**1. What is the Difference Between a Shallow Copy and Deep Copy?**

* **Deepcopy** creates a different object and populates it with the child objects of the original object.
* Therefore, changes in the original object are not reflected in the copy.deepcopy() creates a Deep Copy.
* **Shallow copy**: creates a different object and populates it with the references of the child objects within the original object. Therefore, changes in the original object are reflected in the copy.copy creates a Shallow Copy.
* import copy
* # Original list with nested list
* original\_list = [[1, 2, 3], [4, 5, 6]]
* # Shallow copy
* shallow\_copy = copy.copy(original\_list)
* # Deep copy
* deep\_copy = copy.deepcopy(original\_list)
* # Modify nested list in original\_list
* original\_list[0][0] = 100
* print("Original List:", original\_list)
* print("Shallow Copy:", shallow\_copy)
* print("Deep Copy:", deep\_copy)

Use case:

* Imagine you have a nested data structure like a list of lists or a dictionary of dictionaries, and you want to create a copy of this structure to manipulate it without affecting the original data. Here, you might choose between a shallow copy and a deep copy based on whether you want changes in the copied structure to reflect in the original one.
* Use a shallow copy when you want to create a new object, but you're okay with inner elements being shared between the original and copied objects.
* Use a deep copy when you need a completely independent copy of the original object, especially if you plan to modify the copied object without altering the original one.

### 2. How Is Multithreading Achieved in Python?

* Multithreading usually implies that multiple [threads](https://www.simplilearn.com/tutorials/python-tutorial/python-threading) are executed concurrently. The Python Global Interpreter Lock doesn't allow more than one thread to hold the Python interpreter at that particular point of time. So multithreading in python is achieved through context switching. It is quite different from multiprocessing which actually opens up multiple processes across multiple threads.
* The Global Interpreter Lock (GIL) in Python is a mutex (or a lock) that protects access to Python objects, preventing multiple native threads from executing Python bytecodes simultaneously. This means that in a multi-threaded Python program, only one thread can execute Python bytecode at a time, even on multi-core systems.
* The GIL is a significant factor influencing the performance of multi-threaded Python programs, especially for CPU-bound tasks. However, it doesn't impact I/O-bound tasks much, as threads can release the GIL while waiting for I/O operations.

Here's an example to illustrate the impact of the GIL:

import threading

# A simple function that increments a shared counter

def increment(counter):

for \_ in range(1000000):

counter += 1

counter = 0 # Shared counter

# Create two threads that increment the counter

thread1 = threading.Thread(target=increment, args=(counter,))

thread2 = threading.Thread(target=increment, args=(counter,))

# Start the threads

thread1.start()

thread2.start()

# Wait for the threads to finish

thread1.join()

thread2.join()

print("Counter value:", counter)

* In this example, increment() function is supposed to increment a shared counter by 1,000,000 times. However, because of the GIL, the threads cannot execute Python bytecode concurrently, leading to a race condition. As a result, the final value of the counter may not be what is expected.
* To avoid the GIL limitation in CPU-bound tasks, you can use multiprocessing instead of multithreading, as multiprocessing allows you to bypass the GIL by using multiple processes instead of threads. Alternatively, you can use libraries or extensions written in C or Cython that release the GIL during CPU-intensive operations, allowing for better concurrency.

**3. Discuss Django Architecture.**

Here you can also find a [comprehensive guide on Python Django Tutorial](https://www.simplilearn.com/tutorials/python-tutorial/python-django) that is very easy to understand.

Django is a web service used to build your web pages. Its architecture is as shown:

* Template: the front end of the web page
* Model: the back end where the data is stored
* View: It interacts with the model and template and maps it to the URL
* Django: serves the page to the user

### 4. What Advantage Does the Numpy Array Have over a Nested List?

[Numpy](https://www.simplilearn.com/tutorials/python-tutorial/numpy-tutorial) is written in C so that all its complexities are backed into a simple to use a module. Lists, on the other hand, are dynamically typed. Therefore, Python must check the data type of each element every time it uses it. This makes Numpy arrays much faster than lists.

Numpy has a lot of additional functionality that list doesn’t offer; for instance, a lot of things can be automated in Numpy.

### 5. What are Pickling and Unpickling?

* **Serialization:** is the process of converting an object into a format that can be easily stored, transmitted, or reconstructed later.
* **Deserialization:** is the reverse process, where serialized data is converted back into an object.
* Pickling is also referred to as serialization
* In Python, serialization and deserialization are commonly achieved using libraries like pickle, json, xml, yaml, or msgpack, among others.
* serialization and deserialization using pickle and json as examples.

**Serialization with pickle:**

* import pickle
* # Define a sample object
* data = {'name': 'John', 'age': 30, 'city': 'New York'}
* # Serialize the object to a binary format
* serialized\_data = pickle.dumps(data)
* # Write the serialized data to a file
* with open('data.pkl', 'wb') as file:
* file.write(serialized\_data)

In this example, pickle.dumps() serializes the data dictionary into a binary format. This serialized data can be stored in a file, transmitted over a network, or saved in a database.

**Deserialization with pickle:**

* import pickle
* # Read the serialized data from a file
* with open('data.pkl', 'rb') as file:
* serialized\_data = file.read()
* # Deserialize the binary data back into an object
* deserialized\_data = pickle.loads(serialized\_data)
* print(deserialized\_data) # Output: {'name': 'John', 'age': 30, 'city': 'New York'}

Here, pickle.loads() deserializes the binary data back into the original Python object, which is a dictionary in this case.

**Serialization with json:**

* import json
* # Define a sample object
* data = {'name': 'John', 'age': 30, 'city': 'New York'}
* # Serialize the object to JSON format
* serialized\_data = json.dumps(data)
* # Write the serialized data to a file
* with open('data.json', 'w') as file:
* file.write(serialized\_data)

With JSON serialization, the data is converted into a human-readable format.

**Deserialization with json:**

* import json
* # Read the serialized data from a file
* with open('data.json', 'r') as file:
* serialized\_data = file.read()
* # Deserialize the JSON data back into an object
* deserialized\_data = json.loads(serialized\_data)
* print(deserialized\_data) # Output: {'name': 'John', 'age': 30, 'city': 'New York'}
* json.loads() deserializes the JSON-formatted data back into the original Python object.

In both examples, the data is serialized into a format suitable for storage or transmission, and then deserialized back into its original form for use in the program. Serialization and deserialization are essential techniques for data persistence, inter-process communication, and data exchange between systems in Python.

**6. How is Memory managed in Python?**

Python uses several mechanisms to efficiently allocate, use, and deallocate memory. Here’s a breakdown of the key components:

#### 1. Reference Counting

* **What It Is**: Each Python object keeps track of how many references point to it using a reference count.
* **How It Works**:
  + When a new reference to an object is created, the count increases.
  + When a reference is deleted, the count decreases.
* **Deallocation**: If the reference count reaches zero, Python immediately deallocates the object’s memory.
* **Example**:

import sys

a = [1, 2, 3]

print(sys.getrefcount(a)) # Count includes one from 'a' and one temporary reference from the function call.

#### 2. Garbage Collection

* **Why It’s Needed**: Reference counting cannot handle cycles (e.g., objects that reference each other), which can lead to memory not being freed.
* **How It Works**:
  + **Generational Garbage Collection**: Objects are divided into three generations:
    - **Generation 0**: Newly created objects.
    - **Generation 1**: Objects that survived one collection cycle.
    - **Generation 2**: Long-lived objects.
  + **Collection Triggers**: Garbage collection runs when the number of allocations exceeds a certain threshold.
* **Example**:

import gc

gc.collect() # Manually triggers garbage collection.

#### 3. Memory Allocation

* **Memory Management**: Python uses a built-in memory manager for object allocation.
* **Specialized Allocator**: For small objects (less than 512 bytes), Python uses a specific allocator called **py malloc**.
* **Object Pools**: Objects are organized into pools to minimize fragmentation and enhance allocation speed.
* **Private Heap**: All objects and data structures are stored in a private heap that is managed by the Python interpreter.

