**Q1. What is the purpose of Python's OOP?**

The purpose of Python's Object-Oriented Programming (OOP) is to provide a programming paradigm that organizes code into objects, allowing developers to model real-world entities, encapsulate data and behavior, and create reusable and maintainable code. OOP is one of the key features of Python and is widely used in various applications and libraries.

Here are some of the primary purposes of Python's OOP:

* Modularity and Reusability
* Encapsulation
* Abstraction
* Inheritance
* Polymorphism
* Organizing Code
* Collaboration

Overall, Python's OOP features empower developers to create robust, scalable, and maintainable software solutions, making it one of the most widely used programming paradigms in the Python ecosystem

**Q2. Where does an inheritance search look for an attribute?**

In Python, when an attribute or method is accessed on an object, the interpreter searches for the attribute following a specific order known as the "attribute lookup" or "attribute resolution" process. The inheritance search looks for an attribute in the following order:

* Instance Namespace
* Class Namespace
* Parent Classes
* Base Classes
* Object Class

This inheritance search process allows attributes and methods to be inherited and overridden in subclasses. If an attribute is not found in the subclass, it can be inherited from its parent classes

**Q3. How do you distinguish between a class object and an instance object?**

In Python, the difference between class object and instance object is based on their role and behavior in the object-oriented model:

**Class Object:**

* A class object is an instance of the class itself. In Python, classes are also objects, which means they can be instantiated, assigned to variables, passed as arguments to functions, and so on.
* The class object is the blueprint for creating instance objects. It defines the attributes and methods that will be shared by all instances of that class.
* Class objects are used to create new instances of the class by calling the class as if it were a function. For example, if MyClass is a class, you can create an instance object by doing my\_instance = MyClass().
* Class objects have attributes and methods that apply to the entire class and are shared among all instances. For example, class-level variables are defined in the class object and are accessible to all instances.

**Instance Object:**

* An instance object is an individual object created from a class. It represents a specific occurrence or instantiation of the class.
* When you create an instance of a class, you are essentially creating a new object with its own unique set of attributes and state, distinct from other instances of the same class.
* Instance objects have access to the attributes and methods defined in the class object, as well as the ability to store their own instance-specific data.
* Modifying attributes or invoking methods on an instance affects only that particular instance and doesn't affect other instances of the same class.

**Q4. What makes the first argument in a class’s method function special?**

In Python, the first argument in a class's method function is conventionally named self, and it represents the instance of the class itself. This first argument is special and crucial for the proper functioning of methods in Python classes. Here's why it's special:

**Instance Binding:** The self parameter is used to bind the method to the instance of the class on which the method is called. When you call an instance method on an object, Python automatically passes the instance object as the first argument (self) to the method.

**Accessing Instance Attributes and Methods:** Through self, you can access instance-specific attributes and methods within the method. It provides access to the instance's internal state and allows you to manipulate or retrieve data associated with that specific instance.

**Method Invocation:** The presence of self is essential for correctly invoking instance methods. When you call an instance method using the dot notation (e.g., instance.method()), Python implicitly passes the instance (instance) as the first argument, allowing the method to operate on the specific instance.

**Scope Resolution:** The self parameter ensures proper scope resolution in the class. It allows you to differentiate between instance variables (attributes) and class variables. Without self, Python would interpret variables as local to the method rather than referring to the instance attributes.

**Here's an example to demonstrate the use of self in a class method:**

class MyClass:

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

self.name = name

def say\_hello(self):

print(f"Hello, my name is {self.name}")

my\_instance = MyClass("John")

my\_instance.say\_hello() # Output: Hello, my name is John

**Q5. What is the purpose of the init method?**

The \_\_init\_\_ method in Python is a special method (also known as a constructor) that is automatically called when an instance of a class is created. Its purpose is to initialize the attributes of the instance and perform any setup or configuration that may be required before the object is ready for use.

Here are the main purposes of the \_\_init\_\_ method:

* **Initialization of Instance Attributes**
* **Instance Customization**
* **Setting Default Values**
* **Resource Allocation and Configuration**

**Q6. What is the process for creating a class instance?**

Creating a class instance in Python involves the following steps:

1. **Define the Class:**
2. **Instantiate the Class:**
3. **Initialize the Instance (Optional):**
4. **Use the Instance:**

**Q7. What is the process for creating a class?**

Creating a class in Python involves the following steps:

1. **Class Declaration:**
2. **Class Body:**
3. **Initializer (Optional):**
4. **Class Instantiation:**

**Q8. How would you define the superclasses of a class?**

In Python, the superclasses of a class refer to the classes from which the current class inherits attributes and methods. These superclasses are also known as parent classes or base classes. When a class is defined as a subclass, it can inherit the attributes and methods of one or more superclasses.

