1. Everything in Python is an Object

Python's philosophy: **"Everything is an object"**. This includes:

- Numbers, strings, lists, dicts
- Functions
- Modules
- Classes themselves

```
x = 42
print(type(x))         # <class 'int'>
print(isinstance(x, object))  # True
```

```
<class 'int'>
True
```

```
def hello():
    return "Hi"

print(type(hello))      # <class 'function'>
print(isinstance(hello, object))  # True
```

```
<class 'function'>
True
```

```
class MyClass:
    pass

print(type(MyClass))    # <class 'type'>
print(isinstance(MyClass, object))  # True
```

```
<class 'type'>
True
```

💡 **Tip:** Since classes are objects, you can:

- Pass classes as arguments to functions.
- Return classes from functions.
- Create classes dynamically using `type()`.

```
# Dynamic class creation
DynamicPerson = type("DynamicPerson", (object,), {"greet": lambda self: "Hello"})
p = DynamicPerson()
print(p.greet())  # Hello
```

```
Hello
```

2. Class Anatomy in Python

A class in Python consists of:

- **Attributes (fields)**: Variables associated with the object
  - Instance attributes (`self.name`)
  - Class attributes (`Person.species`)
- **Methods**: Functions inside classes
  - Instance methods (need `self`)
  - Class methods (use `cls`)
  - Static methods (don't use `self` or `cls`)
- **Special Methods**: Dunder methods (`__init__`, `__str__`, `__add__`, etc.)

```
class Car:
    wheels = 4  # class attribute

    def __init__(self, brand, color):
```

```
        self.brand = brand  # instance attribute
        self.color = color

    def drive(self):
        print(f"{self.brand} is driving.")

    @classmethod
    def car_info(cls):
        print(f"A car usually has {cls.wheels} wheels.")

    @staticmethod
    def honk():
        print("Beep beep!")
```

3. Object Creation & Initialization

- Python calls `__new__()` to **allocate memory**.
- Then `__init__()` **initializes the instance**.

💡 **Tip:** You rarely need `__new__`, but it's useful for **singleton patterns** or **immutable objects** like `int` or `str`.

```
class Example:
    def __new__(cls, *args, **kwargs):
        print("Creating instance...")
        return super().__new__(cls)

    def __init__(self, value):
        print("Initializing instance...")
        self.value = value

e = Example(10)
```

```
Creating instance...
Initializing instance...
```

4. Instance vs Class Attributes

- **Instance attributes**: Unique to each object
- **Class attributes**: Shared across all instances

```
class Student:
    school = "XYZ School"  # class attribute

    def __init__(self, name):
        self.name = name  # instance attribute

s1 = Student("Alice")
s2 = Student("Bob")

print(s1.school, s1.name)  # XYZ School Alice
print(s2.school, s2.name)  # XYZ School Bob

# Changing class attribute
Student.school = "ABC School"
print(s1.school)  # ABC School
```

```
XYZ School Alice
XYZ School Bob
ABC School
```

💡 **Trick:** Avoid mutable class attributes like lists or dicts unless intentional, because all instances share them.

```
class BadExample:
    items = []  # shared mutable list

a = BadExample()
b = BadExample()
```

```
a.items.append(1)
print(b.items)  # [1] → b shares the same list!
```

```
[1]
```

✅ **Fix**: Use instance attributes for mutable defaults.

```
class GoodExample:
    def __init__(self):
        self.items = []
```

5. Inheritance & Polymorphism

- **Single Inheritance**

```
class Animal:
    def speak(self):
        print("Some sound")

class Dog(Animal):
    def speak(self):
        print("Bark")

d = Dog()
d.speak()  # Bark
```

```
Bark
```

- **Multiple Inheritance**

```
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...
```

```
Flying...
Swimming...
```

💡 **Tip:** Python uses **Method Resolution Order (MRO)** to decide which method to call. Check it using:

```
print(Duck.mro())
```

```
[<class '__main__.Duck'>, <class '__main__.Flyer'>, <class '__main__.Swimmer'>, <class 'object'>]
```

Start coding or generate with AI.

1. Quick Recap: What is Inheritance?

Inheritance allows a **child (subclass)** to reuse and extend functionality of a **parent (base class)**.

```
class Animal:
    def eat(self):
        print("This animal eats food.")

class Dog(Animal):
```

```
    def bark(self):
        print("Woof!")
```

2. Method Overriding (Redefining Parent Methods)

Child classes can **override** parent methods.

```
class Animal:
    def sound(self):
        print("Some generic sound")

class Dog(Animal):
    def sound(self):
        print("Woof!")
```

➡️ When calling `Dog().sound()`, Python prefers the child's method.