#### 4. Virtual Memory Management

* **What It Is**: Python relies on the operating system's virtual memory management for managing memory beyond its private heap.
* **How It Works**: The OS handles swapping and paging, allowing Python to use more memory than is physically available.

### Practical Implications

1. **Memory Leaks**: Be cautious of cyclic references and large data structures, as they can lead to memory not being freed even when no longer needed.
2. **Performance**: Memory allocation and garbage collection can affect performance. Profiling your application can help identify and optimize memory usage.
3. **Manual Management**: Although Python handles most memory management automatically, you can control garbage collection with the gc module and inspect reference counts using the sys module.

### Example of a Memory Management Issue

Here’s how a cyclic reference can prevent memory from being freed:

class Node:

def \_\_init\_\_(self, value):

self.value = value

self.next = None

a = Node(1)

b = Node(2)

a.next = b

b.next = a # Creates a cycle.

import gc

del a

del b

# The Node objects are still in memory due to the cycle.

# Manually triggering the garbage collector can free this memory.

gc.collect()

### Summary

Python’s memory management combines reference counting, garbage collection, specialized memory allocation, and the operating system's virtual memory management to handle memory efficiently. Understanding these components helps developers write better, more efficient Python code.

**7. How Would You Generate Random Numbers in Python?**

To generate random numbers in Python, you must first import the random module.

The random() function generates a random float value between 0 & 1.

> random.random()

The randrange() function generates a random number within a given range.

Syntax: randrange(beginning, end, step)

Example - > random.randrange(1,10,2)

**8. Are Arguments in Python Passed by Value or by Reference?**

* In Python, arguments are passed by **object reference.** This means that when you pass an argument to a function, you are passing a reference to the object, not a copy of the object itself. However, the behavior may appear to be passing by value or by reference depending on the type of object being passed and how it is mutated within the function.
* means that when you pass an argument to a function, what gets passed is a reference to the actual object, not a copy of the object. This has important implications for how the function can interact with the passed object.

**References vs. Copies:**

* Reference: A reference points to the original object in memory. If you pass a reference to a function, any changes made to the object through this reference will affect the original object.
* Copy: A copy creates a new, separate instance of the object. If you pass a copy to a function, changes made to the copy do not affect the original object.

**Behaviour with Mutable and Immutable Objects:**

* Immutable Objects: Examples include integers, floats, strings, and tuples. Even though you pass a reference, you can't change the original object because these objects don't support item assignment or modification.
* Mutable Objects: Examples include lists, dictionaries, and sets. When you pass a reference to a mutable object, you can modify the object, and these changes will be visible outside the function.

Let's break down the concept:

**Immutable Objects (e.g., integers, strings, tuples):**

* When you pass an immutable object as an argument to a function, you are passing a reference to the object.
* However, since immutable objects cannot be modified in place, any operations that appear to modify the object will actually create a new object.
* Therefore, changes made to immutable objects within a function do not affect the original object outside the function.
* def modify\_immutable(x):
* x += 1 # This creates a new integer object
* print("Inside function:", x)
* num = 10
* modify\_immutable(num)
* print("Outside function:", num) # Output: Outside function: 10

**Mutable Objects (e.g., lists, dictionaries, sets):**

* When you pass a mutable object as an argument to a function, you are still passing a reference to the object.
* Mutable objects can be modified in place, so changes made to a mutable object within a function will affect the original object outside the function.
* def modify\_mutable(lst):
* lst.append(4) # This modifies the original list
* print("Inside function:", lst)
* my\_list = [1, 2, 3]
* modify\_mutable(my\_list)
* print("Outside function:", my\_list) # Output: Outside function: [1, 2, 3, 4]

In summary, in Python, arguments are passed by object reference. For immutable objects, changes made within a function do not affect the original object, while for mutable objects, changes made within a function do affect the original object. This behavior is consistent with the idea of passing by object reference.

**9. What Does the // Operator Do?**

In Python, the / operator performs division and returns the quotient in the float.

For example: 5 / 2 returns 2.

The // operator, on the other hand, returns the quotient in integer.

For example: 5 // 2 returns 2

**10. What Does the ‘is’ Operator Do?**

The ‘is’ operator compares the id of the two objects.

list1=[1,2,3]

list2=[1,2,3]

list3=list1

list1 == list2 # True

list1 is list2 # False

list1 is list3 # True

**11. What Is the Purpose of the Pass Statement?**

The pass statement is used when there's a syntactic but not an operational requirement. For example - The program below prints a string ignoring the spaces.

* var="Simpli learn"
* for i in var:
* if i==" ":
* pass
* else:
* print(i,end="")

Here, the pass statement refers to ‘no action required.’

**12. How Will You Check If All the Characters in a String Are Alphanumeric?**

Python has an inbuilt method isalnum() which returns true if all characters in the string are alphanumeric.

Example -

* >> "abcd123".isalnum()
* Output: True
* >>”abcd@123#”.isalnum()
* Output: False
* Another way is to use regex as shown.
* >>import re
* >>bool(re.match(‘[A-Za-z0-9]+$','abcd123’))
* Output: True
* >> bool(re.match(‘[A-Za-z0-9]+$','abcd@123’))
* Output: False

**13. How Will You Merge Elements in a Sequence?**

There are three types of sequences in Python:

* Lists
* Tuples
* Strings

Example of Lists -

* >>l1=[1,2,3]
* >>l2=[4,5,6]
* >>l1+l2
* Output: [1,2,3,4,5,6]

Example of Tuples -

* >>t1=(1,2,3)
* >>t2=(4,5,6)
* >>t1+t2
* Output: (1,2,3,4,5,6)

Example of String -

* >>s1=“Simpli”
* >>s2=“learn”
* >>s1+s2
* Output: ‘Simplilearn’

**14. How Would You Remove All Leading Whitespace in a String?**

Python provides the inbuilt function lstrip() to remove all leading spaces from a string.

* “ Python”.lstrip
* Output: Python

### 15. How Would You Replace All Occurrences of a Substring with a New String?

### You can replace all occurrences of a substring with a new string in Python using the replace() method for strings or regular expressions (re module). Here are both approaches:

### Using str.replace() method:

### original\_string = "Hello, World! Hello, Universe!"

### old\_substring = "Hello"

### new\_substring = "Hi"

### # Replace all occurrences of old\_substring with new\_substring

### new\_string = original\_string.replace(old\_substring, new\_substring)

### print(new\_string) # Output: Hi, World! Hi, Universe!

### Using regular expressions (re.sub() function):

### import re

### original\_string = "Hello, World! Hello, Universe!"

### old\_substring = "Hello"

### new\_substring = "Hi"

### # Replace all occurrences of old\_substring with new\_substring using regular expression

### new\_string = re.sub(old\_substring, new\_substring, original\_string)

### print(new\_string) # Output: Hi, World! Hi, Universe!

### Both approaches will replace all occurrences of the old\_substring with the new\_substring in the original\_string. Choose the one that best fits your requirements and preferences.

**16. What Is the Difference Between Del and Remove() on Lists?**

**Del():**

* del removes all elements of a list within a given range
* Syntax: del list[start:end]
* Also when only use del without slicing, then it will remove the entire list
* >>lis=[‘a’, ‘b’, ‘c’, ‘d’]
* >>del lis[1:3]
* >>lis
* Output: [“a”,”d”]

remove():

* remove() removes the first occurrence of a particular character
* Syntax: list.remove(element)
* >>lis=[‘a’, ‘b’, ‘b’, ‘d’]
* >>lis.remove(‘b’)
* >>lis
* Output: [‘a’, ‘b’, ‘d’]

Note that in the range 1:3, the elements are counted up to 2 and not 3.

