherited from Animal
dog.play()   # Inherited from Pet
dog.bark()   # Own method
```



☑ MULTILEVEL INHERITANCE

- When a child class becomes a parent for another child class.

```python
# Base class
class Grandparent:
    def show_grandparent(self):
        print("Grandparent's trait")

# Intermediate class
class Parent(Grandparent):
    def show_parent(self):
        print("Parent's trait")

# Derived class
class Child(Parent):
    def show_child(self):
        print("Child's trait")

# Create an instance of Child
```



```
child = Child()
child.show_grandparent()  # Inherited from Grandparent
child.show_parent()       # Inherited from Parent
child.show_child()        # Own method
```

☑ SUPER() METHOD

☠ super() method is used to access the methods of a super class in the derived class.

```
                    super().__init__()
                          # __init__() Calls constructor of the base class
```

Example:

```
# Define a base class called Animal
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        return f"{self.name} makes a sound"

# Define a derived class called Dog that inherits from Animal
class Dog(Animal):
    def __init__(self, name, breed):
        # Use super() to call the __init__ method of the parent class (Animal)
        super().__init__(name)
        self.breed = breed

    def speak(self):
        # Use super() to call the speak method of the parent class (Animal)
        return super().speak() + " and barks"

# Create an instance of the Dog class
my_dog = Dog("Buddy", "Golden Retriever")

# Call the speak method on the Dog instance
print(my_dog.speak())  # Output: Buddy makes a sound and barks
```

☑ CLASS METHOD

♠ A class method is a method which is bound to the class and not the object of the class.
♠ @classmethod decorator is used to create a class method.

Syntax:

```
            @classmethod
            def(cls,p1,p2):
```

Example:

Using class method to change class attribute	keep track of the number of instances

`# Define a class called Person`
`class Person:`
`    # Class attribute`
`    species = "Homo sapiens"`
	`# Define a class called Counter`
	`class Counter:`
	`    # Class attribute to keep track of the number of instances`

```python

    # Instance method to initialize the object
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # Class method to change the class attribute
    @classmethod
    def change_species(cls, new_species):
        cls.species = new_species

    # Instance method to display information
    def display_info(self):
        return f"{self.name} is {self.age} years old
and belongs to the species {self.species}"

# Create an instance of the Person class
person1 = Person("Alice", 30)

# Display initial information
print(person1.display_info())  # Output: Alice is 30
years old and belongs to the species Homo sapiens

# Use the class method to change the class attribute
Person.change_species("Homo erectus")

# Display updated information
print(person1.display_info())  # Output: Alice is 30
years old and belongs to the species Homo erectus
```

```python
    instance_count = 0

    # Instance method to initialize the
object
    def __init__(self):
        # Increment the class attribute each
time a new instance is created
        Counter.instance_count += 1

    # Class method to get the current count
of instances
    @classmethod
    def get_instance_count(cls):
        return cls.instance_count

# Create instances of the Counter class
counter1 = Counter()
counter2 = Counter()
counter3 = Counter()

# Use the class method to get the current
count of instances
print(Counter.get_instance_count())  #
Output: 3
```

☑ @PROPERTY DECORATORS/GETER METHOD

Consider the following class:

```python
lass Employee:
    @property
    def name(self):
        return self.ename
```

If e = Employee() is an object of class employee, we can print (e.name) to print the ename by internally calling name() function.

Example:

```python
# 1). Define a class called Circle
class Circle:
    def __init__(self, radius):
        self._radius = radius  # Initialize
the radius attribute

    # Define a property for the radius
    @property
    def radius(self):
        return self._radius
```

```python
# 2). Define a class called Temperature
class Temperature:
    def __init__(self, celsius):
        self._celsius = celsius  # Initialize the
celsius attribute

    # Define a property for celsius
    @property
    def celsius(self):
        return self._celsius
    # Define a setter for the radius property
    @radius.setter
    def radius(self, value):
        if value < 0:
            raise ValueError("Radius cannot
be negative")
        self._radius = value

    # Define a property for the area (read-
only)
    @property
    def area(self):
        return 3.14159 * (self._radius ** 2)

# Create an instance of the Circle class
circle = Circle(5)
# Access the radius property
print(circle.radius)  # Output: 5

# Set the radius property
circle.radius = 10
# Access the updated radius property
print(circle.radius)  # Output: 10
# Access the area property
print(circle.area)  # Output: 314.159

