PYTHON IMPORTAN QUESTION & ANSWERS FOR INTERVIEW PREPRATION

Table of Content (Clickable)

---------- **TRICKY QUESTIONS** ----------

OOPS IN PYTHON

Q-1. What is the difference between a compiler and an interpreter? and which is used by python?

	COMPILER	INTERPRETER
1	It scans entire program first and translate it into machine code.	It scans the program line by line and translate it into machine code.
2	complier shows all errors and warnings at same time.	Interpreter show one error at a time.
3	Error occurs after scaning the whole program.	Error occurs after each line.
4	Debugging is slow.	Debugging is fast.
5	Execution time is less.	Execution time is more.
6	complier is used by languages such as C, C++ etc.	Interpreter is used by languages such as Java, Python, Javascript etc.

Q-2. What are the different types of data types in Python?

→ Mutable data types in Python (that can be change)
- List
- Sets (it's value can add and remove but can't update!)
- Dictionary

→ Immutable data types in Python (that can't be change)
- Numbers (int, float, complex_number)
- Strings
- Tuples

LIST	Mutable | Ordered | Indexed | Allow duplicate
TUPLE	**Immutable** | Ordered | Indexed | Allow duplicate
DICT	Mutable | Unordered | Doesn't allow duplicate
SET	Mutable | Unordered | Doesn't allow duplicate

Q-3. What is the difference between List and Tuple?

→ The Major difference between List and Tuple is that List is **mutable** List can be changed but Tuple is **Immutable** Tuple can't be changed.

Q-4. How many types of variables exist in Python?

→ There are four types of variables in Python.

1) Static variable or class variable (that can be inside the class and outside a constructor)
2) Instance variable (that can be inside the class's constructor with self, like self.name etc)
3) Local variable (that can be inside a function or method)
4) Global variable (that can be globally available in the program with global keyword)

What is Comprehension code?

→ A code that can be written in more readable form or multiple line of code written in less line of code is called comprehension code.

Q-5. What is Dictionary and List Comprehension in Python?

Comprehension Syntax:

[expression for item in iterable if condition]

expression means that can be print. you can perform actions in expression like i*i or i+i

→ **List Without Comprehension:**

```
list = []
for i in range(15):
    if(i%2==0): # even numbers
        list.append(i)
print(list)
# Output:
# [0, 2, 4, 6, 8, 10, 12, 14]
```

→ **List With Comprehension**

```
list1 = [i for i in range(15) if i%2==0]
print(list1)
# Output:
# [0, 2, 4, 6, 8, 10, 12, 14]
```

→ **Dictionary without Comprehension:**

```
dict = {}
for i in range(11):
    if(i%2==0):
        dict[i] = i*i
print(dict)
# Output:
# {0: 0, 2: 4, 4: 16, 6: 36, 8: 64, 10: 100}
```

→ **Dictionary with Comprehension**

[key:value for item in iterable if condition]

```
dict1 = {i:i*i for i in range(11) if i%2==0}
print(dict1)
# Output:
# {0: 0, 2: 4, 4: 16, 6: 36, 8: 64, 10: 100}
```

Q-6 What are modules and packages in Python?

Module is a single file containing Python code, whereas a package is a collection of modules that are organized in a directory hierarchy.

Types of Modules:

1) Built-in Modules:

- These modules are ready to import and use with the python interpreter. there is no need to install such modules explicitly.

2) External Modules:

- These modules are imported from a third party file or can be installed using a package manager like pip or conda. Since this code is written by someone else, we can install different versions of a same module with time.

Q-7 What is a constructor in Python?

The constructor is a method that is called when an object is created. This method is defined in the class and can be used to initialize basic variables.

If you create four objects, the class constructor is called four times. Every class has a constructor, but its not required to explicitly define it.

```
class A:
    def __init__(self):
        print("constructor called")

obj = A()   # constructor called
obj = A()   # constructor called
obj = A()   # constructor called
obj = A()   # constructor called
```

Q-8 What is Multithreading and Multiprocessing in Python?

Multithreading:

→ Multithreading is a powerful tool in Python that allows you to run multiple threads concurrently in a single program. This can greatly improve the performance of your application, as it allows you to perform multiple tasks simultaneously.

- You can use it to import built in module "import threading"
- or you can use it with "from concurrent.futures import ThreadPoolExecutor"

1) Process:

A process is like a self-contained program that's running on your computer. Imagine it as a person working on a task. Each process has its own memory space, resources, and code that it executes. Processes are separate from each other, which means they can't directly access each other's memory or resources. Each process runs independently and doesn't interfere with the workings of other processes. For example, when you run a web browser, a music player, and a word processor, each of these is a separate process.

2.Thread:

A thread is like a smaller unit of a process. Think of it as a task or a job that a process can perform. Threads within the same process share the same memory space and resources, which allows them to communicate and share data more easily than processes. Threads can run concurrently within a process, which means they can work on different tasks at the same time. Each thread will have its own memory for registers and stacks. However, threads within the same process need to coordinate and manage access to shared resources to avoid conflicts. For instance, in a web browser process, there might be multiple threads handling tasks like rendering web pages, fetching data, and managing user interactions.

There are two ways to achieve multi-threading in Python programs.

1) Using the threading module
2) Using Python concurrent.features

Python threading module provides a simple and straightforward method to implement multi-threading while concurrent.futures module provides a high-level interface for asynchronously executing functions using threads or processes. It includes the ThreadPoolExecutor and ProcessPoolExecutor classes that manage worker threads or processes, respectively.

