# Python Examples: super() in Single Inheritance

## 1. Order and OnlineOrder

class Order:  
 def \_\_init\_\_(self, order\_id):  
 self.order\_id = order\_id  
  
 def process\_payment(self):  
 print(f"Processing payment for Order #{self.order\_id}")  
  
class OnlineOrder(Order):  
 def \_\_init\_\_(self, order\_id, email):  
 super().\_\_init\_\_(order\_id)  
 self.email = email  
  
 def process\_payment(self):  
 super().process\_payment()  
 print(f"Sending confirmation email to {self.email}")  
  
order = OnlineOrder(101, "customer@example.com")  
order.process\_payment()

EXPLAINATION:

The OnlineOrder class extends the Order class, meaning it inherits all the basic features related to orders.

It uses super().\_\_init\_\_(order\_id) to call the parent constructor and avoid repeating code for setting the order ID.

The process\_payment() method is overridden to keep the original behavior and add a new step: sending a confirmation email.

OnlineOrder introduces an additional attribute, email, showing how subclasses can have their own extra properties.

When we create an OnlineOrder and call process\_payment(), Python automatically uses the updated method — demonstrating polymorphism.

## 2. Employee and Manager

class Employee:  
 def \_\_init\_\_(self, name, salary):  
 self.name = name  
 self.salary = salary  
  
 def display(self):  
 print(f"Name: {self.name}, Salary: ₹{self.salary}")  
  
class Manager(Employee):  
 def \_\_init\_\_(self, name, salary, department):  
 super().\_\_init\_\_(name, salary)  
 self.department = department  
  
 def display(self):  
 super().display()  
 print(f"Department: {self.department}")  
  
m = Manager("Shaik", 90000, "IT")  
m.display()

EXPLAINATION:

The Manager class is built on top of Employee, so it automatically gets all the basic features like name and salary.

It uses super().\_\_init\_\_(name, salary) to call the parent constructor and initialize common attributes without repeating code.

The display() method is overridden in Manager to include extra information — specifically, the department the manager belongs to.

Manager also introduces a new attribute called department, which isn’t part of the base Employee class.

When we call m.display(), the program first runs the original display logic, then adds the department info — showing how subclass methods can extend parent behavior.

## 3. Vehicle and Car

class Vehicle:  
 def start(self):  
 print("Vehicle started")  
  
class Car(Vehicle):  
 def start(self):  
 super().start()  
 print("Car is ready to go")  
  
c = Car()  
c.start()

The Car class is derived from Vehicle, so it inherits the general behavior of a vehicle.

It overrides the start() method but still uses super().start() to include the original message before adding its own.

This program focuses only on method logic and doesn’t use any constructor, keeping it simple.

It’s a clear example of **single inheritance**, where one class builds on another.

The overridden start() method in Car adds custom behavior while keeping the original intact — a clean use of method overriding.

## 4. User Login System

class User:  
 def \_\_init\_\_(self, username):  
 self.username = username  
  
 def login(self):  
 print(f"{self.username} logged in")  
  
class Admin(User):  
 def login(self):  
 super().login()  
 print(f"{self.username} has admin privileges")  
  
a = Admin("admin\_user")  
a.login()

EXPLANATION

The Admin class inherits from User, so it automatically gets access to user-related features like the username.

It reuses the constructor from the User class, showing that we don’t need to redefine the setup logic for username.

The login() method is overridden in Admin, but it smartly calls super().login() first to retain the base login message.

After the base login message, it adds an extra line saying the user has admin privileges-this reflects how roles can extend functionality.

When we create an Admin object and call login(), Python runs the overridden method, combining both user and admin behavior -a nice example of polymorphism.

## 5. Shape and Circle

class Shape:  
 def \_\_init\_\_(self):  
 print("This is a shape")  
  
 def area(self):  
 print("Area formula not defined")  
  
class Circle(Shape):  
 def \_\_init\_\_(self, radius):  
 super().\_\_init\_\_()  
 self.radius = radius  
  
 def area(self):  
 super().area()  
 print("Circle Area:", 3.14 \* self.radius \* self.radius)  
  
c = Circle(5)  
c.area()

EXPLAINATION

The Circle class calls the Shape constructor using super().\_\_init\_\_(), which prints a generic message indicating it's a shape.

The area() method in Circle begins by calling super().area() to show a placeholder message before calculating the actual area.

The base class provides a default area() method, acting as a suggestion that subclasses should override it with real logic.

