

Dr. Francis Jesmar P. Montalbo

Python Inheritance

Outline

- Python Inheritance
- Types of Inheritance
- The super method
- Method Resolution Order (MRO)
- Super vs Explicit inheritance

Python Inheritance

- Allows us to define a class that inherits all the methods(behaviors) and properties(states) from another class.
- Parent class is the class being inherited from, also called base class.
- Child class is the class that inherits from another class, also called derived class.

Create a Parent Class

```
class Person:
    def __init__(self, fname, lname):
        self.firstname = fname
        self.lastname = lname

    def printname(self):
        print(self.firstname, self.lastname)

#Use the Person class to create an object, and then execute the printname method:

x = Person("John", "Doe")
x.printname()
```

- Any class can be a parent class, so the syntax is the same as creating any other class:
- Create a class named **Person**, with **firstname** and **lastname** properties, and a **printname** method:

Create a Child Class

- To create a class that inherits the functionality from another class, send the parent class as a parameter when creating the child class:
- Create a class named **Student**, which will inherit the properties(states) and methods(behaviors) from the **Person** class:

```
class Student(Person):  
    pass
```

```
x = Student("Mike", "Olsen")  
x.printname()
```

Execute inheritance

- Now the Student class has the same properties and methods as the Person class.
- Use the **Student** class to create an **object**, and then execute the **printname** method:

```
class Student(Person):  
    def __init__(self, fname, lname):  
        #add properties etc.
```

Add the
__init__()
Function

- A child class that inherits the **properties** and **methods** from its **parent**.
- Add the `__init__()` function can be added to the **child** class.
- Add the `__init__()` function to the **Student** class:
- When you add the `__init__()` function, the **child class will no longer inherit the parent's `__init__()` function**.
- The **child's `__init__()` function overrides the inheritance of the parent's `__init__()` function**.

```
class Student(Person):  
    def __init__(self, fname, lname):  
        Person.__init__(self, fname, lname)
```

Retaining the inheritance

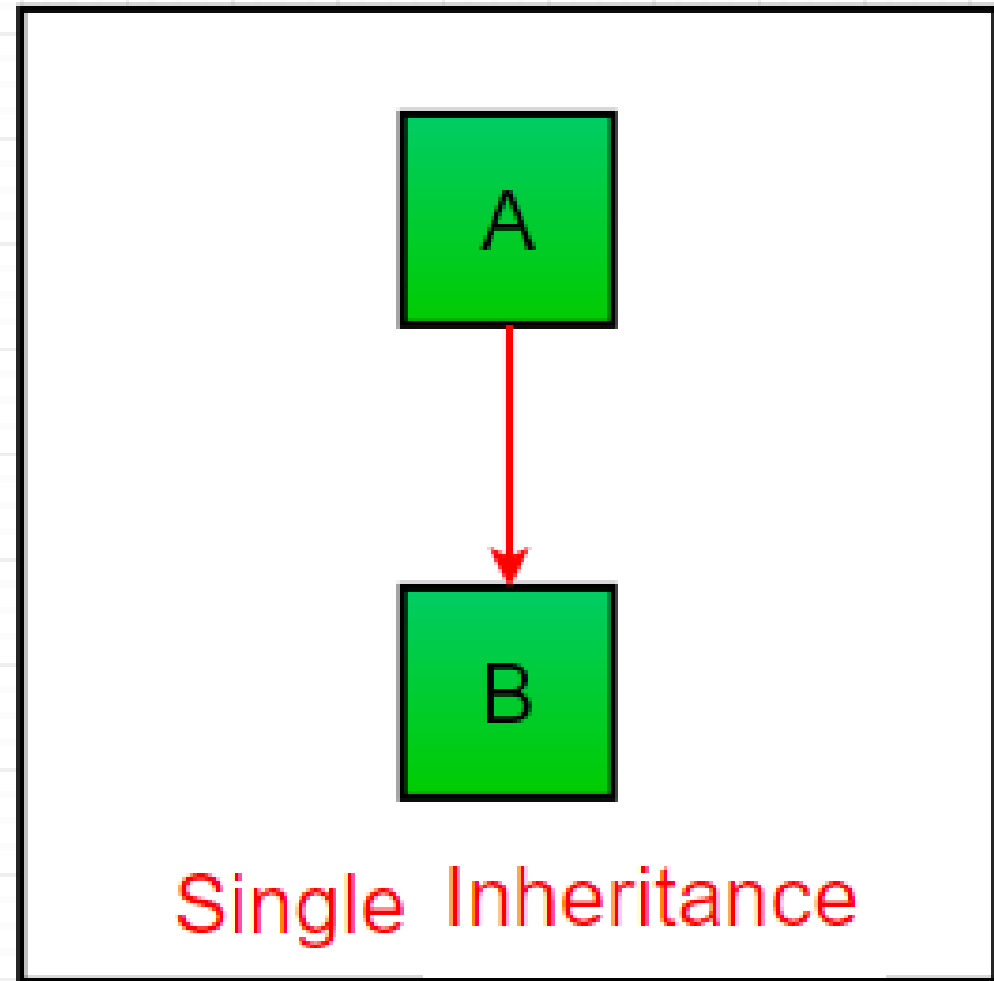
- To keep the inheritance of the parent's `__init__()` function, add a call to the parent's `__init__()` function:
- The example adds the `__init__()` function, and kept the inheritance of the parent class, and we are ready to add functionality in the `__init__()` function.

