### What is the difference between == and is in Python?

a = [1, 2, 3]

b = [1, 2, 3]

print(a == b)  # True, because the contents are the same

print(a is b)  # False, because they are different objects

**Explanation:** **==** checks for equality of values, while **is** checks for identity (i.e., whether they are the same object in memory).

**Encapsulation(bundling the data (attributes) and methods (functions) that operate on the data into a single unit known as a class.)**

. It restricts access to the inner workings of a class and hides the data from outside interference.

Encapsulation is achieved in Python by using private attributes and methods 🡪 which are denoted by prefixing their names with double underscores \_\_.

These private members are not directly accessible from outside the class.

class A:

    \_a = 10

    \_\_b = 20

    #print(\_a)           # as it is inside the class , self is not req.

    def show(self):

        print('protected',self.\_a)  # outside the class so self is req.

        print('private',self.\_\_b)

obj1 =A()

obj1.show()

print(obj1.\_a)

# print(obj.\_\_b) error cz private method can’t be accessed outside from the class

print('by mangling method we can access private member from outside the class',obj1.\_A\_\_b)

10

protected 10

private 20

10

by mangling method we can access private member from outside the class 20

In your class A example, you're using both protected (\_a) and private (\_\_b) attributes, which are a part of encapsulation. Let's analyze it:

Who Operates on the Data?

The Show method operates on the data by displaying the values of the attributes \_a and \_\_b.

Operations on Data:

The Show method retrieves and displays the values of the attributes \_a and \_\_b.

Encapsulation in Action:

1. Data and Methods in One Unit:
   * The class A encapsulates the data (\_a and \_\_b) and the method (Show) that operates on this data.
2. Data Protection:
   * Protected Attribute: \_a is protected, which means it can be accessed and modified within the class and its subclasses.
   * Private Attribute: \_\_b is private, which means it cannot be accessed or modified directly from outside the class. However, it can still be accessed using name mangling.
3. Method Access:
   * The method Show provides a controlled way to access and display the values of the protected and private attributes.

**Abstraction (A process of hiding the implementation details from user) (abstract method, class& abstraction has same exmpl)**

Is the process of hiding the complex implementation details and showing only the essential features of an obj.

It focuses on what an obj does rather than how it achieve its functionalities.

Allowing easier management and understanding of code .

1. from abc import ABC, abstractmethod
2. class Animal(ABC):
3. @abstractmethod # mnemonic : abs – in car— car\_pass another car
4. def sound(self):
5. pass
7. class Dog(Animal):
8. def sound(self):
9. print('dog barks')
10. class Cat(Animal):
11. def sound(self):
12. print('cat meows')
14. dog = Dog()
15. cat = Cat()
16. dog.sound()
17. cat.sound()

dog barks

cat meows

# Characteristics

1. Hiding Complex Implementation Details: by encapsulating implementation details within classes and providing a simplified interface for interaction.

class Calculator:

    def add(self, x, y):

        return x + y

    def subtract(self, x, y):

        return x - y

# Usage

calc = Calculator()

calc.add(5, 3)  # Simplified interface

calc.subtract(5, 3)

1. Encapsulation of Functionality: Abstraction encapsulates functionality within classes, hiding implementation details and exposing only essential features. Classes define methods and attributes that provide access to functionality while hiding the underlying logic.

class BankAccount:

    def \_\_init\_\_(self, balance=0):

        self.balance = balance

    def deposit(self, amount):

        self.balance += amount

    def withdraw(self, amount):

        self.balance -= amount

# Usage

account = BankAccount()

account.deposit(100)  # Encapsulation of deposit functionality

account.withdraw(50)  # Encapsulation of withdraw functionality

1. Promoting Modularity and Reusability: Abstraction promotes modularity by allowing components to be developed independently and composed together to build complex systems. Classes and interfaces provide reusable building blocks that can be combined to create diverse applications.

class Shape:

    def area(self):

        pass

    def perimeter(self):

        pass

class Circle(Shape):

    def \_\_init\_\_(self, radius):

        self.radius = radius

    def area(self):

        return 3.14 \* self.radius \*\* 2

    def perimeter(self):

        return 2 \* 3.14 \* self.radius

class Rectangle(Shape):

    def \_\_init\_\_(self, length, width):

        self.length = length

        self.width = width

    def area(self):

        return self.length \* self.width

    def perimeter(self):

        return 2 \* (self.length + self.width)

# Usage

circle = Circle(5)

print("Circle area:", circle.area())  # Reusability of area calculation

**Abstract Method & class: # mnemonic : abs system in car// car pass another car**

from abc import ABC, abstractmethod

class Animal(ABC):

    @abstractmethod

    def sound(self):

        pass

class Dog(Animal):

    def sound(self):

        print('dog barks')

class Cat(Animal):

    def sound(self):

        print('cat meows')

dog = Dog()

cat = Cat()

dog.sound()

cat.sound()

dog barks

cat meows

static method  
A static method is a method that belongs to a class rather than an instance of a class. It doesn't operate on or modify any class or instance variables. It's defined with the **@staticmethod** decorator in Python and doesn't require a reference to an instance or the class (**self** or **cls** respectively) as its first parameter. **It directly call on the class**

class A:

    pass

    @staticmethod

    def add(x,y):

        return x+y

    @staticmethod

    def sub(x,y):

        return x-y

A.add(2,3)

A.sub(8,2)

Static methods are useful for grouping utility functions related to the class within the class definition.'''

In a static method, **self** is not required as an implicit first parameter because static methods belong to the class rather than instances of the class.

Here's why **self** is not required in static methods:

1. **No Instance Reference**: Static methods are called on the class itself, not on instances of the class. Therefore, they don't have access to instance attributes or methods, and there's no need for the **self** parameter to reference the instance.

Inheritance allows a class (subclass) to inherit attributes and methods from another class (superclass),

Promoting 🡪 code reuse

* creating hierarchical relationships between classes

MRO🡪 Method Resolution Order

(MRO) is the order in which **Python searches for methods and attributes in a class hierarchy**. When you call a method or access an attribute on an object, Python follows a specific sequence to determine which method or attribute to use.

