PROPERTIES OF OOPS

The properties of oops:

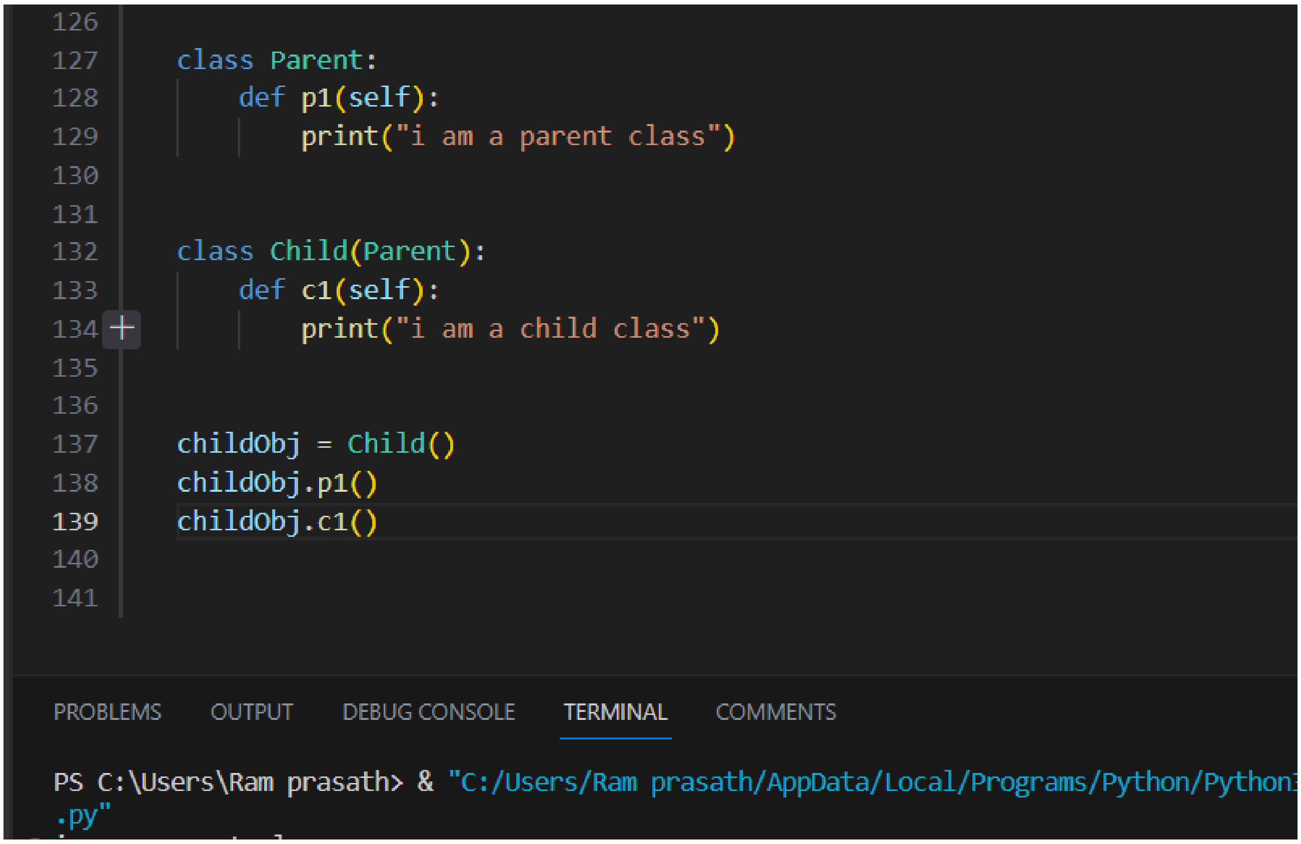
- Inheritance
- Abstraction
- Encapsulation
- Polymorphism

INHERITANCE

- Inheritance allows you to create new classes based on existing classes, reusing their code.
- It establishes a hierarchical structure, with parent and child classes.
- Child classes inherit attributes and methods from their parent class.
- Child classes can override inherited methods or add new ones to customize behavior.
- Inheritance enables polymorphism, treating different objects as instances of a common superclass.

