In Python, interfaces are not explicitly supported as they are in some other programming languages like Java. However, the functionality of interfaces can be mimicked using **abstract base classes (ABCs)** from the abc module. Abstract base classes define methods that must be implemented by derived (subclass) classes, essentially serving the purpose of an interface.

How to Define and Use an Interface in Python

1. Import the abc Module

The abc module allows you to create abstract base classes.

2. Create an Abstract Base Class

Define a class with abstract methods using the @abstractmethod decorator.

3. Inherit from the Abstract Base Class

Any subclass inheriting from the abstract base class must implement the abstract methods.

```python
from abc import ABC, abstractmethod

# Define an interface
class Animal(ABC):
    @abstractmethod
    def make_sound(self):
        """Method to be implemented by subclasses"""
        pass

    @abstractmethod
    def move(self):
        """Method to be implemented by subclasses"""
        pass

# Concrete class implementing the interface
class Dog(Animal):
    def make_sound(self):
        return "Bark"

    def move(self):
        return "Runs on four legs"

class Bird(Animal):
    def make_sound(self):
        return "Chirp"

    def move(self):
        return "Flies in the sky"

# Instantiate the classes
dog = Dog()
bird = Bird()

print(dog.make_sound())  # Output: Bark
print(bird.move())       # Output: Flies in the sky
```

```
Bark
Flies in the sky
```

Key Points:

1. **Abstract Base Class (ABC)**:
    - Classes that inherit from ABC cannot be instantiated directly if they contain abstract methods.
2. **Abstract Methods (@abstractmethod)**:
    - Methods marked with @abstractmethod must be overridden in derived classes.
3. **Polymorphism**:
    - Abstract base classes allow for polymorphism. You can treat instances of derived classes as instances of the base class.

Advantages of Using Interfaces in Python:

- Enforces a contract for subclasses to implement certain methods.
- Encourages a clean and consistent API design.
- Promotes polymorphism, making your code more extensible and reusable.

Rules for implementing Python Interfaces

- Methods defined inside an interface must be abstract.
- Creating an object of an interface is not allowed.
- A class implementing an interface needs to define all the methods of that interface.
- In case a class is not implementing all the methods defined inside the interface, the class must be declared abstract.

In Python, the concept of **formal** and **informal** interfaces refers to how interfaces are implemented and enforced. Here's a comparison:

1. Formal Interface

A **formal interface** explicitly enforces rules by defining abstract base classes (ABCs) using the abc module. These interfaces ensure that any class inheriting from the interface must implement all its required methods.

Characteristics:

- Defined using the abc.ABC class.
- Methods are decorated with @abstractmethod.
- The interface enforces implementation of required methods in subclasses at runtime.
- Suitable for larger, more structured projects.

```
from abc import ABC, abstractmethod

# creating interface
class demoInterface(ABC):
   @abstractmethod
   def method1(self):
      print ("Abstract method1")
      return

   @abstractmethod
   def method2(self):
      print ("Abstract method1")
      return

# class implementing the above interface
class concreteclass(demoInterface):
    def method1(self):
       print ("This is method1")
       return

    def method2(self):
       print ("This is method2")
       return

# creating instance
obj = concreteclass()

# method call
obj.method1()
obj.method2()
```

2. Informal Interface

An **informal interface** is more relaxed and relies on Python's **duck typing** (the idea that if it "quacks like a duck," it is a duck). No explicit base class or @abstractmethod decorators are used. Instead, developers follow a convention where classes implement methods matching a certain pattern.

Characteristics:

- No strict enforcement of method implementation.
- Relies on documentation, naming conventions, or coding discipline.
- Suitable for smaller or less critical projects.
- May lead to runtime errors if methods are missing.

```
class demoInterface:
   def displayMsg(self):
      pass

class newClass(demoInterface):
   def displayMsg(self):
      print ("This is my message")

# creating instance
obj = newClass()

# method call
obj.displayMsg()
```

Comparison Table:

Aspect	Formal Interface	Informal Interface
Definition	Uses `abc.ABC` and `@abstractmethod`.	No explicit definition; based on conventions.
Enforcement	Enforced at runtime by Python.	No enforcement; relies on duck typing.
Error Handling	Errors are raised if methods are missing.	Runtime errors may occur if assumptions are violated.
Use Case	Suitable for large and complex projects.	Suitable for small, simple projects.
Documentation	Self-documented via abstract base classes.	Requires external documentation.

When to Use Which?

