

Study Delight Publications

From  
Classes  
to  
Real-World  
Projects

# PYTHON

## Simplified

Learn  
Practice  
Build  
Apply

### PART 3

## Object-Oriented Programming



Classes &  
Objects



Inheritance



OOP  
Concepts

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

    def study(self):
        print(f"{self.name} is
        learning Python!")
```



Encapsulation

Inheritance

Polymorphism

Abstraction



Abstraction  
& Polymorphism



Modules  
& Packages



Real-World  
Projects



Clear  
Explanations



Practical  
Examples



Step-by-Step  
Projects



Real-World  
Applications



Career Ready  
Skills

[pythonforaiml.com](https://pythonforaiml.com)

# PYTHON PART 3

## Object-Oriented Python — Index

30-PAGE ROADMAP • LEARN • PRACTICE • BUILD

01 What Is a Class?

02 Objects & Instances

03 The self Parameter

04 `__init__` Constructor

05 Instance Attributes

06 Instance Methods

07 Class Attributes

08 Instance vs Class Attribute

09 Encapsulation

10 Properties

11 Inheritance

12 Method Overriding

13 `super()`

14 Multiple Inheritance

15 Method Resolution Order

16 `object` — The Base Class

17 `isinstance()`

18 `__str__`

19 `__repr__`

20 Class Methods

21 Static Methods

22 Dataclasses

23 Composition

24 Abstract Base Classes

25 Operator Overloading

26 Exception Handling in Classes

27 OOP Design Principles

28 Mini Project: Bank Account

29 Mini Project: Library System

30 Part 3 Review & Challenge

[pythonforaiml.com](https://pythonforaiml.com)

Part 3 Index • Object-Oriented Python

## Part 3: OOP Foundations — What Is a Class?

### Explanation

Part 3 develops Object-Oriented Programming (OOP), a way to organise larger programs around objects that combine data with related behaviour. A class acts as a blueprint, while an object is a concrete instance created from that blueprint.

This part moves from the simplest class to constructors, instance state, inheritance, polymorphism, special methods, dataclasses, composition and practical mini-projects.

### Example

```
class Student:
    def study(self):
        print("Student is studying")

s = Student()
s.study()
```

### Output / Result

Student is studying

### Key Idea

The class contains a method. When `s.study()` is called, Python uses the object `s` as the first argument to the method.

### Practice

By the end of Part 3, you should be able to design small classes, choose between inheritance and composition, validate object state, and build a practical OOP model.

### Study Loop

- Predict the output before running the example.
- Change one value and observe the object behaviour.
- Explain the concept without looking at the page.
- Write one small variation from scratch.



## 2. Objects & Instances

### Explanation

An object is a concrete instance of a class. One class can create many independent objects. Each object can hold its own state while sharing the same methods.

### Example

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

dog1 = Dog()
dog2 = Dog()
dog1.bark()
dog2.bark()
```

### Output / Result

```
Woof!
Woof!
```

### Think

dog1 and dog2 are different objects. They were created from the same Dog class, so both have access to bark().

### Practice

Create two Product objects from one Product class.

### Study Loop

- Predict the output before running the example.
- Change one value and observe the object behaviour.
- Explain the concept without looking at the page.
- Write one small variation from scratch.



### 3. The self Parameter

#### Explanation

Inside an instance method, self refers to the current object. It lets a method access attributes and other methods belonging to that particular object.

#### Example

```
class Student:
    def show(self):
        print("This object is:", self)

s = Student()
s.show()
```

#### Output / Result

This object is: <\_\_main\_\_.Student object ...>

#### Important

self is not a special keyword; it is the conventional parameter name for the current instance. Python supplies the instance automatically when you call s.show().

#### Practice

Remember that self is written in the method definition but not normally supplied in the method call.

#### Study Loop

- Predict the output before running the example.
- Change one value and observe the object behaviour.
- Explain the concept without looking at the page.
- Write one small variation from scratch.