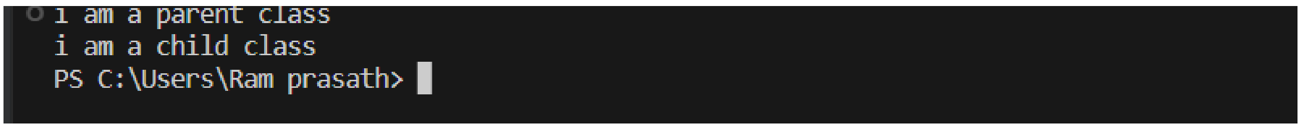
Types of Inheritance

Single Inheritance: A class inherits from a single parent class. It forms a simple parent-child relationship.



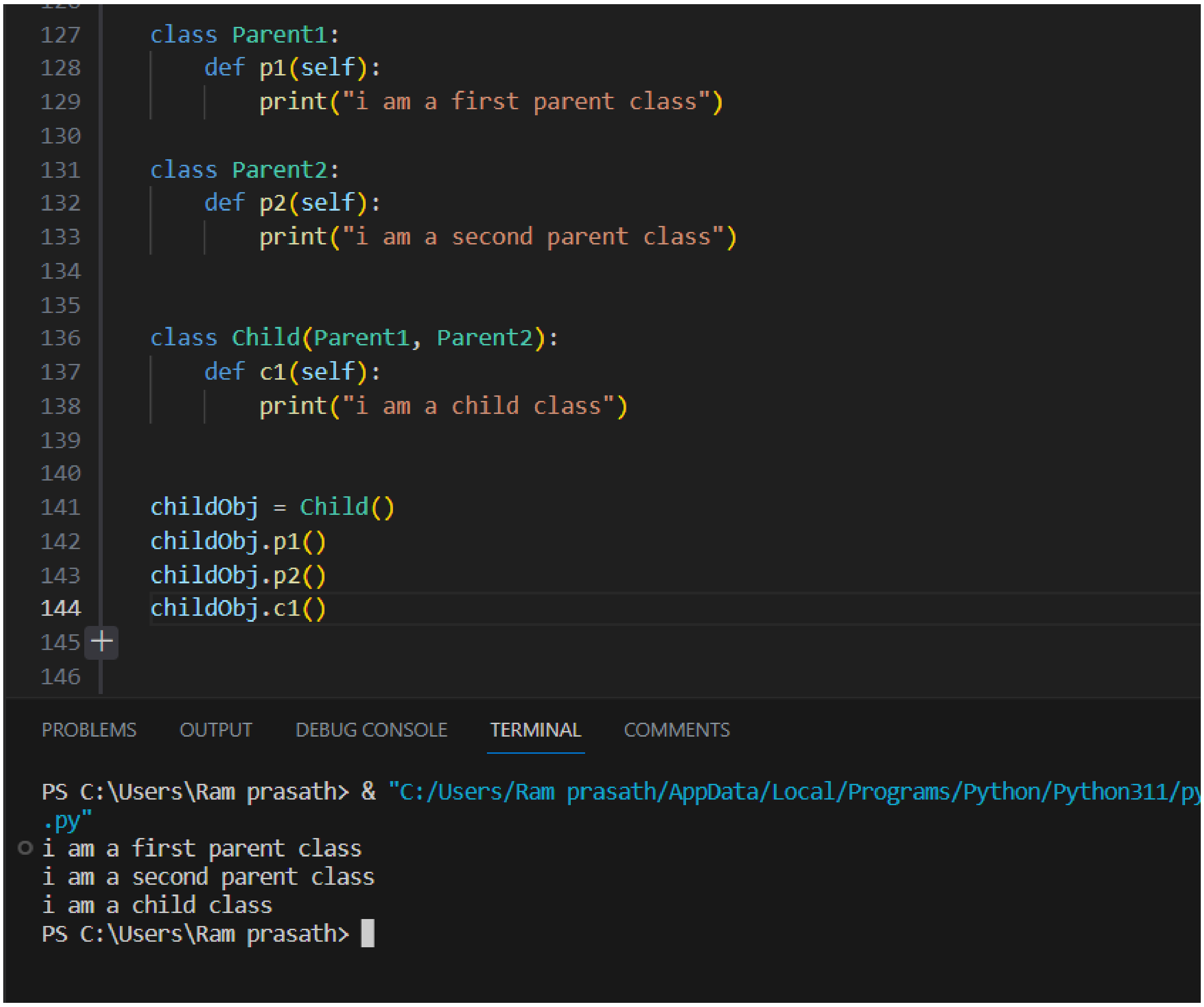
```python
class Parent:
    def p1(self):
        print("i am a parent class")


class Child(Parent):
    def c1(self):
        print("i am a child class")


childObj = Child()
childObj.p1()
childObj.c1()
```

```
PS C:\Users\Ram prasath> & "C:/Users/Ram prasath/AppData/Local/Programs/Python/Python
.py"
```



```
i am a parent class
i am a child class
PS C:\Users\Ram prasath>
```

