Object-oriented Programming

Unit-5

Contents

Introduction

Object-Oriented Programming (OOP) is a programming paradigm that relies on the concept of Objects & Classes.

Object-oriented programming (OOP) is a computer programming model that organizes software design around data, or objects, rather than functions and logic.

An object can be defined as a data field that has unique attributes and behavior.

With the help of this core principle, designing code and building software becomes much easier.

Object-Oriented Programming makes it easier to create modular, reusable, and maintainable software.

Object-oriented programming is an approach for modeling concrete, real-world things, like cars, as well as relations between things, like companies and employees or students and teachers.

OOP models real-world entities as software objects that have some data associated with them and can perform certain operations.

The key takeaway is that objects are at the center of object-oriented programming in Python. In other programming paradigms, objects only represent the data. In OOP, they additionally inform the overall structure of the program.

Core Principles of OOP

Classes and Objects

Encapsulation

Inheritance

Polymorphism

Abstraction

Classes and Objects

Class:

A class is a blueprint for creating objects or A class is a collection of objects.

A class contains the blueprints or the prototype from which the objects are being created. It is a logical entity that contains some attributes and methods.

Some points on Python class:

- Classes are created by keyword class.
- Attributes are the variables that belong to a class.
- Attributes are always public and can be accessed using the dot (.) operator.
  Eg.: Myclass.Myattribute

Class Definition Syntax:

```
class ClassName:
  # Statement-1
  .
  .
  .
  # Statement-N
```

Creating an Empty Class in Python

```
class Dog:
    pass
```

Objects

An object is an instance of a class. When a class is defined, no memory is allocated until an object of that class is created.

The object is an entity that has a state and behavior associated with it. It may be any real-world object like a mouse, keyboard, chair, table, pen, etc.

An object consists of

State: It is represented by the attributes of an object. It also reflects the properties of an object.

Behavior: It is represented by the methods of an object. It also reflects the response of an object to other objects.

Identity: It gives a unique name to an object and enables one object to interact with other objects.

To understand the state, behavior, and identity let us take the example of the class dog (explained above).

The identity can be considered as the name of the dog.

State or Attributes can be considered as the breed, age, or color of the dog.

The behavior can be considered as to whether the dog is eating or sleeping.

Creating an Object(Instantiating objects)

To create an instance of a class, just type its name and parenthesis after it. It’s the same process as invoking a function.

Syntax:

object_name = class_name(); $\longrightarrow$ This is the invocation of constructor

Eg:

obj = Dog()

Dog() calls the class constructor (usually __init__, though it can be named differently). The constructor initializes the object's attributes (often to default values).

“Obj” is the object reference variable that holds the memory address of the created object.

Remember: Python initializes the object's attributes with default values (usually None) or values you provide during object creation

(e.g., obj = Dog("Labrador", 3)).

These attributes and their values are stored in the allocated memory.

Encapsulation

Encapsulation is one of the fundamental concepts in object-oriented programming (OOP).

Encapsulation involves bundling the data (attributes) and methods (functions) that operate on the data into a single unit.

This puts restrictions on accessing variables and methods directly and can prevent the accidental modification of data.

To prevent accidental change, an object’s variable can only be changed by an object’s method. Those types of variables are known as private variables.

It prevents outer classes from accessing and changing attributes and methods of a class.

This also helps to achieve data hiding.

Inheritance

Inheritance is the capability of one class to derive or inherit the properties from another class.

The class that derives properties is called the derived class or child class and the class from which the properties are being derived is called the base class or parent class.

The benefits of inheritance are:

- It represents real-world relationships well.
- It provides the reusability of a code. We don’t have to write the same code again and again. Also, it allows us to add more features to a class without modifying it.
- It is transitive in nature, which means that if class B inherits from another class A, then all the subclasses of B would automatically inherit from class A.

Types of Inheritance

Single Inheritance: Single-level inheritance enables a derived class to inherit characteristics from a single-parent class.

Multilevel Inheritance: Multi-level inheritance enables a derived class to inherit properties from an immediate parent class which in turn inherits properties from his parent class.

Hierarchical Inheritance: Hierarchical-level inheritance enables more than one derived class to inherit properties from a parent class.

Multiple Inheritance: Multiple-level inheritance enables one derived class to inherit properties from more than one base class.

Polymorphism

In object oriented Programming Python, Polymorphism simply means having many forms.

Types of Polymorphism

- **Compile-Time Polymorphism (Static Polymorphism):**
  - Achieved through method overloading and operator overloading.
  - The method to be invoked is determined at compile time.
- **Run-Time Polymorphism (Dynamic Polymorphism):**
  - Achieved through method overriding.
  - The method to be invoked is determined at runtime.

Abstraction

Abstraction means hiding the complex implementation details and showing only the essential features of the object.

Also, when we do not want to give out sensitive parts of our code implementation and this is where data abstraction came.

Data Abstraction in Python can be achieved by creating abstract classes.

Defining a Class

```
class Dog:
    pass
```

You start all class definitions with the class keyword, then add the name of the class and a colon.

Python will consider any code that you indent below the class definition as part of the class's body.

The body of the Dog class consists of a single statement: the pass keyword.

Note: Python class names are written in CapitalizedWords notation by convention. For example if you want to create another class let say “nccs” then your class name would be “Nccs”

Objects

A Python object is an instance of a class. An object is a container for data (attributes) and functions (methods) that act on the data.

Object Instantiation: You create an object (instance) of a class using the class name followed by parentheses (). This is called instantiation.

```
obj = Dog()  # Create an instance named obj
```

Object Memory Initialization

Behind the Scenes: When you create an object, Python allocates memory for it in the heap (a dynamic memory area). The object's memory layout depends on the class definition.

Instance Variables: Space is allocated for each instance variable defined within the class. These variables are unique to each object.

Optional Initialization: The __init__ method (constructor) is a special method that allows you to optionally initialize instance variables during object creation. It's automatically called when you create an object using class_name().

example

```
class Car:
    # Method to start the car's engine
    def start_engine():
        return "The engine is now running."
my_car = Car()
print(my_car.start_engine())
```

Object Creation and Initialization:

When you create an object (my_car = Car()), Python allocates memory for the new object.

The __init__ method is called to initialize the object's attributes.

Constructor

A constructor is a unique function that gets called automatically when an object of a class is created.

Constructors in Python is a special class method for creating and initializing an object instance at that class.

The main purpose of a constructor is to initialize or assign values to the data members of that class.

It cannot return any value other than none.

Every Python class has a constructor; it’s not required to be defined explicitly.

The name of the constructor method is always __init__.

Syntax of Python Constructor

```
class ClassName:
    def __init__(self):
        # initializations and attribute assignments
```

Within the 'init' method, you can define the object's initial state by assigning values to the object's properties or performing other necessary startup procedures.

The self parameter refers to the current class instance to access class attributes and methods.

Rules of Python Constructor

- It starts with the def keyword, like all other functions in Python.
- It is followed by the word init, which is prefixed and suffixed with double underscores with a pair of brackets, i.e., __init__().
- It takes an argument called self, assigning values to the variables.

Remember: “Self” is a reference to the current instance of the class. It is created and passed automatically/implicitly to the __init__() when the constructor is called.

example of a simple class with a constructor:

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

person = Person("John", 30)
print(person.name)
print(person.age)
```

In this example, the __init__ method is called when the Person object is created, and it sets the name and age attributes of the object.

The __init__ method is commonly referred to as the “constructor” because it is responsible for constructing the object. It is called automatically when the object is created, and it is used to initialize the object’s attributes.

Types of Constructors in Python

1. Parameterized Constructor
2. Non-Parameterized Constructor
3. Default Constructor

Default Constructor

When a class is defined without an explicit constructor in Python, Python automatically provides a default constructor.

This default constructor is a basic, non-parameterized constructor that performs no specific task or initialization beyond the basic object creation.

```
class Assignments:
    check = "not done"

    def is_done(self):
        print(self.check)

obj = Assignments()

obj.is_done()
```

Non-Parameterized Constructor

A non-parameterized constructor in Python is a constructor that does not accept any arguments except the mandatory self.

This type of constructor is used for initializing class members with default values or performing standard initialization tasks that do not require external inputs.

```
class Fruits:
    favourite = "Apple"

    def __init__(self):
        self.favourite = "Orange"
    def show(self):
        print(self.favourite)

obj = Fruits()
obj.show()
```

Parameterized Constructor

In Python, a parameterized constructor is a type of constructor that takes additional arguments besides the standard self reference.

These arguments are used to initialize the object's attributes or perform other operations when the object is created.

```
class Family:
    members = 5
    def __init__(self, count):
        print("This is a parameterized constructor")
        self.members = count
    def show(self):
        print("Number of members is", self.members)

family_object = Family(10)
family_object.show()
```

Understanding Self Parameter in Constructors

A reference to the object being created is provided by the self parameter in constructors.

It is a unique parameter that can be used to access the class’s instance variables and methods.

When an object of the class is created, the self parameter, which is always the first parameter of a constructor, is automatically passed.

Example

```
class Employee:
    def __init__(self, name, age, salary):
        self.name = name
        self.age = age
e = Employee("John", 25)
print("Name:", e.name)
print("Age:", e.age)
```

Output:

```
Name: John
Age: 25
```

Explanation of how self parameter works in constructors

In the __init__ method, self refers to the newly created Employee object (like “e” in this case).

When you assign a value to self.name, you're actually assigning it to the name instance variable of the specific object being created. So, e.name becomes "John".

Similarly, self.age assigns the value 25 to the age instance variable of the “e” object.

Key Points

- self is always the first parameter in a constructor's definition.
- You don't explicitly pass self when calling the constructor (like Dog(fido, 3)); Python automatically handles it.
- Inside the constructor, you use self to refer to the current object's instance variables.

types of variable in python according to lifetime

1. Local Variables

Lifetime: Shortest lifetime. They exist only within the execution of a function or method.

Scope: Accessible only within the code block where they are defined (within the function or method body).

2. Global Variables

Lifetime: Entire Python script execution.

Scope: Accessible throughout the entire script.

3. Instance Variables

Lifetime: As long as the object (instance) exists.

Scope: Accessible within the object's methods and throughout the object's lifetime.

4.Class Variables

Lifetime: Throughout the class's existence (as long as the class is defined).

Scope: Shared by all instances of the class, accessed using the class name.

Instance Variables

Instance variables are owned by instances of the class. This means that for each object or instance of a class, the instance variables are different.

They are used to store data that is specific to each object created from the class. Each object has its own copy of the instance variables, meaning changes made to one object's instance variables do not affect others.

How to Define Instance Variables

Instance variables are typically defined within methods of a class, usually in the __init__ method, which is the initializer method that gets called when an object is created.

Here is how you can define and use instance variables in a Python class:

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

When we create a Shark object, we will have to define these variables, which are passed as parameters within the constructor method or another method.

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


new_shark = Shark("Sammy", 5)
```

How we can call to print instance variables?

