WHAT IS PYTHON?

Python is a high-level, general-purpose programming language known for its readability and versatility. It's widely used in various fields, including web development, data science, machine learning, and software development. Python is an interpreted language, meaning code is executed line by line. Its simple syntax, similar to English, makes it beginner-friendly and popular among experienced programmers.

Here's a more detailed breakdown:

Versatile and General-Purpose: Python isn't specialized for a particular task but can be used in many applications.

Readability and Beginner-Friendly: Its syntax is designed to be easy to understand, even for those new to programming.

Interpreted Language: Python code is executed directly, line by line, without needing to be compiled first.

Object-Oriented: Python supports object-oriented programming principles, allowing for modular and reusable code.

Large Standard Library: Python comes with a vast collection of pre-written code modules for various tasks.

Widely Used: It's a popular choice for web development, data analysis, machine learning, and software development.

Cross-Platform: Python runs on various operating systems like Windows, macOS, and Linux

WHAT IS MODULES AND PIP IN PYTHON?

In Python, modules are reusable blocks of code that extend the language's functionality. pip is the package manager for Python, used to install and manage these external modules.

Reusability: Modules are designed to be reused in different parts of a program or even in different programs.

Organization: They help organize code, making it easier to maintain and understand.

Extensibility: Modules provide access to a wide range of functionalities without requiring you to write the code yourself.

Examples: Standard Python modules like math, random, or datetime are pre-installed with the language. External modules can be installed using pip, such as numpy, pandas, or requests.

pip (Package Installer for Python):

- **Installation:** pip is the primary tool for installing Python packages, including external modules.
- **Dependency Management:** It helps manage dependencies between different packages.
- **Virtual Environments:** pip can be used with virtual environments to isolate project dependencies.
- **Command-line usage:** The basic command is pip install <module_name>

COMMENTS IN PYTHON

A **comment** is a line in your code that is **ignored by Python when your program runs**. It's used to **explain code** or leave **notes for yourself or others**.

1. Single-line Comment

Use # at the beginning of the line.

```
# This is a single-line comment

print("Hello, World!")  # This prints a message
```

2. Multi-line Comment

Python doesn't have a true multi-line comment, but you can use triple quotes (' ' ' or " " ") to simulate it.

```
'''
This is a
multi-line comment
'''
print("Welcome to Python")
```

WHAT IS AN ESCAPE SEQUENCE?

Escape sequences start with a **backslash** \, and are used to **represent special characters** in a string that can't be typed directly.

Escape Sequence	Meaning	Example
\n	New Line	"Hello\nWorld"
\t	Tab Space	"Name:\tAlice"
\\	Backslash	"This is a backslash: \\"
\'	Single Quote	'It\'s a sunny day'
\"	Double Quote	"He said, \"Hi!\""

VARIABLES AND DATA TYPES IN PYTHON

What is a Variable?

A **variable** is a **name** that stores a value in memory.
It acts like a **container** for data.

Rules for Naming Variables:

- Must start with a letter or underscore (_)
- Cannot start with a number
- Can only contain letters, numbers, and underscores
- **Case-sensitive** (name ≠ Name)

Python Data Types

Python has built-in data types to classify different kinds of data.

1. Basic Data Types in Python:

Data Type	Example	Description
int	10, -5, 1000	Whole numbers
float	3.14, -2.0	Decimal numbers
str	"Hello"	Text - (string of characters)
bool	True, False	Boolean values

2.Collection Data Types:

Data Type	Example	Description
list	[1, 2, 3]	Ordered, changeable, allows duplicates
tuple	(1, 2, 3)	Ordered, unchangeable, allows duplicates
set	{1, 2, 3}	Unordered, no duplicates
dict	{"name": "Alice", "age": 25}	Key-value pairs

EXERCISE:1

1. Add two numbers
2. Find the square and cube of a number
3. Calculate area of a rectangle
4. Convert temperature (Celsius ↔ Fahrenheit)
5. Simple interest calculator

TYPECASTING IN PYTHON

Typecasting in Python means converting one data type into another. It's useful when you want to perform operations between different data types or format data in a specific way.

There are two types of typecasting in Python:

1. IMPLICIT TYPECASTING

Python automatically converts one data type to another without your involvement.

```
a = 5       # int
b = 2.0     # float

result = a + b
print(result)        # Output: 7.0
print(type(result))  # Output: <class 'float'>
```

2. EXPLICIT TYPECASTING

You manually convert the data type using type functions like:

- `int()` – converts to integer
- `float()` – converts to float
- `str()` – converts to string
- `bool()` – converts to boolean

Examples:

```
# String to int
num_str = "10" #print("Hello" 6,9,0,sep="$",end="JJA")
num_int = int(num_str)
print(num_int)          # Output: 10
# Float to int
f = 7.9
print(int(f))           # Output: 7 (decimal part is removed)

# Int to string
x = 100
print(str(x))           # Output: '100'

# Int to float
```

```
y = 5
print(float(y))         # Output: 5.0
```

TAKING INPUT FROM USER

BASIC EXAMPLE:

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

USING TYPECASTING:

```
age = int(input("Enter your age: "))    # Converts input string
to integer
print("You are", age, "years old.")
```

EXERCISE 2:

1. Take your full name as input and print: "Nice to meet you, <full name>!"
2. Input three numbers and print their total sum.
3. Take two floating-point numbers and print their multiplication result.
4. Take your birth year and current year as input, then print your age.
5. Input the length of one side of a square and print its area and perimeter.
   (Area = side × side, Perimeter = 4 × side)

OPERATORS IN PYTHON

1. Arithmetic Operators (Used for mathematical operations)

Operator	Description	Example
+	Addition	5 + 3 = 8
-	Subtraction	5 - 2 = 3
*	Multiplication	4 * 2 = 8
/	Division	8 / 2 = 4.0
//	Floor Division	8 // 3 = 2
%	Modulus (remainder)	7 % 3 = 1
**	Exponent (power)	2 ** 3 = 8

2. Comparison Operators (Returns True or False)

Operator	Description	Example
==	Equal to	`5 == 5 → True`
!=	Not equal to	`5 != 3 → True`
>	Greater than	`6 > 2 → True`
<	Less than	`2 < 5 → True`
>=	Greater than or equal	`5 >= 5 → True`
<=	Less than or equal	`4 <= 6 → True`

3. Assignment Operators (Used to assign values to variables)

Operator	Description	Example
=	Assign	`x = 5`
+=	Add and assign	`x += 3 → x = x + 3`
-=	Subtract and assign	`x -= 2 → x = x - 2`
*=	Multiply and assign	`x *= 4`
/=	Divide and assign	`x /= 2`
//=	Floor divide and assign	`x //= 3`
%=	Modulus and assign	`x %= 2`
**=	Exponent and assign	`x **= 2`

4. Logical Operators (Used to combine conditional statements)

Operator	Description	Example
and	Returns `True` if both are true	`x > 5 and x < 10`

`or`	Returns `True` if at least one is true	`x < 3 or x > 7`
`not`	Reverses the result	`not(x > 5)`

WHAT ARE STRINGS?

