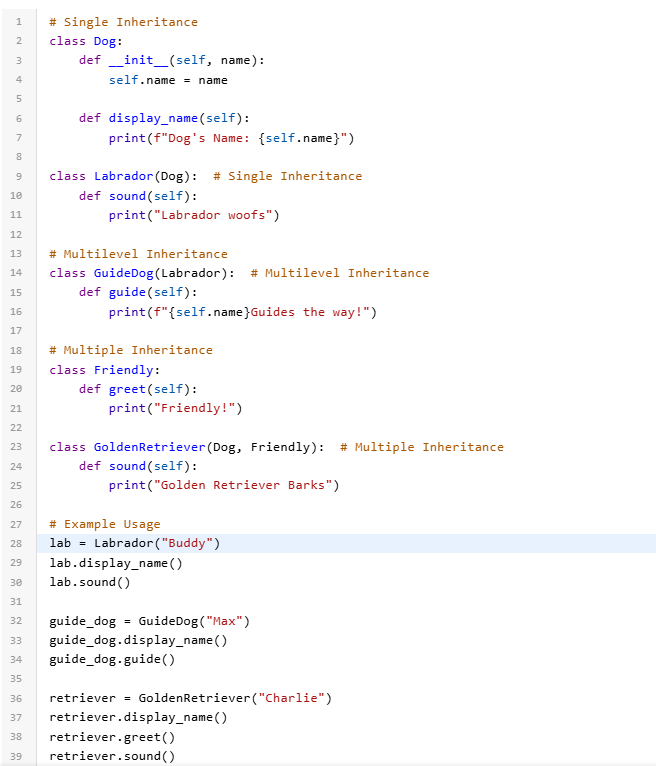
Python Inheritance

Inheritance allows a class (child class) to acquire properties and methods of another class (parent class). It supports hierarchical classification and promotes code reuse.

**Types of Inheritance:**

1. **Single Inheritance:** A child class inherits from a single parent class.
2. **Multiple Inheritance:**A child class inherits from more than one parent class.
3. **Multilevel Inheritance:** A child class inherits from a parent class, which in turn inherits from another class.
4. **Hierarchical Inheritance:** Multiple child classes inherit from a single parent class.
5. **Hybrid Inheritance:** A combination of two or more types of inheritance.



Inheritance is a fundamental concept in [object-oriented programming](https://www.geeksforgeeks.org/python-oops-concepts/)(OOP) that allows a class (called a child or derived class) to inherit attributes and methods from another class (called a parent or base class). This promotes code reuse, modularity, and a hierarchical class structure. In this article, we’ll explore inheritance in [Python](https://www.geeksforgeeks.org/python-programming-language-tutorial/).

**Explanation of Python Inheritance Syntax**

1. **Parent Class**:
   * This is the base class from which other classes inherit.
   * It contains attributes and methods that the child class can reuse.
2. **Child Class:**
   * This is the derived class that inherits from the parent class.
   * The syntax for inheritance is class ChildClass(ParentClass).
   * The child class automatically gets all attributes and methods of the parent class unless overridden.

## Creating a Parent Class

In object-oriented programming, a parent [class](https://www.geeksforgeeks.org/python-classes-and-objects/)(also known as a base class) defines common attributes and methods that can be inherited by other classes. These attributes and methods serve as the foundation for the child classes. By using inheritance, child classes can access and extend the functionality provided by the parent class.

**Creating a Child Class**

A child class (also known as a subclass) is a class that inherits properties and methods from its parent class. The child class can also introduce additional attributes and methods, or even override the ones inherited from the parent.

**super() Function**

[super() function](https://www.geeksforgeeks.org/python-super/) is used to call the parent class’s methods. In particular, it is commonly used in the child class’s \_\_init\_\_() method to initialize inherited attributes. This way, the child class can leverage the functionality of the parent class.

A screenshot of a computer code

Description automatically generated

**Add Properties**

Once inheritance is established, both the parent and child classes can have their own properties. Properties are attributes that belong to a class and are used to store data.

**Explanation:(above program)**

* Person class has properties name and **idnumber**.
* Employee class adds properties salary and post.
* The properties are initialized when an object is created, and they represent the specific data related to the Person and Employee.

**Super()**

In [Python](https://www.geeksforgeeks.org/python-programming-language/), the super() function is used to refer to the parent class or superclass. It allows you to call methods defined in the superclass from the subclass, enabling you to extend and customize the functionality inherited from the parent class.

