PYTHON SELF-LEARNING BOOKLET

Session 6: Classes, Part 1

Welcome to Part 6 of 8! In this session, you will learn how to model real-world things in Python using **classes and objects**. This is your introduction to Object-Oriented Programming (OOP) – one of the most powerful ideas in programming.

📚 Session	🎯 Topic	🧩 Series
Part 6 of 8	Classes & Objects	Python Beginner Track

LEARNING GOALS

What You Will Learn

After finishing this session, you should be able to do all of the following:

1

Class vs. Object

Explain the difference between a **class** (blueprint) and an **object** (instance).

2

Define a Class

Use `class` and `__init__` to define and initialize objects.

3

Use `self`

Correctly apply `self` inside class methods to refer to the current object.

4

Dot Notation

Access attributes and call methods using dot notation on multiple instances.

5

Safe Updates

Add default attribute values and write methods that update attributes with validation.

CONCEPT 1

Why Classes Exist: Modeling the Real World

Programming is not only about writing steps – it is also about **building models** of real things. Consider a student: they have *data* (name, major, credits) and *actions* (introduce themselves, add credits). You could use a dictionary to store data, but a dictionary does not carry its own behavior.

Dictionary Approach (data only)

```
student = {"name": "Ali",
           "major": "Software Engineering",
           "credits": 30}
print(student["name"],
      student["major"])
```

What this code does

- Stores student information in a dictionary.
- Prints values by looking them up with keys.
- The dictionary itself has **no behavior** – no `introduce()` or `add_credits()`.

Classes bundle **data + behavior** together into one structure. That is the key advantage.

CONCEPT 2

OOP Vocabulary: Class, Object, Attributes, Methods

Object-Oriented Programming uses four key ideas. Learn these terms – you will use them throughout this session and beyond.

Class

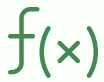
A **blueprint** that defines what all objects of that type know and do. Example: the concept of a "Dog".

Object / Instance

One **specific thing** created from the class. Example: *your* dog named Willie.

Attributes

Data stored inside an object. Example: `name`, `age`.



Methods

Functions inside a class – actions the object can perform. Example: `sit()`, `roll_over()`.

> **Easy analogy:** A class is like a *recipe*. An instance is the *actual cake* you bake from that recipe. Every cake follows the same recipe, but each one is its own separate object.

CONCEPT 3

Abstraction: Choose Only What Matters

A class is a **simplified** model of a real thing. You do not include every detail – only the details that matter for your program. This idea is called **abstraction**.

✅ Include (relevant details)

- `speed` – how fast the car goes
- `fuel` – how much fuel is left
- `drive()` – the action of driving

❌ Exclude (unnecessary details)

- Color of the seatbelt stitching
- Serial number of the air filter
- Temperature of the cup holder

Focus on what your program *needs*. Abstraction keeps classes clean, readable, and purposeful.

CONCEPT 4

Defining a Class: `class`, `__init__`, and `self`

A class starts with the `class` keyword. The special method `__init__()` runs automatically whenever you create a new object. It sets up the object's starting attributes.

Example: Defining a Dog class

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

Breaking it down

- `class Dog:` – defines a new class named `Dog`.
- `__init__` – runs automatically when you create a new `Dog(...)`.
- `self` – refers to the current object being created.
- `self.name`, `self.age` – attributes stored inside each dog object.

Remember: `self` must always be the first parameter of every method in a class. Python passes it automatically – you never include it when calling the method.

