Multiple Inheritance in Python

This report addresses two aspects of multiple inheritance in Python:

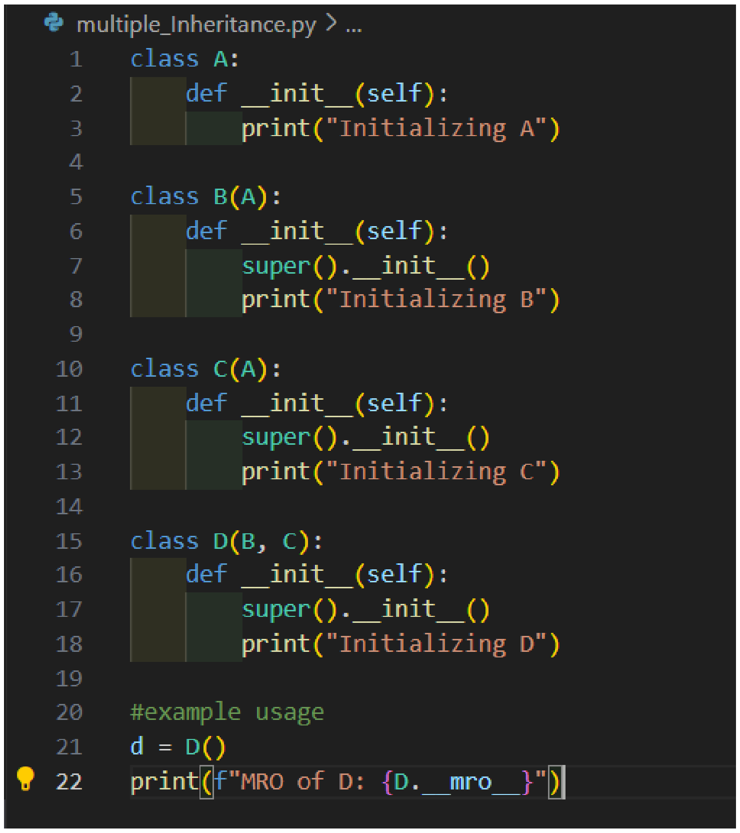
1. How the `super()` function handles multiple inheritance.
2. How Python resolves method calls when multiple parent classes have methods with the same name.

1. How the super() function handles multiple inheritance

In Python, the `super()` function provides a way to call methods from a parent or child class in the method resolution order(MRO). In the context of multiple inheritance, super() is crucial for ensuring that __init__ methods (and other methods) of all parent classes are called correctly and in the right order, avoiding redundant calls and ensuring proper initialization.

Python's MRO determines the order in which base classes are searched when executing a method. For classes with multiple inheritance, python uses the C3 linearization algorithm to determine the MRO. You can inspect the MRO of a class using ClassName.__mro__ or super() help(ClassName).

Let's consider a diamond inheritance problem to illustrate how super() works:

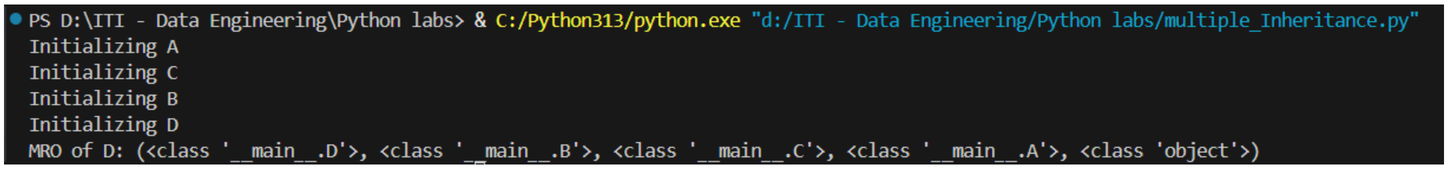


```python
class A:
    def __init__(self):
        print("Initializing A")

class B(A):
    def __init__(self):
        super().__init__()
        print("Initializing B")

class C(A):
    def __init__(self):
        super().__init__()
        print("Initializing C")

class D(B, C):
    def __init__(self):
        super().__init__()
        print("Initializing D")

#example usage
d = D()
print(f"MRO of D: {D.__mro__}")
```