For instance, if you have a script that needs to download multiple files from the internet, you can use a thread pool to download them concurrently without blocking the script's execution.

```
from concurrent.futures import ProcessPoolExecutor
import requests

def downloadFile(url, name):
    print(f"Started downloading file{name}")
    response = requests.get(url)
    with open(f"files/file{name}.jpg", 'wb') as f:
        f.write(response.content)
    print(f"Finished downloading file{name}")

url = "https://picsum.photos/2000/3000"

if __name__ == '__main__':
    with ProcessPoolExecutor() as executor:
        url_list = [url for i in range(40)]
        name_list = [i for i in range(40)]
        results = executor.map(downloadFile, url_list, name_list)
        for r in results:
            print(r)
```

This program will download image and store it concurrently without blocking the script's execution.

Multithreading:

Concept: Involves creating multiple threads within a single process, allowing concurrent execution of tasks.

Memory: Threads share the same memory space, making communication and data sharing efficient.

Execution: Threads are typically scheduled by the operating system, which determines which thread runs at any given time.

GIL: In Python, the Global Interpreter Lock (GIL) prevents multiple threads from running Python code simultaneously, limiting true parallelism.

Use Cases: Well-suited for I/O-bound tasks (e.g., network requests, file operations) where waiting for external resources is common.

Multiprocessing:

Concept: Creates multiple processes, each with its own memory space, enabling parallel execution of tasks.

Memory: Processes have separate memory spaces, requiring explicit mechanisms for communication and data sharing (e.g., queues, pipes).

Execution: Processes are typically scheduled by the operating system, allowing for true parallelism across multiple cores.

GIL: Not affected by the GIL, enabling efficient use of multiple cores for CPU-bound tasks.

Use Cases: Well-suited for CPU-bound tasks (e.g., complex calculations, scientific simulations) that can benefit from parallel execution.

→ Key Differences:

Feature	Multithreading	Multiprocessing
Memory	Shared	Separate
Execution	Concurrency	Parallelism
GIL	Affected	Not Affected
Use Cases	Input/Output - bound	CPU-bound

Q-9 What is a Decorator?

A decorator is just a function that takes another function as an argument, add some kind of functionality and then return another functions

code Example:

```
def decorator_func(func):
    def wrapper_func():
        print("wrapper_func wroked")
        return func()
    print("decorator_func worked")
    return wrapper_func

# calling the function
def show():
    print("Show worked")

decorator_show = decorator_func(show)
decorator_show()
```

```
# Output:
"decorator_func worked"
"wrapper_func wroked"
"Show worked"
```

Q-10 How Memory managed in Python?

Stack Memory:

The refrence of a variable like (a=10) so, "a" is the refrence or name of the variable is stored in the Stack memory and contains the **id** of its object (value).

Private heap Memory:

The objects (values) is stored in the private heap memory and we cannot access it directly. we need to access these objects through its refrence **(id)** from stack memory.

Python Memory Manager:

Python memory manager is the program in Python and its responsibility to get values from private heap memory.

Garbage Collector:

Garbage collector handles the dynamic objects (values) which has no name.

Q-11 What is the difference between Generators and Iterators?

Generator:

Generator is the special type of function that allow you to create an iterable sequence of values. A generator function returns a generator object, which can be used to generate the values one-by-one as you iterate over it.

You can create a generator by using "**yield**" statement in a function. The "**yield**" statement returns a value from the generator and suspends the execution of function until the next value is requested.

Here is an example:

```
def my_generator():
    for i in range(5):
        yield i
gen = my_generator()
print(next(gen))
print(next(gen))
print(next(gen))
# Output:
# 0
# 1
# 2
```

You can iterate all generators by using list() or for loop like,

```
for j in gen:
    print(j)
```

Benifits of Generators:

The main benifit of generators is that they allow you to generate the values on-the-fly (موقع پر) rather than having to create and store the entire sequence in memory like list, tuple and sets.

A generator is the powerful tool for working with large or complex data sets, as you can generate the values as you need them, rather than having to store them all in memory at once.

Iterator:

behind the scene for loop also use iterator and we cannot back using iterator. we used iter() function for iterate and next() function to get the value.

Code example:

```
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)
# Output
# apple
# banana
# cherry
```

Behind the scene a for loop working like this,

```
fruits_iterator = iter(fruits)
while True:
    try:
        fruit = next(fruits_iterator)
        print(fruit)
    except StopIteration:
        break
# Output
# apple
# banana
# cherry
```

Difference between iterator and generator?

Iterator works with iterable objects like, string,list,tuples,sets and dictionaries. it dosen't work with integer. A generator is a special type of function that returns an iterator.

Q-12 What is __init__.py file?

In Python projects, if you create a file called __init__.py in a directory then Python will treat that directory as a package. A package in Python is a collection of modules (individual .py files) that can be imported into other Python files.

In older versions of Python (before 3.3), it was necessary to create an __init__.py file in a package before you could use import statements in the form from mypackage import mymodule. Since version 3.3 and the implementation of PEP 420, Python will automatically create “Namespace Packages” implicitly in many cases. This means that __init__.py is now often optional, but it’s still useful to structure your initialization code and define how statements like from mypackage import * should work.

- Without __init__.py, Python wouldn't recognize your directory as a package and you wouldn't be able to import modules from it using the standard "import" statement.

And what is __pycache__ folder?

The __pycache__ folder is a directory that Python creates in your project when you run a script. This folder contains **.pyc** files, which are compiled versions of your Python scripts. These files are in a format called bytecode, which is a low-level set of instructions that can be executed by a Python interpreter.

Q-13 What is ternary operator in Python?

The syntax for the python ternary statement is as follows:

[if_true] if[expression] else [if_false]

Example:

```
age = 18
print("You can drive") if age >= 18 else print("You cannot drive")
# Output
# You can drive
```

Q-14 What is Inheritance in Python?

In inheritence, the child class can access all the data members and functions defined in the parent class. in python, a derived class can inherit base class by just mentioning the base in the bracket after the derive class name.

