**Function Overloading**

Function overloading refers to defining multiple functions with the same name but different parameters. This concept is common in statically-typed languages like Java or C++, where the compiler differentiates functions by their parameter count or types.

In Python, function overloading is not directly supported. Instead, Python allows flexibility in function arguments (e.g., default arguments, \*args, and \*\*kwargs) to achieve similar functionality.

class Calculator:

def add(self, a, b=0, c=0): # Default values mimic overloading

return a + b + c

calc = Calculator()

print(calc.add(5)) # Output: 5

print(calc.add(5, 10)) # Output: 15

print(calc.add(5, 10, 20)) # Output: 35

**Function Overriding**

Function overriding refers to redefining a method in a child class that is already defined in its parent class. The child class's implementation overrides the parent's method when called through an object of the child class.

class Animal:

def speak(self):

return "Animal makes a sound"

class Dog(Animal):

def speak(self): # Overriding the parent's `speak` method

return "Dog barks"

animal = Animal()

dog = Dog()

print(animal.speak()) # Output: Animal makes a sound

print(dog.speak()) # Output: Dog barks

**Static and Instance Variable**

**Instance Variables**

* Definition: Variables that are specific to an instance (object) of a class.
* Scope: Each object has its own separate copy of instance variables.
* Lifetime: Exists as long as the instance exists.
* How to Declare: Defined inside the \_\_init\_\_ method or other instance methods using self.

class Person:

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

self.name = name # Instance variable

self.age = age # Instance variable

**Static Variables**

* **Definition**: Variables that are shared across all instances of the class.
* **Scope**: A single copy of the variable is shared among all instances of the class.
* **Lifetime**: Exists as long as the class is in memory.
* **How to Declare**: Defined directly inside the class but outside any instance methods (without self).

class Person:

species = "Homo sapiens" # Static variable

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

self.name = name # Instance variable

self.age = age # Instance variable

**When to Use**

* **Instance Variables**: When you need data that is unique to each object.
* **Static Variables**: When you need data that is shared across all objects.

**Ternary condition**

x = 5

result = "Positive" if x > 0 else "Negative" if x < 0 else "Zero"

print(result) # Output: "Positive"

Membership and Identity Operator

**1. Membership:**

Membership refers to checking whether a value is present in a sequence (such as a list, tuple, or string) or a collection (like a set or dictionary).

* **in**: The in operator is used to check if a value exists in a collection.
* **not in**: The not in operator checks if a value does **not** exist in a collection.

# Membership with a list

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

# Check if 3 is in the list

print(3 in numbers) # True

# Check if 6 is not in the list

print(6 not in numbers) # True

**2. Identity:**

Identity refers to comparing the actual memory address or the **identity** of two objects. In Python, this is done using **is** and **is not** operators.

* **is**: The is operator checks if two variables point to the same object in memory (i.e., if they are the exact same object).
* **is not**: The is not operator checks if two variables do not point to the same object in memory.

# Identity comparison with integers

x = 5

y = 5

print(x is y) # True, because small integers are cached and point to the same object

# Identity comparison with lists

a = [1, 2, 3]

b = [1, 2, 3]

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

# Identity with 'is not'

print(a is not b) # True, because 'a' and 'b' are different objects

**Interpreted language**

**Characteristics of Interpreted Languages:**

1. **Execution Line-by-Line**: The interpreter executes code line-by-line, translating each line into machine code or intermediate code as it runs.
2. **No Compilation**: Unlike compiled languages, there is no separate compilation step. The interpreter directly processes the source code.
3. **Portability**: Since the interpreter can be written for different platforms, the same code can be executed on any system with the corresponding interpreter, making interpreted languages often more portable.
4. **Ease of Debugging**: Debugging is often easier since the interpreter stops when it encounters an error and reports the line where the error occurred.

**Immutable data types**

In Python, immutable data types are types whose values cannot be changed once they are created. If you try to modify an immutable object, a new object is created instead.

