**CodeBash · codebash.co.uk**

A-LEVEL RESOURCE

# A-Level OOP Teaching and Learning Booklet

## Python

A complete teaching and learning booklet covering OOP principles, concepts, exemplar A-Level exam questions with mark schemes, and three in-depth capstone programs with step-by-step walkthroughs.

**Created by Eoin Shannon**

CodeBash

Get free resources at **codebash.co.uk/resources**

# Contents

## PART 5 - CAPSTONE PROGRAMS



## PART 6 - ANSWERS

# Part 1 - Introduction to Object-Oriented Programming

What OOP is, why it exists, and how it compares to procedural programming

## 1.1 What is Object-Oriented Programming?

Object-Oriented Programming (OOP) is a programming paradigm that organises code around objects rather than actions. An object combines data (attributes) and the functions that operate on that data (methods) into a single unit. A class acts as the blueprint for creating objects - you define a class once and can create as many objects from it as you need.

## 1.2 Why Use OOP?

- Real-world problems map naturally onto objects. A school system has students, teachers, and classes - each becomes an object with its own data and behaviour.
- Large programs become manageable by splitting responsibility across classes. Each class has a clear, focused job.
- Code can be reused through inheritance. A general class can be extended without rewriting it.
- Changes to one class are less likely to break unrelated parts of the program.
- Multiple programmers can work on different classes simultaneously without conflict.

## 1.3 Advantages and Disadvantages

| Advantages | Disadvantages |
|---|---|
| Reusability - classes can be reused across projects without modification. | Can be over-engineered for small programs - a simple script does not need a class hierarchy. |
| Maintainability - fixing a bug in one class fixes it everywhere that class is used. | Steeper learning curve than procedural programming for beginners. |
| Modularity - each class is a self-contained unit that can be tested independently. | More memory usage - each object carries its own data, plus overhead from the class structure. |
| Scalability - new features can be added as new classes or by extending existing ones. | Poorly designed class hierarchies are harder to change than flat procedural code. |
| Models the real world - objects represent real entities, making programs easier to reason about. | Excessive inheritance chains make code difficult to trace and debug. |

## 1.4 OOP vs Procedural - Same Problem, Two Approaches

The same problem solved two ways - calculating the area and perimeter of a rectangle. The procedural version uses standalone functions and passes data around as arguments. The OOP version bundles the data and functions together inside a class.

**Procedural Approach**

```
def area(length, width):
    return length * width

def perimeter(length, width):
    return 2 * (length + width)

length = 8
width  = 5
print(f"Area: {area(length, width)}")
print(f"Perimeter: {perimeter(length,
width)}")
```

**OOP Approach**

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

    def area(self):
        return self.length *
self.width

    def perimeter(self):
        return 2 * (self.length +
self.width)

r = Rectangle(8, 5)
print(f"Area: {r.area()}")
print(f"Perimeter: {r.perimeter()}")
```

**NOTE**

- Both produce the same output. The OOP version becomes more useful as the program grows - you can create many Rectangle objects without passing length and width to every function call. The rectangle knows its own dimensions.

## 1.5 OOP vs Procedural - Feature Comparison

| Feature | Procedural Approach | OOP Approach |
|---|---|---|
| **Data organisation** | Data passed between standalone functions as arguments. Global variables used for shared state. | Data encapsulated inside objects. Each object owns its data and controls access to it. |
| **Code reuse** | Functions copied or modified for similar behaviour. No formal mechanism for sharing logic. | Inheritance lets subclasses reuse and extend parent class behaviour without rewriting it. |
| **Managing change** | Changing a function's signature may break all callers. Changes ripple unpredictably. | Encapsulation isolates change. Internal class changes do not affect code that uses the class. |
| **Data protection** | Any part of the program can read or write any variable. No built-in protection. | Private attributes hide data. Access is controlled through |

| | | |
|---|---|---|
| | | methods. Validation can be enforced. |
| **Modelling real-world systems** | Requires mental translation of real-world entities into functions and variables. | Objects directly represent real-world entities — a Student object models a real student. |
| **Extensibility** | Adding features often requires modifying existing functions, risking new bugs. | New behaviour added as new classes or subclasses. Existing code stays unchanged. |

## 1.6 Retrieval Quiz - Introduction

**Quick Recall**

Answer each question without looking back, then check your answers.

**Q1**
What is a class?

**Q2**
What is an object?

**Q3**
Name the four pillars of OOP.

**Q4**
What is a method?

**Q5**
Give one advantage of OOP over procedural programming.

Answers in Part 6 at the back of this booklet

# Part 2 - The Four Pillars of OOP

Encapsulation, Inheritance, Polymorphism, and Abstraction - each with worked examples, diagrams, debugging tasks, and mini challenges

## 2.1 - Encapsulation

Keeping data and behaviour together, and hiding implementation details.

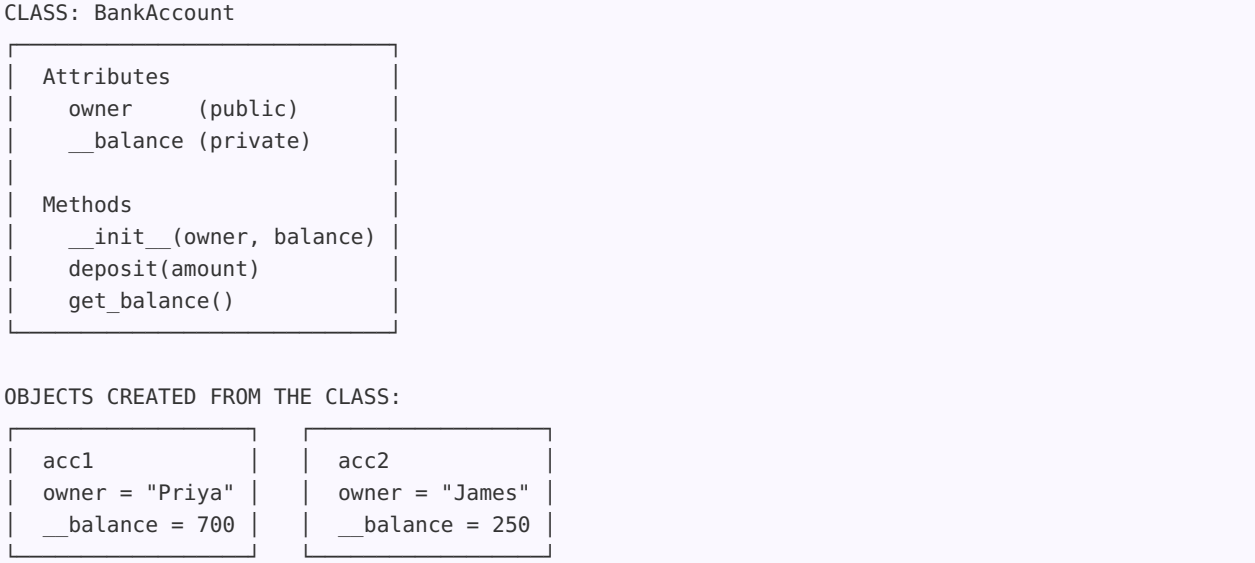
Encapsulation means bundling an object's data (attributes) and the code that works on that data (methods) inside the same class. It also means restricting direct access to internal data from outside the class. In Python, attributes are made private by prefixing their name with a double underscore (__). External code must use methods or properties to read or change them. This protects the object from being put into an invalid state.

**EXAMPLE**

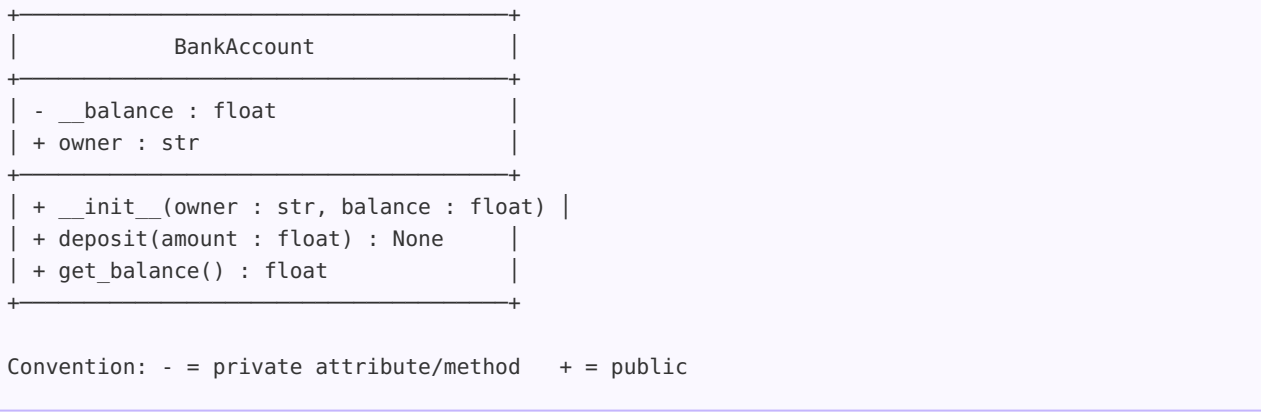
```
class BankAccount:
    def __init__(self, owner, opening_balance):
        self.owner    = owner
        self.__balance = opening_balance   # private

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount

    def get_balance(self):
        return self.__balance

acc = BankAccount("Priya", 500)
acc.deposit(200)
print(acc.get_balance())   # 700
# print(acc.__balance)     # AttributeError - cannot access directly
```

Direct access to __balance is blocked. The only way to change or read it is through the methods the class provides. This is the 'contract' between the class and the code that uses it.

### OBJECT DIAGRAM




### UML CLASS DIAGRAM




### ⚠ COMMON MISTAKES

- Using a single underscore (_balance) instead of double (__balance). Single underscore is just a naming convention — it provides no protection. Double underscore triggers Python's name mangling.
- Trying to access acc.__balance directly. Python name-mangles this to acc._BankAccount__balance, so the direct access raises AttributeError.
- Forgetting self. before attribute names inside methods, creating a local variable that disappears when the method ends instead of an instance attribute.
- Defining the method without self as the first parameter. Python passes the object automatically, so self must be there to receive it.

### Predict the Output (1 of 2)

What will this program output? Write your answer before revealing it.

```
class Counter:
    def __init__(self):
        self.__count = 0

    def increment(self):
        self.__count += 1

    def value(self):
        return self.__count

c = Counter()
c.increment()
c.increment()
c.increment()
print(c.value())
```

Output answer in Part 6 at the back

### Predict the Output (2 of 2)

What will this program output? Write your answer before revealing it.

```
class Box:
    def __init__(self, label):
        self.label = label

b1 = Box("Alpha")
b2 = Box("Beta")
b1.label = "Gamma"
print(b1.label)
print(b2.label)
```

Output answer in Part 6 at the back

### Debug Task

The Scoreboard class below should track a high score, but it crashes. Find and fix the bug.