The superclass relationship forms the foundation of inheritance in object-oriented programming. By inheriting from a superclass, a subclass gains access to its attributes and methods, promoting code reuse and creating a hierarchical structure among related classes.

**example :**

class SubclassName(Superclass1, Superclass2, ...):

**Q9. What is the relationship between classes and modules?**

In Python, classes and modules are both fundamental building blocks used for organizing and structuring code.

They serve different purposes but can be related in several ways:

* **Modules Can Contain Classes**
* **Classes as Members of Modules**
* **Code Reusability**
* **Namespace Management:**
* **Code Organization and Separation of Concerns**
* **Importing Classes from Modules**

**Q10. How do you make instances and classes?**

To create instances and classes in Python, you follow these steps:

* **Define a Class**
* **Define an \_\_init\_\_ Method (Optional)**
* **Instantiate the Class**
* **Access Attributes and Methods**

**Q11. Where and how should be class attributes created?**

Class attributes should be created inside the class body, outside of any class methods. They are defined directly within the class scope, and they are shared by all instances of the class. Class attributes are different from instance attributes, which are unique to each instance of the class.

Class attributes are usually used to store data that is common to all instances of the class or to store constants that should remain the same for all instances. Since class attributes are shared among instances, modifying the value of a class attribute affects all instances of that class.

**Q12. Where and how are instance attributes created?**

Instance attributes are created and initialized within the \_\_init\_\_ method of a class. The \_\_init\_\_ method is a special method known as the constructor, and it is automatically called when a new instance of the class is created. It allows you to set the initial state of the instance by defining its attributes.

To create an instance attribute, you use the self keyword within the \_\_init\_\_ method, followed by the name of the attribute you want to create. The self keyword refers to the instance itself, and it is used to bind the attributes to that specific instance.

**Q13. What does the term "self" in a Python class mean?**

In Python, the term "self" is a convention used as the first parameter in the methods of a class. It refers to the instance of the class itself, and it is used to access and modify the attributes and methods of that particular instance.

When you define a class method, the first parameter must be self. Although you can technically choose any valid variable name for this parameter, it is a widely followed convention in Python to use self. Using self helps to make the code more readable and maintainable, as it is a recognized practice among Python developers.

**Q14. How does a Python class handle operator overloading?**

In Python, operator overloading allows you to define special methods in a class that determine how the built-in operators behave when applied to instances of that class. These special methods are called "magic methods" or "dunder methods" (short for "double underscore" methods) because they are surrounded by double underscores on both sides of their names.

By implementing specific magic methods in your class, you can customize the behavior of operators like `+`, `-`, `\*`, `/`, `==`, `<`, `>`, and many others.

This feature allows you to make instances of your class behave like built-in data types when used with operators.

Here are a few commonly used magic methods for operator overloading:

**1. `\_\_add\_\_(self, other)`: Overloads the `+` operator.**

**2. `\_\_sub\_\_(self, other)`: Overloads the `-` operator.**

**3. `\_\_mul\_\_(self, other)`: Overloads the `\*` operator.**

**4. `\_\_truediv\_\_(self, other)`: Overloads the `/` operator (true division).**

**5. `\_\_floordiv\_\_(self, other)`: Overloads the `//` operator (floor division).**

**6. `\_\_eq\_\_(self, other)`: Overloads the `==` operator for equality comparison.**

**7. `\_\_lt\_\_(self, other)`: Overloads the `<` operator for less than comparison.**

**8. `\_\_gt\_\_(self, other)`: Overloads the `>` operator for greater than comparison.**

**9. `\_\_str\_\_(self)`: Overloads the `str()` function and provides a string representation of the instance.**

**Q15. When do you consider allowing operator overloading of your classes?**

Allowing operator overloading in your classes can be considered in various scenarios when it enhances code readability, usability, and consistency. Here are some situations where operator overloading might be beneficial:

**1. Natural Semantics**

**2. Consistency with Built-in Types**

**3. Mathematical Operations**

**4. Custom Containers**

**5. Code Readability**

However, it's important to use operator overloading judiciously. Overusing or misusing operator overloading can make code less clear and harder to maintain, especially if the behavior differs significantly from the expectations of standard Python operators.