A method from a parent class can be called in Python using the super() function. It’s typical practice in [object-oriented programming](https://www.geeksforgeeks.org/introduction-of-object-oriented-programming/) to call the methods of the superclass and enable method overriding and inheritance. Even if the current class has replaced those methods with its own implementation, calling super() allows you to access and use the parent class’s methods. By doing this, you may enhance and modify the parent class’s behavior while still gaining from it.

**Benefits of Super Function**

* Need not remember or specify the parent class name to access its [methods](https://www.geeksforgeeks.org/list-methods-in-python/). This function can be used both in single and multiple inheritances.
* This implements modularity (isolating changes) and code reusability as there is no need to rewrite the entire function.
* The super function in Python is called dynamically because Python is a dynamic language, unlike other languages.

**Understanding Python super() with \_\_init\_\_() methods**

Python has a reserved method called “\_\_init\_\_.” In Object-Oriented Programming, it is referred to as a constructor. When this method is called it allows the class to initialize the attributes of the class. In an inherited subclass, a parent class can be referred to with the use of the super() function. The super function returns a temporary object of the superclass that allows access to all of its methods to its child class.

**Note:** For more information, refer to [Inheritance in Python](https://www.geeksforgeeks.org/inheritance-in-python/).

**Super function with Single Inheritance**

Let’s take the example of animals. Dogs, cats, and cows are part of animals. They also share common characteristics like –

* They are mammals.
* They have a tail and four legs.
* They are domestic animals.

So, the classes dogs, cats, and horses are a subclass of animal class. This is an example of single inheritance because many subclasses is inherited from a single-parent class.

*# Python program to demonstrate*

*# super function*

**class** **Animals**:

*# Initializing constructor*

**def** \_\_init\_\_(self):

self.legs = 4

self.domestic = **True**

self.tail = **True**

self.mammals = **True**

**def** isMammal(self):

**if** self.mammals:

print("It is a mammal.")

**def** isDomestic(self):

**if** self.domestic:

print("It is a domestic animal.")

**class** **Dogs**(Animals):

**def** \_\_init\_\_(self):

super().\_\_init\_\_()

**def** isMammal(self):

super().isMammal()

**class** **Horses**(Animals):

**def** \_\_init\_\_(self):

super().\_\_init\_\_()

**def** hasTailandLegs(self):

**if** self.tail **and** self.legs == 4:

print("Has legs and tail")

*# Driver code*

Tom = Dogs()

Tom.isMammal()

Bruno = Horses()

Bruno.hasTailandLegs()

**Output :**

It is a mammal.  
Has legs and tail

**Super with Multiple Inheritances**

Let’s take another **example of a super function**, Suppose a class **can fly** and **can swim** inherit from a mammal class and these classes are inherited by the animal class. So the animal class inherits from the multiple base classes. Let’s see the use of [**Python**](https://www.geeksforgeeks.org/python-programming-language/)**super with arguments** in this case.

**class** **Mammal**():

**def** \_\_init\_\_(self, name):

print(name, "Is a mammal")

**class** **canFly**(Mammal):

**def** \_\_init\_\_(self, canFly\_name):

print(canFly\_name, "cannot fly")

*# Calling Parent class*

*# Constructor*

super().\_\_init\_\_(canFly\_name)

**class** **canSwim**(Mammal):

**def** \_\_init\_\_(self, canSwim\_name):

print(canSwim\_name, "cannot swim")

super().\_\_init\_\_(canSwim\_name)

**class** **Animal**(canFly, canSwim):

**def** \_\_init\_\_(self, name):

super().\_\_init\_\_(name)

*# Driver Code*

Carol = Animal("Dog")

**Output :**

The class Animal inherits from two-parent classes – canFly and canSwim. So, the subclass instance Carol can access both of the parent class constructors.

Dog cannot fly  
Dog cannot swim  
Dog Is a mammal

**Super with Multi-Level Inheritance**

Let’s take another **example of a super function**, suppose a class can swim is inherited by canFly, canFly from the mammal class. So the mammal class inherits from the Multi-Level inheritance. Let’s see the use of [**Python**](https://www.geeksforgeeks.org/python-programming-language/)**super with arguments** in this case.

