Inheritance in Python

Introduction

As programs become larger, you often find that different classes have some behavior and data in common.

For example, imagine an application with:

```
User
 ├── name
 ├── email
 └── login()

Student
 ├── name
 ├── email
 ├── course
 └── login()

Instructor
 ├── name
 ├── email
 ├── subject
 └── login()
```

`Student` and `Instructor` both need some of the functionality of `User`.

Instead of writing the same code again, Python allows one class to **inherit** from another.

This is called **inheritance**.

> **Inheritance allows a child class to reuse and extend the attributes and methods of a parent class.**

---

1. The Basic Idea

Inheritance creates a relationship between classes.

```
Parent Class
    
 Child Class
```

The child class receives accessible functionality from the parent class.

For example:

```
class User:
    def login(self):
        print("User logged in")


class Student(User):
    pass
```

Now:

```
student = Student()

student.login()
```

Output:

```
User logged in
```

`Student` did not define `login()` itself.

It inherited it from `User`.

---

2. Parent and Child Classes

Inheritance uses several terms.

Parent class

Also called:

- base class
- superclass

It provides common functionality.

Child class

Also called:

- derived class
- subclass

It inherits from the parent and can add or change functionality.

Example:

```
class User:
    pass


class Student(User):
```

```
    pass
```

Here:

```
User    □ parent
Student □ child
```

---

3. Basic Inheritance Syntax

The syntax is:

```
class Child(Parent):
    pass
```

Example:

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


class Dog(Animal):
    pass
```

Now:

```
dog = Dog()

dog.eat()
```

Output:

```
Eating
```

`Dog` inherits `eat()` from `Animal`.

---

4. Why Use Inheritance?

Inheritance can help when multiple classes have a genuine **is-a relationship**.

For example:

```
Dog is an Animal
Cat is an Animal
Student is a User
Admin is a User
```

The parent contains common functionality.

The child contains specialized functionality.

```
Animal
 □□□ eat()
 □□□ sleep()

Dog
 □□□ bark()

Cat
 □□□ meow()
```

This avoids unnecessarily duplicating shared code.

---

5. Reusing Parent Methods

Suppose:

```
class User:
    def login(self):
        print("Logging in")

    def logout(self):
        print("Logging out")


class Student(User):
    pass
```

Now:

```
student = Student()

student.login()
student.logout()
```

Output:

```
Logging in
Logging out
```

The child automatically has access to the inherited methods.

---

6. Adding New Methods to the Child

A child class can add functionality of its own.

```
class User:
    def login(self):
        print("Logging in")


class Student(User):
    def study(self):
        print("Studying Python")
```

Now:

```
student = Student()

student.login()
student.study()
```

Output:

```
Logging in
Studying Python
```

The `Student` class has:

```
Inherited:
login()

Own:
study()
```

---

7. Adding Instance Attributes

A child class can also have its own attributes.

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


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

Now:

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

The student has:

```
name
course
```

However, this example duplicates the initialization of `name`.

A better approach is usually to reuse the parent's initialization with `super()`.

---

8. The `super()` Function

`super()` allows a child class to access functionality from its parent class.

Example:

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


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

Now:

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

The following happens:

```
Student.__init__()
      
super().__init__(name)
      
User.__init__()
      
self.name = name
```

Then `Student` adds:

```
self.course = course
```

---

9. Why Use `super()`?

Without `super()`:

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

The child duplicates the parent's initialization logic.

With:

```
super().__init__(name)
```

the parent handles its own initialization.

This is useful because if the parent later changes how its state is initialized, the child does not need to duplicate that logic.

---

10. Parent and Child Responsibilities

A useful design is:

```
Parent
 
Common functionality

Child
 
Specialized functionality
```

Example:

```
class User:
    def login(self):
        print("Logging in")

    def logout(self):
        print("Logging out")


class Student(User):
    def submit_assignment(self):
        print("Assignment submitted")
```

`User` manages general user behavior.

`Student` adds student-specific behavior.

---

11. Inheritance Can Reuse Attributes

Consider:

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


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

Now:

```
student = Student(
    "Hafsa",
    "hafsa@example.com",
    "Python"
)
```

The object contains:

`name`
`email`
`course`

`name` and `email` come from the parent initialization.

`course` is added by the child.

---

12. Checking Inheritance With `isinstance()`

Python provides:

`isinstance()`

Example:

```
class Animal:
    pass


class Dog(Animal):
    pass
```

```
dog = Dog()
```

Now:

```
print(isinstance(dog, Dog))
print(isinstance(dog, Animal))
```

Output:

```
True
True
```

Why is the second one `True`?

Because a `Dog` is also an `Animal` through inheritance.

---

13. Checking the Class Relationship

You can also use:

```
issubclass()
```

Example:

```
print(issubclass(Dog, Animal))
```

Output:

```
True
```

This asks:

> Is `Dog` a subclass of `Animal`?

You can also check:

`print(issubclass(Dog, Dog))`

which is:

`True`

because a class is considered a subclass of itself for this purpose.

---

14. Inheritance and Method Lookup

When you call:

`dog.eat()`

Python needs to find `eat()`.

A simplified model is:

`Dog`
`↓`
`Does Dog have eat()?`
`↓`
`No`

```
Check Animal
Found eat()
Execute it
```

If the child defines its own `eat()`, Python finds that version first.

This leads to **method overriding**.

---

15. Method Overriding

A child class can provide its own implementation of a method inherited from the parent.

Example:

```
class Animal:
    def speak(self):
        print("Animal makes a sound")


class Dog(Animal):
    def speak(self):
        print("Dog barks")
```

Now:

```
animal = Animal()
dog = Dog()
```

```
animal.speak()
dog.speak()
```

Output:

```
Animal makes a sound
Dog barks
```

The child has **overridden** the parent's method.

Method overriding is an important part of inheritance and will be explored separately in greater depth.

---

16. Calling the Parent Version With `super()`

Sometimes the child wants to extend the parent's behavior instead of completely replacing it.

Example:

```
class User:
    def login(self):
        print("User login")


class Admin(User):
    def login(self):
        super().login()
        print("Admin permissions loaded")
```

Now:

```
admin = Admin()
```

```
admin.login()
```

Output:

```
User login
Admin permissions loaded
```

The child calls the parent's implementation first and then adds its own behavior.

---

17. Inheritance With `__init__()`

Suppose:

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


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

Create:

```
admin = Admin("Hafsa", ["manage_resources"])
```

The initialization flow is:

```
Admin.__init__()
```

```
        
super().__init__(name)
        
User.__init__()
        
self.name = name
        
back to Admin
        
self.permissions = permissions
```

This lets each class initialize the state it owns.

---

18. A Child Does Not Have to Override Everything

Inheritance does not mean the child must redefine every parent method.

Example:

```
class User:
    def login(self):
        print("Login")

    def logout(self):
        print("Logout")


class Student(User):
    def study(self):
        print("Studying")
```

`Student` automatically gets:

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

and adds:

```
study()
```

Only specialized behavior needs to be added.

---

19. Example: Employee System

Imagine an application with different employee types.

Common employee information:

```
name
employee_id
salary
```

Common behavior:

```
work()
```

Specialized behavior:

```
Developer  write_code()
Designer   design()
Manager    manage_team()
```

We can model this with inheritance:

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

    def work(self):
        print("Employee is working")


class Developer(Employee):
    def write_code(self):
        print("Writing code")


class Designer(Employee):
    def design(self):
        print("Designing")


class Manager(Employee):
    def manage_team(self):
        print("Managing team")
```

Now:

```
developer = Developer("Hafsa", 101)

developer.work()
developer.write_code()
```

Output:

```
Employee is working
Writing code
```

---

20. Inheritance Creates an "Is A" Relationship

A useful test for inheritance is:

> Is the child genuinely a type of the parent?

Examples:

Dog is an Animal
Cat is an Animal
Student is a User
Admin is a User

But:

Car is an Engine
Student is a Course
Computer is a Keyboard

These are not naturally "is-a" relationships.

They may instead represent **composition**, where one object contains or uses another object.

---

21. Inheritance vs Composition

Suppose a `Car` has an engine.

A car is **not** an engine.

Instead:

```
Car
 □□□ has an Engine
```

This is composition.

Example:

```
class Engine:
    def start(self):
        print("Engine started")


class Car:
    def __init__(self):
        self.engine = Engine()

    def start(self):
        self.engine.start()
        print("Car started")
```

Usage:

```
car = Car()

car.start()
```

Here:

```
Car HAS an Engine
```

not:

`Car IS an Engine`

Choosing between inheritance and composition is an important OOP design decision.

---

22. Why Overusing Inheritance Can Be a Problem

Inheritance can be useful, but it should not be used simply to avoid writing a few lines of code.