**Numbers**:

* int, float, complex
* Example: Once you assign a value to an integer, it cannot be changed.

a = 10

a += 1 # This creates a new integer object; a is not modified in place.

print(a) # Output: 11

**Strings (str)**:

* Strings are immutable in Python. Any operation on a string that tries to change its content will create a new string.

s = "hello"

s = s.upper() # Creates a new string object

print(s) # Output: "HELLO"

**Tuples (tuple)**:

* A tuple is an ordered collection of elements, and once created, you cannot modify the elements inside the tuple (e.g., add, remove, or change the values of elements).

t = (1, 2, 3)

# t[0] = 4 # This will raise a TypeError

**Frozen Sets (frozenset)**:

* A frozenset is similar to a set, but it is immutable, meaning you cannot add or remove elements after its creation.

fs = frozenset([1, 2, 3])

# fs.add(4) # This will raise an AttributeError

**Mutable Data Types**

In Python, mutable data types are types whose values or contents can be modified after they are created. These types allow you to change, add, or remove elements without creating a new object.

**Lists (list)**:

* Lists are ordered collections that can contain elements of different types, and you can modify, add, or remove elements.

my\_list = [1, 2, 3]

my\_list[0] = 10 # Modify an existing element

my\_list.append(4) # Add an element at the end

my\_list.remove(2) # Remove an element

print(my\_list) # Output: [10, 3, 4]

**Dictionaries (dict)**:

* Dictionaries are collections of key-value pairs, and you can modify the values associated with a key, add new key-value pairs, or delete keys.

my\_dict = {'a': 1, 'b': 2}

my\_dict['a'] = 10 # Modify a value

my\_dict['c'] = 3 # Add a new key-value pair

del my\_dict['b'] # Remove a key-value pair

print(my\_dict) # Output: {'a': 10, 'c': 3}

**Sets (set)**:

* Sets are unordered collections of unique elements, and you can add or remove elements from a set.

my\_set = {1, 2, 3}

my\_set.add(4) # Add an element

my\_set.remove(2) # Remove an element

print(my\_set) # Output: {1, 3, 4}

**Mutable vs Immutable:**

* **Mutable objects** can be changed in place (e.g., lists, dictionaries, sets).
* **Immutable objects** cannot be changed once they are created (e.g., strings, tuples, numbers).

**Lambda Function**

A **lambda function** in Python is a small anonymous function that is defined using the lambda keyword.

***lambda arguments: expression***

 **lambda** is the keyword to define the function.

 **arguments** is the list of parameters (just like parameters in a normal function).

 **expression** is a single expression that is evaluated and returned by the function. Lambda functions can't have multiple expressions or statements.

# A simple lambda function that adds 10 to a number

**add\_10 = lambda x: x + 10**

**print(add\_10(5)) # Output: 15**

# A lambda function that multiplies two numbers

**multiply = lambda x, y: x \* y**

**print(multiply(2, 3)) # Output: 6**

**Memory Management in Python**

Memory management in Python is handled by the **Python memory manager**, which takes care of the allocation and deallocation of memory in a way that aims to optimize performance and resource usage.

1. **Memory Allocation**

When you create an object in Python (like a variable or data structure), the memory is allocated dynamically. This means that Python automatically handles memory for you without requiring you to specify its size or layout.

Python objects are stored in a private heap, and the **Python memory manager** manages this heap.

**a. Private Heap Space**

* Every Python program has a private heap, where all objects and data structures are stored.
* Python's memory manager interacts with this heap space to manage object creation, deletion, and reallocation.

**2. Automatic Memory Management (Reference Counting)**

Python uses **reference counting** to track how many references exist to each object in memory. Every object in Python has an associated reference count, which is incremented or decremented when a reference to the object is created or destroyed.