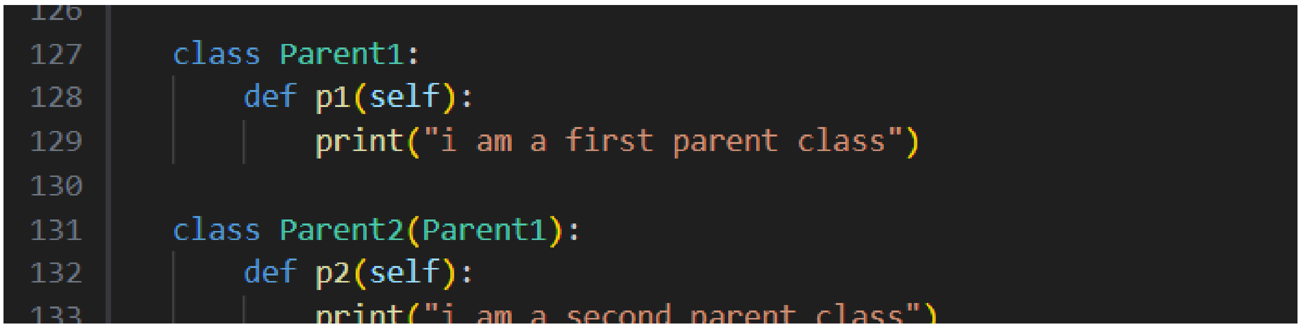
Multiple Inheritance: A class inherits from multiple parent classes. It allows a class to inherit attributes and behaviors from multiple sources.



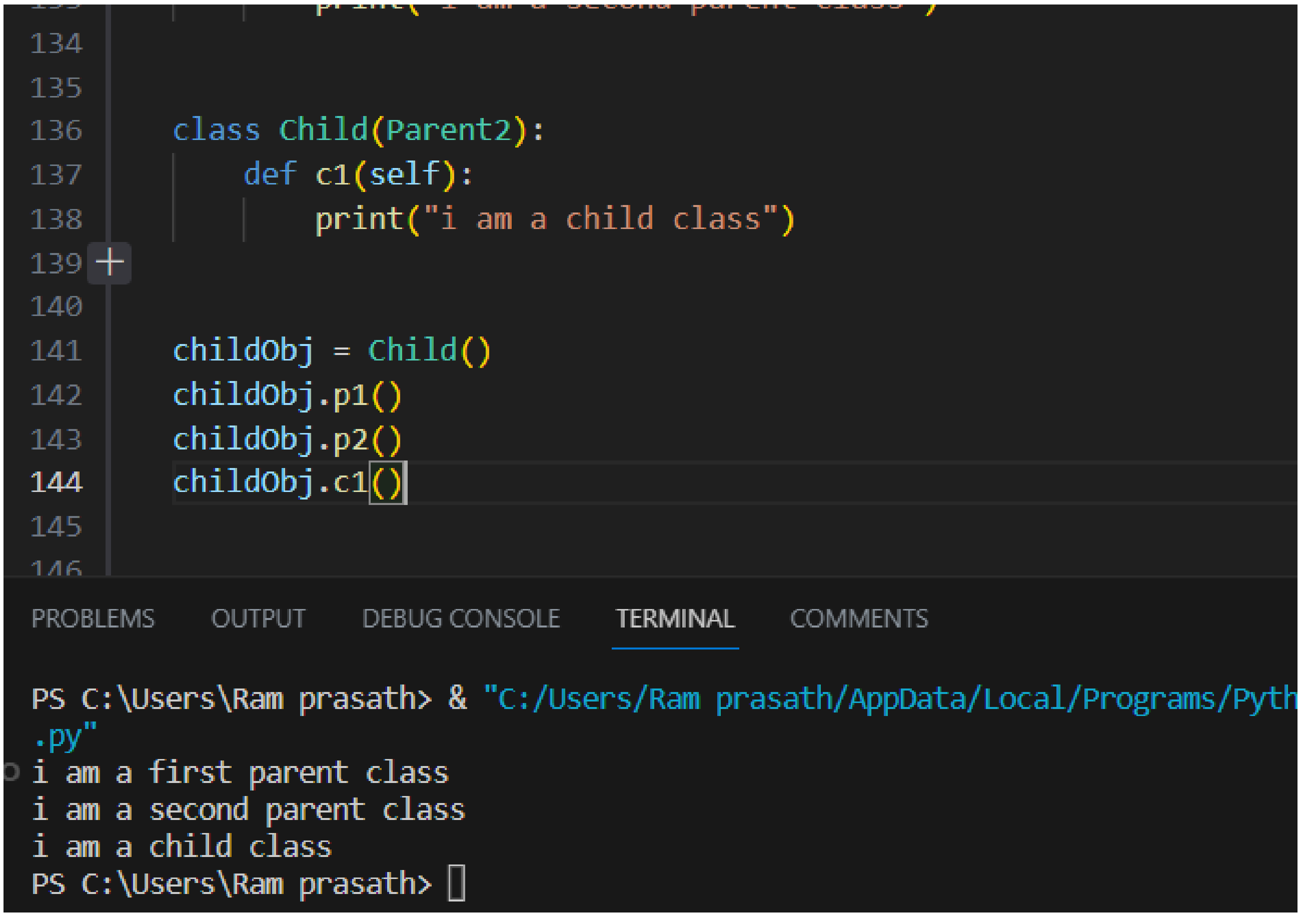
```python
class Parent1:
    def p1(self):
        print("i am a first parent class")

class Parent2:
    def p2(self):
        print("i am a second parent class")


class Child(Parent1, Parent2):
    def c1(self):
        print("i am a child class")


childObj = Child()
childObj.p1()
childObj.p2()
childObj.c1()
```

```
PS C:\Users\Ram prasath> & "C:/Users/Ram prasath/AppData/Local/Programs/Python/Python311/py
.py"
i am a first parent class
i am a second parent class
i am a child class
PS C:\Users\Ram prasath>
```