Example:

```
class A:
    def hello(self):
        print("hello")
class B(A):
    pass

obj = B()
obj.hello()
# Output
# hello
```

Q-15 Explain Break, Continue and Pass Statement?

Break:

A break statement, when used inside the loop, will terminate the loop and exit. if used inside nested loops, it will break out from the current loop.

Continue:

A continue statement will stop the current execution when used inside a loop, and the control will go back to the start of the loop.

Pass:

A pass statement is a null statement. when the python interpreter comes accross the pass statement, it does nothing and is ignored.

Q-16 What is "self" keyword in Python?

The **"self"** parameter is a refrence to the current instance of the class, and is used to access variables that belongs to the class.

Here you can see the self object.

```
class A:
    def hello(self):
        print(self)
obj = A()
obj.hello() # it will print object itself
# Output:
# <__main__.A object at 0x0000019432565B50>
```

Q-17 What is the difference between Pickling and Unpickling in Python?

Pickling:

In python, the pickle module accepts any python object (string,list,etc), transforms it into a string representation, and dumps it into a file by using the dump function. this process is known as pickling.

The function used for this process is pickle.dump()

```
import pickle

# Create a list of objects
data = [10, "Hello", {"name": "Alice", "age": 30}]

# Pickle the data and save it to a file
with open("data.pkl", "wb") as f:
    pickle.dump(data, f)
```

This program will create a file data.pkl and store that object into a string representation or byte stream.

Unpickling:

The process of retrieving the original python object from the stored string representation is called unpickling.

The function used for this process is pickle.load()

```
# Load the pickled data from the file
with open("data.pkl", "rb") as f:
    loaded_data = pickle.load(f)

print(loaded_data)  # Output: [10, 'Hello', {'name': 'Alice', 'age': 30}]
```

They are inverses of each other.

Pickling, also called serialization, involves converting a python object into a series of bytes which can be written out to a file.

Unpickling, or de-serialization, does the opposite. it converts a series of bytes into the python object.

Q-18 What are Python Iterators?

An iterator is an object which contains a countable number of values and it is used to iterate over iterable objects like list, tuples, sets, etc.

- Iterators are used mostly to iterate or convert other objects to an iterator using iter() function.
- Iterator uses iter() and next() functions.
- Every iterator is not a generator.
- But, A generator is a special type of function that returns an iterator.

Example:

```
iter_list = iter(['A','B','C'])
print(next(iter_list))
print(next(iter_list))
print(next(iter_list))
# Output:
# A
# B
# C
```

Q-19 Explain Type Conversion in Python?

In Python, Type conversion refers to the process of changing a value from one data type to another. This is often necessary when performing operations or functions that require specific data types.

Types of Type Conversion

Implicit Conversion:

Python automatically converts data types when it's safe and logical to do so.

Common examples:

Adding an integer to a float: The integer is converted to a float before the addition.

```
a = 2
b = 3.0
print(a+b)
# Output
# 5.0
```

Dividing two integers: The result is a float.

```
a = 6
b = 2
print(a/b)
# Output
# 3.0
```

Explicit Conversion:

You explicitly specify the desired data type using built-in functions.

Common functions:

int(): Converts to an integer.

float(): Converts to a float.

str(): Converts to a string.

bool(): Converts to a boolean (True or False).

list(): Converts to a list.

tuple(): Converts to a tuple.

set(): Converts to a set.

dict(): Converts to a dictionary.

Example:

Implicit conversion

```
x = 10  # Integer
y = 3.5  # Float
z = x + y  # z becomes 13.5 (float)
```

Explicit conversion

```
a = "123"  # String
b = int(a)  # b becomes 123 (integer)
c = float(a)  # c becomes 123.0 (float)
```

Conversion to boolean

```
d = bool(0)  # d becomes False
e = bool(1)  # e becomes True
f = bool("hello")  # f becomes True (non-empty string)
```

Q-20 What is *args & **kwargs?

When you are not clear how many arguments you need to pass to a particular function, then we use *args and **kwargs.

The *args keyword represents a varied number of arguments (list,tuples,etc...). It is used to add together the values of multiple arguments.

Code Example:

```
def great(*names):
```

```
    for name in names:
        print(f"Hello, {name}!")

great("Azhar","Ali","Usman")
# Output
# Hello, Azhar!
# Hello, Ali!
# Hello, Usman!
```

The **kwargs keyword represents an arbitary number of arguments (dictionary) that are passed to a function. **kwargs keywords are stored in a dictionary. you can access each item by referring to the keyword you associated with an argument when you passed the argument.

Code Example:

```
def create_person(**info):
    print("Name:", info["name"])
    print("Age:", info["age"])
    print("City:", info["city"])
create_person(name="Ali",age=22,city="Hyderabad")
# Output
# Name: Ali
# Age: 22
# City: Hyderabad
```

Q-21 What is Open and With statement?

Both statements are used in case of file handling.

- → With the "with" statement, you get better syntax and exceptions handling.
- → With "Open" statement you can ceate,read file like open("filename", "mode")
- → Without "with" statement you need to manually close the file but using "with" statemet it will automatically close the file.

Q-22 What is PYTHONPATH?

PYTHONPATH is an environment variable which is used when a module is imported. whenever a module is imported, PYTHONPATH is also looked up to check for the presence of the imported modules in various directories. You can check PYTHONPATH variable's directories using sys.path

If your imported module not found in PYTHONPATH so, you can add additional directories in python environment variable "PYTHONPATH" where python will look for module.