By following the MRO, Python determines the order in which it searches for methods and attributes,

Process of MRO

* **Child Before Parent:** It starts from the derived class and follows the inheritance chain recursively until it reaches the base classes.
* **Left-to-Right:** It first explores the left-most base class, then its parents, and so on, until it reaches the right-most base class.

**Multiple Inheritance:** In multiple inheritance, the order of the base classes matters because it affects the method resolution order. Depending on the MRO, **super()** may call methods from different parent classes. Due to this complexity, **super()** is less commonly used in multiple inheritance

**Here are the primary purposes and scenarios where super() is most commonly used:**

**1. Calling Parent Class Methods**

super() is most commonly used to call a method from the parent class without explicitly naming the parent class. This is especially useful when dealing with method overriding in child classes.

Example:

class Animal:

    def speak(self):

        print("Animal speaks")

class Dog(Animal):

    def speak(self):

        super().speak()  # Calls the speak method from the Animal class

        print("Dog barks")

dog = Dog()

dog.speak()

Output:

Animal speaks

Dog barks

**2. Ensuring Consistency with Multiple Inheritance**

While super() is less commonly used in complex multiple inheritance scenarios due to potential complications, it can still be very useful to ensure that the MRO is respected. This helps in maintaining a consistent and predictable behavior.

In this example, super() ensures that the \_\_init\_\_ methods of all parent classes are called in the correct order as per the MRO.

2

class A:

    def \_\_init\_\_(self):

        print("A init")

        super().\_\_init\_\_()

class B:

    def \_\_init\_\_(self):

        print("B init")

        super().\_\_init\_\_()

class C(A, B):

    def \_\_init\_\_(self):

        print("C init")

        super().\_\_init\_\_()

c = C()

Output:

C init

A init

B init

**Step-by-Step Execution**

1. **Creating C instance**: When **c = C()** is executed, the **\_\_init\_\_** method of **C** is called first.
2. **Calling super() in C**:
   * **super().\_\_init\_\_()** in **C** calls the **\_\_init\_\_** method of **A**, the next class in the MRO.
3. **Calling super() in A**:
   * **super().\_\_init\_\_()** in **A** calls the **\_\_init\_\_** method of **B**, the next class in the MRO after **A**.
4. **Calling super() in B**:
   * **super().\_\_init\_\_()** in **B** calls the **\_\_init\_\_** method of **object**, the next class in the MRO after **B**.

**3. Mixin Classes**

When using mixin classes, super() can be very helpful to ensure that all necessary methods from multiple mixin classes are called correctly.

In this example, super().save() ensures that the save method from each mixin class (LogMixin and ValidateMixin) is called before the save method of the DataModel class, following the MRO.

class LogMixin:

    def save(self):

        print("Logging data")

        super().save()

class ValidateMixin:

    def save(self):

        print("Validating data")

        super().save()

class DataModel(LogMixin, ValidateMixin):

    def save(self):

        print("Saving data to database")

data\_model = DataModel()

data\_model.save()

Output:

Logging data

Validating data

Saving data to database

**4. Avoiding Hardcoding Parent Class Names**

Using super() avoids hardcoding parent class names in the code, making it easier to maintain and refactor. This is especially beneficial when dealing with large codebases where class hierarchies might change over time.

class Base:

    def \_\_init\_\_(self):

        print("Base init")

class Derived(Base):

    def \_\_init\_\_(self):

        super().\_\_init\_\_()  # No need to hardcode Base

        print("Derived init")

derived = Derived()

Output:

Base init

Derived init

Conclusion

In summary, super() is mostly used for:

Calling parent class methods in a clean and maintainable way.

Ensuring consistent behavior in multiple inheritance scenarios by respecting the MRO.

Facilitating mixin classes where multiple inheritance is used to extend functionality.

Avoiding hardcoding of parent class names, making the code more maintainable and flexible.

In Python, when you create an instance of a derived class (such as **Dog**), Python automatically tries to call the **\_\_init\_\_()** method of the base class (**Animal**) if **\_\_init\_\_()** method hasn't been overridden by the derived class (**Dog**). This is known as constructor chaining or method resolution order (MRO).

However, if the derived class (**Dog**) has its own **\_\_init\_\_()** method and you want to call the **\_\_init\_\_()** method of the base class (**Animal**) explicitly within it, you need to use **super().\_\_init\_\_(name)** to call the base class's **\_\_init\_\_()** method.

So, in the case where you define **\_\_init\_\_()** method in the **Dog** class and want to pass **name** argument to the **\_\_init\_\_()** method of the **Animal** class, you need to use **super().\_\_init\_\_(name)** to explicitly call the base class's **\_\_init\_\_()** method with the **name** parameter.

**What Is Mixin?**

Mixin is a type of multiple inheritance wherein you can combine behaviors and

attributes of more than one parent class.

❏ There are two main situations where mixins are used: to provide a lot of optional

features for a class and to use one particular feature in a lot of different classes

**Polymorphism(**means having many forms) that is same function name (but different signatures) being used for different types

**It allows objects of different classes to be treated as objects of a common superclass**, enabling flexibility and extensibility in code. In Python, polymorphism is achieved through method overriding and duck typing. Here's an example:

class Animal:

    def sound(self):

        print("Animal makes a sound")

class Dog(Animal):

    def sound(self):

        print("Dog barks")

class Cat(Animal):

    def sound(self):

        print("Cat meows")

# Create instances of different classes

animal = Animal()

dog = Dog()

cat = Cat()

# Call the function with different objects

dog.sound()

cat.sound()

Dog barks

Cat meows

Explanation:

Yes, the example also demonstrates polymorphism.

Polymorphism allows objects of different classes to be treated as objects of a common super class .In your code:

Both the **Dog** and **Cat** classes define a **sound()** method, but each class provides a different implementation of this method.

When you call the **sound()** method on a **Dog** object (**dog.sound()**) or a **Cat** object (**cat.sound()**), each object responds with its own unique sound.

Despite calling the same method (**sound()**), the behavior exhibited by each object (barking for a dog and meowing for a cat) is different based on its specific implementation.