In python, anything that you enclose between single or double quotation marks is considered a string. A string is essentially a sequence or array of textual data. Strings are used when working with Unicode characters.

Example:

```
name = "Ayush"
print("Hello, " + name)
```

Output: Hello, Ayush

```
print('He said, "I want to eat an apple".')
```

Multiline Strings:

If our string has multiple lines, we can create them like this:

```
a = """Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua."""
print(a)
```

STRING SLICING & OPERATIONS ON STRING

Length of a String:

We can find the length of a string using len() function.

Example:

```
fruit = "Mango"
len1 = len(fruit)
print("Mango is a", len1, "letter word.")
```

Output:

Mango is a 5 letter word.

String as an array

A string is essentially a sequence of characters also called an array. Thus we can access the elements of this array.

Example:

```
pie = "ApplePie"
```

```
print(pie[:5])
print(pie[6])    #returns character at specified index
```

Output:

Apple
i

Note: This method of specifying the start and end index to specify a part of a string is called slicing.

Slicing Example:

```
pie = "ApplePie"
print(pie[:5])       #Slicing from Start
print(pie[5:])       #Slicing till End
print(pie[2:6])      #Slicing in between
print(pie[-8:])      #Slicing using negative index
```

Output:

Apple
Pie
pleP
ApplePie

Output:

A
B
C
D
E

STRING METHODS

Python provides a set of built-in methods that we can use to alter and modify the strings.

1. **upper() :**

   The upper() method converts a string to upper case.

   Example:

   ```
   str1 = "AbcDEfghIJ"
   print(str1.upper())
   ```

   Output:

   ABCDEFGHIJ

2. **lower()**

   The lower() method converts a string to lower case.

   Example:

   ```
   str1 = "AbcDEfghIJ"
   print(str1.lower())
   ```

   Output:

   Abcdefghij

3. strip() :

The strip() method removes any white spaces before and after the string.

Example:

```
str2 = "     Ayush Spoon     "
print(str2.strip)
```

Output:

Ayush Spoon

4. rstrip() :

The rstrip() removes any trailing characters in right side of string.

Example:

```
str3 = "Hello !!!"
print(str3.rstrip("!"))
```

Output:

Hello

5. replace() :

The replace() method replaces all occurences of a string with another string. Example:

```
str2 = "Silver Spoon"
print(str2.replace("Sp", "M"))
```

Output:

Silver Moon

6. split() :

The split() method splits the given string at the specified instance and returns the separated strings as list items.

Example:

```
str2 = "Silver Spoon"
print(str2.split(" "))      #Splits the string at the
whitespace " ".
```

Output:

['Silver', 'Spoon']

7. capitalize() :

The capitalize() method turns only the first character of the string to uppercase and the rest other characters of the string are turned to lowercase. The string has no effect if the first character is already uppercase.

Example:

```
str1 = "hello"
capStr1 = str1.capitalize()
print(capStr1)
str2 = "hello WorlD"
capStr2 = str2.capitalize()
print(capStr2)
```

Output:

Hello
Hello world

8. center() :

The center() method aligns the string to the center as per the parameters given by the user.

Example:

```
str1 = "Welcome to the Console!!!"
print(str1.center(50))
```

Output:

Welcome to the Console!!!

We can also provide padding character. It will fill the rest of the fill characters provided by the user.

Example:

```
str1 = "Welcome to the Console!!!"
print(str1.center(50, "."))
```

Output:

.....Welcome to the Console!!!.....

9. count() :

The count() method returns the number of times the given value has occurred within the given string.

Example:

```
str2 = "Abracadabra"
countStr = str2.count("a")
print(countStr)
```

Output:

4

10. endswith() :

The endswith() method checks if the string ends with a given value. If yes then return True, else return False.

Example :

```
str1 = "Welcome to the Console !!!"
print(str1.endswith("!!!"))
```

Output:

True

We can even also check for a value in-between the string by providing start and end index positions.

Example:

```
str1 = "Welcome to the Console !!!"
print(str1.endswith("to", 4, 10))
```

Output:

True

11. find() :

The find() method searches for the first occurrence of the given value and returns the index where it is present. If given value is absent from the string then return -1.

Example:

```
str1 = "He's name is Dan. He is an honest man."
print(str1.find("is"))
```

Output:

10

As we can see, this method is somewhat similar to the index() method. The major difference being that index() raises an exception if value is absent whereas find() does not.

Example:

```
str1 = "He's name is Dan. He is an honest man."
print(str1.find("Daniel"))
```

Output:

-1

IF-ELSE STATEMENTS

if-else is used in Python to make decisions in your program.
It helps your program take different actions based on conditions (True/False).
Based on this, the conditional statements are further classified into following types:

- ❖ if
- ❖ if-else
- ❖ if-else-elif
- ❖ nested if-else-elif.

Structure:

```
if condition:
    # code to run if condition is True
else:
    # code to run if condition is False
```

Example1:

```
age = 21
age_normal = 18
if (age> age_normal):
    print("You can drive car.")
else:
    print("You can not drive car.")
```

Output:

You can not drive car.

