

Chapter 11

✓ Chapter 11 – More on OOP & Inheritance (Advanced OOP Concepts)

📖 1. Inheritance – Reusability & Extension

+ What it is:

Inheritance allows a class (**child/derived**) to use properties and methods of another class (**parent/base**), avoiding code duplication.

```
class Employee:
    def info(self, name, salary):
        self.name = name
        self.salary = salary
        print(f"The name is {self.name} and salary is {self.salary}")

class Programmer(Employee): # Inherits everything from Employee
    def status(self):
        print(f"{self.name} is currently inactive")
```

💡 Key Points:

- Single inheritance in `Programmer(Employee)`
- Inherited attributes/methods can be accessed directly

- Parent class methods can be overridden if needed



2. Multiple Inheritance

```
class Employee:
    def info(self, name, salary): ...

class PersonallInfo:
    def age(self, age): ...

class Programmer(Employee, PersonallInfo): ...
```



Key Points:

- Class `Programmer` inherits from both `Employee` and `PersonallInfo`
- This is **multiple inheritance** (more than one base class)
- Python uses **MRO (Method Resolution Order)** to decide which method runs if names clash



3. Multilevel Inheritance

```
class Employee:
    a = 23

class Programmer(Employee):
    b = 43

class Coder(Programmer):
    c = 34
```



Key Points:

- Class `Coder` inherits from `Programmer`, which in turn inherits `Employee`
- `Coder` gets access to all attributes: `a`, `b`, and `c`
- This is a **chain of inheritance**



4. `super()` – Calling Parent Constructor

```
class Employee:
    def __init__(self):
```

```
print("I am Employee")
```

```
class Programmer(Employee):  
    def __init__(self):  
        print("I am Programmer")
```

```
class Coder(Programmer):  
    def __init__(self):  
        super().__init__()  
        print("I am Coder")
```

💡 Key Points:

- `super()` calls parent's constructor or methods
- If `super()` is used, we can control initialization flow
- Helps in diamond problem and multi-level inheritance

🧠 5. `@classmethod` – Working on Class (Not Instance)

```
class Employee:  
    a = 1  
  
    @classmethod  
    def show(cls):  
        print(f"Class attribute a = {cls.a}")
```

💡 Key Points:

- Works with class itself, not object
- Uses `cls` instead of `self`
- Can **access/modify class variables**, not instance ones

🔒 6. Encapsulation with `@property` & `@setter`

```
class Employee:  
    @property  
    def name(self):  
        return f"{self.fname}, {self.lname}"  
  
    @name.setter
```

```
def name(self, value):
    self.fname = value.split()[0]
    self.lname = value.split()[1]
```

💡 Key Points:

- Hides complexity: user assigns `e.name = "John Doe"` and behind the scenes it's split
- `@property` makes a method look like an attribute
- `@setter` allows value assignment while controlling internal behavior

+ 7. Operator Overloading (`__add__` , `__str__`)

```
class Number:
    def __init__(self, n): self.n = n

    def __add__(self, num): return Number(self.n + num.n)

n1 = Number(5)
n2 = Number(3)
print((n1 + n2).n) # 8
```

💡 Key Points:

- `__add__` lets you use `+` between objects
- Dunder methods (`__add__` , `__len__` , etc.) make classes feel like built-ins
- You can customize how operators behave on objects

🧠 Chapter 11 – Advanced Deep Dive: OOPs, Inheritance & Python Magic

🔄 Inheritance – The Real Superpower of OOP

| DRY – Don't Repeat Yourself. Inheritance embodies this philosophy.

🎯 What actually happens?

- Inheritance is not just "reusing" code — it **extends behaviors**.
- The **child class gets a blueprint** (all methods, variables) from parent.

- Python implements this using **Method Resolution Order (MRO)** – it goes from child → leftmost parent → next parent → object.

Types of Inheritance:

Type	Structure	Python Example
Single	$A \rightarrow B$	<code>class B(A)</code>
Multiple	$A + B \rightarrow C$	<code>class C(A, B)</code>
Multilevel	$A \rightarrow B \rightarrow C$	<code>class C(B)</code> where <code>class B(A)</code>
Hierarchical	$A \rightarrow B, A \rightarrow C$	Many subclasses from one superclass
Hybrid	Mix of the above	Real Python code often has hybrid

`super()` – Not Just a Shortcut

Think of `super()` as a controlled way to hand over power to your parent class.