### 17. How Do You Display the Contents of a Text File in Reverse Order?

You can display the contents of a text file in reverse order using the following steps:

* Open the file using the open() function
* Store the contents of the file into a list
* Reverse the contents of the list
* Run a[for loop](https://www.simplilearn.com/tutorials/python-tutorial/python-for-loop) to iterate through the list

18. Differentiate Between append() and extend().

**append():**

* append() adds an element to the end of the list
* Example -
* >>lst=[1,2,3]
* >>lst.append(4)
* >>lst
* Output:[1,2,3,4]

**extend():**

* extend() adds elements from an iterable to the end of the list
* ***Iterable:*  is an object which can be looped over or iterated over with the help of a for loop.**
* Example -
* >>lst=[1,2,3]
* >>lst.extend([4,5,6])
* >>lst
* Output:[1,2,3,4,5,6]

**19. What Is the Output of the below Code? Justify Your Answer.**

* def addToList(val, list=[]):
* list.append(val)
* return list
* list1 = addToList(1)
* list2 = addToList(123,[])
* list3 = addToList('a')
* print ("list1 = ", list1) # list1 = [1, 'a']
* print ("list2 = %s" % list2) # list2 = [123]
* print ("list3 = %s" % list3) # list3 = [1, 'a']

**20. What Is the Difference Between a List and a Tuple?**

**Lists** are mutable while **Tuples** are immutable.

* Example:
* >>lst = [1,2,3]
* >>lst[2] = 4
* >>lst
* Output:[1,2,4]

**Tuple :**

* >>tpl = (1,2,3)
* >>tpl[2] = 4
* >>tpl
* Output: TypeError: 'tuple' object does not support item assignment

There is an error because you can't change the tuple 1 2 3 into 1 2 4.

You have to completely reassign tuple to a new value.

**21. What Is Docstring in Python?**

Docstrings are used in providing documentation to various Python modules, classes, functions, and methods.

def fun(a,b):

'''

This function adds two number

'''

print(fun.\_\_doc\_\_)

**22. How Do You Use Print() Without the Newline?**

The solution to this depends on the Python version you are using.

**Python v2**

>>print(“Hi. ”),

>>print(“How are you?”)

Output: Hi. How are you?

**Python v3**

>>print(“Hi”,end=“ ”)

>>print(“How are you?”)

Output: Hi. How are you?

**23. How Do You Use the Split() Function in Python?**

* The split() method splits a string into a list.
* You can specify the separator, default separator is any whitespace.
* Note: When maxsplit is specified, the list will contain the specified number of elements plus one.
* Ex:
* txt = "hello, my name is Peter, I am 26 years old"
* x = txt.split(", ")

**24. Is Python Object-oriented or Functional Programming?**

Python is a multi-paradigm programming language, which means it supports multiple programming paradigms, including object-oriented programming (OOP), functional programming (FP), and procedural programming.

**Object-Oriented Programming (OOP):**

* Python fully supports object-oriented programming. It allows you to define classes and objects, encapsulate data and behavior within objects, and create inheritance hierarchies.
* Key features of OOP in Python include classes, objects, inheritance, encapsulation, and polymorphism.
* You can define classes with methods and properties, create objects from those classes, and interact with them using methods and attributes.
* class Car:
* def \_\_init\_\_(self, make, model):
* self.make = make
* self.model = model
* def drive(self):
* print(f"Driving the {self.make} {self.model}")
* my\_car = Car("Toyota", "Camry")
* my\_car.drive() # Output: Driving the Toyota Camry

**Functional Programming (FP):**

* While Python is not purely a functional programming language like Haskell or Lisp, it does support functional programming concepts to some extent.
* Python allows you to use functions as first-class citizens, meaning you can pass functions as arguments to other functions, return functions from functions, and assign functions to variables.
* Functional programming features in Python include higher-order functions, lambda functions, map, filter, and reduce functions, list comprehensions, and generator expressions.
* # Example of using map and lambda for functional programming
* numbers = [1, 2, 3, 4, 5]
* squared = map(lambda x: x \*\* 2, numbers)
* print(list(squared)) # Output: [1, 4, 9, 16, 25]

**Procedural Programming:**

* Python also supports procedural programming, where you write code as a sequence of instructions to be executed step by step.
* Procedural programming in Python involves writing functions and organizing code into modules and packages.
* # Example of procedural programming
* def greet(name):
* print(f"Hello, {name}!")
* greet("Alice") # Output: Hello, Alice!

In conclusion, Python is a versatile language that supports various programming paradigms, including object-oriented programming, functional programming, and procedural programming. You can choose the paradigm that best fits your needs and preferences when designing and implementing Python code.

**25. What Are \*args and \*kwargs?**

In Python, you can define functions that accept a variable number of arguments using two techniques: \*args and \*\*kwargs.

**Using \*args (Arbitrary Positional Arguments):**

* The \*args parameter allows you to pass a variable number of positional arguments to a function.
* Inside the function, \*args collects these arguments into a tuple.

You can then iterate over this tuple or access individual arguments using indexing.

* def sum\_values(\*args):
* total = 0
* for num in args:
* total += num
* return total
* result = sum\_values(1, 2, 3, 4, 5)
* print(result) # Output: 15

**Using \*\*kwargs (Arbitrary Keyword Arguments):**

* The \*\*kwargs parameter allows you to pass a variable number of keyword arguments to a function.
* Inside the function, \*\*kwargs collects these arguments into a dictionary where the keys are the parameter names and the values are the argument values.

You can then access these arguments by their parameter names.

* def print\_values(\*\*kwargs):
* for key, value in kwargs.items():
* print(f"{key}: {value}")
* print\_values(name="Alice", age=30, city="New York")
* # Output:
* # name: Alice
* # age: 30
* # city: New York

Use Cases:

**Generic Functions:** When you want to create a generic function that can accept any number of arguments without knowing in advance how many there will be. For example, a function that calculates the sum of a variable number of values.

* def calculate\_total(\*args):
* total = 0
* for num in args:
* total += num
* return total
* result = calculate\_total(10, 20, 30, 40, 50)
* print(result) # Output: 150

**Decorator Functions:** Decorators in Python often use \*args and \*\*kwargs to allow them to wrap functions with any number of arguments.

* def debug(func):
* def wrapper(\*args, \*\*kwargs):
* print("Calling function:", func.\_\_name\_\_)
* print("Positional arguments:", args)
* print("Keyword arguments:", kwargs)
* return func(\*args, \*\*kwargs)
* return wrapper
* @debug
* def add(a, b):
* return a + b
* result = add(10, 20)
* # Output:
* # Calling function: add
* # Positional arguments: (10, 20)
* # Keyword arguments: {}
* print("Result:", result) # Output: Result: 30

Using \*args and \*\*kwargs allows for flexibility and versatility in function definitions, enabling you to write more generic and reusable code.

**26. “in Python, Functions Are First-class Objects.” What Do You Infer from This?**

It means that a function can be treated just like an object. You can assign them to variables, or pass them as arguments to other functions. You can even return them from other functions.

**27. What Is the Output Of: Print(\_\_name\_\_)? Justify Your Answer.**

When you execute print(\_\_name\_\_) directly in a Python script, the output will be "\_\_main\_\_".

* This is because \_\_name\_\_ is a built-in variable in Python that represents the name of the current module. When the Python interpreter runs a script directly, it sets the \_\_name\_\_ variable of that script to "\_\_main\_\_". This indicates that the script is being run as the main program, rather than being imported as a module into another script.

Here's an example to illustrate:

* print(\_\_name\_\_) # Output: \_\_main\_\_

However, if you import the script containing print(\_\_name\_\_) into another script/module, the output will be the name of the imported module/file, rather than "\_\_main\_\_". This is because when a script is imported, Python sets its \_\_name\_\_ variable to the name of the module, rather than "\_\_main\_\_".

