Instance Methods in Python

Introduction

In Object Oriented Programming, objects contain **data** and **behavior**.

We have already learned that:

- **Instance attributes** store an object's data.
- **Instance methods** define what an object can do.

For example, a `User` object may have:

- `name` → data
- `email` → data
- `login()` → behavior
- `logout()` → behavior

An instance method is simply a function defined inside a class that works with a particular object's data.

---

1. What Is an Instance Method?

An **instance method** is a method that belongs to an object and can access or modify that object's instance attributes.

Example:

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

```
    def greet(self):
        print(f"Hello, {self.name}!")
```

Create an object:

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

Call the method:

```
user.greet()
```

Output:

```
Hello, Hafsa!
```

Here:

```
greet()
```

is an instance method because it works with the specific `user` object.

---

2. The Role of `self`

The most important thing to understand about instance methods is `self`.

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

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

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

When we write:

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

Python internally treats the call roughly like:

```
User.greet(user)
```

So `self` receives the `user` object.

That is why this works:

```
self.name
```

It means:

> Get the `name` belonging to this particular object.

---

3. Why Instance Methods Need `self`

Consider two users:

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

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

Now create two objects:

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

Call the method:

```
user1.introduce()
user2.introduce()
```

Output:

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

The same method works differently because `self` refers to a different object each time.

Conceptually:

```
user1 → introduce() → self = user1
user2 → introduce() → self = user2
```

This is one of the main advantages of OOP.

---

4. Instance Methods Can Read Attributes

An instance method can access the object's attributes through `self`.

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

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

Usage:

```
product = Product("Keyboard", 50)

product.show_price()
```

Output:

```
Keyboard: $50
```

The method reads:

```
self.name
self.price
```

from the current object.

---

5. Instance Methods Can Modify Attributes

Instance methods are not limited to reading data.

They can also change an object's state.

```
class BankAccount:
    def __init__(self, balance):
        self.balance = balance

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

Usage:

```
account = BankAccount(100)

account.deposit(50)

print(account.balance)
```

Output:

```
150
```

The method changed:

```
self.balance
```

from `100` to `150`.

This is called **changing object state**.

---

6. Instance Methods Can Have Parameters

An instance method can accept additional parameters.

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

    def study(self, subject):
        print(f"{self.name} is studying {subject}")
```

Usage:

```
student = Student("Hafsa")

student.study("Python")
```

Output:

```
Hafsa is studying Python
```

There are two different types of parameters here:

`def study(self, subject):`

- `self` → the current object
- `subject` → additional information provided by the caller

When calling:

`student.study("Python")`

you normally provide only:

`"Python"`

Python automatically provides `self`.

---

7. `self` Is Not Passed Manually in Normal Method Calls

You normally write:

```
student.study("Python")
```

not:

```
student.study(student, "Python")
```

Python automatically passes the object as `self`.

The following:

```
student.study("Python")
```

is conceptually similar to:

```
Student.study(student, "Python")
```

Understanding this helps explain why `self` exists.

---

8. Instance Methods Can Return Values

An instance method can return information instead of printing it.

```
class Rectangle:
    def __init__(self, width, height):
        self.width = width
```

```
        self.height = height

    def area(self):
        return self.width * self.height
```

Usage:

```
rectangle = Rectangle(5, 4)

result = rectangle.area()

print(result)
```

Output:

```
20
```

The method uses the object's data and returns a calculated value.

---

9. Instance Methods Can Call Other Instance Methods

One instance method can call another instance method using `self`.

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

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

    def show_message(self):
        message = self.greet()
```

```
        print(message)
```

Usage:

```
user = User("Hafsa")

user.show_message()
```

Output:

```
Hello, Hafsa
```

Inside the class:

```
self.greet()
```

means:

> Call `greet()` for this same object.

---

10. Instance Methods Can Contain Business Logic

Methods become especially useful when an object has rules or behavior.

Example:

```
class BankAccount:
    def __init__(self, balance):
        self.balance = balance
```

```
    def withdraw(self, amount):
        if amount <= 0:
            return False

        if amount > self.balance:
            return False

        self.balance -= amount
        return True
```

Usage:

```
account = BankAccount(100)

success = account.withdraw(40)

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

Output:

```
True
60
```

The method doesn't just change data.

