Lecture 08: Multiple Inheritance

Swakkhar Shatabda

B.Sc. in Data Science
Department of Computer Science and Engineering
United International University

February 27, 2024

Multiple Inheritance - Example

```
class Communication:
    def make_call(self):
        print("Making a call")
    def send_message(self):
        print("Sending a message")

class Computing:
    def run_application(self):
        print("Running application")
    def store_data(self):
        print("Storing data")

class PowerManagement:
    def handle_battery(self):
        print("Handling battery")
    def power_saving_mode(self):
        print("Entering power saving mode")
```

Multiple Inheritance - Example

```
class Smartphone(Communication, Computing, PowerManagement):
    pass

smartphone = Smartphone()
smartphone.make_call()
smartphone.send_message()
smartphone.run_application()
smartphone.store_data()
smartphone.handle_battery()
smartphone.power_saving_mode()
```

Multilevel Inheritance

```
class Animal:
    def speak(self):
        print("Animal speaks")

class Dog(Animal):
    def bark(self):
        print("Dog barks")

class Labrador(Dog):
    def fetch(self):
        print("Labrador fetches")

# Using the classes
labrador = Labrador()
labrador.speak()  # Inherits from Animal
labrador.bark()   # Inherits from Dog
labrador.fetch()  # Specific to Labrador
```

The diamond probem

The "diamond problem" occurs when a class inherits from two classes that have a common ancestor. This can lead to ambiguity in method resolution.

```
class A:
    def greet(self):
        print("Hello from class A")
class B(A):
    def greet(self):
        print("Hello from class B")
class C(A):
    def greet(self):
        print("Hello from class C")
class D(B, C):
    pass
# Using the classes
obj = D()
obj.greet()  # What will this print?
```

Class Methods

A classmethod is a method that is bound to the class. It takes the class as its first argument (cls). This allows the method to access and modify class-level variables and perform operations related to the class.

```
class MyClass:
    class_variable = "Class Variable"
    def __init__(self, value):
        self.instance_variable = value
    @classmethod
    def class_method(cls):
        print("This is a class method")
        print("Accessing class variable:", cls.class_variable)
    def instance_method(self):
        print("This is an instance method")
        print("Accessing instance variable:", self.instance_variable)
MyClass.class_method()
obj = MyClass("Instance Variable")
obj.instance_method()
obj.class_method()
```

Class Method - Example

A common use case for a class method in Python is to provide alternative constructors.

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    @classmethod
    def from_string(cls, string):
        name, age = string.split(",")
        return cls(name.strip(), int(age))
    def __str__(self):
        return f"Name: {self.name}, Age: {self.age}"
# Creating a Person instance using the constructor
person1 = Person("Alice", 30)
print(person1)
# Creating a Person instance using the class method
person2 = Person.from_string("Bob,25")
print(person2)
```

Static Methods

Static methods are methods that belong to a class but do not operate on the instance or class. They are defined using the @staticmethod decorator. Static methods do not implicitly receive a reference to either the instance or the class as their first argument (i.e., self or cls).

```
class Date:
    def __init__(self, day, month, year):
        self.day = day
        self.month = month
        self.year = year
    def display(self):
        return f"{self.day}/{self.month}/{self.year}"
    @staticmethod
    def is_valid_date(day, month, year):
        if 1 <= day <= 31 and 1 <= month <= 12 and year >= 0:
            return True
        return False
print(Date.is_valid_date(31, 12, 2023))
print(Date.is_valid_date(32, 13, 2023))
```

A problem with tester codes in modules

A.py

```
class A:
    def __init__(self):
        self.x=12
        self.y=13
    def __str__(self):
        return f"{self.x} {self.y}"

a = A()
print(a)
print(__name__)
```

tester.py

```
from A import A
a = A()
print(a)
print(__name__)
```

If testing in the module is essential

```
12 13
A
12 13
__main__
```

```python
class A:
    def __init__(self):
        self.x=12
        self.y=13
    def __str__(self):
        return f"{self.x} {self.y}"
if __name__ == "__main__":
    a = A()
    print(a)
    print(__name__)
```