* # script.py
* print(\_\_name\_\_) # Output: script
* # main.py
* import script # Output: script

This behavior allows you to differentiate between a script being run directly as the main program (\_\_name\_\_ == "\_\_main\_\_") and a script being imported as a module into another script (\_\_name\_\_ == "module\_name"). It's a useful feature for writing modular and reusable code in Python.

**28. What Is a Numpy Array?**

A NumPy array is a multidimensional grid of values, all of the same type, indexed by a tuple of nonnegative integers. It is the fundamental data structure used for numerical computing with Python's NumPy library.

Key features of NumPy arrays include:

* **Homogeneous Data Type:** Unlike Python lists, NumPy arrays contain elements of the same data type, which allows for more efficient storage and computation.
* **Multidimensional:** NumPy arrays can have any number of dimensions. A 1-dimensional array is similar to a list, a 2-dimensional array is like a matrix, and arrays with more than two dimensions are called tensors.
* **Efficient Operations:** NumPy provides a wide range of mathematical functions and operations optimized for arrays, such as element-wise operations, linear algebra operations, statistical functions, and more. These operations are implemented in C and Fortran, making them much faster than equivalent operations implemented in pure Python.
* **Broadcasting:** NumPy arrays support broadcasting, which allows you to perform arithmetic operations between arrays of different shapes. Broadcasting automatically aligns arrays with compatible shapes, eliminating the need for explicit looping.
* **Memory Efficiency:** NumPy arrays are more memory efficient compared to Python lists, especially for large datasets. They store data in a contiguous block of memory, eliminating the overhead of storing type information for each element.

Here's a basic example of creating and using a NumPy array:

* import numpy as np
* # Create a 1-dimensional array from a Python list
* arr1d = np.array([1, 2, 3, 4, 5])
* # Create a 2-dimensional array (matrix) from a list of lists
* arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
* # Accessing elements of an array
* print(arr1d[0]) # Output: 1
* print(arr2d[1, 2]) # Output: 6
* # Performing operations on arrays
* result = arr1d + 10
* print(result) # Output: [11 12 13 14 15]

NumPy arrays are widely used in scientific computing, data analysis, machine learning, and many other domains where efficient numerical computation is essential. They provide a powerful and flexible data structure for handling large datasets and performing complex numerical operations with ease.

**29. What Is the Difference Between Matrices and Arrays?**

Matrices and arrays are related concepts, but they have some differences in their definitions and usage:

**Matrices:**

* A matrix is a mathematical object composed of rows and columns of numbers, symbols, or expressions, arranged in a rectangular grid.
* Matrices are a specific type of two-dimensional array, often used in linear algebra for representing transformations, systems of linear equations, and other mathematical operations.
* Matrices have a fixed number of rows and columns, and the elements can be of any data type (e.g., integers, floats, complex numbers).
* Matrices are typically used in mathematical contexts and are often manipulated using specific operations such as matrix multiplication, addition, subtraction, and inversion.
* import numpy as np
* # Creating a matrix using NumPy
* matrix = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

**Arrays:**

* An array is a generic term used to describe a collection of elements organized in a specific way, which may include vectors, matrices, tensors, or any other multidimensional data structure.
* Arrays can have any number of dimensions, from one-dimensional (vectors) to multi-dimensional (tensors).
* Arrays can contain elements of any data type, similar to lists in Python.
* Arrays are a fundamental data structure used in programming and scientific computing for storing and manipulating data efficiently.
* import numpy as np
* # Creating a one-dimensional array using NumPy
* array1d = np.array([1, 2, 3, 4, 5])
* # Creating a two-dimensional array (matrix) using NumPy
* array2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

In summary, matrices are a specific type of two-dimensional array used in linear algebra, while arrays are a more general term referring to multidimensional data structures used in programming and scientific computing. Matrices can be represented as arrays, but not all arrays are necessarily matrices.

**30. You Have Uploaded the Dataset in Csv Format on Google Spreadsheet and Shared It Publicly. How Can You Access This in Python?**

We can use the following code:

* >>link = https://docs.google.com/spreadsheets/d/...
* >>source = StringIO.StringIO(requests.get(link).content))
* >>data = pd.read\_csv(source)

**31. You Get the Error “temp.Csv” While Trying to Read a File Using Pandas. Which of the Following Could Correct It?**

Error: Traceback (most recent call last): File "<input>", line 1, in<module>

UnicodeEncodeError: 'ascii' codec can't encode character.

Solution: pd.read\_csv(“temp.csv”, encoding=’utf-8′)

The error relates to the difference between utf-8 coding and a Unicode.

**32. How Can You Copy Objects in Python?**

The function used to copy objects in Python are:

copy.copy for shallow copy and

copy.deepcopy() for deep copy

**33. How Can You Check Whether a Pandas Dataframe Is Empty or Not?**

* The attribute df.empty is used to check whether a pandas data frame is empty or not.
* >>import pandas as pd
* >>df=pd.DataFrame({A:[]})
* >>df.empty
* Output: True

**34. What is PEP 8?**

PEP denotes Python Enhancement Proposal. It's a collection of guidelines for formatting Python code for maximum readability.

**35. Explain Python namespace.**

In Python, a namespace refers to the name that is assigned to each object.

**36. What are decorators in Python?**

A decorator is a design pattern that allows you to extend or modify the behavior of callable objects (functions or methods) without modifying their actual code. Decorators are applied using the "@" syntax.

* def my\_decorator(func):
* def wrapper():
* print("Something is happening before the function is called.")
* func()
* print("Something is happening after the function is called.")
* return wrapper
* @my\_decorator
* def say\_hello():
* print("Hello!")
* say\_hello()

**37. Differentiate between .pyc and .py.**

* The .py files are the source code files for Python.
* The bytecode of the python files are stored in .pyc files, which are created when code is imported from another source.
* The interpreter saves time by converting the source .py files to .pyc files.

**38. What is slicing in Python?**

Slicing is a technique for gaining access to specific bits of sequences such as strings, tuples, and lists.

**39. What are keywords in python?**

In Python, keywords are reserved words with a specific meaning. They are commonly used to specify the type of variables. Variable and function names cannot contain keywords.

**40. On Unix, how do you make a Python script executable?**

Script file should start with #!/usr/bin/env python

**41. What is the use of self in Python?**

Self is used to represent the class instance. In Python, you can access the class's attributes and methods with this keyword. It connects the attributes to the arguments. Self appears in a variety of contexts and is frequently mistaken for a term. Self is not a keyword in Python, unlike in C++.

**42. What are the literals in Python?**

For primitive data types, a literal in Python source code indicates a fixed value.

**43. What are the types of literals in Python?**

* For primitive data types, a literal in Python source code indicates a fixed value. Following are the 5 types of literal in Python:
* **String Literal:** A string literal is formed by assigning some text to a variable that is contained in single or double-quotes. Assign the multiline text encased in triple quotes to produce multiline literals.
* **Numeric Literal:** They may contain numeric values that are floating-point values, integers, or complex numbers.
* **Character Literal:** It is made by putting a single character in double-quotes.
* **Boolean Literal:** True or False
* **Literal Collections:** There are four types of literals such as list collections, tuple literals, set literals, dictionary literals, and set literals.

**44. What are Python modules? Name a few Python built-in modules that are often used.**

Python modules are files that contain Python code. Functions, classes, or variables can be used in this code. A Python module is a .py file that contains code that may be executed. The following are the commonly used built-in modules:

* JSON
* data time
* random
* math
* sys
* OS

**45. What is \_init\_?**

\_init\_ is a constructor or method in Python. This method is used to allocate memory when a new object is created.

**46. What is the purpose of ‘not’, ‘is’, and ‘in’ operators?**

Special functions are known as operators. They take one or more input values and output a result.

not- returns the boolean value's inverse

is- returns true when both operands are true

in- determines whether a certain element is present in a series.

**47. What are the functions help() and dir() used for in Python?**

Both help() and dir() are available from the Python interpreter and are used to provide a condensed list of built-in functions.

