Search Medium

Write

Sign In

Published in Level Up Coding

You have 5 free member-only story left this month. Upgrade for unlimited access.

Ali Bouabdallahi
Feb 23, 2022 · 3 min read · Member only · Listen

S.O.L.I.D Principles explained in Python with examples.

SOLID is a mnemonic abbreviation for a set of design principles created for software development in object-oriented languages. The principles in **SOLID** are intended to foster simpler, more robust and replicable code from software developers. Each letter in SOLID corresponds to a principle for development.

When implemented properly it makes your code more **extendable, logical and easier to read**.

To understand **SOLID** principles, you have to know the use of the interface clearly.

I'm going to try to explain **SOLID** Principles in Python in simplest way so that it's easy for beginners to understand. We'll keep the story easy to just take the examples provided and apply. 🙂

Let's go through each principle one by one:

1 Single Responsibility Principle

The Single Responsibility Principle requires that a class should have only one job. So if a class has more than one responsibility, it becomes coupled. A change to one responsibility results in modification of the other responsibility.

```
# Which is bad a class which has two responsibilities
class User:
    def __init__(self, name, email):
        self.name = name
        self.email = email

    def get_name(self):
        pass

    def authenticate(self, user):
        pass
```

Single Responsibility Principle hosted with ❤ by GitHub view raw

We have a User class which is responsible for both the user properties and user database management. If the application changes in a way that it affect database management functions. The classes that make use of User properties will have to be touched and recompiled to compensate for the new changes. It's like a domino effect, touch one card it affects all other cards in line.

So we simply split the class, we create another class that will handle the one responsibility of storing an user to a database:

```
class User:
    def __init__(self, name, email):
        self.name = name
        self.email = email

    def get_name(self):
        pass

class UserDB:
    def get_user(self, id = 1):
        pass

    def authenticate(self, user):
        pass
```

Single Responsibility Principle hosted with ❤ by GitHub view raw

A common solution to this dilemma is to apply the Facade pattern. For introduction to Facade pattern you can read more. User class will be the Facade for user database management and user properties management.

2 Open-Closed Principle

Software entities(Classes, modules, functions) should be open for extension, not modification.

Let's imagine you have a store, and you give a discount of 20% to your favorite customers using this class When you decide to offer double the 20% discount to VIP customers. You may modify the class like this:

```
class Discount:
    def __init__(self, customer, price):
        self.customer = customer
        self.price = price
    def get_discount(self):
        if self.customer == 'fav':
            return self.price * 0.2
        if self.customer == 'vip':
            return self.price * 0.4
```

Open-Closed Principle hosted with ❤ by GitHub view raw

No, this fails the OCP principle. OCP forbids it. If we want to give a new percent discount maybe, to a different type of customers, you will see that a new logic will be added. To make it follow the OCP principle, we will add a new class that will extend the Discount. In this new class, we would implement its new behavior:

```
class Discount:
    def __init__(self, customer, price):
        self.customer = customer
        self.price = price
    def get_discount(self):
        return self.price * 0.2
class VIPDiscount(Discount):
    def get_discount(self):
        return super().get_discount() * 2
```

Open-Closed Principle hosted with ❤ by GitHub view raw

If you decide 40% discount to super VIP customers, it should be like this:

```
class SuperVIPDiscount(VIPDiscount):
    def get_discount(self):
        return super().get_discount() * 2
```

3 Liskov Substitution Principle

The main idea behind Liskov Substitution Principle is that, for any class, a client should be able to use any of its subtypes indistinguishably, without even noticing, and therefore without compromising the expected behavior at runtime. This means that clients are completely isolated and unaware of changes in the class hierarchy.

More formally:
Let q(x) be a property provable about objects of x of type T. Then q(y) should be provable for objects y of type S where S is a subtype of T.

In simpler terms means that a subclass, child or specialization of an object or class must be suitable by its Parent or SuperClass.

```
class Animal:
    def __init__(self, color, name):
        self.color = color
        self.name = name
    def make_sound(self):
        raise NotImplemented("Subclass must implement abstract method")
    def get_name(self):
        return self.name
class Cat(Animal):
    def make_sound(self):
        return self.name + " says: meow"
class Dog(Animal):
    def make_sound(self):
        return self.name + " says: woof!"
class Duck(Animal):
    def make_sound(self):
        return self.name + " says: quack"
def get_interaction(animal):
    print("Let's hear what a "+animal.get_name()+" says:")
    print(animal.make_sound())
animals = [Cat("black", "Felix"), Dog("white", "Max"), Duck("yellow", "Donald")]
for a in animals:
    get_interaction(a)
```

Liskov Substitution Principle hosted with ❤ by GitHub view raw

Remarks on the LSP The LSP is fundamental to a good object-oriented software design because it emphasizes one of its core traits — polymorphism. It is about creating correct hierarchies so that classes derived from a base use are polymorphic along the parent axis, with respect to the methods on their interface. It is also interesting to notice how this principle relates to the previous one — if we attempt to extend a class with a new feature, we risk breaking the LSP. Extensions to a class may not be well-behaved, and as a result such an extension will not be possible (or, if it is possible, we would have to break the other end of the principle and modify the code in the client that should be closed for modification, which is completely undesirable and unacceptable).

Carefully thinking about how classes in the way that LSP suggests helps us to extend the hierarchy correctly. We could then say that LSP contributes to the OCP.