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

new_shark = Shark("Sammy", 5)
print(new_shark.name)
print(new_shark.age)
```

Scope of Instance Variables

Instance Scope: Instance variables are accessible within the class methods via the self keyword. They are not shared among instances of the class.

Class Variables

Class variables are defined within the class construction. Because they are owned by the class itself, class variables are shared by all instances of the class.

They are defined within the class but outside any of the class methods.

They therefore will generally have the same value for every instance unless you are using the class variable to initialize a variable.

How to Define Class Variables

Class variables are defined directly within the class body. Here is an example how to define and use class variables:

```
class Shark:

    animal_type = "fish"
```

We can create an instance of the Shark class (we’ll call it new_shark) and print the variable by using dot notation:

```
class Shark:
    animal_type = "fish"


new_shark = Shark()
print(new_shark.animal_type)
```

Scope of Class Variables

Class Scope: Class variables are accessible within the class methods via the class name (MyClass.class_variable). They are also accessible through instances but should be accessed using the class name for clarity.

```
class Car:
    wheels = 4
    def __init__(self, brand):
        self.brand = brand
    def display(self):
        print(f'Brand: {self.brand},Wheels: {Car.wheels}')
car1 = Car('Toyota')
car2 = Car('Honda', 'Civic')
car1.display()
Car.wheels = 3
car1.display()
car2.display()
```

Instance Methods

Instance methods are functions defined within a class that operate on instances of that class.

If we use instance variables inside a method, such methods are called instance methods.

These methods can access and modify the object's state and are the most common type of method in object-oriented programming. They take at least one parameter, self, which refers to the instance upon which the method is called.

Defining Instance Methods

Instance methods are defined just like regular functions but within the class body. The first parameter of instance methods is always self, which refers to the object calling the method.

```
class Student:
    def __init__(self, name, age):
        # Instance variable
        self.name = name
        self.age = age
    def show(self):
        print('Name:', self.name, 'Age:', self.age)
```

Calling An Instance Method

We use an object and dot (.) operator to execute the block of code or action defined in the instance method.

```
s = Student("Jessa", 14)

s.showStudentDetails()
```

Example

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def introduce(self):
        print(f"Hello, I'm {self.name} and I'm {self.age} years old.")
person1 = Person("Alice", 25)
person2 = Person("Bob", 30)
# Calling the instance method
person1.introduce()
person2.introduce()
```

We can modify Instance Variables inside Instance Method

We can create Instance Variables in Instance Method

Class Method

Class methods are methods that are called on the class itself, not on a specific object instance.

Therefore, it belongs to a class level, and all class instances share a class method.

A class method is bound to the class and not the object of the class. It can access only class variables.

It can modify the class state by changing the value of a class variable that would apply across all the class objects.

In method implementation, if we use only class variables, we should declare such methods as class methods. The class method has a cls as the first parameter, which refers to the class.

The class method can be called using ClassName.method_name() as well as by using an object of the class.

Define Class Method

Any method we create in a class will automatically be created as an instance method.

We must explicitly tell Python that it is a class method using the @classmethod decorator or classmethod() function.

Class methods are defined inside a class, and it is pretty similar to defining a regular function.

Like, inside an instance method, we use the self keyword to access or modify the instance variables. Same inside the class method, we use the cls keyword as a first parameter to access class variables.

Therefore the class method gives us control of changing the class state.

Create Class Method Using @classmethod Decorator

To make a method as class method, add @classmethod decorator before the method definition, and add cls as the first parameter to the method.

```
class School:
    # class variable
    name = 'ABC School'
    @classmethod
    def school_name(cls):
        print('School Name is :', cls.name)
# call class method
School.school_name()
```

Create Class Method Using classmethod() function

Apart from a decorator, the built-in function classmethod() is used to convert a normal method into a class method.

The classmethod() is an inbuilt function in Python, which returns a class method for a given function.

Syntax:

```
classmethod(function)
```

- function: It is the name of the method you want to convert as a class method.
- It returns the converted class method.

Note: The method you want to convert as a class method must accept class (cls) as the first argument, just like an instance method receives the instance (self).

A classmethod() function is the older way to create the class method in Python. In a newer version of Python, we should use the @classmethod decorator to create a class method.

```
class School:
    name = 'ABC School'
    def school_name(cls):
        print('School Name is :', cls.name)
School.school_name = classmethod(School.school_name)
School.school_name()
```

```
class Student:
    school_name = 'ABC School'
    def __init__(self, name, age):
        self.name = name
        self.age = age
    @classmethod
    def change_school(cls, school_name):
        # class_name.class_variable
        cls.school_name = school_name
    # instance method
    def show(self):
        print(self.name, self.age, 'School:', Student.school_name)
jessa = Student('Jessa', 20)
jessa.show()
# change school_name
Student.change_school('XYZ School')
jessa.show()
```

Can we convert instance method to class method using classmethod()

No, you cannot convert an instance method to a class method directly using classmethod(). An instance method is designed to operate on individual instances of a class and requires access to instance-specific data via self. On the other hand, a class method is designed to operate on the class itself and uses cls to access class-level data.

Instance Method Example

```
class MyClass:
    def __init__(self, value):
        self.value = value

    def instance_method(self):
        return f'Instance value: {self.value}'
# Creating an instance and calling the instance method
obj = MyClass(10)
print(obj.instance_method())  # Output: Instance value: 10
```

Class Method Example

```
class MyClass:
    class_variable = 0
    def __init__(self, value):
        self.value = value
    def instance_method(self):
        return f'Instance value: {self.value}'
    @classmethod
    def class_method(cls):
        return f'Class variable: {cls.class_variable}'
# Using the class method
print(MyClass.class_method())  # Output: Class variable: 0
```

Can we provide same name to class variable and instance variable

Yes, we can because class variable have scope in class level where as instance variable in store in instance of that class, they both are stored in different memory.

However, doing so can lead to confusion and unintended behavior, so it is generally recommended to use different names to avoid ambiguity.

Static Method

A static method is a general utility method that performs a task in isolation. Static methods in Python are similar to those found in Java or C++.

A static method is bound to the class and not the object of the class. Therefore, we can call it using the class name.

A static method doesn’t have access to the class and instance variables because it does not receive an implicit first argument like self and cls. Therefore it cannot modify the state of the object or class.

The static method can be called using ClassName.method_name() as well as by using an object of the class.

```
class Employee:
    @staticmethod
    def sample(x):
        print('Inside static method', x)

# call static method
Employee.sample(10)

# can be called using object
emp = Employee()
emp.sample(10)
```

Define Static Method in Python

Any method we create in a class will automatically be created as an instance method. We must explicitly tell Python that it is a static method using the @staticmethod decorator or staticmethod() function.

Static methods are defined inside a class, and it is pretty similar to defining a regular function. To declare a static method, use this idiom:

```
class C:
    @staticmethod
    def f(arg1, arg2, ...): ...
```

Create Static Method Using @staticmethod Decorator

To make a method a static method, add @staticmethod decorator before the method definition.

The @staticmethod decorator is a built-in function decorator in Python to declare a method as a static method. It is an expression that gets evaluated after our function is defined.

```
class Calculator:

    @staticmethod
    def add(a, b):
        return a + b

# Using the static methods
result_add = Calculator.add(5, 3)
print(f'Result of addition: {result_add}')
```

The staticmethod() function

Some code might use the old method of defining a static method, using staticmethod() as a function rather than a decorator.

You should only use staticmethod() function to define static method if you have to support older versions of Python (2.2 and 2.3). Otherwise, it is recommended to use the @staticmethod decorator.

Syntax:

staticmethod(function)

- function: It is the name of the method you want to convert as a static method.
- It returns the converted static method.

```
class Employee:
    def sample(x):
        print('Inside static method', x)

# convert to static method
Employee.sample = staticmethod(Employee.sample)
# call static method
Employee.sample(10)
```

Call Static Method from Another Method

```
class Test :
    @staticmethod
    def static_method_1():
        print('static method 1')
    @staticmethod
    def static_method_2() :
        Test.static_method_1()
    @classmethod
    def class_method_1(cls) :
        cls.static_method_2()
# call class method
Test.class_method_1()
```

Method and Constructor Overloading

Method Overloading

Two or more methods have the same name but different numbers of parameters or different types of parameters, or both.

These methods are called overloaded methods and this is called method overloading.

```
# First sum method Takes two argument and print their sum
def sum(a, b):
    s = a + b
    print(s)
# Second sum method Takes three argument and print their sum
def sum(a, b, c):
    s = a + b + c
    print(s)
# Uncommenting the below line shows an error
# sum(4, 5)
# This line will call the second sum method
sum(4, 5, 5)
```

Python does not support method overloading like Java or C++. We may overload the methods, but we can only use the latest defined method.

We need to provide optional arguments or *args in order to provide a different number of arguments on calling.

What will be the output

```
class Addition:
  def add(self, a, b):
    x = a+b
    return x
  def add(self, a, b, c):
    x = a+b+c
    return x
obj = example()
print (obj.add(10,20,30))
print (obj.add(10,20))
```

The first call to add() method with three arguments is successful. However, calling add() method with two arguments as defined in the class fails.