* The dir() function returns all properties and methods of the specified object, without the values.
* This function will return all the properties and methods, even built-in properties which are default for all object.
* help() function: The help() function displays the documentation string and also allows you to access help for modules, keywords, attributes, and other items.

**48. Why isn't all the memory de-allocated when Python exits?**

* When Python quits, some Python modules, especially those with circular references to other objects or objects referenced from global namespaces, are not necessarily freed or deallocated.
* Python would try to de-allocate/destroy all other objects on exit because it has its own efficient cleanup mechanism.
* It is difficult to de-allocate memory that has been reserved by the C library.
* Use the **atexit** module to force Python to delete certain things on deallocation.

**49. What is the best way to remove values from a Python array?**

The pop() and remove() methods can be used to remove elements from an array. The difference between these two functions is that one returns the removed value while the other does not.

**50. What are Python libraries?**

A Python library is a group of Python packages. Numpy, Pandas, Matplotlib, Scikit-learn, and many other Python libraries are widely used.

**51. What is pandas dataframe?**

A dataframe is a 2D changeable and tabular structure for representing data with rows and columns labelled.

**52. Explain monkey patching in Python.**

* Monkey patching is a technique used in programming to dynamically modify or extend the behavior of code at runtime.
* It involves altering or replacing existing code or objects without modifying their original source code.

Explanation:

* Dynamic Modification: Monkey patching allows you to modify classes, functions, or modules dynamically during runtime.
* Flexibility: It provides flexibility in extending or modifying the behavior of existing code without changing its original implementation.
* Use Cases: Monkey patching can be useful for debugging, testing, or adding new features to existing code without having to modify its source code directly.

Example:

* Let's say you have a module named math\_operations.py with a function add:
* # math\_operations.py
* def add(a, b):
* return a + b

Now, you want to extend the functionality of the add function by adding support for handling strings as inputs. Instead of modifying the original math\_operations.py file, you can use monkey patching to achieve this:

* # main.py
* import math\_operations
* # Define a new function to handle string inputs
* def add\_string\_support(a, b):
* if isinstance(a, str) or isinstance(b, str):
* return str(a) + str(b)
* else:
* return math\_operations.add(a, b)

# Monkey patch the add function with the new implementation

math\_operations.add = add\_string\_support

# Test the modified add function

print(math\_operations.add(2, 3)) # Output: 5

print(math\_operations.add('hello', 'world')) # Output: helloworld

In this example, we dynamically replace the original add function from math\_operations.py with the add\_string\_support function defined in the main.py file. This allows us to add support for handling string inputs without modifying the original source code.

**53. What is inheritance in Python?**

Inheritance allows one class to gain all of another class's members (for example, attributes and methods). Inheritance allows for code reuse, making it easier to develop and maintain applications.

**54. What are the different types of inheritance in Python?**

The following are the various types of inheritance in Python:

* Single inheritance: The members of a single super class are acquired by a derived class.
* Multiple inheritance: More than one base class is inherited by a derived class.
* Muti-level inheritance: D1 is a derived class inherited from base1 while D2 is inherited from base2.
* Hierarchical Inheritance: You can inherit any number of child classes from a single base class.

**55. Is multiple inheritance possible in Python?**

A class can be inherited from multiple parent classes, which is known as multiple inheritance. In contrast to Java, Python allows multiple inheritance.

**56. What is the return value of trunc()?**

# int

# The math.trunc() method simply remove the decimals.

**57. Highest precedence Operators?**

The operators you listed, ordered by highest precedence, are as follows:

* \*\* (Exponentiation)
* \*, /, % (Multiplication, Division, Modulus)
* +, - (Addition, Subtraction)
* <<, >> (Bitwise Shift Left, Bitwise Shift Right)
* | (Bitwise OR)

**Random NOTES (Not a part of any questions) :**

* 'str' object does not support item assignment.
* Print(type(int)) o/p: <class 'type'>
* Stub a simple but incomplete version of a function.
* i = 0
* while i < 3:
* print i
* i += 1
* else:
* print 0
* The else part is executed when the condition in the while statement is false.
* a = {i: i \* i **for** i **in** range(6)}
* print (a)
* A hash function takes a message of arbitrary length and generates a fixed length code.
* A hash function may give the same hash value for distinct messages.
* D = dict()
* **for** x **in** enumerate(range(2)):
* D[x[0]] = x[1]
* D[x[1]+7] = x[0]
* print(D)
* enumerate() will return a tuple, the loop will have x = (0, 0), (1, 1). Thus D[0] = 0, D[1] = 1, D[0 + 7] = D[7] = 0 and D[1 + 7] = D[8] = 1. Note: Dictionary is unordered, so the sequence of the key-value pair may differ in each output.

**58. Python dump vs dumps and load vs loads?**

**json.dump:**

* This function serializes a Python object and writes it directly to a file.
* Usage: json.dump(obj, fp, \*, skipkeys=False, ensure\_ascii=True, check\_circular=True, allow\_nan=True, cls=None, indent=None, separators=None, default=None, sort\_keys=False, \*\*kw)
* Parameters:
* obj: The Python object to be serialized.
* fp: A file-like object with a write method where the JSON data will be written.
* Other optional parameters for fine-tuning the serialization process.
* Example:
* python
* Copy code
* import json
* data = {'name': 'John', 'age': 30}
* with open('data.json', 'w') as file:
* json.dump(data, file)

**json.dumps:**

* This function serializes a Python object and returns the resulting JSON string.
* Usage: json.dumps(obj, \*, skipkeys=False, ensure\_ascii=True, check\_circular=True, allow\_nan=True, cls=None, indent=None, separators=None, default=None, sort\_keys=False, \*\*kw)
* Parameters:
* obj: The Python object to be serialized.
* Other optional parameters for fine-tuning the serialization process.
* Example:
* python
* Copy code
* import json
* data = {'name': 'John', 'age': 30}
* json\_string = json.dumps(data)
* print(json\_string)

In summary, use json.dump when you want to write JSON data directly to a file and json.dumps when you want to get a JSON string representation of the Python object.

**json.load:**

* This function reads JSON data from a file-like object and deserializes it into a Python object.
* Usage: json.load(fp, \*, cls=None, object\_hook=None, parse\_float=None, parse\_int=None, parse\_constant=None, object\_pairs\_hook=None, \*\*kw)
* Parameters:
* fp: A file-like object containing the JSON data to be read.
* Other optional parameters for fine-tuning the deserialization process.
* Example:
* python
* Copy code
* import json
* with open('data.json', 'r') as file:
* data = json.load(file)
* print(data)

**json.loads:**

* This function deserializes a JSON string into a Python object.
* Usage: json.loads(s, \*, cls=None, object\_hook=None, parse\_float=None, parse\_int=None, parse\_constant=None, object\_pairs\_hook=None, \*\*kw)
* Parameters:
* s: A string containing the JSON data to be deserialized.
* Other optional parameters for fine-tuning the deserialization process.
* Example:
* python
* Copy code
* import json
* json\_string = '{"name": "John", "age": 30}'
* data = json.loads(json\_string)
* print(data)

In summary, use json.load when you want to read and deserialize JSON data directly from a file, and use json.loads when you want to deserialize JSON data from a string.

**59. How do you manage packages in Python?**

Python packages are managed using tools like pip. You can install packages using:

* pip full form: Preferred Installer Program
* pip install package\_name

**60. What are generators in Python?**

* Generators in Python are special functions that can pause and resume their execution, allowing you to iterate over a sequence of values without having to generate and store them all at once in memory.
* They produce values using the "yield" keyword instead of "return", enabling them to generate values lazily as they're needed.
* This makes generators memory efficient and suitable for handling large datasets or infinite sequences.
* You can think of them as functions that produce a series of results over time rather than all at once.

**61. What are iterator?**

* An iterator in Python is an object that represents a stream of data.
* It implements the iterator protocol, which **consists of two methods:**
* **\_\_iter\_\_():** This method returns the iterator object itself. It's called to initialize the iterator.
* **\_\_next\_\_():** This method returns the next item in the stream of data. When there are no more items to return, it raises the StopIteration exception.