Multilevel Inheritance: A class inherits from a parent class, which in turn inherits from another parent class. It creates a chain of inheritance, forming a hierarchical structure.



```python
class Parent1:
    def p1(self):
        print("i am a first parent class")

class Parent2(Parent1):
    def p2(self):
        print("i am a second parent class")
```

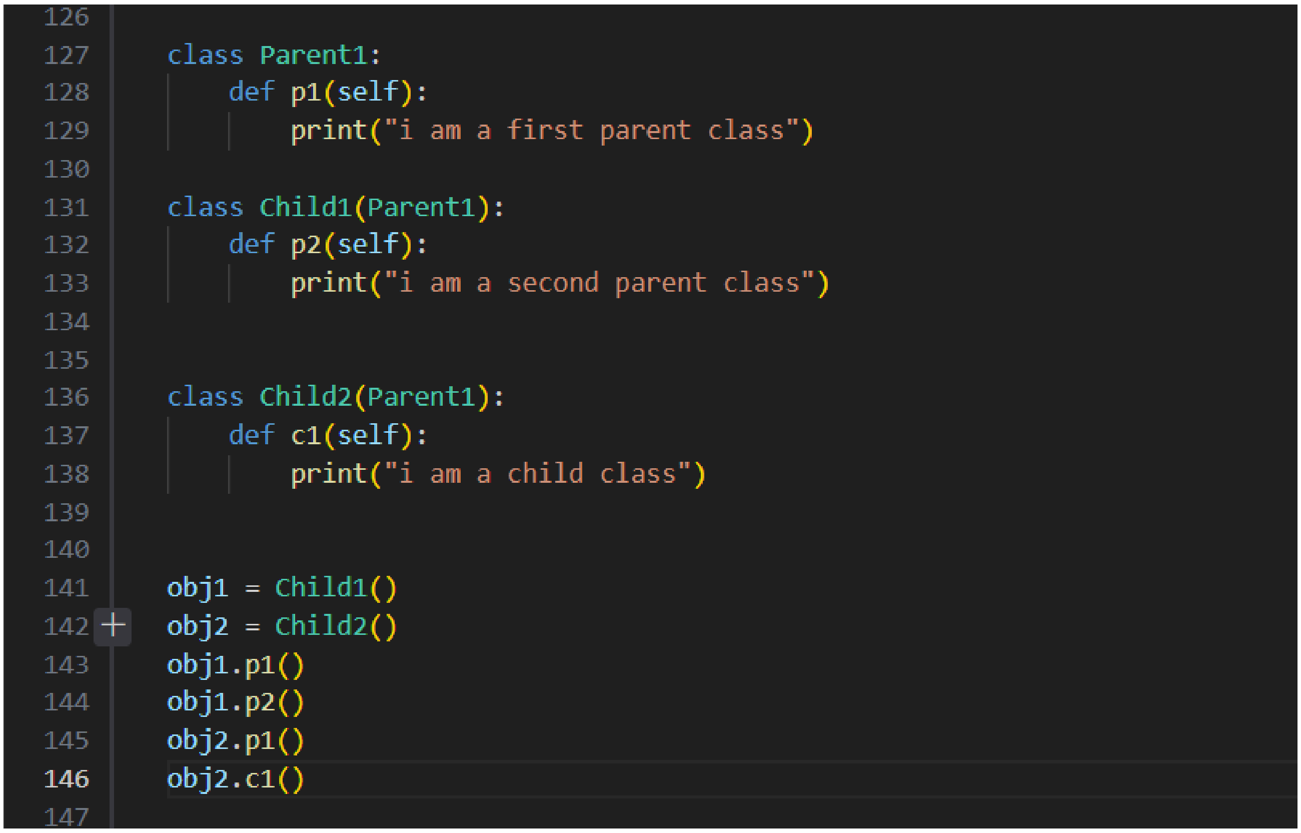


```
        print("i am a second parent class")


class Child(Parent2):
    def c1(self):
        print("i am a child class")


childObj = Child()
childObj.p1()
childObj.p2()
childObj.c1()