The output tells you that Python considers only the latest definition of add() method, discarding the earlier definitions.

In python, unlike other languages, you cannot perform method overloading by using the same method name. Why?

Everything is an object in python, classes, and even methods.

Say you have an object Addition, which is a class (everything in python is an object, so the class Addition is an object too). It has an attribute called - “add”. It's the only attribute that the class Addition can have with that name.

So when you're writing def add(...): ... you have essentially created a object, which here is a method and you are assigning it to the "add" attribute of class Addition. If you write two definitions, then the second definition replaces the first one, just the way that assignment always behaves.

So…

Using Default Arguments

Instead, to perform the same function, i.e. to achieve method overloading, we create a single function that takes multiple parameters. After doing that, all you need to do is check the number of parameters taken as input.

```
class Addition:
  def add(self, a = None, b = None, c = None):
        s = 0
        if a != None and b != None and c != None:
                 s = a + b + c
           elif a != None and b != None:
                 s =  a + b
           return s
```

Using Variable-length Arguments (*args and **kwargs)

```
class Example:
    def display(self, *args):
        if len(args) == 2:
            print(f"Two arguments: a = {args[0]}, b = {args[1]}")
        elif len(args) == 1:
            print(f"One argument: a = {args[0]}")
        else:
            print("No arguments")
example = Example()
example.display()
example.display(10)
example.display(10, 20)
```

By Using Multiple Dispatch Decorator

Multiple Dispatch Decorator Can be installed by:

```
pip3 install multipledispatch
```

```
from multipledispatch import dispatch

@dispatch(int, int)
def product(first, second):
        result = first*second
        print(result)
@dispatch(int, int, int)
def product(first, second, third):
        result = first * second * third
        print(result)
@dispatch(float, float, float)
def product(first, second, third):
        result = first * second * third
        print(result)
product(2, 3)
product(2, 3, 2)
product(2.2, 3.4, 2.3)
```

Constructor Overloading

Python does not support constructor overloading. If you try to overload the constructor, the last implementation will be executed each time.

Any previous implementation will be over-written by the latest one.

Example Consider the following example.

In class Person two __init__() functions are provided, however the implementation of the first __init__() function is over-written by the later implementation.

```
class test:
    def __init__(self, a):
        self.num=a
    def __init__(self,a,b):
        self.num1=a
        self.num2=b
    def add(self):
        print("this is method")
        print(self.num1+self.num2)
obj1=test(1)
obj1.add()
```

Now, Python does not support constructor overloading, but that doesn't mean that the same behavior cannot be achieved using some strategy.

In this section, we will look at two such strategies to define multiple constructors in Python:

- Using default values and optional arguments in __init__() method.
- Checking data type of arguments passed in the __init__() method.

Both of these methods require only one constructor and implementation of the __init__() method.

Example Using Default Arguments

```
class Example:
    def __init__(self, a=None, b=None):
        if a is not None and b is not None:
            self.value = f"Two arguments: a = {a}, b = {b}"
        elif a is not None:
            self.value = f"One argument: a = {a}"
        else:
            self.value = "No arguments"

    def display(self):
        print(self.value)

example1 = Example()
example2 = Example(10)
example3 = Example(10, 20)

example1.display()  # No arguments
example2.display()  # One argument: a = 10
example3.display()  # Two arguments: a = 10, b = 20
```

Using Optional Arguments

```
class Series:
    def __init__(self, *args):
        if len(args) == 2:
            self.firstTerm, self.difference = args[0], args[1]
        elif len(args) == 1:
            self.firstTerm, self.difference = args[0], 0
        else:
            self.firstTerm, self.difference = 0, 0
    def NthTerm(self, N):
        return self.firstTerm + (N - 1) * self.difference
series1 = Series(10, 12)
series2 = Series(10)
series3 = Series(0)
```

Inheritance and its Types

Inheritance can be understood as a mechanism by which we can derive a new class from an existing class we can inherit all the methods, and attributes of the class.

The class from which we are deriving is known as the parent class and the class in which we are deriving the attributes and methods is known as the subclass or child class.

The main purpose of inheritance is to allow the user to code reuse and creator a more specialized class from a general class.

There is no boundation on the child class as it can add new functions or attributes and even it can override the existing ones.

By overriding the existing attributes the access to the parent class’s methods and attributes still remains.

To create a subclass in python, the child class is defined with the name of the parent class in parentheses this allows the child class to inherit all of the attributes and methods of the parent class.

From there the child class can modify or add new attributes and methods as needed. We can simply say that inheritance is a great way to reuse the code.

Syntax

```
class ParentClass:
    # Parent class definition
    pass

class ChildClass(ParentClass):
    # Child class definition
    pass
```

Example

```
class Animal:
    def __init__(self, name):
        self.name = name
    def speak(self):
        return "Some sound"
class Dog(Animal):
    def speak(self):
        return "Woof!"
    def info(self):
        super().info()
        print("I am a dog.")
class Cat(Animal):
    def speak(self):
        return "Meow!"
dog = Dog("Buddy")
cat = Cat("Whiskers")
dog.info()
print(dog.speak())
cat.info()
```

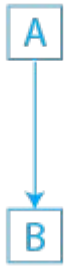
Types of Inheritance in Python

There are mainly 5 types of inheritance in python.

- Single Inheritance
- Multiple Inheritance
- Multilevel Inheritance
- Hierarchical Inheritance
- Hybrid Inheritance

Single Inheritance

Single inheritance can be referred to as the most basic inheritance of all types of inheritance in python. As the name suggests in this we derive a single base class from the single parent class or the existing class. The child class will take or inherit all the attributes of the parent class. It can also add its own methods and attributes or override the methods of the parent class.



Syntax of Single Inheritance in Python

```
class ParentClass:
    # Parent class definition


class ChildClass(ParentClass):
    # Child class definition
```

Example

```
# Parent Class
class Animal:
    def __init__(self, name):
        self.name = name
    def sound(self):
        pass
# Child Class
class Dog(Animal):
    def sound(self):
        return "Bark!"
dog = Dog("Buddy")
print(dog.name)
print(dog.sound())
```

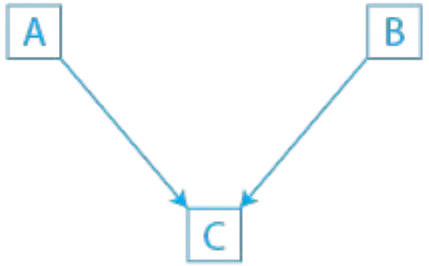
Multiple Inheritance

Multiple inheritance is a type of inheritance in which a new class is derived from two or more existing classes.

We can say that there is a single-child class and there are two or more parent classes.

The new class inherits all the attributes and methods of the parent classes.

The child class can access all the attributes and methods of both parent classes.



Syntax of Multiple Inheritance in Python

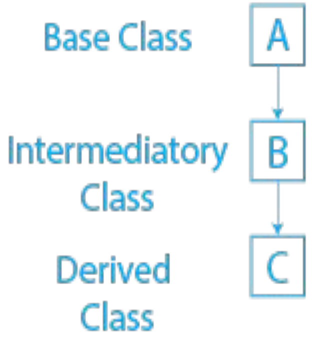
```
class ParentClass1:
    # Parent class 1 definition

class ParentClass2:
    # Parent class 2 definition

class ChildClass(ParentClass1, ParentClass2):
    # Child class definition
```

Example

```
class Person:
    def __init__(self, name):
        self.name = name
    def show_name(self):
        return self.name
class Employee:
    def __init__(self, salary):
        self.salary = salary
    def show_salary(self):
        return self.salary
class Manager(Person, Employee):
    def __init__(self, name, salary):
        Person.__init__(self, name)
        Employee.__init__(self, salary)
manager = Manager("John Doe", 5000)
print(manager.show_name())
print(manager.show_salary())
```

Multi-level Inheritance

Multi-level inheritance is a type of inheritance in which a new class is derived from a parent class, which in turn is derived from another parent class.



Syntax

```
class GrandParentClass:
    # Grandparent class definition

class ParentClass(GrandParentClass):
    # Parent class definition

class ChildClass(ParentClass):
    # Child class definition
```

Example

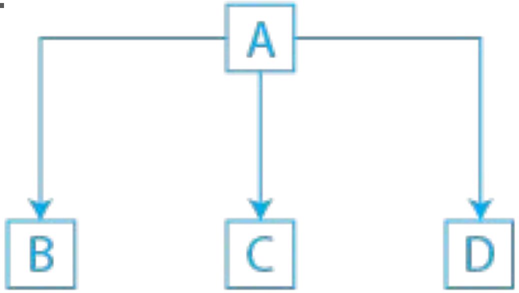
```
# Grandparent Class
class Animal:
    def __init__(self, name):
        self.name = name
    def show_name(self):
        return self.name
# Parent Class
class Dog(Animal):
    def sound(self):
        return "Bark!"
# Child Class
class Bulldog(Dog):
    def run(self):
        return "Running!"
bulldog = Bulldog("Buddy")
print(bulldog.show_name())
print(bulldog.sound())
print(bulldog.run())
```

Hierarchical Inheritance

Hierarchical inheritance can be referred to as type of inheritance in which a new class is derived from a single-parent class, and multiple child classes are derived from the same parent class.

All child classes inherit all the attributes and methods of the parent class.

The child classes can also add their attributes and methods or override the methods of the parent class.