## 4. \_\_init\_\_ Constructor

### Explanation

`__init__` is an initializer method that runs automatically when an object is created. It is commonly used to give the object its initial state.

### Example

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

s = Student("Asha", 21)
print(s.name, s.age)
```

### Output / Result

Asha 21

### Object Setup

When `Student("Asha",21)` runs, Python creates an object and calls `__init__` with that object plus the supplied arguments.

### Practice

Add a course attribute and print it.

### Study Loop

- Predict the output before running the example.
- Change one value and observe the object behaviour.
- Explain the concept without looking at the page.
- Write one small variation from scratch.



## 5. Instance Attributes

### Explanation

An instance attribute belongs to one particular object. Two objects created from the same class can have different attribute values.

### Example

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

a = BankAccount("Asha", 5000)
b = BankAccount("Ravi", 3000)
print(a.balance)
print(b.balance)
```

### Output / Result

```
5000
3000
```

### State

self.balance is stored separately for each object. Changing a.balance does not automatically change b.balance.

### Practice

Create two Employee objects with different names and salaries.

### Study Loop

- Predict the output before running the example.
- Change one value and observe the object behaviour.
- Explain the concept without looking at the page.
- Write one small variation from scratch.



## 6. Instance Methods

### Explanation

An instance method describes behaviour that operates on an object's state. It can read or modify attributes through self.

### Example

```
class Counter:
    def __init__(self):
        self.value = 0
    def increment(self):
        self.value += 1

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

### Output / Result

2

### Behaviour

The method changes the state stored inside c. This is one of the central ideas of OOP: data and related operations live together.

### Practice

Add a reset() method that sets the counter back to zero.

### Study Loop

- Predict the output before running the example.
- Change one value and observe the object behaviour.
- Explain the concept without looking at the page.
- Write one small variation from scratch.



## 7. Class Attributes

### Explanation

A class attribute is shared at the class level. It is useful for values that are common to every instance, such as a category name or a company-wide setting.

### Example

```
class Employee:
    company = "TechCorp"
    def __init__(self, name):
        self.name = name

a = Employee("Asha")
b = Employee("Ravi")
print(a.company)
print(b.company)
```

### Output / Result

```
TechCorp
TechCorp
```

### Shared Data

Unless an instance has its own attribute with the same name, Python can find the value on the class.

### Practice

Create a class attribute `category = 'Student'` and access it from two objects.

### Study Loop

- Predict the output before running the example.
- Change one value and observe the object behaviour.
- Explain the concept without looking at the page.
- Write one small variation from scratch.



## 8. Instance vs Class Attribute

### Explanation

The difference is important: instance attributes usually represent object-specific state, while class attributes represent values shared by the class.

### Example

```
class Student:
    school = "ABC School"
    def __init__(self, name):
        self.name = name

a = Student("Asha")
a.school = "New School"
print(a.school)
print(Student.school)
```

### Output / Result

```
New School
ABC School
```

### Compare

Assigning `a.school` creates an instance attribute that shadows the class attribute for that object. The class attribute itself remains unchanged.

### Practice

Predict what another `Student` object prints for `school`.

### Study Loop

- Predict the output before running the example.
- Change one value and observe the object behaviour.
- Explain the concept without looking at the page.
- Write one small variation from scratch.



## 9. Encapsulation

### Explanation

Encapsulation means keeping data and the operations that work with it together and controlling how the state is changed. Python does not enforce strict private fields like some languages, but naming conventions communicate intent.

### Example

```
class BankAccount:
    def __init__(self, balance):
        self._balance = balance
    def deposit(self, amount):
        if amount > 0:
            self._balance += amount

account = BankAccount(1000)
account.deposit(500)
print(account._balance)
```

### Output / Result

1500

### Design Principle

A leading underscore conventionally means: this attribute is intended for internal use. A method can enforce rules before changing the state.

### Practice

Add a `withdraw()` method that refuses to create a negative balance.

### Study Loop

- Predict the output before running the example.
- Change one value and observe the object behaviour.
- Explain the concept without looking at the page.
- Write one small variation from scratch.



## 10. Properties

### Explanation

A property lets an attribute-like expression use method logic behind the scenes. This can provide a clean interface while still validating or calculating a value.

### Example