It also controls **how** the data can be changed.

---

11. Methods Help Protect Object State

Suppose we have:

```
class BankAccount:
    def __init__(self, balance):
        self.balance = balance
```

Someone could directly write:

```
account.balance = -500
```

This may create an invalid state.

Instead, behavior can be controlled through methods:

```
class BankAccount:
    def __init__(self, balance):
        self.balance = balance

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

    def withdraw(self, amount):
        if 0 < amount <= self.balance:
            self.balance -= amount
```

Now the class has clear rules for changing the account.

This idea connects to **encapsulation**, which will be explored more deeply later.

---

12. Instance Methods Represent Object Behavior

A useful mental model is:

```
Instance attributes □ What the object knows
Instance methods   □ What the object can do
```

For example:

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

BankAccount
 □□□ balance
 □□□ account_number
 □□□ deposit()
 □□□ withdraw()

Task
 □□□ title
 □□□ completed
 □□□ complete()
 □□□ reopen()
```

The class combines data and behavior into one meaningful unit.

---

13. Example: Task Manager

Let's model a simple task.

```
class Task:
```

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

    def complete(self):
        self.completed = True

    def reopen(self):
        self.completed = False

    def status(self):
        if self.completed:
            return "Completed"

        return "Pending"
```

Create a task:

```
task = Task("Learn Python OOP")
```

Check its status:

```
print(task.status())
```

Output:

```
Pending
```

Complete it:

```
task.complete()
```

Check again:

```
print(task.status())
```

Output:

```
Completed
```

Reopen it:

```
task.reopen()
```

Now:

```
print(task.status())
```

Output:

```
Pending
```

Here:

```
complete()
reopen()
status()
```

are all instance methods.

They operate on the same object's state.

---

14. Instance Methods and Multiple Objects

Each object maintains its own state.

```
class Counter:
    def __init__(self):
        self.value = 0

    def increment(self):
        self.value += 1
```

Create two counters:

```
counter1 = Counter()
counter2 = Counter()
```

Change only the first:

```
counter1.increment()
counter1.increment()
```

Now:

```
print(counter1.value)
print(counter2.value)
```

Output:

```
2
0
```

Why?

Because:

```
counter1.value  2
counter2.value  0
```

Each object has its own instance state.

---

15. Returning `self`

An instance method can return the current object.

```
class Counter:
    def __init__(self):
        self.value = 0

    def increment(self):
        self.value += 1
        return self
```

Now:

```
counter = Counter()

counter.increment().increment().increment()

print(counter.value)
```

Output:

```
3
```

This pattern is sometimes used for **method chaining**.

However, returning `self` should only be used when it makes the API clearer.

Do not use it simply because it is possible.

---

16. Method Chaining

Consider:

```
class Query:
    def __init__(self):
        self.filters = []

    def filter(self, condition):
        self.filters.append(condition)
        return self
```

Now:

```
query = Query()

query.filter("active").filter("verified")
```

Each method returns the same object, allowing another method call.

Method chaining is common in some libraries and APIs.

---

17. Instance Method vs Normal Function

A normal function:

```
def calculate_area(width, height):
    return width * height
```

An instance method:

```
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height
```

The main difference is where the data comes from.

With the function:

```
calculate_area(5, 4)
```

you explicitly provide the data.

With the method:

```
rectangle.area()
```

the object already contains:

```
self.width
self.height
```

So the behavior and related data are grouped together.

---

18. When Should You Use an Instance Method?

Use an instance method when an operation:

1. Works with a specific object.
2. Reads the object's instance attributes.
3. Changes the object's state.
4. Represents behavior belonging naturally to that object.

For example:

```
user.login()
account.withdraw(100)
task.complete()
cart.add_item(product)
playlist.remove_song(song)
```

These actions naturally belong to specific objects.