Explanation:

When an instance of D is created (d = D()), the __init__ method of D is called. Inside D.__init__(), super().__init__() is invoked. According to the MRO of D (which is D -> B -> C -> A -> object), super() first calls B.__init__(). Inside B.__init__(), super().__init__() is called again, which then calls C.__init__() (following the MRO). C.__init__() then calls A.__init__() via super(). Finally, A.__init__() is executed, and then the __init__ methods of C, B, and D complete in reverse order of their calls. This ensures that A.__init__() is called only once, and all parent classes are properly initialized.

Output of the example:

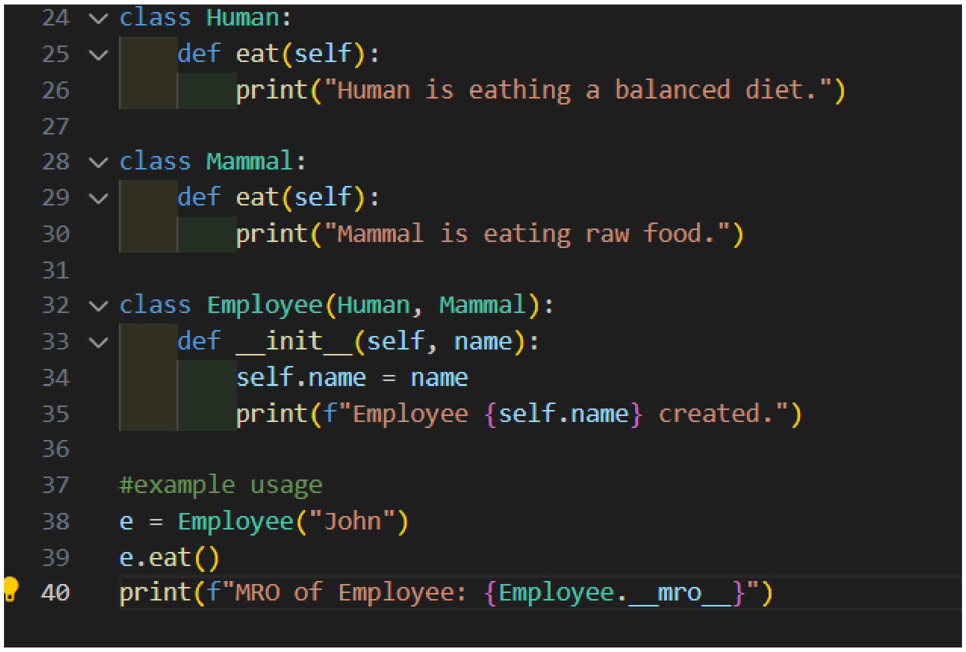


```
PS D:\ITI - Data Engineering\Python labs> & C:/Python313/python.exe "d:/ITI - Data Engineering/Python labs/multiple_Inheritance.py"
Initializing A
Initializing C
Initializing B
Initializing D
MRO of D: (<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__main__.A'>, <class 'object'>)
```

2. How Python resolves method calls when multiple parent classes have methods with the same name.

When multiple parent classes have methods with the same name. Python uses the method resolution order (MRO) to determine which method to call. The MRO defines the order in which the base is searched for a method. Python follows a depth-first, then left-to-right approach, but with constraints that the order of parents is preserved. This is handled by the C3 linearization algorithm, which ensures a consistent and predictable MRO.

