Thinking in Objects: Classes, __init__ & Instance Attributes

AiCE Software Engineering Foundations — Python Programming (Advanced)

Every program you have written so far has organized code as functions and variables. This lesson introduces a fundamentally more powerful way of thinking — bundling related data and behavior together into a single unit called a class. Read slowly here; this is the most important shift in this entire program.

1. The Two Ways of Thinking About Code

When you write a function to calculate a student's grade and another to display their profile, you are thinking procedurally — the program is a sequence of steps and functions that operate on separate data. This approach works well for small programs.

As programs grow larger and more complex, procedural code becomes harder to organize. Functions and the data they work on become spread across the file, loosely connected. Adding a new type of thing — say, a teacher or a course, in addition to a student — requires more functions, more variables, and more careful coordination.

Object-Oriented Programming solves this by bundling the data and the functions that work on that data together into a single unit — a class. Instead of a student being a scattered collection of variables (name, age, scores) acted on by separate functions (get_grade, get_average), a student becomes a single object that knows its own data and can perform its own calculations.

Think of it this way: in the real world, a student is not a dictionary in a list. A student is a person — with a name, an age, scores, and the ability to calculate their own average and describe themselves. Object-Oriented Programming lets you model that reality directly in code.

You Cannot Unsee It

"OBject-Oriented Programming is not a feature of Python — it is a way of thinking about problems. Once you see the world as objects interacting with each other, you cannot unsee it. And your code becomes dramatically more organized as a result."

— Adesina Olagunju, Founder, AiCE

2. What Is a Class?

A class is a blueprint — a template that defines what a particular type of object looks like and what it can do. The class itself is not an object. It is the definition from which objects are created.

Think of a class as an architectural blueprint for a house. The blueprint describes how many rooms there are, where the doors go, and what electrical fittings are included. But the blueprint is not a house. You use the blueprint to build actual houses — each one a separate, real thing based on the same design.

A class defines the structure — what data each object of this type will have, and what actions each object can perform.

An object (or instance) is a specific thing created from that class — with its own actual data.

For example: Student is a class. Amara Diallo, aged 22, with a Math score of 88 — that is a specific student object created from the Student class.

3. Defining a Class

Classes are defined using the class keyword, followed by the class name, and a colon. By convention, class names use PascalCase — every word starts with a capital letter, with no underscores:

```
class Student:
    pass    # pass is a placeholder — the class body will go
here
```

This creates a valid, though empty, class called Student:

```
class Student:
    pass

# Creating objects from the class
student1 = Student()
student2 = Student()

print(student1)
print(student2)
print(type(student1))
```

This produces (the memory addresses will differ on your machine):

```
<__main__.Student object at 0x7f3a1b2c4d50>
<__main__.Student object at 0x7f3a1b2c4e10>
<class '__main__.Student'>
```

Two distinct Student objects were created — both instances of the Student class. Right now they hold no data. The __init__ method fixes that.

4. The __init__ Method — Initializing Objects

The __init__ method is a special method that Python calls automatically whenever a new object is created from a class. It is the constructor — the function that sets up the initial state of each object by assigning its starting data.

__init__ is always written with double underscores on each side — this is called a dunder method (short for double underscore). Python has several dunder methods with special meanings; __init__ is the most important.

The first parameter of __init__ is always self. self refers to the specific object being created — it is how the method knows which object's data to set up.

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

```
# Create two student objects
student1 = Student("Amara Diallo", 22)
student2 = Student("Kofi Mensah", 19)

print(student1.name)
print(student1.age)
print(student2.name)
print(student2.age)
```

This produces:

```
Amara Diallo
22
Kofi Mensah
19
```

When you write Student("Amara Diallo", 22), Python:

1. Creates a new empty Student object.

2. Calls __init__ automatically, passing the new object as self, "Amara Diallo" as name, and 22 as age.

3. Inside __init__, self.name = name stores "Amara Diallo" as the name attribute of this specific object; self.age = age stores 22 as the age attribute.

4. The object — now with its data — is assigned to student1.

student1.name and student2.name are completely separate. Changing one does not affect the other — each object has its own copy of the data.

> **self Is Not a Keyword**
> *self is a convention — it could be named anything, though you must never change it in practice. Python does not enforce the name self, but every Python developer in the world uses it. Using a different name would confuse anyone who reads your code. Always use self.*

5. Instance Attributes

The variables created inside __init__ using self.variable_name are called instance attributes. Each object created from the class gets its own set of these attributes with its own values.

You access an object's attributes using dot notation: object_name.attribute_name:

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

amara = Student("Amara Diallo", 22, "Senegal")
kofi = Student("Kofi Mensah", 19, "Ghana")

# Access attributes
print(f"{amara.name} is {amara.age} years old from
{amara.country}.")
print(f"{kofi.name} is {kofi.age} years old from
{kofi.country}.")

# Attributes can be updated after creation
amara.age = 23
print(f"Updated age: {amara.age}")

# New attributes can be added after creation (but avoid this
pattern)
amara.email = "amara@aice.africa"
print(f"Email: {amara.email}")
```

Each attribute lives inside its own object. amara.name and kofi.name are completely independent values stored in separate objects.

Key Takeaways

A class is a blueprint; an object is a specific instance created from it, with its own real data. Classes are defined with class and PascalCase names, and __init__ — Python's constructor dunder method — runs automatically whenever an object is created, using

self to set up that specific object's instance attributes. Every object's instance attributes are entirely independent of every other object's. The next lesson adds behavior to these objects — methods that let each object perform its own calculations — along with __str__ for controlling how an object displays, and the distinction between class attributes and instance attributes.