The Open-Closed Principle (OCP)

The Open-Closed Principle (OCP)

The Open-Closed Principle (OCP) is one of the SOLID principles of object-oriented design, which states that "software entities (classes, modules, functions, etc.) should be open for extension but closed for modification". This means that you should be able to add new functionality to your code without modifying the existing code.

Example 1:

```
class Shape:
    def area(self):
        pass


class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height


class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14 * self.radius ** 2
```

In this example, we have a Shape class that defines a method area() that all shapes should implement. We also have two subclasses of `Shape`: `Rectangle` and `Circle`, which implement the `area()` method differently.

Now let's say we want to add a new shape, `Triangle`. We can do this without modifying the existing code:

```
class Triangle(Shape):
    def __init__(self, base, height):
        self.base = base
        self.height = height

    def area(self):
        return 0.5 * self.base * self.height
```

We have extended the functionality of our code by adding a new shape, but we didn't modify any of the existing code. This is an example of the Open-Closed Principle in action.

Example 2:

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

    def make_sound(self):
        pass


class Dog(Animal):
    def make_sound(self):
        return "Woof!"
```

```
class Cat(Animal):
    def make_sound(self):
        return "Meow!"


class Cow(Animal):
    def make_sound(self):
        return "Moo!"
```

In this example, we have a base Animal class with an abstract `make_sound()` method that all animals should implement. We also have three subclasses of `Animal`: `Dog`, `Cat`, and `Cow`, each of which implements the `make_sound()` method differently.

Now let's say we want to add a new animal, `Sheep`. We can do this without modifying the existing code:

```
class Sheep(Animal):
    def make_sound(self):
        return "Baa!"
```

We have extended the functionality of our code by adding a new animal, but we didn't modify any of the existing code.

SOLID Principles in Python

{"name"=>"Yakhyokhuja Valikhujaev", "email"=>"yakhyo9696@gmail.com"}

SOLID principles. All text and code examples generated by ChatGPT