Class Attributes and Methods in Python

Introduction

In Python OOP, we have learned about **instance attributes** and **instance methods**.

Now we need to understand something slightly different:

- **Instance attributes** belong to individual objects.
- **Class attributes** belong to the class and can be shared by its objects.
- **Instance methods** work with a specific object.
- **Class methods** work with the class itself.

Understanding this difference is important because it helps you decide where information and behavior should live.

A simple mental model is:

```
Class
□
□□□ Class Attributes
□     Shared class-level data
□
□□□ Class Methods
□     Class-level behavior
□
□□□ Objects
      □
      □□□ Instance Attributes
      □□□ Instance Methods
```

---

1. Instance vs Class Level

Suppose we create a `User` class.

Every user can have a different:

`name`
`email`
`age`

These are instance attributes.

But all users may belong to the same application:

```
application_name = "haas.dev"
```

That information could be stored at the class level.

Example:

```
class User:
    application_name = "haas.dev"

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

Here:

`application_name`

is a **class attribute**.

While:

`self.name`

is an **instance attribute**.

---

2. What Is a Class Attribute?

A **class attribute** is an attribute defined directly inside the class body rather than inside a method such as `__init__()`.

Example:

```
class User:
    platform = "haas.dev"

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

Now:

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

Both objects can access:

```
print(user1.platform)
print(user2.platform)
```

Output:

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

The value is associated with the class.

---

3. Accessing a Class Attribute

The clearest way to access a class attribute is through the class itself:

```
print(User.platform)
```

Output:

```
haas.dev
```

You can also access it through an instance:

```
print(user1.platform)
```

Python can find the attribute on the class when it does not find an instance attribute with that name.

So:

```
user1.platform
```

works because Python looks for `platform` on the object and then on its class.

---

4. Class Attribute vs Instance Attribute

Consider:

```
class User:
```

```
    platform = "haas.dev"

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

Create two objects:

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

Their data is:

```
User class
    platform = "haas.dev"

user1
    name = "Hafsa"

user2
    name = "Asim"
```

So:

```
platform  class-level
name      instance-level
```

---

5. Why Use Class Attributes?

Class attributes are useful when a value logically belongs to the class rather than one particular object.

Examples:

```
class Employee:
    company = "HAAS"

class Circle:
    pi = 3.14159

class User:
    platform = "haas.dev"

class Product:
    tax_rate = 0.15
```

The important question is:

> Is this value a property of every individual object, or is it shared by the class?

If it is shared conceptually, a class attribute may be appropriate.

---

6. Class Attributes Can Be Shared

Example:

```
class Employee:
    company = "HAAS"

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

Now:

```
employee1 = Employee("Hafsa")
employee2 = Employee("Asim")
```

Both see:

```
print(employee1.company)
print(employee2.company)
```

Output:

```
HAAS
HAAS
```

There is one class-level definition that both objects can access.

---

7. Changing a Class Attribute

You can change a class attribute through the class:

```
Employee.company = "New Company"
```

Now:

```
print(employee1.company)
print(employee2.company)
```

Both will normally show:

```
New Company
```

because neither object has its own `company` attribute.

---

8. Creating an Instance Attribute With the Same Name

This is an important concept.

Suppose:

```
class Employee:
    company = "HAAS"

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

Now:

```
employee1.company = "Other Company"
```

This does **not** change the class attribute.

Instead, it creates an instance attribute for `employee1`.

Conceptually:

```
Class:
    company = "HAAS"

employee1:
    company = "Other Company"

employee2:
    no own company
```

Now:

```
print(employee1.company)
print(employee2.company)
print(Employee.company)
```

Output:

```
Other Company
HAAS
HAAS
```

This is called **attribute shadowing**.

---

9. How Python Looks Up Attributes

When you write:

```
employee1.company
```

Python needs to find `company`.

A simplified lookup model is:

```
1. Look on the instance
    
2. If not found, look on the class
    
3. Continue through parent classes if needed
```

So if `employee1` has:

```
employee1.company = "Other Company"
```

Python uses that value instead of the class value.

If the instance doesn't have it, Python can use:

```
Employee.company
```

---

10. Mutable Class Attributes

One of the most important dangers with class attributes is using mutable objects such as lists or dictionaries.

Consider:

```
class Student:
    subjects = []
```

Now:

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

student1.subjects.append("Python")
```

You might expect only `student1` to change.

But:

```
print(student2.subjects)
```

will also show:

```
['Python']
```

Why?

Because both objects are accessing the same class-level list.

Conceptually:

```
Student.subjects
      
      
 
      
student1   student2
```

Both refer to the same list.