Iterators are used to traverse sequences of elements lazily, which means they produce the next value only when requested.

This makes them memory efficient and suitable for working with large datasets or infinite sequences.

* You can create an iterator from an iterable object using the iter() function.
* For example:
* my\_list = [1, 2, 3, 4, 5]
* my\_iter = iter(my\_list)
* print(next(my\_iter)) # Output: 1
* print(next(my\_iter)) # Output: 2
* print(next(my\_iter)) # Output: 3

In this example, my\_iter is an iterator created from the list my\_list.

Each call to next(my\_iter) retrieves the next item from the list.

**62. What are iterable?**

* An **iterable** in Python is any object that can be iterated over, meaning it can be used in a loop such as a **for** loop.
* Essentially, an iterable is any object that has an **\_\_iter\_\_()** method, which returns an iterator. Iterables include data types like lists, tuples, sets, dictionaries, strings, and more.
* my\_list = [1, 2, 3, 4, 5] # List is an iterable
* my\_tuple = (6, 7, 8, 9, 10) # Tuple is an iterable
* my\_string = "hello" # String is an iterable
* for item in my\_list:
* print(item) # Output: 1 2 3 4 5
* for item in my\_tuple:
* print(item) # Output: 6 7 8 9 10
* for char in my\_string:
* print(char) # Output: h e l l o

In each case, the for loop iterates over the elements of the iterable (list, tuple, or string), accessing each item one by one.

Additionally, any object that supports the iterator protocol is considered an iterable. This includes generators, which are a special type of iterable that can be iterated over only once.

**63. Explain iterator and iterable with for loop?**

* In Python, the for loop itself is not an iterator or iterable.
* Instead, it's a control flow statement used to iterate over iterables.
* Here's how it works:
* When you use a for loop, Python internally creates an iterator object from the iterable you're looping over using the iter() function.
* It then repeatedly calls the next() function on this iterator object to retrieve each item from the iterable.
* The loop continues until the iterator raises a StopIteration exception, indicating that there are no more items to iterate over.

So, while the for loop is not an iterator or iterable itself, it utilizes iterators to loop over iterables.

**64. Explain Threads?**

* In programming, a thread refers to the smallest sequence of programmed instructions that can be managed independently by a scheduler.
* Threads are lightweight processes within a program. They enable concurrent execution, allowing multiple parts of a program to run concurrently, thereby improving performance and responsiveness.
* Threads share the same memory space within a process, which means they can access shared data directly. This makes them efficient for communication between different parts of a program. However, it also introduces complexities, such as the need for synchronization to ensure data integrity and avoid race conditions.
* Threads are commonly used in tasks that can be parallelized, such as handling multiple user requests in a web server, performing background tasks while keeping a user interface responsive, or optimizing computations on multi-core processors.
* **Example:**
* Sure! Think of a thread like a worker in a factory. The factory is your program, and each worker (thread) can do different tasks at the same time. They share the same workspace and tools (memory), so they can help each other out and work together efficiently. But sometimes, they need to be careful not to bump into each other or mess up each other's work (synchronization). Overall, threads help your program do multiple things simultaneously, making it faster and more responsive.

**65. What is a Proxy?**

A proxy is a server that acts as an intermediary between your device (such as a computer or smartphone) and the internet. When you connect to the internet through a proxy server, your device sends requests to the proxy server, which then forwards those requests to the target websites or servers on your behalf. Similarly, the proxy server receives responses from the websites and sends them back to your device.

Here's a breakdown of the key components and functions of a proxy:

* Client: This refers to your device, whether it's a computer, smartphone, or any other device connected to the internet.
* Proxy Server: The proxy server sits between your device and the internet. It receives requests from your device and forwards them to the appropriate destinations (websites, servers, etc.). Similarly, it receives responses from those destinations and sends them back to your device.
* Destination Server: This is the final destination of your internet request, such as a website server. The proxy server interacts with destination servers on your behalf.

**How Proxies Work:**

When you configure your device to use a proxy server, all internet traffic from that device is routed through the proxy. Here's a step-by-step overview of how the process works:

* Request Initiation: When you try to access a website or service, your device generates a request.
* Proxy Configuration: Instead of sending the request directly to the destination server, your device sends it to the proxy server.
* Proxy Routing: The proxy server receives the request and processes it. It then forwards the request to the appropriate destination server.
* Destination Server Response: The destination server processes the request and generates a response.
* Response Forwarding: The proxy server receives the response from the destination server and forwards it back to your device.
* Response Delivery: Finally, your device receives the response from the proxy server and displays the content or performs the requested action.

**Functions of Proxies:**

* Anonymity: Proxies can mask your IP address, making it appear as though requests are originating from the proxy server rather than your device. This can enhance privacy and security by hiding your device's identity.
* Content Filtering: Proxies can filter internet traffic based on predefined rules, blocking access to certain websites, content categories, or specific URLs. This is commonly used in corporate environments to enforce internet usage policies.
* Caching: Proxies can cache frequently accessed web content, storing copies of web pages and resources locally. This can improve performance and reduce bandwidth usage by serving cached content instead of fetching it from the original source each time.
* Access Control: Proxies can control access to the internet by requiring authentication or implementing access control lists (ACLs). This allows organizations to restrict internet access to authorized users or groups.
* Load Balancing: Proxies can distribute incoming traffic across multiple servers to optimize performance and ensure high availability. This is commonly used in large-scale web applications and services to handle high traffic loads.

Overall, proxies serve as intermediaries between devices and the internet, providing various functions such as anonymity, content filtering, caching, access control, and load balancing. They play a crucial role in network management, security, and performance optimization in both personal and enterprise environments.

**66. What is a Proxy Server?**

A proxy server is an intermediary server that separates end-users from the websites they browse. When you connect to a website through a proxy server, your request is processed by the proxy server on your behalf. The proxy server then forwards the request to the target website, receives the response, and sends it back to you.

**How Proxies Work:**

* Request Forwarding: When you make a request to access a website, it's sent to the proxy server instead of directly to the target website.
* Anonymity: Proxies can hide your IP address, masking your identity and location from the websites you visit. This is particularly useful for privacy and security purposes
* Content Filtering: Proxy servers can filter requests based on predefined rules. For example, in corporate environments, proxies may block access to certain websites or content categories.
* Caching: Proxies can cache frequently accessed web content, speeding up subsequent requests for the same content.
* Load Balancing: Proxies can distribute incoming requests across multiple servers to optimize performance and ensure high availability.

**Types of Proxies:**

* HTTP Proxies: These proxies are primarily used for HTTP traffic. They can handle web browsing, web scraping, and other HTTP-based interactions.
* HTTPS Proxies: Similar to HTTP proxies, but they also support HTTPS (SSL-encrypted) traffic. This is essential for secure communication with websites that use HTTPS encryption.
* SOCKS Proxies: SOCKS proxies operate at a lower level than HTTP/HTTPS proxies and can handle various types of traffic, including TCP and UDP. They are more versatile but may be slower than HTTP proxies.
* Residential Proxies: These proxies use IP addresses assigned by ISPs to residential users, making them appear more like real users. They are suitable for tasks requiring high anonymity and are often used for web scraping and sneaker botting.
* Datacenter Proxies: Datacenter proxies are hosted on servers in data centers. They offer high speed and reliability but may be more easily detected by websites compared to residential proxies.

**Use Cases for Proxies:**

* Web Scraping: Proxies allow you to scrape data from websites without being blocked or detected. Rotating proxies can help you avoid rate limits and IP bans.
* Ad Verification: Proxies enable advertisers to verify the placement and performance of their ads across different websites and regions
* SEO Monitoring: Proxies help SEO professionals monitor search engine rankings and analyze search engine results pages (SERPs) from different locations
* Market Research: Proxies allow businesses to gather market intelligence by monitoring competitors' websites, pricing, and product listings.
* Content Localization: Proxies enable localization testing by accessing websites from different geographical locations to ensure content is displayed correctly.