```
class Scoreboard:
    def __init__(self):
        high_score = 0          # line 3

    def update(self, score):
        if score > self.high_score:   # AttributeError here
            self.high_score = score

    def show(self):
        print(f"High score: {self.high_score}")

s = Scoreboard()
s.update(150)
s.show()
```

Bug and fix in Part 6 at the back

### Mini Challenge - Temperature Converter Class

Create a Temperature class that stores a temperature privately and provides safe access to it.

**Requirements:**

- Private attribute __celsius.
- Constructor accepts an initial celsius value.
- Method to_fahrenheit() returns (celsius × 9/5) + 32.
- Method set_celsius(value) only accepts values >= -273.15 (absolute zero). Print an error message for invalid input.
- Method get_celsius() returns the stored temperature.

**STARTER CODE**

```
class Temperature:
    def __init__(self, celsius):
        # store celsius privately
        pass

    def to_fahrenheit(self):
        pass

    def set_celsius(self, value):
        # validate before setting
        pass

    def get_celsius(self):
        pass
```

## 2.2 - Inheritance

Building new classes on top of existing ones.

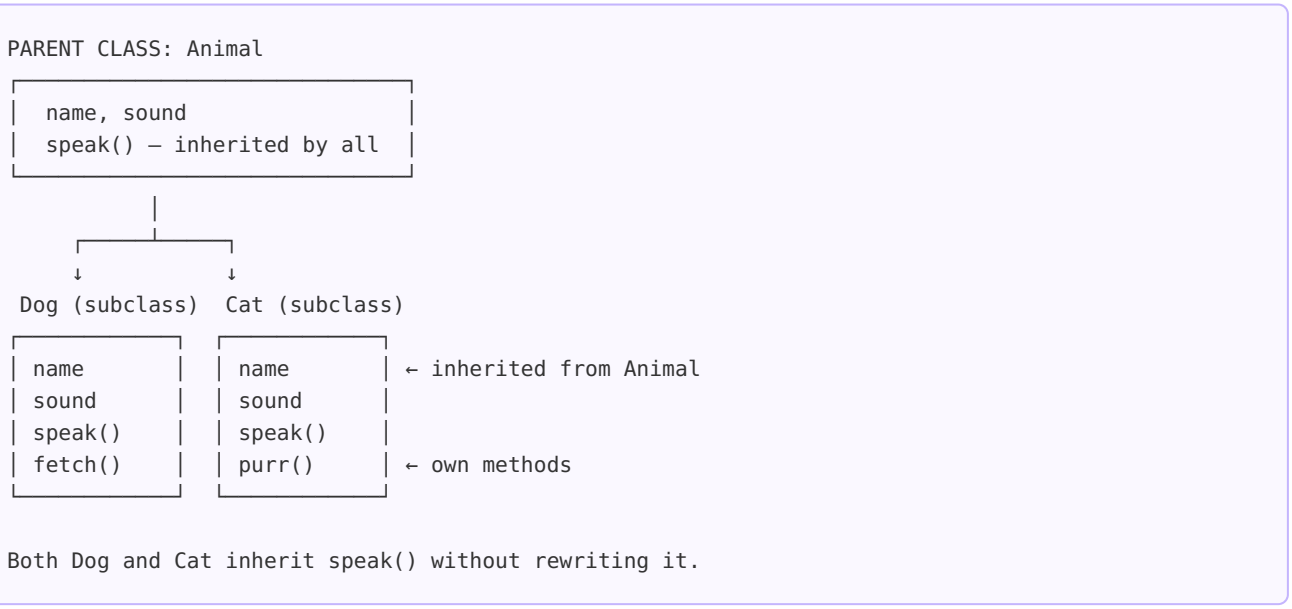
Inheritance allows a new class (the subclass) to take on all the attributes and methods of an existing class (the superclass or parent class). The subclass can then add extra attributes and methods, or override inherited ones. This avoids duplicating code when multiple classes share common behaviour. In Python, the parent class is named in parentheses after the subclass name.

### EXAMPLE

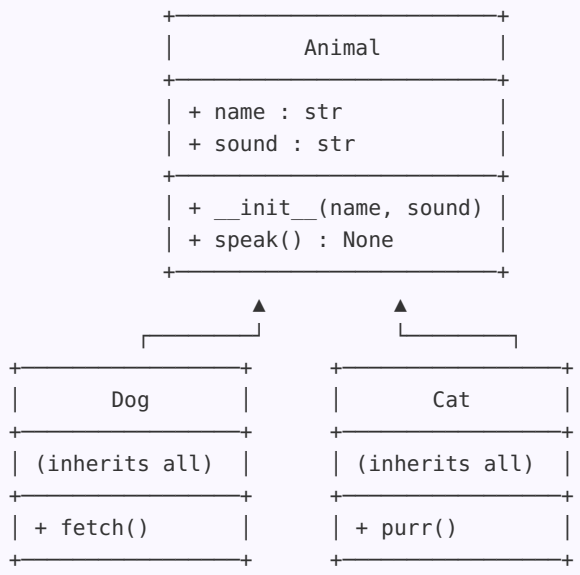
```
class Animal:
    def __init__(self, name, sound):
        self.name  = name
        self.sound = sound

    def speak(self):
        print(f"{self.name} says {self.sound}")

class Dog(Animal):
    def fetch(self):
        print(f"{self.name} fetches the ball!")

class Cat(Animal):
    def purr(self):
        print(f"{self.name} purrs...")

d = Dog("Rex", "woof")
d.speak()    # Rex says woof
d.fetch()    # Rex fetches the ball!

c = Cat("Luna", "meow")
c.speak()    # Luna says meow
```

Dog and Cat both inherit speak() from Animal without rewriting it. Each adds its own unique method. Changing speak() in Animal automatically updates both subclasses.

### OBJECT DIAGRAM



### UML CLASS DIAGRAM




▲ = "inherits from" (hollow arrowhead in formal UML)

### ⚠ COMMON MISTAKES

- Adding __init__ to a subclass but forgetting to call super().__init__(). The parent's attributes are never set, causing AttributeError when methods try to use them.
- Overriding a parent method without calling super().method_name() when you still want the parent's behaviour to run as well.
- Confusing the parent class with the subclass. The subclass IS-A parent (a Dog IS-A Animal), not the other way around.
- Inheriting from a class when composition would be more appropriate. Use inheritance for IS-A relationships, not HAS-A.

## Predict the Output (1 of 2)

What will this program output? Write your answer before revealing it.

```
class Vehicle:
    def __init__(self, make):
        self.make = make

    def describe(self):
        print(f"Vehicle: {self.make}")

class Car(Vehicle):
    def describe(self):
        print(f"Car: {self.make}")

v = Vehicle("Ford")
c = Car("Toyota")
v.describe()
c.describe()
```

Output answer in Part 6 at the back

## Predict the Output (2 of 2)

What will this program output? Write your answer before revealing it.

```
class Parent:
    def greet(self):
        print("Hello from Parent")

class Child(Parent):
    def greet(self):
        super().greet()
        print("Hello from Child")

c = Child()
c.greet()
```

Output answer in Part 6 at the back

### Debug Task

The Dog class below crashes when fetch() is called. Find and fix the bug.

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

    def speak(self):
        print(f"{self.name} makes a sound")

class Dog(Animal):
    def __init__(self, name, breed):
        self.breed = breed     # line 10 — something is missing here

    def fetch(self):
        print(f"{self.name} fetches the ball!")  # AttributeError

d = Dog("Rex", "Labrador")
d.fetch()
```

Bug and fix in Part 6 at the back

### Mini Challenge - Shape Hierarchy

Build a simple inheritance chain for shapes.

**Requirements:**

- Base class Shape: constructor takes colour. Method describe() prints 'A [colour] shape'.
- Subclass Circle: inherits Shape, adds radius. Overrides describe() to print 'A [colour] circle with radius [radius]'.
- Subclass Rectangle: inherits Shape, adds length and width. Overrides describe() appropriately.
- Create one object of each type and call describe() on each.

**STARTER CODE**

```
class Shape:
    def __init__(self, colour):
        pass

    def describe(self):
        pass

class Circle(Shape):
    def __init__(self, colour, radius):
        pass  # call super().__init__() here

class Rectangle(Shape):
    def __init__(self, colour, length, width):
        pass
```

## 2.3 - Polymorphism

Different objects responding to the same method call in their own way.

Polymorphism means 'many forms'. In OOP, it means that different classes can share the same method name, but each class provides its own implementation. A function that calls that method does not need to know which specific class it is dealing with - it just calls the method and the right version runs automatically. This makes code flexible and easy to extend.

**EXAMPLE**

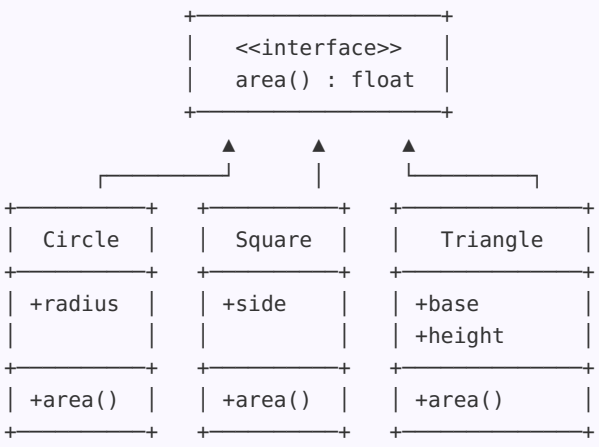
```
class Shape:
    def area(self):
        raise NotImplementedError

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

    def area(self):
        return 3.14159 * self.radius ** 2

class Square:
    def __init__(self, side):
        self.side = side

    def area(self):
        return self.side ** 2

shapes = [Circle(5), Square(4), Circle(3)]
for shape in shapes:
    print(f"{type(shape).__name__}: {shape.area():.2f}")
```

The loop calls .area() on each object without checking what type it is. Each object responds with its own correct calculation. Adding a new shape only requires writing a new class with an area() method - nothing else changes.

**OBJECT DIAGRAM**

```
SHARED METHOD NAME: area()
          |
  ┌───────┼──────────┐
  ↓       ↓          ↓
Circle   Square    Triangle
area() = area() =  area() =
π × r²   s × s     ½ × b × h

All objects respond to the SAME method call.
The caller does not need to know which class it is dealing with.

shapes = [Circle(5), Square(4), Triangle(6, 3)]
for shape in shapes:
    print(shape.area())   ← works on all three
```

### UML CLASS DIAGRAM




Each class provides its own implementation of area().

### ⚠ COMMON MISTAKES

- Giving methods different names across classes (e.g. get_area(), calculate_area(), find_area()). This breaks polymorphism — the shared name is what makes it work.
- Using isinstance() checks or type comparisons before calling methods. If you find yourself writing `if type(obj) == Dog`, polymorphism should handle it instead.
- Confusing polymorphism with overloading. Python does not support true method overloading (same name, different parameter counts). Use default parameters instead.

### Predict the Output

What will this program output? Write your answer before revealing it.

```python
class Printer:
    def output(self, text):
        print(f"[PLAIN] {text}")

class ColorPrinter(Printer):
    def output(self, text):
        print(f"[COLOR] {text}")

class PDFPrinter(Printer):
    def output(self, text):
        print(f"[PDF]   {text}")

queue = [Printer(), ColorPrinter(), PDFPrinter(), Printer()]
for p in queue:
    p.output("Report")
```

Output answer in Part 6 at the back

### Debug Task

The loop below crashes. Find the bug and fix it.

```
class Dog:
    def speak(self):
        return "Woof!"

class Cat:
    def talk(self):        # line 6
        return "Meow!"

class Bird:
    def speak(self):
        return "Tweet!"

animals = [Dog(), Cat(), Bird()]
for animal in animals:
    print(animal.speak())  # AttributeError on Cat
```

Bug and fix in Part 6 at the back

### Mini Challenge - Food Menu

Use polymorphism to build a simple menu system.