Example2:

```
marks = 75

if marks >= 90:
    print("Grade: A")
elif marks >= 75:
    print("Grade: B")
elif marks >= 60:
    print("Grade: C")
else:
    print("Fail")
```

Output:

```
Grade: B
```

MATCH CASE STATEMENT

match-case is used to check **multiple conditions** (like switch-case in C, C++, Java).
It was introduced in **Python 3.10** and later versions

Important Points:

- Use `match-case` only in **Python 3.10 or newer**.
- `case _:` works like the **default** case.
- You can use `|` to check **multiple values** in a single case.
- Use `case _ if condition:` for **conditional matching**.

Example1:

```
fruit = "apple"
match fruit:
    case "apple":
        print("It's red or green.")
    case "banana":
        print("It's yellow.")
    case _:
        print("Unknown fruit.")
```

Example2:

```
value = 2
match value:
    case 1 | 2 | 3:
        print("Between 1 and 3")
    case 4:
        print("It's four")
    case _:
        print("Something else")
```

Example3:

```
number = 10
match number:
    case _ if number % 2 == 0:
        print("Even number")
    case _:
        print("Odd number")
```

INTRODUCTION TO LOOPS

Sometimes a programmer wants to execute a group of statements a certain number of times. This can be done using loops. Based on this loops are further classified into following main types;

- for loop
- while loop

The for Loop

for loops can iterate over a sequence of iterable objects in python. Iterating over a sequence is nothing but iterating over strings, lists, tuples, sets and dictionaries.

Example1: Iterating over a string:

```
name = 'Abhishek'
for i in name:
    print(i, end=", ")
```

Output:
A, b, h, i, s, h, e, k,

Notes: Similarly, we can use loops for lists, sets and dictionaries.

Example2: Iterating over a list:

```
colors = ["Red", "Green","Blue","Yellow"]
for x in colors:
    print(x)
```

Output:
Red
Green
Blue
Yellow

range():

What if we do not want to iterate over a sequence? What if we want to use for loop for a specific number of times?
Here, we can use the range() function.

Example:

```
for k in range(5):
    print(k)
```

Output:

Example3:

```
for k in range(4,9):
    print(k)
```

Output:

0 1 2 3 4	4 5 6 7 8

BREAK AND CONTINUE STATEMENTS IN PYTHON

Python provides two important statements to control the flow of loops:

1. break Statement

- The `break` statement **terminates the loop immediately**, even if the loop condition is still true.
- It is generally used when we want to **exit a loop early**, based on a condition.

Example:

```
for i in range(1, 10):
    if i == 5:
        break
    print(i)
```

Output:

```
1
2
3
4
```

When `i` becomes 5, the `break` statement is executed, and the loop stops.

2. continue Statement

- The `continue` statement **skips the current iteration** and moves to the **next iteration** of the loop.
- It is used when we want to **ignore some values or conditions** but still continue the loop.

Example:

```
for i in range(1, 6):
    if i == 3:
        continue
    print(i)
```

Output:

```
1
2
4
5
```

WHAT IS A FUNCTION IN PYTHON?

A **function** is a reusable block of code that performs a specific task. It helps make the code modular and reduces repetition.

Types of Functions

1. **Built-in Functions** – Already provided by Python (e.g. `print()`, `len()`, `range()`)
2. **User-defined Functions** – Created by the user to perform custom tasks

Defining a Function

```
def function_name(parameters):
    # code block
    return result
```

`def`: Keyword to define a function
`parameters`: Optional values the function can receive
`return`: Sends a value back to where the function was called

Example:

```
def greet(name):
    return "Hello, " + name
print(greet("Amit"))
# Output: Hello, Amit
```

Calling a Function

You "call" a function by using its name followed by parentheses:

```
function_name(arguments)
```

Parameters vs Arguments

- **Parameter**: Variable in the function definition
- **Argument**: Actual value passed during the function call

Function with Default Parameters

```
def greet(name="Guest"):
    return "Hello, " + name

print(greet())          # Output: Hello, Guest
print(greet("Riya"))    # Output: Hello, Riya
```

Return Statement

Used to send the output from a function:

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

Function Without Parameters

```
def show_message():
    print("Welcome to Python!")
show_message()
```

LIST IN PYTHON

Lists are ordered collection of data items.
They store multiple items in a single variable.
List items are separated by commas and enclosed within square brackets [].
Lists are changeable meaning we can alter them after creation.

Example 1:

```
lst1 = [1,2,2,3,5,4,6]
lst2 = ["Red", "Green", "Blue"]
print(lst1)
print(lst2)
```

Output:

[1, 2, 2, 3, 5, 4, 6]
['Red', 'Green', 'Blue']

Example 2:

```
details = ["Abhijeet", 18, "FYBScIT", 9.8]
print(details)
```

Output:

['Abhijeet', 18, 'FYBScIT', 9.8]

BASICS OF LIST INDEXING

Indexes start from 0 (not 1).
You use square brackets [] to access elements.

```
my_list = ['apple', 'banana', 'cherry']
print(my_list[0])  # Output: 'apple'
print(my_list[1])  # Output: 'banana'
print(my_list[2])  # Output: 'cherry'
```

Negative Indexing

Negative numbers count from the end of the list.

```
print(my_list[-1])  # 'cherry'
print(my_list[-2])  # 'banana'
```

```
print(my_list[-3])  # 'apple'
```

IndexError

Trying to access an index that doesn't exist will raise an error:

```
print(my_list[5])  # IndexError: list index out of range
```

Modifying List Elements Using Index

```
my_list[1] = 'mango'
print(my_list)  # ['apple', 'mango', 'cherry']
```

Using Index in Loops

```
for i in range(len(my_list)):
    print(f"Element at index {i} is {my_list[i]}")
```

List Slicing (a related concept)

```
print(my_list[0:2])   # ['apple', 'mango'] — from index 0 to 1
print(my_list[:])     # full list
print(my_list[::2])   # skip every other element
```

LIST METHODS

1.sort()

This method sorts the list in ascending order. The original list is updated

Example 1:

```
colors = ["voilet", "indigo", "blue", "green"]
colors.sort()
print(colors)
num = [4,2,5,3,6,1,2,1,2,8,9,7]
num.sort()
print(num)
```

Output:

['blue', 'green', 'indigo', 'voilet']
[1, 1, 2, 2, 2, 3, 4, 5, 6, 7, 8, 9]

2.What if you want to print the list in descending order?

We must give reverse=True as a parameter in the sort method.