# Attempt to set the area property (will
raise an AttributeError)
# circle.area = 100  # Uncommenting this line
will raise an error
```

```
    # Define a setter for the celsius property
    @celsius.setter
    def celsius(self, value):
        self._celsius = value

    # Define a property for fahrenheit (read-only)
    @property
    def fahrenheit(self):
        return (self._celsius * 9/5) + 32

# Create an instance of the Temperature class
temp = Temperature(25)

# Access the celsius property
print(temp.celsius)  # Output: 25

# Access the fahrenheit property
print(temp.fahrenheit)  # Output: 77.0

# Set the celsius property
temp.celsius = 30

# Access the updated celsius property
print(temp.celsius)  # Output: 30

# Access the updated fahrenheit property
print(temp.fahrenheit)  # Output: 86.0
```

The area property is accessed, but attempting to set it will raise an AttributeError because no setter is defined for it.

☑ @.GETTERS AND @.SETTERS

- The method name with '@property' decorator is called getter method.
- the @property method in Python is used to define getter methods

We can define a function + @ name.setter decorator like below:

```
@name.setter
def name (self,value):
        self.ename = value
```

Example:

```
# Define a class called Rectangle
class Rectangle:
    def __init__(self, width, height):
        self._width = width
        self._height = height

    # Define a property for width (getter)
    @property # get method
    def width(self):
        return self._width
```

```
        if value < 0:
            raise ValueError("Height cannot
be negative")
        self._height = value

    # Define a property for area (read-only)
    @property
    def area(self):
        return self._width * self._height
    # Define a setter for width
    @width.setter
    def width(self, value):
        if value < 0:
            raise ValueError("Width cannot be
negative")
        self._width = value

    # Define a property for height (getter)
    @property
    def height(self):
        return self._height

    # Define a setter for height
    @height.setter
    def height(self, value):
```

```
# Create an instance of the Rectangle class
rect = Rectangle(10, 5)
# Access the width and height properties
print(rect.width)   # Output: 10
print(rect.height)  # Output: 5
# Access the area property
print(rect.area)    # Output: 50

# Set the width and height properties
rect.width = 15
rect.height = 10
# Access the updated properties
print(rect.width)   # Output: 15
print(rect.height)  # Output: 10
print(rect.area)    # Output: 150
```

☑ OPERATOR OVERLOADING IN PYTHON:

- ♠ Operators can be overloaded using dunder methods.
- ♠ These methods are called when a given operator is used on the objects.

☑ Why should we use magic method?

Using dunder methods allows you to define custom behavior for operators, making your code more intuitive P.T.O and readable. Most important for addition, subtraction, multiplication, etc.. with two or more object's values.

Operators in Python can be overloaded using the following methods:

```
p1+p2        # p1.__add__(p2)
p1-p2        # p1.__sub__(p2)
p1*p2        # p1.__mul__(p2)
p1/p2        # p1.__truediv__(p2)
p1//p2       # p1.__floordiv__(p2)
```

Example:

```python
# Define a class called Point
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    # Overload the + operator
    def __add__(self, other):
        print("Other's value: ", other)
        return Point(self.x + other.x,
self.y + other.y)

    # Overload the - operator
    def __sub__(self, other):
        return Point(self.x - other.x,
self.y - other.y)

    # Overload the * operator
    def __mul__(self, other):
        return Point(self.x * other.x,
self.y * other.y)

    # Overload the / operator
    def __truediv__(self, other):
        return Point(self.x / other.x,
self.y / other.y)

    # Overload the // operator
    def __floordiv__(self, other):
        return Point(self.x // other.x,
self.y // other.y)

# Overload the string representation method
for easy printing
    def __str__(self):
        return f"Point({self.x}, {self.y})"
```

```python
    # Define the __len__ method
    def __len__(self):
        return self.x + self.y

# Create instances of the Point class
p1 = Point(10, 20)
p2 = Point(2, 5)

# Use the overloaded + operator
print(p1 + p2)  # Output: Point(12, 25)
# Other's value:  Point(2, 5)
print(p2 + p1)  Output: Point(12, 25)
#Other's value:  Point(10, 20)
```

Other's value will be the object's value which Obj is assigned after the operator(here +).