* When an object's reference count drops to zero (i.e., no references point to it), the memory for that object is deallocated.

a = [1, 2, 3] # reference count of the list is 1

b = a # reference count of the list becomes 2

del a # reference count of the list becomes 1

del b # reference count of the list becomes 0, and the memory is freed

**3**. **Garbage Collection (GC)**

While reference counting handles most of the memory deallocation, it can’t handle certain scenarios, like **cyclic references**, where two or more objects reference each other, making their reference count non-zero even though they are no longer reachable from the program.

Python uses **garbage collection** (GC) as an additional mechanism to detect and clean up cyclic references.

a. **Cyclic Garbage Collection**

Python uses the **gc module** to manage cyclic garbage collection.

The cyclic garbage collector works by identifying groups of objects involved in circular references, breaking the cycle, and then freeing the memory.

**4. Dynamic Typing**

In Python, variables are dynamically typed, which means that the type of a variable is determined at runtime. The memory allocation of objects is therefore determined dynamically as well, allowing Python to be flexible with its memory usage.

a = 42 # Integer type

a = "hello" # String type, memory allocation is changed

**5**. **Memory Deallocation**

Python automatically deallocates memory when objects are no longer needed (i.e., when their reference count drops to zero or they are part of a cyclic reference detected by the garbage collector). This is part of the **automatic memory management** system**.**

**6. Memory Leaks**

Even though Python’s memory management system is highly effective, memory leaks can still occur, especially if references to objects are unintentionally retained (e.g., in global variables, long-lived caches, or circular references that are not cleaned up properly).

**7. Memory Leaks**

Even though Python’s memory management system is highly effective, **memory leaks** can still occur, especially if references to objects are unintentionally retained (e.g., in global variables, long-lived caches, or circular references that are not cleaned up properly).

**CPython**

 **CPython** is the most popular implementation of Python, developed in the C language.

 It reads and executes Python code by first compiling it to an intermediate bytecode, which is then executed by a virtual machine (the **Python Virtual Machine (PVM)**).

It manages memory using reference counting and garbage collection.

**Call by Sharing**

**Call by Sharing** (also known as **Call by Object Reference** or **Call by Assignment**) is a parameter passing mechanism in Python, which is commonly used when passing arguments to functions.

Example of Call by Sharing:

# Function modifying a mutable object (list)

def modify\_list(my\_list):

my\_list.append(4)

print("Inside function:", my\_list)

# Original list

numbers = [1, 2, 3]

modify\_list(numbers)

# The original list is modified outside the function

print("Outside function:", numbers)

**Inside function: [1, 2, 3, 4]**

**Outside function: [1, 2, 3, 4]**

Example with an Immutable Object:

# Function attempting to modify an immutable object (integer)

def modify\_value(x):

x = 10

print("Inside function:", x)

# Original value

num = 5

modify\_value(num)

# The original value remains unchanged outside the function

print("Outside function:", num)

**Inside function: 10**

**Outside function: 5**

**Key Points:**

* **Call by Sharing** means that the function receives a reference to the original object, but whether the object can be changed depends on whether it is mutable or immutable.
* **Mutable objects** (like lists, dictionaries, sets) can be modified inside the function, and the changes are reflected outside.
* **Immutable objects** (like integers, strings, and tuples) cannot be changed directly, so changes to them inside the function do not affect the original value outside the function.

**Call by Value & Call by Reference**

**1. Call by Value:**

* In **call by value**, the actual value of the argument is passed to the function.
* The function works with a copy of the original data, meaning changes made inside the function **do not affect** the original variable.
* This approach ensures that the original data remains unchanged.

def modify\_value(x):

x = 10

print("Inside function:", x)

num = 5

modify\_value(num) # Passes a copy of the value

print("Outside function:", num) # Original value remains unchanged

Inside function: 10

Outside function: 5

**2. Call by Reference:**

* In **call by reference**, a reference to the actual data is passed to the function.
* The function works with the actual data (not a copy), so changes made inside the function **do affect** the original variable.

def modify\_list(my\_list):

my\_list.append(4)

print("Inside function:", my\_list)

numbers = [1, 2, 3]

modify\_list(numbers) # Passes a reference to the list

print("Outside function:", numbers) # Original list is modified

Inside function: [1, 2, 3, 4]

Outside function: [1, 2, 3, 4]

**Falsy values**

In Python, **falsy values** are values that are considered False when evaluated in a boolean context (e.g., in an if statement).

