ScopeTreasure Courses

PYTHON COURSES · LESSON PY.12 · CODE SCOPE SERIES

Objects, Classes and a Mini Project

A class is a blueprint and an object is the thing built from it — learn classes, objects and methods, then build a student-marks mini project.

Introduction

So far, you have stored data in separate variables: one name here, one age there, one list of marks somewhere else. For one student that works. For twenty students it becomes a mess — sixty loose variables, and nothing tying each student's name to their own marks.

Object-oriented programming solves this by bundling data and behaviour together. A **class** is a blueprint — it describes what every student has (a name, a list of marks) and what every student can do (find an average, pass or fail). An **object** is one real thing built from that blueprint — Ali's record, Ayesha's record, each with its own values.

You have already used objects without noticing: a string is an object, and `.upper()` is one of its methods. In this lesson you will build your own class for the first time, and then use it in a mini project that ties together almost everything from this series.

(Roman Urdu: Class aik naqsha hai, aur object us naqshay se bani hui asal cheez hai — har student ka apna naam aur apnay marks, aik hi saaf package mein.)

Object-oriented code clicks the moment you stop thinking of a class as fancy syntax and start seeing it as a container: everything about one student — name, marks, average, result — lives in one place, and nothing about Ali can leak into Ayesha's record. When you find yourself keeping three parallel lists (names, marks, results) and hoping their positions stay in step, that is the exact moment a class earns its place.

Key points

- **A class is a blueprint; an object is one instance built from it.** `class Student:` defines the blueprint; `ali = Student("Ali")` builds one object.
- **Define a class with `class` and an indented body.** Method definitions inside the class use `def`, just like normal functions.
- **`__init__` is the setup method.** It runs automatically when an object is created, and it stores the starting values.
- **`self` means "this object".** Inside a method, `self.name` is that object's own name, different from every other object's name.
- **Attributes are the object's data; methods are its actions.** `ali.name` is an attribute, while `ali.average()` is a method call working on that object's own data.
- **Each object keeps its own values.** Changing `ali.marks` does not change `ayesha.marks`, even though both come from the same class.

- **Classes group related code together.** Instead of loose lists for names and marks, one list of student objects holds everything, neatly matched. Small scripts are fine without classes; classes earn their place when you have many similar things.
- **`__str__` controls what `print()` shows for an object.** Without it, printing an object shows an unhelpful memory address; with it, you get a readable summary.
- **Inheritance reuses a parent class.** A child class like `Teacher(Person)` gets the parent's methods, and adds or changes only what is different.

Examples

Example 1 — Your first class: a Student

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

    def add_mark(self, mark):
        self.marks.append(mark)

    def average(self):
        if len(self.marks) == 0:
            return 0
        return sum(self.marks) / len(self.marks)

ali = Student("Ali")
ali.add_mark(80)
ali.add_mark(90)
print(ali.name)
print(ali.marks)
print(ali.average())
```

What you will get:

```
Ali
[80, 90]
85.0
```

`Student("Ali")` runs `__init__`, storing `"Ali"` in `self.name` and starting an empty marks list. `average` uses `sum()` and `len()`; its `if` guard avoids dividing by zero for a student with no marks yet.

Example 2 — Mini project: a class report

This project ties the series together — classes, lists, loops, conditions, f-strings and a file — and it also shows independence: each student's marks stay with that student only:

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

    def add_mark(self, mark):
        self.marks.append(mark)

    def average(self):
        if len(self.marks) == 0:
            return 0
        return sum(self.marks) / len(self.marks)

    def result(self):
        if self.average() >= 50:
            return "Pass"
        return "Fail"

students = [Student("Ali"), Student("Ayesha"), Student("Bilal")]
students[0].add_mark(80)
students[0].add_mark(90)
students[1].add_mark(95)
students[1].add_mark(100)
students[2].add_mark(40)
students[2].add_mark(45)

lines = []
for student in students:
    line = f"{student.name}: average {student.average()}, {student.result()}"
    print(line)
    lines.append(line)

with open("report.txt", "w") as f:
    for line in lines:
        f.write(line + "\n")
print("Report saved to report.txt")
```

What you will get on the screen:

```
Ali: average 85.0, Pass
Ayesha: average 97.5, Pass
Bilal: average 42.5, Fail
Report saved to report.txt
```

And `report.txt` will hold the same three lines. Adding a fourth student is just one more line in the list.