This ability of objects to exhibit different behaviors in response to the same method call is a characteristic of polymorphism. Therefore, your code example also demonstrates polymorphism.

Another form

print(5+5)       # +operator use as addition

print('5'+'5')    # +operator use as Concat

print(len('anil'))

print(len(['anil','sunil','tushar','rabi','modon']))

**Overriding:**

When a subclass provides a specific implementation for a method which is already defined in its superclass, allowing the subclass to customize/extending the behaviour of the inherited method.

The child class overrides the method inherited from the parent class by providing its own implementations.

class Animal:

    def sound(self):

        print("Animal makes a sound")

class Dog(Animal):

    def sound(self):

        print("Dog barks")

class Cat(Animal):

    def sound(self):

        print("Cat meows")

# Create instances of different classes

animal = Animal()

dog = Dog()

cat = Cat()

# Call the function with different objects

dog.sound()

cat.sound()

In this example, both **Dog** and **Cat** classes override the **sound()** method inherited from the **Animal** superclass with their own implementations The Dog class overrides the **sound** () method inherited from the Animal class by providing its own implementation (print("Dog barks.")). When you call dog. **sound** (), Python executes the overridden **sound** () method defined in the Dog class instead of the one in the Animal class.

In summary, method overriding allows you to provide a more specific implementation of a method in a child class while still preserving the original method in the parent class. It promotes code customization, reusability, and polymorphism in object-oriented programming

**Overloading: (under 1 class- many method(with same name & diff params )**

A class contains more than one method with same name, and different args

When you define multiple methods with the same name in a class, only the last method definition will be retained, effectively overriding any previous definitions.

Python does not support method overloading.

We may overload the methods but can only use the latest defined method.

class Calculator:

    def add(self, a, b):

        return a + b

    def add(self, a, b, c):

        return a + b + c

calc = Calculator()

calc.add(2,3,4)

#calc.add(2,3)   #will show error cz last method has 3 args, but you provide 2 args

Therefore, calling calc.add(2, 3) will result in a TypeError because there's no method in the calculator class that accepts only two arguments.

**9. What is pickling and unpickling?**

- serialization while data transforming in that case pickle module comes into play

- Pickling:(serializes) object -> byte stream

- dumped as a file in the memory

- pickle.dump()

- Unpickling:(deserializes) byte stream -> objects

- loads the object to memory.

- pickle.load()

**1. Generator:**

* Generates values lazily, one at a time, using the **yield** keyword.
* Saves memory by generating values on the fly rather than storing them in memory.
* Can be iterated over using a **for** loop or by calling **next()** function.

**2. Iterator:**

* Provides a way to iterate over a sequence of elements.
* Implemented using the **\_\_iter\_\_()** and **\_\_next\_\_()** methods.
* Iterators maintain internal state to remember the position during iteration.

**3. Decorator:**

* Can be used for logging, caching, authentication, etc.

**11. Instance and Class Method:**

* Instance methods operate on an instance of a class and have access to instance variables.
* Class methods operate on the class itself and have access to class variables.
* Instance methods are defined with **self** as the first parameter, while class methods are defined with **cls**.

class BankAccount:

    def \_\_init\_\_(self, account\_number, balance):

        self.\_\_account\_number = account\_number  # Private attribute

        self.\_\_balance = balance  # Private attribute

'''In the example provided, deposit, withdraw, get\_balance, and get\_account\_number are public methods because

they do not have leading underscores in their names.'''

    def deposit(self, amount):

        self.\_\_balance += amount

    def withdraw(self, amount):

        if amount <= self.\_\_balance:

            self.\_\_balance -= amount

        else:

            print("Insufficient funds!")

    def get\_balance(self):

        return self.\_\_balance

    def get\_account\_number(self):

        return self.\_\_account\_number

# Create an instance of the BankAccount class

account1 = BankAccount("123456789", 1000)

# Try to access private attributes directly (will result in an AttributeError)

# print(account1.\_\_balance)

# Access private attributes via public methods

print("Account Number:", account1.get\_account\_number())  # Output: Account Number: 123456789

print("Balance:", account1.get\_balance())  # Output: Balance: 1000

# Make a deposit

account1.deposit(500)

print("New Balance after deposit:", account1.get\_balance())  # Output: New Balance after deposit: 1500

# Make a withdrawal

account1.withdraw(200)

print("New Balance after withdrawal:", account1.get\_balance())  # Output: New Balance after withdrawal: 1300

BAnkAccount explanation

Yes, your **Bankaccount** class example demonstrates encapsulation. Let's break it down:

**Data and Methods Bundling:**

In your **Bankaccount** class:

* **Data**: **account\_no** and **balance** are the attributes (or data) of the class.
* **Methods**: **Deposit**, **Withdraw**, **get\_balance**, and **get\_accnt\_no** are the methods that operate on the data.

**Who Operates on the Data?**

The methods (**Deposit**, **Withdraw**, **get\_balance**, **get\_accnt\_no**) operate on the data (**account\_no** and **balance**).

**Operations on Data:**

1. **Deposit**: Increases the balance by a specified amount.
2. **Withdraw**: Decreases the balance by a specified amount, but only if there are sufficient funds.
3. **get\_balance**: Returns the current balance.`
4. **get\_accnt\_no**: Returns the account number.

**Encapsulation in Action:**

1. **Data and Methods in One Unit**: The **Bankaccount** class encapsulates both the data (**account\_no** and **balance**) and the methods (**Deposit**, **Withdraw**, **get\_balance**, **get\_accnt\_no**) that operate on this data.
2. **Data Protection**: Direct access to the attributes like **account\_no** and **balance** is not allowed outside the class. Instead, methods are provided to access and modify them (**get\_balance**, **get\_accnt\_no**, **Deposit**, **Withdraw**). This helps in protecting the data from unauthorized access and modification.
3. **Data Integrity**: Methods like **Withdraw** ensure that the balance doesn't go negative, maintaining the integrity of the **balance** data.

In summary, your **Bankaccount** class is a good example of encapsulation where the data and methods are bundled together in a single unit (class), and the methods operate on the data, encapsulating the data's behavior within the class and hiding its implementation details from the outside world.