None ,False ,0 ,0.0 ,0j ,[],() ,{} ,set() ,range(0, 0)

**Pickling in Python**

**Pickling** is the process of serializing an object into a byte stream so it can be stored in a file or sent over a network. This byte stream can later be deserialized (unpickled) to recreate the original Python object.

Pickling is provided by the pickle module in Python.

**Pickling Process**

1. **Serialization**: Converts the Python object into a byte stream.
2. **Deserialization**: Converts the byte stream back into a Python object.

**Packing and Unpacking in Python**

In Python, **packing** and **unpacking** are techniques to work with collections like tuples, lists, and dictionaries in a concise way. They allow assigning multiple values to a single variable or extracting multiple values from a collection efficiently.

**Packing**

**Packing** is the process of combining multiple values into a single variable (usually as a tuple).

# Packing values into a tuple

packed\_data = 1, 2, 3, 4

print(packed\_data) # Output: (1, 2, 3, 4)

**Unpacking**

**Unpacking** is the process of extracting individual elements from a collection and assigning them to variables.

# Unpacking a tuple

a, b, c, d = (1, 2, 3, 4)

print(a, b, c, d) # Output: 1 2 3 4

**Metaclasses in Python**

A **metaclass** is a "class of a class."

 **Classes are instances of metaclasses**:

* When you define a class, Python internally uses a metaclass to create it.
* By default, the metaclass for all classes is type.

 **Metaclasses control the creation of classes**:

* You can customize class creation by overriding specific methods in a metaclass, such as:
  + \_\_new\_\_: Controls the creation of the class object.
  + \_\_init\_\_: Initializes the class after it has been created.

**Frameworks**: Many frameworks like Django use metaclasses to define and customize behaviors (e.g., models).

# Define a simple metaclass

class MyMeta(type):

def \_\_new\_\_(cls, name, bases, dct):

# Add a new attribute to the class

dct['new\_attribute'] = "I was added by the metaclass!"

return super().\_\_new\_\_(cls, name, bases, dct)

# Use the metaclass

class MyClass(metaclass=MyMeta):

pass

# Using the class

print(MyClass.new\_attribute) # Output: I was added by the metaclass!

**Shallow Copy**

A shallow copy creates a new object, but it does not recursively copy objects inside the original object. Instead, it only copies references to the nested objects.

Modifying an element in a nested list affects both original and shallow\_copied.

|  |
| --- |
|  |

|  |
| --- |
| copy.copy() |

**Deep Copy**

A deep copy creates a new object and recursively copies all objects inside the original, resulting in a fully independent copy.

Changes in the nested objects of the deep copy do not affect the original object.

**Pure function**

A **pure function** is a concept from functional programming. It refers to a function that has the following two properties.

**No Side Effects**:

* The function does not modify any external state or variables.
* It relies only on its input parameters and returns a value.

**Deterministic**:

* The function always produces the same output for the same set of inputs.

# Pure function example

def add\_numbers(a, b):

return a + b

# Using the function

result = add\_numbers(2, 3) # Always returns 5 for inputs 2 and 3

print(result) # Output: 5

**Magic** **Methods**

**Magic methods** (also known as **dunder methods** because they begin and end with double underscores, e.g., \_\_init\_\_, \_\_str\_\_) in Python are special methods that allow developers to define how objects of a class behave in various situations. They are used to override or extend the default behavior of Python operators, functions, and objects.

