OOPS:-

Got it! Here's a **detailed, in-depth explanation of all OOP concepts and topics in Python**, with code examples and explanations for each major point. This is like a complete guide to OOP in Python!

**Object-Oriented Programming (OOP) in Python — Detailed Guide**

**1. Introduction to OOP**

OOP is a programming paradigm that uses **objects** and **classes** to organize code. Objects bundle **data** (attributes) and **behavior** (methods) into reusable entities, making programs easier to design, maintain, and extend.

**2. Classes and Objects**

**2.1 Class**

A **class** is a blueprint or template for creating objects. It defines **attributes** (data) and **methods** (functions).

class Car:

def \_\_init\_\_(self, make, model):

self.make = make # instance attribute

self.model = model

def display\_info(self):

print(f"Car: {self.make} {self.model}")

**2.2 Object (Instance)**

An **object** is an instance of a class, created using the class constructor.

my\_car = Car("Toyota", "Corolla")

my\_car.display\_info() # Output: Car: Toyota Corolla

**3. The \_\_init\_\_ Method (Constructor)**

* Special method called automatically when a new object is created.
* Used to initialize attributes.

class Person:

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

self.name = name

self.age = age

When you create an object, Python calls \_\_init\_\_:

p = Person("Alice", 30)

print(p.name) # Alice

**4. Attributes and Methods**

**4.1 Instance Attributes**

* Unique to each object.

p1 = Person("Alice", 30)

p2 = Person("Bob", 25)

print(p1.name) # Alice

print(p2.name) # Bob

**4.2 Class Attributes**

* Shared among all instances of the class.

class Dog:

species = "Canis familiaris" # class attribute

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

self.name = name # instance attribute

d1 = Dog("Buddy")

d2 = Dog("Max")

print(d1.species) # Canis familiaris

print(d2.species) # Canis familiaris

**4.3 Methods**

* Functions defined inside a class.
* Can access attributes and other methods.

class Calculator:

def add(self, a, b):

return a + b

calc = Calculator()

print(calc.add(5, 3)) # 8

**5. Encapsulation**

Encapsulation bundles data (attributes) and methods, and restricts direct access to some components.

**5.1 Access Modifiers (by convention)**

* **Public:** accessible everywhere (normal attributes/methods).
* **Protected:** indicated by a single underscore \_var, should not be accessed outside class or subclass.
* **Private:** indicated by double underscore \_\_var, name-mangled to discourage outside access.

class Employee:

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

self.name = name # public

self.\_salary = salary # protected

self.\_\_bonus = 1000 # private

emp = Employee("John", 5000)

print(emp.name) # John

print(emp.\_salary) # 5000 (but avoid accessing directly)

#print(emp.\_\_bonus) # AttributeError

# Access private via name mangling:

print(emp.\_Employee\_\_bonus) # 1000

**5.2 Getters and Setters (Property Decorators)**

Control access to attributes.

class Person:

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

self.\_age = age

@property

def age(self): # getter

return self.\_age

@age.setter

def age(self, value): # setter

if value < 0:

raise ValueError("Age cannot be negative")

self.\_age = value

p = Person(25)

print(p.age) # 25

p.age = 30

#print(p.age) # 30

#p.age = -5 # ValueError

**6. Inheritance**

Inheritance lets a class (child) acquire properties and behaviors from another class (parent).

**6.1 Single Inheritance**

class Animal:

def speak(self):

print("Animal sound")

class Dog(Animal):

def speak(self):

print("Woof!")

dog = Dog()

dog.speak() # Woof!

**6.2 Multiple Inheritance**

Class inherits from multiple parents.

class Flyer:

def fly(self):

print("Flying")

class Swimmer:

def swim(self):

print("Swimming")

class Duck(Flyer, Swimmer):

pass

d = Duck()

d.fly() # Flying

d.swim() # Swimming

**6.3 Multilevel Inheritance**

class Vehicle:

def start(self):

print("Vehicle started")

class Car(Vehicle):

def drive(self):

print("Driving")

class SportsCar(Car):

def race(self):

print("Racing")

sc = SportsCar()

sc.start() # Vehicle started

sc.drive() # Driving

sc.race() # Racing

**6.4 Method Overriding**

Child class provides specific implementation.