The area of the circle is then calculated using the formula πr², implemented as 3.14 \* radius \* radius.

When we create a Circle object with c = Circle(5) and call c.area(), Python runs the overridden method from the Circle class, showing both base and subclass behavior in sequence.

## 6. Person and Student

class Person:  
 def \_\_init\_\_(self, name):  
 self.name = name  
  
 def show(self):  
 print(f"Name: {self.name}")  
  
class Student(Person):  
 def \_\_init\_\_(self, name, grade):  
 super().\_\_init\_\_(name)  
 self.grade = grade  
  
 def show(self):  
 super().show()  
 print(f"Grade: {self.grade}")  
  
s = Student("Ali", "A")  
s.show()

EXPLANATION

Student class holds both name and grade

super().\_\_init\_\_(name) allows the base constructor to handle the shared property name

show() is overridden in Student, but still calls super().show() to reuse base logic and then adds more info

Instead of redefining name in the Student class, we inherit it from Person

Student("Ali", "A") uses both base and subclass arguments

## 7. BankAccount and SavingsAccount

class BankAccount:  
 def \_\_init\_\_(self, balance):  
 self.balance = balance  
  
 def show\_balance(self):  
 print(f"Balance: ₹{self.balance}")  
  
class SavingsAccount(BankAccount):  
 def \_\_init\_\_(self, balance, interest):  
 super().\_\_init\_\_(balance)  
 self.interest = interest  
  
 def show\_balance(self):  
 super().show\_balance()  
 print(f"Interest Rate: {self.interest}%")  
  
acc = SavingsAccount(10000, 5)  
acc.show\_balance()

EXPLANATION

super().show\_balance() allows us to reuse the parent’s method and extend it

The constructor handles both shared (balance) and unique (interest) fields

super().\_\_init\_\_(balance) calls the parent constructor

**self.interest = interest   
in child class** ie adds a new attribute interest, which is specific to SavingsAccount

acc = SavingsAccount(10000, 5) and method call acc.show\_balance()

## 8. Product and ElectronicProduct

class Product:  
 def \_\_init\_\_(self, name):  
 self.name = name  
  
 def details(self):  
 print(f"Product: {self.name}")  
  
class ElectronicProduct(Product):  
 def \_\_init\_\_(self, name, warranty):  
 super().\_\_init\_\_(name)  
 self.warranty = warranty  
  
 def details(self):  
 super().details()  
 print(f"Warranty: {self.warranty} years")  
  
p = ElectronicProduct("Laptop", 2)  
p.details()

EXPLANATION

**super().\_\_init\_\_(name)**: Calls the base class constructor to initialize the common attribute name for both product types.

**self.warranty = warranty adds subclass-specific info**Shows how child classes can define their own additional attributes.

Extends the base method using super().details() and adds warranty info.

The Product class handles the common part (name), while ElectronicProduct adds a specific feature (warranty).

p = ElectronicProduct("Laptop", 2) creates a subclass object

## 9. Animal and Dog

class Animal:  
 def sound(self):  
 print("Animal sound")  
  
class Dog(Animal):  
 def sound(self):  
 super().sound()  
 print("Dog barks")  
  
d = Dog()  
d.sound()

EXPLANATION

Inheritance is used where Dog inherits the behavior of Animal.

super().sound() allows the child class to use the base class's method before adding its own behavior.

Method overriding enables the subclass to provide a specific implementation.

Demonstrates polymorphism: Dog has its own version of sound().

## When we create a Dog object and call sound(), both the parent and child class behaviors are executed

## 10. Book and EBook

class Book:  
 def \_\_init\_\_(self, title):  
 self.title = title  
  
 def show(self):  
 print(f"Title: {self.title}")  
  
class EBook(Book):  
 def \_\_init\_\_(self, title, file\_size):  
 super().\_\_init\_\_(title)  
 self.file\_size = file\_size  
  
 def show(self):  
 super().show()  
 print(f"File Size: {self.file\_size} MB")  
  
eb = EBook("Python Guide", 5)  
eb.show()

EXPLANATION

EBook inherits from Book and adds file\_size as a digital-specific attribute.

**super().\_\_init\_\_(title) ensures reuse of base constructor.** This avoids repeating the initialization logic for title.

The subclass method calls super().show() to print the title, then adds file size info

On calling eb.show(), both title and file\_size are printed, showing combined use of parent and child data.

The show() method in EBook doesn't replace the original logic but **builds upon it**