Before implementing operator overloading, consider whether it adds clarity and value to your class and if it aligns with the semantics of the operators you plan to overload. When in doubt, favor clarity and explicitness over cleverness.

Additionally, when defining magic methods for operator overloading, ensure that they follow Python's conventions and do not produce unexpected side effects or violate basic operator semantics. Providing well-documented behavior for overloaded operators can also improve the user experience for those interacting with your class.

**Q16. What is the most popular form of operator overloading?**

In Python, one of the most popular forms of operator overloading is overloading the addition (`+`) operator. This is because the `+` operator is widely used for various data types, and overloading it allows custom classes to support addition operations in a more intuitive and natural way.

When overloading the addition operator, you can define how instances of your class should behave when they are combined using the `+` operator. This allows you to implement custom addition logic specific to your class.

**The popularity of overloading the addition operator can be attributed to several reasons:**

1. \*\*Common Usage\*\*: The `+` operator is one of the most frequently used operators in programming, especially for numeric types and sequences like lists, strings, and tuples. By supporting addition, custom classes can interact seamlessly with Python's built-in data types and existing code.

2. \*\*Mathematical Expressions\*\*: Overloading the `+` operator is particularly useful for classes representing mathematical entities like vectors, matrices, complex numbers, and points. It allows you to express mathematical operations more naturally in code.

3. \*\*Convenience\*\*: By overloading the `+` operator, you enable a more concise and expressive syntax for adding instances of your class. Instead of calling a custom method explicitly, users can use the familiar `+` operator.

4. \*\*Consistency\*\*: By mimicking the behavior of built-in types, overloading the `+` operator can provide a consistent interface for users, making your class easier to understand and use.

**Q17. What are the two most important concepts to grasp in order to comprehend Python OOP code?**

To comprehend Python Object-Oriented Programming (OOP) code effectively, two fundamental concepts are crucial:

**1. Classes and Objects:**

Understanding the concept of classes and objects is essential for comprehending Python OOP code. A class is a blueprint or template that defines the structure and behavior of objects. It encapsulates data (attributes) and functions (methods) that operate on that data. An object, on the other hand, is an instance of a class, representing a specific entity or data with its unique state.

Key points to grasp:

* A class is defined using the `class` keyword, and objects are created by calling the class like a function.
* Class attributes are shared among all instances of the class, while instance attributes are unique to each object.
* Methods defined within a class can access and operate on the object's attributes using the `self` keyword.

**2. Inheritance and Polymorphism:**

Understanding inheritance and polymorphism is essential for comprehending how classes can relate to each other and how they can be used interchangeably. Inheritance allows a class (subclass) to inherit attributes and methods from another class (superclass), promoting code reuse and creating a hierarchical relationship.

Key points to grasp:

* Inheritance is defined by specifying the superclass in the class definition using parentheses.
* A subclass can override or extend the behavior of its superclass by redefining or adding methods.
* Polymorphism allows different classes to be treated as instances of a common superclass, allowing for more flexible and generic code.

**Q18. Describe three applications for exception processing.**

Exception processing, also known as error handling or exception handling, is a crucial aspect of programming that allows you to gracefully handle unexpected or exceptional situations in your code. Here are three common applications for exception processing:

**1. Robust Error Handling**

Exception processing is used to handle errors and exceptional conditions that may occur during the execution of a program. By using try-except blocks, you can catch and handle exceptions, preventing your program from crashing and providing meaningful feedback to users when something goes wrong. This is particularly important in production systems, where unexpected errors can occur due to various factors, such as user input, external services, or file operations.

**2. Data Validation and User Input**

Exception processing is useful for data validation, especially when dealing with user input. By catching specific exceptions, you can validate user data and prompt users to provide valid input in case of errors. This ensures that the program operates with valid data and reduces the risk of data-related issues.

**3. Resource Management**

Exception processing is employed to manage resources and ensure they are properly released, even if exceptions occur during their usage. For example, when dealing with files or database connections, you can use try-finally blocks to ensure that resources are properly closed, regardless of whether an exception is raised.