```python
# Use the overloaded - operator
print(p1 - p2)  # Output: Point(8, 15)

# Use the overloaded * operator
print(p1 * p2)  # Output: Point(20, 100)

# Use the overloaded / operator
print(p1 / p2)  # Output: Point(5.0, 4.0)

# Use the overloaded // operator
print(p1 // p2)  # Output: Point(5, 4)

# Use the __len__ method
print(len(p1), end="\t\t")  # Output: 30
print(len(p2))  # Output: 7
```

Other dunder/magic methods in Python:

- `__str__() # used to set what gets displayed upon calling str(obj)`

The __str__ method is **called automatically** when you use the print() function or the str() function on an instance of the Point class. This method provides a human-readable string representation of the object.

- The example code's output without `__str__()` will be look like:

```
<__main__.Point object at 0x000001CB3AFCB980>
<__main__.Point object at 0x000001CB3AFCB980>
<__main__.Point object at 0x000001CB3AFCB980>
<__main__.Point object at 0x000001CB3AFCB980>
<__main__.Point object at 0x000001CB3AFCB980>
30          7
```

0x00000222BB1DB650 – It is memory address.

- `__len__() # used to set what gets displayed upon calling.__len__() or len(obj)`

The __len__() method in Python is used to define the behavior of the len() function for instances of a class.

- Common Uses of __len__():

1. Custom Collection Classes:

If you create a custom collection class (like a custom list, set, or dictionary), you can use __len__() to return the number of elements in the collection.

```python
class MyList:
    def __init__(self, items):
        self.items = items

    def __len__(self):
        return len(self.items)

my_list = MyList([1, 2, 3, 4])
print(len(my_list))  # Output: 4
```

2. Graphical Objects:

For graphical objects like shapes or polygons, __len__() can return the number of vertices or points.

```python
class Polygon:
    def __init__(self, vertices):
        self.vertices = vertices

    def __len__(self):
        return len(self.vertices)

polygon = Polygon([(0, 0), (1, 0), (1, 1), (0, 1)])
# [(0, 0), (1, 0), (1, 1), (0, 1)] is a list of tuples.
print(len(polygon))  # Output: 4
print(type(polygon)) # Output: <class '__main__.Polygon'>
```

3. String-Like Classes:

For classes that represent strings or similar data structures, __len__() can return the length of the string.

```python
class MyString:
    def __init__(self, text):
        self.text = text

    def __len__(self):
        return len(self.text)

my_string = MyString("Hello")
print(len(my_string))  # Output: 5
```

4. Data Structures:

For custom data structures like trees, linked lists, or graphs, __len__() can return the number of nodes or elements.

```python
# Define a class called Node
class Node:
    def __init__(self, value):
        self.value = value
        self.next = None

# Define a class called LinkedList
class LinkedList:
    def __init__(self):
        self.head = None
        self.size = 0

    def addFront(self, value):
        new_node = Node(value)
        new_node.next = self.head
        self.head = new_node
        self.size += 1

    def __len__(self):
        return self.size

# Create an instance of the LinkedList class
linked_list = LinkedList()
linked_list.addFront(1)
linked_list.addFront(2)

# Print the length of the linked list
print(len(linked_list))
# Output: 2
```

PROJECT 2 – THE PERFECT GUESS

We are going to write a program that generates a random number and asks the user to guess it.

If the player's guess is higher than the actual number, the program displays "Lower number please". Similarly, if the user's guess is too low, the program prints "higher number please" When the user guesses the correct number, the program displays the number of guesses the player used to arrive at the number.

Hint: Use the random module.

```python
import random

# Generate a random number between 1 and 9
randNo = random.randint(1, 9)
a = -1 # guessing the input num as -1 for the loop's condition.
guesses = 0 # to keep track of how many times user guessed.

# Loop until the guessed number is equal to the random number
while (a != randNo):
    guesses += 1  # Increment the guess count for each attempt
    a = int(input(f"\n{guesses}. Enter the number: "))
    if(a > randNo):
        print("-->> Lower Number Please")
    elif(a < randNo):
        print("-->> Higher Number Please")

# Print the result after guessing the number correctly
print(f"\n..::You have guessed the number {randNo} correctly in {guesses} attempts\n")
```

Part 12 – ADVANCED PYTHON 1

☑ NEWLY ADDED FEATURES IN PYTHON

Following are some of the newly added features in Python programming language.

☑ WALRUS OPERATOR

The walrus operator (:=), introduced in Python 3.8, allows you to assign values to variables as part of an expression. This operator, named for its resemblance to the eyes and tusks of a walrus, is officially called the "assignment expression."

