Classes and Objects in Python

Introduction

Object Oriented Programming becomes much easier when you understand two basic building blocks:

- **Class**
- **Object**

A class is a **blueprint** or **template**.

An object is a **real instance created from that blueprint**.

For example, imagine a school system.

You may have many students:

- Hafsa
- Asim
- Ali
- Sara

All students have similar information:

- name
- age
- grade

And they can perform similar actions:

- study
- attend class

- submit assignment

Instead of writing separate code structures for every student, we can create one `Student` class and then create multiple student objects from it.

```
class Student:
    pass
```

Then:

```
student1 = Student()
student2 = Student()
student3 = Student()
```

Here:

- `Student` is the **class**
- `student1`, `student2`, and `student3` are **objects**

---

1. What Is a Class?

A class defines what an object should contain and what it should be able to do.

Think of a class as a blueprint.

For example:

```
class Car:
    pass
```

This creates a class called `Car`.

At this point, we have defined the blueprint, but we have not created an actual car object.

We can create objects from it:

```
car1 = Car()
car2 = Car()
```

Now there are two different objects.

```
Car Class
   
   car1
   car2
```

Both objects come from the same class.

---

2. What Is an Object?

An object is an actual instance of a class.

```
class Car:
   pass

car1 = Car()
```

Here:

```
Car     class
car1    object
Car()   creates an object
```

The parentheses after the class name create an instance.

`Car()`

means:

> Create an object based on the `Car` class.

---

3. Class vs Object

This distinction is fundamental.

Class	**Object**
Blueprint	Actual instance
Defines structure	Contains actual data
General	Specific
Created using `class`	Created by calling the class
Example: `Student`	Example: `student1`

Real-world example:

Class → Student

Objects:
• Hafsa
• Asim
• Ali

The class describes what a student can have.

Each object represents one specific student.

---

4. Creating a Class

The basic syntax is:

```
class ClassName:
    # class body
```

Example:

```
class Student:
    pass
```

Python class names are normally written using **PascalCase**:

```
class StudentProfile:
    pass

class BankAccount:
```

```
    pass

class ShoppingCart:
    pass
```

---

5. Creating Objects

After defining a class, create objects by calling it:

```
class Student:
    pass

student1 = Student()
student2 = Student()
```

Now both objects are instances of `Student`.

You can verify this:

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

Output:

```
<class '__main__.Student'>
```

You can also use `isinstance()`:

```
print(isinstance(student1, Student))
```

Output:

`True`

This asks:

> Is `student1` an object of the `Student` class?

---

6. Giving Objects Data

Objects become useful when they contain information.

We can give an object attributes.

```
class Student:
    pass

student1 = Student()

student1.name = "Hafsa"
student1.age = 25
student1.grade = 9
```

Now the object contains:

```
student1
□□□ name □ "Hafsa"
□□□ age □ 25
□□□ grade □ 9
```

We can access these attributes:

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

Output:

```
Hafsa
25
9
```

---

7. Creating Multiple Objects

The biggest benefit is that one class can create many objects.

```
class Student:
    pass

student1 = Student()
student1.name = "Hafsa"
student1.age = 25

student2 = Student()
student2.name = "Asim"
student2.age = 27
```

Now:

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

Output:

```
Hafsa
Asim
```

Both objects have the same general structure, but they contain different data.

```
Student Class
    □
    □□□ student1
    □     □□□ name = Hafsa
    □     □□□ age = 25
    □
    □□□ student2
          □□□ name = Asim
          □□□ age = 27
```

---

8. Why Do We Need `__init__()`?

Setting every attribute manually can become repetitive.

For example:

```
student1 = Student()
student1.name = "Hafsa"
student1.age = 25
student1.grade = 9
```

And again:

```
student2 = Student()
student2.name = "Asim"
student2.age = 27
student2.grade = 10
```

This becomes inconvenient when there are many objects.

We can initialize objects automatically using `__init__()`.

```
class Student:

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

Now create objects like this:

```
student1 = Student("Hafsa", 25, 9)
student2 = Student("Asim", 27, 10)
```

Python automatically calls `__init__()` when each object is created.

---

9. Understanding `self`

`self` refers to the **current object**.

Consider:

```
class Student:

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

When we write:

`student1 = Student("Hafsa", 25)`

Python creates the object and `self` refers to `student1`.

So conceptually:

`self → student1`

When we create:

`student2 = Student("Asim", 27)`

`self` refers to `student2`.

`self → student2`

This is how Python knows which object's data should be stored.