Here, **account\_number** is the parameter passed to the constructor (**\_\_init\_\_** method), and it represents the account number provided when creating a new **BankAccount** object. By using **self.\_\_account\_number = account\_number**, you are assigning the value of this parameter to the private attribute **\_\_account\_number** of the object being initialized (i.e., the newly created **BankAccount** object).

Similarly, the **balance** parameter passed to the constructor is used to set the value of the **\_\_balance** attribute of the object being initialized.

So, when you create a **BankAccount** object like **account1 = BankAccount("123456789", 1000)**, the values **"123456789"** and **1000** are assigned to the **\_\_account\_number** and **\_\_balance** attributes of the **account1** object, respectively, during initialization.

**Threading:**  
Threading refers to the ability of a program to execute multiple tasks concurrently, allowing different parts of the program to run independently. Multithreading specifically refers to the concurrent execution of multiple threads within a single process, enabling parallelism and improved performance in certain applications.

Threading is a programming technique used to execute multiple operations simultaneously, improving performance and responsiveness.

implemented using the **threading** module.

* **Thread: is a pre defined class in py.**  A thread is the smallest unit of execution within a process.
* **Thread class method:** run( ), start( ) ,join( ), isAlive( ), set name( ), get name( ),
* **Concurrency:** Concurrency refers to the ability of a program to execute multiple tasks concurrently. Threads enable concurrency by allowing multiple operations to run simultaneously.
* '''Here only 1 thread is avl, that is Main thread// main thread t1 & t2 obj k one by one run korche so 6 sec time lagche, but
* we want to run this 3 obj simultaneously in 5 sec. which is the concept  of multi threading'''
* from time import sleep
* class A:
* def run (self):
* for i in range(3):
* print('Amal')
* sleep(1)
* class B:
* def run (self):
* for i in range(3):
* print('Kamal')
* sleep(1)
* t1 = A()
* t2 = B()
* t1.run()
* t2.run()

**Multithreading:**

**Multiple threads can run concurrently within the same process, each performing its own task independently.**

When you run a Python script, the Python interpreter compiles the source code into bytecode(an intermediate form), which is then executed by the Python virtual machine (PVM).

• **GIL (Global Interpreter Lock):** (GIL) is a mutex that prevents multiple native threads from executing Python bytecode simultaneously. This means that only one thread can execute Python bytecode within a single Python process at any given time, regardless of the number of threads created.

So, while multiple threads can run concurrently within the same process, the GIL restricts Python bytecode execution to only one thread at a time. This means that although multiple threads exist and can perform tasks concurrently, they cannot execute Python bytecode concurrently due to the GIL.

Practical implementations of Multi threading:

In applications where the performance is limited by input/output operations, such as reading from disk, network communication, or interacting with databases, multithreading can help. Examples include:

* **Web servers**: Handling multiple client requests simultaneously.
* **File processing**: Reading from or writing to multiple files concurrently.
* **Network applications**: Managing multiple connections in chat servers, file transfers, etc.
* **Embedded systems**: Real-time data processing from sensors.
* **Gaming**: Handling different aspects of a game (e.g., rendering, physics, user input) concurrently.

**3. GUI Applications**

Graphical User Interface applications often use multithreading to keep the user interface responsive while performing background tasks. Examples include:

* **Desktop applications**: Running background computations or loading resources without freezing the UI.
* **Mobile apps**: Handling background tasks like data fetching or image processing.

**4. Web Scraping**

When scraping data from multiple websites or pages concurrently, multithreading can significantly speed up the process. Examples include:

from time import sleep

from threading import Thread

class A(Thread):

    def run (self):

        for i in range(3):

            print('Amal')

            sleep(1)

class B(Thread):

    def run (self):

        for i in range(3):

            print('Kamal')

            sleep(1)

t1 = A()

t2 = B()

t1.start()  #start func start the 1st run method

t2.start()  #start func start the 2nd run method simultaniously  #here main thread executes t1 & t2 threads

#t1.join()       # here 3 threads performing t1, t2, and main thread

#t2.join()

print('Hello')

**Decorator:**

 A decorator is a small function that takes another function as an argument and returns a new function that "wraps" the original function. The new function produced by the decorator is then called instead of the original function when it's invoked.

[ Monkey = refers to the practice of dynamically modifying or extending the behavior of a module, class, or object at runtime, without altering its original source code.]

Decorators are often used for adding functionality such as logging, authentication, or caching to functions. **@decorator\_name** syntax

def subtract(func):

    def wrapper(a,b):

        return b-a

    return wrapper

@subtract

def add(a,b):

    return a+b

print(add(3,5))

@subtract

def multi(a,b):

    return a\*b

print(multi(2,3))

* wrapper
* wrapper returns the original function (**func**) with the provided arguments
* decorators return the wrapper function

Logging decorator

In Python, functions are first-class objects, which means they can be passed around and reassigned just like any other variables. When we apply the decorator to the **div** function, the **inner** function is returned by the decorator, and then the **div** variable is reassigned to this **inner** function

**What is the purpose of the \_\_init\_\_ method in Python classes, and how is it used?**

* The **\_\_init\_\_** method, also known as the constructor,
* is automatically created when you define a class in Python.
* It is automatically called when a new instance of the class is created.
* Its primary purpose is to initialize the object's state by setting initial values for its attributes.
* Ternary Operator:

The ternary operator, also known as the conditional expression, is a shorthand way to write an **if-else** statement in a single line.

syntax: value\_if\_true if condition else value\_if\_false

num = 5

# Use the ternary operator to check if the number is positive or negative

result = "Positive" if num > 0 else "Negative"

print(result)  # Output: Positive

* **How Exception Handled In Python?**

**Only Try & Except:**

def divide(x, y):

    try:

        result = x / y

    except ZeroDivisionError:

        print("Error: Division by zero is not allowed.")

        result = None

    return result

result = divide(10, 0)

print(result)

Error: Division by zero is not allowed.