Syntax

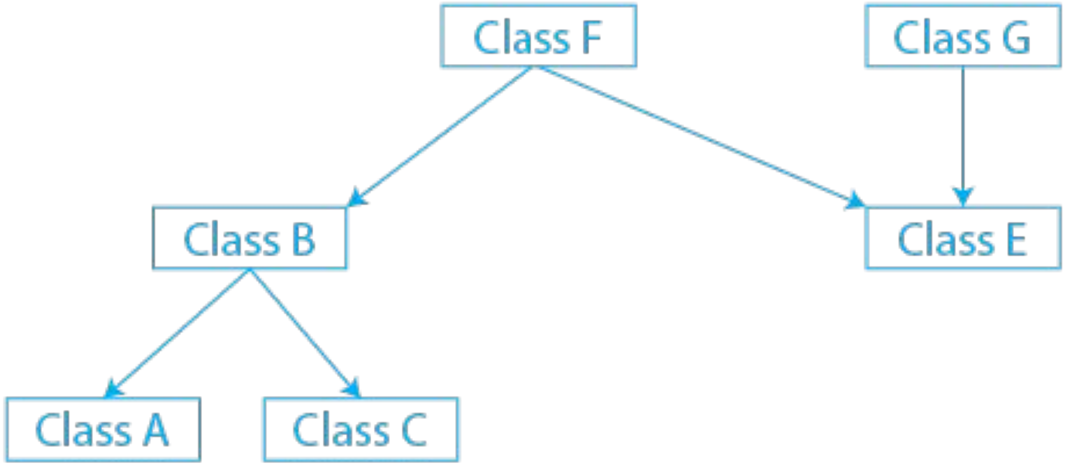
```
class ParentClass:
# Parent class definition

class ChildClass1(ParentClass):
# Child class 1 definition

class ChildClass2(ParentClass):
# Child class 2 definition
```

Example

```
# Parent Class
class Animal:
    def __init__(self, name):
        self.name = name
    def sound(self):
        pass
# Child Class 1
class Dog(Animal):
    def sound(self):
        return "Bark!"
# Child Class 2
class Cat(Animal):
    def sound(self):
        return "Meow!"
dog = Dog("Buddy")
cat = Cat("Mittens")
print(dog.name)
print(dog.sound())
print(cat.name)
print(cat.sound())
```

Hybrid Inheritance

Hybrid inheritance is a type of inheritance that combines two or more types of inheritance, such as single inheritance and multiple inheritances. The syntax for defining a child class with hybrid inheritance is similar to that of multiple inheritances.



Syntax

```
class ParentClass1:

# Parent class 1 definition


class ParentClass2:

# Parent class 2 definition


class ChildClass(ParentClass1, ParentClass2):

# Child class definition
```

Example

```
class A:
    def method_a(self):
        print("This is method A")
class B(A):
    def method_b(self):
        print("This is method B")
class C(A):
    def method_c(self):
        print("This is method C")
class D(B, C):
    def method_d(self):
        print("This is method D")
```

Self Study

method resolution order (MRO)-> very important to understand

Object Class

Dynamic Method Dispatch(DMD)

Can we call method defined in child class only using parent class instances

No, in Python, we cannot call a method defined in a child class using an instance of the parent class alone.

This is because the parent class does not have visibility or access to methods that are specifically defined in its child classes unless those methods are also defined or overridden in the parent class itself.

```
class Animal:
    def speak(self):
        return "Animal speaks"
class Dog(Animal):
    def speak(self):
        return "Dog barks"

    def fetch(self):
        return "Dog fetches ball"

animal = Animal()
dog = Dog()

# Calling methods using parent class instance
print(animal.speak())   # Output: Animal speaks
# The following line will raise an AttributeError because Animal class does not have a fetch
method
# print(animal.fetch())
```

What happen when we call method available in parent class only using both parent and child class instances

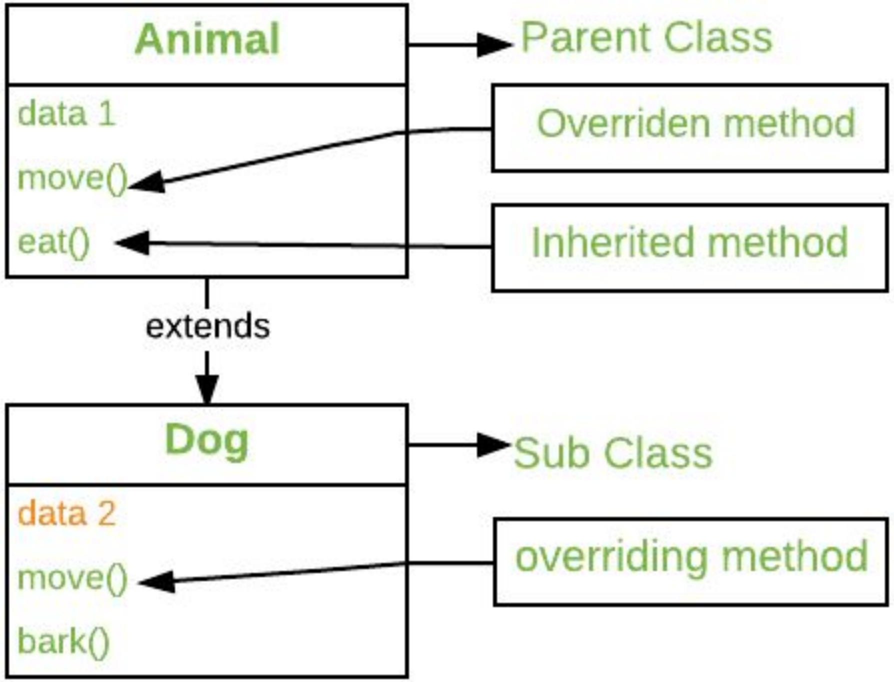
```
print(animal.speak())

print(dog.speak())
```

In this case both will execute the same method i.e parent class method will be executed.

Method Overriding

Method overriding is an ability of any object-oriented programming language that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its super-classes or parent classes.

When a method in a subclass has the same name, same parameters or signature and same return type(or sub-type) as a method in its super-class, then the method in the subclass is said to override the method in the super-class.


Animal
Parent Class
data 1
move()
eat()
Overriden method
Inherited method
extends
Dog
Sub Class
data 2
move()
bark()
overriding method

How Method Overriding Works

1. Definition in Parent Class:
    a. The parent class defines a method.
    b. The method in the parent class is meant to provide a general or default behavior.
2. Override in Child Class:
    a. The child class (subclass) provides a specific implementation of the method with the same name and parameters as the method in the parent class.
    b. This allows the subclass to customize or extend the behavior of the method defined in the superclass.
3. Calling the Method:
    a. When an instance of the subclass calls the overridden method, Python looks for the method in the subclass first.
    b. If found, Python executes the method in the subclass (even if the instance is referenced through the superclass).

Example

```
class Animal:
    def speak(self):
        return "Some generic sound"
# Child class
class Dog(Animal):
    def speak(self):
        return "Woof!"
# Another child class
class Cat(Animal):
    def speak(self):
        return "Meow!"
# Creating instances
dog = Dog()
cat = Cat()
# Calling overridden methods
print(dog.speak())  # Output: Woof!
print(cat.speak())  # Output: Meow!
```

Key Points

Inheritance: Method overriding is enabled through inheritance, where subclasses inherit methods from their superclass.

Same Method Signature: Overriding requires that the method in the subclass has the same method signature (name and parameters) as the method in the superclass.

Customization: Subclasses can provide specific implementations of methods defined in their superclass, allowing for customization and specialization of behavior.

You cannot reduce the scope of a method in method overriding compared to its definition in the superclass.

```
class Animal:
    def speak(self):
        print("Animal speaks")

class Dog(Animal):
    def __speak(self):  # Attempt to reduce scope
        print("Dog barks")

animal = Animal()
dog = Dog()

animal.speak()
dog.speak()
```

Output

```
Animal speaks

Animal speaks
```

If you attempt to reduce the scope of the speak method in the Dog class, such as making it private (__speak), it will not affect the overriding behavior.

Python still considers it an overriding method, and you can still access it through instances of Dog but parent class method will be executed

Access Modifiers

Access Modifiers

Access modifiers in Python refer to mechanisms that control the visibility and accessibility of attributes and methods within classes.

Unlike some other object-oriented programming languages like Java or C++, Python does not have explicit keywords (e.g., public, protected, private) to denote access modifiers.

Instead, Python relies on naming conventions and language features to achieve similar effects.

Visibility and Access Control in Python

1. **Public Access (No Modifier):**
   a. By default, all attributes and methods in Python are considered public.
   b. They can be accessed from within the class, from instances of the class, and from outside the class.

Example

```
 def public_method(self):

        return "Public Method"
```

2. **Protected Access (Convention with Single Underscore _)**
   a. The members of a class that are declared protected are only accessible to a class derived from it.
   b. Data members of a class are declared protected by adding a single underscore '_' symbol before the data member of that class.

Eg:

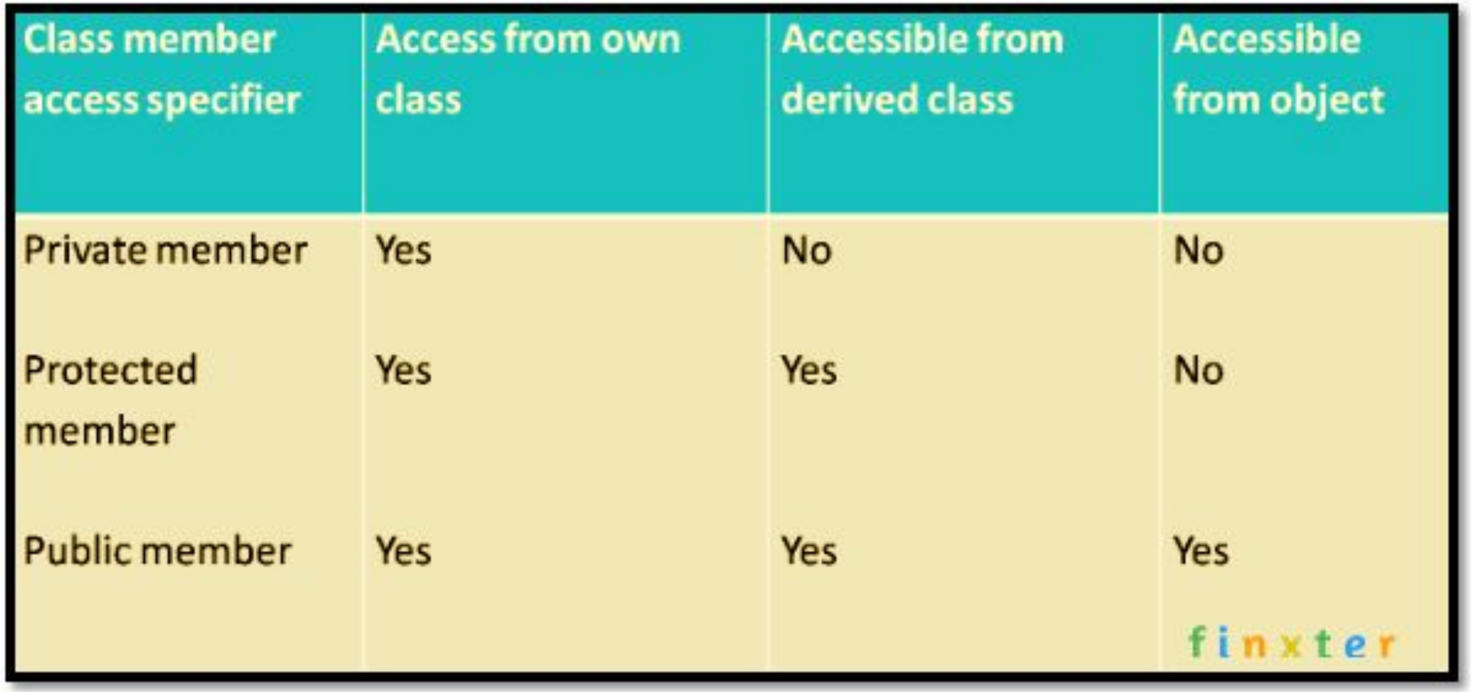
```
_name = None

_roll = None

def _displayRollAndBranch(self):

    print("Roll: ", self._roll)
```