---

10. `self.name` vs `name`

This is one of the most important things to understand.

```
class Student:

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

There are two different things here:

`name`
→

value received by the method

self.name

attribute stored inside the object

When we write:

```
student1 = Student("Hafsa")
```

Python effectively stores:

```
student1.name = "Hafsa"
```

So:

```
print(student1.name)
```

gives:

```
Hafsa
```

---

11. Objects Can Have Different Data

Consider:

```
class Student:

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

Create two objects:

```
student1 = Student("Hafsa", 25)
student2 = Student("Asim", 27)
```

Their data is independent:

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

Output:

```
Hafsa
Asim
```

Changing one does not automatically change the other.

```
student1.age = 26

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

Output:

```
26
27
```

Each object has its own instance data.

---

12. Objects Can Have Behavior

Objects don't only store data.

They can also perform actions.

These actions are called **methods**.

Example:

```
class Student:

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

    def introduce(self):
        print(f"My name is {self.name}.")
```

Create an object:

```
student = Student("Hafsa")
```

Call its method:

```
student.introduce()
```

Output:

```
My name is Hafsa.
```

Now the object contains both:

```
Data
```

name

Behavior
□
introduce()

---

13. Data + Behavior

This is the central idea behind objects.

Instead of keeping data and functions completely separate:

```
name = "Hafsa"

def introduce(name):
    print(f"My name is {name}.")
```

OOP allows us to group related data and behavior:

```
class Student:

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

    def introduce(self):
        print(f"My name is {self.name}.")
```

Now:

```
student = Student("Hafsa")

student.introduce()
```

The student object knows:

- its own data
- the operations that make sense for that data

---

14. Instance Attributes

Attributes stored separately for each object are called **instance attributes**.

Example:

```
class User:

    def __init__(self, username, email):
        self.username = username
        self.email = email
```

Create objects:

```
user1 = User("hafsa", "hafsa@example.com")
user2 = User("asim", "asim@example.com")
```

Each object has its own:

```
username
email
```

So:

```
print(user1.username)
print(user2.username)
```

Output:

```
hafsa
asim
```

---

15. Instance Methods

A method that works with a particular object is called an **instance method**.

Example:

```
class BankAccount:

    def __init__(self, owner, balance):
        self.owner = owner
        self.balance = balance

    def deposit(self, amount):
        self.balance += amount
```

Create an account:

```
account = BankAccount("Hafsa", 5000)
```

Deposit money:

```
account.deposit(2000)
```

Check balance:

```
print(account.balance)
```

Output:

```
7000
```

The method modifies the data belonging to that specific object.

---

16. Why `self` Is Important in Methods

Consider:

```
class BankAccount:

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

    def deposit(self, amount):
        self.balance += amount
```

When we write:

```
account.deposit(1000)
```

the method operates on `account`.

Conceptually:

```
account
  
  balance
  
  deposit()
```

```
        □
        □□□ modifies account.balance
```

Without `self`, the method would not know which object's data it should work with.

---

17. A Complete Example

Let's build a simple `Product` class.

```
class Product:

    def __init__(self, name, price):
        self.name = name
        self.price = price

    def display(self):
        print(f"{self.name}: ${self.price}")

    def apply_discount(self, percentage):
        discount = self.price * percentage / 100
        self.price -= discount
```

Create objects:

```
product1 = Product("Keyboard", 50)
product2 = Product("Mouse", 30)
```

Display them:

```
product1.display()
product2.display()
```

Output:

```
Keyboard: $50
Mouse: $30
```

Apply a discount:

```
product1.apply_discount(10)
```

Now:

```
product1.display()
```

Output:

```
Keyboard: $45.0
```

Notice that the discount affected only `product1`.

---

18. Real World Example: haas.dev Resource

A resource platform can contain many resources.

Each resource may have:

- title
- description
- category
- file name
- view count

Instead of handling everything separately, we can model a resource as an object.

```
class Resource:

    def __init__(self, title, category, file_name):
        self.title = title
        self.category = category
        self.file_name = file_name

    def display(self):
        print(f"{self.title} | {self.category}")

    def open(self):
        print(f"Opening {self.file_name}")
```

Create resources:

```
resource1 = Resource(
    "Python Functions",
    "Python",
    "python-functions.pdf"
)

resource2 = Resource(
    "What Is OOP?",
    "Python",
    "what-is-oop.pdf"
)
```

Now:

```
resource1.display()
resource2.display()
```

Output:

```
Python Functions | Python
What Is OOP? | Python
```

And:

```
resource1.open()
```

Output:

```
Opening python-functions.pdf
```

The same `Resource` class can represent hundreds or thousands of resources.

---

19. Class Attributes vs Instance Attributes

Python classes can also have attributes shared by the class.

Example:

```
class Student:

    school = "haas.dev Academy"

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

Here:

```
school
```

is a **class attribute**.

While:

`self.name`

is an **instance attribute**.