3. Using `super()` for Parent Access

Sometimes you want to **reuse the parent's version** of a method while extending it.

```
class Animal:
    def sound(self):
        print("Generic animal sound")

class Dog(Animal):
    def sound(self):
        super().sound()   # Call parent method
        print("Woof!")
```

Output:

```
Generic animal sound
Woof!
```

4. Multiple Inheritance

Python allows a class to inherit from **multiple parents**.

```
class Flyer:
    def action(self):
        print("I can fly")

class Swimmer:
    def action(self):
        print("I can swim")

class Duck(Flyer, Swimmer):
    pass

d = Duck()
d.action()   # I can fly (why not swim?)
```

👉 Python follows the **Method Resolution Order (MRO)** — left to right in the class definition (`Flyer` before `Swimmer`).

5. Method Resolution Order (MRO)

Use `ClassName.mro()` or `help(ClassName)` to see the order.

```
print(Duck.mro())
```

Output:

```
[<class '__main__.Duck'>, <class '__main__.Flyer'>, <class '__main__.Swimmer'>, <class 'object'>]
```

✅ Python looks for methods in that exact order.

---

6. Diamond Problem in Inheritance

What if a class inherits from two classes that both inherit from the same parent?

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

class B(A):
    def action(self):
        print("B action")

class C(A):
    def action(self):
        print("C action")

class D(B, C):
    pass

d = D()
d.action()
```

Output:

```
B action
```

✅ Because MRO is `D → B → C → A → object`. Python solves the **diamond problem** automatically using MRO.

---

7. Mixins (Reusable Behaviors)

A **mixin** is a small class meant to add extra functionality, not stand alone.

```
class LoggerMixin:
    def log(self, message):
        print(f"[LOG]: {message}")

class Service(LoggerMixin):
    def process(self):
        self.log("Service started")
        print("Processing...")
        self.log("Service finished")

s = Service()
s.process()
```

Output:

```
[LOG]: Service started
Processing...
[LOG]: Service finished
```

✅ Mixins let you **compose behaviors** across classes.

---

8. Abstract Base Classes (ABCs) + Inheritance

Use `abc` module to enforce **method implementation**.

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

class Circle(Shape):
    def __init__(self, r):
        self.r = r
    def area(self):
        return 3.14 * self.r * self.r

c = Circle(5)
print(c.area())
```

✅ Any subclass **must** implement `area()`.

---

9. Class vs Instance Variables in Inheritance

- **Class variables** are shared across instances.
- **Instance variables** are unique per object.

```
class Animal:
    species = "Unknown"   # class variable
    def __init__(self, name):
        self.name = name  # instance variable

class Dog(Animal):
    species = "Canine"

a = Animal("Thing")
d = Dog("Rex")

print(a.species, a.name)  # Unknown Thing
print(d.species, d.name)  # Canine Rex
```

---

10. Best Practices for Advanced Inheritance

✓ Prefer **composition over inheritance** when possible (e.g., "has-a" instead of "is-a"). ✓ Use **mixins** for small, reusable behaviors. ✓ Keep inheritance trees **shallow** (too many levels → complexity). ✓ Always check `mro()` in multiple inheritance scenarios. ✓ Use `super()` consistently (Python 3's `super()` is safe and preferred).

✅ Summary

- **Method overriding & super()** let child classes reuse/extend parent methods.
- **Multiple inheritance** works with Python's **MRO**.
- **Diamond problem** is solved using C3 linearization.
- **Mixins** are a great way to share reusable behavior.
- **Abstract base classes** enforce structure.
- Be mindful of class vs instance variables.

```
Start coding or generate with AI.
```

```
Start coding or generate with AI.
```

📝 Inheritance Practice Problems

🔷 Beginner Level

Problem 1: Employee Inheritance

Create a base class `Employee` with attributes: `name`, `salary`.

- Add a method `get_details()` that prints employee info.
- Inherit a class `Manager` that adds an attribute `department` and overrides `get_details()` to include it.

Expected Behavior:

```
m = Manager("Alice", 8000, "IT")
m.get_details()
# Output: Name: Alice, Salary: 8000, Department: IT
```

Problem 2: Vehicle Hierarchy

- Define a base class `Vehicle` with method `move()`.
- Create subclasses `Car`, `Bike`, and `Truck` that override `move()` appropriately.