Example 3 — A readable object with __str__

Print an object without `__str__` and you get something like `<__main__.Book object at 0x...>` — true, but useless. One short method fixes it:

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

    def __str__(self):
        return f"{self.title} ({self.pages} pages)"

book = Book("Python Basics", 120)
print(book)
print(book.title)
```

What you will get:

```
Python Basics (120 pages)
Python Basics
```

The first line is `print(book)`, which uses `__str__`; the second is `print(book.title)`, which prints the plain attribute. Adding `__str__` to every class you write makes debugging far easier, because your objects describe themselves.

Example 4 — Inheritance: a Teacher is a Person

A teacher is a person with one extra fact: a subject. Instead of copying the whole Person class, `Teacher` inherits from it and adds only the difference:

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

    def introduce(self):
        return f"I am {self.name}"

class Teacher(Person):
    def __init__(self, name, subject):
        super().__init__(name)
        self.subject = subject

teacher = Teacher("Sara", "Python")
print(teacher.introduce())
print(teacher.subject)
```

What you will get:

```
I am Sara
Python
```

`super().__init__(name)` runs the parent's setup, so the teacher gets a name the same way any person does. `introduce()` comes free with inheritance — the child class never defines it, yet the object can use it.

Example 5 — Real-world case: a shop product that sells itself

A shop product has a name, a price, and stock. Its `sell()` method checks the stock, reduces it, and returns the money earned — or returns 0 and changes nothing if there is not enough stock. The rules live inside the object, so no other code can sell what is not there:

```
class Product:
    def __init__(self, name, price, stock):
        self.name = name
        self.price = price
        self.stock = stock

    def sell(self, quantity):
        if quantity <= self.stock:
            self.stock = self.stock - quantity
            return self.price * quantity
        return 0

tea = Product("Tea", 200, 10)
print(tea.sell(3))
print("Stock left:", tea.stock)
print(tea.sell(20))
print("Stock left:", tea.stock)
```

What you will get:

```
600
Stock left: 7
0
Stock left: 7
```

Selling 3 teas earns 600 and leaves 7 in stock. Trying to sell 20 fails safely: it returns 0 and the stock stays 7. Real shop, bank and booking systems work exactly this way — objects guarding their own rules.

Quick recap

- A class is a blueprint; an object is one thing built from it.
- `__init__` sets up a new object; `self` refers to that object.
- Attributes hold an object's data; methods are its actions.
- Each object keeps its own values, independent of other objects.
- Classes shine when you have many similar things to manage.
- `__str__` gives an object a readable printed form.
- Inheritance lets a child class reuse a parent's methods and add its own.

Common mistakes

1. **Forgetting `self` in a method definition** — `def average():` inside a class will fail when called as `ali.average()`. Fix: always write `def average(self):`.

2. **Forgetting `self.` when storing data** — writing `name = name` in `__init__` creates a local variable that vanishes, instead of an attribute. Fix: write `self.name = name`.
3. **Calling the class without brackets** — `ali = Student` stores the blueprint itself, not an object. Fix: write `Student("Ali")` with brackets and arguments.
4. **Sharing one list between objects by accident** — a list created once outside `__init__` and assigned to every object is shared, so one student's marks appear in another's. Fix: create the list inside `__init__` with `self.marks = []`.
5. **Putting the class to work before defining it** — Python runs top to bottom, so creating an object before its class definition fails. Fix: define the class first, create objects after it.

1. **A shared list as a default argument** — `def __init__(self, marks=[])` reuses one list for every object, so one student's marks appear in another's. Fix: use `marks=None` and create a fresh list inside: `self.marks = [] if marks is None else marks`.
2. **A method that forgets to return** — a method that only calculates but never returns gives `None` to the caller, which then breaks the next step. Fix: check every method ends with a `return` when the caller needs its value.

Practice

1. Copy Example 1 and add a method `highest()` that returns the student's highest mark, or 0 if there are no marks. Test it with Ali's marks (answer: 90).
2. Create three student objects of your own, with your own names and marks. Print each student's name, average and Pass/Fail result using the `result()` method.
3. Extend the mini project: add a method `grade()` that returns A for 90+, B for 75+, C for 50+, and F below 50. Update the report lines to include the grade, and re-run it.
4. Save your extended report to a file, then read it back and print it. Write down which lesson each part of the project came from.
5. Write a `BankAccount` class with `deposit(amount)`, `withdraw(amount)` (refusing withdrawals bigger than the balance), and `show_balance()`. Test it with deposits and a too-big withdrawal.
6. Write a `Car` class with brand, model and year, plus a `__str__` method that returns something like `"Toyota Corolla (2020)"`. Create two cars and print both.
7. Extend the Student class from Example 1 with a `__str__` method returning `"Ali: average 85.0"`, and a method `add_marks(marks_list)` that adds several marks at once with a loop.

Practice quiz

1. What is a class in Python?

- A school lesson
- A blueprint for making objects with shared data and actions
- A list of students
- A file type

Answer: A blueprint for making objects with shared data and actions

2. What does __init__ do?

- Deletes an object
- Sets up a new object with its starting values
- Prints the object
- Imports a module

Answer: Sets up a new object with its starting values

3. What does self mean inside a method?

- The class itself
- The particular object the method was called on
- The program
- A variable from outside

Answer: The particular object the method was called on

Watch the video lesson: https://youtu.be/EGucoV9QAts

Read online: https://scopetreasure.com/notes/py-oop-mini-project · More free tools and notes at https://scopetreasure.com