✓ When to use:

- In constructors where multiple parents have `__init__`
- When overriding methods but still want the parent behavior

Python's MRO (important!):

```
print(ClassName.__mro__)
```

This shows the actual order Python uses to resolve method calls across inheritance.

```
class A:
    def show(self): print("A")

class B(A):
    def show(self):
        super().show()
        print("B")
```

Operator Overloading (Dunder/Magic Methods)

Dunder = "Double Underscore" methods like `__add__`, `__str__`, `__len__`...

🎯 Why use it?

- Makes your custom classes behave like built-in types.
- You can define how operators (`+` , `,` , `==`) work with your objects.

✅ Commonly overloaded methods:

Operator	Method	Example
<code>+</code>	<code>__add__</code>	<code>obj1 + obj2</code>
<code>-</code>	<code>__sub__</code>	<code>obj1 - obj2</code>
<code>*</code>	<code>__mul__</code>	<code>obj1 * obj2</code>
<code>==</code>	<code>__eq__</code>	<code>obj1 == obj2</code>
<code>str()</code>	<code>__str__</code>	Used when printing object

🛡️ Encapsulation – `@property` and `@setter`

| Encapsulation ≠ hiding data, it means controlling access.

⚙️ Why use it?

- Prevent direct modification of internal variables
- Add logic behind getting/setting data without changing API

✅ Behind the scenes:

```
class Person:
    @property
    def name(self): return self._name

    @name.setter
    def name(self, val): self._name = val.title()
```

Now setting `p.name = "john doe"` auto-capitalizes it.

📦 Classmethods vs Staticmethods

Type	Decorator	Uses <code>self</code> ?	Uses <code>cls</code> ?	Purpose
Instance Method	(none)	✅	❌	Normal methods using object data
Class Method	<code>@classmethod</code>	❌	✅	Work with class-level data
Static Method	<code>@staticmethod</code>	❌	❌	Utility functions bound to class (not obj)

```


class Pizza:
    base_price = 100

    @classmethod
    def margherita(cls):
        return cls(base_price + 50)

    @staticmethod
    def greet():
        print("Welcome to PizzaHub!")

```

Real World OOP Mapping

Concept	Real World Analogy
Class	Blueprint / Template
Object	Actual Car made from the blueprint
Inheritance	Tesla inherits features from Car
Encapsulation	Private parts of engine you can't touch
Abstraction	Using steering, not worrying about gears
Polymorphism	 works for numbers, strings, lists

Advanced Topics to Explore (Bonus for Deep Learning)

Abstract Base Classes (from `abc` module)

- For defining abstract methods that must be overridden

```

from abc import ABC, abstractmethod
class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

```

Polymorphism (overriding behavior)

```

class Dog:
    def speak(self): return "Bark"

class Cat:
    def speak(self): return "Meow"

```

```
for animal in [Dog(), Cat()]:
    print(animal.speak()) # Same method name, different behavior
```

✓ Duck Typing

| “If it quacks like a duck and walks like a duck, it’s a duck.”

Python doesn’t care about type, it cares about **behavior**.

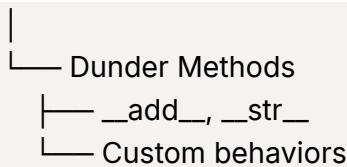
Interview Cracker Questions

Question	Answer Type
What's the difference between classmethod and staticmethod?	Theory + Example
How does Python resolve method calls in multiple inheritance?	MRO explanation
What is the purpose of <code>super()</code> in Python OOP?	Practical reasoning
How is operator overloading implemented in Python?	Code with <code>__add__</code>
What is the role of <code>@property</code> and why is it used?	Use case driven
Can Python classes support multiple inheritance?	Yes + MRO demo

Recap Mind Map (for Visualizing Later in Notion)

```

graph TD
    OOP[OOP Core]
    OOP --- Inheritance
    Inheritance --- Single[Single, Multi-level, Multiple]
    Inheritance --- MRO[MRO, super()]
    OOP --- Encapsulation
    Encapsulation --- Property["@property, @setter"]
    Encapsulation --- Access[Controlled access]
    OOP --- Polymorphism
    Polymorphism --- Method[Method Overriding]
    Polymorphism --- Operator[Operator Overloading]
    OOP --- Abstraction
    Abstraction --- ABC[ABC module, abstractmethod]
    OOP --- Class[Class vs Static vs Instance]
  