Expected Behavior:

```
v = Car()
v.move()  # Car drives on road
```

🔷 Intermediate Level

Problem 3: Bank Accounts

Create a class `Account` with attributes: `balance`.

- Add `deposit()` and `withdraw()` methods.
- Create subclasses `SavingsAccount` and `CheckingAccount`.
  - `SavingsAccount` should have an `interest_rate` and method `add_interest()`.
  - `CheckingAccount` should charge a fee on withdrawals.

Problem 4: Shapes (Polymorphism + Inheritance)

- Create an abstract class `Shape` with abstract method `area()`.
- Implement subclasses `Circle` and `Rectangle` with their own `area()`.

Expected Behavior:

```
shapes = [Circle(5), Rectangle(4, 6)]
for s in shapes:
    print(s.area())
# Output:
# 78.5
# 24
```

🔷 Advanced Level

Problem 5: Multiple Inheritance – Smart Devices

Create classes:

- `Camera` with method `take_photo()`.
- `Phone` with method `make_call()`.
- `SmartPhone` that inherits from both and adds method `browse_internet()`.

Expected Behavior:

```
s = SmartPhone()
s.take_photo()       # Taking photo...
```

```
s.make_call("123")   # Calling 123...
s.browse_internet()  # Browsing internet...
```

Problem 6: Abstract Transport System

- Create an abstract base class `Transport` with abstract methods `move()` and `capacity()`.
- Create subclasses `Bus`, `Train`, `Airplane`.
- Each subclass must implement both methods differently.

Expected Behavior:

```
t = Train()
t.move()      # Train runs on tracks
print(t.capacity())  # 300 passengers
```

Problem 7: Operator Overloading + Inheritance

- Create a base class `Vector` that stores a list of numbers.
- Implement `__add__` to add two vectors.
- Inherit `NamedVector` that also stores a name and overrides `__str__`.

Expected Behavior:

```
v1 = Vector([1,2,3])
v2 = Vector([4,5,6])
print(v1 + v2)  # Vector([5,7,9])

nv = NamedVector("Speed", [10, 20])
print(nv)  # NamedVector Speed: [10, 20]
```

```
Start coding or generate with AI.
```

```
Start coding or generate with AI.
```

6. Encapsulation & Name Mangling

- **Public** → normal attributes
- **Protected** → prefix `_` (convention, not enforced)
- **Private** → prefix `__` (name mangling)

```
class Secret:
    def __init__(self):
        self.public = "visible"
        self._protected = "semi-hidden"
        self.__private = "hidden"

s = Secret()
print(s.public)       # visible
print(s._protected)   # semi-hidden
# print(s.__private)  # AttributeError
print(s._Secret__private)  # hidden (name mangling)
```

```
visible
semi-hidden
hidden
```

💡 **Trick:** Name mangling avoids accidental overrides in subclasses.

```
Start coding or generate with AI.
```

7. Property Decorators (`@property`)

Pythonic way to create **getters and setters**.

🔷 What is `@property`?

The `@property` decorator allows you to turn a **method** into an **attribute**. This lets you **access methods like attributes** without calling them with parentheses.

It's part of Python's **descriptor protocol** and is widely used for **encapsulation**.

🔷 Example: Without vs With `@property`

✅ Without `@property`

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

    def get_area(self):   # method
        return 3.14 * self.radius ** 2

c = Circle(5)
print(c.get_area())   # must call like a function
```

✅ With `@property`

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

    @property
    def area(self):   # property method
        return 3.14 * self.radius ** 2

c = Circle(5)
print(c.area)   # looks like an attribute, but calls the method
```

👉 Now `c.area` behaves like an **attribute**, but behind the scenes, it runs the method.

🔷 Why use `@property`?

1. **Encapsulation (getter/setter control)** You can control access to an attribute — e.g., validate before setting.

```
class Person:
    def __init__(self, name):
        self._name = name   # private attribute (by convention)

    @property
    def name(self):   # getter
        return self._name

    @name.setter
    def name(self, value):   # setter
        if not value.strip():
            raise ValueError("Name cannot be empty!")
        self._name = value

    @name.deleter
    def name(self):   # deleter
        print("Deleting name...")
        del self._name

p = Person("Ali")
```

```
print(p.name)     # calls getter
p.name = "Sara"   # calls setter
del p.name        # calls deleter
```

---

2. **Read-only attributes** Sometimes you want an attribute that can be *read* but not *modified*.

```
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height

    @property
    def area(self):
        return self.width * self.height

r = Rectangle(4, 5)
print(r.area)   # 20
r.area = 100    # ❌ AttributeError (read-only)
```

---

3. **Cleaner API** You can use **attribute-style access** for something that is actually computed dynamically.