**7. Polymorphism**

Ability to use a unified interface with different underlying forms.

**7.1 Method Overriding (Runtime polymorphism)**

Same method name, different behavior.

class Bird:

def fly(self):

print("Bird flying")

class Ostrich(Bird):

def fly(self):

print("Ostriches can't fly")

b = Bird()

o = Ostrich()

b.fly() # Bird flying

o.fly() # Ostriches can't fly

**7.2 Duck Typing (Pythonic Polymorphism)**

Any object with the right methods/attributes can be used.

class Dog:

def speak(self):

print("Woof!")

class Cat:

def speak(self):

print("Meow!")

def make\_animal\_speak(animal):

animal.speak()

make\_animal\_speak(Dog()) # Woof!

make\_animal\_speak(Cat()) # Meow!

**8. Abstraction**

Hiding complex details and exposing only essential features.

**8.1 Abstract Base Classes (ABC)**

Use abc module.

from abc import ABC, abstractmethod

class Shape(ABC):

@abstractmethod

def area(self):

pass

class Rectangle(Shape):

def \_\_init\_\_(self, w, h):

self.w = w

self.h = h

def area(self):

return self.w \* self.h

r = Rectangle(10, 5)

print(r.area()) # 50

# Trying to instantiate Shape() raises error:

# s = Shape() # TypeError: Can't instantiate abstract class

**9. Special (Magic/Dunder) Methods**

Python classes can implement special methods to support built-in operations.

| **Method** | **Purpose** |
| --- | --- |
| \_\_init\_\_ | Constructor |
| \_\_str\_\_ | Informal string representation |
| \_\_repr\_\_ | Official string representation |
| \_\_add\_\_ | Overload + operator |
| \_\_len\_\_ | Support len() |
| \_\_getitem\_\_ | Indexing support |
| \_\_setitem\_\_ | Assign to indexed element |

Example:

class Vector:

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

self.x = x

self.y = y

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

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

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

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

v1 = Vector(1, 2)

v2 = Vector(3, 4)

v3 = v1 + v2

print(v3) # Vector(4, 6)

**10. Class Methods and Static Methods**

**10.1 Instance Methods**

* Default kind, receive self.

**10.2 Class Methods**

* Use @classmethod, receive cls parameter.
* Can modify class state.

class Person:

population = 0

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

self.name = name

Person.population += 1

@classmethod

def get\_population(cls):

return cls.population

p1 = Person("Alice")

p2 = Person("Bob")

print(Person.get\_population()) # 2

**10.3 Static Methods**

* Use @staticmethod, no implicit parameter.
* Utility functions inside class namespace.

class Math:

@staticmethod

def add(x, y):

return x + y

print(Math.add(5, 3)) # 8

**11. Composition vs Inheritance**

* **Inheritance:** "is-a" relationship (e.g., Dog is-an Animal)
* **Composition:** "has-a" relationship (object contains another)

class Engine:

def start(self):

print("Engine started")

class Car:

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

self.engine = Engine()

def start(self):

self.engine.start()

print("Car started")

c = Car()

c.start()

**12. Method Resolution Order (MRO)**

Defines order Python follows to resolve methods in multiple inheritance.

class A:

pass

class B(A):

pass

class C(A):

pass

class D(B, C):

pass

print(D.\_\_mro\_\_)

**13. Data Classes (Python 3.7+)**

Simplify classes that are mainly used to store data.

from dataclasses import dataclass

@dataclass

class Point:

x: int

y: int

p = Point(10, 20)

print(p) # Point(x=10, y=20)

**14. Slots (\_\_slots\_\_)**

* Restrict dynamic creation of attributes.
* Saves memory.

class Person:

\_\_slots\_\_ = ['name', 'age']