**Requirements:**

- Create three classes: Pizza, Burger, Salad.
- Each must have a get_price() method returning a float, and a describe() method that prints the item name and price.
- Write a function print_menu(items) that accepts a list and calls describe() on each item.
- Create a list with at least two of each item type and pass it to print_menu().

**STARTER CODE**

```
class Pizza:
    def get_price(self):
        return 9.99

    def describe(self):
        pass

class Burger:
    pass

class Salad:
    pass

def print_menu(items):
    pass
```

## 2.4 - Abstraction

Hiding complexity and exposing only what is necessary.

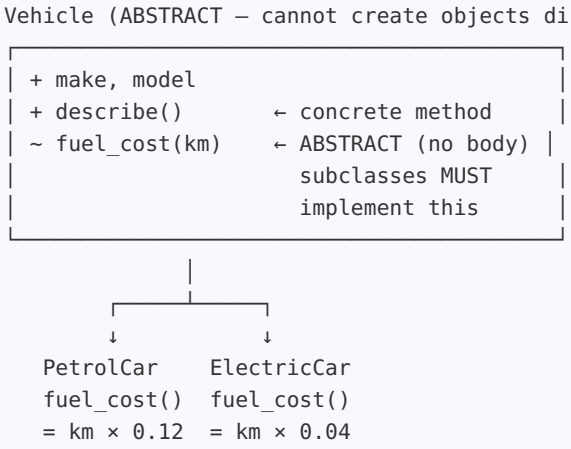
Abstraction means presenting a simplified view of an object to the outside world. Users of a class do not need to understand how it works internally - they only need to know what it does. In Python, abstract classes (using the abc module) can define methods that every subclass must implement, without providing the implementation themselves. This enforces a consistent interface across related classes.

**EXAMPLE**

```
from abc import ABC, abstractmethod

class Vehicle(ABC):
    def __init__(self, make, model):
        self.make  = make
        self.model = model

    @abstractmethod
    def fuel_cost(self, distance_km):
        pass   # subclasses must implement this

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

class PetrolCar(Vehicle):
    def fuel_cost(self, distance_km):
        return distance_km * 0.12   # 12p per km

class ElectricCar(Vehicle):
    def fuel_cost(self, distance_km):
        return distance_km * 0.04   # 4p per km

cars = [PetrolCar("Ford", "Focus"), ElectricCar("Tesla", "Model 3")]
for car in cars:
    car.describe()
    print(f"100km costs: £{car.fuel_cost(100):.2f}")
```

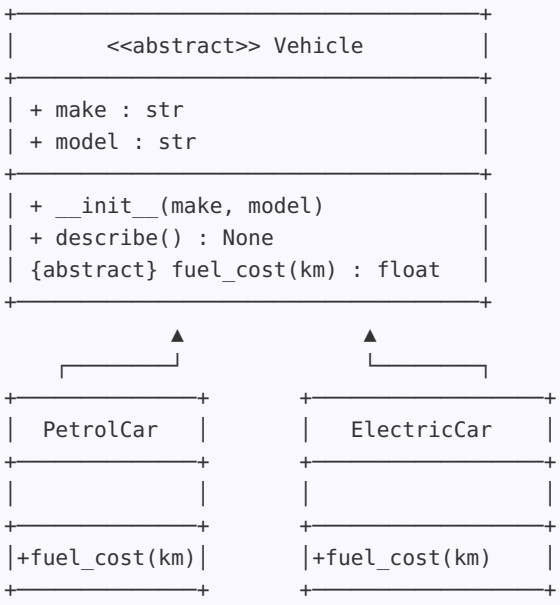
Vehicle cannot be instantiated directly - it is abstract. Any subclass that does not provide fuel_cost() will raise a TypeError when you try to create an object from it. This guarantees all Vehicle subclasses expose the same method.

### OBJECT DIAGRAM




~ = abstract. Attempting Vehicle("Ford","Focus") raises TypeError.

### UML CLASS DIAGRAM




{abstract} = method has no body in the parent class.

### ⚠ COMMON MISTAKES

- Forgetting to import ABC and abstractmethod: `from abc import ABC, abstractmethod`. Without this, Python treats the class as a normal class.
- Forgetting the @abstractmethod decorator. The method looks concrete, no error is raised, and subclasses can skip implementing it — a hidden bug.
- Implementing the wrong method name in the subclass (e.g. fuel_costs() instead of fuel_cost()). The abstract method is not satisfied, and TypeError is raised when creating the object.
- Trying to instantiate the abstract class directly: `v = Vehicle('Ford', 'Focus')` raises TypeError: Can't instantiate abstract class.

### Predict the Output

What will this program output? Write your answer before revealing it.

```python
from abc import ABC, abstractmethod

class Animal(ABC):
    def __init__(self, name):
        self.name = name

    @abstractmethod
    def sound(self):
        pass

    def introduce(self):
        print(f"I am {self.name} and I say {self.sound()}")

class Duck(Animal):
    def sound(self):
        return "quack"

d = Duck("Donald")
d.introduce()
```

Output answer in Part 6 at the back

### Debug Task

The Circle class below raises a TypeError when you try to create an object. Find and fix the bug.

```python
from abc import ABC, abstractmethod

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

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

    def calculate_area(self):     # line 12
        return 3.14159 * self.radius ** 2

c = Circle(5)     # TypeError raised here
print(c.area())
```

Bug and fix in Part 6 at the back

**Mini Challenge - Employee Pay Calculator**

Design an abstract Employee class and two concrete subclasses.

**Requirements:**

- Abstract class Employee: constructor takes name. Abstract method calculate_pay(). Concrete method display() prints '{name} earns £{calculate_pay():.2f} this week'.
- HourlyWorker(Employee): constructor takes name, hours, rate. calculate_pay() returns hours × rate.
- SalariedWorker(Employee): constructor takes name, annual_salary. calculate_pay() returns annual_salary / 52.
- Create one of each and call display() on both.

**STARTER CODE**

```
from abc import ABC, abstractmethod

class Employee(ABC):
    def __init__(self, name):
        pass

    @abstractmethod
    def calculate_pay(self):
        pass

    def display(self):
        pass

class HourlyWorker(Employee):
    pass

class SalariedWorker(Employee):
    pass
```

## 2.5 - Pillars Retrieval Quiz

**Quick Recall - All Four Pillars**

Cover the answers and test yourself. Revisit any pillar section you struggle with.

**Q1**
Which pillar prevents direct access to an object's internal data?

**Q2**
In Python, how do you make an attribute private?

**Q3**
What keyword is used in Python to call the parent class's constructor?

**Q4**
What is the difference between a parent class and a subclass?

**Q5**
What allows different objects to respond to the same method call differently?

**Q6**
What must you import to create an abstract class in Python?

**Q7**
What happens if a subclass does not implement all abstract methods?

**Q8**
What does UML stand for and why is it used?

Answers in Part 6 at the back of this booklet

# Part 3 - Concepts in Depth

10 key topics with code examples, key points, and common mistakes

## 3.1 - Classes and Objects

A class is a blueprint that describes what attributes (data) an object will have and what methods (behaviour) it can perform. An object is a specific instance created from that blueprint. You can create as many objects as you need from a single class, and each object has its own independent copy of the attributes defined in the class.

The class keyword defines the class. By convention, class names use PascalCase (each word capitalised). The object is created by calling the class as if it were a function.

### DEFINING A CLASS AND CREATING OBJECTS

```
class Player:
    pass   # empty class for now

p1 = Player()
p2 = Player()
print(type(p1))           # <class '__main__.Player'>
print(p1 is p2)           # False - two separate objects
p1.name  = "Alice"        # attributes added dynamically
p1.score = 0
print(p1.name, p1.score)  # Alice 0
```

### A CLASS WITH ATTRIBUTES AND A METHOD

```
class Playlist:
    def __init__(self, name):
        self.name   = name
        self.tracks = []

    def add_track(self, title):
        self.tracks.append(title)

    def play(self):
        for i, track in enumerate(self.tracks, 1):
            print(f"{i}. {track}")

my_playlist = Playlist("Morning Run")
my_playlist.add_track("Eye of the Tiger")
my_playlist.add_track("Lose Yourself")
my_playlist.play()
```

### KEY POINTS

- Each object is independent - changing one does not affect another.
- self refers to the specific object the method is being called on.
- The class definition is the blueprint; the object is the thing you work with.

### COMMON MISTAKES

- Forgetting self as the first parameter of every instance method.
- Confusing the class itself with an instance of the class.

## 3.2 - Constructors and __init__

__init__ is the constructor method. Python calls it automatically when you create a new object from a class. Its job is to set up the initial state of the object by assigning values to its attributes. The first parameter is always self, which refers to the object being created. Any other parameters become the arguments you pass when creating the object.

Without __init__, all objects start completely empty. With it, you guarantee every object starts with the data it needs.

### CONSTRUCTOR WITH REQUIRED PARAMETERS

```
class Student:
    def __init__(self, name, year, target_grade):
        self.name         = name
        self.year         = year
        self.target_grade = target_grade
        self.grades       = []   # every student starts with an empty grade list

    def add_grade(self, subject, mark):
        self.grades.append((subject, mark))

s = Student("James", 12, "A")
s.add_grade("CS", 78)
s.add_grade("Maths", 85)
print(s.name, s.grades)
```

### CONSTRUCTOR WITH DEFAULT PARAMETER VALUES

```
class Message:
    def __init__(self, sender, body, read=False, priority=1):
        self.sender   = sender
        self.body     = body
        self.read     = read
        self.priority = priority

m1 = Message("Alice", "See you at 5")
m2 = Message("Server", "Disk full", priority=5)
print(m1.read, m1.priority)   # False 1
print(m2.priority)             # 5
```

**KEY POINTS**

- __init__ is called once when the object is created, not every time a method runs.
- self.attribute = value stores the value on the specific object being created.
- Default parameter values let you create objects with or without every argument.

**COMMON MISTAKES**

- Writing __init with one underscore instead of two on each side.
- Not using self. before attribute names, creating a local variable instead of an attribute.

## 3.3 - Instance Attributes vs Class Attributes

Instance attributes are defined inside __init__ using self.name = value. They belong to one specific object and are independent between objects.

Class attributes are defined directly inside the class body, outside any method. They are shared by all objects of that class. Changing a class attribute affects every object unless an object has overridden it with its own instance attribute of the same name.

Class attributes are useful for values that are the same for all instances - such as a counter tracking how many objects have been created, or a constant configuration value.