Let's consider a scenario where 'Human' and 'Mammal' classes both have an 'eat' method with different implementation, and 'Employee' inherits from both:

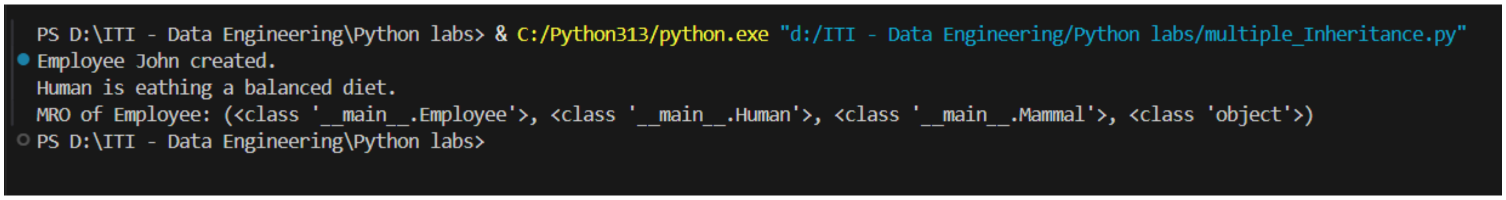


```python
class Human:
    def eat(self):
        print("Human is eathing a balanced diet.")

class Mammal:
    def eat(self):
        print("Mammal is eating raw food.")

class Employee(Human, Mammal):
    def __init__(self, name):
        self.name = name
        print(f"Employee {self.name} created.")

#example usage
e = Employee("John")
e.eat()
print(f"MRO of Employee: {Employee.__mro__}")
```

Explanation:

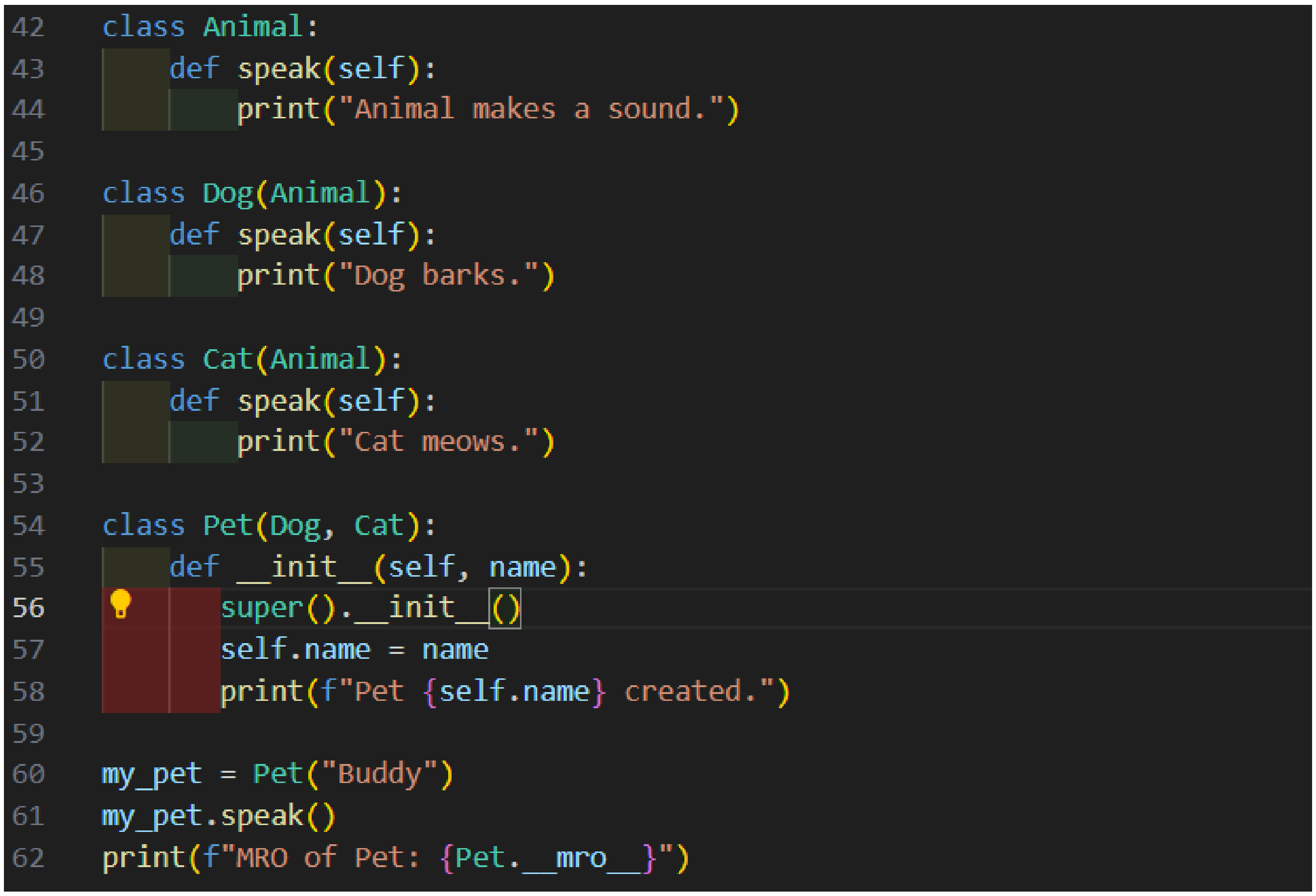
When emp.eat() is called, Python looks for the eat method in the Employee class first. If it's not found there, it follows the MRO of Employee. In this case, the MRO for Employee is Employee -> Human -> Mammal -> object. Therefore, Python finds eat in Human first and executes Human.eat(). If Employee itself had an eat method, that method would override the parent methods.