Create objects:

```
student1 = Student("Hafsa")
student2 = Student("Asim")
```

Both can access:

```
print(student1.school)
print(student2.school)
```

Output:

```
haas.dev Academy
haas.dev Academy
```

But their names are different:

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

Output:

```
Hafsa
Asim
```

Think:

```
Class Attribute
    
Shared concept

Instance Attribute
    
Object-specific data
```

---

20. Checking Object Identity

Two objects can come from the same class without being the same object.

```
class User:
    pass

user1 = User()
user2 = User()
```

Check:

```
print(user1 is user2)
```

Output:

```
False
```

They are two separate objects.

Even though:

```
type(user1) == type(user2)
```

is:

```
True
```

This distinction matters when working with objects and memory.

---

21. Objects Can Be Stored in Lists

Objects work naturally with Python data structures.

```
class Student:

    def __init__(self, name, grade):
        self.name = name
        self.grade = grade
```

Create students:

```
students = [
    Student("Hafsa", 9),
    Student("Asim", 10),
    Student("Ali", 9)
]
```

Now we can loop through them:

```
for student in students:
    print(student.name)
```

Output:

```
Hafsa
Asim
Ali
```

This becomes extremely useful in real applications.

For example:

```
List
 
User objects
 
Product objects
 
Order objects
 
Resource objects
```

---

22. Objects Can Interact With Other Objects

Real applications often contain objects that work together.

For example:

```
class User:

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

```
class Course:

    def __init__(self, title):
        self.title = title
        self.students = []

    def add_student(self, student):
        self.students.append(student)
```

Create objects:

```
user1 = User("Hafsa")
course = Course("Python")
```

Add the user:

```
course.add_student(user1)
```

Now the `Course` object contains a `User` object.

This is an important OOP pattern:

> Objects can contain and work with other objects.

---

23. Why Classes and Objects Are Useful

Without classes, a large application can become difficult to organize.

Imagine an e-commerce application.

You might need:

```
Users
Products
Orders
Payments
Shopping carts
Reviews
Categories
```

Each entity has:

- data
- behavior
- relationships

Classes allow us to model those entities.

For example:

```
User
 □□□ name
 □□□ email
 □□□ login()

Product
 □□□ name
 □□□ price
 □□□ apply_discount()

Order
```

□□□ items
□□□ total
□□□ calculate_total()

This gives the application a clearer structure.

---

24. When Should You Create a Class?

A class can be useful when:

1. You have related data

name
email
age

2. You have related behavior

login()
logout()
update_profile()

3. You need multiple similar objects

User 1
User 2
User 3
...

4. You want to model a real concept

Examples:

`User`
`Product`
`Order`
`Invoice`
`Vehicle`
`Student`
`Course`
`Resource`

5. The program is becoming difficult to organize

Classes can help divide a large system into meaningful components.

---

25. When You Don't Need a Class

Not every piece of Python code needs OOP.

If you only need a small calculation:

```
def calculate_tax(price, rate):
    return price * rate
```

A class may add unnecessary complexity.

The goal is not:

> "Put everything inside classes."

The goal is:

>

> "Use classes when objects make the problem easier to model and organize."

---

26. Common Beginner Mistakes

Mistake 1: Forgetting `self`

Incorrect:

```
class Student:

    def __init__(name):
        name = name
```

Correct:

```
class Student:

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

Mistake 2: Using a class without creating an object

Defining:

```
class Student:
    pass
```

does not automatically create a student.

You need:

```
student = Student()
```

Mistake 3: Confusing the class with the object

`Student`

is the class.

`student`

is an object.

Mistake 4: Forgetting parentheses when creating an object

Incorrect:

```
student = Student
```

This assigns the class itself.

Normally:

```
student = Student()
```

creates an object.

Mistake 5: Forgetting that each object has its own instance data

```
student1.name = "Hafsa"
student2.name = "Asim"
```

These are separate attributes belonging to separate objects.

Mistake 6: Making everything a class

Classes are useful, but they are not automatically better than functions.

Choose the structure that makes the problem easier to understand.

---

27. Mental Model

When you see a problem, ask:

```
What things exist?
    
What information does each thing have?
    
What can each thing do?
    
Do I have many similar things?
    
Would a class organize them better?
```

For example, an online store:

Thing → Product

Information:
• name
• price
• stock

Actions:
• apply_discount()
• increase_stock()
• decrease_stock()

That naturally suggests:

```
class Product:
    ...
```

---

28. A Practical Workflow

When designing a class:

Step 1: Identify the entity

Example:

```
Product
```

Step 2: Identify its data

```
name
price
stock
```

Step 3: Identify its behavior

```
apply_discount()
sell()
restock()
```

Step 4: Create the class

```
class Product:
    ...
```

Step 5: Initialize the data

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

Step 6: Add methods

```
def sell(self):
    ...
```

Step 7: Create objects

```
keyboard = Product(...)
mouse = Product(...)
```

Step 8: Test object behavior

```
keyboard.sell()
```

This workflow turns real-world concepts into code.