Example:

```
colors = ["voilet", "indigo", "blue", "green"]
colors.sort(reverse=True)
print(colors)
num = [4,2,5,3,6,1,2,1,2,8,9,7]
num.sort(reverse=True)
print(num)
```

Output:

['voilet', 'indigo', 'green', 'blue']
[9, 8, 7, 6, 5, 4, 3, 2, 2, 2, 1, 1]

Note : The reverse parameter is set to False by default.

3.reverse()

Example:

```
colors = ["voilet", "indigo", "blue", "green"]
colors.reverse()
print(colors)
num = [4,2,5,3,6,1,2,1,2,8,9,7]
num.reverse()
print(num)
```

Output:

```
['green', 'blue', 'indigo', 'voilet']
[7, 9, 8, 2, 1, 2, 1, 6, 3, 5, 2, 4]
```

4.index()

This method returns the index of the first occurrence of the list item.

Example:

```
colors = ["voilet", "green", "indigo", "blue", "green"]
print(colors.index("green"))

num = [4,2,5,3,6,1,2,1,3,2,8,9,7]
print(num.index(3))
```

Output:

1
3

5.count()

Returns the count of the number of items with the given value.

Example:

```
colors = ["voilet", "green", "indigo", "blue", "green"]
print(colors.count("green"))
```

Output:

2

6.copy()

Returns copy of the list. This can be done to perform operations on the list without modifying the original list.

Example:

```
colors = ["voilet", "green", "indigo", "blue"]
newlist = colors.copy()
print(colors)
print(newlist)
```

Output:

['voilet', 'green', 'indigo', 'blue']
['voilet', 'green', 'indigo', 'blue']

7.append():
This method appends items to the end of the existing list.

Example:
```
colors = ["voilet", "indigo", "blue"]
colors.append("green")
print(colors)
```

Output:
['voilet', 'indigo', 'blue', 'green']

8.insert():
This method inserts an item at the given index. User has to specify index and the item to be inserted within the insert() method.

Example:
```
colors = ["voilet", "indigo", "blue"]
#          [0]        [1]       [2]

colors.insert(1, "green")   #inserts item at index 1
# updated list: colors = ["voilet", "green", "indigo", "blue"]
#       indexs              [0]       [1]     [2]       [3]

print(colors)
```

Output:
['voilet', 'green', 'indigo', 'blue']

9.extend():
This method adds an entire list or any other collection datatype (set, tuple, dictionary) to the existing list.

Example:
```
#add a list to a list
colors = ["voilet", "indigo", "blue"]
rainbow = ["green", "yellow", "orange", "red"]
colors.extend(rainbow)
print(colors)
```
Output:
['voilet', 'indigo', 'blue', 'green', 'yellow', 'orange', 'red']

10.Concatenating two lists:
You can simply concatenate two lists to join two lists.

Example:
```
colors = ["voilet", "indigo", "blue", "green"]
colors2 = ["yellow", "orange", "red"]
```

```
print(colors + colors2)
```

Output:
['voilet', 'indigo', 'blue', 'green', 'yellow', 'orange', 'red']

TUPLES IN PYTHON

Tuples are ordered collection of data items. They store multiple items in a single variable. Tuple items are separated by commas and enclosed within round brackets (). Tuples are unchangeable meaning we can not alter them after creation.

Example:
```
tuple1 = (1,2,2,3,5,4,6)
tuple2 = ("Red", "Green", "Blue")
print(tuple1)
print(tuple2)
```

Output:
(1, 2, 2, 3, 5, 4, 6)
('Red', 'Green', 'Blue')

Tuple Indexes

Each item/element in a tuple has its own unique index. This index can be used to access any particular item from the tuple. The first item has index [0], second item has index [1], third item has index [2] and so on.

Example:
```
country = ("Spain", "Italy", "India",)
#            [0]     [1]      [2]
```

❖ **Accessing tuple items:**

I. Positive Indexing:

As we have seen that tuple items have index, as such we can access items using these indexes.

Example:
```
country = ("Spain", "Italy", "India",)
#            [0]      [1]      [2]
print(country[0])
print(country[1])
```

Output:
Spain
Italy

II. Negative Indexing:

Example:
```
country = ("Spain", "Italy", "India", "England", "Germany")
#            [0]      [1]      [2]       [3]        [4]
```

```
print(country[-1]) # Similar to print(country[len(country) - 1])
print(country[-3])
```

Output:

Germany
India

III. Check for item:

We can check if a given item is present in the tuple. This is done using the in keyword.

Example 1:

```
country = ("Spain", "Italy", "India", "England", "Germany")
if "Germany" in country:
    print("Germany is present.")
else:
    print("Germany is absent.")
```

Output:

Germany is present.

IV. Range of Index:

You can print a range of tuple items by specifying where do you want to start, where do you want to end and if you want to skip elements in between the range.

Syntax:

```
Tuple[start : end : jumpIndex]
```

Note: jump Index is optional. We will see this in given examples.

Example 1: Printing elements within a particular range:

```
animals = ("cat", "dog", "bat", "mouse", "pig", "horse", "donkey", "goat", "cow")
print(animals[3:7])     #using positive indexes
print(animals[-7:-2])   #using negative indexes
```

Output:

('mouse', 'pig', 'horse', 'donkey')
('bat', 'mouse', 'pig', 'horse', 'donkey')

Example 2: Printing all element from a given index till the end

```
animals = ("cat", "dog", "bat", "mouse", "pig", "horse", "donkey", "goat", "cow")
print(animals[4:])      #using positive indexes
print(animals[-4:])     #using negative indexes
```

Output:

('pig', 'horse', 'donkey', 'goat', 'cow')
('horse', 'donkey', 'goat', 'cow')
When no end index is provided, the interpreter prints all the values till the end.

Example 3: printing all elements from start to a given index

```
animals = ("cat", "dog", "bat", "mouse", "pig", "horse",
"donkey", "goat", "cow")
print(animals[:6])      #using positive indexes
print(animals[:-3])     #using negative indexes
```

Output:

('cat', 'dog', 'bat', 'mouse', 'pig', 'horse')
('cat', 'dog', 'bat', 'mouse', 'pig', 'horse')
When no start index is provided, the interpreter prints all the values from start up to the end index provided.

