What Is OOP? Object Oriented Programming in Python

Introduction

As Python programs become larger, writing everything as separate variables and functions can become difficult to manage.

For example, imagine building an application for haas.dev.

You might have:

```
user_name = "Hafsa"
user_email = "hafsa@example.com"
user_role = "developer"

def login():
    ...

def logout():
    ...

def update_profile():
    ...
```

This works for a small program.

But what happens when you have:

- 100 users
- Different user types
- Different data for every user
- Functions that work with specific users
- Projects with hundreds of files

Managing everything separately becomes harder.

Object Oriented Programming, or OOP, provides a way to organize related data and behavior together.

The core idea is:

> **An object combines data and the operations that work with that data.**

---

1. What Does OOP Mean?

OOP stands for:

Object Oriented Programming

It is a programming approach where we design programs around **objects**.

An object can represent something from the real world or something from the application.

Examples:

```
User
Product
Car
BankAccount
Student
Course
Resource
Order
```

`Payment`

Each object can have:

Data

Information about the object.

For a `User`:

```
name
email
role
```

Behavior

Things the object can do.

```
login()
logout()
update_profile()
```

So we can think of an object as:

```
Object
□
□□□ Data
□
□□□ Behavior
```

---

2. A Real World Example

Think about a car.

A car has data:

```
brand
model
color
speed
```

A car also has behavior:

```
start()
stop()
accelerate()
brake()
```

Instead of managing these separately, OOP allows us to represent the car as an object.

```
Car
□
□□□ Data
□   □□□ brand
□   □□□ model
□   □□□ color
□   □□□ speed
□
□□□ Behavior
    □□□ start()
    □□□ stop()
    □□□ accelerate()
    □□□ brake()
```

---

3. What Is an Object?

An **object** is a specific instance of something.

For example:

`Car`

can describe the general idea of a car.

But:

`Toyota Corolla`

can be one specific car.

Another could be:

`Honda Civic`

Both are objects created from the same general design.

Think:

`Class`

`Blueprint`

`Object`

`Actual thing created from blueprint`

---

4. What Is a Class?

A **class** is a blueprint for creating objects.

For example:

```
class Car:
    pass
```

This creates a class called `Car`.

The class describes what a car object can be.

We can create objects from it:

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

Now:

```
car1 → Car object
car2 → Car object
```

They are separate objects created from the same class.

---

5. Class vs Object

This distinction is extremely important.

Class

A blueprint or template.

```
class Car:
    pass
```

Object

An actual instance created from that class.

```
car1 = Car()
```

Think about a house.

```
House blueprint
       □
   Class
       □
 □□□□□□□□□□□□□□□
 □           □
House 1     House 2
Object      Object
```

The blueprint is not the actual house.

Similarly, the class is not the object.

---

6. Creating a Simple Class

```
class Student:
```

```
    pass
```

Now create an object:

```
student1 = Student()
```

We can check its type:

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

You will see something similar to:

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

This tells us that `student1` is an object created from the `Student` class.

---

7. Adding Data to an Object

We can add attributes to an object.

```
class Student:
    pass

student1 = Student()

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

Now the object contains:

```
student1
```

□
□□□ name □ "Hafsa"
□□□ age □ 25

We can access them:

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

Output:

```
Hafsa
25
```

---

8. What Is an Attribute?

An **attribute** is data associated with an object.

Example:

```
student1.name
student1.age
```

Here:

```
name □ attribute
age  □ attribute
```

Other examples:

```
car.color
```

`user.email`
`product.price`
`account.balance`

A simple mental model:

> **Attributes describe an object.**

---

9. What Is a Method?

A **method** is a function that belongs to a class or object.

Example:

```
class Student:

    def introduce(self):
        print("I am a student")
```

Create an object:

```
student1 = Student()
```

Call the method:

```
student1.introduce()
```

Output:

```
I am a student
```

So:

```
Attribute → Data
Method    → Behavior
```

---

10. Functions vs Methods

A normal function:

```
def greet():
    print("Hello")
```

A method:

```
class Student:

    def greet(self):
        print("Hello")
```

The difference is that a method belongs to a class/object.

You call:

```
greet()
```

for the standalone function.

But:

`student.greet()`

for the method.

---

11. Understanding `self`

One of the first confusing things about OOP is:

`self`

Consider:

```
class Student:

    def introduce(self):
        print("I am a student")
```

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

If:

`student1 = Student()`

then:

`student1.introduce()`

means the method operates on `student1`.

---

12. Using `self` With Attributes

```
class Student:

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

Create the object:

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

student1.introduce()
```

Output:

```
Hafsa
```

Here:

```
self.name
```

means:

> The `name` attribute belonging to this particular object.

---

13. Multiple Objects

This is where OOP becomes useful.