```

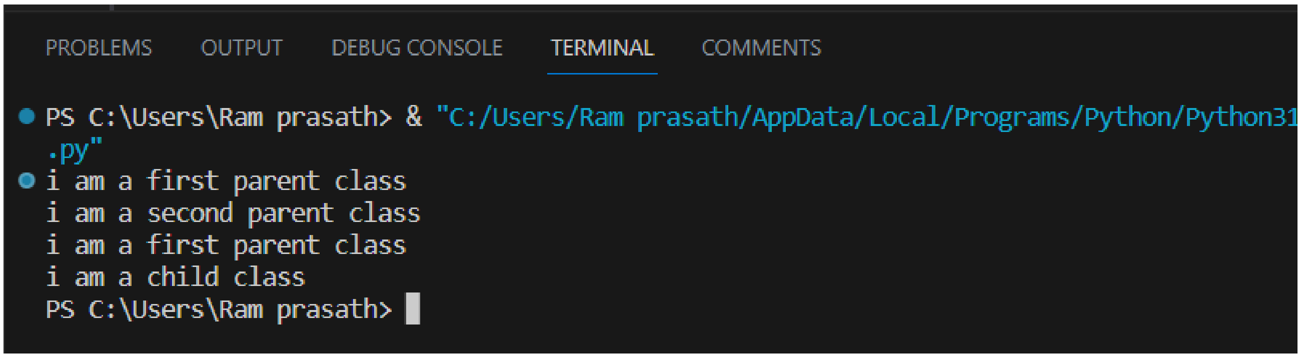
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL COMMENTS

```
PS C:\Users\Ram prasath> & "C:/Users/Ram prasath/AppData/Local/Programs/Pyth
.py"
i am a first parent class
i am a second parent class
i am a child class
PS C:\Users\Ram prasath>
```

Hierarchical Inheritance: Multiple classes inherit from a single parent class. It allows for different child classes to inherit from the same superclass.



```

class Parent1:
    def p1(self):
        print("i am a first parent class")

class Child1(Parent1):
    def p2(self):
        print("i am a second parent class")


class Child2(Parent1):
    def c1(self):
        print("i am a child class")


obj1 = Child1()
obj2 = Child2()
obj1.p1()
obj1.p2()
obj2.p1()
obj2.c1()

```



```
PS C:\Users\Ram prasath> & "C:/Users/Ram prasath/AppData/Local/Programs/Python/Python31
.py"
i am a first parent class
i am a second parent class
i am a first parent class
i am a child class
PS C:\Users\Ram prasath>
```