\_\_init\_\_, \_\_str\_\_, \_\_add\_\_, \_\_sub\_\_ etc

Operator Overloading

**Operator overloading** in Python allows you to define custom behavior for operators (like +, -, \*, etc.) when they are used with objects of a user-defined class.

class Vector:

def \_\_init\_\_(self, x, y):

self.x = x

self.y = y

def \_\_add\_\_(self, other):

# Overloading the `+` operator

return Vector(self.x + other.x, self.y + other.y)

def \_\_str\_\_(self):

return f"Vector({self.x}, {self.y})"

# Creating two vector objects

v1 = Vector(1, 2)

v2 = Vector(3, 4)

# Adding the vectors

v3 = v1 + v2 # This calls the `\_\_add\_\_` method

print(v3) # Output: Vector(4, 6)

In this example:

* The \_\_add\_\_ method defines how the + operator works for Vector objects. When you use v1 + v2, Python automatically calls v1.\_\_add\_\_(v2).

**Method Used to Create Destructor**

In Python, destructors are created using the **\_\_del\_\_** method. It is invoked when an object is about to be destroyed (garbage collected).

class MyClass:

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

self.name = name

print(f"Object {self.name} created")

def \_\_del\_\_(self):

print(f"Object {self.name} destroyed")

obj = MyClass("Test")

del obj # Explicitly delete the object

**Decorators**

A **decorator** in Python is a function that modifies the behavior of another function or method. It is often used for logging, enforcing access control, memoization, or instrumentation.

# Decorator function

**def uppercase\_decorator(func):**

**def wrapper():**

**result = func()**

**return result.upper()**

**return wrapper**

**# Applying the decorator**

**@uppercase\_decorator**

**def greet():**

**return "hello"**

**print(greet()) # Output: HELLO**

def logger(func):

def wrapper(\*args, \*\*kwargs):

print(f"Executing {func.\_\_name\_\_}")

return func(\*args, \*\*kwargs)

return wrapper

@logger

def add(a, b):

return a + b

**def deco(func):**

**def wrap():**

**print("before")**

**func()**

**print("after")**

**return wrap**

**@deco**

**def greet():**

**print("hello")**

**greet()**

**Generators**

Generators are a type of iterable in Python that allow you to lazily generate a sequence of values using the yield keyword.

A generator is defined like a normal function but uses yield instead of return to provide values.

def countdown(n):

while n > 0:

yield n

n -= 1

# Use the generator

for num in countdown(5):

print(num)

**Iterator**

Objects that can be iterated using \_\_iter\_\_ and \_\_next\_\_.

class Counter:

def \_\_init\_\_(self, start, end):

self.current = start

self.end = end

def \_\_iter\_\_(self):

return self

def \_\_next\_\_(self):

if self.current > self.end:

raise StopIteration

self.current += 1

return self.current - 1

**Context Managers in Python**

Context managers are a way of managing resources in Python. They are commonly used for tasks like opening and closing files, managing database connections, or locking and unlocking resources. Context managers ensure that resources are properly acquired and released, even if an error occurs during execution.

The most common way to use a context manager is with the with statement.

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

content = file.read()

# The file is automatically closed after the block, even if an exception occurs.

**How Context Managers Work**

A context manager typically has two methods:

1. **\_\_enter\_\_**: Called when entering the with block.
2. **\_\_exit\_\_**: Called when exiting the with block, even if an exception occurs.

**Use of GIL**

The GIL ensures that only one thread executes Python bytecode at a time in CPython, the standard Python interpreter. It's a mechanism to simplify memory management, especially in multi-threaded environments.

**Use Case of Class Method**

Class methods are used for operations related to the class itself rather than instances. For example, they can create alternative constructors or manage class-level attributes:

**Why Python**

Python is popular because of its simplicity, readability, and versatility. Its large standard library, vast community support, and extensive frameworks for web development, data analysis, machine learning, etc., make it suitable for both beginners and professionals.