None

Try, Except, Else,Finally:

def convert\_to\_int(string):

    try:

        # Try to convert the string to an integer

        number = int(string)

    except ValueError:

        # Handle the case where the string cannot be converted to an integer

        print("Error: Unable to convert the string to an integer.")

        number = None

    else:

        # If no exceptions occurred, print the converted integer

        print(f"The converted integer is: {number}")

    finally:

        # Perform cleanup actions, if any (none in this example)

        pass

# Test the function with valid and invalid strings

convert\_to\_int("123")    # Output: The converted integer is: 123

convert\_to\_int("abc")    # Output: Error: Unable to convert the string to an integer.

* **try Block:**
  + The **try** block is used to wrap the code that may raise an exception.
  + It allows you to test a block of code for errors.
  + If an exception occurs within the **try** block, control is transferred to the corresponding **except** block.
* **except Block:**
  + The **except** block is used to handle exceptions that occur within the **try** block.
  + It catches and handles specific types of exceptions that may occur.
  + You can have multiple **except** blocks to handle different types of exceptions.
* **else Block:**
  + The **else** block is executed if no exceptions occur in the **try** block.
  + It allows you to execute code that should run only when no exceptions are raised.
  + It's useful for separating the code that may raise exceptions from the code that should run only if the **try** block succeeds.
* **finally Block:**
  + The **finally** block is always executed regardless of whether an exception occurred or not.
  + It's typically used to clean up resources or perform cleanup actions that should always occur, such as closing files or releasing locks.

It's executed even if there is a **return** statement inside the **try** or **except** blocks.

**Copy:**

Shallow Copy

A shallow copy makes a new copy of an object but does not create new copies of the objects that are inside the original object. Instead, it just copies the references to those inner objects. This means that if you change the inner objects in the copied object, the changes will also appear in the original object.

import copy

original\_list = [1, 2, [3, 4]]

shallow\_copy\_list = copy.copy(original\_list)

# Modifying the nested list in the shallow copy

shallow\_copy\_list[2][0] = 'changed'

print("Original List:", original\_list)  # Output: [1, 2, ['changed', 4]]

print("Shallow Copy List:", shallow\_copy\_list)  # Output: [1, 2, ['changed', 4]]

Shallow copies duplicate as little as possible. A shallow copy of a collection is a copy of the **collection structure, not the elements.** With a shallow copy, two collections now share the individual elements.

The **reference is copied but the referred object is not,** therefore the original object and its clone refer to the same object.

What Gets Copied:

The shallow copy operation creates a new list.

The elements of the original list are copied to the new list, but only references to the nested objects are copied, not the nested objects themselves.

* When we modify the nested list inside shallow\_copied\_list, we are modifying the object that both lists reference.
* Therefore, both original\_list and shallow\_copied\_list show the modification.

What is a Reference?

A reference is essentially a pointer to an object's memory location.

When you copy a reference, both variables (the original and the copy) point to the same object in memory.

* However, the nested list [3, 4] is the same object in memory, referenced by both original\_list and shallow\_copied\_list.

For a single-level list (no nesting), shallow and deep copy produce the same results because there are no nested objects to reference.

* Shallow Copy:
  + Copies the structure of the object.
  + References to nested objects are copied.
  + Changes to nested objects affect both original and copied objects.
  + Use copy.copy() to create a shallow copy.
* Deep Copy:

Deep copies duplicate everything. **all of the elements in the original collection duplicated.**

* + Copies the structure and all nested objects.
  + A new instance of each nested object is created.
  + Changes to nested objects do not affect the original object.
  + Use copy.deepcopy() to create a deep copy.

Deep Copy

A deep copy makes a new copy of an object and also makes new copies of all the objects that are inside the original object. This means that changes to the inner objects in the copied object ***do not affect the original object.***

import copy

original\_list = [1, 2, [3, 4]]

deep\_copy\_list = copy.deepcopy(original\_list)

# Modifying the nested list in the deep copy

deep\_copy\_list[2][0] = 'changed'

print("Original List:", original\_list)  # Output: [1, 2, [3, 4]]

print("Deep Copy List:", deep\_copy\_list)  # Output: [1, 2, ['changed', 4]]

Original List: [1, 2, [3, 4]]

Deep Copy List: [1, 2, ['changed', 4]]

* **Local Variable:**
* A variable declared inside a function is called a local variable.
* It can only be accessed within the function where it is declared.
* Once the function execution is completed, the local variable is destroyed.

def my\_function():

    x = 10  # This is a local variable

    print(x)

my\_function()  # Output: 10

print(x)  # Error! x is not defined outside the function

* **Global Variable:**
* A variable declared outside of all functions is called a global variable.
* It can be accessed and modified anywhere in the program.
* To modify a global variable inside a function, you need to use the **global** keyword.

a = 18

def func():

global a

a = 10

print(a) o/p = 18

print(a) o/p = 10

func()

print(a) o/p = 10

a = 15

def func():

global a

print(a) o/p = 15

a = 10

print(a) o/p = 10

func()

print(a) o/p = 10

x = 10  # This is a global variable

def my\_function():

    global x  # Using global keyword to modify the global variable

    x = 20

print(x) o/p = 10

my\_function()

print(x)  # Output: 20, as the global variable has been modified

In summary, local variables are confined to the function where they are declared, while global variables can be accessed and modified anywhere in the program.

1. **'and' Operator:**
   * The and operator returns True if both operands are True.
   * If any operand is False, it returns False.
   * It evaluates the expressions from left to right and short-circuits if it encounters a False value. This means that if the first operand is False, the second operand is not evaluated because the result will always be False regardless of its value.
2. **'or' Operator:**
   * The or operator returns True if at least one of the operands is True.
   * If both operands are False, it returns False.
   * Similar to the and operator, it evaluates the expressions from left to right and short-circuits if it encounters a True value. This means that if the first operand is True, the second operand is not evaluated because the result will always be True regardless of its value.