Here is a code to check your PYTHONPATH:

```
import sys
print(sys.path)
# Output
# ['c:\\Users\\HAYAT TRADERS\\Desktop\\PythontApnacollege', 'C:\\Users\\HAYAT
TRADERS\\AppData\\Local\\Programs\\Python\\Python312\\python312.zip', 'C:\\Users\\HAYAT
TRADERS\\AppData\\Local\\Programs\\Python\\Python312\\DLLs', 'C:\\Users\\HAYAT
TRADERS\\AppData\\Local\\Programs\\Python\\Python312\\Lib', 'C:\\Users\\HAYAT
```

```
TRADERS\\AppData\\Local\\Programs\\Python\\Python312', 'C:\\Users\\HAYAT
TRADERS\\AppData\\Local\\Programs\\Python\\Python312\\Lib\\site-packages']
```

Q-23 How Exception is handled in Python?

Exception is handled in Python using these keywords:

- **Try**: This block will test the exceptional error to occur.
- **Except**: Here you can handle the error.
- **Else**: If there is no exception then this block will be executed.
- **Finally**: Finally block always gets executed either exception is generated or not.

Example:

```
try:
    pass
   # some code
except:
    pass
    # handling of exception
else:
    pass
    # execute if no exception
finally:
    pass
    # Always execude
```

Q-24 What is PIP in Python?

PIP is the package manager for Python packages. we can use pip to install packages that do not come with python.

Syntax:

pip install <package_name>

Q-25 Where Python is used?

- Web Application
- Desktop Application
- Database Application
- Networking Application
- Machine Learning
- Artificial Intelligence
- Data Analysis
- IOT Applications
- Games and many more...

Q-26 How to use F-String and Format / replacement operator?

How to use f-string:

```
name = 'Ali'
role = 'Python Developer'
print(f"Hello, My name is {name} and I'm {role}")
# Output:
# Hello, My name is Ali and I'm Python Developer
```

How to use format Operator:

```
name = 'Ali'
role = 'Python Developer'
print(("Hello, My name is {} and I'm {}").format(name,role))
# Output:
# Hello, My name is Ali and I'm Python Developer
```

Q-27 How to get all keys in a dictionary?

We can get all keys of a dictionary using **keys()** function.

Example:

```
d = {'A':1, 'B':2, 'C':3}
x = d.keys()
print([k for k in x])
# Output:
# ['A','B','C']
```

Q-28 What is the difference between Abstraction & Encapsulation?

Abstraction and encapsulation are two crucial concepts of Object Oriented Programming(OOP) in the field of computer science that are used to make code more manageable, secure, and efficient.

What is Abstraction in Python?

Abstraction is a technique that involves hiding the implementation details of a class and only exposing the essential features of the class to the user. This allows the user to work with objects of the class without having to worry about the details of how the class works.

For example, let's say we have a class called **Car** that has methods such as **start_engine(), accelerate()**, and **brake()**. The user only needs to know that they can start the engine, accelerate, and brake the car, but they don't need to know how these methods are implemented.

Here's what the code would look like:

```
class Car:
    def start_engine(self):
        # Implementation details hidden
        print("Engine started")

    def accelerate(self):
        # Implementation details hidden
        print("Accelerating")

    def brake(self):
```

```
        # Implementation details hidden
        print("Braking")

tesla = Car()
tesla.start_engine()  # Engine started
tesla.accelerate()  # Accelerating
tesla.brake()  # Breaking
```

In this example, the user can use the Car class in a straightforward way, without having to worry about the implementation details.

What is Encapsulation in Python?

Encapsulation is a technique that involves wrapping data and functions into a single unit (class). This allows you to control the way the data and functions are accessed and helps to prevent accidental modification of the data.

For example, let's say we have a class called **BankAccount** that has attributes such as balance and **account_number** and methods such as **deposit()** and **withdraw().** We want to ensure that the balance can only be modified through the **deposit()** and **withdraw()** methods and not directly. Here's what the code would look like:

```
class BankAccount:
    def __init__(self, balance, account_number):
        self.__balance = balance
        self.__account_number = account_number

    def deposit(self, amount):
        self.__balance += amount

    def withdraw(self, amount):
        if self.__balance >= amount:
            self.__balance -= amount
        else:
            print("Insufficient funds")

    def get_balance(self):
        return self.__balance

user = BankAccount(5000,123)
print(user.get_balance())  # 5000
user.deposit(5000)
print(user.get_balance())  # 10000
user.withdraw(3000)
print(user.get_balance())  # 7000
```

In this example, the balance and account_number attributes are prefixed with two underscores, which makes them **private**. This means that they can only be accessed within the **BankAccount** class and not outside of it. The user can only modify the balance through the **deposit()** and **withdraw()** methods, ensuring that the data is secure.

Conclusion!

Abstraction involves hiding the implementation details of a class and only exposing the essential features, while **Encapsulation** involves wrapping data and functions into a single unit or object and controlling the way they are accessed.

In Python, Abstraction is achieved by creating classes and hiding the implementation details of the methods.

Encapsulation is achieved by making the data and methods private and only accessing them through public methods.

By understanding and using abstraction and encapsulation in Python, you can make your code more organized and maintainable, and ensure that your

Q-29 What is if __name__ == '__main__' ?

In Short: It Allows You to Execute Code When the File Runs as a Script, but Not When It's Imported as a Module.

Here is an example:

```
# data.py file
def hello():
    print('hello')

hello()
```

```
# main.py file
import data

data.hello()
# Output
# hello
# hello
```

In this example you can see the problem that function is calling 2 times.

After using **if __name__ == '__main__'**

```
# data.py file
def hello():
    print('hello')

if __name__ == '__main__':
    hello()
```

```
# main.py file
import data

data.hello()
# Output
# hello
```

Now it prints only one time, so that how its work.

---------- Tricky Questions ----------

Q-30 Does Python support multiple inheritance? or Diamond problem?

Yes, Python supports multiple inheritance.