```
class Temperature:
    def __init__(self, celsius):
        self._celsius = celsius
    @property
    def celsius(self):
        return self._celsius

t = Temperature(25)
print(t.celsius)
```

### Output / Result

25

### Interface

The @property decorator turns the celsius method into an attribute-style access. It is useful when reading a value requires controlled logic.

### Practice

Create a property that returns Fahrenheit from stored Celsius.

### Study Loop

- Predict the output before running the example.
- Change one value and observe the object behaviour.
- Explain the concept without looking at the page.
- Write one small variation from scratch.



## 11. Inheritance

### Explanation

Inheritance allows a new class to reuse and extend another class. The child class receives accessible behaviour from the parent and can add or customise it.

### Example

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

class Dog(Animal):
    pass

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

### Output / Result

Some sound

### Reuse

Dog inherits speak() from Animal. Inheritance is useful when there is a genuine 'is-a' relationship.

### Practice

Create Cat(Animal) and reuse speak() before overriding it.

### Study Loop

- Predict the output before running the example.
- Change one value and observe the object behaviour.
- Explain the concept without looking at the page.
- Write one small variation from scratch.



## 12. Method Overriding

### Explanation

A child class can provide its own implementation of a method inherited from its parent. This is called overriding.

### Example

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

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

Dog().speak()
```

### Output / Result

Woof

### Polymorphism

Python chooses the most specific implementation available on the object's class. Dog's `speak()` therefore replaces the inherited version for Dog objects.

### Practice

Override `speak()` in a Cat class.

### Study Loop

- Predict the output before running the example.
- Change one value and observe the object behaviour.
- Explain the concept without looking at the page.
- Write one small variation from scratch.



## 13. super()

### Explanation

super() provides a convenient way to call behaviour from a parent class. It is especially useful when a child class wants to extend rather than completely replace the parent's initialization.

### Example

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

class Student(Person):
    def __init__(self, name, course):
        super().__init__(name)
        self.course = course

s = Student("Asha", "Python")
print(s.name, s.course)
```

### Output / Result

Asha Python

### Parent Reuse

super().\_\_init\_\_(name) runs the parent initialization so Student does not need to duplicate the name assignment.

### Practice

Add an age attribute in Student while still reusing Person's constructor.

### Study Loop

- Predict the output before running the example.
- Change one value and observe the object behaviour.
- Explain the concept without looking at the page.
- Write one small variation from scratch.



## 14. Multiple Inheritance

### Explanation

Python allows a class to inherit from more than one parent. This can be useful, but it should be used carefully because multiple inheritance can make relationships harder to understand.

### Example

```
class Flyer:
    def fly(self):
        print("Flying")

class Swimmer:
    def swim(self):
        print("Swimming")

class Duck(Flyer, Swimmer):
    pass

d = Duck()
d.fly()
d.swim()
```

### Output / Result

```
Flying
Swimming
```

### Use Carefully

Duck receives methods from both parents. Python uses a method resolution order (MRO) to decide which implementation to use when names overlap.

### Practice

Find the MRO of Duck using `Duck.mro()`.

### Study Loop

- Predict the output before running the example.
- Change one value and observe the object behaviour.
- Explain the concept without looking at the page.
- Write one small variation from scratch.



## 15. Method Resolution Order

### Explanation

When inheritance becomes complex, Python needs a consistent rule for finding methods. The Method Resolution Order, or MRO, describes the search path.

### Example

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

print(C.mro())
```

### Output / Result

```
[<class '__main__.C'>, <class '__main__.B'>, <class '__main__.A'>, <class 'object'>]
```

### Trace It

Python searches C, then B, then A, and finally object. Understanding MRO helps explain behaviour in multiple-inheritance designs.