4 Interface Segregation Principle

Make fine grained interfaces that are client specific Clients should not be forced to depend upon interfaces that they do not use. This principle deals with the disadvantages of implementing big interfaces.

To illustrate this complexity, we will go with a classic example because it is highly significant and easily understandable. The Classic Example

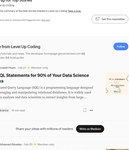
```
class IWorker:
    def work(self):
        raise NotImplemented("Implement this method")

class IFeedingHuman:
    def feed_human(self):
        pass

class IEarnMoney:
    def get_money(self):
        pass

class HumanWorker(IWorker, IFeedingHuman, IEarnMoney):
    def work(self):
        pass
    def feed_human(self):
        pass
    def get_money(self):
        pass

class RobotWorker(IWorker):
    def work(self):
        pass
```

Interface Segregation Principle hosted with ❤ by GitHub view raw

Another nice trick is that in our business logic, a single class can implement several interfaces if needed. So we can provide a single implementation for all the common methods between the interfaces. The segregated interfaces will also force us to think of our code more from the client's point of view, which will in turn lead to loose coupling and easy testing. So, not only here we made our code better to our clients, we also made it easier for ourselves to understand, test and implement.

5 Dependency Inversion Principle

Dependency should be on abstractions not concretions. High level modules should not depend upon low level modules. Both low and high level modules should depend on the same abstractions. Abstractions should not depend on details. Details should depend upon abstractions.



There comes a point in software development where our app will be largely composed of modules. When this happens, we have to close things up by using dependency injection. High level components depending on low level components is function. To create specific behavior you can use techniques like inheritance or interfaces.

```
class Engine:
    def __init__(self, power=None):
        self.power = power
    def start(self):
        print("Engine started...")
        pass
    def stop(self):
        print("Engine stopped...")
        pass
class Car:
    def __init__(self, name=None, engine=None):
        self.name = name
        self.engine = engine
    def start(self):
        print("Car "+self.name+" is starting...")
        self.engine.start()
    def stop(self):
        print("Car "+self.name+" is stopping...")
        self.engine.stop()
class ElectricEngine(Engine):
    def __init__(self, power=None):
        super().__init__(power)
    def start(self):
        print("Electric engine started...")
        pass
    def stop(self):
        print("Electric engine stopped...")
        pass
class ElectricCar(Car):
    def __init__(self, name=None, engine=None):
        super().__init__(name, engine)
    def start(self):
        print("Electric car "+self.name+" is starting...")
        self.engine.start()
    def stop(self):
        print("Electric car "+self.name+" is stopping...")
        self.engine.stop()
if __name__ == '__main__':
    car = Car("Test", ElectricEngine())
    car.start()
    car.stop()
    electric_car = ElectricCar("Test", ElectricEngine())
    electric_car.start()
    electric_car.stop()
```

Dependency Inversion Principle hosted with ❤ by GitHub view raw

You can learn more:

Solid 5 Class

Clean Code in Python

Software Development · Python · Programming · Software Patterns · Development

Sign up for Top Stories
By Level Up Coding
A monthly summary of the best stories shared to Level Up Coding. See a sample

Enter your email address | Get this newsletter

More from Level Up Coding

Follow

Coding tutorials and news. The developer homepage gitconnected.com && learn.codecademy.com

Trending Stories · Feb 21 · Member only

13 SQL Statements for 80% of Your Data Science Tasks

Structured Query Language (SQL) is a programming language designed for managing and manipulating relational databases. It is widely used by data analysts and data scientists to extract insights from large...

Data Science · 10 min read

Share your ideas with millions of readers. Write on Medium

Mohamed Elzanaty · Feb 20 · Member only

I am a bad software developer and this is my life

Hello everyone, I'm Mohamed Elzanaty, a software developer and discovery. The journey began seven months ago when I was employed at a job that didn't make me happy, but I couldn't pinpoint why. When...

Programming · 4 min read

Sophie Lenski · Feb 21 · Member only

Weekly #23ed Richard Person Says If You Understand the Stanford Experiment, You'll Become Wealthy

He was paid to origami to discuss his cards ranchers and farmers believe shooting bears how to design and build with microservices. Anonymous...

Coding · 3 min read

Jacob Bennett · Nov 15, 2022 · Member only

Use Gh like a senior engineer

Gh is a powerful tool that lets you do more when you know how to use it. — I've used these features of Gh for years across teams and projects. I'm still developing optimal ways around some workflows I like to...

Programming · 4 min read

TrianaPfister · Jan 25 · Member only

20 Pandas Functions for 80% of your Data Science Tasks

Master these Functions and Get Your Work Done — Pandas is one of the most widely used libraries in the data science community and for...

Data Science · 10 min read

Read more from Level Up Coding

Text unlimited access

Ali Bouabdallahi
25 Followers

Follow

More from Medium

Jacob Bennett · in Level Up Coding
Use Gh like a senior engineer

Mohamed Elzanaty · Feb 20 · Member only
I am a bad software developer and this is my life

Ali Bouabdallahi · in Towards Data Sci...
10 Python Decorators To Take Your Code To The Next Level

Ali Bouabdallahi · in Level Up Coding
Stop Using Excel or C Name, Class or Who

Help · About · Writer · Blog · Careers · Privacy · Terms · Text to speech · Teams