3. Private Access (Convention with Double Underscore __)

a. The members of a class that are declared private are accessible within the class only,
b. Data members of a class are declared private by adding a double underscore '__' symbol before the data member of that class.

Eg:

```
__name = None

__roll = None

def __displayDetails(self):

    print("Name: ", self.__name)

    print("Roll: ", self.__roll)
```



Class member access specifier	Access from own class	Accessible from derived class	Accessible from object
Private member	Yes	No	No
Protected member	Yes	Yes	No
Public member	Yes	Yes	Yes

Abstract Class

Abstraction

Abstraction is the concept in object-oriented programming that is used to hide the internal functionality of the classes from the users.

Abstraction is implemented using the abstract classes.

Abstract Class

An abstract class in Python is a class that is designed to be a blueprint for other classes.

It cannot be instantiated on its own and typically includes one or more abstract methods.

Abstract methods are methods that are declared but contain no implementation.

The primary purpose of abstract classes is to define a common interface for a group of related classes.

Purpose of Abstract Classes

Common Interface: Abstract classes provide a way to enforce a common interface for all its subclasses. This ensures that certain methods are implemented in all subclasses.

Code Reusability: Abstract classes can contain common code that can be reused by subclasses.

Design Patterns: They are often used in design patterns, such as the Template Method Pattern, where the abstract class defines the structure and common behavior, while subclasses provide specific implementations.

Creating Abstract Classes in Python

In Python, abstract classes are created using the abc module. The abc module stands for Abstract Base Classes and provides the necessary tools to define abstract classes and methods.

Step-by-Step Implementation

1. Import the abc module:

Eg:

```
from abc import ABC, abstractmethod
```

2. Define an abstract class:
    a. Inherit from ABC, which stands for Abstract Base Class.
    b. Use the @abstractmethod decorator to declare abstract methods.

```
class Animal(ABC):
    @abstractmethod
    def sound(self):
        pass
   @abstractmethod
    def move(self):
        pass
```

3. Define concrete subclasses:
    a. Subclasses must implement all abstract methods.
    b. If a subclass does not implement all abstract methods, it will also be considered abstract and cannot be instantiated.

```
class Dog(Animal):
    def sound(self):
        return "Bark"

    def move(self):
        return "Run"

class Bird(Animal):
    def sound(self):
        return "Chirp"

    def move(self):
        return "Fly"
```

4. Instantiate the subclasses:

```
dog = Dog()
bird = Bird()

print(dog.sound())
print(dog.move())
print(bird.sound())
print(bird.move())
```

Key Points to Remember

Instantiation: You cannot create an instance of an abstract class directly. Attempting to do so will raise a TypeError.

Abstract Methods: Methods in the abstract class that are marked with the @abstractmethod decorator must be implemented by any concrete subclass. If they are not, the subclass will also be considered abstract.

Concrete Methods: Abstract classes can also contain concrete methods (methods with implementation). Subclasses can use these methods directly or override them as needed.

What will be the output

```
from abc import ABC, abstractmethod
class Vehicle(ABC):
    @abstractmethod
    def start_engine(self):
        pass
    def stop_engine(self):
        print("Engine stopped")
class Car(Vehicle):
    def start_engine(self):
        print("Car engine started")
# Create an instance of Car
car = Car()
car.start_engine()
car.stop_engine()
```

Output

```
Car engine started

Engine stopped
```

What will be the output

```
from abc import ABC, abstractmethod
class Vehicle(ABC):
    @abstractmethod
    def start_engine(self):
        pass
    @abstractmethod
    def stop_engine(self):
       pass
class Car(Vehicle):
    def start_engine(self):
        print("Car engine started")
def stop_engine(self):
        print("Engine stopped")
car = Car()
car.start_engine()
car.stop_engine()
```

Output

```
Car engine started

Engine stopped
```

What will be the output

```
from abc import ABC, abstractmethod
class Vehicle(ABC):
    @abstractmethod
    def start_engine(self):
        pass
    @abstractmethod
    def stop_engine(self):
       pass
class Car(Vehicle):
    def start_engine(self):
        print("Car engine started")
car = Car()
car.start_engine()
car.stop_engine()
```

Output

Error, because we cannot instantiate abstract class

Operator Overloading

Operator overloading in Python refers to the ability to define custom behavior for standard operators (like +, -, *, etc.) when they are used with user-defined classes.

This allows objects of custom classes to interact using these operators in a manner that is natural and intuitive.

Such as, we use the "+" operator for adding two integers as well as joining two strings or merging two lists.

We can achieve this as the "+" operator is overloaded by the "int" class and "str" class.

The user can notice that the same inbuilt operator or function is showing different behaviour for objects of different classes. This process is known as operator overloading.

Example

```
print (14 + 32)


# Now, we will concatenate the two strings

print ("NCCS" + "COLLEGE")

# Concatenate 2 lists

a = ['A', 'B'] + [1, 2, 3]

# We will check the product of two numbers

print (23 * 14)

# Here, we will try to repeat the String

print ("X Y Z " * 3)
```

The phenomenon of adding alternate/different meaning to an action done by an operator beyond their predefined operational role is known as operator overloading.

The process of utilizing an operator in different ways depending on the operands is known as operator overloading.

In Python, you can change how an operator behaves with different data types. The operators are methods defined in their respective classes. Operator overloading is the process of defining methods for operators.

So, what happens when we utilize them with user-defined class objects?

Example

```
class Circle:
    def __init__(self, radius):
        self.__radius = radius

c1 = Circle(2)
c2 = Circle(8)
c3 = c1 + c2
print(c3)
```

```
Traceback (most recent call last):
  File "", line 9, in
TypeError: unsupported operand type(s) for +: 'Circle' and 'Circle'
```

We can see that a TypeError was triggered because Python didn’t know how to combine two Circle objects. However, we may accomplish this work in Python by using operator overloading. But first, let’s talk about special functions.

Python Special Functions

Python uses special methods (also known as magic methods or dunder methods) to enable operator overloading.

These methods have double underscores at the beginning and end of their names. Each special method corresponds to a particular operator.

They are called "double underscore" functions because they have a double underscore prefix and suffix, such as __init__() or __add__().

__init__() function is invoked whenever a new object of that class is created.

Overloading the + Operator

Let’s say, we have two objects that are physical representations of a class (user-defined data type).

If we try to add two objects using the binary ‘+’ operator, the compiler throws an error since the compiler does not know how to add two objects.

So we define a method for an operator, which is referred to as operator overloading. We can overload all existing operators but not build new ones. The method’s name should start and conclude with a double underscore (__).

To overload the + operator, we must include the __add__() function in the class. Within this function, we can do whatever we want.

```
class Circle:
    def __init__(self, radius):
        self.radius = radius
    def __str__(self):
        return 'Radius of the circle is: {}'.format(self.radius)
    def __add__(self, other_circle):
        return Circle( self.radius + other_circle.radius )
c1 = Circle(2)
c2 = Circle(6)
c3 = c1 + c2
print(c3)
```

In the above example, we added the __add__() function, which allows us to use the + operator to combine two circle objects.

We create a new object and return it to the caller within the __add__() method.

What actually happens is that, when you use c1 + c2, Python calls c1.__add__(c2) which in turn is Circle.__add__(c1,c2).

After this, the addition operation is carried out the way we specified.

Operator	Expression	Special Function
Addition	c1 + c2	c1.__add__(c2)
Subtraction	c1 – c2	c1.__sub__(c2)
Multiplication	c1 * c2	c1.__mul__(c2)
Power	c1 ** c2	c1.__pow__(c2)
Division	c1 / c2	c1.__truediv__(c2)
Floor Division	c1 // c2	c1.__floordiv__(c2)
Remainder	c1 % c2	c1.__mod__(c2)
Bitwise Left Shift	c1 << c2	c1.__lshift__(c2)
Bitwise Right Shift	c1 >> c2	c1.__rshift__(c2)
Bitwise AND	c1 & c2	c1.__and__(c2)
Bitwise OR	c1 | c2	c1.__or__(c2)

Magic or Dunder Methods

Magic methods in Python are the special methods that start and end with the double underscores. They are also called dunder methods.

Magic methods are not meant to be invoked directly by you, but the invocation happens internally from the class on a certain action. For example, when you add two numbers using the + operator, internally, the __add__() method will be called.