```
class Student:

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

Create two objects:

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

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

Now:

```
student1.introduce()
student2.introduce()
```

Output:

```
My name is Hafsa
My name is Asim
```

The same method works with different objects.

That is because `self` refers to whichever object called the method.

---

14. The `__init__()` Method

Usually, we do not want to manually assign attributes after creating every object.

Instead, Python provides a special method:

`__init__()`

It runs automatically when an object is created.

Example:

```
class Student:

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

Now:

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

Python automatically calls:

```
__init__("Hafsa", 25)
```

and stores:

```
name  Hafsa
age   25
```

---

15. Constructor Concept

You will often hear `__init__()` called a **constructor** in beginner Python discussions.

More precisely, `__init__()` initializes an already-created object.

For practical learning, remember:

> **`__init__()` runs when you create an object and initializes its attributes.**

Example:

```
class User:

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

Create:

```
user = User("Hafsa", "hafsa@example.com")
```

Now:

```
print(user.name)
print(user.email)
```

---

16. Class With Attributes and Methods

A more complete example:

```
class Student:
```

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

    def introduce(self):
        print(f"My name is {self.name}")
        print(f"I am {self.age} years old")
```

Create an object:

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

Use it:

```
student.introduce()
```

Output:

```
My name is Hafsa
I am 25 years old
```

Now we have:

```
Student object
□
□□□ name
□□□ age
□□□ introduce()
```

---

17. Objects Can Have Different Data

The class defines the structure.

Each object stores its own values.

```
student1 = Student("Hafsa", 25)
student2 = Student("Asim", 28)
```

Now:

```
student1
name → Hafsa
age  → 25

student2
name → Asim
age  → 28
```

Both use the same class.

But their data is different.

---

18. Adding Behavior

Objects become more useful when they contain behavior.

Example:

```
class BankAccount:

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

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

    def show_balance(self):
        print(self.balance)
```

Create:

```
account = BankAccount("Hafsa", 1000)
```

Deposit:

```
account.deposit(500)
```

Check balance:

```
account.show_balance()
```

Output:

```
1500
```

The object manages both:

Data:
balance

Behavior:
deposit()
show_balance()

---

19. Why Is This Better?

Without OOP:

```
balance = 1000

def deposit(amount):
    global balance
    balance += amount
```

This becomes difficult when you have multiple accounts.

With OOP:

```
account1 = BankAccount("Hafsa", 1000)
account2 = BankAccount("Asim", 2000)
```

Now each object maintains its own state.

```
account1
balance  1000

account2
balance  2000
```

The same methods work with both.

---

20. Encapsulation

One important idea in OOP is **encapsulation**.

Encapsulation means keeping related data and behavior together and controlling how that data is accessed or changed.

For example:

```
class BankAccount:

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

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

Instead of allowing every part of the program to change the balance however it wants, the class provides a controlled operation:

```
account.deposit(500)
```

This helps protect the object's state.

---

21. The Four Common OOP Concepts

When learning OOP, you will often encounter four major concepts:

```
1. Encapsulation
2. Inheritance
3. Polymorphism
4. Abstraction
```

These concepts will be studied more deeply in later topics.

For now, understand the basic idea.

Encapsulation

Keep data and related behavior together.

Inheritance

Create a new class based on an existing class.

Polymorphism

Different objects can respond to the same operation in different ways.

Abstraction

Hide unnecessary implementation details and expose what users need.

---

22. Inheritance Preview

Suppose we have:

```
class Animal:

    def eat(self):
        print("Eating")
```

We can create:

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

Now:

```
dog = Dog()
dog.eat()
```

The `Dog` class gets behavior from `Animal`.

This is called **inheritance**.

We will study inheritance separately.

---

23. Polymorphism Preview

Different objects can provide the same method name but behave differently.

```
class Dog:

    def speak(self):
        print("Bark")


class Cat:

    def speak(self):
        print("Meow")
```

Now:

```
dog = Dog()
cat = Cat()

dog.speak()
cat.speak()
```

Same method name:

`speak()`

Different behavior:

`Dog → Bark`
`Cat → Meow`

This is a basic example of polymorphism.

---

24. Abstraction Preview

Suppose you use a car.

You know:

`start()`
`accelerate()`
`brake()`

You do not need to understand every internal engine operation to drive it.

Similarly, software can expose a simple interface while hiding complicated internal implementation.

That idea is called **abstraction**.

---

25. Class Attributes

Not every attribute has to belong to one specific object.

A class can also have a **class attribute**.