**Use Case of Class Method**

Class methods are used for operations related to the class itself rather than instances. For example, they can create alternative constructors or manage class-level attributes:

class MyClass:

class\_attr = 0

@classmethod

def increment\_class\_attr(cls):

cls.class\_attr += 1

**Multithreading Limitations**

Due to the GIL in CPython, multithreading doesn't achieve true parallelism for CPU-bound tasks. Use multiprocessing for parallel computation instead.

**Process vs Threads**

* **Threads:** Share the same memory space, lightweight, but limited by the GIL.
* **Processes:** Have separate memory spaces, are heavier, but can run independently for true parallelism.

**Method Overloading in Python**

Python doesn’t support traditional method overloading. Instead, you can use default arguments or \*args and \*\*kwargs to achieve similar functionality:

def greet(name, msg="Hello"):

print(f"{msg}, {name}")

is**vs**==

* is: Checks if two references point to the same object.
* ==: Checks if the values of two objects are equal.

**Why is Set Elements Unique**

Sets inherently enforce uniqueness because they use hashing for membership checks.

**Loops vs Recursion**

* **Loops:** Iterative, memory-efficient, and often faster.
* **Recursion:** Elegant for problems with smaller subproblems but may lead to stack overflow for deep recursion.

**Best Case Complexity of Insertion Sort**

**O(n)** when the array is already sorted

**Typecasting**

Converting one data type into another:

x = int("10") # Converts string to integer

**Ternary Conditional Statement**

result = "Even" if num % 2 == 0 else "Odd"

**While Loop on List**

lst = [1, 2, 3]

i = 0

while i < len(lst):

print(lst[i])

i += 1

**.pyc File**

Compiled Python files for optimization, generated automatically to improve module loading time.

**Constructor in Python**

The \_\_init\_\_ method initializes instance variables when an object is created.

\_\_new\_\_**vs**\_\_init\_\_

* \_\_new\_\_: Creates the object.
* \_\_init\_\_: Initializes it.

**Why Is Everything Called an Object**

Everything in Python (functions, data types, etc.) inherits from the object class.

**self**

Refers to the instance of the class.

**PEP 8, NumPy**

* **PEP 8:** Python's style guide.
* **NumPy:** Library for numerical computations

**Modules and Packages**

* **Modules:** Single .py files.
* **Packages:** Collections of modules organized in directories.

**raise**

Used to raise exceptions:

raise ValueError("Invalid input")

**Encapsulation and Polymorphism**

1. **Encapsulation**:
   * Encapsulation hides the internal state of an object and only exposes a controlled interface.
   * Achieved through **access specifiers** like public, protected, and private.
     + Public: Accessible everywhere (self.name).
     + Protected: Accessible within the class and its subclasses (self.\_name).
     + Private: Accessible only within the class (self.\_\_name).

class Person:

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

self.\_\_name = name # Private variable

def get\_name(self):

return self.\_\_name

def set\_name(self, name):

self.\_\_name = name

**Access Specifiers**

Python uses naming conventions to simulate access specifiers:

1. Public (name): Accessible anywhere.
2. Protected (\_name): Accessible within class and subclasses.
3. Private (\_\_name): Name mangling restricts access outside the class.

class MyClass:

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

self.public = "Public"

self.\_protected = "Protected"

self.\_\_private = "Private"

**Polymorphism**:

* The ability of an object to take on many forms.
* Achieved through **method overriding** and **operator overloading**

**Example** (Method Overriding):

class Animal:

def speak(self):

return "Animal sound"

class Dog(Animal):

def speak(self):

return "Bark"

class Cat(Animal):

def speak(self):

return "Meow"

# Demonstration

animals = [Dog(), Cat()]

for animal in animals:

print(animal.speak())