Built-in classes in Python define many magic methods. Use the dir() function to see the number of magic methods inherited by a class.

```
print(dir(int))

#output:

#['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__', '__delattr__',

#'__dir__', '__divmod__', '__doc__', '__eq__', '__float__', '__floor__', '__floordiv__',

#'__format__', '__ge__', '__getattribute__', '__getnewargs__', '__gt__', '__hash__',

#'__index__', '__init__', '__init_subclass__', '__int__', '__invert__', '__le__',

#'__lshift__', '__lt__', '__mod__', '__mul__', '__ne__', '__neg__', '__new__', '__or__',

#'__pos__', '__pow__', '__radd__', '__rand__', '__rdivmod__', '__reduce__', '__reduce_ex__',

#'__repr__', '__rfloordiv__', '__rlshift__', '__rmod__', '__rmul__', '__ror__', '__round__',

#'__rpow__', '__rrshift__', '__rshift__', '__rsub__', '__rtruediv__', '__rxor__', '__setattr__',

#'__sizeof__', '__str__', '__sub__', '__subclasshook__', '__truediv__', '__trunc__', '__xor__', 'bit_length',

#'conjugate', 'denominator', 'from_bytes', 'imag', 'numerator', 'real', 'to_bytes']
```

For example, the __add__ method is a magic method which gets called when we add two numbers using the + operator.

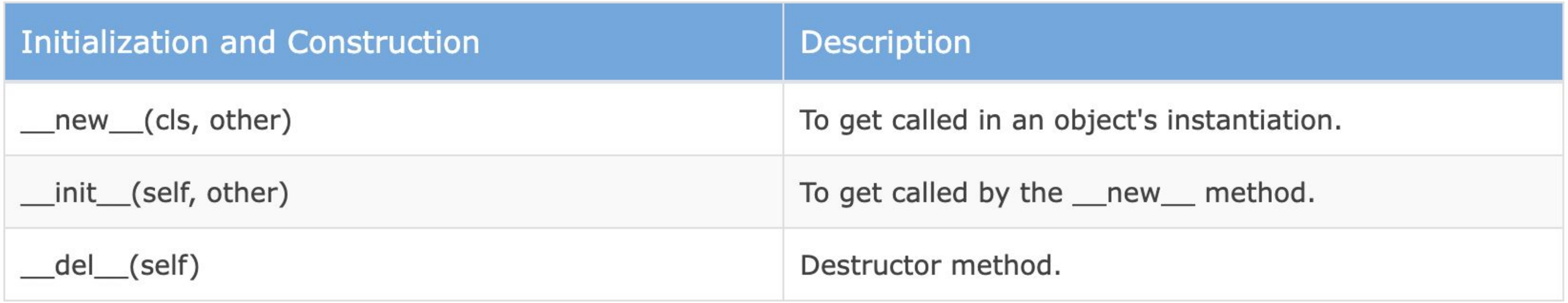
```
num=10

res = num.__add__(5)

print(res)
```

__new__() method

Python the __new__() magic method is implicitly called before the __init__() method. The __new__() method returns a new object, which is then initialized by __init__().

```
class Employee:
    def __new__(cls):
        print ("__new__ magic method is called")
        inst = object.__new__(cls)
                return inst
    def __init__(self):
        print ("__init__ magic method is called")
        self.name='NCCS'
```

Important Magic Methods



Initialization and Construction	Description
__new__(cls, other)	To get called in an object's instantiation.
__init__(self, other)	To get called by the __new__ method.
__del__(self)	Destructor method.

Unary operators and functions	Description
__pos__(self)	To get called for unary positive e.g. +someobject.
__neg__(self)	To get called for unary negative e.g. -someobject.
__abs__(self)	To get called by built-in abs() function.
__invert__(self)	To get called for inversion using the ~ operator.
__round__(self,n)	To get called by built-in round() function.
__floor__(self)	To get called by built-in math.floor() function.
__ceil__(self)	To get called by built-in math.ceil() function.
__trunc__(self)	To get called by built-in math.trunc() function.

Augmented Assignment	Description
__iadd__(self, other)	To get called on addition with assignment e.g. a +=b.
__isub__(self, other)	To get called on subtraction with assignment e.g. a -=b.
__imul__(self, other)	To get called on multiplication with assignment e.g. a *=b.
__ifloordiv__(self, other)	To get called on integer division with assignment e.g. a //=b.
__idiv__(self, other)	To get called on division with assignment e.g. a /=b.
__itruediv__(self, other)	To get called on true division with assignment
__imod__(self, other)	To get called on modulo with assignment e.g. a%=b.
__ipow__(self, other)	To get called on exponentswith assignment e.g. a **=b.
__ilshift__(self, other)	To get called on left bitwise shift with assignment e.g. a<<=b.
__irshift__(self, other)	To get called on right bitwise shift with assignment e.g. a >>=b.
__iand__(self, other)	To get called on bitwise AND with assignment e.g. a&=b.
__ior__(self, other)	To get called on bitwise OR with assignment e.g. a|=b.
__ixor__(self, other)	To get called on bitwise XOR with assignment e.g. a ^=b.

Type Conversion Magic Methods	Description
__int__(self)	To get called by built-int int() method to convert a type to an int.
__float__(self)	To get called by built-int float() method to convert a type to float.
__complex__(self)	To get called by built-int complex() method to convert a type to complex.
__oct__(self)	To get called by built-int oct() method to convert a type to octal.
__hex__(self)	To get called by built-int hex() method to convert a type to hexadecimal.
__index__(self)	To get called on type conversion to an int when the object is used in a slice expression.
__trunc__(self)	To get called from math.trunc() method.

String Magic Methods	Description
__str__(self)	To get called by built-int str() method to return a string representation of a type.
__repr__(self)	To get called by built-int repr() method to return a machine readable representation of a type.
__unicode__(self)	To get called by built-int unicode() method to return an unicode string of a type.
__format__(self, formatstr)	To get called by built-int string.format() method to return a new style of string.
__hash__(self)	To get called by built-int hash() method to return an integer.
__nonzero__(self)	To get called by built-int bool() method to return True or False.
__dir__(self)	To get called by built-int dir() method to return a list of attributes of a class.
__sizeof__(self)	To get called by built-int sys.getsizeof() method to return the size of an object.

Operator Magic Methods	Description
__add__(self, other)	To get called on add operation using + operator
__sub__(self, other)	To get called on subtraction operation using - operator.
__mul__(self, other)	To get called on multiplication operation using * operator.
__floordiv__(self, other)	To get called on floor division operation using // operator.
__truediv__(self, other)	To get called on division operation using / operator.
__mod__(self, other)	To get called on modulo operation using % operator.
__pow__(self, other[, modulo])	To get called on calculating the power using ** operator.
__lt__(self, other)	To get called on comparison using < operator.
__le__(self, other)	To get called on comparison using <= operator.
__eq__(self, other)	To get called on comparison using == operator.
__ne__(self, other)	To get called on comparison using != operator.
__ge__(self, other)	To get called on comparison using >= operator.

Exception Handling

Introduction

An unwanted or unexpected events that disturb the normal flow of program is called an exception.

Example:

- SleepingException
- TyrePunchuredException
- ArithmeticError
- OverflowError ...etc

Notes

When an Exception occurs the normal flow of the program is disrupted and the program/Application terminates abnormally, which is not recommended, therefore, these exceptions are to be handled.

Purpose of Exception Handling

- It is highly recommended to handle exceptions. The main objective of exception handling is graceful (normal) termination of the program.

What is the meaning of exception handling?

- Exception handling doesn't mean repairing an exception. We have to define alternative way to continue rest of the program normally. This way of defining alternative is nothing but exception handling.
- Exceptions hinder normal execution of program. Exception handling is the process of handling exceptions in such a way that they do not hinder normal execution of the program.

Example:

Suppose our programming requirement is to read data from remote file locating at London. At runtime if London file is not available then our program should not be terminated abnormally.

We have to provide a local file to continue rest of the program normally. This way of defining alternative is nothing but exception handling.

```
try:
    # Code that might raise an exception
    # read data from London file
    risky_code()
except FileNotFoundError:
    # Code that runs if the exception occurs
    handle_exception(e)
else:
    # Code that runs if no exception occurs
    run_if_no_exception()
```

Error Vs Exception

Errors:

Broad category of problems that prevent your program from functioning correctly.

Can occur during various stages:

Syntax Errors: These are the most basic errors, occurring during the compilation phase when the Python interpreter checks your code's structure. They involve typos, missing punctuation (colons, parentheses), or incorrect syntax. Examples include a missing colon after a function definition or an unmatched parenthesis.

Logical Errors: These are flaws in your program's logic that might not be apparent until runtime. They often stem from mistakes in your problem-solving approach or incorrect assumptions. Examples include infinite loops due to missing increments, incorrect calculations, or unintended side effects.

Runtime Errors: These are unexpected events that happen during program execution, often due to external factors or code issues that surface only when the program runs. These can be further categorized:

Exceptions:Exceptions are a specific type of runtime error that disrupts the normal flow of your program's execution. They represent conditions that you can potentially handle and recover from within your code.

Other runtime errors (e.g., MemoryError if you run out of memory, SystemExit if you use sys.exit()).

Exceptions:

A specific type of runtime error that interrupts the normal flow of your program.

