Design Pattern:

Design patterns are a well-described solution to the most commonly encountered problems which occur during software development. Design pattern represents the best practices evolved over a period of time by experienced software developers. They promote reusability which leads to a more robust and maintainable code.

Creational

Behavioral

Functional

**Singleton Design Pattern:**

Singleton pattern is a creational pattern that allows only one instance of a class to be created which will be available to the whole application. The major advantage of the Singleton design pattern is its saves memory because the single instance is reused again and again.

**Drawback of Singletin Design Pattern**:   
Singleton causes code to be tightly coupled. The singleton object is exposed globally and is available to a whole application. Thus, classes using this object become tightly coupled; any change in the global object will impact all other classes using it.

They hide dependencies instead of exposing them.

Singleton Pattern does not support inheritance.

* A builder design pattern is used to construct a complex object. It is designed to solve the issues with the factory and abstract design pattern.

**Factory design pattern:**

Factory classes provide flexibility in terms of design. Below are some of the benefits of factory class:

•Factory design pattern results in more decoupled code as it allows us to hide creational logic from dependent code

•It allows us to introduce an Inversion of the Control container.

•It gives you a lot more flexibility when it comes time to change the application as our creational logic is hidden from dependent code.

The major difference between these two is, a factory pattern creates an object through inheritance and produces only one Product. On the other hand, an abstract factory pattern creates the object through composition and produce families of products.

**Observer design pattern:**

Observer design pattern is one of the behavioral design patterns which defines one-to-many dependencies between objects & is useful when we are interested in a state of an object and we want to get notified when there is any change in the state of the Object. In the Observer design pattern, when one object changes its state, all its dependent objects are automatically notified, the object is called Subject, and dependents are called Observers.

**Decorator design pattern:**

The decorator pattern, also known as a structural pattern is used to add additional functionality to a particular object at runtime. It wraps the original object through a decorator object. For example, when you are buying a burger, you can customize it by adding extra filling and sauces, now the cost of these items has to be added to the final price. The customization will differ from customer to customer and offer from a shop. Creating different classes of burgers with different fillings will end up creating a lot of classes. The decorator solves this problem by extending the functionality of a single Burger class at runtime based on customer requests.

**Command Design Pattern**:

A command design pattern focuses on how different classes and objects behave mutually. Because mutual behavior can affect the software system’s response to user requests, any loosely coupled action can disrupt the software’s functionality, and this pattern helps identify such errors.

**Chain of responsibility**:

Handling incoming http request in a chain of objects. This also allow developers to dynamically insert objects into the chain and thus easily change the way http requests are handled.

SOLID Principles explained in Python with examples.

**Single Responsibility Principle**:  
 SRP Should server single responsibility.  
 When designing our classes, we should aim to put related features together,  
 so whenever they tend to change, they change for the same reason. And we should  
 try to separate features if they will change for different reasons. - Steve Fenton

**Open/Closed Principle**:

Software entities(Classes, modules, functions) should be open for extension, not modification. We can easily able to extends and accommodate new changes.

**Liskov Substitution Principle:**

A sub-class must be substitutable for its super-class. The aim of this principle is to ascertain that a sub-class can assume the place of its super-class without errors. If the code finds itself checking the type of class then, it must have violated this principle. Let’s use our Animal example.

**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.

If you have a base class with many methods, possibly not all of your subclasses are going to need them, maybe just a few. But due to inheritance, you will be able to call these methods on all the subclasses, even on those that don’t need it. This means a lot of interfaces that are unused, unneeded and will result in bugs when they get accidentally called.

This principle is meant to prevent this from happening. We should make interfaces as small as possible, so that we don’t need to implement functions we don’t need. Instead of one big base class, we should split them into multiple ones. They should only have methods that make sense for each, and then have our subclasses inherit from them.

**Dependency Inversion Principle**:

Dependency should be on abstractions not concretions

A. High-level modules should not depend upon low-level modules. Both should depend upon abstractions.

B. 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 clear things up by using dependency injection. High-level components depending on low-level components to function.