# 'and' operator

print(True and True)   # Output: True

print(True and False)  # Output: False

print(False and True)  # Output: False

print(False and False) # Output: False

# 'or' operator

print(True or True)    # Output: True

print(True or False)   # Output: True

print(False or True)   # Output: True

print(False or False)  # Output: False

* **Use and when you want both conditions to be True.**
* **Use or when you want at least one condition to be True. [mnemonic : at least: kom kore: Or choto than And]**

**What is "Open" and "With" statement in Python?**

**open() Function**

The **open()** function is used to open a file. It returns a file object which can be used to perform various operations like reading, writing, appending, etc., on the file.

file = open("example.txt", "r")  # Opens a file named "example.txt" in read mode

content = file.read()

print(content)

file.close()  # It's important to close the file after operations are done

**with Statement**

The **with** statement in Python is used to simplify exception handling and resource management, especially when working with file objects, sockets, etc. The advantage of using **with** is that it ensures the file is properly closed after its suite finishes, even if an error occurs.

with open("example.txt", "r") as file:

    content = file.read()

    print(content)

# File is automatically closed at the end of the with block

In the **with** statement example, the file is automatically closed after the indented block of code is executed, so there's no need to explicitly close the file using **file.close()**.

Using the **with** statement is generally considered a better practice for file handling in Python as it makes the code cleaner and more readable while ensuring that resources are properly managed.

**For pdf file reading:**

from PyPDF2 import PdfReader

# Open the PDF file

with open('example.pdf', 'rb') as file:

    # Create a PdfReader object

    pdf\_reader = PdfReader(file)

    # Iterate through each page and extract text

    for page in pdf\_reader.pages:

        print(page.extract\_text())

**Monkey patching** refers to the practice of dynamically modifying or extending the behavior of a module, class, or object at runtime, without altering its original source code. It allows you to change the behavior of existing code without having to modify it directly, which can be useful for testing, debugging, or adding temporary fixes.

**Example 1:**

def add(x,y):

return x + y

def multi(x,y):

return x \* y

def patched\_add(x,y):

return x - y

def patched\_multi(x,y):

return x / y

add = patched\_add

multi = patched\_multi

print(add(9,5)) o/p = 4

print(multi(16,2)) o/p = 8.0

# Define a custom function for division

def custom\_divide(a, b):

    if b == 0:

        return float('inf')  # Return positive infinity for division by zero

    return a / b

# Original division function

def divide(a, b):

    return a / b

# Print the result of division by zero using the original function

print("Result of division by zero using original function:", divide(5, 0))  # Output: ZeroDivisionError

# Monkey patching: Replace the original divide function with the custom\_divide function

divide = custom\_divide

# Print the result of division by zero using the monkey patched function

print("Result of division by zero using monkey patched function:", divide(5, 0))  # Output: inf

* what is recursion?
* Recursion is a programming technique where a function calls itself directly or indirectly to solve a problem. In recursive functions, the function repeatedly breaks down the problem into smaller, more manageable subproblems until it reaches a base case, where the solution can be computed directly without further recursion.
* Here's a basic example of a recursive function to calculate the factorial of a number:

def factorial(n):

# Base case: if n is 0 or 1, return 1

if n == 0 or n == 1:

return 1

# Recursive case: n \* factorial(n-1)

else:

return n \* factorial(n-1)

* In this example, the **factorial** function calls itself recursively with a smaller argument (**n-1**) until it reaches the base case (**n == 0 or n == 1**), at which point it returns 1. Then, it backtracks, multiplying each returned value by the current value of **n** until it reaches the original input value.

whitespace typically includes spaces, tabs, newline characters, and sometimes carriage returns.

**10. What are in-built Data Types in Python**

If the value can change, the object is called **mutable**, while if the value cannot change, the object is called **immutable**.

|  |  |
| --- | --- |
| **DataType** | **Mutable Or Immutable?** |
| Boolean (bool) | Immutable |
| Integer (int) | Immutable |
| Float | Immutable |
| String (str) | Immutable |
| tuple | Immutable |
| frozenset | Immutable |
| list | Mutable |
| set | Mutable |
| dict | Mutable |

**14. Explain Break, Continue andtatement**

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

for i in range(1,11):

    if i == 5:

        break

    print(i)

1

2

3

4

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

for i in range(1,11):

    if i % 2 == 0:

        continue

    print(i)

1

3

5

7

9

1. **Continue Statement**:
   * If the condition **i % 2 == 0** is true (i.e., **i** is even), the **continue** statement is executed. This causes the loop to skip the rest of the code in the current iteration and move on to the next iteration.
2. **Print Statement**:
   * If the condition **i % 2 == 0** is false (i.e., **i** is odd), the **print(i)** statement is executed, printing the current value of **i**.

Iteration 1: i = 1

Condition: 1 % 2 == 0 → False

print(i) executes, outputting 1.

Iteration 2: i = 2

Condition: 2 % 2 == 0 → True

continue executes, skipping print(i).

Iteration 3: i = 3

Condition: 3 % 2 == 0 → False

print(i) executes, outputting 3.

Iteration 4: i = 4

Condition: 4 % 2 == 0 → True

continue executes, skipping print(i).

● A **pass** statement is a null statement. When the Python interpreter comes across the pass statement, it does nothing and is ignored.

* The ‘self’ parameter is a reference to the current instance of the class, and is used to access variables that belongs to the class.
* Overhead: 64 bytes
* Strings ('A', 'B', 'AB'): 3 strings \* 50 bytes/string = 150 bytes
* Integer (1): 28 bytes
* List Object: 40 bytes

Detailed Calculation of memory usage of [a,b,'c',1,2,3]

1. List Object Overhead:
   * List overhead: 64 bytes (approximate for 64-bit system)
2. Element Pointers:
   * 6 elements \* 8 bytes per pointer = 48 bytes
3. Individual Elements:
   * Variables a and b: Assuming these are references, the memory they occupy is dependent on what they point to, but each reference itself is 8 bytes. So, 2 \* 8 bytes = 16 bytes.
   * String 'c': 49 bytes
   * Integers 1, 2, and 3: 3 \* 28 bytes = 84 bytes

Total Memory Usage

Summing these up:

1. List object: 64 bytes
2. Element pointers: 48 bytes
3. Individual elements: 16 bytes (references to a and b) + 49 bytes (string) + 84 bytes (integers)