ABSTRACTION

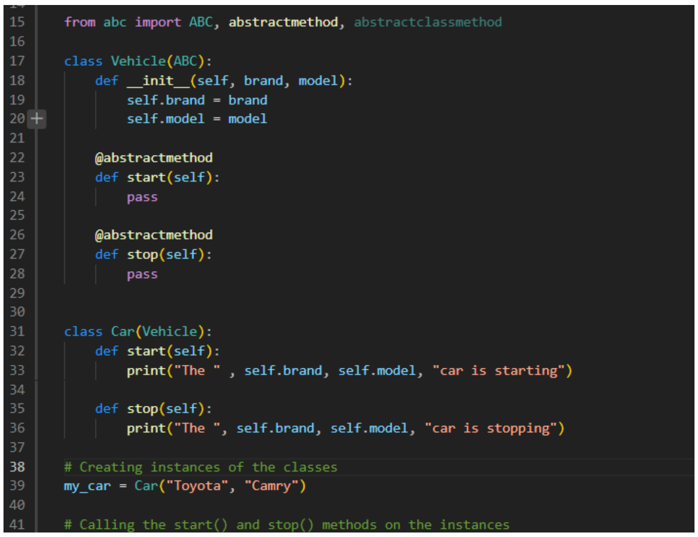
Abstraction in python is defined as a process of handling complexity by hiding unnecessary information from the user. This is one of the core concepts of object-oriented programming (OOP) languages.

Abstraction In Real World:

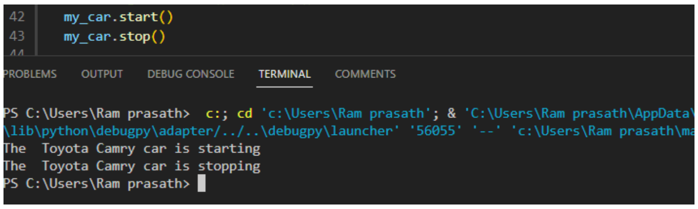
For example, we all use the social platforms and contact our friends, chat, share images etc., but we don't know how these operations are happening in the background.

Importance:

Abstraction provides a programmer to hide all the irrelevant data/process of an application in order to reduce complexity and increase the efficiency of the program.



```python
from abc import ABC, abstractmethod, abstractclassmethod

class Vehicle(ABC):
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model

    @abstractmethod
    def start(self):
        pass

    @abstractmethod
    def stop(self):
        pass


class Car(Vehicle):
    def start(self):
        print("The " , self.brand, self.model, "car is starting")

    def stop(self):
        print("The ", self.brand, self.model, "car is stopping")

# Creating instances of the classes
my_car = Car("Toyota", "Camry")

# Calling the start() and stop() methods on the instances
```