**class** **Mammal**():

**def** \_\_init\_\_(self, name):

print(name, "Is a mammal")

**class** **canFly**(Mammal):

**def** \_\_init\_\_(self, canFly\_name):

print(canFly\_name, "cannot fly")

*# Calling Parent class*

*# Constructor*

super().\_\_init\_\_(canFly\_name)

**class** **canSwim**(canFly):

**def** \_\_init\_\_(self, canSwim\_name):

print(canSwim\_name, "cannot swim")

super().\_\_init\_\_(canSwim\_name)

**class** **Animal**(canSwim):

**def** \_\_init\_\_(self, name):

*# Calling the constructor*

*# of both the parent*

*# class in the order of*

*# their inheritance*

super().\_\_init\_\_(name)

*# Driver Code*

Carol = Animal("Dog")

**Output :**

Dog cannot swim  
Dog cannot fly  
Dog Is a mammal

**MRO stands for Method Resolution Order. It is the order in which Python looks for a method in a hierarchy of classes.**

**Method Overriding in Python**

**Method overriding is an ability of any object-oriented programming language that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its super-classes or parent classes. When a method in a subclass has the same name, the same parameters or signature, and same return type(or sub-type) as a method in its super-class, then the method in the subclass is said to override the method in the super-class.**

A screenshot of a computer

Description automatically generated

A screenshot of a computer program

Description automatically generated

A screenshot of a computer program

Description automatically generated

A screenshot of a computer program

Description automatically generated

A screenshot of a computer program

Description automatically generated

A screenshot of a computer program

Description automatically generated

A screenshot of a computer program

Description automatically generated

**Python Polymorphism**

Polymorphism allows methods to have the same name but behave differently based on the object’s context. It can be achieved through method overriding or overloading.

**Types of Polymorphism**

1. **Compile-Time Polymorphism**: This type of polymorphism is determined during the compilation of the program. It allows methods or operators with the same name to behave differently based on their input parameters or usage. It is commonly referred to as method or operator overloading.
2. **Run-Time Polymorphism**: This type of polymorphism is determined during the execution of the program. It occurs when a subclass provides a specific implementation for a method already defined in its parent class, commonly known as method overriding.

A screenshot of a computer program

Description automatically generated

**Explanation:**

**1. Run-Time Polymorphism:**

* Demonstrated using method overriding in the Dog class and its subclasses (Labrador and Beagle).
* The correct sound method is invoked at runtime based on the actual type of the object in the list.

**2. Compile-Time Polymorphism:**

* Python does not natively support method overloading. Instead, we use a single method (add) with default arguments to handle varying numbers of parameters.
* Different behaviors (adding two or three numbers) are achieved based on how the method is called.

A screenshot of a computer code

Description automatically generated

Duck Typing is a [type system](https://www.geeksforgeeks.org/type-systemsdynamic-typing-static-typing-duck-typing/) used in dynamic languages. For example, Python, Perl, Ruby, PHP, Javascript, etc. where the type or the class of an object is less important than the method it defines. Using Duck Typing, we do not check types at all. Instead, we check for the presence of a given method or attribute.

**Polymorphism in OOPs**

In [OOP](https://www.geeksforgeeks.org/python-oops-concepts/), polymorphism allows methods in different [classes](https://www.geeksforgeeks.org/python-classes-and-objects/)to share the same name but perform distinct tasks. This is achieved through inheritance and interface design. Polymorphism complements other OOP principles like [inheritance](https://www.geeksforgeeks.org/inheritance-in-python/)(sharing behavior) and encapsulation (hiding complexity) to create robust and modular applications.

A screenshot of a computer program

Description automatically generated

* **Polymorphic Behavior:** A list of shape objects (Rectangle and Circle) is created. A for loop iterates through the list, calling the area method on each object. The method executed is determined by the object’s type, showcasing polymorphism.

**Types of Polymorphism**

**Compile-time Polymorphism**

* Found in statically typed languages like Java or C++, where the behavior of a function or operator is resolved during the program’s compilation phase.
* Examples include [method overloading](https://www.geeksforgeeks.org/python-method-overloading/) and operator overloading, where multiple functions or operators can share the same name but perform different tasks based on the context.
* In Python, which is dynamically typed, compile-time polymorphism is not natively supported. Instead, Python uses techniques like dynamic typing and duck typing to achieve similar flexibility.

**Runtime Polymorphism**

* Occurs when the behavior of a method is determined at runtime based on the type of the object.
* In Python, this is achieved through [method overriding](https://www.geeksforgeeks.org/method-overriding-in-python/): a child class can redefine a method from its parent class to provide its own specific implementation.
* Python’s dynamic nature allows it to excel at runtime polymorphism, enabling flexible and adaptable code.