Types of Inheritance in Python



Single Inheritance

- Enables a derived class to inherit properties from a single parent class, thus enabling code reusability and the addition of new features to existing code.





Sample single inheritance

```
# Python program to demonstrate
# single inheritance

# Base class
class Parent:
    def func1(self):
        print("This function is in parent class.")

# Derived class

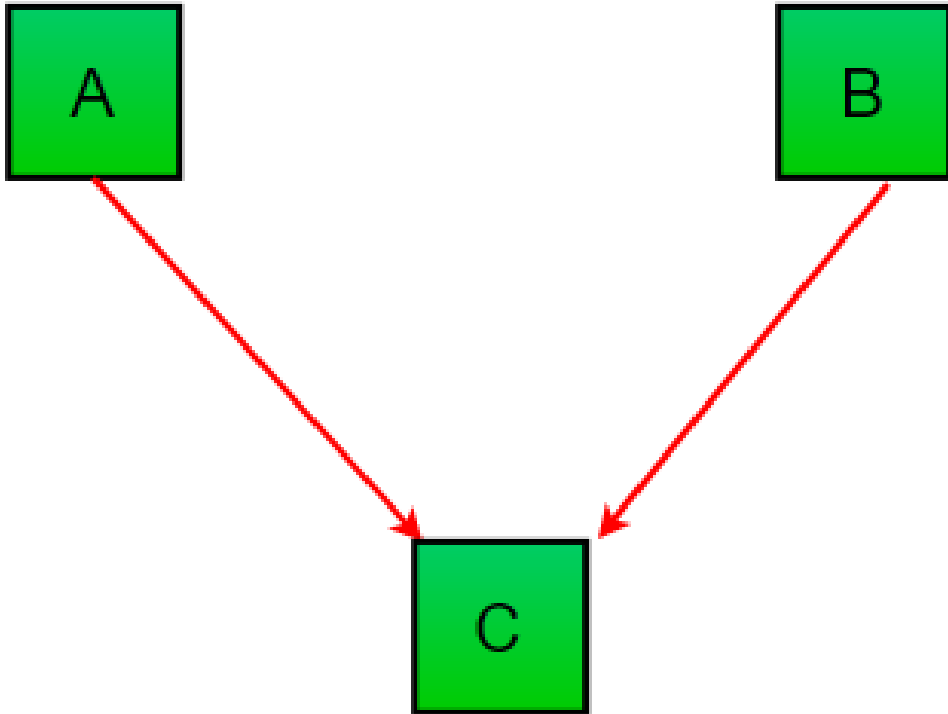
class Child(Parent):
    def func2(self):
        print("This function is in child class.")

# Driver's code
object = Child()
object.func1()
object.func2()
```

```
This function is in parent class.
This function is in child class.
```

Multiple inheritance

- When a class can be derived from more than one base class this type of inheritance is called multiple inheritances. In multiple inheritances, all the features of the base classes are inherited into the derived class.



Multiple Inheritance

Multiple Inheritance sample

```
# Python program to demonstrate
# multiple inheritance

# Base class1
class Mother:
    mothername = ""

    def mother(self):
        print(self.mothername)

# Base class2
class Father:
    fathername = ""

    def father(self):
        print(self.fathername)

# Derived class
class Son(Mother, Father):
    def parents(self):
        print("Father :", self.fathername)
        print("Mother :", self.mothername)

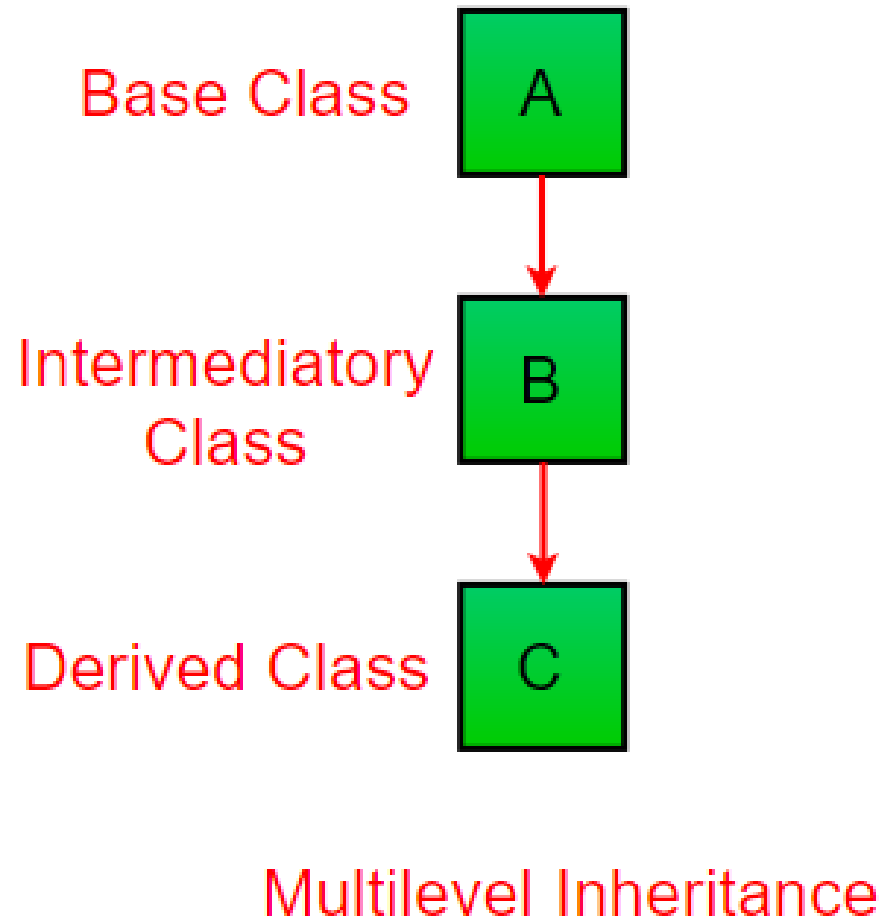
# Driver's code
s1 = Son()
s1.fathername = "RAM"
s1.mothername = "SITA"
s1.parents()
```