- **Formal Interface**: Use when strict enforcement is necessary, especially in large projects where multiple developers are involved.
- **Informal Interface**: Use for simpler cases where flexibility is more important than strict adherence to a predefined structure.

Self-Study: Why Use zope.interface?

Absolutely! `zope.interface` offers more flexible and extensive tools for defining and checking interfaces compared to `ABC`.
Step 3: Verify Implementation
`zope.interface` provides tools to verify that a class implements the specified interface correctly.
Benefits over ABC:
- **Attribute Definition**: `zope.interface` allows defining attributes within the interface, which `ABC` does not support directly.
- **Flexible Checks**: Provides tools like `verifyObject` and `verifyClass` to check if an object or class conforms to an interface, which is more flexible than the static enforcement in `ABC`.
- **Component Design**: Designed for component-based architectures, making it suitable for large, modular applications.
In contrast, with `ABC`:
- Attributes cannot be defined directly within the interface.
- Static enforcement: You must manually check at instantiation time.

Task 1:

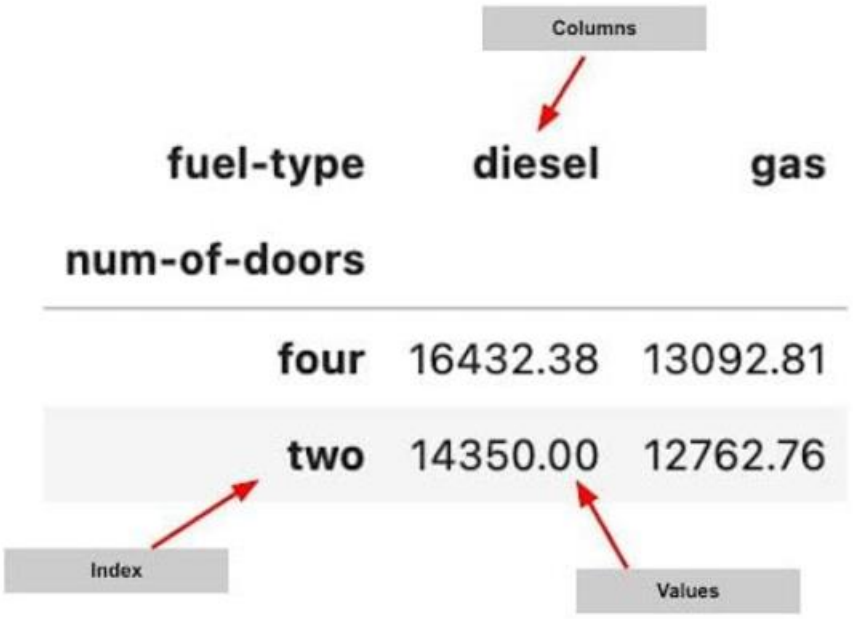
Suppose you are tasked with developing a payment gateway system for Bangabandhu Sheikh Mujibur Rahman Digital University, Bangladesh. The administration insists on using a **formal interface** to ensure that all payment modules implement strict methods like process_payment() and generate_receipt(). However, one of your teammates suggests using an **informal interface** for greater flexibility, allowing developers to implement only the methods they need, depending on the payment type (e.g., credit card, PayPal, cryptocurrency).

Write a Python code to show how a formal interface would enforce method implementation for a secure and consistent payment system.

In a payment processing system, explain when you would choose to use an abstract class over an interface. Justify your answer with examples of scenarios where each would be most appropriate.

***A pandas pivot table has three main elements:

- Index: This specifies the row-level grouping.
- Column: This specifies the column level grouping.
- Values: These are the numerical values you are looking to summarize.



1. What is pivot table? Why use need to use this?

   is a way to group and aggregate data to provide meaningful summaries.

2. Create the following Dataset.

Product	Region	Salesperson	Quantity	Revenue	Month
Laptop	East	Alice	5	5000	January
Tablet	West	Bob	10	3000	January
Smartphone	East	Charlie	15	7500	February
Laptop	West	Eve	8	8000	February
Tablet	East	Alice	12	3600	March
Smartphone	West	Bob	20	10000	March

3. Create a pivot table to calculate total revenue for each product category (Product) by region (Region). `pivot_table = df.pivot_table(values='Revenue', index='Product', columns='Region', aggfunc='sum'`

4.Calculate total revenue for each product category (Product) by both region (Region) and month (Month). `pivot_table = df.pivot_table(values='Revenue', index='Product', columns=['Region', 'Month'], aggfunc='sum', fill_value=0)`