What is diamond problem?

Java does not allow multiple inheritance where one class can inherit properties from more than one class. it is known as the **diamond problem**.

Example of multiple inheritance in Python:

```
class A:
    def abc(self):
        print("a")

class B(A):
    def abc(self):
        print("b")

class C(A):
    def abc(self):
        print("c")

class D(B,C):
    pass

d = D()
d.abc()

# Output:
# b
```

it will work on sequential manner (order).

Q-31 How to initialize Empty List, Tuple, Dict and Set?

```
# Empty List:
a = []
# Empty Tuple:
a = ()
# Empty Dict:
a = {}
# Empty Set:
a = set()
```

Q-32 What is the difference between .py & .pyc file?

- **.py** files contain the source code of a program. whereas, **.pyc** file contains the bytecode of your program (compiled code).
- Python complies the .py files and saves it as .pyc files, so it can reference them in subsequent invocations.
- The .pyc file contain the compiled bytecode of Python source files. This code is then executed by Python's virtual machine.

Q-33 How slicing works in string manipulation?

Syntax:

str_obj[start_position : end_position : step]

code Example:

```
s = 'HelloWorld'
# step_1:
print(s[:])
# Output:
# HelloWorld

# step_2:
print(s[::])
# Output:
# HelloWorld

# step_3:
print(s[:5])
# Output:
# Hello

# step_4:
print(s[2:5])
# Output:
# llo

# step_5:
print(s[2:8:2])
# Output:
# loo

# step_6:
print(s[8:1:-1]) # it will reverse the order
# Output:
# lroWoll

# step_7:
print(s[-4:-2])
# Output:
# or
```

```
# step_8:
print(s[::-1]) # it will reverse #it is also used for Palindrome words (eye reverse(eye))
# Output:
# dlroWolleH
```

Q-34 Can we concatenate two Tuples? if yes? How? Since Tuple is Immutable.

Yes, we can using **+ operator** like,

```
tuple_1 = (1,2,3)
print(id(tuple_1))  # 1264060499008
tuple_2 = (7,8,9)
tuple_1 = tuple_1 + tuple_2
print(id(tuple_1))  # 1264057177888
print(tuple_1)

# Output:
# 1,2,7,8,9
```

In this case the tuple_1 reference will change because it's immutable but in list reference will remain same like object id. you can check id using id() function.

Q-35 What is the difference between List & Array?

We can use array by importing it in python like,

importing "array" for array creations

```
import array as arr

# creating an array with integer type
a = arr.array('i', [1, 2, 3])
print(type(a))
for i in a:
    print(i, end=" ")

# Output:
# <class 'array.array'>
# 1 2 3
```

Difference between List & Array:

- The list can store the value of different data types whereas Array cannot. Array can store the value of same data type.
- The list are built-in data structure so we don't need to import it whereas we need to import Array before work with array.

Q-36 How to read multiple values from single input?

By using split() method

```
# using list comprehension with map function
x = list(map(int, input("Enter multiple value: ").split()))
print("List of values: ",x)

# using list comprehension with for loop
x = [int(x) for x in input("Enter multiple value: ").split()]
print("Number of list is: ",x)

# using list comprehension with for loop and split() with comma ,
x = [int(x) for x in input("Enter multiple value: ").split(",")]
print("Number of list is: ",x)
```

Q-37 How to copy & delete a dictionary?

Delete by using clear()

```
d1 = {'A':1, 'B':2, 'C':3}
d1.clear()
print(d1)
# Output
# {}
```

Delete by using pop()

```
d1 = {'A':1, 'B':2, 'C':3}
d1.pop('A')
print(d1)  # {'B':2, 'C':3}
```

Delete by using del keyword

```
d1 = {'A':1, 'B':2, 'C':3}
del d1['B']
print(d1)  # {'A':1, 'C':3}
```

Copy by using copy()

```
d1 = {'A':1, 'B':2, 'C':3}
d2 = d1.copy()
print(d2)
# Output
# {'A': 1, 'B': 2, 'C': 3}
```

Note:

Copy by using = sign (this way is not safe for copy because it will not create copy it will reference the objects into another variable)

```
d1 = {'A':1, 'B':2, 'C':3}
d2 = d1
```

```
print(id(d1)) # 2058013255552
print(id(d2)) # 2058013255552
```

It will print the same id because, the reference is same.

Q-38 What is the difference between Anonymous & Lambda function?

- In practical both are same but theoretically are different.
- If you assign anonymous function to a variable, then it will call as lambda function otherwise it is anonymous function.

Q-39 What is GIL (Global Interpreter Lock)?

The Global Interpreter Lock (GIL) in Python is a locking mechanism that ensures that only one thread can execute Python code at a time, even on multi-core processors.

Q-40 What is Namespace and its types in Python?

Namespace:

In python we deal with variables, functions, libraries and modules etc.

There is the chance the name of a variable you are going to use is already existing as name of another variable or as the name of another function or another method.

In such scenario, we need to learn about how all these names are managed by python program.

this is the concept of namespace.

There are 3 types of Namespace:

- Local Namespace
- Global Namespace
- Built-in Namespace

you can check all built-in namespace using this code.

```
builtin_names = dir(__builtins__)
for name in builtin_names:
    print(name)
```

It will print all built-in namespace.

Q-41 Explain Recursion by reversing a List?

Reverse a list using recursion:

```
def reverse_list(lst):
    if not lst:
        return []
    return [lst[-1]] + reverse_list(lst[:-1])

print(reverse_list([1,2,3,4]))
# Output
```

```
# [4, 3, 2, 1]
```

Q-42 What is the difference between map(), filter() and reduce() ?

Map()

- The map() function iterates through all items in the given iterable and executes the function we passed as an argument on each of them. and it returns map object.