---

19. When Should You Not Use an Instance Method?

Not every function needs to become an instance method.

If an operation does not need object state, another type of function or method may be more appropriate.

For example:

```
def is_valid_email(email):
    return "@" in email
```

This function doesn't need a `User` object.

Later, you will learn about:

- `@classmethod`

- `@staticmethod`

These provide different ways of placing behavior inside classes.

---

20. Common Mistake: Forgetting `self`

Incorrect:

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

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

Calling:

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

causes an error because the method does not accept the automatically supplied object.

Correct:

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

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

---

21. Common Mistake: Using the Parameter Instead of the Attribute

Consider:

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

Here:

`name`

is the parameter.

`self.name`

is the instance attribute.

Inside another method, use:

`self.name`

because you want the value stored in the object.

---

22. Common Mistake: Changing the Wrong Object

Consider:

```
class Counter:
    def __init__(self):
```

```
        self.value = 0

    def increment(self):
        self.value += 1
```

If:

```
counter1.increment()
```

only `counter1` changes.

It does not change:

```
counter2
```

because each call operates on its own object.

---

23. Common Mistake: Using `print()` When You Need `return`

This:

```
class Rectangle:
    def area(self):
        print(self.width * self.height)
```

displays the result but doesn't give it back to the caller.

Often this is more useful:

```
class Rectangle:
    def area(self):
```

```
        return self.width * self.height
```

Now you can do:

```
area = rectangle.area()

print(area)
```

or:

```
if rectangle.area() > 20:
    print("Large rectangle")
```

A good rule:

> Use `return` when other code needs the result. Use `print()` when the method's purpose is specifically to display something.

---

24. Designing Good Instance Methods

Good instance methods usually have:

One clear responsibility

Instead of:

```
process_everything()
```

prefer focused methods such as:

```
validate()
save()
calculate_total()
send_email()
```

Meaningful names

Prefer:

```
calculate_total()
```

over:

```
do_it()
```

Appropriate object ownership

Ask:

> Does this behavior naturally belong to this object?

For example:

```
cart.add_item(product)
```

makes sense because a cart manages its items.

---

25. Instance Methods in a Real Application

Imagine a haas.dev resource system.

A resource might contain:

```
class Resource:
    def __init__(self, title, category):
        self.title = title
        self.category = category
        self.views = 0

    def open(self):
        self.views += 1
        print(f"Opening: {self.title}")

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

    def popularity(self):
        return self.views > 100
```

Usage:

```
resource = Resource(
    "Python Functions",
    "Python"
)

resource.open()
resource.open()

print(resource.get_info())
print(resource.views)
print(resource.popularity())
```

Possible output:

```
Opening: Python Functions
Opening: Python Functions
Python Functions | Python
2
False
```

The resource object stores its own state and provides methods to work with that state.

---

26. The Object Thinking Pattern

When designing a class, ask four questions:

```
1. What does this object know?
      
2. What can this object do?
      
3. Which data does each behavior need?
      
4. Which state can each behavior change?
```

For a `Task`:

```
What does it know?
 title
 completed

What can it do?
 complete()
 reopen()
 status()

What state can it change?
```

 completed

This turns OOP from syntax into a design technique.

---

27. Instance Method Execution Flow

When you write:

```
task.complete()
```

think of the flow as:

```
task
 
find complete()
 
Python passes task as self
 
method executes
 
self.completed changes
 
object now has updated state
```

This mental model is more important than memorizing the syntax.

---

28. Instance Methods vs Instance Attributes

Instance Attribute	Instance Method
Stores data	Defines behavior
Represents object state	Works with object state
Usually accessed with `self.name`	Called with `object.method()`
Can be changed	Can change attributes
Example: `self.name`	Example: `self.login()`

Think:

```
Attributes = data
Methods = behavior
```

Together:

```
Object = Data + Behavior
```

---

29. Mini Project: Simple Bank Account

Build a class with:

- account holder

- balance
- deposit method
- withdraw method
- balance method

Starter structure:

```
class BankAccount:
    def __init__(self, holder, balance=0):
        self.holder = holder
        self.balance = balance

    def deposit(self, amount):
        pass

    def withdraw(self, amount):
        pass

    def get_balance(self):
        pass
```

Your goal is to implement all three methods.