CONCEPT 5

Creating Objects (Instances)

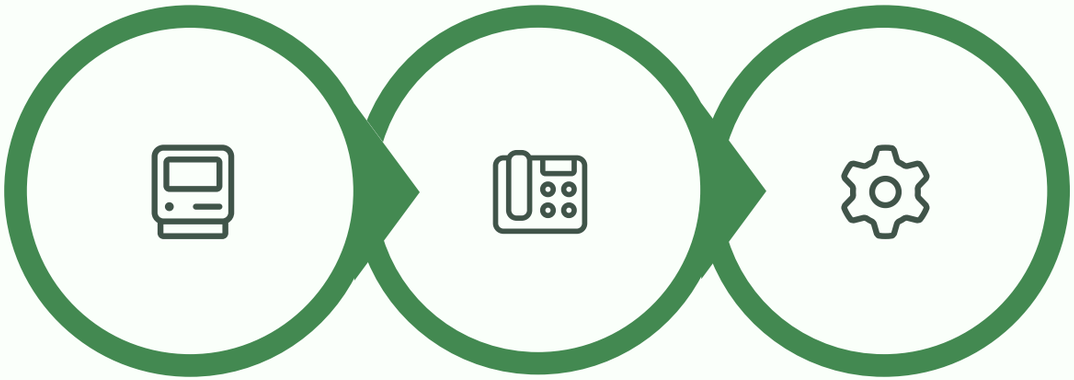
To create an object, you "call" the class like a function – just pass in the required arguments. Python automatically runs `__init__` behind the scenes.

Creating a Dog instance

```
my_dog = Dog("Willie", 6)
```

What this code does

- Creates one `Dog` instance.
- Automatically runs `Dog.__init__` with `"Willie"` and `6`.
- Stores the created object in the variable `my_dog`.



Write Class **Call Class** **Run __init__**

The class is the blueprint; calling it produces a brand-new, independent object in memory.

CONCEPT 6

Accessing Attributes with Dot Notation

Once you have an object, use **dot notation** to read the data stored inside it. The pattern is: `object_name.attribute_name`.

Reading attributes

```
print(my_dog.name)  # Willie
print(my_dog.age)   # 6
```

What this code does

- Uses dot notation to reach *inside* the object `my_dog`.
- Reads and prints the values that were stored when the object was created.

Think of the dot as a possessive: `my_dog.name` means "the `name` belonging to `my_dog`."

CONCEPTS 7 & 8

Adding and Calling Methods

A **method** is a function defined inside a class. Every method takes `self` as its first parameter so it can access the object's own attributes. You call methods with dot notation too.

Defining methods

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

    def sit(self):
        print(f"{self.name} is now sitting.")

    def roll_over(self):
        print(f"{self.name} rolled over!")
```

Calling methods

```
my_dog = Dog("Willie", 6)
my_dog.sit()
# Willie is now sitting.
my_dog.roll_over()
# Willie rolled over!
```

What this code does

- `sit()` and `roll_over()` use `self.name` to personalize output.
- Methods are called with parentheses: `my_dog.sit()`.

CONCEPT 9

Multiple Instances: Many Objects, One Blueprint

Once a class is defined, you can create as many instances as you need. Each instance stores its **own data**, but they all share the same methods defined in the class.

```
my_dog = Dog("Willie", 6)
your_dog = Dog("Lucy", 3)

my_dog.sit()        # Willie is now sitting.
your_dog.roll_over() # Lucy rolled over!
```

my_dog

`name = "Willie"`

`age = 6`

your_dog

`name = "Lucy"`

`age = 3`

Shared Methods

`sit()`

`roll_over()`

Changing `my_dog.name` will **not** affect `your_dog.name`. Each object lives independently in memory.

CONCEPT 10

Default Attributes: Starting Values

Sometimes every new object should begin with the same default value. You can set this directly inside `__init__` – no need to pass it as an argument.

Setting a default value

```
class Student:
    def __init__(self, name, major):
        self.name = name
        self.major = major
        self.credits = 0  # default value
```

What this code does

- Every new `Student` starts with `credits = 0` automatically.
- You only need to pass `name` and `major` when creating a student.
- The default can be changed later using methods.

CONCEPT 11

A Method That Shows Information

You can write a method that uses multiple attributes together to display a summary of the object. This is a very common and useful pattern.

Adding `show_info()`