In summary, exception processing is a powerful mechanism that allows you to handle errors, validate data, and manage resources in a controlled manner. By utilizing try-except blocks, you can make your programs more robust, user-friendly, and reliable in the face of unexpected situations.

**Q19. What happens if you don't do something extra to treat an exception?**

If you don't handle an exception in your code, it will result in an uncaught exception, and your program will terminate abruptly, displaying an error traceback. This can be problematic for the user experience and may not provide any helpful information about what went wrong, making it challenging to diagnose the issue.

When an uncaught exception occurs, Python will display a traceback, which is a detailed report of the chain of function calls leading up to the exception. The traceback includes the file name, line number, and specific code responsible for raising the exception.

**Q20. What are your options for recovering from an exception in your script?**

When an exception occurs in your script, you have several options for recovering from it, depending on the nature of the exception and your desired behavior. Exception handling using try-except blocks allows you to gracefully handle errors and continue the execution of your script in a controlled manner.

Here are some common options for recovering from an exception:

**1. Error Reporting**

**2. Default Values or Recovery Logic**

**3. Retrying the Operation**

**4. Exiting Gracefully**

**5. Cleanup and Resource Management**

By using these options, you can tailor your exception handling to suit the specific requirements of your script, enabling you to recover from errors, maintain data integrity, and provide a better user experience.

**Q21. Describe two methods for triggering exceptions in your script.**

**This are two more methods for triggering exceptions in your Python script:**

**1. AssertionError:**

The `assert` statement in Python is used to assert the truth of an expression. If the expression evaluates to `False`, an `AssertionError` is raised. This can be used for sanity checks in your code to ensure that certain conditions are met.

**2. IndexError:**

The `IndexError` is raised when attempting to access an index in a sequence (like a list or a string) that is out of range or does not exist

By utilizing these methods and other built-in exceptions or custom exceptions, you can handle specific scenarios in your code effectively

**Q22. Identify two methods for specifying actions to be executed at termination time, regardless of whether or not an exception exists.**

In Python, you can specify actions to be executed at termination time, regardless of whether an exception was raised or not, by using two methods:

**1. try-finally Block:**

The try-finally block is a construct that ensures that a specific block of code (the finally block) is executed, regardless of whether an exception occurred in the try block or not.

**2. atexit Module:**

The atexit module provides a way to register functions that should be executed when the Python interpreter is about to terminate, whether it exits normally or due to an uncaught exception.

Both methods can be valuable for ensuring that essential actions, such as resource cleanup, logging, or closing files, are performed reliably, even in scenarios where exceptions might be raised.

**Q23. What is the purpose of the try statement?**

The purpose of the try statement in Python is to define a block of code where exceptional situations, such as errors or exceptions, may occur. It allows you to catch and handle these exceptions gracefully, preventing your program from crashing and providing a controlled way to respond to unexpected situations.

The try statement is part of Python's exception handling mechanism, which allows you to anticipate and manage errors that may arise during program execution. By using the try statement, you can isolate the potentially problematic code within a specific block, known as the try block.

**The basic syntax of the try statement is as follows:**

try:

# Code that may raise an exception

except SomeException as e:

# Exception handling code

Using the try statement, you can effectively handle exceptional situations, providing a more robust and user-friendly experience for your Python programs.

**Q24. What are the two most popular try statement variations?**

The two most popular variations of the try statement in Python are:

**1. try-except block**:

This is the most commonly used form of the try statement. It allows you to catch and handle specific exceptions that may occur in the try block.

In this variation, the except block is used to catch a specific exception (e.g., ValueError, ZeroDivisionError, or a custom exception) and handle it accordingly. You can have multiple except blocks to handle different types of exceptions individually.

**2. try-except-else block:**

The try-except-else variation allows you to specify a block of code to be executed if no exceptions occur in the try block. It provides a way to separate the code that may raise exceptions from the code that should run only when no exceptions are raised.

The else block is optional and is executed only if the try block completes without raising any exceptions. If an exception occurs, the except block is executed, and the else block is skipped.

The try-except block is essential for handling exceptions and preventing program crashes, while the try-except-else block allows you to execute code that should run only when no exceptions occur. Using these variations, you can build robust and error-resistant code, providing appropriate responses to various scenarios that may arise during program execution.