A deep inheritance hierarchy can become difficult to understand:

A

B

C

D

E

A change in a parent class can affect many subclasses.

Good inheritance usually represents a clear conceptual relationship.

---

23. Single Inheritance

Python supports a class inheriting from one parent class.

Example:

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


class Dog(Animal):
    pass
```

This is called **single inheritance**.

---

24. Multilevel Inheritance

A class can inherit from a class that itself inherits from another class.

Example:

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


class Mammal(Animal):
    def walk(self):
        print("Walking")


class Dog(Mammal):
    def bark(self):
        print("Barking")
```

Now:

```
dog = Dog()

dog.eat()
dog.walk()
dog.bark()
```

The inheritance chain is:

```
Animal
  ↓
Mammal
  ↓
Dog
```

---

25. Hierarchical Inheritance

Multiple child classes can inherit from the same parent.

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


class Dog(Animal):
    pass


class Cat(Animal):
    pass
```

The structure is:

```
    Animal
    /   \
  Dog    Cat
```

Both children inherit from `Animal`.

---

26. Multiple Inheritance

Python also supports multiple inheritance.

A class can inherit from more than one parent:

```
class Camera:
    def take_photo(self):
        print("Taking photo")


class GPS:
    def get_location(self):
        print("Getting location")


class Smartphone(Camera, GPS):
    pass
```

Now:

```
phone = Smartphone()
```

```
phone.take_photo()
phone.get_location()
```

Output:

```
Taking photo
Getting location
```

The structure is:

```
Camera □□□□
      □
   Smartphone
      □
GPS □□□□□□□
```

Multiple inheritance is powerful but can introduce complexity, especially when parents contain methods with the same names.

---

27. Method Resolution Order

When multiple inheritance is involved, Python needs to determine which class should be searched first.

This is called **Method Resolution Order**, or **MRO**.

Example:

```
class A:
    pass


class B(A):
```

```
    pass


class C(B):
    pass
```

You can inspect the MRO:

```
print(C.mro())
```

Python returns the order in which it searches classes.

You can also use:

```
print(C.__mro__)
```

Understanding MRO becomes especially important with multiple inheritance and `super()`.

---

28. Inheritance of Class Attributes

Class attributes can also be inherited.

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


class Student(User):
    pass
```

Now:

```
print(Student.platform)
```

Output:

```
haas.dev
```

`Student` can access the class attribute inherited from `User`.

---

29. Overriding a Class Attribute

A child can provide its own class attribute:

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


class Admin(User):
    platform = "haas.dev Admin"
```

Now:

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

Output:

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

The child version takes precedence when accessed through `Admin`.

---

30. Inheritance and `super()`

`super()` is not simply "the parent class."

It follows Python's method resolution order.

For simple single inheritance:

```
super().method()
```

often means:

> Call the next appropriate implementation in the inheritance chain.

This distinction becomes important with multiple inheritance.

---

31. Example: haas.dev Resource Types

Imagine haas.dev has different resource types.

Common properties:

```
title
category
```

Common behavior:

```
open()
```

Specialized resources could add their own behavior.

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

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


class PDFResource(Resource):
    def download(self):
        print(f"Downloading {self.title}")


class QuizResource(Resource):
    def start_quiz(self):
        print(f"Starting quiz: {self.title}")
```

Now:

```
pdf = PDFResource(
    "Python OOP",
    "Python"
)

quiz = QuizResource(
    "Python Quiz",
    "Python"
)
```

Both inherit:

`open()`

But each has specialized behavior:

`PDFResource`
 download()

`QuizResource`
 start_quiz()

---

32. Extending the haas.dev Example

We can also override behavior.

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

    def open(self):
        print(f"Opening resource: {self.title}")


class PDFResource(Resource):
    def open(self):
        print(f"Opening PDF: {self.title}")

    def download(self):
        print(f"Downloading PDF: {self.title}")
```

Now:

```
pdf = PDFResource(
    "Python OOP",
    "Python"
)

pdf.open()
pdf.download()
```

Output:

```
Opening PDF: Python OOP
Downloading PDF: Python OOP
```

The child specializes the parent's behavior.

---

33. Inheritance and Code Reuse

One advantage of inheritance is reducing duplication.

Without inheritance:

```
class Student:
    def login(self):
        print("Login")

    def logout(self):
        print("Logout")


class Admin:
    def login(self):
```

```
        print("Login")

    def logout(self):
        print("Logout")
```

The same code appears twice.

With inheritance:

```
class User:
    def login(self):
        print("Login")

    def logout(self):
        print("Logout")


class Student(User):
    pass


class Admin(User):
    pass
```

The common behavior exists in one place.

---

34. Inheritance and Maintainability

Suppose the login logic changes.

Without inheritance, you might need to update several classes.

With inheritance:

```
class User:
    def login(self):
        print("New login process")
```

Child classes can automatically receive the updated behavior unless they override it.

This can make shared behavior easier to maintain.

However, inheritance should still be used only when the relationship is appropriate.

---

35. Common Mistake: Using Inheritance for Everything

Bad design:

```
Car  Engine
```

because a car has an engine.

Better:

```
Car
    Engine
```

through composition.

Remember:

```
Inheritance  IS-A
```

Composition HAS-A

---

36. Common Mistake: Forgetting `super().__init__()`

Consider:

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


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

Now:

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

print(student.name)
```

This fails because the parent's initialization was never called.

Correct:

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

---

37. Common Mistake: Repeating Parent Code

Instead of:

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

prefer:

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

when the parent already owns the common initialization.

---

38. Common Mistake: Confusing Inheritance With Object Creation

Inheritance defines a relationship between classes.

This:

```
class Student(User):
    pass
```

does not create a `Student` object.

It creates a new class.

An object is created later:

```
student = Student()
```

Think:

```
class definition  creates a class
class inheritance  defines class relationship
class call  creates an object
```

---

39. Common Mistake: Expecting Child Changes to Automatically Change the Parent

Suppose:

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


class Student(User):
    platform = "Student Portal"
```

Now:

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

Output:

```
haas.dev
```

Student Portal

The subclass's attribute does not modify the parent's attribute.

---

40. Common Mistake: Deep Inheritance Hierarchies

This can become difficult:

Entity

User

Employee

Developer

SeniorDeveloper

TeamLead

The behavior of a class may depend on many parent classes.

Before creating another inheritance layer, ask:

> Does this really represent a meaningful type relationship?

If not, composition or another design may be better.

---

41. When Inheritance Is a Good Fit

Inheritance is often useful when:

There is a genuine "is-a" relationship

`Dog is an Animal`

Child classes share meaningful behavior

```
PDFResource
QuizResource
VideoResource
```

The parent represents a meaningful abstraction

```
Resource
   
specific resource types
```

Child classes need specialized behavior

```
Developer  write_code()
Designer   design()
```

---

42. When Composition May Be Better

Consider composition when the relationship is:

```
has-a
uses-a
contains-a
```

Examples:

```
Car has an Engine
User has a Profile
Order has a Payment
Computer has a Keyboard
```

Composition can keep classes more independent.

---

43. Inheritance Mental Model

Think of inheritance as:

```
Parent
 □
Common identity + behavior
 □
Child
 □
Specialized identity + behavior
```

Example:

```
          User
           □
   □□□□□□□□□□□□□□□□□□□□
   □             □
 Student         Admin
   □             □
 course        permissions
```

Common:

name
email
login()
logout()

Specialized:

Student → study()
Admin → manage_users()

---

44. Mini Project: Employee System

Create:

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

    def work(self):
        print("Employee is working")
```

Then create:

Developer
Designer
Manager

Each should inherit from `Employee`.

Add:

```
Developer → write_code()
Designer  → create_design()
Manager   → manage_team()
```

Use `super().__init__()` for the common employee data.

Example:

```
developer = Developer("Hafsa", 101)

developer.work()
developer.write_code()
```

---

45. 5 Minute Challenge

Create an `Animal` hierarchy.

Parent:

```
Animal
 ├── name
 └── eat()
```

Children:

```
Dog
 └── bark()

Cat
```

□□□ meow()

Requirements:

1. Dog and Cat must inherit from Animal.
2. Use super().__init__() if the children define their own constructors.
3. Add one unique method to each child.
4. Create one object of each class.
5. Call both inherited and child-specific methods.

Bonus:

Override speak() in both child classes.

---

46. Exercises

Exercise 1: Users

Create:

User
□□□ name
□□□ email
□□□ login()

Student
□□□ study()

Instructor
□□□ teach()

Use inheritance.

Exercise 2: Vehicles

Create:

```
Vehicle
 □□□ brand
 □□□ start()

Car
 □□□ drive()

Bike
 □□□ ride()
```

Use `super()` to initialize the common data.

