Python:
Master the Art of Design Patterns

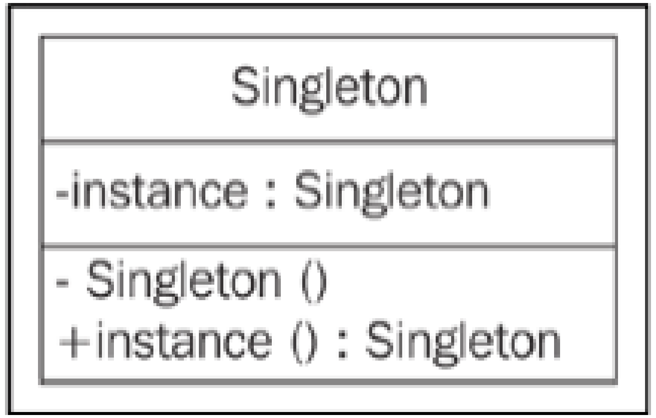
1. The Singleton Design Pattern

Understanding the Singleton design pattern

- Singleton provides you with a mechanism **to have one, and only one**, object of a given type and provides a global point of access.
- Hence, Singletons are typically used in cases such as **logging** or **database operations**, printer spoolers, and many others, where there is a need to have only one instance that is available across the application to avoid conflicting requests on the same resource.
- **For example, we may want to use one database object to perform operations on the DB to maintain data consistency** or one object of the logging class across multiple services to dump log messages in a particular log file sequentially.

Understanding the Singleton design pattern

- In brief, the intentions of the Singleton design pattern are as follows:
  - ✓ Ensuring that one and only one object of the class gets created
  - ✓ Providing an access point for an object that is global to the prograı
  - ✓ Controlling concurrent access to resources that are shared
- The following is the UML diagram for Singleton:



- A simple way of implementing Singleton is **by making the constructor private and creating a static method** that does the object initialization.

Implementing a classical Singleton in Python

- Here is a sample code of the Singleton pattern in Python v3.5. In this example, we will do two major things:

    1. We will allow the creation of only one instance of the Singleton class.

    2. If an instance exists, we will serve the same object again.

```
class Singleton(object):
    def __new__(cls):
       if not hasattr(cls, 'instance'):
         cls.instance = super(Singleton, cls).__new__(cls)
       return cls.instance

s = Singleton()
print("Object created", s)

s1 = Singleton()
print("Object created", s1)
```

* The hasattr method (Python's special method to know if an object has a certain property)

```
Object created <__main__.Singleton object at 0x00000214420459C0>
Object created <__main__.Singleton object at 0x00000214420459C0>
```

Lazy instantiation in the Singleton pattern

- One of the use cases for the **Singleton pattern is lazy instantiation**.
- For example, **in the case of module imports, we may accidently create an object even when it's not needed**.
- Lazy instantiation makes sure that the object gets created when it's actually needed.
- Consider lazy instantiation as the way to work with reduced resources and create them only when needed.

Lazy instantiation in the Singleton pattern

```python
class Singleton:

    __instance = None

    def __init__(self):
        if not Singleton.__instance:
            print(" __init__ method called..")
        else:
            print("Instance already created:", self.getInstance())

    @classmethod
    def getInstance(cls):
        if not cls.__instance:
            cls.__instance = Singleton()
        return cls.__instance

s = Singleton()                                        ## class initialized, but object not created
print("Object created", Singleton.getInstance()) ## Gets created here
s1 = Singleton()                                         ## instance already created
```

The Monostate Singleton pattern

- GoF's Singleton design pattern says that there should be one and only one object of a class.
- However, typically what a programmer needs is to have instances sharing the same state.
- Developers should be bothered about the state and behavior rather than the identity.
- As the concept is based on all objects sharing the same state, it is also known as the Monostate pattern.

The Monostate Singleton pattern

```python
class Borg:
    __shared_state = {"1":"2"}

    def __init__(self):
        self.x = 1
        self.__dict__ = self.__shared_state
        pass

b = Borg()
b1 = Borg()
b.x = 4

print("Borg Object 'b': ", b)          ## b and b1 are distinct objects
print("Borg Object 'b1': ", b1)
print("Object State 'b':", b.__dict__) ## b and b1 share same state
print("Object State 'b1':", b1.__dict__)
```

```python
class Borg(object):
     _shared_state = {}

     def __new__(cls, *args, **kwargs):
       obj = super(Borg, cls).__new__
                           (cls, *args, **kwargs)

       obj.__dict__ = cls._shared_state
       return obj
```

The following is the output of the preceding snippet:

```
Borg Object 'b':  <__main__.Borg object at 0x102078da0>
Borg Object 'b1':  <__main__.Borg object at 0x102078dd8>
Object State 'b': {'x': 4, '1': '2'}
Object State 'b1': {'x': 4, '1': '2'}
```

Singletons and metaclasses

- A metaclass is a class of a class, which means that the class is an instance of its metaclass.
- With metaclasses, programmers get an opportunity to create classes of their own type from the predefined Python classes.
- For instance, if you have an object, MyClass, you can create a metaclass, MyKls, that redefines the behavior of MyClass to the way that you need. Let's understand them in detail.