**Q25. What is the purpose of the raise statement?**

The purpose of the raise statement in Python is to explicitly raise an exception. It allows you to signal that an exceptional condition has occurred in your code and trigger the specified exception to be handled further up in the call stack. Essentially, the raise statement interrupts the normal flow of the program and transfers control to the nearest enclosing try-except block that can handle the raised exception.

**The basic syntax of the raise statement is as follows:**

raise SomeException("Optional error message")

**Q26. What does the assert statement do, and what other statement is it like?**

The assert statement in Python is used for debugging and testing purposes. It allows you to check a condition and immediately trigger an AssertionError if the condition evaluates to False. The assert statement helps you ensure that certain conditions hold true during program execution. It acts as a form of self-check for developers and is typically used during development and testing to catch potential errors early.

**The basic syntax of the assert statement is as follows:**

assert condition, "Optional error message"

condition: The expression that is checked for truthiness. If it evaluates to False, the assert statement raises an AssertionError.

"Optional error message": An optional error message that can be provided to provide more context about the assertion failure. This message is displayed when the assert statement raises an AssertionError.

**Q27. What is the purpose of the with/as argument, and what other statement is it like?**

The with/as statement in Python is used for resource management. It provides a convenient and concise way to ensure that resources, such as files, network connections, or database connections, are properly acquired and released when they are no longer needed. The with/as statement is often referred to as the "context management" or "context manager" statement.

**The basic syntax of the with/as statement is as follows:**

with expression as variable:

Here's how the with/as statement works:

1. The expression is an object that represents the resource you want to manage. The resource must have an associated "context manager" that defines the methods \_\_enter\_\_() and \_\_exit\_\_(). The \_\_enter\_\_() method is called when entering the context, and the \_\_exit\_\_() method is called when leaving the context.

2. The as variable part allows you to assign the result of the \_\_enter\_\_() method to a variable. This variable can be used to reference the resource within the with block.

3. The with/as statement takes care of acquiring the resource and automatically releasing it, even if an exception occurs within the with block. This ensures that resources are properly cleaned up and prevents resource leaks.

**Q28. What are \*args, \*\*kwargs?**

`\*args` and `\*\*kwargs` are special syntax used in Python function definitions to allow a variable number of arguments to be passed to a function. They are often referred to as "argument unpacking" mechanisms.

**1. args:**

The `\*args` syntax is used to pass a variable-length non-keyword argument list to a function. It allows you to pass an arbitrary number of positional arguments to the function. The name `args` is a convention, but you can use any valid variable name preceded by `\*`.

Inside the function, `\*args` is treated as a tuple containing all the positional arguments passed to the function. You can use it to define functions that can accept a variable number of arguments.

**2. kwargs:**

The `\*\*kwargs` syntax is used to pass a variable-length keyword argument dictionary to a function. It allows you to pass an arbitrary number of keyword arguments to the function. The name `kwargs` is a convention, but like with `\*args`, you can use any valid variable name preceded by `\*\*`.

Inside the function, `\*\*kwargs` is treated as a dictionary containing all the keyword arguments passed to the function. You can use it to define functions that can accept additional named arguments.

You can use \*args and \*\*kwargs together in a function definition if you want to accept both variable-length positional and keyword arguments. The order should be \*args first, followed by \*\*kwargs

**Q29. How can I pass optional or keyword parameters from one function to another?**

You can pass optional or keyword parameters from one function to another by using \*args and \*\*kwargs in the function's argument list. This allows you to forward the variable-length arguments and keyword arguments received by one function to another function without explicitly specifying each parameter.

By using \*args and \*\*kwargs, you can make your functions more flexible and reusable, allowing them to accept a variable number of arguments and keyword parameters without explicitly defining each one. This is particularly useful when you have functions that wrap or extend the behavior of other functions and need to pass the received arguments down the call chain.

**Q30. What are Lambda Functions?**

Lambda functions in Python are a way to create small, anonymous, and one-line functions without using the def keyword. They are also known as "anonymous functions" because they do not have a formal name, and they are typically used for simple, short tasks.

**The syntax for a lambda function is as follows:**

lambda arguments: expression

lambda: The keyword used to define a lambda function.

arguments: The comma-separated list of input arguments (similar to the arguments in a regular function).

expression: The single expression that the lambda function evaluates and returns.