```

OOP Practice Problems — Deep Notes & Explanations

Problem 1: Inheriting a 2D Vector to make a 3D Vector

```
# Define a 2D vector class with x and y coordinates
class Vector2D:
    def __init__(self, x, y):
        self.x = x
        self.y = y

# Inherit Vector2D to create Vector3D by adding a z coordinate
class Vector3D(Vector2D):
    def set_properties(self, x, y, z):
        super().__init__(x, y) # Initialize base class members
        self.z = z
        print(f"3D Vector: x={self.x}, y={self.y}, z={self.z}")

# Create object and call method
a = Vector3D(23, 43)
a.set_properties(12, 12, 12) # Overwrites with new values

b = Vector2D(43, 93)
print(b.x, b.y)
```

Concepts Covered:

- Single Inheritance
- Use of `super()` to reuse parent constructor
- Overwriting attributes

Problem 2: Multilevel Inheritance with a Barking Dog

```

# Base class
class Animal:
    pass

# Intermediate class inheriting Animal
class Pets(Animal):
    pass

# Dog inherits from Pets → Animal (Multilevel Inheritance)
class Dog(Pets):
    @staticmethod
    def bark():
        print("Bark")

a = Dog()
Dog.bark()

```

✓ Concepts Covered:

- Multilevel Inheritance
- Use of `@staticmethod` (no self or cls required)
- Behavioral extension in subclasses

Problem 3: Using @property to calculate salary after increment

```

class Employee:
    salary = 78753
    increment = 20 # in percent

    @property
    def salary_after_increment(self):
        return self.salary * (1 + self.increment / 100)

    @salary_after_increment.setter
    def salary_after_increment(self, new_salary):
        # Reverse-calculate increment based on new salary
        self.increment = ((new_salary / self.salary) - 1) * 100

a = Employee()

```

```
a.salary_after_increment = 100000 # Updates increment based on target salary
print(f"New increment is: {a.increment:.2f}%")
```

✓ Concepts Covered:

- `@property` and `@setter` decorators
- Reverse computation via setter
- Encapsulation of business logic

Problem 4: Complex Number Addition using Operator Overloading

```
class Complex:
    def __init__(self, real, imag):
        self.real = real
        self.imag = imag

    def __add__(self, other):
        return Complex(self.real + other.real, self.imag + other.imag)

    def __str__(self):
        return f"{self.real} + {self.imag}i"

c1 = Complex(3, 5)
c2 = Complex(2, 4)
print(f"Sum = {c1 + c2}")
```

✓ Concepts Covered:

- Operator Overloading (`__add__`)
- String representation (`__str__`)
- Clean and intuitive syntax for custom types

Problem 5: Vector Addition and Dot Product with Operator Overloading

```
class Vector:
    def __init__(self, x, y, z):
        self.x = x
        self.y = y
```

```

        self.z = z

    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y, self.z + other.z)

    def __mul__(self, other):
        # Dot product returns a scalar
        return self.x * other.x + self.y * other.y + self.z * other.z

    def __str__(self):
        return f"{self.x}i + {self.y}j + {self.z}k"

# Input values from user
print("Enter components of Vector A:")
a = Vector(int(input("i: ")), int(input("j: ")), int(input("k: ")))

print("Enter components of Vector B:")
b = Vector(int(input("i: ")), int(input("j: ")), int(input("k: ")))

print(f"Vector A + B = {a + b}")
print(f"Dot Product A * B = {a * b}")

```

✓ Concepts Covered:

- Operator Overloading for `+` and `*`
- Dot product returns scalar (unlike vector in original)
- Dynamic vector math using class abstraction

Problem 6: Override `__len__()` to Return Vector Dimension

```

class Vector:
    def __init__(self, x, y, z):
        self.x = x
        self.y = y
        self.z = z

    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y, self.z + other.z)

    def __mul__(self, other):

```

```

    return self.x * other.x + self.y * other.y + self.z * other.z

def __len__(self):
    return 3 # Always 3D for this implementation

def __str__(self):
    return f"{self.x}i + {self.y}j + {self.z}k"

print("Enter components of Vector A:")
a = Vector(int(input("i: ")), int(input("j: ")), int(input("k: ")))

print("Enter components of Vector B:")
b = Vector(int(input("i: ")), int(input("j: ")), int(input("k: ")))

print(f"Dimension of vector is: {len(a)}")
print(f"Addition: {a + b}")
print(f"Dot Product: {a * b}")

```

✓ Concepts Covered:

- Custom behavior using `__len__()` dunder method
- Integration with built-in functions (`len()`)
- Strong understanding of Python's data model