### Practice

Use `C.mro()` on a small hierarchy you create yourself.

### Study Loop

- Predict the output before running the example.
- Change one value and observe the object behaviour.
- Explain the concept without looking at the page.
- Write one small variation from scratch.



## 16. object — The Base Class

### Explanation

Every normal Python class ultimately derives from object. This provides fundamental behaviour and methods that Python objects use.

### Example

```
class Student:
    pass

s = Student()
print(isinstance(s, object))
```

### Output / Result

True

### Foundation

Even when you do not explicitly write `class Student(object)`, modern Python automatically makes `Student` inherit from `object`.

### Practice

Check `isinstance(10, object)` and `isinstance([], object)`.

### Study Loop

- Predict the output before running the example.
- Change one value and observe the object behaviour.
- Explain the concept without looking at the page.
- Write one small variation from scratch.



## 17. isinstance()

### Explanation

`isinstance()` checks whether an object is an instance of a class or compatible subclass. It is useful when a program genuinely needs to branch based on object type.

### Example

```
class Animal: pass
class Dog(Animal): pass

d = Dog()
print(isinstance(d, Dog))
print(isinstance(d, Animal))
```

### Output / Result

```
True
True
```

### Type Check

Because Dog inherits from Animal, a Dog object is also considered an instance of Animal.

### Practice

Test a few values with `isinstance(value, int)` and `isinstance(value, str)`.

### Study Loop

- Predict the output before running the example.
- Change one value and observe the object behaviour.
- Explain the concept without looking at the page.
- Write one small variation from scratch.



## 18. \_\_str\_\_

### Explanation

The `__str__` method controls the human-friendly string representation used by `str()` and `print()` for an object.

### Example

```
class Product:
    def __init__(self, name, price):
        self.name = name
        self.price = price
    def __str__(self):
        return f"{self.name} - ₹{self.price:.2f}"

p = Product("Book", 499)
print(p)
```

### Output / Result

Book - ₹499.00

### Readable Objects

Without a custom `__str__`, printing an object normally gives a less useful representation. A good `__str__` makes objects easier to inspect.

### Practice

Create a `Student.__str__` that displays name and course.

### Study Loop

- Predict the output before running the example.
- Change one value and observe the object behaviour.
- Explain the concept without looking at the page.
- Write one small variation from scratch.



## 19. \_\_repr\_\_

### Explanation

`__repr__` is intended to provide a useful representation for developers and debugging. A good `repr` often contains enough information to understand the object's state.

### Example

```
class Point:
    def __init__(self,x,y):
        self.x=x; self.y=y
    def __repr__(self):
        return f"Point(x={self.x}, y={self.y})"

print(Point(2,3))
```

### Output / Result

```
Point(x=2, y=3)
```

### Debugging

While `__str__` focuses on friendly display, `__repr__` commonly focuses on an informative developer representation.

### Practice

Add `__repr__` to a `Product` class.

### Study Loop

- Predict the output before running the example.
- Change one value and observe the object behaviour.
- Explain the concept without looking at the page.
- Write one small variation from scratch.



## 20. Class Methods

### Explanation

A class method receives the class as its first argument, conventionally named `cls`. It can be called through the class and is often used for alternative constructors or class-level behaviour.

### Example

```
class Student:
    school = "ABC"
    @classmethod
    def school_name(cls):
        return cls.school

print(Student.school_name())
```

### Output / Result

ABC

`cls`

The `@classmethod` decorator changes the first argument from an instance to the class itself.

### Practice

Create a class method that returns a class-level category.

### Study Loop

- Predict the output before running the example.
- Change one value and observe the object behaviour.
- Explain the concept without looking at the page.
- Write one small variation from scratch.



## 21. Static Methods

### Explanation

A static method belongs conceptually to a class but does not need an instance or class reference. It is useful for helper logic closely related to the class's purpose.