Father : RAM

Mother : SITA

Multilevel inheritance

- In multilevel inheritance, features of the base class and the derived class are further inherited into the new derived class. This is similar to a relationship representing a child and a grandfather.



Multilevel inheritance sample

```
# Python program to demonstrate
# multilevel inheritance

# Base class

class Grandfather:

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

# Intermediate class

class Father(Grandfather):
    def __init__(self, fathername, grandfathername):
        self.fathername = fathername

        # invoking constructor of Grandfather class
        Grandfather.__init__(self, grandfathername)

# Derived class

class Son(Father):
    def __init__(self, sonname, fathername, grandfathername):
        self.sonname = sonname

        # invoking constructor of Father class
        Father.__init__(self, fathername, grandfathername)

    def print_name(self):
        print('Grandfather name :', self.grandfathername)
        print("Father name :", self.fathername)
        print("Son name :", self.sonname)

# Driver code
s1 = Son('Prince', 'Rampal', 'Lal mani')
print(s1.grandfathername)
s1.print_name()
```

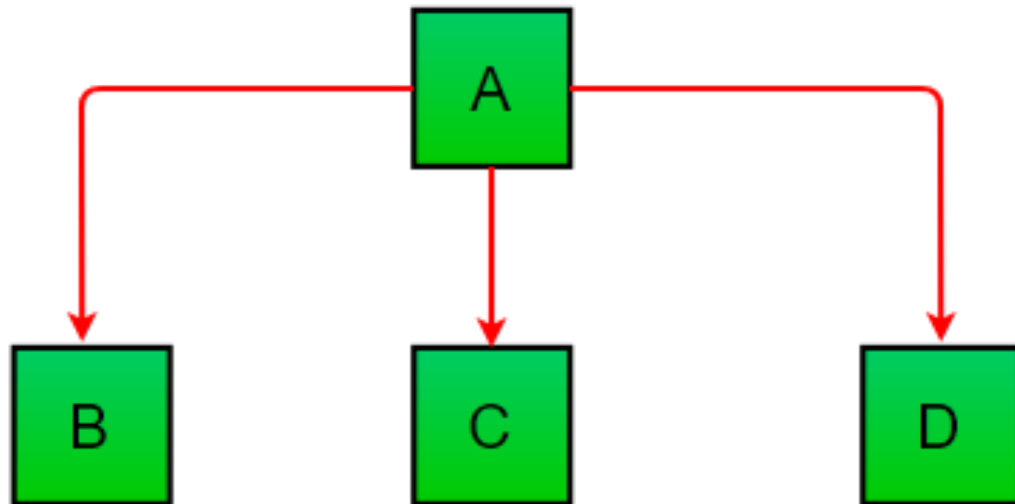
Lal mani

Grandfather name : Lal mani

Father name : Rampal

Son name : Prince

Hierarchical inheritance



Hierarchical Inheritance

- When more than one derived class are created from a single base this type of inheritance is called hierarchical inheritance. In this program, we have a parent (base) class and two child (derived) classes.