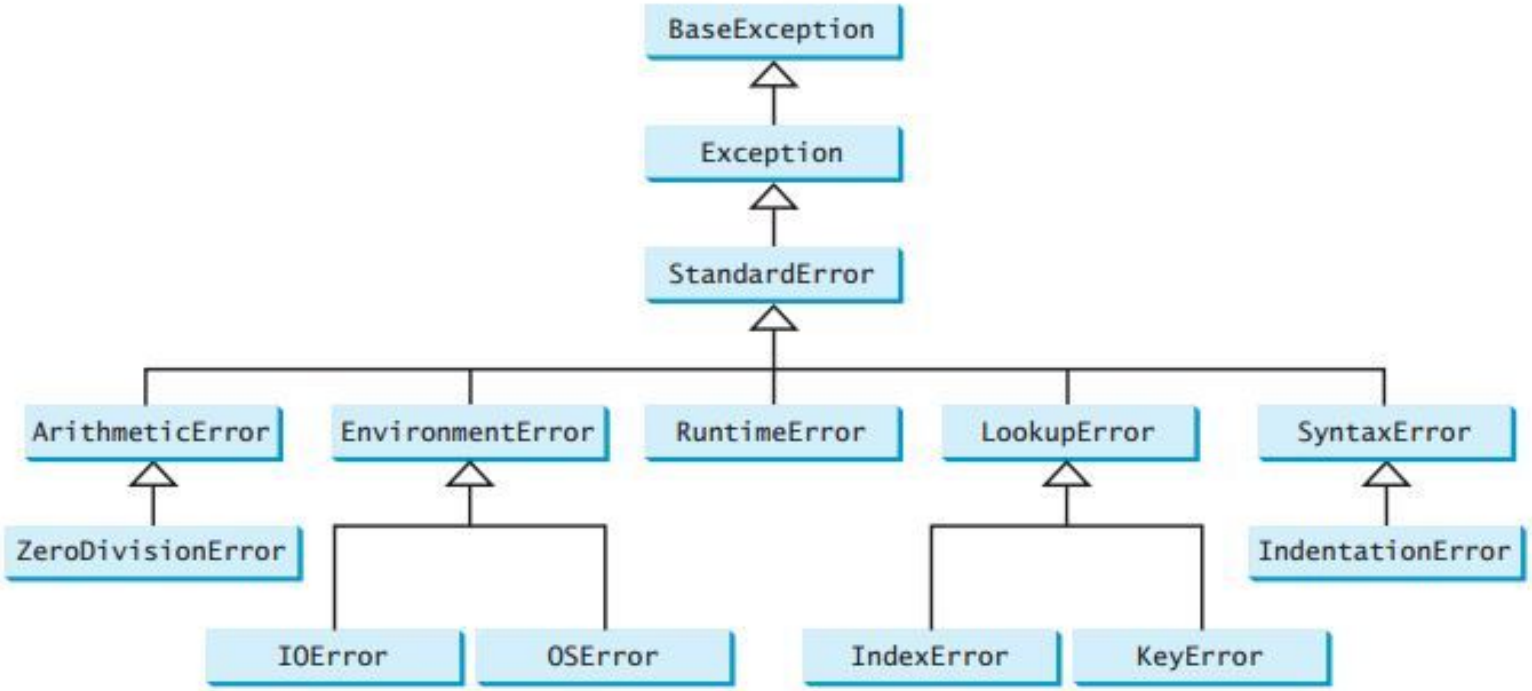
Represent conditions that you can potentially handle and recover from within your code.

Inherit from the base class BaseException (or its subclass Exception).

Built-in exceptions (like ZeroDivisionError, TypeError) provide informative messages about the issue.

Exception Hierarchy

- In Python, all exceptions are derived from the BaseException class. This means that every exception in Python is an instance of BaseException or one of its subclasses.
- The BaseException class is the top-level class in the exception hierarchy. It provides some common methods that all exceptions can use, such as __str__ and __repr__. However, you should not use BaseException directly in your code as it is too broad and can catch any type of exception.
- Instead, you should use more specific exception classes that are subclasses of BaseException. Python provides a number of built-in exception classes that you can use, and you can also create your own custom exception classes.


BaseException
Exception
StandardError
ArithmeticError
EnvironmentError
RuntimeError
LookupError
SyntaxError
ZeroDivisionError
IndentationError
IOError
OSError
IndexError
KeyError

Default Exception Handling in Java

When an error condition occurs within a function, an exception object is raised using the raise keyword.

The Python interpreter doesn't find any try-except block surrounding the code where the exception occurred.

Since there's no except block to catch the exception, the interpreter starts propagating (moving) the exception up the call stack. This means it looks for a try-except block in the function that called the function where the exception was raised, and so on.

If the exception continues to propagate up the call stack without encountering a matching try-except block, the program reaches a point where there are no more functions to call. At this stage, the program terminates abnormally (crashes).

Upon program termination due to an unhandled exception, the interpreter prints a traceback to the console.

Note: This exception object carries information about the error, including its type (e.g., ZeroDivisionError) and often a message describing the issue.

Information typically included in a traceback

- The first line of the traceback indicates the type of exception that was raised (e.g., ZeroDivisionError, IndexError, FileNotFoundError).
- Often, the exception type is accompanied by a specific error message.
- Each subsequent line in the traceback details a function call that led to the exception.
- In some traceback formats, you might also see the name of the function that was called and the arguments that were passed to it.

Example:

```
Traceback (most recent call last):
  File "my_script.py", line 10, in divide
    result = a / b
ZeroDivisionError: division by zero
```

How Programmer handles an exception?

try: The block of code to be tested for errors while it is being executed.

except: The block of code that will be executed if an error occurs in the try block.

else: The block of code to be executed if no errors occur in the try block.

finally: The block of code that will be executed regardless of whether an error occurs or not.

```
try:
    # Code that might raise an exception
    risky_code()
except SomeException as e:
    # Code that runs if the exception occurs
    handle_exception(e)
else:
    # Code that runs if no exception occurs
    run_if_no_exception()
finally:
    # Code that runs no matter what
    always_run_this()
```

Example

```
try:
    file = open('non_existent_file.txt', 'r')
    content = file.read()
except FileNotFoundError as e:
    print(f"Error: {e}")
else:
    print("File read successfully")
    file.close()
finally:
    print("Executing finally block")
```

Example: Exception Handling Using try...except

```
try:
    numerator = 10
    denominator = 0
    result = numerator/denominator
    print(result)
except:
    print("Error: Denominator cannot be 0.")

# Output: Error: Denominator cannot be 0.
```

Catching Specific Exceptions in Python

For each try block, there can be zero or more except blocks. Multiple except blocks allow us to handle each exception differently.

The argument type of each except block indicates the type of exception that can be handled by it.

```
try:
    even_numbers = [2,4,6,8]
    print(even_numbers[5])

except ZeroDivisionError:
    print("Denominator cannot be 0.")
except IndexError:
    print("Index Out of Bound.")

# Output: Index Out of Bound
```

Python try with else clause

In some situations, we might want to run a certain block of code if the code block inside try runs without any errors.

For these cases, you can use the optional else keyword with the try statement.

```
# program to print the reciprocal of even numbers

try:
    num = int(input("Enter a number: "))
    assert num % 2 == 0
except:
    print("Not an even number!")
else:
    reciprocal = 1/num
    print(reciprocal)
```

Output

If we pass an odd number:

```
Enter a number: 1

Not an even number!
```

if we pass 0, we get ZeroDivisionError

Enter a number: 0

```
Traceback (most recent call last):

  File "<string>", line 7, in <module>

    reciprocal = 1/num

ZeroDivisionError: division by zero
```

Note: Exceptions in the else clause are not handled by the preceding except clauses.

Python try...finally

In Python, the finally block is always executed no matter whether there is an exception or not.

The finally block is optional. And, for each try block, there can be only one finally block.

```
try:
    numerator = 10
    denominator = 0
    result = numerator/denominator
    print(result)
except:
    print("Error: Denominator cannot be 0.")

finally:
    print("This is finally block.")
```

Output

```
Error: Denominator cannot be 0.

This is finally block.
```

Multiple Exceptions

Handling multiple exceptions in Python allows you to specify different behaviors for different types of errors. You can catch multiple exceptions by using multiple except blocks. This way, you can handle each specific type of exception differently.

```
try:
    risky_code()
except ExceptionType1:
    # Handle ExceptionType1
except ExceptionType2:
    # Handle ExceptionType2
except ExceptionTypeN:
    # Handle ExceptionTypeN
else:
    run_if_no_exception()
finally:
    # Code that runs no matter what
```

```
try:
    file = open('non_existent_file.txt', 'r')
    result = 10 / 0
except FileNotFoundError:
    print("Error: File not found.")
except ZeroDivisionError:
    print("Error: Division by zero is not allowed.")
except Exception as e:
    print(f"An unexpected error occurred: {e}")
else:
    print("File read successfully and division performed.")
    file.close()
finally:
    print("Executing finally block")
```

Catching Multiple Exceptions in One except Block

```
try:
    # Trying to open a file and divide by zero
    file = open('non_existent_file.txt', 'r')
    result = 10 / 0
except (FileNotFoundError, ZeroDivisionError) as e:
    print(f"An error occurred: {e}")
except Exception as e:
    print(f"An unexpected error occurred: {e}")
else:
    print("File read successfully and division performed.")
    file.close()
finally:
    print("Executing finally block")
```

Creating a custom exception

Creating a custom exception in Python involves defining a new class that inherits from the built-in Exception class (or any of its subclasses). This allows you to add custom behavior or attributes to your exception.

Step 1: Define the Custom Exception Class

The first step in creating a custom exception is to define a new class that inherits from Python’s built-in Exception class.

```
class MyCustomException(Exception):
    pass
```

Step 2: Add Custom Behavior or Attributes(optional)

While not strictly mandatory, a constructor (__init__()) method allows you to customize the behavior of your exception when it's raised. You can optionally define arguments to capture additional error details.

```
class InvalidAgeException(Exception):
    def __init__(self, age):
        self.age = age
        super().__init__(f"Invalid age: {age}")  # Call base class constructor
```

Raise Your Custom Exception:

Use the raise keyword to raise your custom exception within your code when an error condition is encountered. You can optionally pass arguments to the constructor if you defined one.

```
def check_age(age):
    if age < 18:
        raise InvalidAgeException(age)
```

Using try...except for Exception Handling:

Once you've created your custom exception, you can leverage Python's try...except block to handle it along with other potential exceptions:

```
try:
   check_age(15)
except InvalidAgeException as e:
   print(f"Error: {e}")  # Access the error message using the exception object
else:
   print("Age is valid.")
```

Modules and Packages

Modular programming refers to the process of breaking a large, unwieldy programming task into separate, smaller, more manageable subtasks or modules.

Individual modules can then be cobbled together like building blocks to create a larger application.