🔷 How it works under the hood

The `@property` decorator is actually a **class** (like `@classmethod`). Simplified version:

```
class property:
    def __init__(self, fget=None, fset=None, fdel=None):
        self.fget = fget
        self.fset = fset
        self.fdel = fdel

    def __get__(self, instance, owner):
        return self.fget(instance)

    def __set__(self, instance, value):
        if self.fset is None:
            raise AttributeError("can't set attribute")
        self.fset(instance, value)

    def __delete__(self, instance):
        if self.fdel is None:
            raise AttributeError("can't delete attribute")
        self.fdel(instance)
```

So:

- `@property` → makes a getter
- `@x.setter` → makes a setter
- `@x.deleter` → makes a deleter

✅ Summary

- `@property` turns a method into an attribute.
- Allows **getter/setter/deleter** control while keeping a clean syntax.
- Useful for **encapsulation, validation, computed attributes, and read-only values**.

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

    @property
    def radius(self):
        return self._radius
```

```
    @radius.setter
    def radius(self, value):
        if value <= 0:
            raise ValueError("Radius must be positive")
        self._radius = value

c = Circle(5)
print(c.radius)  # 5
c.radius = 10    # OK
# c.radius = -3  # ValueError
```

```
5
```

💡 **Tip:** Use `@property` to keep a clean API without exposing internal attributes.

8. Special Methods (Magic / Dunder Methods)

Method	Purpose
`__init__`	Constructor
`__new__`	Memory allocation
`__str__`	Human-readable string
`__repr__`	Official string representation
`__len__`	Support `len(obj)`
`__getitem__`	Indexing support `obj[key]`
`__setitem__`	Setting item `obj[key]=value`
`__add__`	Overload `+` operator
`__call__`	Make object callable like a function

Example: Custom Vector Class

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

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

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

v1 = Vector(1, 2)
v2 = Vector(3, 4)
print(v1 + v2)  # Vector(4, 6)
```

```
Vector(4, 6)
```

💡 **Trick:** Implementing `__repr__` properly makes debugging easier.

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

    def __repr__(self):
        return f"Person(name='{self.name}', age={self.age})"

    def __str__(self):
        return f"{self.name}, {self.age} years old"

p = Person("Alice", 30)

print(repr(p))   # Person(name='Alice', age=30)
print(str(p))    # Alice, 30 years old
print(p)         # Alice, 30 years old (because print() calls __str__)
```

```
Start coding or generate with AI.
```

Start coding or generate with AI.

📝 Encapsulation Practice Problems

🔷 Beginner Level

Problem 1: Bank Account (Private Balance)

Create a class `BankAccount` with:

- A **private attribute** `__balance`.
- Methods `deposit(amount)` and `withdraw(amount)` that modify the balance safely.
- A method `get_balance()` to check the balance.

Expected Behavior:

```
acc = BankAccount(1000)
acc.deposit(500)
print(acc.get_balance())   # 1500
acc.withdraw(2000)         # Error: Insufficient balance
```

Problem 2: Student Data

Create a `Student` class with **private attributes** `__name` and `__grade`.

- Provide getter and setter methods for both.
- Validate that `grade` is always between 0 and 100.

🔷 Intermediate Level

Problem 3: Employee Salary Protection

Create a class `Employee`:

- Store `__salary` as private.
- Add a setter that prevents setting salary below 3000.
- Add a getter to display salary.

Expected Behavior:

```
emp = Employee("Ali", 2500)
# Should raise: Salary must be >= 3000
```

Problem 4: Encapsulated Shopping Cart

Create a `ShoppingCart` class:

- Store items in a private list `__items`.
- Provide methods `add_item(item)`, `remove_item(item)`, and `view_cart()`.
- Prevent direct access to `__items`.

🔷 Advanced Level

Problem 5: Property Decorator with Encapsulation

Create a class `Temperature`:

- Private attribute `__celsius`.
- Use `@property` and `@setter` to allow reading/writing temperature.
- Automatically convert negative values to `0` (no negative temperatures).

Expected Behavior:

```
t = Temperature(25)
print(t.celsius)   # 25
t.celsius = -10
print(t.celsius)   # 0
```

Problem 6: Secure Login System

Create a class `User`:

- Private attributes: `__username`, `__password`.
- Provide a method `check_password(pwd)` to validate login.
- Disallow direct access to password.

Problem 7: Banking with Inheritance + Encapsulation

Create a base class `Account`:

- Private balance with getters/setters.
- Methods `deposit()` and `withdraw()`.

Create subclass `SavingsAccount`:

- Adds interest rate.
- Method `add_interest()` updates balance.

Expected Behavior:

```
s = SavingsAccount(1000, 0.05)
s.add_interest()
print(s.get_balance())   # 1050
```

```
Start coding or generate with AI.
```

```
Start coding or generate with AI.
```

9. Classmethods & Staticmethods

```
class Temperature:
    scale = "Celsius"

    @classmethod
    def set_scale(cls, new_scale):
        cls.scale = new_scale

    @staticmethod
    def c_to_f(c):
        return c * 9/5 + 32

Temperature.set_scale("Fahrenheit")
print(Temperature.scale)  # Fahrenheit
print(Temperature.c_to_f(0))  # 32
```

```
Fahrenheit
32.0
```

Rules:

- `@staticmethod`: No access to `cls` or `self`
- `@classmethod`: Access to class (`cls`) but not instance (`self`)

---

10. Metaclasses (Advanced)

- **Metaclasses** define **how classes themselves are created**.
- Everything is an object; classes are instances of `type`.

```
class Meta(type):
    def __new__(cls, name, bases, dct):
        print(f"Creating class {name}")
        return super().__new__(cls, name, bases, dct)

class MyClass(metaclass=Meta):
    pass

# Output: Creating class MyClass
```

```
Creating class MyClass
```

💡 **Tip:** Metaclasses are powerful for **frameworks, ORM models**, or **automatic registration of classes**.

11. Python Tips & Tricks with Classes

1. **Dynamic attributes**

```
class Person: pass
p = Person()
p.name = "Alice"  # Add attribute at runtime
```

2. **Dynamic methods**

```
def greet(self):
    print("Hello!")
import types
p.say_hello = types.MethodType(greet, p)
p.say_hello()  # Hello!
```

```
Hello!
```

3. **Using `__slots__`** to save memory for many objects

🔷 Code

```
class Point:
    __slots__ = ("x", "y")

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

🔷 What `__slots__` Does

Normally, Python stores an object's attributes in a **dictionary (`__dict__`)**, so you can dynamically add new attributes at runtime.

But when you define `__slots__`, Python:

1. **Restricts attributes**
   - Only the names listed in `__slots__` (`"x"` and `"y"`) can be assigned.
   - You **cannot** add new attributes dynamically.

   ```
   p = Point(3, 4)
   p.z = 10   # ❌ AttributeError: 'Point' object has no attribute 'z'
   ```

2. **Saves memory**
   - Instead of a per-object dictionary, Python uses a more compact structure.
   - Useful when creating **many instances** (e.g., millions of `Point`s).

3. **Slightly faster attribute access**

- Because attribute lookups avoid dictionary overhead.

🔷 Example Usage

```
p1 = Point(1, 2)
print(p1.x, p1.y)   # 1 2

p1.x = 10
print(p1.x)         # 10

# ❌ Can't add new attributes
p1.z = 5
# AttributeError: 'Point' object has no attribute 'z'
```

✅ **Summary:**

- `__slots__` **limits attributes** and **reduces memory usage**.
- Good for performance in **data-heavy classes** like points, vectors, or signal samples.
- But: it removes flexibility (no `__dict__`, no dynamic attributes, and no multiple inheritance in some cases).

```
class Point:
    __slots__ = ("x", "y")
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

4. **Callable objects**

```
class Adder:
    def __init__(self, n):
        self.n = n
    def __call__(self, x):
        return x + self.n

add5 = Adder(5)
print(add5(10))  # 15
```

```
15
```

Great question 👍 Let's break down **`__call__`** in Python.

🔷 What is `__call__`?

- `__call__` is a **special method** in Python.
- If you define it in a class, then the **instance of that class becomes callable like a function**.

👉 When you write:

```
obj(x, y)
```

Python actually runs:

```
obj.__call__(x, y)
```

🔷 Example

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

    def __call__(self, greeting):
        return f"{greeting}, {self.name}!"
```

```
greet_john = Greeter("John")
print(greet_john("Hello"))  # Hello, John!
print(greet_john("Hi"))     # Hi, John!
```

✅ Here, `greet_john("Hello")` is the same as `greet_john.__call__("Hello")`.

🔷 Why is it Useful?