Hierarchical Inheritance sample

```
# Python program to demonstrate
# Hierarchical inheritance

# Base class
class Parent:
    def func1(self):
        print("This function is in parent class.")

# Derived class1

class Child1(Parent):
    def func2(self):
        print("This function is in child 1.")

# Derived class2

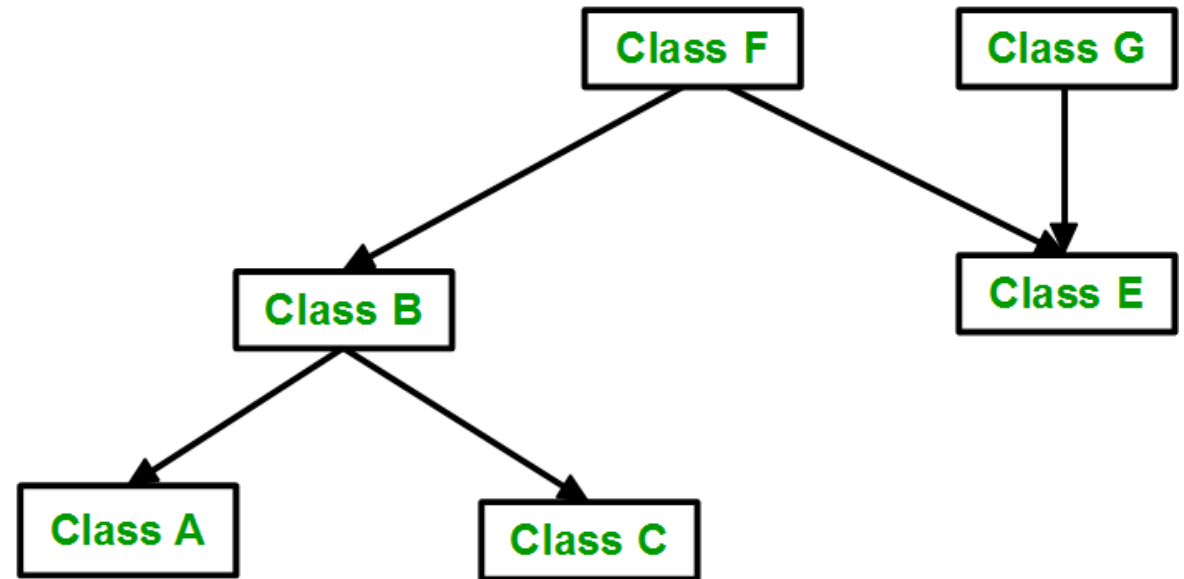
class Child2(Parent):
    def func3(self):
        print("This function is in child 2.")

# Driver's code
object1 = Child1()
object2 = Child2()
object1.func1()
object1.func2()
object2.func1()
object2.func3()
```

```
This function is in parent class.
This function is in child 1.
This function is in parent class.
This function is in child 2.
```

Inheritance consisting of multiple types of inheritance is called hybrid inheritance.

Hybrid inheritance



Hybrid inheritance sample

```
# Python program to demonstrate
# hybrid inheritance

class School:
    def func1(self):
        print("This function is in school.")

class Student1(School):
    def func2(self):
        print("This function is in student 1. ")

class Student2(School):
    def func3(self):
        print("This function is in student 2.")

class Student3(Student1, School):
    def func4(self):
        print("This function is in student 3.")

# Driver's code
object = Student3()
object.func1()
object.func2()
```

```
This function is in school.
This function is in student 1.
```

The super method

Definition and Usage

- The `super()` function is used to give access to methods and properties of a parent or sibling class.
- Allows us to avoid using the base class name explicitly
- The `super()` function returns an object that represents the parent class.
- Working with Multiple Inheritance

super()

- Creates a class that will inherit all the methods and properties from another class:

```
class Parent:
    def __init__(self, txt):
        self.message = txt

    def printmessage(self):
        print(self.message)
```

```
class Child(Parent):
    def __init__(self, txt):
        super().__init__(txt)
```

```
x = Child("Hello, and welcome!")
```

```
x.printmessage()
```

Hello, and welcome!

Super() Function

- Python **super() function** will make the child class inherit all the methods and properties from its parent:
- The **super()** function now takes the **__init__()** of the **Person** class.
- The **super()** function also prevents inconsistencies of reference when inheriting from a parent class.

```
class Student(Person):  
    def __init__(self, fname, lname):  
        super().__init__(fname, lname)
```



Person properties/states

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

    def drive(self):
        print(f"{self.name} is driving at {self.speed} mph.")

class Car(Vehicle):
    def __init__(self, name, speed, color):
        super().__init__(name, speed)
        self.color = color

    def honk(self):
        print(f"{self.name} is honking its horn in {self.color} color.")

my_car = Car("Tesla Model S", 150, "red")
my_car.drive() # calling the method from the parent class
my_car.honk()
```

Example using super()

- The super() builtin returns a proxy object (temporary object of the superclass) that allows us to access methods of the base class.


```
class Animal:
    def __init__(self, name):
        self.name = name
        self.health = 100

    def eat(self):
        self.health += 10
        print(f"{self.name} is eating. Health is now {self.health}.")

class Dog(Animal):
    def __init__(self, name):
        super().__init__(name)
        self.breed = "Unknown"

    def bark(self):
        print("Woof woof!")

    def wag_tail(self):
        print(f"{self.name} is wagging its tail.")