```python
if (n := len([1, 2, 3, 1, 2, 3])) > 3:
    print(f"List is too long ({n} elements, expected <= 3)")

else:
    print("List is not too long, expected > 3")

# Output: List is too long (6 elements, expected <= 3)
```

In this example, n is assigned the value of len([1, 2, 3, 4, 5]) and then used in the comparison within the if statement.

Ex_1:

```python
if (n := 10) > 0:
    print(f"{n} is positive")
```

Ex_2:

```python
# Read numbers from a list and stop when a negative number is
found
numbers = [3, 5, 7, -1, 2, 3, 4, 10]
print("Processing number: ", end="")
while (n := numbers.pop(0)) > 0:
    print(n, end=" ") # Output: Processing number: 3 5 7
```

Ex_3:

```python
# Create a list of squares
that are greater than 10
numbers = [1, 2, 3, 4, 5]
squares = [square for num in
numbers if (square := num *
num) > 10]
print(squares)
```

Ex_4:

```python
# Get user input and check if it's an integer
while (user_input := input("Enter a number: ")).isdigit():
    print(f"You entered: {user_input}")

# This loop will continue prompting the user for input until you
enter something that is not a digit.
```

☑ TYPES DEFINITIONS IN PYTHON

Type hints are added using the colon (:) syntax for variables and the -- syntax for function return types.

```python
# Variable type hint
age: int = 25
# Function type hints
def greeting(name: str) -> str:
       return f"Hello, {name}!"
# Usage
print(greeting("Alice")) # Output: Hello, Alice!
```

☑ ADVANCED TYPE HINTS

Python's typing module provides more advanced type hints, such as List, Tuple, Dictionary, and Union. You can import List, Tuple and Dictionary types from the typing module like this:

```python
from typing import List, Tuple, Dict, Union
```

The syntax of types looks something like this:

```python
from typing import List, Tuple, Dict, Union
# List of integers
numbers: List[int] = [1, 2, 3, 4, 5]
# Tuple of a string and an integer
person: Tuple[str, int] = ("Alice", 30)
# Dictionary with string keys and integer values
scores: Dict[str, int] = {"Alice": 90, "Bob": 85}
# Union type for variables that can hold multiple types
identifier: Union[int, str] = "ID123"
identifier = 12345 # Also valid
```

These annotations help in making the code self-documenting and allow developers to understand the data structures used at a glance.

☑ MATCH CASE/SWITCH CASE

Python 3.10 introduced the match statement, which is similar to the switch statement found in other programming languages.

The basic syntax of the match statement involves matching a variable against several cases using the case keyword.

```
def http_status(status):
      match status:
            case 200:
                  return "OK"
            case 404:
                  return "Not Found"
            case 500:
                  return "Internal Server Error"
            case _:
                  return "Unknown status"
# Usage
print(http_status(200)) # Output: OK
print(http_status(404)) # Output: Not Found
print(http_status(500)) # Output: Internal Server Error
print(http_status(403)) # Output: Unknown status
```

☑ DICTIONARY MERGE & UPDATE OPERATORS

- ☠ New operators | and |= allow for merging and updating dictionaries.

```
dict1 = {'a': 1, 'b': 2}
dict2 = {'b': 3, 'c': 4}
merged = dict1 | dict2
print(merged) # Output: {'a': 1, 'b': 3, 'c': 4}
```

☑ EXCEPTION HANDLING IN PYTHON

There are many built-in exceptions which are raised in python when something goes wrong.

Exception in python can be handled using a try statement. The code that handles the exception is written in the except clause.

- ☠ When the exception is handled, the code flow continues without program interruption.
- ☠ We can also specify the exception to catch.

☑ RAISING EXCEPTIONS

We can raise custom exceptions using the ‘raise’ keyword in python.

Example:

```
raise ZeroDivisionError("Cannot divide by zero")
raise NegativeNumberError("Negative numbers are not allowed")
raise ValueError("Age cannot be negative")
```

Code:

```python
def validate_age(age):
    if age < 0:
        raise ValueError("Age cannot be negative")
    elif age < 18:
        raise ValueError("Age must be at least 18")
    return age

# Taking age input from the user
try:
    user_age = int(input("Please enter your age: "))
    valid_age = validate_age(user_age)
    print(f"Valid age: {valid_age}")
except ValueError as e:
    print(f"Validation Error: {e}")
```

☑ TRY WITH ELSE CLAUSE

- The else block runs if no exceptions were raised.

☑ TRY WITH FINALLY

The finally block always runs, regardless of whether an exception was raised or not, and prints a completion message.

Code: (all in one) —

```python
# Define a custom exception for large numerators
class LargeNumeratorError(Exception):
    pass
# This class can be useful for handling specific error conditions in a clear and organized way. # Essential for the custom LargeNumeratorError.