1. **Makes objects behave like functions**
   - Function-like syntax but with the power of a class.
2. **Keeps state between calls**
   - Unlike normal functions, the object can remember data in its attributes.
3. **Used in frameworks**
   - **PyTorch / TensorFlow** models use `__call__` so you can do:

     ```
     output = model(input_data)
     ```

     instead of `model.forward(input_data)`.
4. **Decorators / Wrappers**
   - `__call__` allows creating objects that wrap and modify other functions.

🔷 Mini Real-World Example

```
class Power:
    def __init__(self, exp):
        self.exp = exp

    def __call__(self, x):
        return x ** self.exp

square = Power(2)
cube   = Power(3)

print(square(5))  # 25
print(cube(5))    # 125
```

- `square` acts like a function that squares numbers.
- `cube` acts like a function that cubes numbers.

✅ **Summary:** `__call__` makes class instances behave like functions while keeping state. It's especially useful in **ML models, decorators, and functional-style programming**.

Would you like me to also show you the **difference between using `__call__` vs a normal method** (like `add()` or `forward()`)?

12. Summary

- **Classes = Blueprints**, objects = instances.
- **Everything in Python is an object**, even classes.
- **Attributes**: Instance vs Class
- **Methods**: Instance, Class, Static
- **Encapsulation**: Public, Protected, Private
- **Magic Methods**: Overload operators, indexing, calling
- **Inheritance**: Single & Multiple
- **Metaclasses**: Customize class creation
- **Tips/Tricks**: Dynamic attributes, `__slots__`, callable objects, singletons.

Start coding or generate with AI.

🔷 `@abstractmethod` in Python

1. What is it?

- `@abstractmethod` is a **decorator** from the `abc` (Abstract Base Classes) module.
- It marks a method as **abstract**, meaning subclasses **must** override/implement it.
- You can't instantiate a class that has unimplemented abstract methods.

👉 It enforces a **contract**: all subclasses must implement the required methods.

---

2. Basic Example

```
from abc import ABC, abstractmethod

class Animal(ABC):  # Abstract Base Class
    @abstractmethod
    def make_sound(self):
        pass

class Dog(Animal):
    def make_sound(self):
        return "Woof!"

class Cat(Animal):
    def make_sound(self):
        return "Meow!"

# a = Animal()  # ❌ Error: Can't instantiate abstract class
d = Dog()
print(d.make_sound())  # Woof!
```

✅ Here:

- `Animal` defines a **blueprint** with `make_sound()`.
- Subclasses **must** implement it.

---

3. Abstract Methods with Implementation

You can even provide a **default implementation**:

```
from abc import ABC, abstractmethod

class Vehicle(ABC):
    @abstractmethod
    def move(self):
        print("This vehicle moves...")  # Default behavior

class Car(Vehicle):
    def move(self):
        super().move()
        print("Car is driving on the road.")

c = Car()
c.move()
# Output:
# This vehicle moves...
# Car is driving on the road.
```

---

4. Abstract Properties and Static/Class Methods

You can also apply `@abstractmethod` to `@property`, `@classmethod`, and `@staticmethod`:

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @property
    @abstractmethod
    def area(self):
        pass

    @staticmethod
    @abstractmethod
    def shape_type():
        pass

class Circle(Shape):
    def __init__(self, r):
        self.r = r

    @property
    def area(self):
        return 3.14 * self.r**2

    @staticmethod
    def shape_type():
        return "Circle"

c = Circle(5)
print(c.area)         # 78.5
print(c.shape_type()) # Circle
```

---

5. Why Use `@abstractmethod`?

- **Enforces consistency** → All subclasses follow the same interface.
- **Improves readability** → Declares "what must exist".
- **Helps large teams** → Developers know what to implement.

---

6. Real-World Analogy

Think of it like a **job contract**:

- The company (abstract class) says: "Every employee must have a `work()` method."
- Each role (subclass) defines how they "work".