Example 4: Print alternate values

```
animals = ("cat", "dog", "bat", "mouse", "pig", "horse",
"donkey", "goat", "cow")
print(animals[::2])     #using positive indexes
print(animals[-8:-1:2]) #using negative indexes
```

Output:

('cat', 'bat', 'pig', 'donkey', 'cow')
('dog', 'mouse', 'horse', 'goat')

❖ Manipulating Tuples

Tuples are immutable, hence if you want to add, remove or change tuple items, then first you must convert the tuple to a list. Then perform operation on that list and convert it back to tuple.

Example:

```
countries = ("Spain", "Italy", "India", "England", "Germany")
temp = list(countries)
temp.append("Russia")       #add item
temp.pop(3)                 #remove item
temp[2] = "Finland"         #change item
countries = tuple(temp)
print(countries)
```

Output:

('Spain', 'Italy', 'Finland', 'Germany', 'Russia')

❖ Tuple methods

As tuple is immutable type of collection of elements it have limited built in methods.They are explained below

➢ count() Method

The count() method of Tuple returns the number of times the given element appears in the tuple.

Syntax:

```
tuple.count(element)
```

Example

```
Tuple1 = (0, 1, 2, 3, 2, 3, 1, 3, 2)
res = Tuple1.count(3)
print('Count of 3 in Tuple1 is:', res)
```

Output
3

➢ index() method

The Index() method returns the first occurrence of the given element from the tuple.

Syntax:

```
tuple.index(element, start, end)
```

Note: This method raises a ValueError if the element is not found in the tuple.

Example:

```
Tuple = (0, 1, 2, 3, 2, 3, 1, 3, 2)
res = Tuple.index(3)
print('First occurrence of 3 is', res)
```

Output
3

WHAT IS A SET IN PYTHON?

A **set** is an unordered, unindexed collection of **unique elements**.
It is **mutable** (can be changed) but cannot contain **mutable items** (like lists or dictionaries).
Defined using curly braces `{}` or the `set()` function.

```
my_set = {1, 2, 3, 4}
another_set = set([3, 4, 5])
```

KEY FEATURES OF SETS:

Feature	Description
Unordered	No indexing, elements have no order
No Duplicates	All elements are unique
Mutable	You can add or remove items

CREATING A SET:

```
# Empty set (use set(), not {})
empty_set = set()
# Set with values
fruits = {'apple', 'banana', 'mango'}
```

COMMONLY USED SET METHODS:

Method	Description	Example
`add()`	Adds an element to the set	`my_set.add(10)`
`remove()`	Removes a specific element (error if not found)	`my_set.remove(2)`

Method	Description	Example
`discard()`	Removes element (no error if not found)	`my_set.discard(5)`
`pop()`	Removes and returns a random item	`item = my_set.pop()`
`clear()`	Empties the entire set	`my_set.clear()`
`copy()`	Returns a shallow copy	`new_set = my_set.copy()`
`update()`	Adds multiple elements from another set/list	`my_set.update([7, 8])`

SET OPERATIONS:

Operation	Symbol / Method	Description
Union	`` `	`orset1.union(set2)` `
Intersection	`& or set1.intersection(set2)`	Common elements in both sets
Difference	`- or set1.difference(set2)`	Elements in `set1` but not in `set2`
Symmetric Difference	`^ or set1.symmetric_difference(set2)`	Elements in either set, but not both

EXAMPLES:

```
a = {1, 2, 3}
b = {2, 3, 4}

print(a | b)   # Union: {1, 2, 3, 4}
print(a & b)   # Intersection: {2, 3}
print(a - b)   # Difference: {1}
print(a ^ b)   # Symmetric Difference: {1, 4}
```

SET MEMBERSHIP

```
if 2 in my_set:
    print("2 is present")

if 9 not in my_set:
    print("9 is not present")
```

WHAT IS A DICTIONARY IN PYTHON?

A **dictionary** is an **unordered collection** of **key-value pairs**.
Each key is **unique** and maps to a value.
Defined using curly braces `{}` with keys and values separated by a colon `:`

```
student = {'name': 'John', 'age': 21, 'course': 'Python'}
```

KEY FEATURES OF DICTIONARIES:

Feature	Description
Key-Value	Each element is a key-value pair
Unordered	Order is preserved from Python 3.7+
Mutable	You can change, add, or remove items
No Duplicates	Keys must be unique

CREATING A DICTIONARY:

```
# Basic dictionary
person = {'name': 'Alice', 'age': 25}

# Empty dictionary
empty_dict = {}

# Using dict() function
data = dict(name='Bob', age=30)
```

ACCESSING AND MODIFYING ITEMS:

```
# Access value by key
print(person['name'])          # Output: Alice

# Add or update a value
person['age'] = 26             # Updates age
person['city'] = 'London'      # Adds new key-value pair

# Using get() (returns None if key doesn't exist)
print(person.get('gender'))    # Output: None
```

REMOVING ITEMS:

Method	Description	Example
`pop(key)`	Removes item by key	`person.pop('age')`
`popitem()`	Removes the last inserted item	`person.popitem()`
`del`	Deletes key or entire dictionary	`del person['name']`
`clear()`	Removes all items	`person.clear()`

LOOPING THROUGH A DICTIONARY:

```
# Loop through keys
for key in person:
    print(key)

# Loop through values
for value in person.values():
    print(value)

# Loop through key-value pairs
for key, value in person.items():
```

```
    print(key, value)
```

USEFUL DICTIONARY METHODS:

Method	Description
`keys()`	Returns all keys
`values()`	Returns all values
`items()`	Returns all key-value pairs as tuples
`get(key)`	Returns value for key, or None if not found
`update(dict2)`	Updates dictionary with another dictionary
`copy()`	Returns a shallow copy
`fromkeys(keys, val)`	Creates new dict with keys and a default value
`setdefault()`	Returns value of key, sets default if not found

EXAMPLE: FULL DICTIONARY WORKFLOW:

```
student = {
    'name': 'Rahul',
    'roll': 101,
    'subject': 'Math'
}

# Add new key
student['marks'] = 95

# Update existing value
student['subject'] = 'Science'

# Access value
print(student.get('name'))

# Loop through keys and values
for k, v in student.items():
    print(f"{k}: {v}")
```


HOW IMPORTING IN PYTHON WORKS

Importing in Python is the process of loading code from a Python module into the current script. This allows you to use the functions and variables defined in the module in your current script, as well as any additional modules that the imported module may depend on.