**Considerations:**

* Reliability: Choose proxies from reputable providers to ensure reliability and uptime.
* Security: Be cautious when using free proxies, as they may log your activity or inject ads into web pages.
* Legality: Ensure your use of proxies complies with applicable laws and regulations, especially when accessing restricted or sensitive content.
* Performance: Test the performance of proxies, including speed and latency, to ensure they meet your requirements.

In summary, proxies offer various benefits for internet-related tasks, including anonymity, content filtering, and performance optimization. By understanding how proxies work and choosing the right type for your use case, you can enhance your online activities while maintaining privacy and security.

**67. What is an IP?**

* An IP (Internet Protocol) address is a uniquely-identifying string of numbers assigned to each Internet-connected device or any device connected to a network. The address identifies and allows these devices to communicate with each other, either on an internal or external computer network.
* Any device that transmits or receives network traffic gets an IP. 192.168.10.1 is an IP address example; any device with that IP can communicate with other devices across the same network. All governing devices, whether DHCP server, router, or Internet service provider, all use some sort of IP address management (IPAM) to prevent the assignment of the same Internet Protocol address to multiple devices. When there exists two devices with the same IP on the same network, an IP conflict prevents data from correct transmission and reception on these devices.
* Your Internet activity then goes through your Internet service provider and routes back to you via the IP address assigned to you. Your address isn't permanent, though; it can change based on your connection and the network you choose to connect to. It works, theoretically, as a digital version of your home address. It’s like how you need a home address to receive mail, and the sender must have your correct home number and zip code; you need an Internet Protocol address to connect to the Internet, and any device trying to connect to yours needs the proper IP for a successful connection.
* An IP can indicate which Internet service provider you use and your Internet speed. It also reveals information about your general location. It could indicate your country, state, city, or postal region. However, you can’t find someone’s exact location from their IP. The information is intended for other devices and networks to identify your device, not for others to track you down.

**68. Explain the term interpreted programming language.**  
An interpreter is a computer program that translates and executes instructions written in a programming language. It reads source code written in a high-level programming language and translates it into machine code or intermediate code that the computer's processor can understand and execute. Unlike compilers, which translate the entire program before execution, interpreters translate code line by line and execute it immediately. This allows for rapid development and easier debugging, as changes to the code can be tested without the need for recompilation. Interpreters are commonly used in languages like Python, JavaScript, and Ruby.

**69: What is your understanding of PEP8?**

PEP is an acronym for Python Enhancement Proposal. PEP defines rules for Python code formatting to achieve maximum readability. An important question in Python Developer Job Interviews, this question might be framed in different ways to confuse you.

**70: Describe the use of Python modules. List some commonly used Python modules.**

Python modules are Python code files that can contain either functions, classes or variables. A Python module is executable code stored in a .py file.

Commonly used built-in modules in Python include:

* os
* sys
* random
* math
* date time
* JSON

**71: What are the types of inheritance supported in Python**

Inheritance allows the programmer to reuse the code and makes it simpler to create and manage any program. It provides one class to access all the members (say attributes and methods) of another class, so that the same methods, variables, etc. do not have to be written again. The class from which the other class will be inheriting methods and attributes is called the super class or base class and the class that is inheriting is called a child class or derived class.

There are many different types of inheritance in Python:

* Single Inheritance: When a derived class gains the members from a single base class.
* Multi level inheritance: When derived class a1 inherits members from super class s1, and a2 inherits members from super class s2.
* Hierarchical inheritance: When many child classes inherit members from one super class.
* Multiple inheritance: When a derived class inherits members from more than one super classes.

**72: What is \_\_init\_\_ method?**

The \_\_init\_\_ method in Python is a special method used for initializing newly created objects. It is also known as the constructor method. When you create a new instance of a class, the \_\_init\_\_ method is automatically called.

Here's a brief explanation of the \_\_init\_\_ method:

* Purpose: The primary purpose of the \_\_init\_\_ method is to initialize the attributes of the newly created object. It allows you to specify initial values for the object's attributes.
* Syntax: The \_\_init\_\_ method is defined within a class like any other method, but its name is specifically \_\_init\_\_. It takes at least one argument, self, which refers to the instance being created. Additional arguments can be specified to initialize other attributes.
* Automatic Invocation: When you create a new instance of a class using the class name followed by parentheses (e.g., obj = MyClass()), Python automatically calls the \_\_init\_\_ method of that class to initialize the object.
* Initialization Logic: Inside the \_\_init\_\_ method, you can initialize instance variables (attributes) using the self parameter. These variables represent the state of the object and can be accessed throughout the class.

**73: What is self in Python?**

In Python, self is a conventionally used name for the first parameter of instance methods in a class. It represents the instance of the class, allowing you to access and modify attributes and methods of that instance within the class.

Here's what you need to know about self:

* **Instance Method Parameter**: In Python, when you define methods within a class, you typically include self as the first parameter in the method definition. However, when you call the method, you don't explicitly pass an argument for self – Python handles this automatically.
* **Instance Reference:** Inside instance methods, self refers to the instance of the class itself. It allows you to access the attributes and methods of the specific instance on which the method is called.
* **Naming Convention:** While self is not a reserved keyword in Python, it is a widely adopted convention to use the name self for the first parameter in instance methods. Using any other name is technically allowed, but it's considered best practice to stick with self to improve code readability and maintainability.
* **Method Invocation:** When you call an instance method, you do so using the dot notation (instance.method()). Python automatically passes the instance (i.e., self) as the first argument to the method, allowing you to access instance-specific attributes and methods.

**74: List all errors that occur in python.**

* Name Error
* Syntax Error
* Import error
* Index Error
* Type Error
* Value Error

**75: What is context switching in python?**

In Python, "context switching" typically refers to the process of switching between different execution contexts, such as switching between threads or coroutines. This is particularly relevant in asynchronous programming, where tasks may be paused and resumed asynchronously without blocking the entire program.

**Thread Context Switching:**

* In Python, threading is managed by the Python Global Interpreter Lock (GIL), which allows only one thread to execute Python bytecode at a time. As a result, true parallelism is not achieved with threads alone in CPython.
* However, context switching between threads can still occur when a thread voluntarily yields control or when it's preempted by the operating system due to time slicing.
* Context switching between threads can be managed explicitly using synchronization primitives like locks, semaphores, and condition variables to coordinate access to shared resources.

**Coroutine Context Switching:**

* Python's asynchronous programming model, based on coroutines and the asyncio library, involves frequent context switching between coroutines.
* Coroutines are lightweight, cooperative multitasking units that can voluntarily yield control back to the event loop using the await keyword.
* When a coroutine awaits on an asynchronous operation (e.g., I/O operation), it suspends its execution and allows other coroutines to run until the operation completes.

Context switching between coroutines is managed by the event loop, which schedules and executes coroutines based on their readiness to run.

**Generator Context Switching:**

* Python generators, created using functions with yield statements, also involve context switching.
* When a generator function is called, it returns a generator iterator. Each time the next() function is called on the iterator, the generator function executes until it encounters a yield statement, at which point it yields control back to the caller.
* The caller can then resume the generator by calling next() again, causing the generator to resume execution from the point of the last yield statement.
* This process involves implicit context switching between the generator and its caller.

Overall, context switching in Python occurs in various contexts such as threading, asynchronous programming with coroutines, and generator-based iterators, allowing for efficient multitasking and concurrency in Python programs.

**76: What is the use of context managers in Python and what is context manager?**

Context managers in Python are objects that are used to manage resources and define setup and teardown actions that should occur around a block of code. They are typically used with the with statement to ensure that resources are properly acquired and released, even in the presence of exceptions or other errors.