dog1 = Dog("Buddy")
dog1.eat() # calling the method from the Animal class using super()
dog1.bark()
dog1.wag_tail()
```

Using super() in single inheritance

- In the case of single inheritance, we use super() to refer to the base class.

Method Resolution Order (MRO)

Method Resolution Order(MRO)

- Denotes the way a programming language resolves a method or attribute.
- Python supports classes inheriting from other classes. The class being inherited is called the **Parent** or **Superclass**, while the class that inherits is called the **Child** or **Subclass**.
- In python, MRO defines the order in which the **base classes are searched** when executing a method.
- First, the **method or attribute is searched within a class** and then it **follows the order we specified while inheriting**. This **order is also called Linearization of a class** and **set of rules** are called **MRO**.
- While inheriting from another class, the interpreter needs a way to resolve the methods that are being called via an instance.

Python MRO Sample

```
# Python program showing  
# how MRO works  
  
class A:  
    def rk(self):  
        print(" In class A")  
class B(A):  
    def rk(self):  
        print(" In class B")  
  
r = B()  
r.rk()
```

In class B

Explanation

- In the example the methods that are invoked is **from class B** but **NOT** from class A, and this is due to **MRO**.
- The order that follows in the above code is class B –> class A
- In multiple inheritances, the methods are executed based on the order specified while inheriting the classes.
- For the languages that support single inheritance, method resolution order is not interesting, but the languages that support multiple inheritance method resolution order plays a very crucial role.

Additional MRO Example

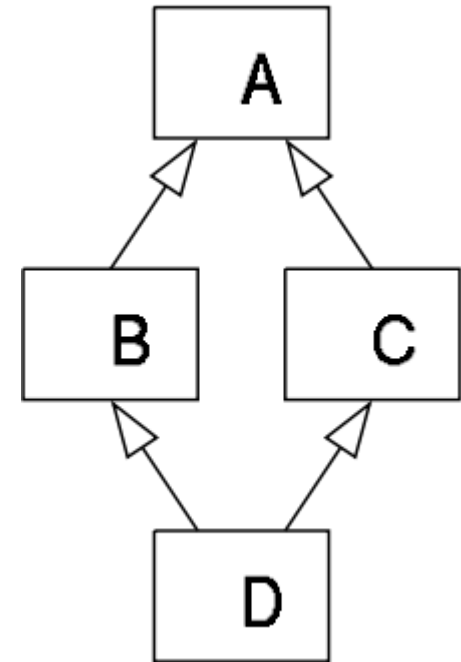
```
# Python program showing
# how MRO works

class A:
    def rk(self):
        print(" In class A")
class B(A):
    def rk(self):
        print(" In class B")
class C(A):
    def rk(self):
        print("In class C")