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

self.name = name

self.age = age

**15. Exception Handling in OOP**

Use try-except inside methods to handle errors.

class Calculator:

def divide(self, a, b):

try:

return a / b

except ZeroDivisionError:

print("Error: Division by zero")

calc = Calculator()

print(calc.divide(10, 2)) # 5.0

print(calc.divide(10, 0)) # Error message

**Summary Table of OOP Concepts**

| **Concept** | **Description** |
| --- | --- |
| Class | Blueprint for objects |
| Object | Instance of a class |
| Attributes | Data stored in objects |
| Methods | Functions inside classes |
| Encapsulation | Data hiding using access control |
| Inheritance | Child inherits from parent |
| Polymorphism | Same interface, different behavior |
| Abstraction | Hide complexity via abstract classes |
| Magic Methods | Special methods like \_\_init\_\_, \_\_str\_\_, etc. |
| Class & Static Methods | Methods bound to class or no binding |
| Composition | Objects composed of other objects |
| Data Classes | Boilerplate reduction for data storing classes |
| Slots | Memory optimization by attribute restriction |
| Exception Handling | Handling errors gracefully inside classes |

If you want, I can create code samples or exercises for any specific topic. Just say which ones!

Absolutely! Here’s a **detailed explanation of the different types of inheritance in Python**, with examples for each.

**Types of Inheritance in Python**

Inheritance allows a class (child/subclass) to inherit attributes and methods from another class (parent/superclass). Python supports several inheritance types:

**1. Single Inheritance**

* One child class inherits from one parent class.
* Simple and straightforward.

**Example:**

class Animal:

def speak(self):

print("Animal speaks")

class Dog(Animal):

def speak(self):

print("Dog barks")

d = Dog()

d.speak() # Dog barks

**2. Multiple Inheritance**

* One child class inherits from **multiple** parent classes.
* Child inherits attributes and methods from all parents.

**Example:**

class Flyer:

def fly(self):

print("Flying")

class Swimmer:

def swim(self):

print("Swimming")

class Duck(Flyer, Swimmer):

pass

d = Duck()

d.fly() # Flying

d.swim() # Swimming

**3. Multilevel Inheritance**

* A class inherits from a class which itself is inherited from another class.
* Forms a chain of inheritance.

**Example:**

class Vehicle:

def start(self):

print("Vehicle started")

class Car(Vehicle):

def drive(self):

print("Car driving")

class SportsCar(Car):

def race(self):

print("Sports car racing")

sc = SportsCar()

sc.start() # Vehicle started

sc.drive() # Car driving

sc.race() # Sports car racing

**4. Hierarchical Inheritance**

* Multiple child classes inherit from a single parent class.
* Sibling classes share common features from the parent.

**Example:**

class Animal:

def speak(self):

print("Animal speaks")

class Dog(Animal):

def speak(self):

print("Dog barks")

class Cat(Animal):

def speak(self):

print("Cat meows")

d = Dog()

c = Cat()

d.speak() # Dog barks

c.speak() # Cat meows

**5. Hybrid Inheritance**

* Combination of two or more types of inheritance.
* Complex inheritance trees.

**Example:**

class Animal:

def speak(self):

print("Animal speaks")

class Flyer(Animal):

def fly(self):

print("Flying")

class Swimmer(Animal):

def swim(self):

print("Swimming")

class Duck(Flyer, Swimmer):

def speak(self):

print("Duck quacks")

d = Duck()

d.speak() # Duck quacks

d.fly() # Flying

d.swim() # Swimming

**Summary Table**

| **Type** | **Description** | **Example Use Case** |
| --- | --- | --- |
| Single | One child inherits from one parent | Dog inherits Animal |
| Multiple | One child inherits from multiple parents | Duck inherits Flyer and Swimmer |
| Multilevel | Chain of inheritance | SportsCar inherits Car, which inherits Vehicle |
| Hierarchical | Multiple children inherit from one parent | Dog and Cat inherit Animal |
| Hybrid | Combination of above types | Complex class structures |

If you want, I can explain **Method Resolution Order (MRO)** too, which is important for multiple and hybrid inheritance!

