🧬 Python OOP — Inheritance (Colorful, Canva-Style Notes)

🕐 What is Inheritance?

A class (**child / derived**) can reuse and extend code from another class (**parent / base**). This promotes **code reuse**, **hierarchies**, and **clean design**.

🔑 Basic Syntax

```
class Parent:
    def func1(self):
        print("This is from Parent class")

class Child(Parent):  # Inherit Parent
    def func2(self):
        print("This is from Child class")

obj = Child()
obj.func1()  # ✅ Parent method
obj.func2()  # ✅ Child method
```

🧱 Types of Inheritance in Python

Type	Structure	When to Use
Single	`B(A)`	Simple extension of one base class.
Multilevel	`C(B(A))`	Layered specializations (chain).
Multiple	`C(A, B)`	Combine behaviors from many bases; prefer **mixins**.
Hierarchical	`B(A)`, `C(A)`	Many children share one base.
Hybrid	Combination	Complex real-world models; mind the **MRO**.

---

1) Single Inheritance

```
class A:
    def feature1(self):
        print("Feature 1 from A")
```

```
class B(A):
    def feature2(self):
        print("Feature 2 from B")
```

2) Multilevel Inheritance

```
class A:
    def feature1(self):
        print("Feature 1")

class B(A):
    def feature2(self):
        print("Feature 2")

class C(B):
    def feature3(self):
        print("Feature 3")

obj = C()
obj.feature1()  # from A
```

3) Multiple Inheritance

```
class A:
    def feature1(self):
        print("Feature 1 from A")

class B:
    def feature2(self):
        print("Feature 2 from B")

class C(A, B):  # Multiple inheritance
    def feature3(self):
        print("Feature 3 from C")

obj = C()
obj.feature1()
obj.feature2()
```

4) Hierarchical Inheritance

```
class Parent:
    def func1(self):
        print("Parent Function")
```

```
class Child1(Parent):
    def func2(self):
        print("Child1 Function")

class Child2(Parent):
    def func3(self):
        print("Child2 Function")
```

5) Hybrid Inheritance + 🔍Diamond Problem

> When a class inherits from two branches that share a common ancestor. Python resolves this using **MRO (Method Resolution Order)**.

```
class A:
    def show(self): print("A")
class B(A): pass
class C(A): pass
class D(B, C): pass

print(D.mro())  # ♎ [D, B, C, A, object]
```

🕐Method Overriding

Child redefines a parent method with the **same name**.

```
class Parent:
    def show(self):
        print("Parent Method")

class Child(Parent):
    def show(self):
        print("Child Method")

Child().show()  # ➜ Child Method
```

🧰`super()` — Call the Parent

Use `super()` inside the child to call the parent's implementation (constructors or methods).

```
class Parent:
    def __init__(self, name):
        print("Parent __init__")
        self.name = name

class Child(Parent):
    def __init__(self, name, age):
        super().__init__(name)        # ✅ calls Parent
        self.age = age
        print("Child __init__")

c = Child("Zahid", 24)
```

You can also **extend behavior**:

```
class Logger:
    def log(self):
        print("log from Logger")

class Service(Logger):
    def log(self):
        super().log()           # keep parent behavior
        print("log from Service")
```

MRO (Method Resolution Order)

- Python follows **C3 linearization** to decide the order of lookup.
- Inspect it with: `ClassName.mro()` or `ClassName.__mro__`.
- In `class D(B, C)`, lookup goes **left-to-right** respecting parent order.

```
class X: pass
class Y: pass
class Z(X, Y): pass
print([cls.__name__ for cls in Z.mro()])  # ['Z', 'X', 'Y', 'object']
```

🔗 Good Practices

- Prefer **composition over inheritance** when relationships aren't "is-a".
- Keep base classes **small & focused**.
- If you must use multiple inheritance, design **mixins** (small, behavior-only bases).
- Always check `` when debugging complex inheritance.

🧪 Quick Self-Check

- Can you explain **single vs multiple** inheritance?
- What does `` ` do in a child's `` **init**`?
- How does Python avoid the **diamond problem**?

🗜️ Mini Challenge

Create a `Vehicle` base with `start()`, then make `Car(Vehicle)` and `ElectricMixin` with `charge()`. Build `Tesla(Car, ElectricMixin)` that overrides `start()` but still calls parent with `super()`.