---

29. Mini Project: Simple Task Manager

Build a small task manager using classes and objects.

Requirements:

Each task should have:

- title
- completed status

Each task should be able to:

- display itself
- mark itself complete

Start with:

```
class Task:

    def __init__(self, title):
        self.title = title
        self.completed = False

    def complete(self):
        self.completed = True

    def display(self):
        status = "Done" if self.completed else "Pending"
        print(f"{self.title} - {status}")
```

Create tasks:

```
task1 = Task("Learn Python")
task2 = Task("Build a project")
```

Display:

```
task1.display()
task2.display()
```

Complete a task:

```
task1.complete()
```

Display again:

```
task1.display()
```

Expected output:

```
Learn Python - Done
```

Extend the project by adding:

- task IDs
- priority
- due dates
- delete functionality
- a list containing multiple tasks
- a method to display all tasks

---

30. 5 Minute Challenge

Create a `Book` class.

Each book should have:

`title`
`author`
`pages`

Add a method:

`display_info()`

Create at least two book objects.

For example:

```
book1 = Book("Python Basics", "Hafsa", 250)
book2 = Book("Clean Code", "Asim", 400)
```

Then display their information.

Challenge: Add a `read()` method that keeps track of how many pages have been read.

---

31. Exercises

Exercise 1

Create a `Car` class with:

`brand`
`model`
`year`

Create two objects.

Exercise 2

Create a `Rectangle` class with:

`width`
`height`

Add a method:

`area()`

that returns:

`width × height`

Exercise 3

Create a `BankAccount` class with:

`owner`
`balance`

Add:

`deposit()`
`withdraw()`

Exercise 4

Create a `Student` class with:

`name`
`marks`

Add a method that calculates the average marks.

Exercise 5

Create a `Product` class with:

`name`
`price`
`stock`

Add methods to:

`sell()`
`restock()`

---

32. Mini Quiz

Question 1

What is a class?

A. A Python loop

B. A blueprint for creating objects
C. A list
D. A string

Answer: B

Question 2

What is an object?

A. An instance of a class
B. A function
C. A module
D. A variable type

Answer: A

Question 3

What does `self` represent?

A. The class itself
B. The current object
C. The Python interpreter
D. A global variable

Answer: B

Question 4

What does `__init__()` commonly do?

A. Deletes an object
B. Initializes an object's data
C. Imports a module
D. Stops a program

Answer: B

Question 5

What is the difference between these?

`Student`

and:

`Student()`

Answer:

`Student` refers to the class.

`Student()` creates an instance of that class.

---

33. Interview Questions

Beginner

1. What is a class in Python?
2. What is an object?
3. What is the difference between a class and an object?
4. How do you create an object?
5. What is `self`?
6. What does `__init__()` do?
7. What are instance attributes?
8. What are instance methods?
9. What is a class attribute?
10. Can one class create multiple objects?

Intermediate

11. Why is `self` required in instance methods?
12. What happens when `ClassName()` is called?
13. How are instance attributes different between objects?
14. What is the difference between class and instance attributes?
15. Can an object contain another object?
16. Can objects be stored in lists or dictionaries?
17. When should you use a class instead of a function?
18. Why shouldn't every piece of code automatically be converted into a class?

---

34. Cheat Sheet

```
# Define a class
class Student:
    pass
```

```
# Create an object
student = Student()

# Initialize an object
class Student:

    def __init__(self, name, age):
        self.name = name
        self.age = age

# Create object with data
student = Student("Hafsa", 25)

# Access attributes
print(student.name)
print(student.age)

# Add a method
class Student:

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

    def introduce(self):
        print(self.name)

# Call a method
student.introduce()

# Check object type
print(type(student))

# Check whether an object belongs to a class
print(isinstance(student, Student))
```

---

35. Key Takeaways

- A **class** is a blueprint for objects.
- An **object** is an instance of a class.
- One class can create many objects.
- Objects can contain **attributes**.
- Objects can perform actions through **methods**.
- `self` refers to the current object.
- `__init__()` initializes object data when an object is created.
- Instance attributes belong to individual objects.
- Class attributes can be shared by objects.
- Objects can be stored inside lists, dictionaries, and other data structures.
- Objects can interact with other objects.
- Classes are useful for organizing related data and behavior.
- Not every Python problem requires a class.
- Good OOP starts by identifying **entities, their data, and their behavior**.

The fundamental model is:

```
CLASS
  
Blueprint
  
OBJECT
  
Data + Behavior
```

Once this becomes clear, the next step is learning how Python classes are designed in more detail: initialization, attributes, methods, class methods, static methods, encapsulation, inheritance, polymorphism, abstraction, and special methods.

Website: https://dev-roast-app.vercel.app

Visit haas.dev for more resources and guides.