Once a module is imported, you can use any of the functions and variables defined in the module by using the dot notation. For example, to use the sqrt function from the math module, you would write:

```
import math
result = math.sqrt(9)
print(result)  # Output: 3.0
```

from keyword:

You can also import specific functions or variables from a module using the from keyword. For example, to import only the sqrt function from the math module, you would write:

```
from math import sqrt

result = sqrt(9)
print(result)  # Output: 3.0
```

You can also import multiple functions or variables at once by separating them with a comma:

```
from math import sqrt, pi

result = sqrt(9)
print(result)  # Output: 3.0

print(pi)  # Output: 3.141592653589793
```

importing everything:

It's also possible to import all functions and variables from a module using the * wildcard. However, this is generally not recommended as it can lead to confusion and make it harder to understand where specific functions and variables are coming from.

```
from math import *

result = sqrt(9)
print(result)  # Output: 3.0

print(pi)  # Output: 3.141592653589793
```

Python also allows you to rename imported modules using the as keyword. This can be useful if you want to use a shorter or more descriptive name for a module, or if you want to avoid naming conflicts with other modules or variables in your code.

The "as" keyword:

```
import math as m

result = m.sqrt(9)
print(result)  # Output: 3.0

print(m.pi)  # Output: 3.141592653589793
```

The dir function:

Finally, Python has a built-in function called dir that you can use to view the names of all the functions and variables defined in a module. This can be helpful for exploring and understanding the contents of a new module.

```
import math

print(dir(math))
```

This will output a list of all the names defined in the math module, including functions like sqrt and pi, as well as other variables and constants.

PYTHON FILE HANDLING MODES

Basic Syntax to Open a File

```
file = open("filename.txt", "mode"
```

File Modes in Python

Mode	Name	Description
'r'	**Read**	Opens the file for reading. File **must exist**.
'w'	**Write**	Opens the file for writing. **Overwrites** the file if it exists, or creates a new file.
'a'	**Append**	Opens the file for appending. Adds content to the end without deleting existing data.
'x'	**Exclusive Creation**	Creates a new file. Fails if the file already exists.
't'	**Text Mode**	Default mode. File is handled as text.
'b'	**Binary Mode**	File is handled as binary (images, videos, etc.).

Combined Modes

Mode	Description
'rt'	Read text (default)
'rb'	Read binary
'wt'	Write text (overwrites)
'wb'	Write binary (overwrites)
'at'	Append text
'ab'	Append binary
'r+'	Read and write (file must exist)
'w+'	Write and read (file is overwritten)
'a+'	Append and read

Examples

1. Reading a File

```
f = open("data.txt", "r")
content= f.read())
print(content)
f.close()
```

2. Writing to a File

```
f = open("data.txt", "w")
f.write("Hello, world!")
f.close()
```

3. Appending to a File

```
f = open("data.txt", "a")
f.write("\nNew line added.")
f.close()
```

4. Read + Write

```
f = open("data.txt", "r+")
data = f.read()
print(data)
f.write("\nExtra line.")
f.close()
```

Tips

Always use `close()` to close files or use:

```
with open("file.txt", "r") as f:
    data = f.read()
```

with automatically closes the file.

SOME MORE METHODS IN FILE HANDLING

readlines() method:

The readline() method reads a single line from the file. If we want to read multiple lines, we can use a loop.

```
f = open('myfile.txt', 'r')
while True:
    line = f.readline()
    if not line:
        break
    print(line)
```

The readlines() method reads all the lines of the file and returns them as a list of strings.

writelines() method:

The writelines() method in Python writes a sequence of strings to a file. The sequence can be any iterable object, such as a list or a tuple.
Here's an example of how to use the writelines() method:

```
f = open('myfile.txt', 'w')
lines = ['line 1\n', 'line 2\n', 'line 3\n']
f.writelines(lines)
f.close()
```

This will write the strings in the lines list to the file myfile.txt. The \n characters are used to add newline characters to the end of each string.
Keep in mind that the writelines() method does not add newline characters between the strings in the sequence. If you want to add newlines between the strings, you can use a loop to write each string separately:

```
f = open('myfile.txt', 'w')
lines = ['line 1', 'line 2', 'line 3']
for line in lines:
    f.write(line + '\n')
f.close()
```

seek() function:

The seek() function allows you to move the current position within a file to a specific point. The position is specified in bytes, and you can move either forward or backward from the current position. For example:

```
with open('file.txt', 'r') as f:
  # Move to the 10th byte in the file
  f.seek(10)

  # Read the next 5 bytes
  data = f.read(5)
```

tell() function:

The tell() function returns the current position within the file, in bytes. This can be useful for keeping track of your location within the file or for seeking to a specific position relative to the current position. For example:

```
with open('file.txt', 'r') as f:
  # Read the first 10 bytes
  data = f.read(10)

  # Save the current position
  current_position = f.tell()

  # Seek to the saved position
  f.seek(current_position)
```

truncate() function:

When you open a file in Python using the open function, you can specify the mode in which you want to open the file. If you specify the mode as 'w' or 'a', the file is opened in write mode and you can write to the file. However, if you want to truncate the file to a specific size, you can use the truncate function.
Here is an example of how to use the truncate function:

```
with open('sample.txt', 'w') as f:
  f.write('Hello World!')
  f.truncate(5)

with open('sample.txt', 'r') as f:
  print(f.read())
```

LAMBDA FUNCTION IN PYTHON

A **lambda function** is an **anonymous (nameless) function** defined using the `lambda` keyword.
It is used for creating **small, one-line functions** without a name.
Typically used when a short function is needed for a short period of time.