**Q31. Explain Inheritance in Python with an example?**

Inheritance is a fundamental concept in object-oriented programming (OOP) that allows a new class (called the derived class or subclass) to inherit the characteristics (attributes and methods) of an existing class (called the base class or superclass). This enables code reuse and promotes a hierarchical relationship among classes, where the subclass can extend or modify the behavior of the superclass.

In Python, you can implement inheritance using the class definition. The derived class is created by specifying the base class in parentheses after the subclass name.

Example :

# Base class (Superclass)

class Animal:

def \_\_init\_\_(self, name, species):

self.name = name

self.species = species

def make\_sound(self):

return "Some generic sound"

def display\_info(self):

return f"{self.name} is a {self.species}"

# Derived class (Subclass)

class Dog(Animal):

def \_\_init\_\_(self, name, breed):

# Call the constructor of the base class

super().\_\_init\_\_(name, species="Dog")

self.breed = breed

def make\_sound(self):

return "Woof!"

def display\_info(self):

return f"{self.name} is a {self.breed} {self.species}"

# Create instances of the classes

animal = Animal("Generic Animal", "Unknown")

dog = Dog("Buddy", "Labrador")

# Use methods from the base class

print(animal.display\_info()) # Output: Generic Animal is a Unknown

print(animal.make\_sound()) # Output: Some generic sound

# Use methods from the derived class, which overrides the base class method

print(dog.display\_info()) # Output: Buddy is a Labrador Dog

print(dog.make\_sound()) # Output: Woof!

In this example, we have a base class called Animal, which represents generic animals. It has attributes name and species, as well as methods make\_sound() and display\_info(). The make\_sound() method returns a generic sound, and the display\_info() method displays the animal's name and species.

The derived class, Dog, is created by inheriting from the Animal class. It adds a new attribute called breed and overrides the make\_sound() and display\_info() methods. The make\_sound() method returns "Woof!" specifically for dogs, and the display\_info() method includes the breed in the output.

When we create instances of the classes (animal and dog) and call their methods, we can see how the subclass inherits the behavior of the base class and can override or extend it as needed. This way, you can create specialized classes with shared functionality, promoting code reusability and making your code more organized and maintainable.

**Q32. Suppose class C inherits from classes A and B as class C(A,B).Classes A and B both have their own versions of method func(). If we call func() from an object of class C, which version gets invoked?**

**Answer :**

In Python, when a class inherits from multiple base classes (multiple inheritance), the order in which those base classes are listed in the inheritance declaration (class C(A, B)) matters. The method resolution order (MRO) determines the order in which Python looks for methods in the inheritance hierarchy. The MRO follows the C3 Linearization algorithm to ensure a consistent and predictable order.

In the case of calling a method like func() from an object of class C, Python will follow the MRO to determine which version of the method gets invoked. Python will search for func() in the following order:

C: Python first looks in the class C itself to see if it has a func() method. If it does, this version will be invoked.

A: If func() is not found in C, Python will look in class A (the first base class listed).

B: If func() is not found in A, Python will then look in class B (the second base class listed).

The method that is found first in this search order will be the one that gets invoked. If func() is found in C, it will use that method. If not, it will use the one found in either A or B, depending on their order in the inheritance declaration.

**Q33. Which methods/functions do we use to determine the type of instance and inheritance?**

In Python, you can use the following methods and functions to determine the type of an instance and to inspect its inheritance hierarchy:

1. **type() function:**
2. **isinstance() function:**
3. **issubclass() function:**
4. **type() attribute:**
5. **super() function:**

**Q34. Explain the use of the 'nonlocal' keyword in Python.**

The nonlocal keyword in Python is used to indicate that a variable is not local to the current function's scope, but it also isn't a global variable. Instead, it refers to a variable in the nearest enclosing (non-global) scope that has that variable defined. In other words, it allows you to work with variables in an outer (enclosing) function's scope, typically within nested functions or closures.

**Here's how you can use the nonlocal keyword:**

1. **Accessing Outer Function's Variable:**
2. **Creating Closures:**
3. **Avoiding Unintended Variable Creation:**

**Q35. What is the global keyword?**

The global keyword in Python is used to indicate that a variable is a global variable, meaning it is not bound to a specific function or local scope but is available throughout the entire program. When you declare a variable as global using the global keyword, you can both access and modify that variable from anywhere in your code

**Here's how you can use the global keyword:**

1. **Declaring a Global Variable:**
2. **Accessing a Global Variable:**
3. **Avoiding Local Variable Creation:**