```
class Student:
    def __init__(self, name, major):
        self.name = name
        self.major = major
        self.credits = 0

    def show_info(self):
        print(f"{self.name} studies "
              f"{self.major} and has "
              f"{self.credits} credits.")
```

Example output

```
s1 = Student("Lina", "Math")
s1.show_info()
# Lina studies Math
# and has 0 credits.
```

What this code does

- `show_info()` accesses all three attributes via `self`.
- Printing a summary from inside the object keeps your main code clean.

CONCEPT 12

Modifying Attributes Directly

You can change an attribute from outside the class using dot notation. This is quick, but it comes with a risk – there are no checks on what value you set.

Direct modification

```
student1 = Student("James", "Math")
student1.credits = 20
student1.show_info()
# James studies Math
# and has 20 credits.
```

The risk

- This *works*, but there are no guards.
- Nothing stops you from writing `student1.credits = -999`.
- You could accidentally set credits to a string or `None`.

Direct attribute changes are fine in small scripts, but in larger programs, always prefer **methods with validation**.

CONCEPT 13

Updating Attributes Safely with a Method

A better approach is to use a method that **validates** the input before applying the change. This keeps your object's data reliable.

Adding `update_credits()`

```
def update_credits(self, new_value):
    if new_value >= 0:
        self.credits = new_value
    else:
        print("Credits cannot be negative!")
```

Testing it

```
student1 = Student("James", "Math")

student1.update_credits(30)
student1.show_info()
# James studies Math
# and has 30 credits.

student1.update_credits(-10)
# Credits cannot be negative!
```

The method acts as a **gatekeeper** – only valid values get through. This is one of the key advantages of combining data and behavior in a class.

CONCEPT 14

Incrementing Values: Adding Gradually

Instead of replacing a value entirely, you often want to **increase it step by step**. Use `+=` inside a method to add to the existing value.

Adding `add_credits()`

```
def add_credits(self, amount):
    if amount > 0:
        self.credits += amount
    else:
        print("Amount must be positive!")
```

Example use

```
student1 = Student("James", "Math")
student1.add_credits(5)
print(student1.credits)  # 5
student1.add_credits(10)
print(student1.credits)  # 15
```

What `+=` means

`self.credits += amount` is the same as writing `self.credits = self.credits + amount`. It adds to the existing value rather than replacing it.

CONCEPT 15

Allow Changes, But Never Go Below Zero

Sometimes you want to allow both positive *and* negative adjustments (for corrections), while still preventing the total from going below zero.

The `change_credits()` method

```
def change_credits(self, amount):
    self.credits += amount
    if self.credits < 0:
        self.credits = 0
```

Goal behavior

- `+10` → credits go up ✅
- `-3` → credits go down ✅
- Result goes below o → reset to `0` ✅

```
s2.change_credits(15)  # credits = 15
s2.change_credits(-50) # would be -35
                       # → reset to 0
```

STEP-BY-STEP EXAMPLE

Putting It All Together

This complete example combines all the key ideas from this session: defining a class, default attributes, multiple methods, multiple instances, and safe updates.

```
class Student:
    def __init__(self, name, major):
        self.name = name
        self.major = major
        self.credits = 0  # default

    def introduce(self):
        print(f"Hi, I'm {self.name}, and I study {self.major}.")

def show_info(self):
    print(f"{self.name} studies {self.major} and has {self.credits} credits.")

def update_credits(self, new_value):
    if new_value >= 0:
        self.credits = new_value
    else:
        print("Credits cannot be negative!")

def change_credits(self, amount):
    self.credits += amount
    if self.credits < 0:
        self.credits = 0
```

STEP-BY-STEP EXAMPLE

Running the Full Example

```
# Create multiple students (multiple instances)
s1 = Student("Lina", "Software Engineering")
s2 = Student("Max", "Business")

# Use methods
s1.introduce()   # Hi, I'm Lina, and I study Software Engineering.
s2.introduce()   # Hi, I'm Max, and I study Business.

# Show default credits
s1.show_info()   # Lina studies Software Engineering and has 0 credits.
s2.show_info()   # Max studies Business and has 0 credits.