What is Module in Python?

A module in Python is a single file that contains Python code in the form of functions, executable statements, variables, and classes with having a .py extension file.

A module acts as a self-contained unit of code that can be imported and used in other programs or modules.

What is package in python?

A package, on the other hand, is a collection of modules organized in a directory. Packages allow us to group multiple related modules together under a common namespace, making it easier to organize and structure our code base.

Working with Modules

Modules can be imported and used in other programs, modules, and packages. They’re very beneficial in an application, since they break down the application function into smaller, manageable, and logical units.

For instance, say we want to create a web application: the application is going to need code for connecting to a database, code for creating database models, code that’s going to be executed when a user visits a certain route, and so on.

We can put all the code in one file, but then the code very quickly becomes unmaintainable and unreadable. By using modules, we can break down the code into units that are more manageable. We’ll put all the code needed to connect to the database in one file, code for database models is put in another file, and code for the routes into a module. Breaking the code down into those modules promotes organization, reusability, and maintainability.

Creating a simple module

To create a module in Python, open up an IDE or text editor, create a file, and give it a descriptive name and a .py extension.

For this example, let’s call it sample.py and enter in the following code:

Example

```
# sample.py

# create a variable in the module
sample_variable  = "This is a string variable in the sample.py module"

# A function in the module
def say_hello(name):
  return f"Hello, {name}  welcome to this simple module."

# This is another function in the module
def add(a, b):
  return f"The sum of {a} + {b} is = {a+b}"

print(sample_variable)
print(say_hello("KAKASHI"))
print(add(2, 3))
```

While modules are meant to be used in other parts of the program or an application, we can run them independently.

To run this module, we need to have Python installed in our development environment. We can run it on the terminal using the following command:

```
python sample.py
```

Or we can use the following command:

```
python3 sample.py
```

This will return the following output:

```
This is a string variable in the sample.py module

Hello, kabaki welcome to this simple module.

The sum of 2 + 3 is = 5
```

we can run it as a standalone, but most modules are made to be used in other modules or other parts of a Python program.

So to use variables, functions, and classes from one module in another module we have to import the module.

There are different ways of importing modules, so let’s look at them.

Using the import statement

We can use the import statement to make the contents of one module available for use in another module. Consider our sample.py from above: to use its contents in another module, we just import it:

```
import sample

print(sample.sample_variable)
print(sample.say_hello(“John”))
print(sample.add(2, 3))
```

Using the from keyword

We can also use the from keyword to import specific functions or variables. Say a module has a large number of functions and variables defined in it and we don’t want to use all of them. We can specify the functions or variables we want to use, using the from keyword:

```
from sample import add

print(add(10, 4))
```

Using as

We can use as to provide an alias or an alternate name for the module.

```
import sample as sp

result = sp.add(5, 5)
print(result)
print(sp.say_hello("KAKASHI"))
```

Introducing Packages

A package in Python is a way of organizing related modules into a directory. This provides a better way of organizing code, enabling us to group modules that serve a common purpose or are part of the same component.

Packages are particularly beneficial when structuring larger projects or libraries. For instance, consider the case of a web application where we have code for different database models, views, and utilities.

It would make a lot of sense if we created a models package with different modules for the different models in an application.

Building and managing packages

While packages organize related code modules in one directory, just putting the modules in a directory doesn’t make it a package.

For Python to identify a directory as a package or a subpackage, the directory must contain a special file named __init__.py.

This file notifies Python that the directory containing it should be treated as a package or a subpackage.

This file could be empty, and most of the time it is, but it can also contain initialization code, and it plays a vital role in Python’s package structure and import mechanisms. So using __init__.py tells Python that we are intentionally creating a package, thereby helping it differentiate between a package and an ordinary directory.

Packages can have a hierarchical structure, meaning we can create subpackages within our packages to further organize our code.

```
my_package/
├── __init__.py
├── module1.py
└── subpackage/
    ├── __init__.py
    ├── submodule1.py
    └── submodule2.py
```

This diagram shows my_package is the main package, and subpackage is a subpackage within it. Both directories have an __init__.py file. Using this kind of structure helps us organize our code into a meaningful hierarchy.

Creating packages and subpackages

To create a package, we first create a directory that's going to contain our modules. Then we create an __init__.py file. Then we create our modules in it, along with any subpackages.

Say we're building a calculator application: let's create a package for various calculations, so create a directory in our terminal or our IDE and name it calculator.

In the directory, create the __init__.py file, then create some modules. Let's create three modules, add.py, subtract.py, and multiply.py. In the end, we'll have a directory structure similar to this:

```
calculator/
├── __init__.py
├── add.py
├── subtract.py
└── multiply.py
```

Let’s put some samples in those files. Open the add.py module and put in the following code:

```
# add.py
def add(a, b):
  """
  Adds two numbers and returns the result.
  :param a: First number.
  :param b: Second number.
  :return: Sum of a and b.
  """
  return a + b
```

```
# subtract.py
def subtract(a, b):
  """
  Subtracts two numbers and returns the result.
  :param a: First number.
  :param b: Second number.
  :return: Difference of a and b.
  """
  return a - b
```

Importing from packages

To import modules from packages or subpackages, there are two main ways. We can either use a relative import or an absolute import.

Absolute imports

Absolute imports are used to directly import modules or subpackages from the top-level package, where we specify the full path to the module or package we want to import.

Here’s an example of importing the add module from the calculator package:

```
# calculate.py

from calculator.add import add
result = add(5, 9)
print(result)
```

The above example shows an external module — calculate.py — that imports the add() function from the add module using an absolute import by specifying the absolute path to the function.

Relative imports

Relative imports are used to import modules or packages relative to the current module’s position in the package hierarchy. Relative imports are specified using dots (.) to indicate the level of relative positioning.

In order to demonstrate relative imports, let’s create a subpackage in the calculator package, call the subpackage multiply, then move the multiply.py module into that subpackage, so that we’ll have an updated package structure like this:

```
calculator/
├── __init__.py
├── add.py
├── subtract.py
└── multiply/
    ├── __init__.py
    └── multiply.py
```

to import the multiply module, we could use the code below:

```
from .multiply import multiply

result = multiply(5, 9)
print(result)
```

The __all__ attribute

There are times when we may use all modules from a package or subpackages, or all functions and variables from a module, so typing out all names becomes quite cumbersome. So we want a way to specify that we’re importing functions and variables that a module has to offer or all modules that package offers.

To set up what can be imported when a user wants to import all offerings from a module or a package, Python has the __all__ attribute, which is a special attribute that’s used in modules or packages to control what gets imported when a user uses the from module import * statement. This attribute allows us to specify a list of names that will be considered “public” and will be imported when the wildcard (*) import is used.

Using the __all__ attribute in modules

```
# mymodule.py
__all__ = ['public_function', 'PublicClass']

def public_function():
    print("This is a public function.")

def _private_function():
    print("This is a private function.")

class PublicClass:
    def __init__(self):
        print("This is a public class.")

class _PrivateClass:
    def __init__(self):
        print("This is a private class.")
```

__all__ = ['public_function', 'PublicClass'] specifies that only public_function and PublicClass should be accessible when using from mymodule import *.

Using the Module:

```
from mymodule import *

public_function()  # This will work

PublicClass()  # This will work

_private_function()  # This will raise NameError: name '_private_function' is not defined

_PrivateClass()  # This will raise NameError: name '_PrivateClass' is not defined
```

Key Points:

- **Controlling Namespace:** The __all__ attribute helps you control the namespace of your module by specifying which names should be exported.
- **Private Members:** Names not included in __all__ are considered private and are not imported when using from module import *.
- **Convention:** By convention, names starting with an underscore (_) are considered private, but this is not enforced by the language. Using __all__ helps enforce this convention.

When __all__ is Not Defined:

```
# mymodule.py without __all__

def public_function():
    print("This is a public function.")

def _private_function():
    print("This is a private function.")

class PublicClass:
    def __init__(self):
        print("This is a public class.")

class _PrivateClass:
    def __init__(self):
        print("This is a private class.")
```

```
from mymodule import *

public_function()  # This will work

PublicClass()  # This will work

_private_function()  # This will NOT raise an error because _private_function is also imported

_PrivateClass()  # This will NOT raise an error because _PrivateClass is also imported
```

Enumeration

Enumeration in Python is handled by the enum module, which provides a way to define a set of named values.

These values can be used to represent a finite set of possible options or states, making code more readable and maintainable.

Enumerations are created using the Enum class from the enum module.

Creating Enumerations

To create an enumeration, you define a class that inherits from enum.Enum.

```
from enum import Enum

class Color(Enum):
    RED = 1
    GREEN = 2
    BLUE = 3
```

Accessing Enumeration Members

```
# Access by name
print(Color.RED)  # Output: Color.RED

# Access by value
print(Color(1))  # Output: Color.RED

# Access the name and value
print(Color.RED.name)  # Output: RED
print(Color.RED.value)  # Output: 1
```

Iterating over Enumeration Members

You can iterate over the members of an enumeration:

```
for color in Color:
    print(color)
```

Extending Enums

```
from enum import Enum

class EmptyEnum(Enum):
    pass

class Color(EmptyEnum):
    RED = 1
    GREEN = 2
    BLUE = 3

class ExtendedColor(EmptyEnum):
    YELLOW = 4
    ORANGE = 5

print(list(Color))
print(list(ExtendedColor))
```

Output

```
[<Color.RED: 1>, <Color.GREEN: 2>, <Color.BLUE: 3>]

[<ExtendedColor.YELLOW: 4>, <ExtendedColor.ORANGE: 5>]
```

Remember :

- Enum can’t be instiantiated even though using class keyword
- Are iteratable and return enum member
- Provide human readable string representation for their members
- Provide hashable members that can be used as dictionary keys
- can't be subclassed unless the base enum has no members

```
from enum import Enum


class Color(Enum):
    RED = 1
    GREEN = 2
    BLUE = 3


# Attempt to subclass the Color enum
class ExtendedColor(Color):
    YELLOW = 4
#This will raise a TypeError.
```