Great question! When we talk about **abstraction in programming**, especially in Python, it generally refers to **hiding the complex implementation details** and exposing only the essential features to the user.

In Python and OOP, **abstraction** can be realized in different ways or "types" depending on how much and what you hide and how you expose interfaces.

**Types of Abstraction**

**1. Data Abstraction**

* Hides the internal data of an object.
* Users interact with the data only through methods (getters/setters or properties).
* Prevents direct access to the internal state.
* Achieved through **encapsulation** (using private/protected variables).

**Example:**

class Account:

def \_\_init\_\_(self, owner, balance):

self.owner = owner

self.\_\_balance = balance # private attribute

def deposit(self, amount):

if amount > 0:

self.\_\_balance += amount

def withdraw(self, amount):

if 0 < amount <= self.\_\_balance:

self.\_\_balance -= amount

else:

print("Insufficient funds")

def get\_balance(self):

return self.\_\_balance

acc = Account("John", 1000)

print(acc.get\_balance()) # 1000

acc.withdraw(200)

print(acc.get\_balance()) # 800

# Direct access is blocked:

# print(acc.\_\_balance) # AttributeError

**2. Procedural Abstraction**

* Focuses on hiding the complex logic or steps inside methods/functions.
* The user knows *what* the method does but not *how* it does it.
* The method provides a clear interface.

**Example:**

class Calculator:

def add(self, a, b):

return a + b # user doesn’t need to know how addition is done internally

calc = Calculator()

print(calc.add(5, 3)) # 8

**3. Abstract Classes and Methods (Interface Abstraction)**

* Uses abstract base classes (ABCs) to **define interfaces without implementation**.
* Forces subclasses to implement abstract methods.
* Hides implementation details by not providing any in the base class.

**Example:**

from abc import ABC, abstractmethod

class Vehicle(ABC):

@abstractmethod

def start\_engine(self):

pass

class Car(Vehicle):

def start\_engine(self):

print("Car engine started")

class Bike(Vehicle):

def start\_engine(self):

print("Bike engine started")

c = Car()

c.start\_engine() # Car engine started

b = Bike()

b.start\_engine() # Bike engine started

Here, the **details of start\_engine() are abstracted**; the interface demands it but doesn’t implement it.

**4. Interface Abstraction (Informal in Python)**

* Python does not have formal interfaces like Java, but ABCs simulate interfaces.
* Classes can define methods that act as a contract for subclasses.
* Ensures subclasses provide required methods.

**Summary**

| **Type** | **What is Abstracted?** | **How?** | **Example** |
| --- | --- | --- | --- |
| Data Abstraction | Internal data/state | Encapsulation (\_\_private, properties) | Account balance hiding |
| Procedural Abstraction | Implementation details of operations | Methods/functions | Calculator add() hides addition steps |
| Abstract Classes/Methods | Interface or API without implementation | abc module, @abstractmethod | Vehicle class with abstract start\_engine() |
| Interface Abstraction | Contract to implement certain methods | ABCs or method signatures | Similar to abstract classes |

If you'd like, I can also explain **how abstraction relates to encapsulation** or provide examples of abstraction in bigger real-world systems!

Sure! When we talk about **encapsulation in Python**, it basically means **restricting access to certain parts of an object to protect the internal state and only expose what’s necessary**.

Encapsulation is usually classified by **how strictly the access to attributes is controlled**, often using naming conventions.

**Types of Encapsulation in Python**

Python doesn't enforce strict access control like some other languages (e.g., private/public keywords in Java or C++), but it follows naming conventions to indicate the level of access.