**Multiple Inheritance in Python**

Python supports multiple inheritance, where a class can inherit from multiple parent classes. The **Method Resolution Order (MRO)** determines the order in which Python looks for methods in the hierarchy.

class A:

def display(self):

print("A")

class B:

def display(self):

print("B")

class C(A, B): # Inherits from A and B

pass

c = C()

c.display() # Outputs: "A" because A appears first in MRO

**Method Resolution Order (MRO)**

MRO defines the order in which Python looks for a method or attribute. It uses the **C3 Linearization** algorithm.

range**vs**xrange

* range (Python 3): Generates numbers lazily (like xrange in Python 2).
* xrange (Python 2): Lazy evaluation.

**Interpreter Language**

Python is interpreted, meaning the code is executed line-by-line. This provides flexibility but can be slower than compiled languages

\_\_**call**\_\_

Allows an object to be called like a function by defining the \_\_call\_\_ method.

class CallableClass:

def \_\_call\_\_(self, x):

return x \* x

obj = CallableClass()

print(obj(5)) # Outputs: 25

**dir()**

The dir() function in Python is used to list the attributes and methods of an object. When called without arguments, it returns the list of names in the current local scope. When called with an object as an argument, it returns the list of names of the object's attributes and methods.

# Without arguments

print(dir())

# With an object

class Example:

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

self.attribute = 'value'

def method(self):

pass

example = Example()

print(dir(example))

 dir() without arguments lists the names in the current local scope.

 dir(example) lists the attributes and methods of the example object, which would include attribute and method.

**Class Method vs Static Method**

1. **Class Method**:
   * Operates on the class, not the instance.
   * Defined with @classmethod.
   * Takes cls as the first parameter.
2. **Static Method**:
   * Does not depend on instance or class data.
   * Defined with @staticmethod.

class MyClass:

count = 0

@classmethod

def increment\_count(cls):

cls.count += 1

@staticmethod

def greet():

print("Hello!")

**Abstraction**

**Definition**: Abstraction is the concept of hiding the complex implementation details of a system and exposing only the essential features or behavior to the user. It focuses on **what an object does** rather than **how it does it**.

**How to achieve abstraction in Python**:

1. Using **Abstract Base Classes (ABC)** from the abc module.
2. Defining methods in the base class without implementing them, and enforcing implementation in derived classes.

**Key Points**:

* Abstract classes cannot be instantiated.
* Abstract methods act as a contract for subclasses to implement specific methods.

from abc import ABC, abstractmethod

class Animal(ABC):

@abstractmethod

def make\_sound(self):

pass # Abstract method

class Dog(Animal):

def make\_sound(self):

return "Bark"

class Cat(Animal):

def make\_sound(self):

return "Meow"

# Example usage

animals = [Dog(), Cat()]

for animal in animals:

print(animal.make\_sound())

**Getters and Setters**

**Definition**:

* **Getters**: Methods used to access the value of an object's private or protected attribute.
* **Setters**: Methods used to modify or set the value of an object's private or protected attribute.

In Python, getters and setters are commonly implemented using the @property decorator.

class Person:

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

self.\_\_name = name # Private attribute

# Getter

@property

def name(self):

return self.\_\_name

# Setter

@name.setter

def name(self, value):

self.\_\_name = value

# Example usage

person = Person("Alice") # Initialize the object

print(person.name) # Access the name using the getter

person.name = "Bob" # Set a new name using the setter

print(person.name) # Access the updated name

**Advantages**:

1. Provides controlled access to private attributes.
2. Allows for validation or transformation during attribute retrieval or modification.
3. Encapsulates the logic, ensuring that data integrity is maintained.

\_\_name\_\_**in Python**

In Python, \_\_name\_\_ is a special built-in variable that holds the name of the current module. It is commonly used to determine if a script is being run as the main program or if it is being imported as a module into another script.