The syntax is:

map(function, iterable(s))

Example:

```
fruits = ["Apple", "Banana", "Pear"]
map_obj = map(lambda s:s[0] == "A", fruits)
print(list(map_obj))

# Output:
# [True, False, False]
```

Filter()

- The filter() function takes a function object and an iterable and ceates a new list. As the name suggests, filter() forms a new list that contain only elements that satisfy a certain condition. i.e the function we passed True.

The syntax is:

filter(function, iterable(s))

Example:

```
fruits = ["Apple", "Banana", "Pear", "Apricot"]
filter_obj = filter(lambda s: s[0] == "A", fruits)
print(list(filter_obj))

# Output:
# ["Apple", "Apricot"]
```

Reduce()

- The reduce() function works differently than map() and filter(). It doesn't return a new list based on the function and iterable we've passed. Instead, it returns a single value. Also, in Python 3 reduce() isn't a built-in function anymore, that's why we need to import it before using. it can be found in the functools module.

The syntax is:

reduce(function, sequence[, initial])

Example:

```
from functools import reduce
```

```
list = [2,4,7,3]
print(reduce(lambda x,y: x+y, list))
print("With an initial value: ", reduce(lambda x,y: x+y, list, 10))

# Output:
# 16
# With an initial value: 26
```

Q-43 What is the difference between Shallow copy & Deep copy?

Shallow Copy:

- Shallow copies duplicate as little as possible. A shallow copy of a collection is a copy of the collection structure, not the elements. with a shallow copy, two collections now share the individual elements.
- Shallow copy is creating a new object and then copying the non-static fields of the current object to the new object. If the field is value type, a bit by bit copy of the field is performed. if the field is reference type, the reference is copied but the referred object is not, therefore the original object and its clone refer to the same object.

Deep Copy:

- Deep copies duplicate everything. A deep copy of a collection is two collections with all of the elements in the original collection duplicate.
- Deep copy is creating a new object and then copying the non-static fields of the current object to the new object. if a field is a value type, a bit by bit copy of the field is performed. if a field is a refrence type, a new copy of the referred object is performed. a deep copy of an object is a new object with entirely new instance variables. it doesn't share objects with the old. while performing deep copy the classes to be cloned must be flagged as [Serializable].

Q-44 How an object is copied?

Same as shallow copy and deep copy.

Q-45 What is operator overloading and dunder method?

Dunder methods in python are special methods.

- In python, we sometimes see method names with a double underscore __ such as the __init__ method that every class has. These method are called "dunder" methods.
- In python, dunder methods are used for operator overloading and customizing some other function's behavior.

Some Examples:

\+ __add__(self, other)

\- __sub__(self, other)

* __mul__(self, other)

/ __truediv__(self, other)

// __floodiv__(self, other)

% __mod__(self, other)

** __pow__(self, other)

>> __rshift__(self, other)

<< __lshft__(self, other)

& __and__(self, other)

| __or__(self, other)

^ __xor__(self, other)

Q-46 Draw some patterns / graph?

```
Q1 - Print this Pattern
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
# Q1 Solution
for i in range(1,6):
    for j in range(1,6):
        print("*", end=" ")   # end="" means it shouldn't jump into new line it should
remain in same line
    print(" ")  # for new line

Q2 - Print this Pattern
*
* *
* * *
* * * *
* * * * *
# Q2 Solution
for i in range(1,6):
    for j in range(1,i+1):
        print("*", end=" ")
    print(" ")
```

```
Q3 - Print this Pattern
* * * * *
* * * *
* * *
* *
*
# Q3 Solution
for i in range(6,1,-1):
    for j in range(1,i):
```

```
        print("*", end=" ")
    print(" ")
```

```
Q4 - Print this Pattern
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
# Q4 Solution
for i in range(1,6):
    for j in range(1,i+1):
        print(j, end=" ")
    print(" ")
```

```
Q5 - Print this Pattern
5 4 3 2 1
4 3 2 1
3 2 1
2 1
1
# Q5 Solution
for i in range(5,0, -1):
    for j in range(i,0,-1):
        print(j, end=" ")
    print(" ")
```

```
Q6 - Print this Pattern
        *
      * * *
    * * * * *
  * * * * * * *
* * * * * * * * *
# Q6 Solution
for i in range(1,6):
    for j in range(6,i-1,-1):
        print(" ", end=" ")
    for k in range((2*i -1)):
        print("*", end=" ")
    print(" ")
```

```
Q7 - Print this Pattern
* * * * * * * * *
  * * * * * * *
    * * * * *
      * * *
        *
# Q7 Solution
```

```
for i in range(5,0,-1):
    for j in range(5,i-1,-1):
        print(" ", end=" ")
    for k in range((2*i -1)):
        print("*", end=" ")
    print(" ")
```

This Pattern is Your HomeWork:

```
Q8 - Print this Pattern
        *
      * * *
    * * * * *
  * * * * * * *
* * * * * * * * *
  * * * * * * *
    * * * * *
      * * *
        *
```

---------- OOPS IN PYTHON ----------

Important Topics in OOP:

1. **Class**
2. **Object**
3. **Abstraction**
4. **Encapsulation**
5. **Polymorphism**
6. **Inheritance**

What is a Class?

Class is the blueprint for an object. before creating an object, we need to define its structure like, in a car factory they create a structure for a car before its creation (object) so, that structure or blueprint is called a **class**.
"we also called datatype a class". Like, string is a datatype if you check its type it will return **class string**.
Here is example:

```
my_number = 123
print(type(my_number))
# Output
# <class 'int'>
```

What is an object?

