```
"""
Abstraction in Python, a core principle of Object-Oriented Programming (OOP),
focuses on handling complexity by hiding unnecessary details and
exposing only the essential functionalities to the user.
This allows developers to work with high-level concepts without
needing to understand the intricate underlying implementation.

Key aspects of abstraction in Python:
Hiding Complexity:
Abstraction aims to simplify interaction with complex systems by presenting a
simplified interface. Users or other parts of the code interact with this interface
without needing to know the internal workings.
Abstract Classes and Methods:
In Python, abstraction is primarily achieved through Abstract Base Classes (ABCs)
and abstract methods, provided by the abc module.
Abstract Class: A class that cannot be instantiated directly and serves as a
blueprint for other classes. It is defined by inheriting from ABC.
Abstract Method: A method declared within an abstract class but without an
implementation. Subclasses inheriting from the abstract class must provide concrete
implementations for these abstract methods.

Benefits:
Abstraction leads to more maintainable, scalable, and understandable code by
promoting:
Modularity: Breaking down complex systems into manageable, independent components.
Reusability: Defining common behaviors in abstract classes that can be inherited by
multiple subclasses.
Encapsulation: Hiding implementation details within classes, allowing users to
focus on what an object does rather than how it does it.

Why do we need Data Abstraction?
It hides internal logic and only shows the necessary details, making it easier to
use complex systems.
Sensitive or unnecessary details are not exposed, reducing chances of misuse or
accidental changes.
Users can focus on what the object does instead of how it does it.
Internal changes don’t affect external code, making it easier to update or modify
code components.
"""
from abc import ABC, abstractmethod



"""
class Animal(ABC):
    @abstractmethod
    def make_sound(self):
        pass


class Lion(Animal):
    def make_sound(self):
        print("Lions Roar....")

class Dog(Animal):
    def make_sound(self):
        print("Dogs Bark...")
```

```
lion = Lion()
lion.make_sound()

dog = Dog()
dog.make_sound()


animal = Animal()
"""


class Animal(ABC):
    @property
    @abstractmethod
    def animal_type(self):
        pass

    @abstractmethod
    def make_sound(self):
        pass


class Lion(Animal):
    @property
    def animal_type(self):
        return "Lion"

    def make_sound(self):
        print("Lions Roar....")

class Dog(Animal):
    @property
    def animal_type(self):
        return "Dog"

    def make_sound(self):
        print("Dogs Bark...")


lion = Lion()
print(lion.animal_type)
lion.make_sound()

dog = Dog()
print(dog.animal_type)
dog.make_sound()


# animal = Animal()
```