Exercise 3: Resources

Create:

```
Resource
 □□□ title
 □□□ category
 □□□ open()

PDFResource
 □□□ download()

QuizResource
 □□□ start_quiz()
```

Then override `open()` for `PDFResource`.

Exercise 4: Employees

Create:

```
Employee
 □□□ name
 □□□ salary
 □□□ work()

Developer
 □□□ write_code()

Designer
 □□□ design()
```

Use `super().__init__()`.

---

47. Mini Quiz

1. What is inheritance?

A. Creating variables
B. Reusing and extending functionality from another class
C. Creating a loop
D. Importing a module

Answer: B

---

2. What is the parent class?

A. The class that inherits
B. The class being inherited from
C. The object
D. The method

Answer: B

---

3. What does this mean?

```
class Student(User):
    pass
```

Answer: `Student` inherits from `User`.

---

4. What does `super()` help with?

Answer: It provides access to the next class in the inheritance hierarchy, commonly the parent implementation in simple inheritance.

---

5. What is method overriding?

Answer: When a child class provides its own implementation of a method inherited from its parent.

---

6. What does `isinstance(student, User)` check?

Answer: Whether `student` is an instance of `User` or one of its subclasses.

---

7. What does `issubclass(Student, User)` check?

Answer: Whether `Student` is a subclass of `User`.

---

8. What is the difference between IS-A and HAS-A?

Answer:

IS-A inheritance
HAS-A composition

Example:

Dog IS-A Animal
Car HAS-A Engine

---

48. Interview Questions

Q1. What is inheritance in Python?

Inheritance allows one class to reuse and extend the attributes and methods of another class.

Q2. What is the difference between a parent and child class?

The parent provides common functionality. The child inherits that functionality and can add or override behavior.

Q3. What is `super()`?

`super()` provides a way to access the next implementation in the class's method resolution order. It is commonly used to call parent initialization or behavior.

Q4. Why use `super().__init__()`?

It allows a child class to reuse the parent's initialization logic rather than duplicating it.

Q5. What is method overriding?

It occurs when a child class defines a method with the same name as an inherited method, providing specialized behavior.

Q6. What is `isinstance()`?

It checks whether an object is an instance of a specified class or its subclasses.

Q7. What is `issubclass()`?

It checks whether one class is a subclass of another.

Q8. What is multiple inheritance?

Multiple inheritance occurs when a class inherits from more than one parent class.

Example:

```
class Smartphone(Camera, GPS):
    pass
```

Q9. What is MRO?

MRO stands for **Method Resolution Order**. It determines the order Python follows when looking for methods and attributes through an

inheritance hierarchy.

Q10. When should inheritance be avoided?

Inheritance should generally be avoided when the relationship is not a meaningful "is-a" relationship or when composition provides a clearer design.

---

49. Inheritance Cheat Sheet

Basic inheritance

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


class Dog(Animal):
    pass
```

Add child behavior

```
class Dog(Animal):
    def bark(self):
        print("Barking")
```

Parent constructor

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


class Student(User):
```

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

Override method

```
class Dog(Animal):
    def speak(self):
        print("Bark")
```

Call parent behavior

```
def speak(self):
    super().speak()
    print("More behavior")
```

Check object relationship

```
isinstance(obj, Class)
```

Check class relationship

```
issubclass(Child, Parent)
```

Check MRO

```
Class.mro()
```

Core relationship

IS-A Inheritance
HAS-A Composition

---

50. Key Takeaways

1. **Inheritance** allows one class to reuse and extend another class.
2. The inherited-from class is the **parent**, **base**, or **superclass**.
3. The inheriting class is the **child**, **derived**, or **subclass**.
4. A child automatically receives accessible functionality from its parent.
5. A child can add its own attributes and methods.
6. A child can override inherited methods.
7. `super()` is commonly used to reuse parent initialization and behavior.
8. `isinstance()` checks an object's class relationship.
9. `issubclass()` checks a relationship between classes.
10. Python supports single, multilevel, hierarchical, and multiple inheritance.
11. MRO determines the order Python follows when searching the inheritance hierarchy.
12. Inheritance works best when there is a meaningful **is-a** relationship.
13. Composition is often better for **has-a** relationships.
14. Avoid unnecessarily deep or complicated inheritance hierarchies.
15. Good inheritance separates **common behavior** from **specialized behavior**.

The central mental model is:

```
        Parent
          
  Common data + behavior
          
        Child
          
  Specialized behavior
```

Visit haas.dev for more resources and guides.

haas.dev

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