### CLASS ATTRIBUTE AS A COUNTER

```
class Character:
    count = 0   # class attribute - shared by all instances

    def __init__(self, name, char_class):
        self.name       = name
        self.char_class = char_class
        self.hp         = 100
        Character.count += 1   # increment the shared counter

    def __str__(self):
        return f"{self.name} ({self.char_class})"

c1 = Character("Aldric", "Warrior")
c2 = Character("Lyra",   "Mage")
c3 = Character("Finn",   "Rogue")
print(Character.count)   # 3
```

### CLASS ATTRIBUTE AS A SHARED CONSTANT

```
class Spaceship:
    MAX_CREW = 6   # same for all spaceships of this type

    def __init__(self, name):
        self.name = name
        self.crew = []

    def add_crew(self, member):
        if len(self.crew) < Spaceship.MAX_CREW:
            self.crew.append(member)
        else:
            print(f"{self.name} is at full capacity ({Spaceship.MAX_CREW})")

s = Spaceship("Endeavour")
for name in ["Ali", "Beth", "Cam", "Dina", "Ed", "Fran", "Gus"]:
    s.add_crew(name)
print(len(s.crew))   # 6
```

**KEY POINTS**

- Instance attributes: one per object, defined in __init__ with self.
- Class attributes: one copy shared by all objects, defined in the class body.
- Access class attributes via ClassName.attribute to avoid confusion with instance attributes.

**COMMON MISTAKES**

- Modifying a mutable class attribute (like a list) via self - this changes the shared copy.
- Using self.COUNT when you mean ClassName.COUNT, which creates an instance attribute instead.

## 3.4 - Instance Methods, Class Methods, and Static Methods

Instance methods take self as the first parameter and can access or modify the object's attributes. They are the most common type.

Class methods take cls as the first parameter (decorated with @classmethod) and can access or modify class attributes. They are often used as alternative constructors.

Static methods have no self or cls parameter (decorated with @staticmethod). They are utility functions that logically belong to the class but do not need to access any instance or class data.

### ALL THREE METHOD TYPES ON ONE CLASS

```
class Temperature:
    scale = "Celsius"   # class attribute

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

    def describe(self):                         # instance method
        return f"{self.degrees} {Temperature.scale}"

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

    @staticmethod
    def celsius_to_fahrenheit(c):               # static method
        return c * 9 / 5 + 32

t = Temperature(20)
print(t.describe())                             # 20 Celsius
Temperature.set_scale("Kelvin")
print(t.describe())                             # 20 Kelvin
print(Temperature.celsius_to_fahrenheit(100)) # 212.0
```

### CLASS METHOD AS AN ALTERNATIVE CONSTRUCTOR

```
class Date:
    def __init__(self, day, month, year):
        self.day   = day
        self.month = month
        self.year  = year

    @classmethod
    def from_string(cls, date_str):
        day, month, year = date_str.split("/")
        return cls(int(day), int(month), int(year))

    def __str__(self):
        return f"{self.day:02}/{self.month:02}/{self.year}"

d1 = Date(15, 3, 2026)
d2 = Date.from_string("22/06/2025")
print(d1)   # 15/03/2026
print(d2)   # 22/06/2025
```

**KEY POINTS**

- Use instance methods when you need to access self.

- Use class methods when you need to access or change class-level data.
- Use static methods for helper functions that relate to the class but need no object data.

**COMMON MISTAKES**

- Forgetting the @classmethod or @staticmethod decorator.
- Using self in a static method (there is no self).

## 3.5 - Encapsulation - Private Attributes

In Python, prefixing an attribute name with double underscores (__name) makes it private through a mechanism called name mangling. Python renames it internally to _ClassName__name, so accidental access from outside the class is blocked.

A single underscore (_name) is a weaker convention meaning 'intended for internal use' - it is not enforced by Python, just a signal to other programmers.

Private attributes should be set and read through methods. This allows the class to validate new values before accepting them, protecting the object from invalid states.

### PRIVATE ATTRIBUTE WITH VALIDATION

```
class Speedometer:
    def __init__(self):
        self.__speed = 0   # private

    def accelerate(self, amount):
        if amount > 0:
            self.__speed = min(self.__speed + amount, 200)  # max 200 km/h

    def brake(self, amount):
        self.__speed = max(self.__speed - amount, 0)

    def get_speed(self):
        return self.__speed

s = Speedometer()
s.accelerate(120)
s.accelerate(100)   # capped at 200
print(s.get_speed())   # 200
s.brake(50)
print(s.get_speed())   # 150
```

### DEMONSTRATING NAME MANGLING

```
class Config:
    def __init__(self, api_key):
        self.__api_key = api_key

    def get_masked_key(self):
        return "*" * (len(self.__api_key) - 4) + self.__api_key[-4:]

c = Config("sk-abc123xyz789")
print(c.get_masked_key())   # **********x789
# print(c.__api_key)        # AttributeError
print(c._Config__api_key)   # sk-abc123xyz789 - works but bad practice
```

**KEY POINTS**

- __ (double underscore) applies name mangling - the attribute is not truly hidden.
- _ (single underscore) is a convention only - Python does not enforce it.
- Validation logic in setter methods prevents invalid data reaching private attributes.

**COMMON MISTAKES**

- Thinking __ makes attributes completely inaccessible - they can still be reached via _ClassName__attr.
- Accessing __ attributes from a subclass - name mangling applies per class, which breaks inheritance.

## 3.6 - Properties with @property

Python's @property decorator lets you define a method that is accessed like an attribute, with no parentheses. This gives you the clean syntax of direct attribute access while running validation or transformation logic behind the scenes.

@property defines the getter. @attribute_name.setter defines the setter - this is where you validate new values. @attribute_name.deleter defines behaviour when del is called on it.

Properties are the Pythonic alternative to writing explicit get_x() and set_x() methods.

### PROPERTY WITH GETTER AND SETTER

```
class Circle:
    def __init__(self, radius):
        self.radius = radius   # calls the setter

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

    @radius.setter
    def radius(self, value):
        if value < 0:
            raise ValueError("Radius cannot be negative")
        self.__radius = value

    @property
    def area(self):
        return 3.14159 * self.__radius ** 2

c = Circle(7)
print(c.radius)        # 7
print(c.area)          # 153.93...
c.radius = 10
print(c.area)          # 314.15...
# c.radius = -1        # ValueError
```

### READ-ONLY PROPERTY (GETTER ONLY)

```
class UserProfile:
    def __init__(self, username, email):
        self.__username = username.lower()
        self.__email    = email

    @property
    def username(self):
        return self.__username

    @property
    def display_name(self):
        return f"@{self.__username}"

u = UserProfile("Alice_B", "alice@example.com")
print(u.username)      # alice_b
print(u.display_name)  # @alice_b
# u.username = "bob"   # AttributeError - no setter defined
```

**KEY POINTS**

- @property makes a method callable as an attribute - no brackets needed.
- Use @name.setter to add write access with optional validation.
- Omitting the setter creates a read-only property.

**COMMON MISTAKES**

- Calling a property with brackets: c.area() - this raises a TypeError.
- Storing the value in self.radius inside the setter, which calls the setter again - infinite recursion. Use self.__radius.

## 3.7 - Inheritance and super()

Inheritance lets a subclass receive all the attributes and methods of a parent class automatically. The subclass can then add new attributes and methods, or override inherited ones.

super() gives access to the parent class from inside the subclass. In __init__, it is almost always used to call the parent constructor so the parent's attributes are set up correctly before the subclass adds its own.

Python supports multiple inheritance (a class inheriting from more than one parent), but this is an advanced topic and can be complex to manage.

### SINGLE INHERITANCE WITH SUPER()

```
class Enemy:
    def __init__(self, name, hp, damage):
        self.name   = name
        self.hp     = hp
        self.damage = damage

    def attack(self):
        return self.damage

    def is_alive(self):
        return self.hp > 0

class Boss(Enemy):
    def __init__(self, name, hp, damage, special_attack):
        super().__init__(name, hp, damage)   # call Enemy.__init__
        self.special_attack = special_attack
        self.enraged        = False

    def enrage(self):
        self.enraged = True
        self.damage  = int(self.damage * 1.5)
        print(f"{self.name} becomes enraged!")

    def attack(self):
        if self.enraged:
            return self.damage + self.special_attack
        return self.damage

dragon = Boss("Ancient Dragon", 5000, 80, 200)
dragon.enrage()
print(dragon.attack())   # 320
```

### EXTENDING INHERITED BEHAVIOUR WITH SUPER()

```
class Post:
    def __init__(self, author, text):
        self.author = author
        self.text   = text
        self.likes  = 0

    def like(self):
        self.likes += 1

    def display(self):
        print(f"[{self.author}] {self.text}  ({self.likes} likes)")

class SponsoredPost(Post):
    def __init__(self, author, text, sponsor):
        super().__init__(author, text)
        self.sponsor = sponsor

    def display(self):
        super().display()              # call parent version first
        print(f"  Sponsored by {self.sponsor}")

p = SponsoredPost("CodeBash", "Free OOP booklet!", "CodeBash Ltd")
p.like()
p.like()
p.display()
```

**KEY POINTS**

- super().__init__() must be called to initialise the parent class's attributes.
- A subclass can override any method from its parent.
- Calling super().method_name() from an override lets you run the parent version first.

**COMMON MISTAKES**

- Forgetting to call super().__init__() - the parent's attributes are never created.
- Calling Parent.__init__(self, ...) directly instead of super() - works but is less maintainable.

## 3.8 - Polymorphism and Method Overriding

Polymorphism ('many forms') allows objects of different classes to be used interchangeably when they share a common interface - typically a method with the same name.

Method overriding is when a subclass provides its own version of a method defined in the parent class. When that method is called on a subclass object, Python runs the subclass version, not the parent's.

This makes it possible to write code that works with a base class, and then later extend the program with new subclasses without modifying the existing code.

#### POLYMORPHISM WITH A SHARED INTERFACE

```
class Notification:
    def __init__(self, recipient, message):
        self.recipient = recipient
        self.message   = message

    def send(self):
        raise NotImplementedError("Subclasses must implement send()")

class EmailNotification(Notification):
    def send(self):
        print(f"Email to {self.recipient}: {self.message}")

class SMSNotification(Notification):
    def send(self):
        print(f"SMS to {self.recipient}: {self.message[:160]}")

class PushNotification(Notification):
    def send(self):
        print(f"Push to {self.recipient}: {self.message[:50]}")

def broadcast(notifications):
    for n in notifications:
        n.send()   # same call, different behaviour per type

queue = [
    EmailNotification("alice@example.com", "Your order has shipped"),
    SMSNotification("07700900123", "Your code is 4821"),
    PushNotification("user_id_42", "New message from Bob"),
]
broadcast(queue)
```

### ISINSTANCE() CHECK WITH POLYMORPHIC OBJECTS

```
class Shape:
    def area(self):
        raise NotImplementedError

class Triangle(Shape):
    def __init__(self, base, height):
        self.base   = base
        self.height = height

    def area(self):
        return 0.5 * self.base * self.height

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

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

shapes = [Triangle(6, 4), Rectangle(5, 3), Triangle(10, 2)]
total_area = sum(s.area() for s in shapes)
print(f"Total area: {total_area}")   # 31.0

for s in shapes:
    if isinstance(s, Triangle):
        print(f"Triangle with base {s.base}")
```

**KEY POINTS**

- Overriding replaces the parent method for that subclass only.
- isinstance(obj, ClassName) returns True if obj is an instance of ClassName or a subclass of it.
- Python resolves which method to call at runtime based on the object's actual type.

**COMMON MISTAKES**

- Misspelling the method name in the subclass - the parent version runs silently instead.
- Assuming isinstance(obj, Parent) returns False for subclass instances - it returns True.