**1. Public Encapsulation (Public Members)**

* Attributes and methods are **fully accessible** from outside the class.
* No underscores before the name.
* This is the default in Python.

class Employee:

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

self.name = name # public attribute

self.salary = salary # public attribute

emp = Employee("Alice", 5000)

print(emp.name) # Accessible

print(emp.salary) # Accessible

**2. Protected Encapsulation (Protected Members)**

* Indicated by a **single underscore prefix** \_variable.
* By convention, it means **“use with caution”** or **intended for internal use**.
* Still accessible from outside but should be treated as non-public.
* Helps subclasses access while discouraging outside access.

class Employee:

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

self.\_name = name # protected attribute

self.\_salary = salary # protected attribute

emp = Employee("Bob", 6000)

print(emp.\_name) # Accessible but discouraged

print(emp.\_salary) # Accessible but discouraged

**3. Private Encapsulation (Private Members)**

* Indicated by **double underscore prefix** \_\_variable.
* Triggers **name mangling**, making it harder to access from outside.
* Enforced by Python to avoid accidental access or modification.

class Employee:

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

self.\_\_name = name # private attribute

self.\_\_salary = salary # private attribute

emp = Employee("Charlie", 7000)

#print(emp.\_\_name) # AttributeError

# Access via name mangling:

print(emp.\_Employee\_\_name) # Charlie

print(emp.\_Employee\_\_salary) # 7000

**4. Very Private (Name Mangling) — Internally enforced privacy**

* Same as private encapsulation above.
* It’s Python’s way to avoid naming conflicts especially in inheritance.

**Summary Table**

| **Encapsulation Type** | **Syntax** | **Access Level** | **Usage** |
| --- | --- | --- | --- |
| Public | variable | Accessible everywhere | Default, for general use |
| Protected | \_variable | Accessible in class & subclasses; discouraged externally | For internal use, inheritance |
| Private | \_\_variable | Name mangled; not accessible externally | For strict internal use |

**Note on Encapsulation in Python:**

* Python trusts the programmer (consenting adults principle).
* Encapsulation is more about **convention** than enforcement.
* You can always access “private” members if you really want to, but it’s discouraged.

If you want, I can also explain **how encapsulation relates to properties (getters/setters)** or provide **examples of enforcing encapsulation with property decorators**!

Sure! Here’s a detailed explanation of **Polymorphism** in Python, covering the concept, types, and examples.

**Polymorphism in Python**

**Polymorphism** means **“many forms”** — the ability of different objects to be accessed through the same interface, highlighting their own specific behaviors.

In Python, polymorphism allows you to write code that works on objects of different classes as long as they implement the required methods.

**Why Polymorphism?**

* Simplifies code.
* Allows **flexible and extensible** programs.
* Supports **code reusability**.
* Enables writing **generic functions** or methods.

**Types of Polymorphism in Python**

**1. Compile-time Polymorphism (Method Overloading)**

* Python does **not** support traditional method overloading like Java or C++.
* But default arguments and variable-length arguments allow similar behavior.

class Math:

def add(self, a, b=0, c=0):

return a + b + c

m = Math()

print(m.add(5)) # 5

print(m.add(5, 10)) # 15

print(m.add(5, 10, 15))# 30

**2. Run-time Polymorphism (Method Overriding)**

* A child class **overrides** a method of the parent class.
* The method that gets called depends on the **object’s runtime type**.

class Animal:

def speak(self):

print("Animal speaks")

class Dog(Animal):

def speak(self):

print("Dog barks")

class Cat(Animal):

def speak(self):

print("Cat meows")

def make\_animal\_speak(animal):

animal.speak()

a = Animal()

d = Dog()

c = Cat()

make\_animal\_speak(a) # Animal speaks

make\_animal\_speak(d) # Dog barks

make\_animal\_speak(c) # Cat meows

**3. Duck Typing (Pythonic Polymorphism)**

* Python uses duck typing: **"If it walks like a duck and quacks like a duck, it’s a duck."**
* You don’t care about the type of the object, just if it has the right methods.