# Function to divide two numbers with exception handling
def divide_numbers():
    try:
        # Ask the user for input
        numerator = float(input("Enter the numerator: "))
        denominator = float(input("Enter the denominator: "))

        # Raise an exception if the numerator is too large
        if numerator > 1000:
            raise LargeNumeratorError("Numerator is too large")

        # Perform the division
        result = numerator / denominator

    except ValueError as e:
        # Handle the case where the input is not a number
        print(f"ValueError: {e}")

    except ZeroDivisionError as e:
        # Handle the case where the denominator is zero
        print(f"ZeroDivisionError: {e}")

    except LargeNumeratorError as e:
        # Handle the case where the numerator is too large
        print(f"LargeNumeratorError: {e}")
    except Exception as e:
        # Handle any other exceptions
        print(f"An unexpected error occurred: {e}")

    else:
        # This block runs if no exceptions were raised
        print(f"The result is: {result}")

    finally:
        # This block always runs, regardless of exceptions
        print("Execution completed.")

# Call the function
divide_numbers()
```

The else block runs if no exceptions were raised and prints the result of the division.

☑ IF __NAME__ == '__MAIN__' IN PYTHON

'__name__' evaluates to the name of the module in python from where the program is ran.

If the module is being run directly from the command line, the '__name__' is set to string "__main__". Thus, this behavior is used to check whether the module is run directly or imported to another file.

Simple Examples:

Step 1: (create a file with any name "XYmain.py")	Step 2: (Create the Second Script [anyName.py])

`def Myfunction():`
`    print("Hello, World!")`
 	
`Myfunction()`
`print(__name__)`
 	
`Output:`
 	
`Hello, World!`
`__main__`
	`from Xmain import Myfunction`
	
	`Output:`
	
	`Hello, World!`
	`XYmain`
	
	`# XYmain -- The module name Printed because of the print(__name__), which is executed from Imported module named "Xmain.py"`

Example:

Step 1: Create the First Script

Create a file named my_script.py with the following content:

```
# my_script.py

# Define a function
def my_function():
    print("Hello, World!")

# Call the function
my_function()
```

```
# Print the special variable __name__
print(__name__)

# Check if the script is being run directly
if __name__ == "__main__":
    print("This script is being run directly.")
else:
    print("This script is being imported as a module.")
```

If I run the program:

```
Hello, World!
__main__
This script is being run directly.
```

Step 2: Create the Second Script

Create another file named import_script.py with the following content:

```
import my_script
```

If I run the program:

```
Hello, World!
my_script
This script is being imported as a module.
```

☑ THE GLOBAL KEYWORD

- ‘global’ keyword is used to modify the variable outside of the current scope.

Ex:

```
a = 89

def fun():
    global a # a is declared as global (pointing to the global variable) #
Without here an Error will be raised.
    print("Before Changing the value:", a) # 89
    a = 14
    print("Inside function\t:", a) # 14

fun()
print("Outside function:", a) # 14
```

☑ ENUMERATE FUNCTION IN PYTHON

The enumerate function in Python is used to add a counter to an iterable and returns it as an enumerate object. This is useful when you need both the index and the value while looping through a list, tuple, or other iterable.

- index is the counter and fruits is the iterable.

The enumerate function returns an enumerate object that provides both the index and the value for each item in Example’s fruits.

