Creational design patterns - Singleton

Key terms

Design patterns

> A design pattern is a general, reusable solution to a commonly occurring problem within a given context in software design. Design patterns are formalized best practices that the programmer can use to solve common problems when designing an application or system.

Creational design patterns

> Creational design patterns provide various object creation mechanisms, which increase flexibility and reuse of existing code.

Singleton

> The singleton pattern is a software design pattern that restricts the instantiation of a class to one object. This is useful when exactly one object is needed to coordinate actions across the system.

Singleton

Problem

- `Shared resource` - Imagine you have a class that is responsible for managing the database connection. You want to make sure that only one instance of this class exists in your application. If you create multiple instances of this class, you will end up with multiple database connections, which is not what you want. Similarly, there can be a class that is responsible for managing the logging mechanism. You want to make sure that only one instance of this class exists in your application. If you create multiple instances of this class, you will end up with multiple log files, which is not what you want.
- `Single access point` - Applications often require configuration. For example, you might want to configure the database connection parameters. You want to make sure that only one instance of this class exists in your application. A configuration class should have a single access point to the configuration parameters. If you create multiple instances of this class, you will end up with multiple configuration files.

Solution

Singleton pattern is a creational design pattern that lets you ensure that a class has only one instance, while providing a global access point to this instance. To implement the Singleton pattern, the following steps are required:

- `Constructor hiding` - The constructor of the singleton class should be private or protected. This will prevent other classes from instantiating the singleton class.
- `Global access point` - The singleton class should provide a global access point to get the instance of the singleton class. This global access point should be static and should return the same instance of the singleton class every time it is called. If the instance does not exist, it should create the instance and then return it.

Simple singleton - The Gang of Four implementation

Step 1 - Constructor hiding
The first step is to hide the constructor by making it private. This will prevent other classes from instantiating the singleton class.

> ❗ Python does not have access modifiers like Java. For now, we will throw an error if the constructor is called.

```
class Database:
    def __init__(self):
        raise RuntimeError("Cannot instantiate Database class")
```

Step 2 - Global access point

The above code restricts the instantiation of the Database class. Now, we need to provide a global access point to get the instance of the Database class. We can do this by creating a static method that returns the instance of the Database class. If the instance does not exist, it should create the instance and then return it.

```
class Database:
    _instance = None

    def __init__(self):
        raise RuntimeError("Cannot instantiate Database class")

    @classmethod
    def get_instance(cls):
        return cls._instance
```

💡 The `@classmethod` decorator is used to create a class method. A class method is a method that is bound to the class and not the object of the class. It takes the class as the first argument. The `cls` argument is used to access the class attributes and methods.

The difference between a class method and a static method is that a class method can access and modify the class state. A static method cannot access or modify the class state.

Step 3 - Singleton logic

To implement the `get_instance()` method, we need to create a static variable of the Database class. This variable will hold the instance of the Database class. We will initialize this variable to null. The `get_instance()` method will check if the instance variable is null. If it is null, it will create a new instance of the Database class and assign it to the instance variable. Finally, it will return the instance variable. **This is known as lazy initialization**.

```
class Database:
    _instance = None

    def __init__(self):
        raise RuntimeError("Cannot instantiate Database class")

    @classmethod
    def get_instance(cls):
        if cls._instance is None:
            cls._instance = cls.__new__(cls)
        return cls._instance
```

💡 The `__new__()` method is an example of a dunder method. Dunder methods are special methods that are surrounded by double underscores.

The `__new__()` method is called when an object is created. It is responsible for creating the object and returning it. The `__init__()` method is called after the object is created. It is responsible for initializing the object.

A more pythonic implementation

The above implementation is the one proposed by the Gang of Four. However, it is not very pythonic since it requires private access modifiers. While we can raise an error, it is not an ideal solution. As mentioned, Python exposes the `__new__()` dunder method to create an object. This method can be used for alternative initialization. We can use this method to create a singleton object.

```
class Database:
    _instance = None

    def __new__(cls):
        if cls._instance is None:
            cls._instance = super().__new__(cls)
        return cls._instance
```

In the above implementation, we are overriding the `__new__()` method to check if the instance variable is null. If it is null, we are creating a new instance of the Database class and assigning it to the instance variable. Finally, we are returning the instance variable.

To create an object above we are calling the `__new__()` method on the super class. The super class is the class from which the Database class inherits. In this case, the super class is the `object` class. The `object` class is the base class of all classes in Python.

Now if we try to create an object of the Database class, we will get the same instance every time.

```
db1 = Database()
db2 = Database()

print(db1 is db2) # True
```

Using a metaclass

> 💡 A metaclass is a class whose instances are classes. In other words, a metaclass is a class that creates a class. A metaclass is also known as a class factory.

Another pythonic implementation of the singleton pattern is to use a metaclass. We can create a metaclass that will create a singleton class. It exposes a `__call__()` dunder method that will create an instance of the singleton class. The `__call__()` dunder method is called when an object is created. It is responsible for creating the object and returning it.