### Example

```
class MathTools:  
    @staticmethod  
    def square(n):  
        return n*n  
  
print(MathTools.square(7))
```

### Output / Result

49

### Helper Logic

A static method is still namespaced under the class, but Python does not automatically pass self or cls.

### Practice

Create a static method `is_even(n)`.

### Study Loop

- Predict the output before running the example.
- Change one value and observe the object behaviour.
- Explain the concept without looking at the page.
- Write one small variation from scratch.



## 22. Dataclasses

### Explanation

Dataclasses reduce boilerplate for classes that mainly store data. The `@dataclass` decorator can generate an initializer and useful representations automatically.

### Example

```
from dataclasses import dataclass

@dataclass
class Student:
    name: str
    age: int

s = Student("Asha", 21)
print(s)
```

### Output / Result

```
Student(name='Asha', age=21)
```

### Modern Python

Dataclasses are excellent for simple records where you want readable, structured objects without manually writing every method.

### Practice

Create a Product dataclass with name, price and category.

### Study Loop

- Predict the output before running the example.
- Change one value and observe the object behaviour.
- Explain the concept without looking at the page.
- Write one small variation from scratch.



## 23. Composition

### Explanation

Composition means building a class from objects of other classes. Instead of saying 'is-a', composition often expresses 'has-a'. This can produce flexible designs.

### Example

```
class Engine:
    def start(self):
        print("Engine started")

class Car:
    def __init__(self):
        self.engine = Engine()
    def start(self):
        self.engine.start()

Car().start()
```

### Output / Result

Engine started

### Has-a

Car has an Engine. Composition can be easier to change than a deep inheritance hierarchy because each component can have a focused responsibility.

### Practice

Create a Computer that contains a CPU object.

### Study Loop

- Predict the output before running the example.
- Change one value and observe the object behaviour.
- Explain the concept without looking at the page.
- Write one small variation from scratch.



## 24. Abstract Base Classes

### Explanation

Sometimes a parent class should define an interface without providing a complete implementation. Abstract base classes can require child classes to implement specific methods.

### Example

```
from abc import ABC, abstractmethod

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

class Square(Shape):
    def __init__(self, side): self.side=side
    def area(self): return self.side**2

print(Square(5).area())
```

### Output / Result

25

### Contract

The abstract method communicates a contract: every concrete Shape should provide area(). Instantiating Shape directly is not the intended use.

### Practice

Create Circle(Shape) with an area() method.

### Study Loop

- Predict the output before running the example.
- Change one value and observe the object behaviour.
- Explain the concept without looking at the page.
- Write one small variation from scratch.



## 25. Operator Overloading

### Explanation

Python lets classes define how operators behave through special methods. For example, `__add__` can define what `+` means for your objects.

### Example

```
class Point:
    def __init__(self,x,y): self.x=x; self.y=y
    def __add__(self,other):
        return Point(self.x+other.x,self.y+other.y)
    def __repr__(self): return f"Point({self.x},{self.y})"

print(Point(1,2)+Point(3,4))
```

### Output / Result

Point(4,6)

### Special Methods

Operator overloading can make domain-specific objects feel natural to use, but the behaviour should remain intuitive.

### Practice

Define `__eq__` for a simple Point class.

### Study Loop

- Predict the output before running the example.
- Change one value and observe the object behaviour.
- Explain the concept without looking at the page.
- Write one small variation from scratch.



## 26. Exception Handling in Classes

### Explanation

Classes often validate input and may raise exceptions when an operation is invalid. Exception handling keeps failure paths explicit and prevents silent corruption of state.

### Example

```
class BankAccount:
    def __init__(self, balance=0): self.balance=balance
    def withdraw(self, amount):
        if amount > self.balance:
            raise ValueError("Insufficient balance")
        self.balance -= amount

try:
    BankAccount(100).withdraw(200)
except ValueError as e:
    print(e)
```

### Output / Result

Insufficient balance

### Safe State

Raising an exception is appropriate when an operation cannot satisfy its contract. The caller can decide how to handle the failure.