Example 1: Add two numbers

```
add = lambda a, b: a + b
print(add(5, 3))  # Output: 8
```

Example 2: Square of a number

```
square = lambda x: x * x
print(square(4))  # Output: 16
```

MAP FILTER REDUCE IN PYTHON

1. **`MAP()`** – APPLY A FUNCTION TO EACH ITEM

```
def square(x):
    return x * x

numbers = [1, 2, 3, 4]
squared = list(map(square, numbers))
print(squared)  # Output: [1, 4, 9, 16]
```

2. **`FILTER()`** – KEEP ITEMS THAT MATCH A CONDITION

```
def is_even(x):
    return x % 2 == 0

numbers = [1, 2, 3, 4, 5]
evens = list(filter(is_even, numbers))
print(evens)  # Output: [2, 4]
```

3. **REDUCE ( )** – REDUCE TO A SINGLE VALUE

```
from functools import reduce

def multiply(x, y):
    return x * y

numbers = [1, 2, 3, 4]
product = reduce(multiply, numbers)
print(product)  # Output: 24
```

Summary Table (Using Named Functions):

Function	Custom Function Example	What It Does	Output Example
`map()`	`square(x)`	Transforms each item	`[1, 4, 9, 16]`
`filter()`	`is_even(x)`	Filters based on condition	`[2, 4]`
`reduce()`	`multiply(x, y)`	Combines all items into one value	`24`

INTRODUCTION TO OBJECT-ORIENTED PROGRAMMING

Object-Oriented Programming (OOP) is a method of writing programs using **objects** and **classes**. It helps make code more organized, reusable, and easier to understand.

Why use OOP?

- Organize code into **real-world entities**
- Make code **modular and reusable**
- Improve **data security and structure**
- Reduce code duplication and improve maintainability

Key Concepts in Python OOP

1. **Class**: A blueprint to create objects. It defines properties (variables) and methods (functions).
2. **Object**: An instance of a class. It represents a real-world entity.
3. **Encapsulation**: Hiding internal details and only exposing what is necessary.
4. **Abstraction**: Showing only essential features and hiding complex details.
5. **Inheritance**: One class can use properties and methods of another class.
6. **Polymorphism**: Same function name behaves differently for different classes.

Simple Example:

```
class Car:
    name = "Neeraj"
    game = "Card Game"

    def start(self):
```

```
        print("Car started")

my_car = Car()
print(my_car.name)
my_car.start()
```

In the example above:

- `Car` is a **class**
- `my_car` is an **object**
- `start()` is a **method**

WHAT IS A CONSTRUCTOR

- It is a function that runs **automatically** when an object is created.
- In Python, the constructor method is named `__init__()`.
- It is used to assign **initial values** to object properties (like `name`, `age`, etc.).

Syntax of Constructor in Python

```
class ClassName:
    def __init__(self, parameters):
        # code to initialize the object
```

Example of Constructor

```
class Student:
    def __init__(self, name, marks):   # constructor
        self.name = name
        self.marks = marks

    def show(self):
        print("Name:", self.name)
        print("Marks:", self.marks)

s1 = Student("Neeraj", 90)   # object created, constructor
runs
s1.show()
```

Output:

Name: Neeraj
Marks: 90

Types of Constructors in Python:

Type	Description
Default Constructor	Constructor with no parameters
Parameterized Constructor	Constructor with parameters to initialize values

DEFAULT CONSTRUCTOR EXAMPLE:

```
class Demo:
```

```
    def __init__(self):      # No parameters
        print("Object Created")

obj = Demo()
```

PARAMETERIZED CONSTRUCTOR EXAMPLE:

```
class Demo:
    def __init__(self, message):   # With parameter
        print("Message:", message)

obj = Demo("Hello Python")
```

Summary:

- `__init__()` is a **constructor**
- It runs **automatically** when an object is created
- Used to **initialize** object variables

INHERITANCE IN PYTHON:

Code **reusability**: You don't have to write the same code again.
It helps in **organizing** code in a hierarchical manner.
You can **extend** or **modify** behavior of the parent class in the child class.

Basic Syntax of Inheritance:

```
class Parent:
    def display(self):
        print("This is Parent class.")

class Child(Parent):  # Inheriting Parent class
    def show(self):
        print("This is Child class.")

obj = Child()
obj.display()  # Accessing Parent class method
obj.show()     # Accessing Child class method
```

Types of Inheritance in Python:

Type	Description	Example
Single	One child class inherits from one parent class	`class B(A)`
Multiple	One child inherits from more than one parent class	`class C(A, B)`
Multilevel	One class inherits from a child class which in turn inherits from another	`class C(B), class B(A)`
Hierarchical	Multiple child classes inherit from the same parent class	`class B(A), class C(A)`
Hybrid	Combination of more than one type of inheritance	Combination of above

Access Modifiers in Python:

In Python, **access modifiers** are used to **control the visibility (accessibility)** of class members (variables and methods). Unlike some other languages like Java or C++, Python doesn't have strict access control, but it provides **naming conventions** to simulate them.

`Super()` Keyword in Python:

- `super()` is a built-in function used to call the **parent class's methods or constructor**.
- It is mostly used in **inheritance** to reuse code from the parent class.

Why use `super()`

- To reuse parent class code (constructor or methods)
- To maintain clean and DRY (Don't Repeat Yourself) code
- Helps in multiple inheritance by following Method Resolution Order (MRO)

Example 1: Using `super()` to Call Parent Constructor

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

class Student(Person):
    def __init__(self, name, age, student_id):
        super().__init__(name, age)  # calls Person's
constructor
        self.student_id = student_id
```

Example 2: Calling Parent Method Using `super()`

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def show_info(self):
        print(f"Name: {self.name}, Age: {self.age}")

class Student(Person):
    def __init__(self, name, age, student_id):
        super().__init__(name, age)
        self.student_id = student_id

    def show_info(self):
        super().show_info()  # call parent method
        print(f"Student ID: {self.student_id}")
```

Key Points:

- super() automatically refers to the immediate parent class.
- You can use super() to call any method, not just __init__.
- Works well with method overriding — lets you extend parent functionality.

SINGLE LEVEL INHERITANCE IN PYTHON

Single Level Inheritance means that **a child class inherits from a single parent class**. Child class gets access to all the **methods and properties** of the parent class.

EXAMPLE 1: WITH CONSTRUCTOR AND `SUPER()`

```
class Person:
  def __init__(self, name):
    self.name = name
    print(f"Person created: {self.name}")

class Student(Person):
  def __init__(self, name, roll):
    super().__init__(name) # Call parent constructor
    self.roll = roll
    print(f"Student created: Roll No {self.roll}")

s = Student("Neeraj", 101)
```

Output:

```
Person created: Neeraj
Student created: Roll No 101
```

KEY POINTS:

- In single level inheritance:
- One **parent** → One **child**
- Child can **override** parent methods.
- Use `super()` to call parent constructor or methods.
- Helps in **code reusability**.