To create a metaclass, we need to inherit from the `type` class. The `type` class is the metaclass of all classes in Python. The `type` class is responsible for creating all classes in Python.

```
class SingletonMeta(type):
    _instances = {}

  def __call__(cls):
    if cls not in cls._instances:
      cls._instances[cls] = super().__call__()
    return cls._instances[cls]
```

Here, we create a metaclass called `SingletonMeta`. It has a static variable called `_instances` that will hold the instances of the singleton class. We are using a dictionary to store the instances since we can reuse this metaclass for different singleton classes.

The overriden `__call__()` dunder method checks if the class is present in the dictionary. If it is not present, it creates a new instance of the class and adds it to the dictionary. Finally, it returns the instance of the class.

Now, we can create a singleton class by inheriting from the `SingletonMeta` metaclass.

```
class Database(metaclass=SingletonMeta):
    pass
```

Now if we try to create an object of the Database class, we will get the same instance every time.

```
db1 = Database()
db2 = Database()

print(db1 is db2) # True
```

Thread safety

The above code is not thread-safe. If two threads create an object at the same time, both threads will check if the instance variable is null. Both threads will find that the instance variable is null and will create a new instance of the Database class. This will result in two instances of the Database class.

To make the above code thread-safe, we will need to synchronise the creation of the instance of the Database class. This can be done using the Lock class from the threading module.

```
from threading import Lock

class SingletonMeta(type):
    _instances = {}
    _lock: Lock = Lock()

    def __call__(cls):
        with cls._lock:
            if cls not in cls._instances:
                cls._instances[cls] = super().__call__()
        return cls._instances[cls]
```

Here, we are using the `with` statement to acquire the lock. The `with` statement is used to wrap the execution of a block of code with methods defined by a context manager. The lock will be released when the execution of the block of code is finished. This ensures that only one thread can create an instance of the Database class at a time.

Now, if two threads try to create an instance of the Database class at the same time, one thread will acquire the lock and create an instance of the Database class. The other thread will wait for the lock to be released. Once the lock is released, it will acquire the lock, but it will find that the instance variable is not null. It will return the existing instance of the Database class.

Double-checked locking

The above code is thread-safe, however it is not efficient.
The lock is only required when the instance variable is null. If the instance has already been created, there is no need to acquire the lock. This is where double-checked locking comes in.

💡Double-checked locking is a software design pattern used to reduce the overhead of acquiring a lock by first testing the locking criterion (the instance variable) without actually acquiring the lock. Only if the locking criterion check indicates that locking is required does the actual locking logic proceed.

So we will implement the double-checked locking pattern as follows:

1. Check if the instance variable is null. If it is not null, return the instance variable.
2. If the instance variable is null, enter the synchronized block.
3. Check the instance variable again. If it is still null, create a new instance of the Database class and assign it to the instance variable.

The double check is done to ensure that only one thread creates an instance of the Database class. If two threads enter the synchronized block at the same time, one thread will create an instance of the Database class and assign it to the instance variable. The other thread will find that the instance variable is not null and will return the existing instance of the Database class.

```
from threading import Lock

class SingletonMeta(type):
    _instances = {}
    _lock: Lock = Lock()

    def __call__(cls):
        if cls not in cls._instances: # FIRST CHECK
            with cls._lock:
                if cls not in cls._instances: # SECOND CHECK
                    cls._instances[cls] = super().__call__()
        return cls._instances[cls]
```

Summary

- The singleton pattern is a creational design pattern that lets you ensure that a class has only one instance, while providing a global access point to this instance.
- Use cases of singleton pattern
  - Shared resource like database connection, logging mechanism
  - Object that should be instantiated only once like configuration object
- Hide the constructor of the singleton class by making it private so that other classes cannot instantiate the singleton class.
- Add a static method that returns the instance of the singleton class. If the instance does not exist, it should create the instance and then return it.

- Pythonic implementation
  - Use `__new__()` dunder method to create an object
  - Use a metaclass to create a singleton class
- Thread safety
  - Make the object creation synchronised
  - Use double-checked locking.

🖥️ Code

- Simple singleton (https://github.com/scaleracademy/lld-python/blob/main/design-patterns/src/creational/singleton/simple/database.py)
- Pythonic (https://github.com/scaleracademy/lld-python/blob/main/design-patterns/src/creational/singleton/pythonic/database.py)
- Metaclass (https://github.com/scaleracademy/lld-python/blob/main/design-patterns/src/creational/singleton/metaclass/singleton.py)
- Double Checked Locking (https://github.com/scaleracademy/lld-python/blob/main/design-patterns/src/creational/singleton/threadsafe/singleton.py)

Reading list

- Dunder methods (https://www.pythonmorsels.com/what-are-dunder-methods/)
- Python metaclasses (https://realpython.com/python-metaclasses/)
- Global object pattern (https://python-patterns.guide/python/module-globals/)