```
# List of fruits
```

```
fruits = ['Apple', 'Banana', 'Mango', 'Peach']

# Using enumerate to get index and value
for index, fruit in enumerate(fruits):
    print(index, fruit)

print()

# List of fruits & animal
fruits = ['Apple', 'Banana', 'Bread', 'Rice']
animals = ['Parrot', 'Monkey', 'Mouse', 'Human']

# Using enumerate and zip to get index and values from both lists
for index, (fruit, animal) in enumerate(zip(fruits, animals),start=1):
    print(index, fruit, animal)
```

☑ DIFFERENCES BETWEEN ENUMERATE AND RANGE

Feature	**enumerate**	**range**
Purpose	Adds a counter to an iterable	Generates a sequence of numbers
Usage	Looping with index and value	Looping over a sequence of numbers
Syntax	enumerate(iterable, start=0)	range(start, stop, step)

☑ LIST COMPREHENSIONS

- List Comprehension is an elegant way to create lists based on existing lists.

Example: 1	Example: 2

`list1 = [1, 7, 12, 11, 22]`
`list2 = [item for item in list1 if item > 8]`
 	
`print(list2)`
	`myList = [1,2,3,4,5]`
	
	`squaredList = [value*value for value in myList]`
	`print(squaredList)`

Part 13 – ADVANCED PYTHON 2

☑ LAMBDA FUNCTIONS

- Function created using an expression using ‘lambda’ keyword.
- Can be used as a normal function

Syntax:
`lambda arguments:expressions`

```
# A lambda function that adds 10 to the input
add_ten = lambda x: x + 10
print(add_ten(5))  # Output: 15
# x act like a parameter
```

```python
# A lambda function that multiplies two numbers
multiply = lambda a, b: a * b
print(multiply(5, 6))  # Output: 30

# A lambda function that sums three numbers
sum_three = lambda a, b, c: a + b + c
print(sum_three(5, 6, 2))  # Output: 13
# a,b,e act like a parameter
```

☑ JOIN METHOD (STRINGS)

- Creates a string from iterable objects

```python
l = ["apple", "mango", "banana"]
result = ", and, ".join(l)
print(result)
```

The above line will return -- apple, and, mango, and, banana –

☑ FORMAT METHOD (STRINGS)

- Formats the values inside the string into a desired output.