- In Python, everything is an object. If we say a=5, then type(a) returns , which means a is of the int type. However, type(int) returns , which suggests the presence of a metaclass as int is a class of the type type

Singletons and metaclasses

- The definition of class is decided by its metaclass, so when we create a class with class A, Python creates it by A = type(name, bases, dict):
  - ✓ name: This is the name of the class
  - ✓ base: This is the base class
  - ✓ dict: This is the attribute variable

```
class MyInt(type):
    def __call__(cls, *args, **kwargs):
        print("***** Here's My int *****", args)
        print("Now do whatever you want with these objects...")
        return type.__call__(cls, *args, **kwargs)

class int(metaclass=MyInt):
    def __init__(self, x, y):
        self.x = x
        self.y = y

i = int(4,5)
```

The following is the output of the preceding code:

```
***** Here's My int ***** (4, 5)
Now do whatever you want with these objects...
```

Singletons and metaclasses

- metaclasses: The preceding philosophy is used in the Singleton design pattern as well.
- **As the metaclass has more control over class creation and object instantiation**, it can be used to create Singletons. (Note: To control the creation and initialization of a class, metaclasses override the __new__ and __init__ method.

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

    def __call__(cls, *args, **kwargs):
        if cls not in cls._instances:
            cls._instances[cls] = super(MetaSingleton, cls).__call__(*args, **kwargs)
        return cls._instances[cls]

class Logger(metaclass=MetaSingleton):
    pass

logger1 = Logger()
logger2 = Logger()
print(logger1, logger2)
```

A real-world scenario – the Singleton pattern, part 1

- As a practical use case, we will look at a database application to show the use of Singletons. Consider an example of **a cloud service that involves multiple read and write operations on the database**.
- The complete cloud service is split across multiple services that perform database operations. An action on the UI (web app) internally will call an API, which eventually results in a DB operation.
- It's clear that the shared resource across different services is the database itself. So, if we need to design the cloud service better, the following points must be taken care of:
  - ✓ Consistency across operations in the database–one operation shouldn't result in conflicts with other operations
  - ✓ Memory and CPU utilization should be optimal for the handling of multiple operations on the database

A real-world scenario – the Singleton pattern, part 1

```
import sqlite3

class MetaSingleton(type):

    _instances = {}
    def __call__(cls, *args, **kwargs):
        if cls not in cls._instances:
            cls._instances[cls] = super(MetaSingleton, cls).__call__(*args, **kwargs)
        return cls._instances[cls]


class Database(metaclass=MetaSingleton):
    connection = None
    def connect(self):
        if self.connection is None:
            self.connection = sqlite3.connect("db.sqlite3")
            self.cursorobj = self.connection.cursor()
        return self.cursorobj

db1 = Database().connect()
db2 = Database().connect()

print ("Database Objects DB1", db1)
print ("Database Objects DB2", db2)
```

The output of the preceding code is given here:

```
Database Objects DB1 <sqlite3.Cursor object at 0x102464570>
Database Objects DB2 <sqlite3.Cursor object at 0x102464570>
```

A real-world scenario – the Singleton pattern, part 2

- Let's consider another scenario where we implement **health check services** (such as Nagios) for our infrastructure.
- **We create the HealthCheck class, which is implemented as a Singleton.**
- We also maintain a list of servers against which the health check needs to run.
- If a server is removed from this list, the health check software should detect it and remove it from the servers configured to check.
- In the following code, **the hc1 and hc2 objects are the same as the class in Singleton**. Servers are added to the infrastructure for the health check with the addServer() method.
- First, the iteration of the health check runs against these servers.
- The changeServer() method removes the last server and adds a new one to the infrastructure scheduled for the health check.
- So, when the health check runs in the second iteration, it picks up the changed list of servers.

A real-world scenario – the Singleton pattern, part 1

```python
class HealthCheck:

    _instance = None
    def __new__(cls, *args, **kwargs):
        if not HealthCheck._instance:
            HealthCheck._instance = super(HealthCheck, cls).__new__(cls, *args, **kwargs)
        return HealthCheck._instance

    def __init__(self):
        self._servers = []

    def addServer(self):
        self._servers.append("Server 1")
        self._servers.append("Server 2")
        self._servers.append("Server 3")
        self._servers.append("Server 4")

    def changeServer(self):
        self._servers.pop()
        self._servers.append("Server 5")
```

```python
hc1.addServer()
print("Schedule health check for servers (1)..")
for i in range(4):
    print("Checking ", hc1._servers[i])


hc2.changeServer()
print("Schedule health check for servers (2)..")
for i in range(4):
    print("Checking ", hc2._servers[i])
```

The output of the code is as follows:

```
Schedule health check for servers (1)..
Checking  Server 1
Checking  Server 2
Checking  Server 3
Checking  Server 4
Schedule health check for servers (2)..
Checking  Server 1
Checking  Server 2
Checking  Server 3
Checking  Server 5
```

Drawbacks of the Singleton patter

- While Singletons are used in multiple places to good effect, there can be a few gotchas with this pattern.
- As Singletons have a global point of access, the following issues can occur:
  - ✓ **Global variables can be changed by mistake at one place and, as the developer may think that they have remained unchanged, the variables get used elsewhere in the application.**
  - ✓ Multiple references may get created to the same object. **As Singleton creates only one object, multiple references can get created at this point to the same object.**
  - ✓ All classes that are dependent on global variables get **tightly coupled as a change to the global data by one class** can inadvertently impact the other class.