Total memory usage: 64+48+16+49+84=261 bytes64+48+16+49+84=261 bytes

| **Property** | **List** | **Set** | **Tuple** | **Dictionary** |
| --- | --- | --- | --- | --- |
| **Mutability** | Mutable | Mutable | Immutable | Mutable |
| **Ordered** | Ordered | Unordered | Ordered | Unordered |
| **Duplicate** | Allows duplicates | Does not allow duplicates | Allows duplicates | Does not allow duplicate keys |
| **Indexing** | Supports indexing | Does not support indexing | Supports indexing | Does not support indexing |
| **Syntax** | **[ ]** | **{ }** | **( )** | **{ key: value }** |
| **Example** | **[1, 2, 3]** | **{1, 2, 3}** | **(1, 2, 3)** | **{'a': 1, 'b': 2}** |
| **Functions** | **append(), extend(), insert(), remove(), pop(), clear(), index(), count(), sort(), reverse()** | **add(), remove(), discard(), pop(), clear(), union(), intersection(), difference(), symmetric\_difference(), update()** | **count(), index()** | **get(), keys(), values(), items(), update(), pop(), popitem()** |

| **Common Functions** | **Data Structures** |
| --- | --- |
| **clear()** | List, Set, Dictionary |
| **count()** | List, Tuple |
| **index()** | List, Tuple |
| **pop()** | List, Set, Dictionary |

**LIST Functions:**

1. **append()**: Adds an element to the end of the list. my\_list.append(4)
2. **extend()**: Extends the list by appending elements from another list. my\_list.extend([5, 6])
3. **insert()**: Inserts an element at a specified position. my\_list.insert(0, 0)
4. **remove()**: Removes the first occurrence of a specified value. Raises an error if the element is not found. my\_list.remove(4)
5. **pop()**: Removes an element at a specified position or the last element if the index is not provided. my\_list.pop()
6. **clear()**: Removes all elements from the list.
7. **index()**: Returns the index of the first occurrence of a specified value. index = my\_list.index(3)
8. **count()**: Returns the number of occurrences of a specified value.
9. **sort()**: Sorts the list in ascending order.
10. **reverse()**: Reverses the order of the list.

**SET Function:**

1. **add()**: Adds an element to the set.
2. **remove()**: Removes a specified element from the set. Raises an error if the element is not found.
3. **discard()**: Removes a specified element from the set if it is present.
4. **pop()**: Removes and returns an arbitrary element from the set.
5. **clear()**: Removes all elements from the set.
6. **union()**: Returns a new set containing all elements from both sets.
7. **intersection()**: Returns a new set containing elements that are common to both sets.
8. **difference()**: Returns a new set containing elements present in the first set but not in the second set.
9. **symmetric\_difference()**: Returns a new set containing elements that are present in either of the sets, but not common to both.
10. **update()**: Updates the set with the union of itself and others.
11. **{}** is used to define static sets with known elements.
12. **set()** is used to create sets from iterable objects and is more flexible, as it can handle various data types and automatically removes duplicates.

**Tuple Function:**

1. **count()**: Returns the number of occurrences of a specified value.
2. **index()**: Returns the index of the first occurrence of a specified value.

**Dictionary Function:**

1. **get()**: Returns the value of the specified key. Returns None if the key does not exist.
2. **keys()**: Returns a view object containing the keys of the dictionary.
3. **values()**: Returns a view object containing the values of the dictionary.
4. **items()**: Returns a view object containing the key-value pairs of the dictionary.
5. **update()**: Updates the dictionary with the specified key-value pairs.
6. **pop()**: Removes the element with the specified key and returns its value.
7. **popitem()**: Removes the last inserted key-value pair and returns it.

| **Function** | **List** | **Set** | | **Tuple** | **Dictionary** |
| --- | --- | --- | --- | --- | --- |
| **append()** | Adds an element to the end of the list. | N/A | | N/A | N/A |
| **extend()** | Extends the list by appending elements from another list. | N/A | | N/A | N/A |
| **insert()** | Inserts an element at a specified position. | N/A | | N/A | N/A |
| **remove()** | Removes the first occurrence of a specified value. | Removes a specified element. | | N/A | N/A |
| **pop()** | Removes an element at a specified position or the last element if the index is not provided. | Removes and returns an arbitrary element. | | N/A | Removes the element with the specified key and returns its value. |
| **clear()** | Removes all elements from the list. | Removes all elements from the set. | | N/A | Removes all elements from the dictionary. |
| **index()** | Returns the index of the first occurrence of a specified value. | N/A | | Returns the index of the first occurrence of a specified value. | N/A |
| **count()** | Returns the number of occurrences of a specified value. | N/A | | Returns the number of occurrences of a specified value. | N/A |
| **sort()** | Sorts the list in ascending order. | N/A | | N/A | N/A |
| **reverse()** | Reverses the order of the list. | N/A | | N/A | N/A |
| **add()** | N/A | Adds an element to the set. | | N/A | N/A |
| **discard()** | N/A | Removes a specified element from the set if it is present. | | N/A | N/A |
| **pop()** | N/A | Removes and returns an arbitrary element. | | N/A | N/A |
| **clear()** | N/A | Removes all elements from the set. | | N/A | N/A |
| **union()** | N/A | Returns a new set containing all elements from both sets. | | N/A | N/A |
| **intersection()** | N/A | Returns a new set containing elements that are common to both sets. | | N/A | N/A |
| **difference()** | N/A | Returns a new set containing elements present in the first set but not in the second set. | | N/A | N/A |
| **symmetric\_difference()** | N/A | Returns a new set containing elements that are present in either of the sets, but not common to both. | | N/A | N/A |
| **update()** | N/A | Updates the set with the union of itself and others. | | N/A | Updates the dictionary with the specified key-value pairs. |
| **get()** | N/A | N/A | | N/A | Returns the value of the specified key. |
| **keys()** | N/A | N/A | | N/A | Returns a view object containing the keys of the dictionary. |
| **values()** | N/A | N/A | | N/A | Returns a view object containing the values of the dictionary. |
| **items()** | N/A | N/A | | N/A | Returns a view object containing the key-value pairs of the dictionary. |
| **popitem()** | N/A | N/A | | N/A | Removes the last inserted key-value pair and returns it. |
| **LIST** | | | | **ARRAY** | | | |
| The list can store the value of different types. | | | | It can only consist of value of same type. | | | |
| The list cannot handle the direct arithmetic operations. | | | | It can directly handle arithmetic operations. | | | |
| The lists are the build-in data structure so we don't need to import it. | | | | We need to import the array before work with the array | | | |
| The lists are less compatible than the array to store the data. | | | | An array are much compatible than the list. | | | |
| It consumes a large memory. | | | | It is a more compact in memory size comparatively list. | | | |
| It is suitable for storing the longer sequence of the data item. | | | | It is suitable for storing shorter sequence of data items. | | | |
| We can print the entire list using explicit looping. | | | | We can print the entire list without using explicit looping. | | | |