A context manager in Python is any object that implements the context management protocol, which involves defining \_\_enter\_\_() and \_\_exit\_\_() methods. These methods define the behavior that occurs when entering and exiting the context managed by the object.

Here's what you need to know about the use and definition of context managers:

* Resource Management: Context managers are commonly used for resource management tasks such as opening and closing files, acquiring and releasing locks, establishing and closing database connections, etc. They ensure that resources are properly cleaned up and released, even if an error occurs during execution.
* With Statement: The with statement is used to invoke a context manager. It ensures that the \_\_enter\_\_() method of the context manager is called before entering the block of code and that the \_\_exit\_\_() method is called afterward, regardless of whether an exception occurs within the block.
* Automatic Resource Cleanup: Context managers provide a convenient way to ensure that resources are automatically cleaned up when they are no longer needed, without requiring explicit cleanup code or relying solely on the programmer to remember to release resources.
* Custom Context Managers: You can create custom context managers by defining classes that implement the context management protocol by providing \_\_enter\_\_() and \_\_exit\_\_() methods. Alternatively, you can use the contextlib module to create context managers using generator functions or by using the @contextmanager decorator.

Example of using a context manager with file handling:

# Using a context manager to open and automatically close a file

with open("example.txt", "w") as f:

f.write("Hello, world!")

In this example, the open() function returns a file object that acts as a context manager. When the with statement is executed, it automatically calls the \_\_enter\_\_() method of the file object, which opens the file. After the block of code inside the with state.

**77. Explain the process of compilation and linking in python.**

In Python, the process of compilation and linking is different from compiled languages like C or C++. Python is an interpreted language, which means that Python code is executed directly by the Python interpreter without the need for explicit compilation into machine code. However, there are still steps involved in the execution of Python code:

**Source Code:**

* Python source code is written in .py files. These files contain text-based code written in the Python programming language.

**Parsing and Compilation:**

* When you run a Python script, the Python interpreter first parses the source code to create an abstract syntax tree (AST). This AST represents the structure of the code and is used for compilation.
* Python then compiles the AST into bytecode, which is a low-level representation of the Python code. This bytecode is not machine code but rather an intermediate representation that is platform-independent.

**Execution:**

* The Python interpreter executes the bytecode generated from the compilation step. It uses a virtual machine (known as the Python Virtual Machine or PVM) to interpret and execute the bytecode instructions.
* During execution, the interpreter performs various tasks such as memory management, garbage collection, and dynamic typing.

There is no explicit linking step in Python as there is in compiled languages like C or C++. Python does not produce standalone executables that need to be linked with external libraries. Instead, Python code is executed directly by the interpreter, which dynamically loads and links any necessary modules or libraries at runtime.

**Summary:**

* Compilation involves parsing the source code, generating an abstract syntax tree, and compiling it into bytecode.
* There is no separate linking step, as the interpreter dynamically loads and links modules and libraries at runtime.
* Execution occurs directly by the Python interpreter, which interprets and executes the bytecode instructions using the Python Virtual Machine.ment completes, the \_\_exit\_\_() method is called, which automatically closes the file, ensuring proper cleanup.

**78. What is OOP and its 4 pillers?**

Object-Oriented Programming (OOP) is a programming paradigm that focuses on organizing code into objects, which are instances of classes representing real-world entities. Python fully supports OOP principles and provides features such as classes, inheritance, encapsulation, and polymorphism. Here's an explanation of various OOP-related topics in Python:

**Classes and Objects:**

* Classes: Classes are blueprints for creating objects. They define the attributes (data) and methods (functions) that all objects of that class will have.
* Objects: Objects are instances of classes. They represent specific instances of real-world entities and have their own unique state (attribute values) and behavior (methods).

**Attributes and Methods:**

* Attributes: Attributes are variables that store data associated with objects. They define the state of an object. Attributes can be accessed and modified using dot notation (object.attribute).
* Methods: Methods are functions defined within a class that operate on objects of that class. They define the behavior of objects and can access and modify object attributes.

**Inheritance:**

* Inheritance allows a class (subclass) to inherit attributes and methods from another class (superclass). Subclasses can extend or override the behavior of the superclass.
* Python supports single inheritance, where a class can inherit from one superclass, and multiple inheritance, where a class can inherit from multiple superclasses.

**Types of Inheritance:**

**Single Inheritance:**

* In single inheritance, a class (subclass) can inherit attributes and methods from only one superclass.
* Syntax:
* class Superclass:
* # Superclass definition
* class Subclass(Superclass):
* # Subclass inherits from Superclass
* # Subclass-specific attributes and methods
* Example:
* class Animal:
* def sound(self):
* print("Animal makes a sound")
* class Dog(Animal):
* def sound(self):
* print("Dog barks")
* dog = Dog()
* dog.sound() # Output: "Dog barks"

In this example, the Dog class inherits the sound() method from the Animal superclass. The sound() method in the Dog class overrides the method in the Animal class.

**Multiple Inheritance:**

* Multiple inheritance allows a class to inherit attributes and methods from more than one superclass.
* Syntax:
* class Superclass1:
* # Superclass1 definition
* class Superclass2:
* # Superclass2 definition
* class Subclass(Superclass1, Superclass2):
* # Subclass inherits from Superclass1 and Superclass2
* # Subclass-specific attributes and methods
* Example:
* class Bird:
* def fly(self):
* print("Bird flies")
* class Mammal:
* def walk(self):
* print("Mammal walks")
* class Bat(Bird, Mammal):
* pass
* bat = Bat()
* bat.fly() # Output: "Bird flies"
* bat.walk() # Output: "Mammal walks"

In this example, the Bat class inherits from both the Bird and Mammal superclasses. It inherits the fly() method from Bird and the walk() method from Mammal.

**Method Resolution Order (MRO):**

* In multiple inheritance, Python uses the C3 linearization algorithm to determine the order in which methods are resolved (searched) in the inheritance hierarchy.
* The \_\_mro\_\_ attribute of a class provides the method resolution order.
* Example:
* class A:
* pass
* class B(A):
* pass
* class C(A):
* pass
* class D(B, C):
* pass
* print(D.\_\_mro\_\_) # Output: (<class '\_\_main\_\_.D'>, <class '\_\_main\_\_.B'>, <class '\_\_main\_\_.C'>, <class '\_\_main\_\_.A'>, <class 'object'>)
* In this example, the method resolution order for class D is D -> B -> C -> A -> object.

**Super() Function:**

* The super() function allows access to methods and properties of the superclass within the subclass.
* It is commonly used to invoke superclass methods from overridden methods in the subclass.
* Example:
* class Animal:
* def sound(self):
* print("Animal makes a sound")
* class Dog(Animal):
* def sound(self):
* super().sound() # Invoke superclass method
* print("Dog barks")
* dog = Dog()
* dog.sound()
* Output:
* Animal makes a sound
* Dog barks

In this example, the Dog class invokes the sound() method of the Animal superclass using super().sound() before printing "Dog barks".

**Encapsulation:**

* Encapsulation refers to the bundling of data (attributes) and methods that operate on that data within a single unit (class). It hides the internal state of objects from external access and modification, providing data abstraction and protection.
* In Python, encapsulation is achieved using access modifiers such as public, protected, and private attributes and methods (using single or double underscores).

**Polymorphism:**

* Polymorphism allows objects of different classes to be treated as objects of a common superclass. It enables the use of a single interface to represent multiple types of objects.
* Polymorphism in Python is achieved through method overriding, where a subclass provides a specific implementation of a method defined in its superclass.

**Abstraction:**

* Abstraction involves hiding complex implementation details and exposing only essential features of an object or class. It allows users to focus on what an object does rather than how it does it.
* In Python, abstraction is achieved through classes and interfaces, providing a clear separation between interface and implementation.

Overall, OOP in Python provides a powerful and flexible way to structure and organize code, enabling developers to write modular, maintainable, and reusable software. It promotes code reusability, scalability, and easier maintenance through encapsulation, inheritance, and polymorphism.