class Bird:

def fly(self):

print("Bird is flying")

class Airplane:

def fly(self):

print("Airplane is flying")

def lets\_fly(flier):

flier.fly()

b = Bird()

a = Airplane()

lets\_fly(b) # Bird is flying

lets\_fly(a) # Airplane is flying

**Summary Table**

| **Type** | **Description** | **Python Support** | **Example** |
| --- | --- | --- | --- |
| Method Overloading (Compile-time) | Same method name, different parameters | Limited (via default args) | add(self, a, b=0, c=0) |
| Method Overriding (Run-time) | Child class redefines parent method | Yes | Dog.speak() overrides Animal.speak() |
| Duck Typing | Interface-based polymorphism | Yes | Any object with .fly() method |

If you want, I can also explain **operator overloading** as another form of polymorphism in Python! Would you like that?

Absolutely! Here’s a **detailed explanation of all major OOP topics in Python** with examples, so you get a solid understanding from basics to advanced concepts.

**1. Classes and Objects**

* **Class**: Blueprint/template for creating objects (instances).
* **Object**: Instance of a class.
* **Attributes**: Data variables associated with a class or object.
* **Methods**: Functions defined inside a class that operate on objects.

**Example:**

class Dog:

# Class attribute

species = "Canis familiaris"

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

# Instance attributes

self.name = name

self.age = age

def speak(self):

return f"{self.name} says Woof!"

# Creating an object

dog1 = Dog("Buddy", 3)

print(dog1.name) # Buddy

print(dog1.speak()) # Buddy says Woof!

**2. Encapsulation**

* Restrict access to object data (attributes).
* Use naming conventions:
  + Public: accessible anywhere.
  + Protected: \_var — by convention, for internal use.
  + Private: \_\_var — name mangled, harder to access.

**Getters and Setters:**

class Person:

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

self.\_\_name = name # private

def get\_name(self):

return self.\_\_name

def set\_name(self, name):

self.\_\_name = name

p = Person("Alice")

print(p.get\_name()) # Alice

p.set\_name("Bob")

print(p.get\_name()) # Bob

**Using property decorators (Pythonic way):**

class Person:

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

self.\_\_name = name

@property

def name(self):

return self.\_\_name

@name.setter

def name(self, name):

self.\_\_name = name

p = Person("Alice")

print(p.name) # Alice

p.name = "Bob"

print(p.name) # Bob

**3. Inheritance**

* Mechanism to create a new class from an existing class.
* Child class inherits methods and attributes of the parent.
* Supports code reuse and polymorphism.

**Example:**

class Animal:

def speak(self):

print("Animal speaks")

class Dog(Animal):

def speak(self):

print("Dog barks")

d = Dog()

d.speak() # Dog barks

**Types of Inheritance:**

* Single: One child, one parent.
* Multiple: One child, multiple parents.
* Multilevel: Chain inheritance.
* Hierarchical: Multiple children from one parent.
* Hybrid: Combination of above.

**4. Polymorphism**

* Same method name behaves differently depending on object.
* Achieved by method overriding and duck typing.

**Method Overriding:**

class Bird:

def fly(self):

print("Flying")

class Penguin(Bird):

def fly(self):

print("Penguins cannot fly")

b = Bird()

p = Penguin()

b.fly() # Flying

p.fly() # Penguins cannot fly

**Duck Typing Example:**

class Cat:

def sound(self):

print("Meow")

class Dog:

def sound(self):

print("Woof")

def make\_sound(animal):

animal.sound()

make\_sound(Cat()) # Meow

make\_sound(Dog()) # Woof

**5. Abstraction**

* Hiding internal details and showing only essential features.
* Use **abstract base classes (ABC)** and **abstract methods**.

from abc import ABC, abstractmethod

class Shape(ABC):

@abstractmethod

def area(self):

pass

class Rectangle(Shape):

def \_\_init\_\_(self, w, h):

self.w = w

self.h = h

def area(self):

return self.w \* self.h

r = Rectangle(5, 3)

print(r.area()) # 15

**6. Constructors and Destructors**

* **Constructor** (\_\_init\_\_): Initializes new objects.
* **Destructor** (\_\_del\_\_): Called when object is garbage collected.