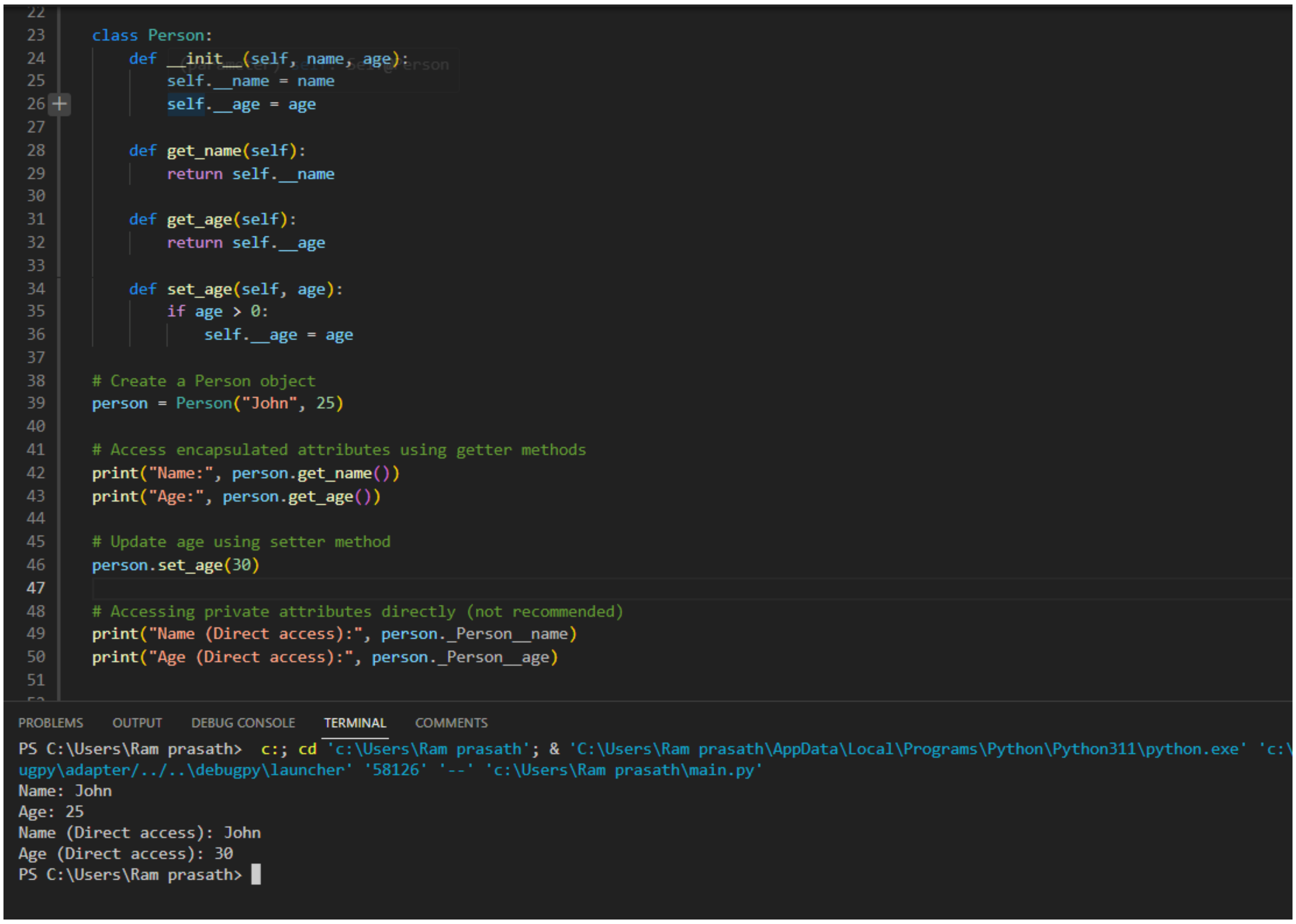
```python
    my_car.start()
    my_car.stop()
```

```
PS C:\Users\Ram prasath>  c:; cd 'c:\Users\Ram prasath'; & 'C:\Users\Ram prasath\AppData\
\lib\python\debugpy\adapter/../..\debugpy\launcher' '56055' '--' 'c:\Users\Ram prasath\ma
The  Toyota Camry car is starting
The  Toyota Camry car is stopping
PS C:\Users\Ram prasath> 
```

ENCAPSULATION

Encapsulation is an important feature of object-oriented programming. It involves combining data and functions within a class.

It describes the idea of wrapping data and the methods. This puts restrictions on accessing variables and methods directly and can prevent the accidental modification of data.



```python
class Person:
    def __init__(self, name, age):
        self.__name = name
        self.__age = age

    def get_name(self):
        return self.__name

    def get_age(self):
        return self.__age

    def set_age(self, age):
        if age > 0:
            self.__age = age

# Create a Person object
person = Person("John", 25)

# Access encapsulated attributes using getter methods
print("Name:", person.get_name())
print("Age:", person.get_age())

# Update age using setter method
person.set_age(30)

# Accessing private attributes directly (not recommended)
print("Name (Direct access):", person._Person__name)
print("Age (Direct access):", person._Person__age)
```