```python
print("{} is a good {}".format("Sagar", "boy")) #1.
print("{} is a good {}".format("Roxy", "boy")) #2.
# output:
# Sagar is a good boy
# Roxy is a good boy
```

☑ MAP, FILTER & REDUCE

- Map applies a function to all the items in an input_list.
- The filter() function constructs an iterator from elements of an iterable for which a function returns true.
- Reduce applies a rolling computation to sequential pair of elements.
- The reduce() function applies a function of two arguments cumulatively to the items of an iterable, from left to right, to reduce the iterable to a single value. This function is available in the functools module.

Example_1:

Map:	Filter:	Reduce:

`# Function to double the value`
`def double(n):`
`    return n * 2`
 	
`numbers = [1, 2, 3, 4, 5]`
`doubled_numbers = map(double, numbers)`
	`# Function to check if a number is even`
	`def is_even(n):`
	`    return n % 2 == 0`
	
	`numbers = [1, 2, 3, 4, 5, 6]`
	`even_numbers = filter(is_even, numbers)`
		`from functools import reduce`
		
		`# Function to multiply two numbers`
		`def multiply(x, y):`
		`    return x * y`
		
		`numbers = [1, 2, 3, 4]`

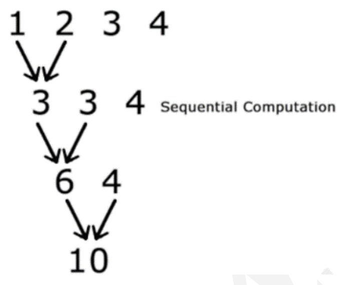
`print(list(doubled_numbers))`
`# Output: [2, 4, 6, 8, 10]`
	`print(list(even_numbers))  #`
	`Output: [2, 4, 6]`
		`product = reduce(multiply,`
		`numbers)`
		`print(product)  # Output: 24`

Ex_1: The Reduce function: [1,2,3,4] {left to right}.

1st loop: 1*2 = 2

2nd loop: 2*3 = 6

3rd loop: 6*4 = 24 (answer)



Example: (with lambda):

map() with lambda:	filter() with lambda:	reduce() with lambda:

`numbers = [1, 2, 3, 4, 5]`
`# Using lambda to double each`
`number`
`doubled_numbers = list(map(lambda`
`x: x * 2, numbers))`
`print(doubled_numbers)  # Output:`
`[2, 4, 6, 8, 10]`
	`numbers = [1, 2, 3, 4, 5, 6]`
	`# Using lambda to filter even`
	`numbers`
	`even_numbers =`
	`list(filter(lambda x: x % 2 ==`
	`0, numbers))`
	`print(even_numbers)  # Output:`
	`[2, 4, 6]`
		`from functools import reduce`
		
		`numbers = [1, 2, 3, 4]`
		`# Using lambda to multiply all`
		`numbers`
		`product = reduce(lambda x, y: x`
		`* y, numbers)`
		`print(product)  # Output: 24`

Part14 – VIRTUAL ENVIRIONMENT

An environment which is same as the system interpreter but is isolated from the other Python environments on the system.

INSTALLATION

To use virtual environments, we write:

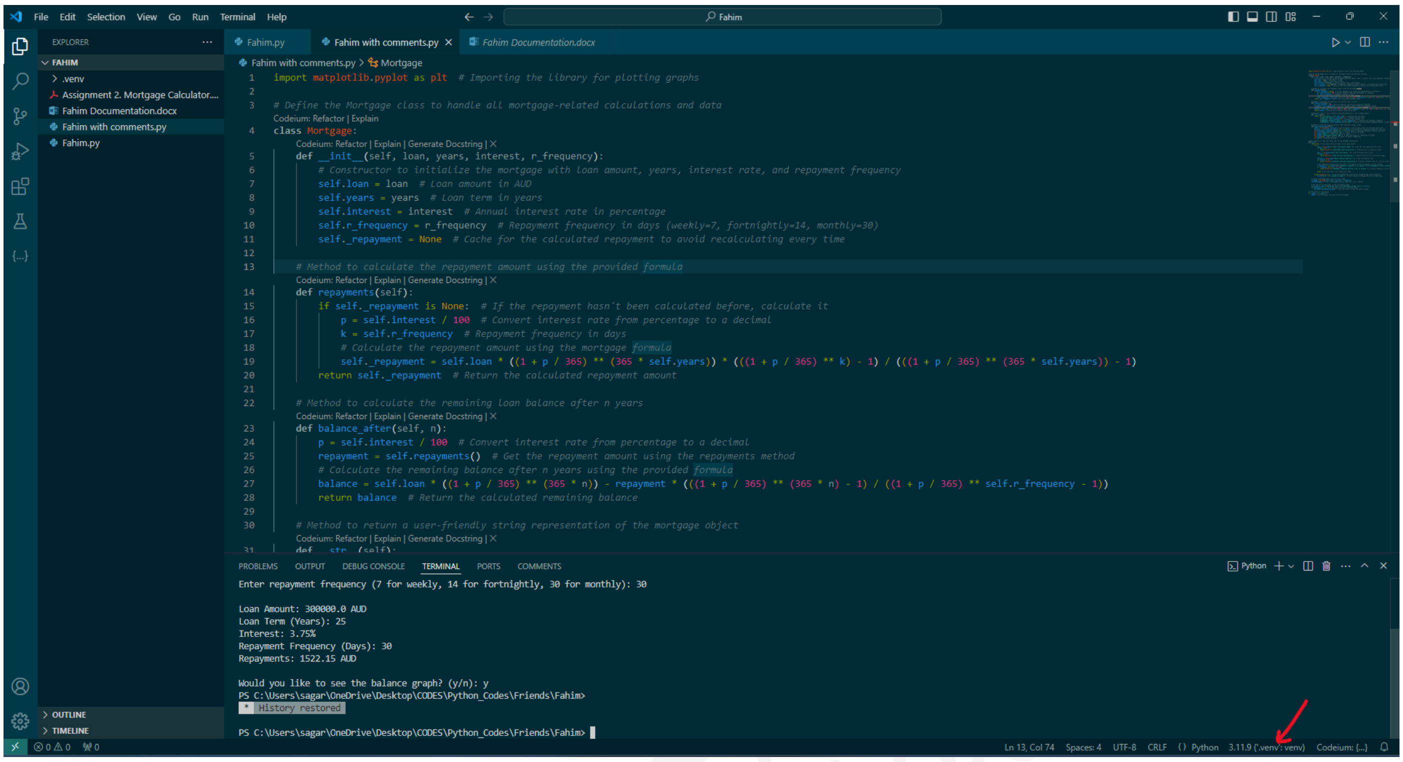
pip install virtualenv # Install the package

We create a new environment using:

virtualenv myprojectenv # Creates a new venv

Or,

P.T.O


Fahim.py
Fahim with comments.py
Fahim Documentation.docx
Fahim with comments.py > Mortgage
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS COMMENTS
Ln 13, Col 74 Spaces: 4 UTF-8 CRLF {} Python 3.11.9 ('.venv': venv)