## 3.9 - Abstract Base Classes

An abstract class is a class that cannot be instantiated directly. Its purpose is to define a common interface that all subclasses must follow. You mark a class as abstract by inheriting from ABC (from the abc module). Any method decorated with @abstractmethod must be overridden in every concrete subclass.

Abstract classes sit between a pure interface (all methods abstract) and a regular class (no abstract methods). They let you provide some shared implementation while still enforcing that subclasses supply the missing pieces.

#### ABSTRACT CLASS ENFORCING A REQUIRED INTERFACE

```
from abc import ABC, abstractmethod

class PaymentMethod(ABC):
    def __init__(self, owner):
        self.owner = owner

    @abstractmethod
    def process(self, amount):
        pass

    @abstractmethod
    def get_details(self):
        pass

    def receipt(self, amount):
        print(f"Payment of £{amount:.2f} processed for {self.owner}")
        self.get_details()

class CreditCard(PaymentMethod):
    def __init__(self, owner, last_four):
        super().__init__(owner)
        self.last_four = last_four

    def process(self, amount):
        print(f"Charging £{amount:.2f} to card ending {self.last_four}")

    def get_details(self):
        print(f"Card: **** **** **** {self.last_four}")

class PayPal(PaymentMethod):
    def __init__(self, owner, email):
        super().__init__(owner)
        self.email = email

    def process(self, amount):
        print(f"PayPal transfer of £{amount:.2f} from {self.email}")

    def get_details(self):
        print(f"PayPal: {self.email}")

# p = PaymentMethod("Alice")  # TypeError - cannot instantiate abstract class
card = CreditCard("Alice", "4821")
card.process(49.99)
card.receipt(49.99)
```

### ABSTRACT CLASS WITH PARTIAL IMPLEMENTATION

```
from abc import ABC, abstractmethod

class Report(ABC):
    def __init__(self, title):
        self.title = title

    @abstractmethod
    def generate(self):
        pass

    def print_header(self):
        print("=" * 40)
        print(f"  {self.title}")
        print("=" * 40)

class SalesReport(Report):
    def __init__(self, title, revenue, costs):
        super().__init__(title)
        self.revenue = revenue
        self.costs   = costs

    def generate(self):
        self.print_header()
        print(f"Revenue: £{self.revenue:,.2f}")
        print(f"Costs:   £{self.costs:,.2f}")
        print(f"Profit:  £{self.revenue - self.costs:,.2f}")

r = SalesReport("Q1 Sales", 125000, 87000)
r.generate()
```

**KEY POINTS**

- Inheriting from ABC and using @abstractmethod prevents direct instantiation.
- Any subclass that does not implement all abstract methods is also abstract.
- Abstract classes can contain regular methods alongside abstract ones.

**COMMON MISTAKES**

- Forgetting to import ABC and abstractmethod from the abc module.
- Spelling @abstractMethod with a capital M - Python will not recognise it as a decorator.

## 3.10 - Magic Methods (Dunder Methods)

Magic methods (also called dunder methods, from 'double underscore') let you define how your objects behave with Python's built-in operations and functions. They are always named with double underscores on each side: __name__.

__str__ defines what print() and str() output for your object. __repr__ defines a developer-facing representation (used in the REPL and for debugging). __len__ allows len() to work on your object. __add__, __sub__, __eq__ etc. define how arithmetic and comparison operators behave.

### __STR__, __REPR__, __LEN__, AND __EQ__

```
class Hand:
    """A hand of playing cards."""
    def __init__(self):
        self.cards = []

    def add_card(self, card):
        self.cards.append(card)

    def __len__(self):
        return len(self.cards)

    def __str__(self):
        return "Hand: " + ", ".join(self.cards)

    def __repr__(self):
        return f"Hand({self.cards!r})"

    def __eq__(self, other):
        return len(self) == len(other)

h1 = Hand()
h2 = Hand()
h1.add_card("Ace of Spades")
h1.add_card("King of Hearts")
h2.add_card("7 of Clubs")
h2.add_card("3 of Diamonds")

print(h1)           # Hand: Ace of Spades, King of Hearts
print(len(h1))      # 2
print(h1 == h2)     # True - same number of cards
```

### ARITHMETIC OPERATORS WITH __ADD__ AND __MUL__

```
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 __mul__(self, scalar):
        return Vector(self.x * scalar, self.y * scalar)

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

    def magnitude(self):
        return (self.x ** 2 + self.y ** 2) ** 0.5

v1 = Vector(3, 4)
v2 = Vector(1, 2)
v3 = v1 + v2
v4 = v1 * 3
print(v3)              # Vector(4, 6)
print(v4)              # Vector(9, 12)
print(v1.magnitude())  # 5.0
```

**KEY POINTS**

- __str__ is for end users; __repr__ is for developers and should ideally be unambiguous.
- Define __eq__ if your objects need to be compared with ==.
- Magic methods integrate your class with Python's built-in syntax and functions.

**COMMON MISTAKES**

- Returning a value from __str__ using print() inside it - it should return a string, not print one.
- Forgetting that __add__ must return a new object, not modify self.

# Part 4 - A-Level Exam Questions

12 questions ranging from 2 to 8 marks, with full mark schemes

### Question 1

[2 marks]

State what is meant by the term 'class' in object-oriented programming. [2 marks]

Mark scheme in Part 6 at the back

### Question 2

[2 marks]

Explain the difference between a class and an object. [2 marks]

Mark scheme in Part 6 at the back

### Question 3

[4 marks]

Explain what is meant by encapsulation in OOP. Give one advantage of using encapsulation in a large program. [4 marks]

Mark scheme in Part 6 at the back

### Question 4

[4 marks]

Explain what is meant by inheritance in OOP and describe one benefit of using it. [4 marks]

Mark scheme in Part 6 at the back

### Question 5

[4 marks]

Explain what is meant by polymorphism. Give a programming example to support your answer. [4 marks]

Mark scheme in Part 6 at the back

### Question 6 [2 marks]

State two differences between an abstract class and a concrete class. [2 marks]

Mark scheme in Part 6 at the back

### Question 7 [6 marks]

A school library system stores information about books. Each book has a title, author, ISBN, and a flag to indicate whether it is currently on loan.

Write a Python class called Book that:
- stores title, author, and ISBN as public attributes
- stores the on_loan status as a private attribute, initially False
- has a method check_out() that sets on_loan to True, or prints an error if already on loan
- has a method return_book() that sets on_loan to False
- has a __str__ method that returns a string in the format: Title - Author (ISBN: xxx)
[6 marks]

Mark scheme in Part 6 at the back

**Question 8** [6 marks]

The following class is defined:

```
class Vehicle:
    def __init__(self, make, model, year):
        self.make  = make
        self.model = model
        self.year  = year

    def describe(self):
        return f'{self.year} {self.make} {self.model}'
```

Write a subclass ElectricVehicle that:
- inherits from Vehicle
- adds a battery_kWh attribute
- overrides describe() to append '(Electric)' to the result from the parent method
- has a range_km property that returns battery_kWh multiplied by 6

[6 marks]

Mark scheme in Part 6 at the back

### Question 9 [4 marks]

Study the following Python code:

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

    def speak(self):
        return 'Some sound'

class Dog(Animal):
    def speak(self):
        return 'Woof'

class Cat(Animal):
    def speak(self):
        return 'Meow'

animals = [Dog('Rex'), Cat('Luna'), Animal('Bob')]
for a in animals:
    print(a.name + ': ' + a.speak())
```

Write the exact output produced by this code. [4 marks]

Mark scheme in Part 6 at the back

### Question 10 [4 marks]

Explain what the __str__ method does in Python and why it is useful when working with objects. [4 marks]

Mark scheme in Part 6 at the back

### Question 11 [2 marks]

State what is meant by 'method overriding'. [2 marks]

Mark scheme in Part 6 at the back

**Question 12** [8 marks]

A music streaming application manages playlists. Design and implement Python classes to model this system.

Your solution must include:
- A Song class storing title, artist, and duration in seconds
- A Playlist class storing a name and a list of Song objects
- A method add_song(song) on Playlist
- A method total_duration() on Playlist that returns the total duration as mm:ss
- A __str__ method on both classes
- A Premium Playlist subclass that also stores a max_songs limit and prevents adding songs beyond that limit

[8 marks]

Mark scheme in Part 6 at the back

# Part 5.1 - RPG Combat System

Capstone program walkthrough

A text-based RPG combat simulator demonstrating OOP design in a realistic context. The program models characters, weapons, and turn-based combat. It covers: abstract base classes, inheritance hierarchies, encapsulation, property validation, polymorphism, and magic methods.

**CONCEPTS COVERED**

- Abstract base class (Character)
- Inheritance (Warrior, Mage, Archer)
- Encapsulation (__hp, __max_hp)
- Properties for read-only access
- Method overriding (attack())
- Polymorphism in the combat loop
- __str__ and __repr__

## Step 1 Define the abstract Character base class

All characters share a name, hit points (HP), and the ability to attack. We use an abstract base class to enforce that every character type provides its own attack() method. HP is private to prevent external code from setting it to an invalid value. A property gives read-only access.

**CODE - STEP 1**

```
from abc import ABC, abstractmethod
import random

class Character(ABC):
    def __init__(self, name, hp, defence):
        self.__name    = name
        self.__hp      = hp
        self.__max_hp  = hp
        self.defence   = defence

    @property
    def name(self):
        return self.__name

    @property
    def hp(self):
        return self.__hp

    @property
    def max_hp(self):
        return self.__max_hp

    def take_damage(self, damage):
        actual = max(0, damage - self.defence)
        self.__hp = max(0, self.__hp - actual)
        return actual

    def is_alive(self):
        return self.__hp > 0

    def heal(self, amount):
        self.__hp = min(self.__max_hp, self.__hp + amount)

    @abstractmethod
    def attack(self, target):
        pass

    def __str__(self):
        return f"{self.__name} (HP: {self.__hp}/{self.__max_hp})"

    def __repr__(self):
        return f"{type(self).__name__}('{self.__name}', hp={self.__hp})"
```

## Step 2 Create the Warrior, Mage, and Archer subclasses

Each subclass adds its own attributes and implements attack() in its own way. The Warrior deals reliable damage. The Mage uses spells with mana cost. The Archer has a critical hit chance. All call super().__init__() to set up the parent class first.

**CODE - STEP 2**

```
class Warrior(Character):
    def __init__(self, name):
        super().__init__(name, hp=120, defence=8)
        self.strength = 20

    def attack(self, target):
        damage = self.strength + random.randint(0, 10)
        actual = target.take_damage(damage)
        print(f"{self.name} strikes {target.name} for {actual} damage.")
        return actual


class Mage(Character):
    def __init__(self, name):
        super().__init__(name, hp=80, defence=2)
        self.spell_power = 30
        self.__mana      = 100

