Abstraction and **encapsulation** are both important concepts in object-oriented programming (OOP), but they address different aspects of designing and structuring code.

1. **Abstraction:**
   - **Definition:** Abstraction is the process of hiding the complex implementation details and exposing only the essential features or functionalities of an object.
   - **Purpose:** It allows programmers to work with high-level concepts without being concerned about the low-level details. Abstraction focuses on what an object does rather than how it achieves its functionality.
   - **Implementation in Python:** In Python, abstraction is often achieved through abstract classes or interfaces. Abstract classes may contain abstract methods that must be implemented by subclasses, defining a common interface for a group of related classes.

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

class Circle(Shape):
    def area(self):
        # Implementation for calculating the area of a circle
        pass
```

2. **Encapsulation:**
   - **Definition:** Encapsulation is the bundling of data (attributes) and methods (functions) that operate on the data into a single unit or class. It restricts direct access to some of an object's components, providing data protection and implementation hiding.
   - **Purpose:** It helps in organizing the code, preventing unintended interference or modification of the internal state of an object. Encapsulation promotes the concept of information hiding, where the internal details are hidden and only a controlled interface is exposed.

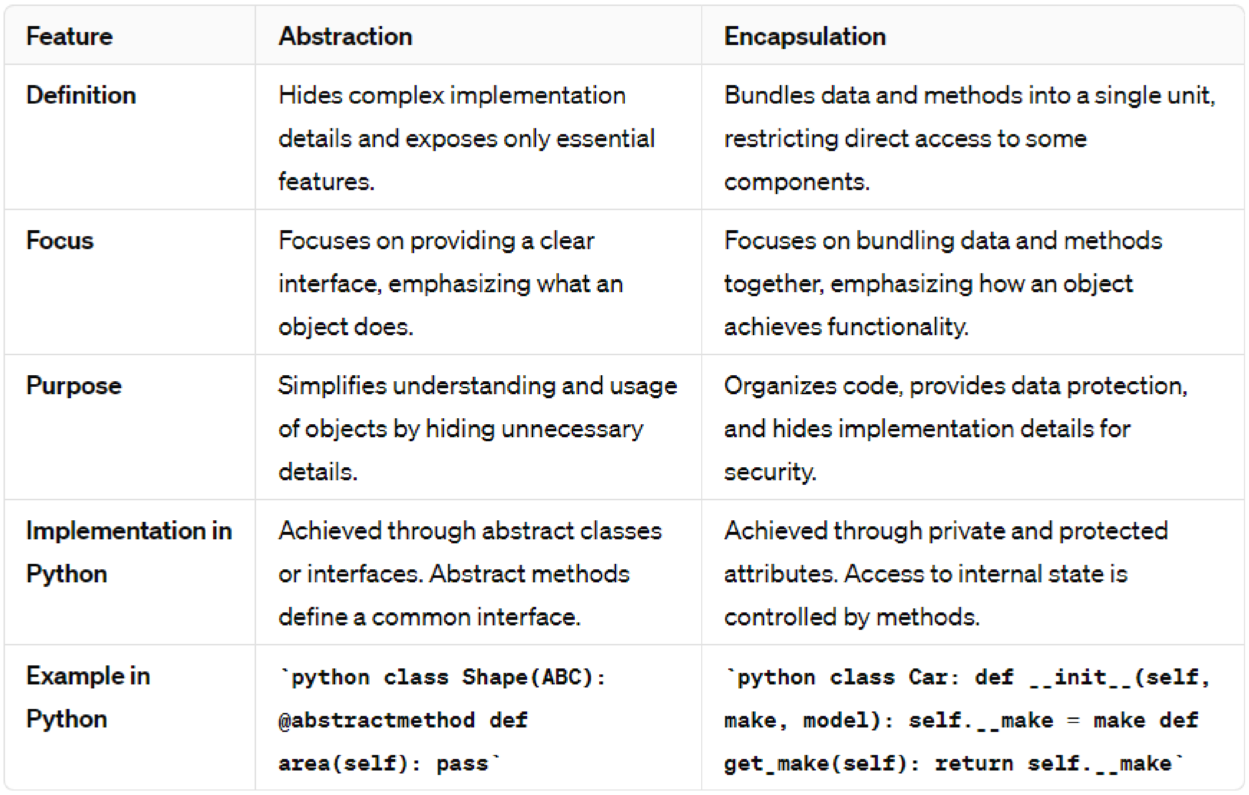
- **Implementation in Python:** In Python, encapsulation is achieved through the use of private and protected attributes. Private attributes are denoted by a double underscore (`__`), and protected attributes are denoted by a single underscore (`_`).

```
class Car:
    def __init__(self, make, model):
        self.__make = make  # Private attribute
        self._model = model  # Protected attribute

    def get_make(self):
        return self.__make

    def set_make(self, new_make):
        self.__make = new_make
```

- In the example above, **`__make`** is a private attribute, and **`get_make`** and **`set_make`** are methods used to access and modify the value of **`__make`**. This demonstrates encapsulation by controlling access to the internal state of the **`Car`** class.

In summary, abstraction focuses on hiding the complexity of implementation details by providing a clear and simplified interface, while encapsulation involves bundling data and methods together and controlling access to the internal state of an object. Both concepts contribute to creating more modular, maintainable, and secure code in an object-oriented paradigm.

A tabular comparison of abstraction and encapsulation:



Feature	Abstraction	Encapsulation
Definition	Hides complex implementation details and exposes only essential features.	Bundles data and methods into a single unit, restricting direct access to some components.
Focus	Focuses on providing a clear interface, emphasizing what an object does.	Focuses on bundling data and methods together, emphasizing how an object achieves functionality.
Purpose	Simplifies understanding and usage of objects by hiding unnecessary details.	Organizes code, provides data protection, and hides implementation details for security.
Implementation in Python	Achieved through abstract classes or interfaces. Abstract methods define a common interface.	Achieved through private and protected attributes. Access to internal state is controlled by methods.
Example in Python	`python class Shape(ABC): @abstractmethod def area(self): pass`	`python class Car: def __init__(self, make, model): self.__make = make def get_make(self): return self.__make`

In the examples, the `**Shape**` abstract class uses abstraction to define a common interface for shapes, and the `**Car**` class uses encapsulation to control access to its internal state through private attributes and methods.