Expected usage:

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

account.deposit(500)
account.withdraw(200)

print(account.get_balance())
```

Expected result:

1300

---

30. 5 Minute Challenge

Create a `Playlist` class.

It should contain:

`name`
`songs`

Add these instance methods:

`add_song()`
`remove_song()`
`show_songs()`

Example:

```
playlist = Playlist("My Playlist")

playlist.add_song("Song A")
playlist.add_song("Song B")

playlist.show_songs()

playlist.remove_song("Song A")

playlist.show_songs()
```

Think about:

- Which data belongs to the object?
- Which methods modify that data?
- What should each method return?

---

31. Exercises

Exercise 1

Create a `Car` class with:

`brand`
`model`
`speed`

Add:

`accelerate()`
`brake()`
`show_speed()`

`accelerate()` should increase speed.

`brake()` should decrease speed without allowing speed to become negative.

Exercise 2

Create a `Student` class with:

`name`

`marks`

Add:

`add_mark()`
`average()`
`has_passed()`

The `average()` method should return the average mark.

Exercise 3

Create a `ShoppingCart` class with:

`items`

Add:

`add_item()`
`remove_item()`
`total_items()`

Exercise 4

Create a `Counter` class with:

`value`

Add:

`increment()`
`decrement()`
`reset()`

---

32. Mini Quiz

1. What is an instance method?

A. A variable inside a class
B. A method that works with a specific object
C. A Python package
D. A loop inside a class

Answer: B

---

2. What does `self` represent?

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

Answer: C

---

3. Which is a correct instance method?

```
class User:
```

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

A. Yes
B. No

Answer: A

---

4. Can an instance method change instance attributes?

Answer: Yes.

Example:

```
self.balance -= amount
```

---

5. Why can two objects produce different results from the same method?

Answer: Because `self` refers to the specific object making the method call, and each object can have different instance data.

---

33. Interview Questions

Beginner

Q: What is an instance method in Python?

An instance method is a method defined inside a class that operates on a specific object and normally receives that object through `self`.

Q: Why is `self` used?

`self` provides access to the current object's attributes and other instance methods.

Q: Is `self` a keyword?

No. `self` is the conventional parameter name used for the current instance.

Q: Can an instance method modify an object's state?

Yes. It can read and modify instance attributes.

Intermediate

Q: What happens when `obj.method()` is called?

Python binds the method to `obj` and supplies the object as the first argument, conventionally named `self`.

Q: What is the difference between a method and a function?

A method is associated with a class or object, while a function is generally a standalone callable. An instance method specifically operates on an instance.

Q: Can one instance method call another?

Yes:

```
self.other_method()
```

Q: Can an instance method return a value?

Yes. It can use `return` just like a normal function.

---

34. Instance Method Cheat Sheet

Basic method

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

Method using an attribute

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

Method with parameters

```
def add_score(self, score):
    self.score += score
```

Method returning a value

```
def get_score(self):
    return self.score
```

Method changing state

```
def complete(self):
    self.completed = True
```

Calling an instance method

`object.method()`

Calling another method

`self.method()`

Main idea

`self = current object`

---

35. Key Takeaways

1. An **instance method** defines behavior for a specific object.
2. Instance methods normally receive the current object through `self`.
3. `self` allows a method to access instance attributes.
4. Instance methods can read object state.
5. Instance methods can modify object state.
6. Instance methods can accept additional parameters.
7. Instance methods can return values.
8. One instance method can call another through `self`.
9. Different objects can produce different results from the same method because each object has its own state.
10. Good OOP design connects related data and behavior.

The core mental model is:

`Class`
`↓`
`Objects`
`↓`
`Each object has its own state`
`↓`
`Instance methods work with that state`

Once you understand instance methods, the next important step is learning how Python supports **class-level behavior** through class methods and static methods.

Visit haas.dev for more resources and guides.

haas.dev

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