    @property
    def mana(self):
        return self.__mana

    def attack(self, target):
        if self.__mana < 20:
            print(f"{self.name} is out of mana and throws a weak punch.")
            actual = target.take_damage(5)
        else:
            self.__mana -= 20
            damage = self.spell_power + random.randint(-5, 15)
            actual = target.take_damage(damage)
            print(f"{self.name} casts Fireball on {target.name} for {actual} damage. "
                  f"(Mana: {self.__mana})")
        return actual


class Archer(Character):
    def __init__(self, name):
        super().__init__(name, hp=95, defence=4)
        self.base_damage   = 18
        self.crit_chance   = 0.25

    def attack(self, target):
        crit   = random.random() < self.crit_chance
        damage = self.base_damage * (2 if crit else 1) + random.randint(0, 8)
        actual = target.take_damage(damage)
        label  = "CRITICAL HIT" if crit else "fires an arrow"
        print(f"{self.name} {label}! {target.name} takes {actual} damage.")
        return actual
```

## Step 3 Build the combat loop

The combat function takes two Character objects. Because both are Character instances, we can call .attack() on either without knowing their specific type. This is polymorphism in practice - the correct subclass version runs automatically.

**CODE - STEP 3**

```
def combat(fighter_a, fighter_b):
    print(f"\n{'='*40}")
    print(f"  {fighter_a.name}  vs  {fighter_b.name}")
    print(f"{'='*40}")
    turn = 0
    while fighter_a.is_alive() and fighter_b.is_alive():
        turn += 1
        print(f"\n-- Round {turn} --")
        fighter_a.attack(fighter_b)
        if fighter_b.is_alive():
            fighter_b.attack(fighter_a)
        print(f"  {fighter_a}")
        print(f"  {fighter_b}")

    winner = fighter_a if fighter_a.is_alive() else fighter_b
    print(f"\n{winner.name} wins after {turn} rounds!")
    return winner
```

## Step 4 Add a Boss class using inheritance

A Boss is a more powerful Warrior that becomes enraged when its HP drops below 30%. This extends the Warrior class, calling super().__init__() to set up the warrior's attributes, then adding boss-specific ones. It overrides attack() to add the enrage mechanic.

**CODE - STEP 4**

```
class Boss(Warrior):
    def __init__(self, name, hp_multiplier=2):
        super().__init__(name)
        # manually inflate HP to simulate a tougher enemy
        self._Character__hp     = int(self.max_hp * hp_multiplier)
        self._Character__max_hp = int(self.max_hp * hp_multiplier)
        self.strength = 30
        self.__enraged = False

    def attack(self, target):
        if self.hp / self.max_hp < 0.3 and not self.__enraged:
            self.__enraged = True
            self.strength  = int(self.strength * 1.8)
            print(f"*** {self.name} ENRAGES! Strength surges to {self.strength}! ***")
        return super().attack(target)

    def __str__(self):
        status = " [ENRAGED]" if self.__enraged else ""
        return super().__str__() + status
```

## Step 5 Run the full program

Everything comes together here. We create characters from different subclasses and run them through combat. The combat() function works with any Character subclass without modification - that is the power of polymorphism and abstraction.

**CODE - STEP 5**

```
if __name__ == "__main__":
    hero    = Warrior("Aldric")
    mage    = Mage("Lyra")
    archer  = Archer("Finn")
    boss    = Boss("Dragon Lord")

    print("\n*** QUALIFIER ROUNDS ***")
    combat(hero, mage)
    combat(archer, Warrior("Grunt"))

    print("\n*** BOSS ENCOUNTER ***")
    combat(hero, boss)
```

**COMPLETE PROGRAM**

```python
from abc import ABC, abstractmethod
import random

class Character(ABC):
    def __init__(self, name, hp, defence):
        self.__name   = name
        self.__hp     = hp
        self.__max_hp = hp
        self.defence  = defence

    @property
    def name(self):
        return self.__name

    @property
    def hp(self):
        return self.__hp

    @property
    def max_hp(self):
        return self.__max_hp

    def take_damage(self, damage):
        actual = max(0, damage - self.defence)
        self.__hp = max(0, self.__hp - actual)
        return actual

    def is_alive(self):
        return self.__hp > 0

    def heal(self, amount):
        self.__hp = min(self.__max_hp, self.__hp + amount)

    @abstractmethod
    def attack(self, target):
        pass

    def __str__(self):
        return f"{self.__name} (HP: {self.__hp}/{self.__max_hp})"

    def __repr__(self):
        return f"{type(self).__name__}('{self.__name}', hp={self.__hp})"


class Warrior(Character):
    def __init__(self, name):
        super().__init__(name, hp=120, defence=8)
        self.strength = 20

    def attack(self, target):
        damage = self.strength + random.randint(0, 10)
        actual = target.take_damage(damage)
        print(f"{self.name} strikes {target.name} for {actual} damage.")
        return actual


class Mage(Character):
    def __init__(self, name):
        super().__init__(name, hp=80, defence=2)
        self.spell_power = 30
        self.__mana      = 100

    @property
    def mana(self):
        return self.__mana

    def attack(self, target):
```

```
        if self.__mana < 20:
            print(f"{self.name} is out of mana and throws a weak punch.")
            return target.take_damage(5)
        self.__mana -= 20
        damage = self.spell_power + random.randint(-5, 15)
        actual = target.take_damage(damage)
        print(f"{self.name} casts Fireball on {target.name} for {actual} damage.")
        return actual


class Archer(Character):
    def __init__(self, name):
        super().__init__(name, hp=95, defence=4)
        self.base_damage = 18
        self.crit_chance = 0.25

    def attack(self, target):
        crit   = random.random() < self.crit_chance
        damage = self.base_damage * (2 if crit else 1) + random.randint(0, 8)
        actual = target.take_damage(damage)
        label  = "CRITICAL HIT" if crit else "fires an arrow"
        print(f"{self.name} {label}! {target.name} takes {actual} damage.")
        return actual


class Boss(Warrior):
    def __init__(self, name):
        super().__init__(name)
        self._Character__hp     = 240
        self._Character__max_hp = 240
        self.strength  = 30
        self.__enraged = False

    def attack(self, target):
        if self.hp / self.max_hp < 0.3 and not self.__enraged:
            self.__enraged = True
            self.strength  = int(self.strength * 1.8)
            print(f"*** {self.name} ENRAGES! ***")
        return super().attack(target)


def combat(a, b):
    print(f"\n{a.name} vs {b.name}")
    turn = 0
    while a.is_alive() and b.is_alive():
        turn += 1
        a.attack(b)
        if b.is_alive():
            b.attack(a)
    winner = a if a.is_alive() else b
    print(f"{winner.name} wins after {turn} rounds!")
    return winner


if __name__ == "__main__":
    combat(Warrior("Aldric"), Mage("Lyra"))
    combat(Archer("Finn"), Boss("Dragon Lord"))
```

### Extension Ideas

- Add a Healer subclass that can choose between attacking and healing its lowest-HP ally.
- Add equipment (Weapon, Armour) as separate classes composed onto characters.
- Build a party system where multiple characters on each side take turns.

- Add experience points and a level_up() method that increases attributes.
- Store battle history and implement a log_to_file() method using file I/O.

# Part 5.2 - Social Media Simulator

Capstone program walkthrough

A simulation of a basic social media platform. This program models users, posts, comments, and likes using composition (objects containing other objects) rather than just inheritance. It demonstrates how real-world systems are built by composing small, well-defined classes together.

**CONCEPTS COVERED**

- Composition (Post contains a list of Comment objects)
- Class attributes (total post counter)
- Properties with validation
- __str__ and __repr__
- Static methods for formatting
- isinstance() type checking
- Separation of responsibilities between classes

## Step 1 Define the User class

A User has a username, bio, and a list of followers. The username is private and read-only after creation. The follower list is managed through methods rather than accessed directly. A class attribute counts the total number of registered users.

**CODE - STEP 1**

```
class User:
    total_users = 0

    def __init__(self, username, bio=""):
        self.__username = username.lower().strip()
        self.bio        = bio
        self.__followers = []
        User.total_users += 1

    @property
    def username(self):
        return self.__username

    def follow(self, other_user):
        if not isinstance(other_user, User):
            raise TypeError("Can only follow User objects")
        if other_user not in self.__followers:
            self.__followers.append(other_user)
            print(f"@{self.__username} now follows @{other_user.username}")

    def follower_count(self):
        return len(self.__followers)

    @staticmethod
    def validate_username(username):
        return username.replace("_", "").isalnum() and len(username) >= 3

    def __str__(self):
        return f"@{self.__username} ({self.follower_count()} following)"

    def __repr__(self):
        return f"User('{self.__username}')"
```

Step 2

## Define the Post and Comment classes

A Post belongs to a User and contains a body of text plus a list of comments and a list of users who have liked it. A Comment belongs to a User and a Post. This is composition - Post is composed of Comment objects. A class attribute on Post tracks how many posts have been created in total.

**CODE - STEP 2**

```
from datetime import datetime

class Comment:
    def __init__(self, author, text):
        if not isinstance(author, User):
            raise TypeError("Author must be a User")
        self.author    = author
        self.text      = text
        self.timestamp = datetime.now()

    def __str__(self):
        time_str = self.timestamp.strftime("%H:%M")
        return f"  [{time_str}] @{self.author.username}: {self.text}"

    def __repr__(self):
        return f"Comment('{self.author.username}', '{self.text[:20]}...')"


class Post:
    post_count = 0

    def __init__(self, author, body):
        if not isinstance(author, User):
            raise TypeError("Author must be a User")
        self.author    = author
        self.body      = body
        self.timestamp = datetime.now()
        self.__comments = []
        self.__likes    = []
        Post.post_count += 1
        self.post_id    = Post.post_count

    def like(self, user):
        if user not in self.__likes:
            self.__likes.append(user)

    def unlike(self, user):
        if user in self.__likes:
            self.__likes.remove(user)

    def add_comment(self, comment):
        if not isinstance(comment, Comment):
            raise TypeError("Must be a Comment object")
        self.__comments.append(comment)

    def get_likes(self):
        return len(self.__likes)

    def get_comments(self):
        return list(self.__comments)

    def __str__(self):
        date_str = self.timestamp.strftime("%d/%m/%Y %H:%M")
        output   = f"Post #{self.post_id} by @{self.author.username} [{date_str}]\n"
        output  += f"{self.body}\n"
        output  += f"Likes: {self.get_likes()} | Comments: {len(self.__comments)}"
        return output
```

## Step 3 Add a Feed class to aggregate posts

A Feed holds a collection of posts and provides methods to display, filter, and sort them. This class is responsible only for managing and displaying posts - it does not handle users or comments directly. Keeping each class focused on one responsibility makes the system easier to extend.

**CODE - STEP 3**

```
class Feed:
    def __init__(self, name):
        self.name  = name
        self.__posts = []

    def add_post(self, post):
        if not isinstance(post, Post):
            raise TypeError("Only Post objects can be added")
        self.__posts.append(post)

    def top_posts(self, n=3):
        return sorted(self.__posts, key=lambda p: p.get_likes(), reverse=True)[:n]

    def posts_by(self, user):
        return [p for p in self.__posts if p.author.username == user.username]

    def display(self):
        print(f"\n=== {self.name} Feed ===")
        if not self.__posts:
            print("  (no posts)")
        for post in reversed(self.__posts):   # newest first
            print(post)
            for c in post.get_comments():
                print(c)
            print()
```

## Step 4 Run a simulation

Now we create users, posts, comments, and likes to test the full system. Notice that the Feed.display() method works without knowing anything about how Post or Comment work internally - it just calls their __str__ methods.

**CODE - STEP 4**