Example:

```
class Student:

    school = "Example School"
```

Now:

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

Both can access:

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

Output:

```
Example School
Example School
```

The value belongs to the class rather than being separately created for every object.

---

26. Instance Attributes vs Class Attributes

Instance attribute

Belongs to a specific object.

```
class Student:

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

Each student can have a different name.

Class attribute

Shared by the class unless overridden.

```
class Student:

    school = "Example School"
```

Think:

```
Instance attribute
→ each object has its own value

Class attribute
→ shared class-level value
```

---

27. OOP Is About Modeling

A powerful way to understand OOP is to think:

> **What things exist in my application, and what can those things do?**

For an e-commerce application:

User
Product
Cart
Order
Payment

For a learning platform:

Student
Course
Lesson
Quiz
Resource

For haas.dev:

Resource
Category
Quiz
User
Article

Then ask:

What data does each thing have?

What behavior does each thing have?

For example:

Resource

□
□□□ title
□□□ description
□□□ category
□□□ file_name
□
□□□ publish()
□□□ update()
□□□ archive()

This is object-oriented thinking.

---

28. OOP Does Not Mean Everything Must Be a Class

A common beginner mistake is thinking:

> "If I am using OOP, every piece of code must be inside a class."

Not true.

Python supports multiple programming styles.

You can use:

- Procedural programming
- Functional programming
- Object-oriented programming
- A combination of these approaches

Use OOP when objects and their relationships make the program easier to understand and maintain.

---

29. When Is OOP Useful?

OOP becomes particularly useful when:

1. The application is large

Many components need to work together.

2. You have many similar entities

For example:

```
100 users
500 products
1000 orders
```

3. Objects have both data and behavior

For example:

```
BankAccount
balance + deposit()
```

4. You need reusable structures

A class can define a structure that can create many objects.

5. You need clear organization

Related data and behavior can stay together.

---

30. When OOP May Be Unnecessary

For a tiny script:

```
name = "Hafsa"

print(f"Hello {name}")
```

Creating a class would add unnecessary complexity.

You do not need:

```
class Greeting:
    ...
```

for every simple task.

A good developer asks:

> **Will a class make this code easier to understand, reuse, or maintain?**

If not, a simpler approach may be better.

---

31. Procedural vs Object Oriented Thinking

Procedural style

Think primarily about:

What steps should the program perform?

Example:

```
def deposit(balance, amount):
    return balance + amount

balance = deposit(1000, 500)
```

Object oriented style

Think:

What object owns this behavior?

Example:

```
account.deposit(500)
```

The difference is not that one is always correct and the other is wrong.

They are different ways of organizing a program.

---

32. A Complete Beginner Example

```
class User:

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

    def introduce(self):
```

```
        print(f"Name: {self.name}")
        print(f"Email: {self.email}")

    def change_email(self, new_email):
        self.email = new_email
```

Create an object:

```
user = User(
    "Hafsa",
    "hafsa@example.com"
)
```

Display information:

```
user.introduce()
```

Change the email:

```
user.change_email("new@example.com")
```

Display again:

```
user.introduce()
```

The object contains both:

```
Data
□□□ name
□□□ email

Behavior
□□□ introduce()
```

□□□ change_email()

---

33. A Simple Mental Model

Remember this:

CLASS
□
Blueprint

OBJECT
□
Actual instance

ATTRIBUTE
□
Object's data

METHOD
□
Object's behavior

__init__()
□
Initializes object data

self
□
Current object

This mental model will make the next OOP topics much easier.

---

34. Real World Example: haas.dev

Suppose haas.dev has resources.

Instead of storing everything separately:

```
title = "Python Logging Basics"
description = "Learn Python logging"
category = "Python"
```

we could model a resource:

```
class Resource:

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

    def display(self):
        print(self.title)
        print(self.description)
        print(self.category)
```

Create resources:

```
resource1 = Resource(
    "Python Logging Basics",
    "Learn Python logging",
    "Python"
)

resource2 = Resource(
```

```
    "What Is OOP?",
    "Learn object oriented programming",
    "Python"
)
```

Now each resource is an object.

```
resource1
□□□ title
□□□ description
□□□ category
□□□ display()

resource2
□□□ title
□□□ description
□□□ category
□□□ display()
```

The same class defines the structure for both.

---

35. Common Beginner Mistakes

Mistake 1: Confusing a class with an object

```
class User:
    pass
```

`User` is the class.

```
user = User()
```

`user` is the object.

Mistake 2: Forgetting `self`

Incorrect:

```
class User:

    def greet():
        print("Hello")
```

Correct:

```
class User:

    def greet(self):
        print("Hello")
```

Mistake 3: Forgetting `self.`

Incorrect:

```
class User:

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

Correct:

```
class User:
```

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

The first `name` is the parameter.