class Car:

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

self.model = model

print(f"{self.model} created")

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

print(f"{self.model} destroyed")

c = Car("Tesla")

del c # Explicitly deletes object

**7. Class and Instance Variables**

* **Instance variables**: unique to each object.
* **Class variables**: shared across all instances.

class Employee:

company = "ABC Corp" # class variable

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

self.name = name # instance variable

e1 = Employee("John")

e2 = Employee("Jane")

print(e1.company) # ABC Corp

print(e2.company) # ABC Corp

Employee.company = "XYZ Ltd"

print(e1.company) # XYZ Ltd

**8. Static Methods and Class Methods**

* **Static methods** don’t access instance or class data.
* **Class methods** receive the class as the first argument.

class MyClass:

@staticmethod

def static\_method():

print("Static method called")

@classmethod

def class\_method(cls):

print(f"Class method called from {cls}")

MyClass.static\_method()

MyClass.class\_method()

**9. Method Overloading and Overriding**

* Python doesn’t support method overloading by default.
* Can use default arguments or variable arguments.

class Math:

def add(self, a, b=0, c=0):

return a + b + c

m = Math()

print(m.add(5)) # 5

print(m.add(5, 10)) # 15

print(m.add(5, 10, 15))# 30

* Method overriding is done by redefining the method in child class.

**10. Operator Overloading**

* Define how operators behave for your objects by implementing **magic methods** like \_\_add\_\_, \_\_sub\_\_, etc.

class Point:

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

self.x = x

self.y = y

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

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

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

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

p1 = Point(1, 2)

p2 = Point(3, 4)

print(p1 + p2) # (4, 6)

**11. Exception Handling in OOP**

* Use try-except blocks inside class methods.
* Create custom exceptions.

class NegativeValueError(Exception):

pass

class Calculator:

def divide(self, a, b):

if b == 0:

raise ZeroDivisionError("Cannot divide by zero")

if a < 0 or b < 0:

raise NegativeValueError("Negative values not allowed")

return a / b

calc = Calculator()

try:

print(calc.divide(10, 0))

except ZeroDivisionError as e:

print(e)

**12. Magic Methods / Dunder Methods**

* Special methods with double underscores that let you customize object behavior.

| **Method** | **Purpose** |
| --- | --- |
| \_\_init\_\_ | Constructor |
| \_\_str\_\_ | String representation |
| \_\_repr\_\_ | Official string representation |
| \_\_eq\_\_ | Equality check (==) |
| \_\_len\_\_ | Length with len() |
| \_\_call\_\_ | Make object callable |

Example:

class Person:

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

self.name = name

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

return f"Person named {self.name}"

p = Person("Alice")

print(p) # Person named Alice

**13. Composition vs Inheritance**

* **Inheritance**: “Is-a” relationship (Dog is an Animal).
* **Composition**: “Has-a” relationship (Car has an Engine).

class Engine:

def start(self):

print("Engine started")

class Car:

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

self.engine = Engine() # Car has an Engine

def start(self):

self.engine.start()

print("Car started")

c = Car()

c.start()

**14. Multiple Inheritance and Method Resolution Order (MRO)**

* Python supports multiple inheritance.
* MRO determines the order in which base classes are searched.

class A:

def greet(self):

print("Hello from A")

class B(A):

def greet(self):

print("Hello from B")

class C(A):

def greet(self):

print("Hello from C")

class D(B, C):

pass

d = D()

d.greet() # Hello from B

print(D.mro()) # Shows the method resolution order

**15. Inner Classes**

* Classes defined inside other classes.

class Outer:

class Inner:

def greet(self):

print("Hello from Inner")

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

self.inner = Outer.Inner()

o = Outer()

o.inner.greet() # Hello from Inner

**16. Design Principles and Patterns (Advanced)**

* SOLID principles guide writing maintainable code.
* Design patterns like Singleton, Factory, Observer.
* Example: Singleton ensures only one instance of a class.