```python
import matplotlib.pyplot as plt  # Importing the library for plotting graphs

# Define the Mortgage class to handle all mortgage-related calculations and data
class Mortgage:
    def __init__(self, loan, years, interest, r_frequency):
        # Constructor to initialize the mortgage with loan amount, years, interest rate, and repayment frequency
        self.loan = loan  # Loan amount in AUD
        self.years = years  # Loan term in years
        self.interest = interest  # Annual interest rate in percentage
        self.r_frequency = r_frequency  # Repayment frequency in days (weekly=7, fortnightly=14, monthly=30)
        self._repayment = None  # Cache for the calculated repayment to avoid recalculating every time

    # Method to calculate the repayment amount using the provided formula
    def repayments(self):
        if self._repayment is None:  # If the repayment hasn't been calculated before, calculate it
            p = self.interest / 100  # Convert interest rate from percentage to a decimal
            k = self.r_frequency  # Repayment frequency in days
            # Calculate the repayment amount using the mortgage formula
            self._repayment = self.loan * ((1 + p / 365) ** (365 * self.years)) * (((1 + p / 365) ** k) - 1) / (((1 + p / 365) ** (365 * self.years)) - 1)
        return self._repayment  # Return the calculated repayment amount

    # Method to calculate the remaining loan balance after n years
    def balance_after(self, n):
        p = self.interest / 100  # Convert interest rate from percentage to a decimal
        repayment = self.repayments()  # Get the repayment amount using the repayments method
        # Calculate the remaining balance after n years using the provided formula
        balance = self.loan * ((1 + p / 365) ** (365 * n)) - repayment * (((1 + p / 365) ** (365 * n) - 1) / ((1 + p / 365) ** self.r_frequency - 1))
        return balance  # Return the calculated remaining balance

    # Method to return a user-friendly string representation of the mortgage object
    def __str__(self):
```

```
Enter repayment frequency (7 for weekly, 14 for fortnightly, 30 for monthly): 30

Loan Amount: 300000.0 AUD
Loan Term (Years): 25
Interest: 3.75%
Repayment Frequency (Days): 30
Repayments: 1522.15 AUD

Would you like to see the balance graph? (y/n): y
PS C:\Users\sagar\OneDrive\Desktop\CODES\Python_Codes\Friends\Fahim>
 * History restored

PS C:\Users\sagar\OneDrive\Desktop\CODES\Python_Codes\Friends\Fahim>
```

PIP FREEZE COMMAND

'pip freeze' returns all the package installed in a given python environment along with the versions.

```
pip freeze > requirements .txt
```

The above command creates a file named 'requirements.txt' in the same directory containing the output of 'pip freeze'.

We can distribute this file to other users, and they can recreate the same environment using:

```
pip install –r requirements.txt
```

Tricks

Trick: 1 -- (round())

```python
#. print(round(123.123123 ,2), end="°C") # rounding the number to the 2 decimal.
```

Trick: 2 -- (Swapping Variables)

```python
a, b = 5, 10
a, b = b, a
print(a, b)  # Output: 10 5
```

Trick: 3 -- (Finding the Most Frequent Element in a List)

```python
from collections import Counter
data = [1, 2, 2, 3, 3, 3, 4, 4, 4, 4]
most_common = Counter(data).most_common(1)[0][0]
print(most_common)  # Output: 4
```

Trick: 4 -- (Reversing a String)

```
s = "Python"
reversed_s = s[::-1]
print(reversed_s)  # Output: nohtyP
```

Trick: 5 -- (Using zip to Iterate Over Two Lists)

```
names = ['Sagar', 'Polok', 'Pushu']
scores = [85, 90, 95]
for name, score in zip(names, scores):
    print(f"{name}: {score}")

# Output:
# Sagar: 85
# Polok: 90
# Pushu: 95
```

Trick: 6 -- (Using defaultdict for Counting)

```
from collections import defaultdict
# List of items
items = ['apple', 'banana', 'apple', 'orange', 'banana', 'apple']
# Using defaultdict to count occurrences
count = defaultdict(int)
for item in items:
    count[item] += 1
print(count)  # Output: defaultdict(<class 'int'>, {'apple': 3, 'banana': 2, 'orange': 1})
```

----------------------------- END -----------------------------