`self.name` is the object's attribute.

Mistake 4: Creating classes for everything

Not every problem requires OOP.

Use the simplest design that solves the problem clearly.

Mistake 5: Making classes too complicated

A class should have a clear responsibility.

Avoid creating one huge class that manages unrelated things.

---

36. Five Minute Challenge

Create a `Book` class.

It should have:

`title`

`author`
`price`

Create a method:

`display_info()`

Example:

```
book = Book(
    "Python Basics",
    "Hafsa",
    1500
)
```

Calling:

`book.display_info()`

should display the book information.

Then create at least **two different book objects**.

---

37. Exercises

Exercise 1

Create a `Car` class with:

`brand`
`model`

`color`

Exercise 2

Create a `Student` class with:

`name`
`age`
`grade`

and a method:

`introduce()`

Exercise 3

Create a `BankAccount` class with:

`owner`
`balance`

and methods:

`deposit()`
`withdraw()`

Exercise 4

Create a `Product` class with:

`name`
`price`
`quantity`

and a method that calculates the total value:

`price × quantity`

Exercise 5

Create two objects from the same class and give them different values.

Observe how each object maintains its own data.

---

38. Mini Quiz

1. What does OOP stand for?

A. Object Oriented Programming
B. Object Organized Process
C. Operational Object Programming
D. Organized Output Program

Answer: A

---

2. What is a class?

A. A variable

B. A blueprint for creating objects
C. A loop
D. A module

Answer: B

---

3. What is an object?

A. An instance of a class
B. A Python keyword
C. A function parameter
D. A package

Answer: A

---

4. What does `self` refer to?

A. The class name
B. The current object
C. The Python interpreter
D. The parent class

Answer: B

---

5. What does `__init__()` do?

A. Deletes an object
B. Initializes an object's attributes
C. Imports a module
D. Creates a loop

Answer: B

---

6. What is an attribute?

A. Data associated with an object
B. A Python package
C. A loop
D. An exception

Answer: A

---

7. What is a method?

A. A variable
B. A function associated with a class or object
C. A module
D. A data type

Answer: B

---

39. Interview Questions

Beginner

1. What is OOP?

OOP is a programming approach that organizes software around objects containing data and behavior.

2. What is a class?

A class is a blueprint used to create objects.

3. What is an object?

An object is an instance of a class.

4. What is an attribute?

An attribute is data associated with an object.

5. What is a method?

A method is a function defined as part of a class.

Intermediate

6. What is `self` in Python?

`self` refers to the current object on which a method is operating.

7. What is `__init__()`?

It is a special method used to initialize an object's attributes when the object is created.

8. What is the difference between a class and an object?

A class is the blueprint; an object is a specific instance created from that blueprint.

9. What is encapsulation?

Encapsulation means keeping related data and behavior together while controlling how the object's state is accessed or modified.

Advanced

10. What are the four commonly discussed principles of OOP?

Encapsulation
Inheritance
Polymorphism
Abstraction

11. Does Python require OOP for every program?

No. Python supports multiple programming styles, and OOP should be used when it improves organization, reuse, or maintainability.

---

40. Cheat Sheet

Create a class

```
class User:
    pass
```

Create an object

```
user = User()
```

Initialize attributes

```
class User:

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

Create object with data

```
user = User("Hafsa")
```

Access attribute

```
print(user.name)
```

Create method

```
class User:

    def greet(self):
        print("Hello")
```

Call method

```
user.greet()
```

Multiple objects

```
user1 = User("Hafsa")
user2 = User("Asim")
```

Class attribute

```
class User:

    platform = "haas.dev"
```

Module logger inside a class-based project

```
import logging

logger = logging.getLogger(__name__)
```

---

41. Key Takeaways

- OOP stands for **Object Oriented Programming**.
- OOP organizes programs around objects.
- Objects combine **data and behavior**.
- A **class** is a blueprint.
- An **object** is an instance of a class.
- **Attributes** represent an object's data.
- **Methods** represent an object's behavior.
- `self` refers to the current object.
- `__init__()` initializes an object's data.
- Multiple objects can be created from the same class.
- Each object can have its own state.
- OOP commonly discusses encapsulation, inheritance, polymorphism, and abstraction.
- OOP is useful for organizing larger applications and modeling entities.
- Not every Python program needs classes.
- Good OOP design focuses on clear responsibilities rather than creating classes unnecessarily.

The core mental model:

CLASS
  □
Blueprint
  □
OBJECT
  □□□ Attributes □ Data
  □□□ Methods □ Behavior

Visit haas.dev for more resources and guides.

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