# classes ordering
class D(B, C):
    pass

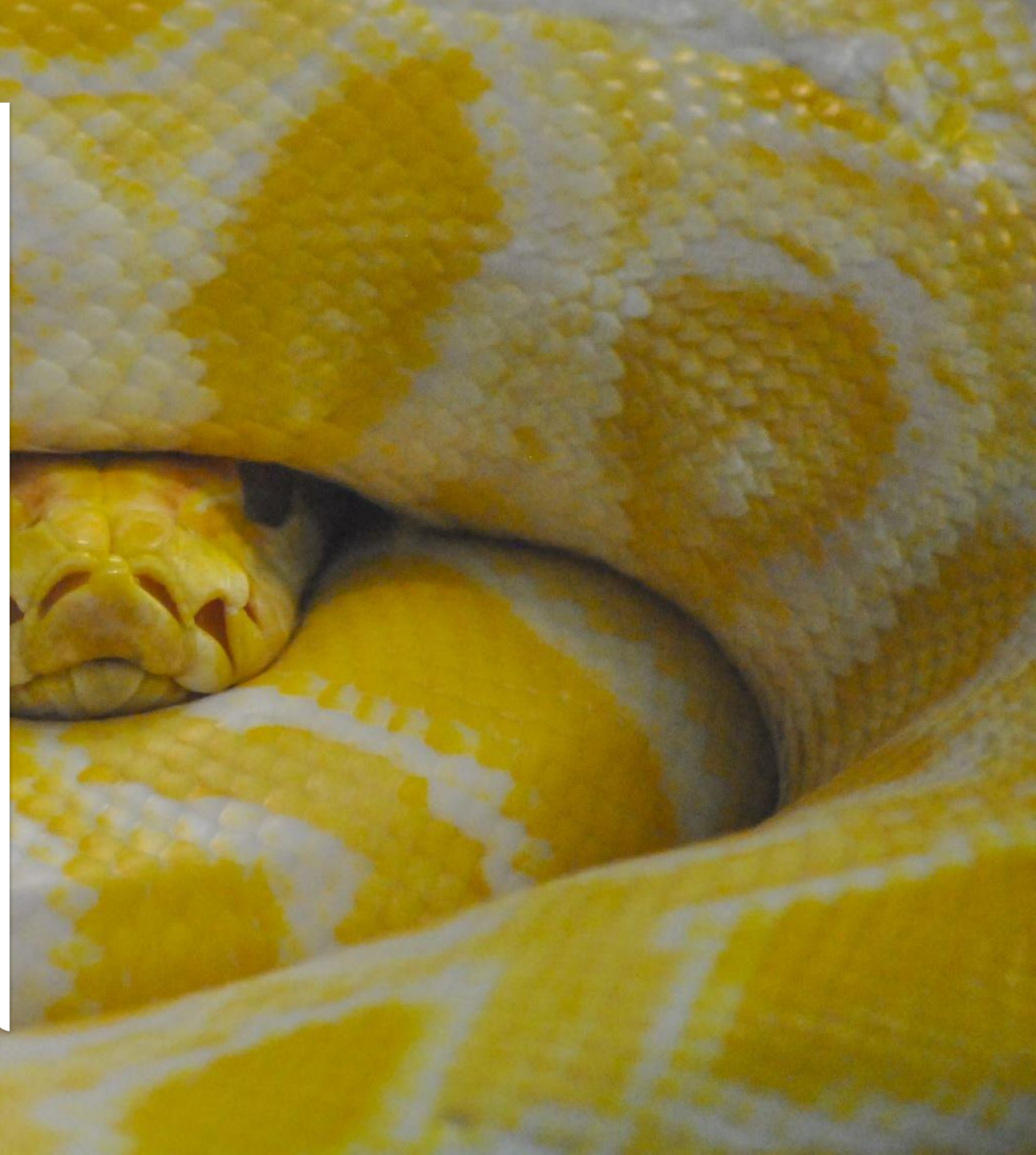
r = D()
r.rk()
```

In class B



Diamond Inheritance Explanation

- Python follows a **depth-first lookup** order and hence ends up calling the method from class A. By following the method resolution order, the lookup order as follows.
- Class D -> Class B -> Class C -> Class A
- Python follows depth-first order to resolve the methods and attributes.
- Hence, from the previous example, it executes the method in class B.



REMINDER

MRO is from bottom to top and left to right. It's a Depth-First-Search (DFS) Algorithm

The method is searched in the class of the object.

If it's not found, it is searched in the immediate super class.

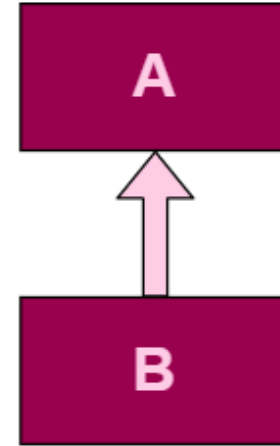
In the case of multiple super classes, it is searched left to right, in the order by which was declared by the developer.

More examples

- `def class C(B, A)`
- In this case, the MRO would be `C -> B -> A`.
- Since B was mentioned first in the class declaration, it will be searched for first while resolving a method.

More examples

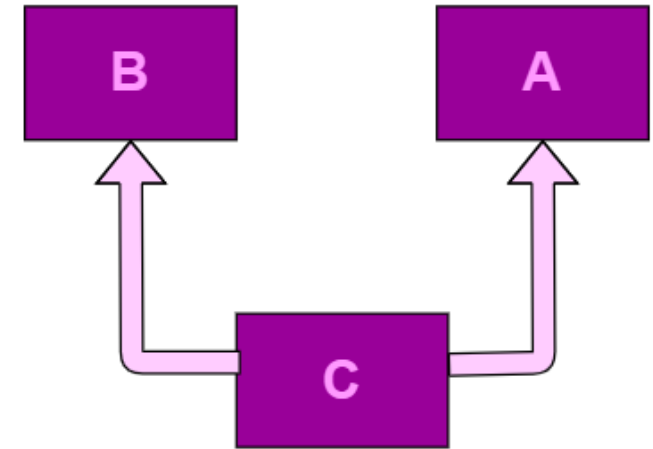
- B -> A
- B.method() called



```
1 class A:
2     def method(self):
3         print("A.method() called")
4
5 class B(A):
6     def method(self):
7         print("B.method() called")
8
9 b = B()
10 b.method()
```

More examples

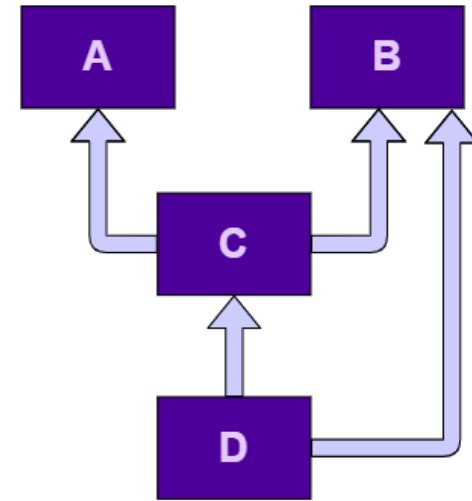
- C -> B -> A
- A.method() called



```
1 class A:
2     def method(self):
3         print("A.method() called")
4
5 class B:
6     pass
7
8 class C(B, A):
9     pass
10
11 c = C()
12 c.method()
```

More examples

- D -> C -> A -> B
- A.method() called