**Behavior of**\_\_name\_\_

1. **When a Python file is run directly**:
   * The \_\_name\_\_ variable is set to "\_\_main\_\_".
   * This indicates that the file is being executed as the main program.
2. **When a Python file is imported as a module**:
   * The \_\_name\_\_ variable is set to the name of the module (the filename without the .py extension).

dataclass**in Python**

A dataclass in Python is a decorator provided by the dataclasses module introduced in Python 3.7. It simplifies the creation of classes by automatically generating special methods like \_\_init\_\_, \_\_repr\_\_, \_\_eq\_\_, and more based on the class attributes.

**Why Use**dataclass**?**

* To reduce boilerplate code for classes primarily used to store data.
* Automatically generates common methods like:
  + \_\_init\_\_ for initializing the object.
  + \_\_repr\_\_ for a human-readable string representation.
  + \_\_eq\_\_ for comparing objects.

**Without**dataclass**:**

class Point:

def \_\_init\_\_(self, x, y):

self.x = x

self.y = y

def \_\_repr\_\_(self):

return f"Point(x={self.x}, y={self.y})"

# Example usage

p1 = Point(1, 2)

print(p1) # Output: Point(x=1, y=2)

**With**dataclass

from dataclasses import dataclass

@dataclass

class Point:

x: int

y: int

# Example usage

p1 = Point(1, 2)

print(p1) # Output: Point(x=1, y=2)

**Types of Arguments**

**Positional Arguments**: Passed in order.

def greet(name):

print(f"Hello, {name}")

greet("Alice")

**Keyword Arguments**: Passed by name.

greet(name="Alice")

**Default Arguments**: Provide default values.

def greet(name="World"):

print(f"Hello, {name}")

greet()

**Variable-Length Arguments**:

* \*args: Tuple of positional arguments.
* \*\*kwargs: Dictionary of keyword arguments.

def print\_all(\*args, \*\*kwargs):

print(args, kwargs)

print\_all(1, 2, 3, a=4, b=5)

**Variadic Functions in Python**

Variadic functions are functions that accept a variable number of arguments. Python provides two ways to define variadic functions:

1. \*args: Accepts a variable number of **positional arguments** as a tuple.
2. \*\*kwargs: Accepts a variable number of **keyword arguments** as a dictionary.

**Using**\*args

The \*args parameter allows a function to accept an arbitrary number of positional arguments. These arguments are packed into a tuple.

def sum\_numbers(\*args):

return sum(args)

print(sum\_numbers(1, 2, 3)) # Output: 6

print(sum\_numbers(10, 20, 30, 40)) # Output: 100

**Using**\*\*kwargs

The \*\*kwargs parameter allows a function to accept an arbitrary number of keyword arguments. These arguments are packed into a dictionary.

def print\_details(\*\*kwargs):

for key, value in kwargs.items():

print(f"{key}: {value}")

print\_details(name="Alice", age=30, city="New York")

# Output:

# name: Alice

# age: 30

# city: New York

**Combining**\*args**and**\*\*kwargs

def display\_info(\*args, \*\*kwargs):

print("Positional arguments:", args)

print("Keyword arguments:", kwargs)

display\_info(1, 2, 3, name="Alice", age=30)

# Output:

# Positional arguments: (1, 2, 3)

# Keyword arguments: {'name': 'Alice', 'age': 30}

**Closures in Python**

A **closure** is a function object that retains access to the variables in its **lexical scope**, even when the function is called outside that scope. Closures are useful for **encapsulation** and creating functions with persistent state.

**How Closures Work**

* A **nested function** is defined inside another function.
* The inner function has access to variables from the enclosing function's scope.
* Even after the enclosing function has finished executing, the inner function "remembers" the variables in its lexical scope.

def outer\_function(msg):

# This is the enclosing function

def inner\_function():

# Inner function has access to 'msg'

print(f"Message: {msg}")

return inner\_function

# Create a closure

closure = outer\_function("Hello, World!")

closure() # Output: Message: Hello, World!