```
Start coding or generate with AI.
```

```
Start coding or generate with AI.
```

Nice 👍 I see your notebook covers **everything about Python classes**: class anatomy, attributes, inheritance, encapsulation, properties, magic methods, static/class methods, metaclasses, and even tricks like `__slots__` and singletons .

Let's build a **set of exercises** based directly on these topics, moving from beginner → advanced → challenge level.

📝 Exercises on Python Classes

🔷 Beginner Level

Exercise 1: Class Anatomy

1. Create a class `Car` with:
    - Class attribute: `wheels = 4`

- Instance attributes: `brand`, `color`
- Method `drive()` that prints `<brand> is driving.`

2. Create two cars and test the method.

Exercise 2: Instance vs Class Attributes

1. Make a class `Student` with:
   - Class attribute: `school = "XYZ School"`
   - Instance attribute: `name`
2. Create two students and show how changing `Student.school` affects both objects.

Exercise 3: Encapsulation

1. Write a class `Secret` with:
   - Public attribute `visible`
   - Protected attribute `_hidden`
   - Private attribute `__very_hidden`
2. Access them in different ways.
3. Print the name-mangled version of `__very_hidden`.

🔷 Intermediate Level

Exercise 4: Inheritance & Polymorphism

1. Create base class `Animal` with method `speak()`.
2. Subclasses: `Dog` (says "Bark"), `Cat` (says "Meow").
3. Write a loop that calls `speak()` on a list of animals.

Exercise 5: Property Decorators

1. Make a class `Circle` with a private attribute `_radius`.
2. Add a `@property` to get it.
3. Add a setter that raises `ValueError` if radius ≤ 0.
4. Test with valid and invalid values.

Exercise 6: Special Methods

1. Implement a class `Vector(x, y)` with:
   - `__add__` to support `+`
   - `__repr__` for readable display
2. Test `v1 + v2`.

Exercise 7: Classmethods & Staticmethods

1. Create `Temperature` with class attribute `scale = "Celsius"`.
2. Add:
   - `@classmethod set_scale(new_scale)`
   - `@staticmethod c_to_f(celsius)`
3. Test both.

🔷 Advanced Level

Exercise 8: Using `__new__`

1. Create a class `Example` that prints messages inside `__new__` and `__init__`.
2. Observe the order of execution when creating an instance.

Exercise 9: Metaclasses

1. Write a metaclass `Meta` that prints the class name being created.
2. Create a class `MyClass` with this metaclass.

3. What output do you see?

Exercise 10: Dynamic Attributes and Methods

1. Create a `Person` instance dynamically add an attribute `age`.
2. Attach a new method `say_hello(self)` dynamically using `types.MethodType`.
3. Call the method.

Exercise 11: `__slots__` for Memory Optimization

1. Create class `Point` with `__slots__ = ("x", "y")`.
2. Try to assign `p.z = 5`.
3. What error do you get?

Exercise 12: Singleton Pattern

1. Implement a `Singleton` class that only allows one instance.
2. Create two objects and confirm `a is b`.

🔥 Challenge Exercises

Challenge 1: Mini Banking System

- Create a class `BankAccount` with private attribute `__balance`.
- Add methods:
  - `deposit(amount)`
  - `withdraw(amount)` (with error if insufficient balance)
  - `@property balance` (read-only)
- Create two accounts and test transactions.

Challenge 2: Library System

- Class `Book(title, author)` with `__repr__`.
- Class `Library` with methods:
  - `add_book(book)`
  - `list_books()`
- Use a class attribute `books = []` (and show why this is problematic).
- Fix by using an instance attribute.

Challenge 3: Callable Objects

- Create a class `Multiplier` with `__call__` that multiplies input by a fixed factor.
- Example:

```
double = Multiplier(2)
print(double(10))  # 20
```

Would you like me to **also provide solutions (code)** for these exercises, or keep them as assignments for you to try?

```
Start coding or generate with AI.
```

Perfect 🚀 Let's solve all the **class exercises** I gave you. I'll provide **Python code solutions** with explanations.

✅ Solutions: Python Class Exercises

🔷 Beginner Level

Exercise 1: Class Anatomy

```
class Car:
    wheels = 4  # class attribute

    def __init__(self, brand, color):
        self.brand = brand      # instance attribute
        self.color = color

    def drive(self):
        print(f"{self.brand} is driving.")

# Test
c1 = Car("Toyota", "Red")
c2 = Car("BMW", "Black")
c1.drive()  # Toyota is driving.
c2.drive()  # BMW is driving.
```

Exercise 2: Instance vs Class Attributes

```
class Student:
    school = "XYZ School"

    def __init__(self, name):
        self.name = name

s1 = Student("Alice")
s2 = Student("Bob")

print(s1.school, s1.name)  # XYZ School Alice
print(s2.school, s2.name)  # XYZ School Bob

# Change class attribute
Student.school = "ABC School"
print(s1.school)  # ABC School
print(s2.school)  # ABC School
```

Exercise 3: Encapsulation

```
class Secret:
    def __init__(self):
        self.public = "visible"
        self._protected = "semi-hidden"
        self.__private = "hidden"

s = Secret()
print(s.public)       # visible
print(s._protected)   # semi-hidden
# print(s.__private)  # AttributeError
print(s._Secret__private)  # hidden (via name mangling)
```

🔷 Intermediate Level

Exercise 4: Inheritance & Polymorphism