Everything in Python is an object. like int value, float value, boolean value, string value, function, list etc.
If someone ask how is it possible so, **here is an example to proof it:**

```
my_number = 123
# applying string class method to a int class object
```

```
print(my_number.upper())
# Output
# 'int' object has no attribute 'upper'
```

In this example you can see that the value of int class **123** is an object.

What is a constructor?

In Python, a constructor is a special method called when an object is created. Its purpose is to assign values to the data members within the class when an object is initialized. it's take object itself as an argument and we can also pass data or property values in the argument of the constructor __init__(self) **self, is the reference to the object**.

What is Magic (dunder) Methods?

- They are special methods in python which are defined inside class.
- They begin and end with double underscore (__)
- They are special where because they get called themselves.

Instance variable:

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

s1 = Student('Ali',23)
s2 = Student('Sami',19)
```

s1 and **s2** above are instance variables which have diff values of attributes and also in the __init__() method variables **self.name** and **self.age** are also called instance variables.

static variables or class variables:

class variables those are inside the class and outside the __init__() method is called class level variables.

```
class Student:
    college = "GCUH"
    def __init__(self, name, age):
        self.name = name
        self.age = age

s1 = Student('Ali',23)
```

college is the class or static variable.

Access Modifiers (public, private, protected):

In high-level programming languages like C++, Java, etc., private, protected, and public keywords are used to control the access of class members or variables. However, Python has no such keywords. Python uses a convention of

prefixing the name of the variable or method with a single underscore(_) or double underscore(__) to emulate the behavior of protected and private access specifiers.

Access modifiers are used for the restrictions of access any other class has on the particular class and its variables and methods. In other words, access modifiers decide whether other classes can use the variables or functions of a specific class or not. The arrangement of private and protected access variables or methods ensures the principle of data **encapsulation**. In Python, there are three types of access modifiers.

- Public Access Modifier
- Protected Access Modifier
- Private Access Modifier

Public Access Modifier:

Example of public access modifier:

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

self.name and **self.age** are public access modifier.

Protected Access Modifier:

In the case of a protected class, its members and functions can only be accessed by the classes derived from it, i.e., its child class or classes. No other environment is permitted to access it. To declare the data members as protected, we use a single underscore "_" sign before the data members of the class.

Example of protected access modifier:

```
# Parent class
class employee:
      def __init__(self):
            self._name= "Ali" # protected attribute
            self._age= "22" # protected attribute

# Child class
class Manager(employee):
    pass

obj = Manager()
print(obj._name)
print(obj._age)
# Output
# Ali
# 22
```

In this example you can see child class (which is derived from parent class) can get the protected members of the parent class.

Private Access Modifier:

In the case of private access modifiers, the variables and functions can only be accessed within the class. The private restriction level is the highest for any class. To declare the data members as private, we use a double underscore "__" sign before the data members of the class. Here is a suggestion not to try to access private variables from outside the class because it will result in an AttributeError.

Example of private access modifier:

```
class employee:
    def __init__(self, name, age):
        self.__name = name # private attribute
        self.__age = age # private attribute
```

getters and setters method in Python:

If you come from a language like Java or C++, then you're probably used to writing getter and setter methods for every attribute in your classes. These methods allow you to access and mutate private attributes while maintaining encapsulation. In Python, you'll typically expose attributes as part of your public API and use properties when you need attributes with functional behavior.

we mostly use it when we use private variables for getting that value we create a method it's called getter (we can define any name) and for update that value we create a method it's called setter.

Here is an example of getter and setter method:

```
class employee:
    def __init__(self, name, age):
        self.__name = name # private attribute
        self.__age = age # private attribute

    # getter
    def get_name(self):
        print(self.__name)

    # setter
    def set_name(self, name):
        self.__name = name

emp1 = employee("Ali",32)
emp1.get_name() # Ali
emp1.set_name("Azhar")
emp1.get_name() # Azhar
```

In this example you can see **setter** and **getter** method.

Static Method (@staticmethod):

A static method does not receive an implicit first argument like **" self "**. A static method is also a method that is bound to the class and not the object of the class. This method can't access or modify the class state. It is present in a class because it makes sense for the method to be present in class. before create static method we write **@staticmethod**

Here is an example:

```python
class Car:

    @staticmethod
    def about_car():
        print("This is the car which has four wheels")

my_car = Car()
my_car.about_car()
# Output
# This is the car which has four wheels
```

Encapsulation:

The binding of our data members/attributes and methods in a single unit (class) is known as Encapsulation.

Inheritance:

- Inheritance allows us to define a class that inherits all the methods and properties from another class.
- Parent class is the class being inherited from, also called base class.
- Child class is the class that inherits from another class, also called derived class.
- We inherit non-private attributes, methods and constructor and (other magic methods).
- We cannot access private attributes and methods.
- Parent cannot inherit from child only child can inherit from parent.

super keyword:

- **super()** keyword is used to access methods of parent for our child class.
- cannot be used outside.
- can only access methods including __init__() and public attributes.

```python
class A:
    name = "A"
    def hello(self):
        print("hello")

class B(A):
    def hi(self):
        # get name
        print(super().name)
        # get method
        super().hello()

obj = B()
obj.hi()
# Output
# A
# hello
```

In this example you can see we got method and public attribute using super() keyword in child class.

Types of Inheritance:

- Simple
- Hierarchal
- Multilevel
- Multiple
- Hybrid

Simple Inheritance:

simple is that type where one is child and one is parent class.

```
class A:
    pass

class B(A):
    pass
```

Heirarichal Inheritance:

- Hierarchal is that type where two is child and one is parent.
- Both children get properties of parent.
- children are not related to each other.

```
class A:
    pass

class B(A):
    pass

class C(A):
    pass
```

Multilevel Inheritance:

- Multilevel is that type where one is child and is parent and one is grandparent.
- lower levels get properties from above levels.
- super() of each class can reach out to just the parent (not to the grandparent).