### Practice

Add validation that a deposit amount must be positive.

### Study Loop

- Predict the output before running the example.
- Change one value and observe the object behaviour.
- Explain the concept without looking at the page.
- Write one small variation from scratch.



## 27. OOP Design Principles

### Explanation

Good OOP is not about creating a class for every small thing. The real goal is clear responsibility. A class should have a meaningful purpose and a small, understandable interface.

### Example

Example design:

Student → stores student data

Course → stores course information

Enrollment → connects Student and Course

Report → generates output

### Output / Result

Separate responsibilities make code easier to change and test.

### Design Check

Think about which object owns each piece of behaviour. Avoid classes that become giant collections of unrelated methods.

### Practice

Design Check

### Study Loop

- Predict the output before running the example.
- Change one value and observe the object behaviour.
- Explain the concept without looking at the page.
- Write one small variation from scratch.



## 28. Mini Project: Bank Account

### Explanation

Let's combine constructors, instance state, methods, validation and string representation into a small practical class. The object protects its state by exposing meaningful operations.

### Example

```
class BankAccount:
    def __init__(self, owner, balance=0):
        self.owner=owner
        self.balance=balance
    def deposit(self, amount):
        if amount<=0: raise ValueError("Invalid amount")
        self.balance += amount
    def withdraw(self, amount):
        if amount>self.balance: raise ValueError("Insufficient")
        self.balance -= amount
    def __str__(self):
        return f"{self.owner}: ₹{self.balance:.2f}"
```

### Output / Result

```
account=BankAccount("Asha",1000)
account.deposit(500)
account.withdraw(200)
print(account)
Asha: ₹1300.00
```

### Build

This project demonstrates a useful OOP pattern: state is stored in the object, while methods enforce the rules for changing that state.

### Practice

Extend the project with `transfer(other, amount)`. Validate both accounts before changing either balance.

### Study Loop

- Predict the output before running the example.
- Change one value and observe the object behaviour.
- Explain the concept without looking at the page.
- Write one small variation from scratch.



## 29. Mini Project: Library System

### Explanation

A small library system is a good exercise in composition. A Library can contain Book objects, and methods can handle adding, searching and borrowing books.

### Example

```
class Book:
    def __init__(self, title):
        self.title = title
        self.available = True
    def borrow(self):
        if not self.available: return False
        self.available = False
        return True

book = Book("Python")
print(book.borrow())
print(book.borrow())
```

### Output / Result

```
True
False
```

### Project Step

The Book object owns its availability state. A larger Library class could manage many Book objects and provide search or reporting features.

### Practice

Add Library.add\_book(), find\_book() and list\_available().

### Study Loop

- Predict the output before running the example.
- Change one value and observe the object behaviour.
- Explain the concept without looking at the page.
- Write one small variation from scratch.



## 30. Part 3 Review & Challenge

### Explanation

Part 3 introduced the core ideas of Object-Oriented Python: classes, objects, self, constructors, instance and class attributes, methods, encapsulation, inheritance, polymorphism, special methods, dataclasses and composition.

### Example

Review checklist:

1. Design a class with `__init__`.
2. Store instance state with `self`.
3. Add methods that protect valid state.
4. Use inheritance only when the relationship is genuine.
5. Prefer composition when objects have components.
6. Add `__str__` or `__repr__` when useful.

### Output / Result

Final Challenge: build a small e-commerce model with Product, Cart and Customer classes. Add products, calculate a cart total, validate quantities and display a readable receipt.

### Mastery

Part 4 will move into practical Python: files, exceptions, iterators, generators, testing, packages and real-world program structure.

### Practice

### Mastery

### Study Loop

- Predict the output before running the example.
- Change one value and observe the object behaviour.
- Explain the concept without looking at the page.
- Write one small variation from scratch.