✓ Combined Summary Table – Chapter 11: More on OOP and Inheritance

#	Concept / Topic	Description & Key Insights	Deep Dive / Real Logic	Code / Practice Problems
1	Inheritance	Mechanism to reuse code by creating new classes from existing ones	Parent class passes attributes & methods to child. Promotes DRY code.	<code>inheritance.py</code> (Employee → Programmer), <code>problem1.py</code> (2D → 3D Vector)
2	Single Inheritance	One child inherits from one parent class	Simple hierarchy. Child can override or extend parent methods.	<code>class</code> <code>Programmer(Employee)</code>
3	Multiple Inheritance	One class inherits from multiple parents	Combines features from multiple classes. Be	<code>multiple_inheritance.py</code> (Programmer inherits Employee + PersonallInfo)

			cautious of method resolution order (MRO).	
4	Multilevel Inheritance	Class inherits from a class that inherits from another class	Chain of inheritance: A → B → C	<code>multilevel_inheritance.py</code> , <code>problem2.py</code> (Animal → Pets → Dogs)
5	Constructor Chaining with <code>super()</code>	Use <code>super()</code> to call parent class's constructor/method	Maintains the proper flow in multiple inheritance	<code>super.py</code> , <code>problem1.py</code>
6	Dunder Methods (Magic Methods)	Special methods that start and end with <code>_</code>	<code>__init__</code> , <code>__add__</code> , <code>__str__</code> , <code>__len__</code> , etc. Customize object behavior	<code>problem4.py</code> , <code>problem5.py</code> , <code>problem6.py</code>
7	Operator Overloading	Using dunder methods to give operators like <code>+</code> , <code>*</code> custom behavior for your objects	Makes custom classes work like built-in types. <code>__add__</code> for addition, <code>__mul__</code> for dot product, etc.	<code>operator_overload.py</code> , <code>problem4.py</code> (complex), <code>problem5.py</code> (vector)
8	Encapsulation	Hiding internal details using getters/setters	Use <code>@property</code> and <code>@setter</code> decorators. Protects class state, provides control.	<code>decorators.py</code> , <code>problem3.py</code> (salary/increment logic)
9	@property Decorator	Turns method into attribute-like access	Readable and clean syntax for computed properties	<code>decorators.py</code> , <code>problem3.py</code>
10	@setter Decorator	Allows assignment to property	Behind-the-scenes logic while updating a value	Used to reverse-calculate increment in <code>problem3.py</code>
11	@classmethod	Works with class (<code>cls</code>) not instance (<code>self</code>)	Can be used to manipulate class variables, create alternate constructors	<code>classmethods.py</code>
12	@staticmethod	Doesn't receive <code>self</code> or <code>cls</code>	Used for helper methods related to class	<code>problem2.py</code> (Dog.bark),

				<code>problem4.py</code> (<code>Calculator.greet</code>)
13	Overriding Class vs Instance Attributes	Instance attribute overrides class attribute	Class attribute is shared, but once instance has same name — it hides the class one	<code>problem3.py</code> , <code>classmethods.py</code>
14	Dimension Logic with <code>__len__()</code>	Customize object length with <code>__len__</code>	e.g., 3D vector should return 3 when passed to <code>len()</code>	<code>problem6.py</code>
15	Custom <code>__str__()</code>	Controls how object is printed	Improves debugging, logging, output clarity	Used in <code>problem4.py</code> , <code>problem5.py</code> , <code>problem6.py</code>
16	Composition	Using classes inside other classes	Alternative to inheritance. "Has-a" relationship instead of "Is-a".	Not directly used, but worth exploring
17	Polymorphism	Same method name behaving differently across classes	Through method overriding or duck typing	Seen in action via inheritance overriding (e.g., <code>status()</code> in <code>Programmer</code> , <code>bark()</code> in <code>Dog</code>)

Final Takeaways for Real-World Use

Concept	Real-World Application
Inheritance	Building reusable models: <code>User</code> → <code>Admin</code> → <code>Moderator</code>
Operator Overloading	Building a Matrix or Vector library
<code>@property</code> / <code>@setter</code>	Django ORM / Config management
<code>super()</code>	Frameworks like Flask/Django use it everywhere
<code>@classmethod</code> / <code>@staticmethod</code>	Alternate constructors, util factories
<code>__str__</code> , <code>__len__</code> , <code>__add__</code>	Making your objects act like native types (clean APIs)