```
if __name__ == "__main__":
    alice = User("Alice_B", "CS teacher and coder")
    bob   = User("Bob_CS",  "Year 12 student")
    carol = User("Carol_T", "Head of Science")

    alice.follow(bob)
    bob.follow(alice)

    p1 = Post(alice, "Just released a free Python OOP booklet - link in bio!")
    p2 = Post(bob,   "Finished my A-Level CS project. Used OOP throughout.")
    p3 = Post(carol, "Great CPD session today on teaching with technology.")

    p1.like(bob)
    p1.like(carol)
    p2.like(alice)

    p1.add_comment(Comment(bob,   "This is really useful, thanks!"))
    p1.add_comment(Comment(carol, "Sharing with my department."))
    p2.add_comment(Comment(alice, "Well done - that is a tough project!"))

    feed = Feed("Global")
    feed.add_post(p1)
    feed.add_post(p2)
    feed.add_post(p3)
    feed.display()

    print("Top post:", feed.top_posts(1)[0].body)
    print(f"Total users: {User.total_users}")
    print(f"Total posts: {Post.post_count}")
```

**COMPLETE PROGRAM**

```
from datetime import datetime


class User:
    total_users = 0

    def __init__(self, username, bio=""):
        self.__username  = username.lower().strip()
        self.bio         = bio
        self.__followers = []
        User.total_users += 1

    @property
    def username(self):
        return self.__username

    def follow(self, other_user):
        if other_user not in self.__followers:
            self.__followers.append(other_user)

    def follower_count(self):
        return len(self.__followers)

    def __str__(self):
        return f"@{self.__username} ({self.follower_count()} following)"


class Comment:
    def __init__(self, author, text):
        self.author    = author
        self.text      = text
        self.timestamp = datetime.now()

    def __str__(self):
        return f"  [{self.timestamp.strftime('%H:%M')}] @{self.author.username}:
{self.text}"


class Post:
    post_count = 0

    def __init__(self, author, body):
        self.author    = author
        self.body      = body
        self.timestamp = datetime.now()
        self.__comments = []
        self.__likes    = []
        Post.post_count += 1
        self.post_id    = Post.post_count

    def like(self, user):
        if user not in self.__likes:
            self.__likes.append(user)

    def add_comment(self, comment):
        self.__comments.append(comment)

    def get_likes(self):
        return len(self.__likes)

    def get_comments(self):
        return list(self.__comments)

    def __str__(self):
        date_str = self.timestamp.strftime("%d/%m/%Y %H:%M")
        return (f"Post #{self.post_id} by @{self.author.username} [{date_str}]\n"
                f"{self.body}\n"
```

```
                f"Likes: {self.get_likes()} | Comments: {len(self.get_comments())}")


class Feed:
    def __init__(self, name):
        self.name    = name
        self.__posts = []

    def add_post(self, post):
        self.__posts.append(post)

    def top_posts(self, n=3):
        return sorted(self.__posts, key=lambda p: p.get_likes(), reverse=True)[:n]

    def display(self):
        print(f"\n=== {self.name} Feed ===")
        for post in reversed(self.__posts):
            print(post)
            for c in post.get_comments():
                print(c)
            print()


if __name__ == "__main__":
    alice = User("Alice_B", "CS teacher")
    bob   = User("Bob_CS",  "Student")
    carol = User("Carol_T", "Head of Science")

    p1 = Post(alice, "Free Python OOP booklet - link in bio!")
    p2 = Post(bob,   "Finished my A-Level project. Used OOP throughout.")

    p1.like(bob)
    p1.like(carol)
    p2.like(alice)

    p1.add_comment(Comment(bob,   "Really useful, thanks!"))
    p1.add_comment(Comment(carol, "Sharing with my department."))

    feed = Feed("Global")
    feed.add_post(p1)
    feed.add_post(p2)
    feed.display()
```

### Extension Ideas

- Add a Story class (posts that expire after 24 hours) as a subclass of Post.
- Add a DirectMessage class linking two users with a list of Message objects.
- Add a Hashtag class and let posts reference multiple Hashtag objects.
- Implement search(keyword) on Feed that returns all posts containing a keyword.
- Serialize the feed to a JSON file and reload it on next run.

# Part 5.3 - Online Shop System

Capstone program walkthrough

An online shop with a product hierarchy, shopping cart, and order processing. This program uses abstract classes with concrete subclasses, properties for data validation, composition (Cart contains Product objects), and class methods for creating objects from data.

**CONCEPTS COVERED**

- Abstract class with abstract method (apply_discount)
- Concrete subclasses (PhysicalProduct, DigitalProduct, SubscriptionProduct)
- Properties with validation for price and stock
- Composition (Cart, Order)
- Class method as alternative constructor
- __str__, __repr__, __len__
- Raising exceptions for invalid operations

## Step 1 Abstract Product base class

All products have a name, SKU, and price. Price is validated through a property to prevent negative values. The apply_discount() method is abstract - each product type handles discounts differently. A class method provides an alternative constructor for loading products from a dictionary (e.g. from a database or JSON file).

**CODE - STEP 1**

```
from abc import ABC, abstractmethod


class Product(ABC):
    def __init__(self, name, sku, price):
        self.name  = name
        self.sku   = sku
        self.price = price   # uses the property setter

    @property
    def price(self):
        return self.__price

    @price.setter
    def price(self, value):
        if value < 0:
            raise ValueError(f"Price cannot be negative: {value}")
        self.__price = round(value, 2)

    @abstractmethod
    def apply_discount(self, percent):
        pass

    @classmethod
    def from_dict(cls, data):
        return cls(data["name"], data["sku"], data["price"])

    def __str__(self):
        return f"{self.name} (SKU: {self.sku}) - £{self.price:.2f}"

    def __repr__(self):
        return f"{type(self).__name__}('{self.name}', '{self.sku}', {self.price})"
```

## Step 2 Concrete product subclasses

PhysicalProduct tracks stock and cannot be purchased if out of stock. DigitalProduct has unlimited stock but tracks download count. SubscriptionProduct has a monthly price and billing cycle. Each implements apply_discount() in its own way.

**CODE - STEP 2**

```
class PhysicalProduct(Product):
    def __init__(self, name, sku, price, stock, weight_kg):
        super().__init__(name, sku, price)
        self.stock     = stock
        self.weight_kg = weight_kg

    def apply_discount(self, percent):
        self.price = self.price * (1 - percent / 100)

    def purchase(self, quantity=1):
        if quantity > self.stock:
            raise ValueError(f"Only {self.stock} in stock")
        self.stock -= quantity
        return f"Purchased {quantity}x {self.name}"

    def __str__(self):
        return super().__str__() + f" [Stock: {self.stock}]"


class DigitalProduct(Product):
    def __init__(self, name, sku, price, file_size_mb):
        super().__init__(name, sku, price)
        self.file_size_mb  = file_size_mb
        self.__download_count = 0

    def apply_discount(self, percent):
        if percent > 50:
            percent = 50   # cap digital discounts at 50%
        self.price = self.price * (1 - percent / 100)

    def download(self):
        self.__download_count += 1
        return f"Downloading {self.name} ({self.file_size_mb}MB)..."

    def download_count(self):
        return self.__download_count

    def __str__(self):
        return super().__str__() + f" [Digital, {self.file_size_mb}MB]"
```

## Step 3 ShoppingCart with __len__ and validation

The cart stores (product, quantity) tuples. Using __len__ means len(cart) returns the number of items. The total property calculates the running total. remove_item() raises a descriptive exception if the item is not in the cart.

**CODE - STEP 3**

```
class ShoppingCart:
    def __init__(self, owner):
        self.owner = owner
        self.__items = []   # list of (Product, int) tuples

    def add_item(self, product, quantity=1):
        if not isinstance(product, Product):
            raise TypeError("Only Product objects can be added")
        for i, (p, q) in enumerate(self.__items):
            if p.sku == product.sku:
                self.__items[i] = (p, q + quantity)
                return
        self.__items.append((product, quantity))

    def remove_item(self, sku):
        for i, (p, q) in enumerate(self.__items):
            if p.sku == sku:
                self.__items.pop(i)
                return
        raise ValueError(f"Item with SKU {sku} not in cart")

    @property
    def total(self):
        return sum(p.price * q for p, q in self.__items)

    def __len__(self):
        return sum(q for _, q in self.__items)

    def __str__(self):
        if not self.__items:
            return f"{self.owner}'s cart is empty"
        lines = [f"{self.owner}'s cart:"]
        for product, qty in self.__items:
            lines.append(f"  {qty}x {product.name} @ £{product.price:.2f}")
        lines.append(f"  Total: £{self.total:.2f}")
        return "\n".join(lines)

    def get_items(self):
        return list(self.__items)
```

## Step 4 Order class finalising a purchase

An Order is created from a cart snapshot. It has a status that moves through defined states. A PhysicalProduct's stock is decremented when the order is placed. A DigitalProduct triggers a download. This separates cart logic (ongoing) from order logic (a completed record).

**CODE - STEP 4**

```
from datetime import datetime


class Order:
    STATUSES = ["pending", "confirmed", "dispatched", "delivered", "cancelled"]
    order_count = 0

    def __init__(self, cart):
        Order.order_count += 1
        self.order_id  = Order.order_count
        self.owner     = cart.owner
        self.__items   = cart.get_items()
        self.total     = cart.total
        self.__status  = "pending"
        self.placed_at = datetime.now()

    def confirm(self):
        for product, qty in self.__items:
            if isinstance(product, PhysicalProduct):
                product.purchase(qty)
            elif isinstance(product, DigitalProduct):
                for _ in range(qty):
                    product.download()
        self.__status = "confirmed"
        print(f"Order #{self.order_id} confirmed. Total: £{self.total:.2f}")

    def advance_status(self):
        idx = Order.STATUSES.index(self.__status)
        if idx < len(Order.STATUSES) - 1:
            self.__status = Order.STATUSES[idx + 1]
        else:
            print("Order is already in final state")

    @property
    def status(self):
        return self.__status

    def __str__(self):
        date_str = self.placed_at.strftime("%d/%m/%Y")
        return f"Order #{self.order_id} [{date_str}] for {self.owner} -
{self.__status.upper()} - £{self.total:.2f}"
```

## Step 5 Full demonstration

The demonstration creates a range of products, adds them to a cart, applies a discount, then places an order and moves it through statuses. Every class does its own job, and the classes communicate through well-defined methods.

**CODE - STEP 5**

```
if __name__ == "__main__":
    keyboard = PhysicalProduct("Mechanical Keyboard", "KB-001", 89.99, stock=15,
weight_kg=1.2)
    ebook    = DigitalProduct("Python OOP Guide",    "EB-101", 12.99, file_size_mb=8)
    headset  = PhysicalProduct("Gaming Headset",     "HS-202", 49.99, stock=3,
weight_kg=0.4)

    keyboard.apply_discount(10)
    ebook.apply_discount(20)

    cart = ShoppingCart("Alice")
    cart.add_item(keyboard, 1)
    cart.add_item(ebook, 2)
    cart.add_item(headset, 1)

    print(cart)
    print(f"Total items in cart: {len(cart)}")

    order = Order(cart)
    order.confirm()
    order.advance_status()   # confirmed -> dispatched
    print(order)
    print(f"eBook downloads so far: {ebook.download_count()}")
```

**COMPLETE PROGRAM**