```
1 class A:
2     def method(self):
3         print("A.method() called")
4
5 class B:
6     def method(self):
7         print("B.method() called")
8
9 class C(A, B):
10     pass
11
12 class D(C, B):
13     pass
14
15 d = D()
16 d.method()
```



Super and Inheritance via Explicit `__init__`

- **Method Resolution Order (MRO):** When multiple inheritance is used, `super()` determines the order in which parent classes are searched for methods and attributes using the Method Resolution Order (MRO).
 - The MRO is a linear ordering of the inheritance graph that guarantees that every class's method is called only once.
 - By using `super()`, you can rely on Python's built-in MRO algorithm to determine the correct order of parent classes to call.
 - If you reference the parent class directly, you may need to manually specify the order in which the parent classes are searched.
-

super vs explicit
__init__

Super and Inheritance via Explicit `__init__`

- **Maintainability:** Using `super()` can make your code more maintainable in case the inheritance hierarchy changes in the future. For example, if you add or remove a parent class, you may need to update all the references to the parent class in your code. Using `super()` eliminates the need to make such changes, as long as the new class has the same method signature as the old one.

Super and Inheritance via Explicit `__init__`

- **Flexibility:** Using `super()` can make your code more flexible, as it allows you to override methods in intermediate classes without breaking the inheritance chain. If you reference the parent class directly, you will need to manually call the superclass's method to maintain the inheritance chain.


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

    def eat(self, food):
        print(f"{self.name} started eating {food}.")
        print("its only possible to print this without copying this code, except through super")
```

```
class Dog(Animal):
    def __init__(self, name):
        super().__init__(name)

    def eat(self):
        super().eat(food="dog food")
```

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

    def eat(self):
        print(f"This is a cat named {self.name} eating.")
```

```
dog = Dog("John")
cat = Cat("Missy")
```

```
dog.eat()
cat.eat()
```

```
print(Dog.mro())
```

John started eating dog food.

its only possible to print this without copying this code, except through super

This is a cat named Missy eating.

```
[<class '__main__.Dog'>, <class '__main__.Animal'>, <class 'object'>]
```

In the code, the use of `super()` in the Dog class is preferred over explicitly calling the `eat` method of the Animal class, because it makes the code more flexible and easier to maintain. However, in the Cat class, the `__init__` method is not calling the `__init__` method of the parent class at all, which can cause unexpected behavior if the Animal class has any important initialization logic. So, in this case, it would be better to use `super()` in the `__init__` method of the Cat class to ensure that the parent class is properly initialized.

What should I use?



Super over explicit

- **USING SUPER:**
- **Allows easier maintenance and modification of the class hierarchy.** If you need to change the order of inheritance, or you add or remove a parent class, you won't have to update every reference to the parent class method in every subclass. Instead, you just update the `super()` call in the subclass.
- **Gives a more flexible inheritance.** If you have multiple levels of inheritance, using `super()` ensures that you are calling the correct method in the correct parent class, regardless of the order of the inheritance.
- **Provides a more concise and easier to read code.** When using `super()`, you don't have to repeat the name of the parent class or the method being called, which can make the code more readable and easier to understand.

Important Reason to use Super!

- The reason why we super is that if in case that our parent that is providing inheritance changes, all of the other child classes inheriting from it will also require changes, may it be field names or behavior.
- Explicit makes everything more sophisticated, redundant, highly error prone, slows down production time, and can result to highly difficult maintenance!!
- Super makes it easier to read code, modify, scale, and debug.

Further explainers

- When a child class overrides a method from its parent class, it may still want to use the parent class's implementation of that method. By using `super()`, the child class can call the parent class's method and then add its own modifications.
- When a child class needs to add new behavior to its parent class's method, it can call the parent class's method with `super()` and then add its own modifications before or after the parent class's code.
- When a child class needs to override its parent class's `__init__` method, it can use `super()` to call the parent class's `__init__` method first and then add its own initialization code.

It's a “syntactic sugar”

```
1 # Inheritance without super()
2
3 class Person:
4     def __init__(self, name, age):
5         self.name = name
6         self.age = age
7
8     def say_hello(self):
9         print(f"Hello, my name is {self.name} and I am {self.age} years old.")
10
11 class Student(Person):
12     def __init__(self, name, age, major):
13         Person.__init__(self, name, age)
14         self.major = major
15
16     def say_major(self):
17         print(f"I am studying {self.major}.")
18
19     def say_hello(self):
20         print(f"Hi, I am {self.name}, a {self.age}-year-old student studying {self.major}.")
21         Person.say_hello(self)
22
23 my_student = Student("John", 20, "Computer Science")
24 my_student.say_hello()
25 my_student.say_major()
```

```
1 # Inheritance with super()
2
3 class Person:
4     def __init__(self, name, age):
5         self.name = name
6         self.age = age
7
8     def say_hello(self):
9         print(f"Hello, my name is {self.name} and I am {self.age} years old.")
10
11 class Student(Person):
12     def __init__(self, name, age, major):
13         super().__init__(name, age)
14         self.major = major
15
16     def say_major(self):
17         print(f"I am studying {self.major}.")
18
19     def say_hello(self):
20         print(f"Hi, I am {self.name}, a {self.age}-year-old student studying {self.major}.")
21         super().say_hello()
22
23 my_student = Student("John", 20, "Computer Science")
24 my_student.say_hello()
25 my_student.say_major()
```

Codebase will most likely expand!

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

    def speak(self):
        print(f"{self.name} says: ")

class Dog(Animal):
    def speak(self):
        super().speak()
        print("Woof!")

class Cat(Animal):
    def speak(self):
        super().speak()
        print("Meow!")

class DogCat(Dog, Cat):
    def speak(self):
        super().speak()
        print("Bow-wow-meow!")
```

In this example, we have four classes: Animal, Dog, Cat, and DogCat. Dog and Cat both inherit from Animal. DogCat inherits from both Dog and Cat.