A screenshot of a computer program

Description automatically generated

**Inheritance Class Polymorphism**

Inheritance-based polymorphism occurs when a subclass overrides a method from its parent class, providing a specific implementation. This process of re-implementing a method in the child class is known as [**Method Overriding**](https://www.geeksforgeeks.org/method-overriding-in-python/).

A screenshot of a computer program

Description automatically generated

**Python Encapsulation**

Encapsulation is the bundling of data (attributes) and methods (functions) within a class, restricting access to some components to control interactions.

A class is an example of encapsulation as it encapsulates all the data that is member functions, variables, etc.

**Types of Encapsulation:**

1. **Public Members**: Accessible from anywhere.
2. **Protected Members**: Accessible within the class and its subclasses.
3. **Private Members**: Accessible only within the class.

A screenshot of a computer program

Description automatically generated

A screenshot of a computer

Description automatically generated

https://www.geeksforgeeks.org/getter-and-setter-in-python/

**Abstraction**

difference between **concept** and **implementation**.

1. **Conceptual meaning of abstraction:**
   * Abstraction refers to **hiding the complex details** and **showing only the relevant information**.
   * It’s about defining an interface (what something does) without specifying how it does it.
   * This allows you to focus on what an object can do rather than how it does it.
2. **Practical implementation in Python (abstract classes):**
   * Python uses **abstract classes** (via the abc module) to enforce that certain methods must be defined in child classes.
   * An abstract class can have **abstract methods** (methods with no body) and sometimes **concrete methods** (methods with a body).

**Why does this look disconnected from the "hiding data" idea?**

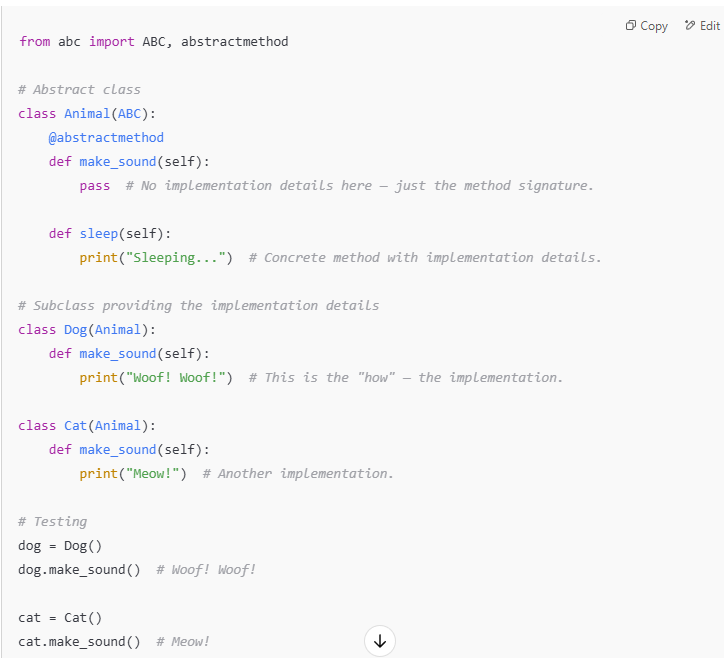
* The "hiding data" part of abstraction is more about **interfaces and design** rather than the literal hiding of data members.
* Abstract classes focus on hiding the "implementation details" by ensuring that only method signatures (what must be done) are exposed, while the "how to do it" part is left for subclasses.

**hat are Implementation Details?**

* **Implementation details** refer to the **actual code or logic** that defines how a method works — the step-by-step process to achieve a result.
* It’s the **inner workings** or **specific steps** that make a method or function do what it does.

**🚀 In Abstract Classes:**

When we say abstract classes hide the "implementation details," we mean they **only define the method names and expected behavior** (the *what*) but **leave the actual code (the *how*) for subclasses to provide**.



**🔥 Breaking it down:**

* **Abstract method (make\_sound)**:  
  Defined in Animal, but it has **no implementation** — just the method signature (def make\_sound(self):).  
  It answers: **What should an Animal be able to do?** (make a sound)
* **Implementation details**:  
  The **subclasses** (Dog, Cat) provide the "how" — the specific sound each animal makes.  
  Each subclass implements its version of make\_sound().

**🎯 Why is this useful?**

* It **enforces a structure** — every subclass **must** implement make\_sound().
* The abstract class provides a consistent interface without dictating how the subclasses work internally.
* This lets you **hide the irrelevant details** of how each animal makes a sound and focus only on the fact that **all animals can make a sound**.