```
class Animal:
    def speak(self):
        print("Some sound")

class Dog(Animal):
    def speak(self):
        print("Bark")
```

```
class Cat(Animal):
    def speak(self):
        print("Meow")

animals = [Dog(), Cat(), Animal()]
for a in animals:
    a.speak()
# Bark, Meow, Some sound
```

Exercise 5: Property Decorators

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

    @property
    def radius(self):
        return self._radius

    @radius.setter
    def radius(self, value):
        if value <= 0:
            raise ValueError("Radius must be positive")
        self._radius = value

c = Circle(5)
print(c.radius)  # 5
c.radius = 10
print(c.radius)  # 10
# c.radius = -3  # ValueError
```

Exercise 6: Special Methods

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

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

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

v1 = Vector(1, 2)
v2 = Vector(3, 4)
print(v1 + v2)  # Vector(4, 6)
```

Exercise 7: Classmethods & Staticmethods

```
class Temperature:
    scale = "Celsius"

    @classmethod
    def set_scale(cls, new_scale):
        cls.scale = new_scale

    @staticmethod
    def c_to_f(c):
        return c * 9/5 + 32

Temperature.set_scale("Fahrenheit")
```

```
print(Temperature.scale)  # Fahrenheit
print(Temperature.c_to_f(0))  # 32.0
```

🔷 Advanced Level

Exercise 8: Using `__new__`

```
class Example:
    def __new__(cls, *args, **kwargs):
        print("Creating instance...")
        return super().__new__(cls)

    def __init__(self, value):
        print("Initializing instance...")
        self.value = value

e = Example(10)
# Output:
# Creating instance...
# Initializing instance...
```

Exercise 9: Metaclasses

```
class Meta(type):
    def __new__(cls, name, bases, dct):
        print(f"Creating class {name}")
        return super().__new__(cls, name, bases, dct)

class MyClass(metaclass=Meta):
    pass
# Output: Creating class MyClass
```

Exercise 10: Dynamic Attributes and Methods

```
import types

class Person: pass

p = Person()
p.age = 25  # dynamic attribute
print(p.age)  # 25

def greet(self):
    print("Hello!")

p.say_hello = types.MethodType(greet, p)
p.say_hello()  # Hello!
```

Exercise 11: `__slots__`

```
class Point:
    __slots__ = ("x", "y")
    def __init__(self, x, y):
        self.x = x
        self.y = y

p = Point(1, 2)
print(p.x, p.y)  # 1 2
# p.z = 5  # AttributeError: 'Point' object has no attribute 'z'
```

Exercise 12: Singleton Pattern

```
class Singleton:
    _instance = None
    def __new__(cls, *args, **kwargs):
        if not cls._instance:
            cls._instance = super().__new__(cls)
        return cls._instance

a = Singleton()
b = Singleton()
print(a is b)  # True
```

🔥 Challenge Exercises

Challenge 1: Mini Banking System

```
class BankAccount:
    def __init__(self, balance=0):
        self.__balance = balance

    def deposit(self, amount):
        self.__balance += amount

    def withdraw(self, amount):
        if amount > self.__balance:
            raise ValueError("Insufficient balance")
        self.__balance -= amount

    @property
    def balance(self):
        return self.__balance

# Test
acc1 = BankAccount(100)
acc1.deposit(50)
print(acc1.balance)  # 150
acc1.withdraw(70)
print(acc1.balance)  # 80
```

Challenge 2: Library System

```
class Book:
    def __init__(self, title, author):
        self.title, self.author = title, author

    def __repr__(self):
        return f"{self.title} by {self.author}"

class Library:
    def __init__(self):
        self.books = []  # ✅ instance attribute

    def add_book(self, book):
        self.books.append(book)

    def list_books(self):
        return self.books

# Test
lib = Library()
```

```
lib.add_book(Book("1984", "George Orwell"))
lib.add_book(Book("Python 101", "Guido van Rossum"))
print(lib.list_books())
```

Challenge 3: Callable Objects

```
class Multiplier:
    def __init__(self, factor):
        self.factor = factor

    def __call__(self, x):
        return x * self.factor

double = Multiplier(2)
print(double(10))  # 20
```

✅ That’s a **complete solution set** for all exercises.

Do you want me to also **design a set of real-world project-style class exercises** (e.g., Student Management, Shopping Cart, Hospital System) that combine multiple concepts like inheritance, encapsulation, and magic methods?

Double-click (or enter) to edit