DESIGN PATTERNS

Singleton Design Pattern

Singleton design pattern is a creational design pattern that makes sure that a class has only one instance and is globally accessible by all other classes. Singleton design pattern reduces memory usage by sharing a single instance across the application.

When will we need Singleton Design Pattern?

Singleton pattern might be useful if the three conditions are met.

1. You need to control concurrent access to a shared resource.
2. Only one instance of the object is sufficient throughout the context of the application.
3. More than one independent parts of the application require access to the resource.

How Does Singleton Design Pattern Works?

Singleton Design Pattern has two core participants Singleton and Client.

Singleton:

Is a class with only one instance, and it's globally accessible by other classes across our application.

How to Make a Class Singleton?

It should have private constructor so that no instance of it can be created by other classes.

It should have a public static method so that then it can be called to get its singleton instance.

Client:

The Client interacts with the singleton class to get its shared instance and use it to call its methods.

Implementation of Singleton in Python:

```
class Logger():
    _instance = None
    def __init__(self):
        raise RunTimeError("Can Not Create Instance , call instance()
        instead")
    @classmethod
    def instance(cls):
        if cls._instance is None:
            cls._instance = cls.__new__(cls)
        return cls._instance
l = Logger() # Can Not Create Instance, call instance() instead
l = Logger.instance()
l1 = Logger.instance()
print("Are l1 and l2 same object ?" ,l is l2)
```

we can implement Singleton Class in more pythonic way like this

```
class Logger():
    _instance = None
    def __new__(cls):
        if cls._instance is None:
            print('Creating new Object")
            cls._instance = super(Logger,cls).__new__(cls)
        return cls._instance
l1 = Logger()
l2 = Logger()
print("Are l1 and l2 same object ?" , l1 is l2)
```

Factory Design Pattern:

The factory method pattern is a creational pattern that uses factory methods to deal with the problem of creating objects without having to specify the exact class of the object that will be created.

This is done by calling factory method specified in an interface and implemented by child classes, or implemented in a base class and optionally overridden by derived classes—rather than by calling a constructor.

When to use factory design pattern?

The factory design pattern is used when we have a superclass with multiple sub-classes and based on input, we need to return one of the sub-class.

Implementation of Factory Design Pattern in Python:

```
from abc import ABC, abstractmethod

class AbstractCar(ABC):

    @abstractmethod

    def get_body_type(self):

        pass


class SedanCar(AbstractCar):

    def __init__(self):

        self.body = "Sedan"

    def get_body_type(self):

        return f'Body type : {self.body}'


class HatchBakCar(AbstractCar):

    def __init__(self):

        self.body =  "HatchBak"

    def get_body_type(self):
        return f'Body Type : {self.body}'


class PickupCar(AbstractCar):
    def __init__(self):
        self.body =  "Pickup"
    def get_body_type(self):
        return f'Body Type : {self.body}'


class CarFactory():
    @staticmethod
    def build_plan(plan):
        try:
            if plan == "Sedan":
                return SedanCar()
            elif plan == "HatchBak":
                return HatchBakCar()
            elif plan == "Pickup":
                return PickupCar()
            raise AssertionError("Car Type not valid")
        except AssertionError as e:
            print(e)


plan = input("Enter the plan for car")
car = CarFactory.build_plan(plan)
body = car.get_body_type()
print(body)
```