---

11. Avoid Shared Mutable State When It Should Be Independent

Usually, this is what you actually want:

```
class Student:
    def __init__(self):
        self.subjects = []
```

Now:

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

student1.subjects.append("Python")
```

Then:

```
print(student1.subjects)
print(student2.subjects)
```

Output:

```
['Python']
[]
```

Each object gets its own list.

A useful rule:

> If mutable data should be different for every object, make it an instance attribute.

---

12. What Is a Class Method?

A **class method** is a method that operates on the class rather than a particular instance.

It is created using the:

```
@classmethod
```

decorator.

Example:

```
class User:
    platform = "haas.dev"

    @classmethod
    def get_platform(cls):
        return cls.platform
```

Now call it:

```
print(User.get_platform())
```

Output:

```
haas.dev
```

---

13. The Role of `cls`

Instance methods use:

```
self
```

Class methods use:

```
cls
```

`cls` refers to the **class**.

Compare:

```
def instance_method(self):
```

with:

```
@classmethod
def class_method(cls):
```

Think:

```
self → current object
cls  → current class
```

---

14. Calling a Class Method

Example:

```
class User:
    platform = "haas.dev"

    @classmethod
    def get_platform(cls):
        return cls.platform
```

Call:

```
User.get_platform()
```

Python supplies the class as `cls`.

Conceptually:

```
User.get_platform()
```

is similar to:

```
User.get_platform(User)
```

The important difference is that the first argument represents the class.

---

15. Class Methods Can Also Be Called Through Objects

Given:

```
user = User()
```

you can write:

```
user.get_platform()
```

Python still binds the method to the class.

However, when the behavior is explicitly class-level, calling it through the class is often clearer:

```
User.get_platform()
```

---

16. Why Use Class Methods?

Class methods are useful when behavior:

- needs access to class-level data
- should operate on the class
- creates objects in an alternative way

- provides a class-level operation

A very common use is an **alternative constructor**.

---

17. Class Methods as Alternative Constructors

Suppose:

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

Normally:

```
user = User("Hafsa", 25)
```

But perhaps you also want to create a user from a string:

```
"Hafsa,25"
```

A class method can provide another way:

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

    @classmethod
    def from_string(cls, data):
        name, age = data.split(",")
```

```
        return cls(name, int(age))
```

Now:

```
user = User.from_string("Hafsa,25")

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

Output:

```
Hafsa
25
```

This is called an **alternative constructor**.

---

18. Why Use `cls()` Instead of the Class Name?

Inside a class method, prefer:

```
return cls(name, int(age))
```

instead of:

```
return User(name, int(age))
```

Why?

Because `cls` represents the class on which the method was called.

This becomes especially useful when inheritance is involved.

For example, if another class inherits from `User`, `cls(...)` can create an instance of that subclass rather than always creating a `User`.

This makes the class method more reusable.

---

19. Class Method Example: Resource

Imagine a haas.dev resource system.

```
class Resource:
    platform = "haas.dev"

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

    @classmethod
    def from_data(cls, data):
        title, category = data.split("|")
        return cls(title, category)
```

Now:

```
resource = Resource.from_data(
    "Python Functions|Python"
)

print(resource.title)
print(resource.category)
```

Output:

Python Functions
Python

The class method provides another way to construct a Resource.

---

20. Class Methods Can Access Class Attributes

Example:

```
class Product:
    tax_rate = 0.15

    @classmethod
    def get_tax_rate(cls):
        return cls.tax_rate
```

Usage:

```
print(Product.get_tax_rate())
```

Output:

```
0.15
```

The class method accesses:

```
cls.tax_rate
```

because tax_rate belongs to the class.

---

21. Class Attributes and Class Methods Together

These two concepts often work together.

```
class Employee:
    company = "HAAS"

    @classmethod
    def change_company(cls, new_name):
        cls.company = new_name
```

Usage:

```
Employee.change_company("HAAS Technologies")

print(Employee.company)
```

Output:

```
HAAS Technologies
```

The class method changes class-level state.

---

22. Instance Method vs Class Method

Feature	Instance Method	Class Method
Decorator	None	@classmethod

First parameter	`self`	`cls`
Refers to	Current object	Current class
Main purpose	Instance behavior	Class-level behavior
Accesses instance data	Yes	Not directly
Accesses class data	Yes	Yes
Typical call	`obj.method()`	`Class.method()`

Mental model:

```
Instance Method
   
self
   
specific object

Class Method
   
cls
   
class
```

---

23. Class Method vs Instance Method Example

Consider:

```
class User:
    platform = "haas.dev"

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

    def greet(self):
        return f"Hello, {self.name}"

    @classmethod
    def platform_name(cls):
        return cls.platform
```

Now:

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

Instance method:

```
print(user.greet())
```

Output:

```
Hello, Hafsa
```

It needs a specific user.

Class method:

```
print(User.platform_name())
```

Output:

```
haas.dev
```

It doesn't need a specific user.

---

24. What About Methods That Need Neither `self` nor `cls`?

Sometimes a method belongs logically to a class but does not need:

- instance data
- class data

For this situation, Python provides:

```
@staticmethod
```

Example:

```
class MathTools:

    @staticmethod
    def add(a, b):
        return a + b
```

Usage:

```
print(MathTools.add(5, 3))
```

Output:

```
8
```

A static method does not automatically receive `self` or `cls`.

Static methods will be covered separately in greater depth.

---

25. Three Types of Methods

At this point, you can think about three major method types:

```
Instance Method
    
self
    
Works with object

Class Method
    
cls
    
Works with class

Static Method
    
No automatic self/cls
    
Utility behavior related to the class
```

Example:

```
class User:
    platform = "haas.dev"

    def greet(self):
        return f"Hello, {self.name}"

    @classmethod
    def get_platform(cls):
        return cls.platform

    @staticmethod
    def is_valid_name(name):
        return bool(name.strip())
```

---

26. Choosing the Correct Method Type

When designing a method, ask:

Does it need a specific object's data?

Use an **instance method**.

```
user.greet()
```

Does it need class-level data or represent a class-level operation?

Use a **class method**.

```
User.get_platform()
```

Does it need neither instance nor class state?

Consider a **static method**.

```
User.is_valid_name("Hafsa")
```

This decision keeps class design clear.

---

27. A Realistic Example

Consider a `Resource` class:

```
class Resource:
    total_resources = 0
    platform = "haas.dev"

    def __init__(self, title):
        self.title = title
        Resource.total_resources += 1

    def get_title(self):
        return self.title

    @classmethod
    def get_total_resources(cls):
        return cls.total_resources
```

Create resources:

```
resource1 = Resource("Python Functions")
resource2 = Resource("Python OOP")
resource3 = Resource("Python Exceptions")
```

Now:

```
print(Resource.get_total_resources())
```

Output:

```
3
```

Here:

```
title
```

is specific to each resource.

But:

```
total_resources
```

belongs to the class.

---

28. Class Attributes as Counters

A common use of class attributes is keeping track of how many objects have been created.

```
class User:
    total_users = 0

    def __init__(self, name):
        self.name = name
        User.total_users += 1
```

Now:

```
user1 = User("Hafsa")
user2 = User("Asim")
user3 = User("Ali")

print(User.total_users)
```

Output:

```
3
```

The counter represents information about the collection of `User` objects rather than one particular user.

---

29. A Better Version Using `cls`

You may also see:

```
class User:
    total_users = 0

    def __init__(self, name):
        self.name = name
        type(self).total_users += 1
```

For simple cases, using:

```
User.total_users
```

is easier to understand.

When designing inheritance-friendly classes, `type(self)` or other class-aware approaches can become useful.

The key point is:

> Class attributes represent shared class-level state, but inheritance can make shared state more subtle.

---

30. Constants as Class Attributes

Class attributes can also hold values that are conceptually constants.

Python does not enforce constants, but uppercase naming communicates the intention.

```
class Config:
    MAX_LOGIN_ATTEMPTS = 5
    DEFAULT_LANGUAGE = "en"
```

Access them:

```
print(Config.MAX_LOGIN_ATTEMPTS)
```

Output:

```
5
```

These values are associated with the class rather than individual objects.

---

31. Class Attributes and Inheritance

Class attributes can be inherited.

```
class Animal:
    kingdom = "Animalia"

class Dog(Animal):
    pass
```

Now:

```
print(Dog.kingdom)
```

Output:

```
Animalia
```

`Dog` can access the class attribute defined by `Animal`.

A subclass can also define its own value:

```
class Dog(Animal):
    kingdom = "Animalia"
```

This replaces the inherited lookup with the subclass's own attribute.

Inheritance and overriding will be explored separately.

---

32. Common Mistake: Assuming Instance Assignment Changes the Class

Consider:

```
class User:
    platform = "haas.dev"
```

Then:

```
user = User()
user.platform = "Other"
```

This does not change:

```
User.platform
```

Instead, `user` now has its own `platform`.

So:

```
print(user.platform)
```

gives:

```
Other
```

while:

```
print(User.platform)
```

still gives:

```
haas.dev
```

---

33. Common Mistake: Using a Mutable Class Attribute

Avoid:

```
class Cart:
    items = []
```

if each cart should have its own items.

Use:

```
class Cart:
    def __init__(self):
        self.items = []
```

Otherwise, multiple objects can accidentally share the same list.

---

34. Common Mistake: Forgetting `@classmethod`

Incorrect:

```
class User:
    platform = "haas.dev"

    def get_platform(cls):
        return cls.platform
```

This is still an instance method.

Python expects the first argument to be the instance.

Correct:

```
class User:
    platform = "haas.dev"

    @classmethod
    def get_platform(cls):
        return cls.platform
```

Now:

```
User.get_platform()
```

works as a class-level call.

---

35. Common Mistake: Using `self` in a Class Method

Incorrect:

```
class User:
    platform = "haas.dev"

    @classmethod
    def get_platform(self):
        return self.platform
```

This may technically work because the parameter receives the class, but it is confusing and violates the conventional meaning of `self`.

Use:

```
@classmethod
def get_platform(cls):
    return cls.platform
```

Use:

```
self → instance
cls  → class
```

---

36. Common Mistake: Making Everything a Class Attribute

Not every value should be shared.

Bad design:

```
class User:
    name = ""
    email = ""
```

Every user does not have the same name and email.

Instead:

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

Class attributes should represent information that logically belongs to the class.

---

37. Design Question: Who Owns the Data?

Before creating an attribute, ask:

> Who owns this information?

For a `User`:

`name`
`email`
`age`

belong to individual users.

For the application:

`platform_name`
`version`

may belong to the class.

For a `Resource`:

`title`
`category`
`views`

belong to each resource.

But:

`platform = "haas.dev"`

can be class-level.

This ownership question is a useful OOP design tool.

---

38. Design Question: Who Owns the Behavior?

Use a similar question for methods:

> Does this behavior belong to one object or to the class?

Example:

`resource.open()`

A specific resource is being opened.

Instance method.

But:

`Resource.from_data(...)`

creates a resource from input.

Class method.

---

39. Practical Example: Product Catalog

```
class Product:
    store_name = "HAAS Store"
    tax_rate = 0.15

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

    def price_with_tax(self):
        return self.price * (1 + self.tax_rate)

    @classmethod
    def change_tax_rate(cls, new_rate):
        cls.tax_rate = new_rate
```

Create:

```
product = Product("Keyboard", 100)
```

Instance behavior:

```
print(product.price_with_tax())
```

Class-level behavior:

```
Product.change_tax_rate(0.20)
```

Now the class has:

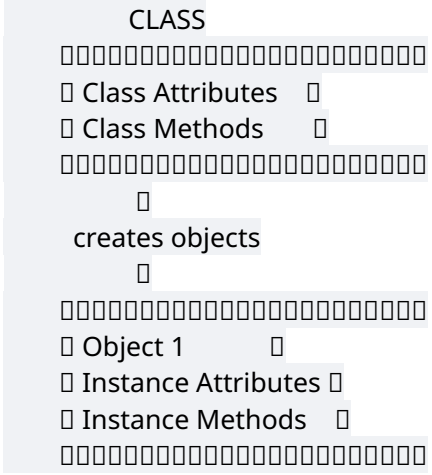
```
tax_rate = 0.20
```

This demonstrates how class-level configuration and instance-specific data can work together.

---

40. A Useful Mental Model

Think of a class as having two layers:



Class-level information describes the class or shared configuration.

Instance-level information describes one particular object.

---

41. Quick Comparison

```
class User:
    platform = "haas.dev"

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

    def greet(self):
```

```
        return f"Hello, {self.name}"

    @classmethod
    def get_platform(cls):
        return cls.platform
```

Class attribute

`User.platform`

Instance attribute

`user.name`

Instance method

`user.greet()`

Class method

`User.get_platform()`

---

42. Mini Project: Resource Manager

Create a `Resource` class with:

Class attributes

`platform`
`total_resources`

Instance attributes

`title`

`category`

Instance methods

`show_info()`

Class methods

`get_total_resources()`

Starter:

```
class Resource:
    platform = "haas.dev"
    total_resources = 0

    def __init__(self, title, category):
        self.title = title
        self.category = category
        Resource.total_resources += 1

    def show_info(self):
        return f"{self.title} | {self.category}"

    @classmethod
    def get_total_resources(cls):
        return cls.total_resources
```

Test it:

```
resource1 = Resource("Python Functions", "Python")
resource2 = Resource("Python OOP", "Python")

print(resource1.show_info())
print(Resource.get_total_resources())
```

Expected output:

```
Python Functions | Python
2
```

---

43. 5 Minute Challenge

Create a `Book` class.

Class attributes

```
library_name = "City Library"
total_books = 0
```

Instance attributes

```
title
author
```

Instance method

```
show_info()
```

Class method

```
get_total_books()
```

Create at least three books and verify that:

```
Book.get_total_books()
```

returns the correct number.

Then change:

`Book.library_name`

and verify that all books can access the updated class-level value.

---

44. Exercises

Exercise 1: Students

Create:

`class Student:`

Use:

`school`
`name`
`grade`

Make `school` a class attribute and `name` and `grade` instance attributes.

Exercise 2: Employees

Create:

`class Employee:`

Use:

company
name
salary

Add a class method that changes the company name.

Exercise 3: Products

Create:

class Product:

Use:

tax_rate
name
price

Add:

price_with_tax()
change_tax_rate()

Exercise 4: Users

Create:

class User:

Use:

`platform`
`name`

Add:

`greet()`
`get_platform()`

Make `get_platform()` a class method.

---

45. Mini Quiz

1. What is a class attribute?

A. Data stored only inside one object
B. Data associated with the class
C. A function parameter
D. A local variable

Answer: B

---

2. What does `cls` normally represent?

A. Current object
B. Current method
C. Current class
D. Parent object

Answer: C

---

3. Which decorator creates a class method?

A. `@staticmethod`
B. `@classmethod`
C. `@class`
D. `@method`

Answer: B

---

4. Which one is an instance attribute?

`self.name`

Answer: Instance attribute.

---

5. Which one is a class attribute?

```
class User:
    platform = "haas.dev"
```

Answer: `platform`.

---

6. What happens when an instance gets its own attribute with the same name as a class attribute?

Answer: The instance attribute shadows the class attribute for that object.

---

7. Why can a mutable class attribute cause problems?

Answer: Multiple objects can accidentally share the same mutable object, such as a list or dictionary.

---

46. Interview Questions

Q1. What is the difference between an instance attribute and a class attribute?

An instance attribute belongs to a particular object:

`self.name = "Hafsa"`

A class attribute belongs to the class:

`platform = "haas.dev"`

Q2. What is a class method?

A class method is a method bound to the class rather than a particular instance and is created with `@classmethod`.

Q3. What is `cls`?

`cls` is the conventional name for the first parameter of a class method and refers to the class.

Q4. What is the difference between `self` and `cls`?

`self` □ current instance

`cls` → current class

Q5. Can an instance access a class attribute?

Yes.

If an instance does not have an attribute with that name, Python can find the class attribute.

Q6. Does changing an instance attribute change the class attribute?

No.

For example:

```
user.platform = "Other"
```

creates or changes an attribute on that particular instance.

Q7. Why are class methods often used as alternative constructors?

Because they can receive different input formats and return a new instance using `cls(...)`.

Q8. Why should mutable data usually not be stored as a class attribute?

Because it can unintentionally be shared between multiple instances.

---

47. Cheat Sheet

Class attribute

```
class User:
    platform = "haas.dev"
```

Instance attribute

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

Access class attribute

```
User.platform
```

Access through object

```
user.platform
```

Instance method

```
def greet(self):
    return self.name
```

Class method

```
@classmethod
def get_platform(cls):
    return cls.platform
```

Call class method

```
User.get_platform()
```

Instance vs class

`self` → object
`cls` → class

Shared class-level value

```
class Product:
    tax_rate = 0.15
```

Independent mutable state

```
class Cart:
    def __init__(self):
        self.items = []
```

---

48. Key Takeaways

1. **Class attributes** belong to the class and can be accessed by its instances.
2. **Instance attributes** belong to individual objects.
3. Python looks on the instance before looking on the class during normal attribute lookup.
4. An instance attribute can **shadow** a class attribute with the same name.
5. Mutable class attributes can accidentally create shared state.
6. **Instance methods** use `self` and work with a particular object.
7. **Class methods** use `cls` and work with the class.
8. `@classmethod` is used to define class methods.
9. Class methods are commonly useful as **alternative constructors**.
10. Use class attributes for information that logically belongs to the class.
11. Use instance attributes for information that should be independent for each object.
12. Choose the method type based on whether the behavior needs object state, class state, or neither.

The core mental model is:

Class-level

class attributes
class methods

shared/class behavior

Instance-level

instance attributes
instance methods

object-specific behavior

Visit haas.dev for more resources and guides.

haas.dev

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