Output of the example:



```
PS D:\ITI - Data Engineering\Python labs> & C:/Python313/python.exe "d:/ITI - Data Engineering/Python labs/multiple_Inheritance.py"
Employee John created.
Human is eathing a balanced diet.
MRO of Employee: (<class '__main__.Employee'>, <class '__main__.Human'>, <class '__main__.Mammal'>, <class 'object'>)
PS D:\ITI - Data Engineering\Python labs>
```

Let's look at another example with a more complex inheritance hierarchy:

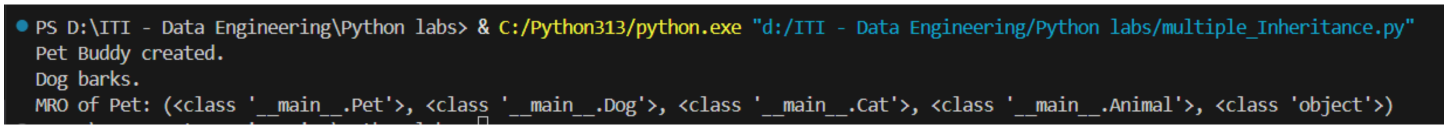


```python
class Animal:
    def speak(self):
        print("Animal makes a sound.")

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

class Cat(Animal):
    def speak(self):
        print("Cat meows.")

class Pet(Dog, Cat):
    def __init__(self, name):
        super().__init__()
        self.name = name
        print(f"Pet {self.name} created.")

my_pet = Pet("Buddy")
my_pet.speak()
print(f"MRO of Pet: {Pet.__mro__}")
```

Explanation:

For Pet("Buddy").speak(), the MRO is Pet -> Dog -> Cat -> Animal -> object. Python finds speak in Dog first, so Dog.speak() is executed.

Output of the example:



```
PS D:\ITI - Data Engineering\Python labs> & C:/Python313/python.exe "d:/ITI - Data Engineering/Python labs/multiple_Inheritance.py"
Pet Buddy created.
Dog barks.
MRO of Pet: (<class '__main__.Pet'>, <class '__main__.Dog'>, <class '__main__.Cat'>, <class '__main__.Animal'>, <class 'object'>)
```

Conclusion

Python’s multiple inheritance, while powerful, requires a clear understanding of the Method Resolution Order (MRO) and the super() function. The super() function ensures that parent class methods, especially __init__, are called correctly and once in complex inheritance hierarchies. When method names conflict across parent classes , python’s MRO (determined by the C3 linearization algorithm) provides a predictable mechanism for resolving which method is called, prioritizing classes earlier in the MRO. By understanding and utilizing these concepts, developers can effectively leverage multiple inheritance in python while maintaining code clarity and avoiding unexpected behavior.