Diagram (Text Form):

```
Parent
  ↑
Child
```

USE CASES:

- Reuse common functionality (e.g., login system, user info).
- Add specific behaviour in child class (e.g., student, employee).

WHAT IS MULTIPLE INHERITANCE?

Multiple inheritance means a class can **inherit from more than one parent class**.

EXAMPLE:

```
class Father:
    def skills(self):
        print("Father: Cooking")

class Mother:
    def skills(self):
        print("Mother: Painting")
```

```
class Child(Father, Mother):
    def skills(self):
        print("Child: ", end="")
        super().skills()
# Will call Father's skills() because Father is listed first

obj = Child()
obj.skills()
```

OUTPUT:

```
Child: Cooking
```

IMPORTANT NOTES:

- Python **resolves conflicts using MRO** (Method Resolution Order).
- The method of the first parent (from left to right) is called if there is a name conflict.
- To see MRO of a class, use:

```
print(Child.__mro__)
```

Using All Parents' Methods:
If you want to call methods from **all parent classes**, do it explicitly:

```
class Child(Father, Mother):
    def skills(self):
        Father.skills(self)
        Mother.skills(self)
```

ADVANTAGES:

- Allows **code reuse** from multiple sources.
- Good for combining features from multiple classes.

DISADVANTAGES:

- Can become **confusing** when multiple parents have the **same method name** (conflict).
- Can lead to **complex dependency structure**.

WHAT IS MULTILEVEL INHERITANCE:

Multilevel inheritance means that a class **inherits from a child class**, which in turn **inherits from another parent class**.

It creates a **chain of inheritance** like:
Grandparent → Parent → Child

EXAMPLE:

```
class Grandfather:
    def show_grandfather(self):
        print("I am the Grandfather.")

class Father(Grandfather):
    def show_father(self):
        print("I am the Father.")
```

```
class Son(Father):
    def show_son(self):
        print("I am the Son.")

# Create object of the last class in the chain
obj = Son()

# Access all methods
obj.show_grandfather()
obj.show_father()
obj.show_son()
```

OUTPUT:

```
I am the Grandfather.
I am the Father.
I am the Son.
```

KEY POINTS:

- The child class gets access to **all features** of parent and grandparent classes.
- It promotes **code reuse** across multiple levels.
- Python allows any number of levels in the inheritance chain.

ADVANTAGES:

- Helps in building a clear hierarchy.
- Promotes **step-by-step inheritance** of features.

DISADVANTAGES:

- If the chain becomes too long, it can become **complex to manage**.
- Changes in the top-level class may affect all subclasses.

WHAT IS HYBRID INHERITANCE

Hybrid inheritance is a combination of **two or more types of inheritance** (like single, multiple, multilevel, hierarchical) in a single program.

It creates a **complex structure** where a class can inherit from multiple classes in different ways.

Example Structure:

Imagine this:

```
    A
   / \
  B   C
   \ /
    D
```

Here:

- B and C inherit from A (Hierarchical)
- D inherits from both B and C (Multiple)

This combination is known as **Hybrid Inheritance**.

CODE EXAMPLE:

```
class A:
    def feature_a(self):
        print("Feature A")

class B(A):
    def feature_b(self):
        print("Feature B")

class C(A):
    def feature_c(self):
        print("Feature C")

class D(B, C):  # Inherits from both B and C
    def feature_d(self):
        print("Feature D")

# Create object of class D
obj = D()

# Accessing features from all parent classes
obj.feature_a()
obj.feature_b()
obj.feature_c()
obj.feature_d()
```

OUTPUT:

```
Feature A
Feature B
Feature C
Feature D
```

KEY CONCEPTS:

- Python uses **MRO (Method Resolution Order)** to resolve conflicts in Hybrid Inheritance.
- You can check the order using:

```
print(D.__mro__)
```

ADVANTAGES:

- Allows flexible and **powerful class design**.
- Enables combining multiple features from different class hierarchies.

DISADVANTAGES:

- Can be **difficult to manage and understand**, especially if class names or method names conflict.
- Increases **complexity** of the codebase.

SUMMARY:

- Hybrid Inheritance = **Mix of multiple inheritance types**.
- Use it carefully to avoid confusion.
- Python handles it smartly using MRO (left-to-right order).

Python random **Module**

The `random` module in Python is used to generate **random numbers** and perform **random operations**, such as selecting random elements from a list, shuffling, or generating random data for simulations and games.

You need to import it before using:

```
import random
```

COMMON FUNCTIONS IN `RANDOM` **MODULE**

Function	Description	Example
`random.random()`	Returns a random float number between **0.0 and 1.0**	`random.random()` → 0.7345
`random.randint(a, b)`	Returns a random **integer** between **a and b** (both included)	`random.randint(1, 10)` → 7
`random.randrange(start, stop[, step])`	Returns a random number from a given range (like `range()`)	`random.randrange(0, 10, 2)` → 8
`random.choice(sequence)`	Returns a **random element** from a list, tuple, or string	`random.choice(['apple', 'banana', 'cherry'])` → `'banana'`
`random.choices(sequence, k=n)`	Returns a **list of n random elements** (with replacement)	`random.choices([1, 2, 3], k=2)` → `[2, 2]`
`random.sample(sequence, k=n)`	Returns **n unique random elements** (without replacement)	`random.sample([1, 2, 3, 4], k=2)` → `[3, 1]`
`random.shuffle(list)`	**Shuffles** the elements of a list in place (changes original order)	`random.shuffle(my_list)`

1. Random integer

```
import random
num = random.randint(1, 100)
print(num)
```

2. Random choice from list

```
fruits = ['apple', 'banana', 'mango', 'cherry']
print(random.choice(fruits))
```

3. Shuffle list

```
numbers = [1, 2, 3, 4, 5]
random.shuffle(numbers)
print(numbers)
```

4. random.randrange(start, stop, step)

```
print(random.randrange(0, 10, 2))  # even numbers between 0-10
```