Each class has a speak() method, but Dog and Cat override the parent's speak() method to include the sound they make. DogCat also overrides speak(), but instead of choosing between Dog and Cat, it combines both sounds.

Now imagine we want to add a new class called Bird that inherits from Animal and has its own unique sound. Without super(), we would have to manually call the speak() method of each parent class in order to correctly output the sound of the Bird instance:

```
class Bird(Animal):
    def __init__(self, name, sound):
        super().__init__(name)
        self.sound = sound

    def speak(self):
        Animal.speak(self)
        print(self.sound)
```

We have to manually call Animal.speak(self) to output the animal's name before outputting the bird's sound. If we added more classes and more levels of inheritance, it would become even more complex to correctly call all the parent's methods.

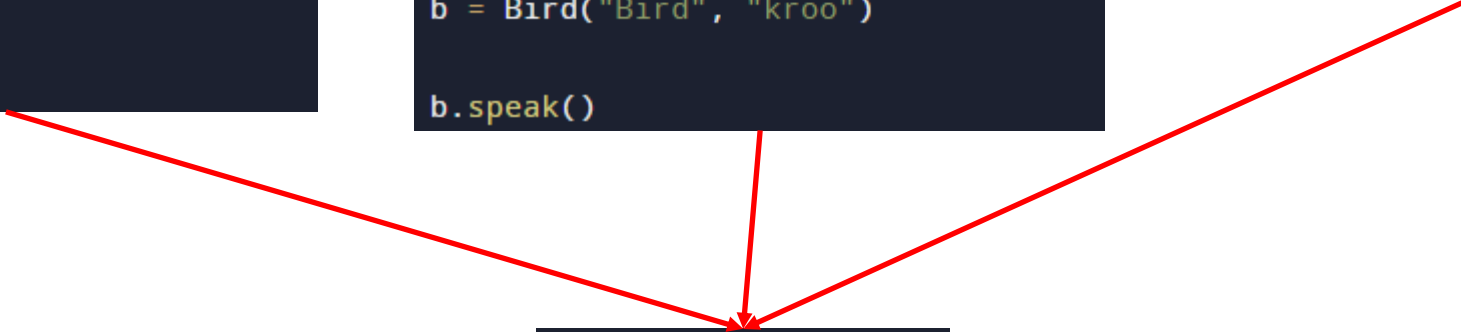
With super(), however, we can simply call super().speak() and it will automatically call the speak() method of the parent class in the order of inheritance, making it much easier to read and maintain:

Without and With super for your inherited methods

```
class Bird(Animal):  
    def __init__(self, name):  
        super().__init__(name)  
  
    def speak(self, sound):  
        Animal.speak(self)  
        print(sound)  
  
b = Bird("Bird")  
  
b.speak("kroo")
```

```
class Bird(Animal):  
    def __init__(self, name, sound):  
        super().__init__(name)  
        self.sound = sound  
  
    def speak(self):  
        Animal.speak(self)  
        print(self.sound)  
  
b = Bird("Bird", "kroo")  
  
b.speak()
```

```
class Bird(Animal):  
    def speak(self):  
        super().speak()  
        print("kroo")  
  
b = Bird("Bird")  
  
b.speak()
```

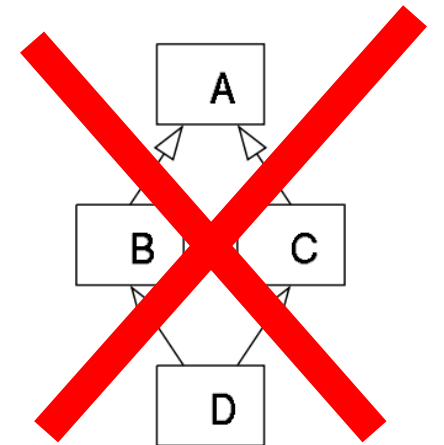


```
Bird says:  
kroo
```


Reminders about Inheritance

REMEMBER

- Even with inheritance's advantages, DO NOT ABUSE THEM, it can or might cause a diamond inheritance problem.
- The "diamond problem" (sometimes referred to as the "**Deadly Diamond of Death**") is an ambiguity that arises when two classes B and C inherit from A, and class D inherits from both B and C.
- You are engineering a software not illustrating an artwork.
- Keep it simple and optimal.



USE MIXINS!

How to solve
this problem?



Next Topics

- More super with other forms of inheritance
- Do's and Don'ts of super()
- Mixins
- Compositions and Aggregations

Try me!

Create an OOP program that exemplifies inheritance in the following slide.