# Safe updates
s1.update_credits(30)
s1.show_info()   # Lina studies Software Engineering and has 30 credits.

s1.update_credits(-10)  # Credits cannot be negative!
s1.show_info()   # Still 30 credits.

# Allow positive/negative changes, but never below 0
s2.change_credits(15)
s2.show_info()   # Max studies Business and has 15 credits.

s2.change_credits(-50)  # Would go below 0 → reset to 0
s2.show_info()   # Max studies Business and has 0 credits.
```

STEP-BY-STEP EXAMPLE

Walkthrough: What Each Step Does

01

`class Student`

Defines the blueprint for all student objects.

02

`__init__`

Stores `name` and `major`, and sets `credits` to a default of `0`.

03

`introduce()` & `show_info()`

Display information using the object's own attributes via `self`.

04

`update_credits()`

Only accepts non-negative values – rejects bad input with a message.

05

`change_credits()`

Allows adding or subtracting, but corrects the total if it drops below `0`.

06

Two instances: `s1` & `s2`

They share the same methods but keep their own separate data.

COMMON MISTAKES

5 Mistakes to Avoid

These are the most common errors beginners make when writing classes. Study them carefully – they are easy to prevent once you know them.

❌ Forgetting `self` in method definitions

Wrong: `def show_info():`

Right: `def show_info(self):`

❌ Forgetting `self.` when storing attributes

Wrong: `name = name` (creates a local variable, not an attribute)

Right: `self.name = name`

❌ Calling a method without parentheses

Wrong: `s1.show_info` (refers to the method object, doesn't run it)

Right: `s1.show_info()`

❌ Accessing instance attributes from the class name

Wrong: `Student.name`

Right: `s1.name` (access from an instance)

❌ Changing attributes directly without validation

Risk: `s1.credits = -999` – no checks!

Better: use `update_credits()` which validates the input.

PRACTICE

Practice Tasks

Work through these tasks in order – each one builds on the previous. Try to write and run the code yourself before checking any solutions.

1

Mini Design Task

Pick something familiar (phone, pet, course). Write down 2 - 3 **attributes** and 2 - 3 **methods** it should have. Paper or notes are fine.

2

Basic `Student` Class

Create a class with attributes `name` and `major`. Create one student and print both attributes using dot notation.

3

Add `introduce()`

Make `introduce()` print: *"Hi, I'm <name>, and I study <major>."* Create 3 students and call `introduce()` for each.

4

Default Credits + Info Method

Add `credits = 0` in `__init__`. Add `show_info()` that prints name, major, and credits.

5

Add Safe Updating

Add `update_credits(new_value)` that rejects negative values. Test it with both `10` and `-5`.

EXPLORATION QUESTIONS

Think Deeper: Exploration Questions

These questions have no single right answer – they are designed to help you **experiment and reason** about the code. Try them after completing the practice tasks.

1 Independent Instances

Create two students and change `credits` for only one of them. Does the other change too? Why or why not? (Hint: think about what each variable stores.)

2 Improve `change_credits()`

Modify `change_credits()` so that it prints a message like *"Credits reset to 0."* when it has to prevent a negative total.

3 What About Strings?

In `update_credits()`, what happens if someone passes a string like `"ten"`? What error do you get? How might you prevent it using basic Python you already know?

SUMMARY

Session 6 Summary

You have covered the essential foundations of Python classes. Here is everything you learned in one place:

Classes & Objects

A **class** is a blueprint. An **object (instance)** is a specific example created from that blueprint.

`__init__` & `self`

`__init__()` sets up new objects. `self` always refers to the current object.

Dot Notation

Use `obj.attribute` to read data and `obj.method()` to call actions.

Default Attributes

Set a value directly in `__init__` to give every new object a consistent starting state.

Safe Updates

Updating attributes through methods with validation is safer than changing them directly.

Up next: Session 7 – Classes, Part 2 will cover inheritance, allowing one class to build on another.