```
class Grandparent:
    pass

class Father(Grandparent):
    pass

class Child(Father):
    pass
```

Multiple Inheritance:

- Multiple is that type where child has multiple parents.
- if method "A()" is not in child and it is present in both parents so, python will check first parent not second one.

```
class Father:
    def A(self):
        print("A from Father")

class Mother:
    def A(self):
        print("A from Mother")

class Child(Father, Mother):
    pass

child1 = Child()
child1.A()
# Output
# A from Father
```

In this example you can see both parent class has **A()** method but Python did get only first parent like Father class.

Hybrid Inheritance:

- it is the mixture of inheritance.
- it is also called (diamond problem)

Now, it's your time learn more about Hybrid Inheritance.

Polymorphism:

Poly (multiple/many)

Morph (shape/form)

In simple words, polymorphism is when a single entity can take multiple form.

In Python, we achieve polymorphism by below:

1. Method overriding
2. Method overloading
3. Operator overloading

- Major benefit is that code becomes a lot clean to read and work upon.

Method overloading:

Two or more methods or (constructors) have the same name but different numbers of parameters or different types of parameters, or both. These methods are called overloaded methods and this is called method overloading.

How to achieve method overloading? here is the code:

```
def sum(a,b):
    print(a+b)

sum(2,3)  # 5

def sum(a,b,c):
    print(a+b+c)

sum(2,3,4)  # 9
sum(2,3) # Error
```

Solution:

- change the method name sum(a,b) to **sum2(a,b)** and sum(a,b,c) to **sum3(a,b,c)**

```
def sum2(a,b):
    print(a+b)

def sum3(a,b,c):
    print(a+b+c)

def sumBetter(a,b,c=0):
    if(c==0):
        sum2(a,b)
    else:
        sum3(a,b,c)

sumBetter(2,3)   # 5
sumBetter(2,3,4)  # 9
```

note: You can also do same thing with constructor.

Method overriding:

Method overriding in Python occurs when a child class defines a method that has the same name and parameters as a method in its parent class. The child class method overrides or replaces the parent class method when called on an instance of the child class.

Example:

```
class Animal:
    def sound(self):
        print("Animal sound...")

class Dog(Animal):
    def sound(self):
        print("Bark...")

obj = Dog()
```

```
obj.sound()  # Bark...
```

In this example sound() method has been override.

Operator overloading:

Python operator overloading refers to a single operator's capacity to perform several operations based on the class (type) of operands. like, a "+" operator works differently with multiple class type object like, int, str, list etc...

Example:

```
print(1+1)  # 2 (addition)
print("1" + "1")  # 11  (concatination)
print([1,2]+[3,4])  # [1,2,3,4] (merge)
```

Abstraction:

Abstract → Abstract Art (hidden)

Abstraction is the fundamental concept of OOPS that involve hiding complex implementation details & showing only essential features.

Example:

Phone → take call and make call (but you don't know how it's happening.)

Similarly, we have a lot of examples in real life.

- mobile
- human body
- TV remote

and you are using python built-in methods like list.append() but, don't know how it's working.

Abstract Method & Class

Import the ABC class and abstractmethod for creating abstract class and methods in python.
from abc import ABC, abstractmethod

1. Inherit ABC class
2. Should have an abstract method

code example:

```
from abc import ABC, abstractmethod

class User(ABC):
    @abstractmethod
    def login(self):
        pass

    def logout(self):
        print("logout")
```

```
    @abstractmethod
    def auth(self):
        pass

class Buyer(User):
    pass

obj = Buyer()
obj.logout()
# Output
# TypeError: Can't instantiate abstract class Buyer without an implementation for abstract
methods 'auth', 'login'
```

In this example you can see the error because we didn't implement the abstract method.

After implementation of abstract methods **'auth'**, **'login'**

```
class Buyer(User):
    def login(self):
        if(self.auth):
            print("Logged in User")

    def auth(self):
        return True

obj = Buyer()
obj.logout()
# Output
# logout
```

- So, Abstract classes are useful when we have a group of related objects that should share common features (login, logout).
- but should/will have different implementation.
- Helping in preparing a blueprint for other classes.
- we mostly create important methods in abstract class and use it inherit class with different implementation.

Difference between @staticmethod and @property?

@staticmethod and @property are two distinct decorators in Python, each serving a specific purpose:

@staticmethod:

- Purpose: Defines a method that belongs to the class itself, not to instances of the class.
- Characteristics: Doesn't have access to the self-parameter (the instance of the class).
- Can be called directly on the class or through an instance.
- Useful for utility functions that don't require access to instance-specific data and don't need to modify.

Example:

```
class MyClass:
    @staticmethod
    def greet(name):
        print(f"Hello, {name}!")
```

```
# Calling the static method
MyClass.greet("Alice")  # Output: Hello, Alice!

obj = MyClass()
obj.greet("Bob")        # Output: Hello, Bob!
```

@property:

- Purpose: Defines a property that can be accessed and potentially modified like a regular attribute, but its behavior is controlled by getter, setter, and deleter methods.
- Characteristics: Provides a way to control access and modification of attributes.
- Can be used to validate input, calculate values on-the-fly, or perform other actions.

Example:

```
class Person:
    def __init__(self, first_name, last_name):
        self._first_name = first_name
        self._last_name = last_name

    @property
    def full_name(self):
        return f"{self._first_name} {self._last_name}"

    @full_name.setter
    def full_name(self, name):
        first, last = name.split()
        self._first_name = first
        self._last_name = last


# Using the property
person = Person("Alice", "Smith")
print(person.full_name)  # Output: Alice Smith

person.full_name = "Bob Johnson"
print(person.full_name)  # Output: Bob Johnson
```