**Map() Function [**Syn. **map(function, iterable(s)) ]**

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

fruit = ["Apple", "Banana", "Pear"]

map\_object = list(**map**(lambda s: s[0] == "A", fruit))

print(map\_object)

**Output:**

[True, False, False]

# Map squares to a list of numbers

numbers = [1, 2, 3, 4, 5]

squared = list(map(lambda x: x\*\*2, numbers))

print(squared)  # Output: [1, 4, 9, 16, 25]

**Filter() Function**

The filter() function takes a function object and an iterable and creates a new list.

As the name suggests, filter() forms a new list that contains only elements that satisfy a certain condition, i.e. the function we passed returns True.

The syntax is:

**filter(function, iterable(s))**

fruit = ["Apple", "Banana", "Pear"]

filter\_object = (list(**filter**(lambda s: s[0] == "A", fruit))

print (filter\_object))

**Output:**

['Apple', 'Apricot']

# Filter even numbers from a list

numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

even\_numbers = list(filter(lambda x: x % 2 == 0, numbers))

print(even\_numbers)  # Output: [2, 4, 6, 8, 10]

**Reduce() Function**

It returns a single value.

Also, in Python 3 reduce() isn't a built-in function anymore, and it can be found in the functools module.

The syntax is:

**reduce(function, sequence[, initial])**

from functools import reduce

list = [2, 4, 7, 3]

print(reduce(lambda x, y: x + y, list))

print("With an initial value: " + str(reduce(lambda x, y: x + y, list, 10))

**Output:**

16

With an initial value: 26

# Compute the sum of numbers in a list

from functools import reduce

numbers = [1, 2, 3, 4, 5]

sum\_of\_numbers = reduce(lambda x, y: x + y, numbers)

print(sum\_of\_numbers)  # Output: 15

Explain the concept of \*args and **kwargs in function definitions.**

* **\*args** allows a function to accept any number of positional arguments, while **\*\*kwargs** allows a function to accept any number of keyword arguments.

    def func(\*args, \*\*kwargs):

    print("Positional arguments:", args)

    print("Keyword arguments:", kwargs)

func(1, 2, 3, a=4, b=5)

 Output: Positional arguments: (1, 2, 3)

         Keyword arguments: {'a': 4, 'b': 5}

**Difference Between Set and Frozenset in Python**

In Python, both sets and frozensets are collections of unique elements. However, there are some key differences between them:

1. **Mutability**:
   * **Set**: Mutable, meaning you can add, remove, and modify elements after the set is created.
   * **Frozenset**: Immutable, meaning you cannot change the elements after the frozenset is created.
2. **Methods**:
   * **Set**: Supports methods that modify the set, such as add(), remove(), and discard().
   * **Frozenset**: Does not support methods that modify the frozenset. It supports only read-only operations.

**Set** :

my\_set = {1, 2, 3}

print("Original set:", my\_set)

my\_set.add(4)

print("Set after adding 4:", my\_set)

my\_set.remove(2)

print("Set after removing 2:", my\_set)

Frozen set

# Creating a frozenset

my\_frozenset = frozenset([1, 2, 3]) o/p : ({1,2,3})

print("Original frozenset:", my\_frozenset)

try:

    my\_frozenset.add(4)

except AttributeError as e: (AttributeError or Exception j kono 1 ta lekha jabe)

    print("Error:", e)

# Attempting to remove an element from the frozenset (will raise an AttributeError)

try:

    my\_frozenset.remove(2)

except AttributeError as e:

    print("Error:", e)

**PEP 8**

PEP 8 is a document that provides guidelines and best practices on how to write Python code.

**Key PEP 8 Guidelines:**

* **Indentation**: Use 4 spaces per indentation level.
* **Maximum Line Length**: Limit all lines to a maximum of 79 characters.
* **Blank Lines**: Use blank lines to separate top-level function and class definitions.
* **Imports**: Place imports at the top of the file, grouped into standard library imports, related third-party imports, and local application/library-specific imports.
* **Naming Conventions**: Use meaningful names for variables, functions, classes, and modules.

A context manager in Python allows for resource management by defining the setup and cleanup actions that should be executed before and after a block of code, respectively.

use case: to manage resources such as file streams, database connections, and network connections, ensuring they are properly acquired and released.

**Key Features**

* **Setup and Cleanup**: Context managers ensure that resources are properly acquired before the block of code is executed and are reliably released afterward.
* **with Statement**: Context managers are typically used with the with statement, which provides a clean and readable syntax for resource management.

### Creating a Context Manager

There are two primary ways to create a context manager in Python:

1. Using a class that implements the \_\_enter\_\_ and \_\_exit\_\_ methods.
2. Using the contextlib module to create a context manager from a generator function.

**\_\_new\_\_**:

* Creates a new instance of the class.
* Called before \_\_init\_\_.
* Returns the new instance.
* Used primarily for creating instances of immutable types.

Chetu pvt. Ltd

Prefetch select related

Custom middleware

Frozen set and set difference

Yield and return difference

Generator,

Deep and shallow

With statement

Pep 8 and pip

Diff == and is

Local and global variable

Context manager

Class and static method

\_\_New\_\_ and \_\_init\_\_ func

Return and yield

Filter and get

Middleware

Codeing round

s = "datacamp"

dictionary = ["data", "camp", "cam", "lack"]  
you have to find vowel from each word after create all this word from s

Extra :

Unittest

Memory management