Solid principles and when to use it

When you build a Python project using object-oriented programming (OOP), planning how the different classes and objects will interact to solve your specific problems is an important part of the job. This planning is known as object-oriented design (OOD), and getting it right can be a challenge. If you're stuck while designing your Python classes, then the **SOLID principles** can help you out.

SOLID is a set of five object-oriented design principles that can help you write more maintainable, flexible, and scalable code based on well-designed, cleanly structured classes. These principles are a fundamental part of object-oriented design best practices.

Object-Oriented Design in Python: The SOLID Principles

When it comes to writing classes and designing their interactions in Python, you can follow a series of principles that will help you build better object-oriented code. One of the most popular and widely accepted sets of standards for object-oriented design (OOD) is known as the SOLID principles.

But what are these SOLID principles? SOLID is an acronym that groups five core principles that apply to object-oriented design. These principles are the following:

1. Single-responsibility principle (SRP)
2. Open–closed principle (OCP)
3. Liskov substitution principle (LSP)
4. Interface segregation principle (ISP)
5. Dependency inversion principle (DIP)

Single-Responsibility Principle (SRP)

The single-responsibility principle states that:

A class should have only one reason to change.

This means that a class should have only one responsibility, as expressed through its methods. If a class takes care of more than one task, then you should separate those tasks into separate classes.

This principle is closely related to the concept of separation of concerns, which suggests that you should split your programs into different sections. Each section must address a separate concern.

Open-Closed Principle (OCP)

It means that:

Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.

Having to make changes to your base class means that your class is open to modification. That violates the open-closed principle. How can you fix your class to make it open to extension but closed to modification?

By turning it into an abstract base class (ABC), its methods will be overridden in all the subclasses.
In every case, you'll have to implement the required interface, which also makes your classes polymorphic.

Liskov Substitution Principle (LSP)

This principle has been a fundamental part of object-oriented programming. The principle states that:

Subtypes must be substitutable for their base types.

For example, if you have a piece of code that works with a `Shape` class, then you should be able to substitute that class with any of its subclasses, such as `Circle` or `Rectangle`, without breaking the code.

This principle is about making your subclasses behave like their base classes without breaking anyone's expectations when they call the same methods.

Interface Segregation Principle (ISP)

The interface segregation principle (ISP) comes from the same mind as the single-responsibility principle. The principal's main idea is that:

Clients should not be forced to depend upon methods that they do not use. Interfaces belong to clients, not to hierarchies.

In this case, *clients* are classes and subclasses, and *interfaces* consist of methods and attributes. In other words, if a class doesn't use particular methods or attributes, then those methods and attributes should be segregated into more specific classes.

Dependency Inversion Principle (DIP)

The dependency inversion principle (DIP) is the last principle in the SOLID set. This principle states that:

Abstractions should not depend upon details. Details should depend upon abstractions.

That sounds pretty complex. Here's an example that will help to clarify it. Say you're building an application and have a `FrontEnd` class to display data to the users in a friendly way. The app currently gets its data from a database, so you end up with the following code:

```python
# app_dip.py
class FrontEnd:
    def __init__(self, back_end):
        self.back_end = back_end
    def display_data(self):
        data = self.back_end.get_data_from_database()
        print("Display data:", data)
class BackEnd:
    def get_data_from_database(self):
        return "Data from the database"
```

In this example, the `FrontEnd` class depends on the `BackEnd` class and its concrete implementation. You can say that both classes are tightly coupled. This coupling can lead to scalability issues. For example, say that your app is growing fast, and you want the app to be able to read data from a REST API. How would you do that?

You may think of adding a new method to `BackEnd` to retrieve the data from the REST API. However, that will also require you to modify `FrontEnd`, which should be closed to modification, according to the open-closed principle.

To fix the issue, you can apply the dependency inversion principle and make your classes depend on abstractions rather than on concrete implementations like `BackEnd`. In this specific example, you can introduce a `DataSource` class that provides the interface to use in your concrete classes:

```python
# app_dip.py
from abc import ABC, abstractmethod
class FrontEnd:
    def __init__(self, data_source):
        self.data_source = data_source
    def display_data(self):
        data = self.data_source.get_data()
        print("Display data:", data)
class DataSource(ABC):
    @abstractmethod
    def get_data(self):
        pass
class Database(DataSource):
    def get_data(self):
        return "Data from the database"
class API(DataSource):
    def get_data(self):
        return "Data from the API"
```

In this redesign of your classes, you've added a `DataSource` class as an abstraction that provides the required interface, or the `.get_data()` method. Note how `FrontEnd` now depends on the interface provided by `DataSource`, which is an abstraction.

Then you define the `Database` class, which is a concrete implementation for those cases where you want to retrieve the data from your database. This class depends on the `DataSource` abstraction through inheritance. Finally, you define the `API` class to support retrieving the data from the REST API. This class also depends on the `DataSource` abstraction.

Here's how you can use the `FrontEnd` class in your code:

```
>>>
>>> from app_dip import API, Database, FrontEnd

>>> db_front_end = FrontEnd(Database())
>>> db_front_end.display_data()
Display data: Data from the database

>>> api_front_end = FrontEnd(API())
>>> api_front_end.display_data()
Display data: Data from the API
```

Here, you first initialize `FrontEnd` using a `Database` object and then again using an `API` object. Every time you call `.display_data()`, the result will depend on the concrete data source that you use. Note that you can also change the data source dynamically by reassigning the `.data_source` attribute in your `FrontEnd` instance.