```python
from abc import ABC, abstractmethod
from datetime import datetime


class Product(ABC):
    def __init__(self, name, sku, price):
        self.name  = name
        self.sku   = sku
        self.price = price

    @property
    def price(self):
        return self.__price

    @price.setter
    def price(self, value):
        if value < 0:
            raise ValueError("Price cannot be negative")
        self.__price = round(value, 2)

    @abstractmethod
    def apply_discount(self, percent):
        pass

    def __str__(self):
        return f"{self.name} (SKU: {self.sku}) - £{self.price:.2f}"


class PhysicalProduct(Product):
    def __init__(self, name, sku, price, stock, weight_kg):
        super().__init__(name, sku, price)
        self.stock     = stock
        self.weight_kg = weight_kg

    def apply_discount(self, percent):
        self.price = self.price * (1 - percent / 100)

    def purchase(self, quantity=1):
        if quantity > self.stock:
            raise ValueError(f"Only {self.stock} in stock")
        self.stock -= quantity

    def __str__(self):
        return super().__str__() + f" [Stock: {self.stock}]"


class DigitalProduct(Product):
    def __init__(self, name, sku, price, file_size_mb):
        super().__init__(name, sku, price)
        self.file_size_mb     = file_size_mb
        self.__download_count = 0

    def apply_discount(self, percent):
        self.price = self.price * (1 - min(percent, 50) / 100)

    def download(self):
        self.__download_count += 1

    def download_count(self):
        return self.__download_count


class ShoppingCart:
    def __init__(self, owner):
        self.owner   = owner
        self.__items = []
```

```
    def add_item(self, product, quantity=1):
        for i, (p, q) in enumerate(self.__items):
            if p.sku == product.sku:
                self.__items[i] = (p, q + quantity)
                return
        self.__items.append((product, quantity))

    @property
    def total(self):
        return sum(p.price * q for p, q in self.__items)

    def __len__(self):
        return sum(q for _, q in self.__items)

    def get_items(self):
        return list(self.__items)

    def __str__(self):
        lines = [f"{self.owner}'s cart:"]
        for p, q in self.__items:
            lines.append(f"  {q}x {p.name} @ £{p.price:.2f}")
        lines.append(f"  Total: £{self.total:.2f}")
        return "\n".join(lines)


class Order:
    STATUSES    = ["pending", "confirmed", "dispatched", "delivered"]
    order_count = 0

    def __init__(self, cart):
        Order.order_count += 1
        self.order_id  = Order.order_count
        self.owner     = cart.owner
        self.__items   = cart.get_items()
        self.total     = cart.total
        self.__status  = "pending"
        self.placed_at = datetime.now()

    def confirm(self):
        for product, qty in self.__items:
            if isinstance(product, PhysicalProduct):
                product.purchase(qty)
            elif isinstance(product, DigitalProduct):
                for _ in range(qty):
                    product.download()
        self.__status = "confirmed"

    def advance_status(self):
        idx = Order.STATUSES.index(self.__status)
        if idx < len(Order.STATUSES) - 1:
            self.__status = Order.STATUSES[idx + 1]

    @property
    def status(self):
        return self.__status

    def __str__(self):
        date_str = self.placed_at.strftime("%d/%m/%Y")
        return f"Order #{self.order_id} [{date_str}] - {self.__status.upper()} - £
{self.total:.2f}"


if __name__ == "__main__":
    keyboard = PhysicalProduct("Mechanical Keyboard", "KB-001", 89.99, 15, 1.2)
    ebook    = DigitalProduct("Python OOP Guide",    "EB-101", 12.99, 8)
    headset  = PhysicalProduct("Gaming Headset",     "HS-202", 49.99, 3,  0.4)

    keyboard.apply_discount(10)
```

```
    ebook.apply_discount(20)

    cart = ShoppingCart("Alice")
    cart.add_item(keyboard, 1)
    cart.add_item(ebook,    2)
    cart.add_item(headset,  1)

    print(cart)
    order = Order(cart)
    order.confirm()
    order.advance_status()
    print(order)
```

### Extension Ideas

- Add a VoucherCode class that applies a discount to an entire cart.
- Create a Wishlist class that users can move items from into a cart.
- Add a ReviewSystem that stores Review objects on each product.
- Implement a CSV import using PhysicalProduct.from_dict() on each row.
- Add a stock notification system using an observer-style pattern.

# Part 6 - Answers

All answers, model outputs, debug solutions, and exam mark schemes. Attempt each task before consulting this section.

## 6.1 Introduction Retrieval Quiz (Section 1.6)

**Q1 - What is a class?**

A blueprint or template that defines the attributes and methods an object will have.

**Q2 - What is an object?**

A specific instance created from a class. Each object has its own copy of the class's attributes.

**Q3 - Name the four pillars of OOP.**

Encapsulation, Inheritance, Polymorphism, Abstraction.

**Q4 - What is a method?**

A function defined inside a class that operates on the object's data.

**Q5 - Give one advantage of OOP over procedural programming.**

Any one of: reusability, maintainability, modularity, scalability, real-world modelling.

## 6.2 Predict the Output (Part 2)

**ENCAPSULATION (1 OF 2)**

```
3
```

**ENCAPSULATION (2 OF 2)**

```
Gamma
Beta

(Objects are independent — changing b1's attribute does not affect b2.)
```

**INHERITANCE (1 OF 2)**

```
Vehicle: Ford
Car: Toyota

(Car overrides describe(), so its version runs instead of Vehicle's.)
```

**INHERITANCE (2 OF 2)**

```
Hello from Parent
Hello from Child

(super().greet() runs the parent version first, then the child continues.)
```

**POLYMORPHISM**

```
[PLAIN] Report
[COLOR] Report
[PDF]   Report
[PLAIN] Report
```

**ABSTRACTION**

```
I am Donald and I say quack
```

## 6.3 Debug Task Solutions (Part 2)

**ENCAPSULATION - THE BUG**

Line 3: `high_score = 0` creates a local variable inside __init__ that is discarded when the method ends. `self.high_score` is never created as an instance attribute.

**ENCAPSULATION - THE FIX**

Change line 3 to `self.high_score = 0`.

**INHERITANCE - THE BUG**

Line 10: Dog.__init__ sets self.breed but never calls super().__init__(name). The Animal constructor never runs, so self.name is never set.

**INHERITANCE - THE FIX**

Add `super().__init__(name)` as the first line inside Dog.__init__.

**POLYMORPHISM - THE BUG**

Line 6: Cat defines talk() instead of speak(). The loop calls speak() on every object, but Cat doesn't have a speak() method.

**POLYMORPHISM - THE FIX**

Rename Cat's method from talk() to speak().

**ABSTRACTION - THE BUG**

Line 12: Circle implements calculate_area() instead of area(). The abstract method area() is never overridden, so Python refuses to create a Circle object.

**ABSTRACTION - THE FIX**

Rename calculate_area to area.

## 6.4 Pillars Retrieval Quiz (Section 2.5)

**Q1 - Which pillar prevents direct access to an object's internal data?**

Encapsulation. Private attributes restrict direct access; public methods provide controlled access.

**Q2 - In Python, how do you make an attribute private?**

Prefix the attribute name with double underscores, e.g. self.__balance.

**Q3 - What keyword is used in Python to call the parent class's constructor?**

super().__init__() calls the parent class constructor.

**Q4 - What is the difference between a parent class and a subclass?**

The parent (superclass) defines shared attributes and methods. The subclass inherits them and can add or override behaviour.

**Q5 - What allows different objects to respond to the same method call differently?**

Polymorphism. Each class provides its own implementation of the shared method name.

**Q6 - What must you import to create an abstract class in Python?**

from abc import ABC, abstractmethod

**Q7 - What happens if a subclass does not implement all abstract methods?**

A TypeError is raised when you try to create an object from that subclass.

**Q8 - What does UML stand for and why is it used?**

Unified Modelling Language. It provides a standard visual notation for designing and documenting class structures before coding.

## 6.5 Exam Question Mark Schemes (Part 4)

### Question 1 [2 marks]

- A class is a template/blueprint (1) from which objects are created/instantiated (1).
- Accept: 'defines the attributes and methods that objects of that type will have'.

**Question 2 [2 marks]**

- A class is the definition/blueprint (1); an object is a specific instance created from that class (1).
- Accept an analogy: 'the class is the cookie cutter; the object is the cookie'.

**Question 3 [4 marks]**

- Encapsulation: bundling data (attributes) and methods that operate on that data together in a class (1).
- Encapsulation: restricting direct access to an object's internal data from outside the class (1).
- Advantage: prevents external code from setting attributes to invalid values (1).
- Advantage: the internal implementation can be changed without breaking code that uses the class (1).

**Question 4 [4 marks]**

- A subclass/child class can take on/inherit the attributes and methods of a superclass/parent class (1).
- Without needing them to be rewritten/redefined (1).
- Benefit: reduces code duplication (1).
- Benefit: a change to the parent class automatically applies to all subclasses (1).

**Question 5 [4 marks]**

- Different classes can share the same method name (1).
- Each class provides its own implementation of that method (1).
- Code that calls the method does not need to know the specific class of the object (1).
- Suitable example: different Shape subclasses each with an area() method (1).

**Question 6 [2 marks]**

- An abstract class cannot be instantiated directly; a concrete class can (1).
- An abstract class contains at least one abstract method that subclasses must implement; a concrete class provides implementations for all its methods (1).

**Question 7 [6 marks]**

- class Book with __init__(self, title, author, isbn) correctly defined (1).
- self.__on_loan = False (private attribute, correctly named with double underscore) (1).
- check_out(): sets self.__on_loan = True (1); prints message / raises error if already on loan (1).
- return_book(): sets self.__on_loan = False (1).
- __str__: returns correctly formatted string using attribute values (1).

**Question 8 [6 marks]**

- class ElectricVehicle(Vehicle): correct syntax (1).
- __init__ calls super().__init__(make, model, year) correctly (1).
- self.battery_kWh = battery_kWh added (1).
- describe() overrides correctly and calls super().describe() (1).
- Returns super().describe() + ' (Electric)' or equivalent (1).
- @property decorator on range_km, returns self.battery_kWh * 6 (1).

### Question 9 [4 marks]

- Rex: Woof (1)
- Luna: Meow (1)
- Bob: Some sound (1)
- All three lines printed in correct order with no additional output (1 for all three correct).

### Question 10 [4 marks]

- __str__ defines a string representation of an object (1).
- It is called automatically when the object is passed to print() or str() (1).
- Without __str__, print(obj) outputs a memory address / unhelpful default string (1).
- Useful because it provides readable output for debugging and display (1).

### Question 11 [2 marks]

- A subclass provides its own implementation of a method (1) that was defined in the parent class (1).

### Question 12 [8 marks]

- Song class with correct __init__(title, artist, duration) (1).
- Song __str__ returns a readable string with title and artist (1).
- Playlist class with correct __init__(name) and self.songs = [] (1).
- add_song(song) appends to self.songs (1).
- total_duration() sums durations and converts to mm:ss correctly (1).
- Playlist __str__ lists songs (1).
- PremiumPlaylist inherits from Playlist, adds max_songs attribute (1).
- add_song() overridden to check limit and print/raise error if reached (1).

# CodeBash

codebash.co.uk/resources

More free resources for CS teachers