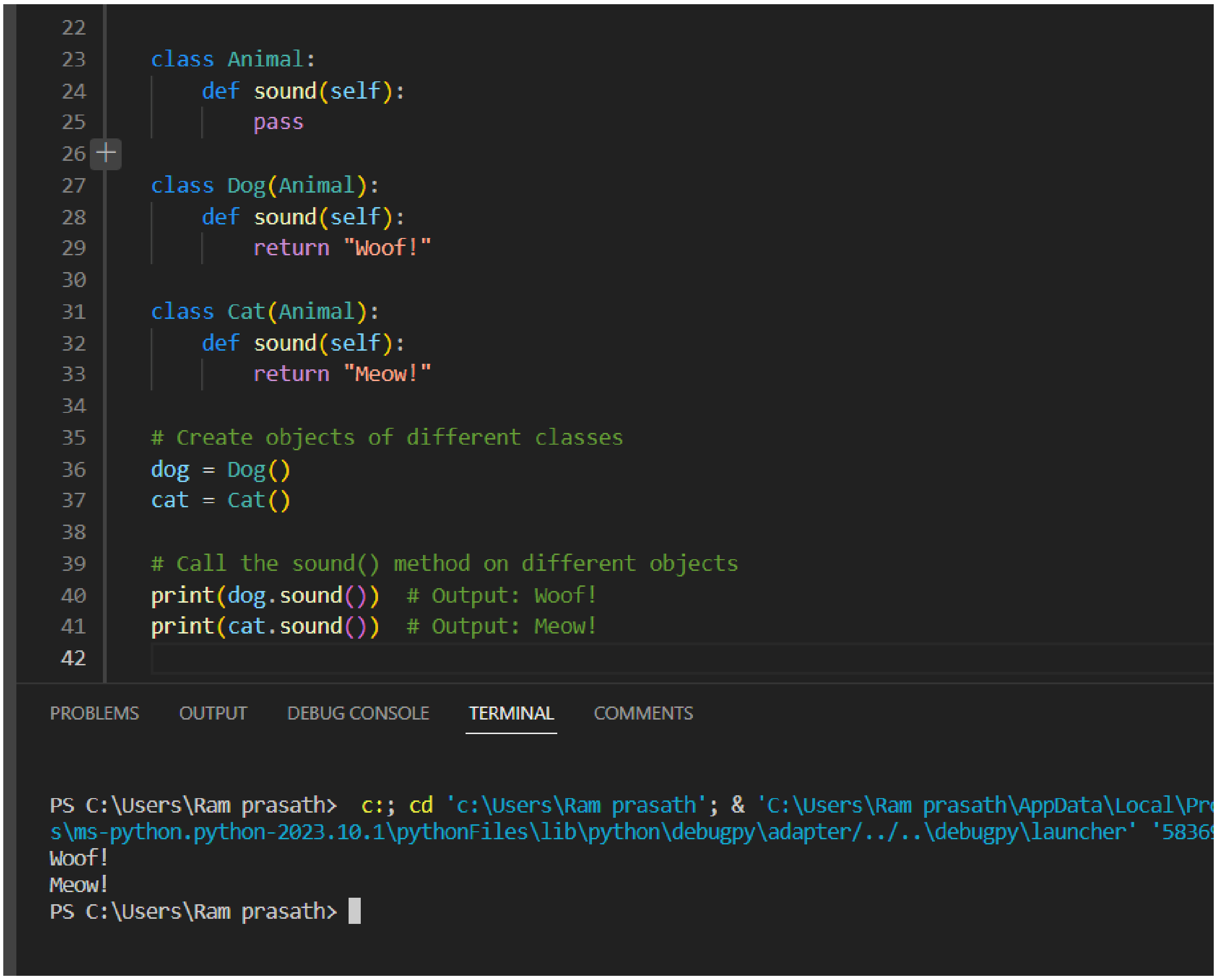
```
PS C:\Users\Ram prasath>  c:; cd 'c:\Users\Ram prasath'; & 'C:\Users\Ram prasath\AppData\Local\Programs\Python\Python311\python.exe' 'c:\
ugpy\adapter/../..\debugpy\launcher' '58126' '--' 'c:\Users\Ram prasath\main.py'
Name: John
Age: 25
Name (Direct access): John
Age (Direct access): 30
PS C:\Users\Ram prasath> 
```

In the example I provided, encapsulation is applied in the Person class.
By encapsulating the attributes and providing access through getter and setter methods, the Person class hides the internal details and provides a controlled interface for interacting with the encapsulated data.

This ensures that the attributes are accessed and modified in a controlled manner, promoting data integrity and encapsulation principles.

POLYMORPHISM

- Polymorphism allows objects of different classes to be treated as objects of a common superclass.
- It enables different objects to respond to the same method or function call in different ways.
- Polymorphism promotes code reusability and flexibility by providing a single interface to handle multiple object types.
- It simplifies code organization and makes it easier to write adaptable and generic code.



```python

class Animal:
    def sound(self):
        pass

class Dog(Animal):
    def sound(self):
        return "Woof!"

class Cat(Animal):
    def sound(self):
        return "Meow!"

# Create objects of different classes
dog = Dog()
cat = Cat()

# Call the sound() method on different objects
print(dog.sound())  # Output: Woof!
print(cat.sound())  # Output: Meow!

```

```
PS C:\Users\Ram prasath>  c:; cd 'c:\Users\Ram prasath'; & 'C:\Users\Ram prasath\AppData\Local\Pr
s\ms-python.python-2023.10.1\pythonFiles\lib\python\debugpy\adapter/../..\debugpy\launcher' '5836
Woof!
Meow!
PS C:\Users\Ram prasath> 
```

This demonstrates polymorphism, as the same method name (sound()) is called on different objects (of different classes), and each object responds with its specific behavior.