Singleton Design Pattern

Instructors **Battista Biggio**, **Angelo Sotgiu** and **Leonardo Regano**

M.Sc. in Computer Engineering, Cybersecurity and Artificial Intelligence
University of Cagliari, Italy

The Singleton Design Pattern

Singleton provides a mechanism to have only one object of a given class and provides a global point of access (even for concurrent access) to resources that are shared

Use:

- one database object to perform operations on the DB
- one object to manage printer spoolers
- one object of the logging class across multiple services to dump log messages in a particular log file sequentially.
- ...

There are various ways to implement a singleton pattern.

The most common one is based on:

1. denying access to the public constructor of the class by making it *private*;
2. providing a public static method that returns the object instance (and creates it if needed). The method can keep track of whether the first object has already been instantiated, so that only one is instantiated.

However, in Python it is not possible to make the constructor[1] `__new__` private.

---

[1] Recall that in Python `__init__` is not the constructor but is the method that initializes the object, which is created by `__new__`.

```
class Singleton:

  _instance = None

  def __init__(self):
    raise RuntimeError("You cannot use the public constructor,
          call `get_instance` instead")

  def _init(self):
    # initialize the object
    pass

  @staticmethod
  def get_instance():
    if Singleton._instance is None:
      Singleton._instance = Singleton.__new__(Singleton)
      Singleton._instance._init()
    return Singleton._instance
```

```
s1 = Singleton.get_instance()
s2 = Singleton.get_instance()
print("s1 ->", hex(id(s1)))
print("s2 ->", hex(id(s2)))
```

Output

```
--------------------
s1 -> 0x7ab6cd99fe20
s2 -> 0x7ab6cd99fe20
```

A most *pythonic* solution it to override the `__new__` method.
The first object created must be kept by the class (i.e., stored on a class attribute) and returned in subsequent calls.

```
class Singleton:

  def __new__(cls):
    if not hasattr(cls, "_instance"):
      print("1st call")
      cls._instance = super().__new__(cls)
    else:
      print("already exists")
    return cls._instance
```

Instead of declaring and empty `_instance` private class attribute, we can check for its existence by using the `hasattr()` method.

Notes on overriding `__new__`

The `__new__` method is called with the class as its first argument `cls`, even if you don't explicitly include it in the method signature.

After the new object instance is created and returned, the `__init__` method is then invoked. It will receive the new object instance and all the other arguments received by the `__new__` method.

When you define `__new__` without `*args` or `**kwargs`, you are implicitly saying that the method takes only one argument (the class itself), and doesn't accept any additional arguments.
If the `__init__` expects some arguments, this will result in a `TypeError` during the object instantiation.

To resolve this, you should include `*args, **kwargs` in your method signature to accept any additional arguments.

A note about `*args`, `**kwargs`

We can use an asterisk to pass **variable-length arguments**. The arguments are then passed as a tuple. Additionally, double asterisks can be used to pass **variable-length keyword arguments**, where the arguments are passed as a dictionary.

```
def f(*args, **kwargs):
  print("variable length arguments:")

  for el in args:
    print (el)

  print("variable length keyword arguments:")
    for k in kwargs:
      print (k, "->", (kwargs[k]))

f(1, 2, 3, a=10, b=20)
```

```
Output
--------------------
variable length arguments:
1
2
3
variable length keyword arguments:
a -> 10
b -> 20
```

We instantiate a Singleton with an initial value. We want the initial value not to change if the client tries to create a second instance of the Singleton. The above code is not enough.

```
# add to the SINGLETON class:
# def __init__(self, x):
#   self.x = x

# test it with this code:
a1 = Singleton("first value")
a2 = Singleton("second value")
print(hex(id(a1)), hex(id(a2)), a1.x, a2.x )
```

We must verify two conditions: firstly, that no other Singletons have been instantiated (by checking the class attribute from `__new__`), and secondly, from the `__init__` method, that this is the first instance.

```
class Singleton:

  def __new__(cls, *args, **kwargs):
    if not hasattr(cls, "_instance"):
      print("1st call")
      cls._instance = super().__new__(cls)
    else:
      print ("already exists")
    return cls._instance

  def __init__(self, x):
    if not hasattr(self, "_initialized"):
      self.x = x
      self._initialized = True


a1 = Singleton("first value")
a2 = Singleton("second value")
```

There is another approach to achieve the same effect in a more elegant way. We use what we've learned about **metaclasses** to create a metaclass whose instances are `Singleton` classes.

When instantiating an object, the metaclass `__call__` method is invoked, which is responsible to call the `__new__` and `__init__` methods of the class.

We thus need to overwrite the `__call__` method, in order to decide if the `__new__` (and the `__init__`) methods should be called.

```
class MetaSingleton(type):

  _dict_of_instances = dict()

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


class ST1(metaclass=MetaSingleton):
  def __init__(self, x=None):
    # It will be executed only the first time!
    self.x = x


s1_a = ST1("first value")
s1_b = ST1("first value")
print("x:", s1_a.x, s1_b.x, "object address:", hex(id(s1_a)), hex(id(s1_b)))
```

Example. A Singleton to manage Database operations

```
import sqlite3


class DatabaseSingleton:

  def __new__(cls, *args, **kwargs):
    if not hasattr(cls, "_singleton"):
      print("1st call")
      cls._singleton = super().__new__(cls)
    else:
      print("already exists")
    return cls._singleton

  def __init__(self, db_name="db.sqlite3"):
    if not hasattr(self, "_initialized"):
      self._initialized = True
      self.db_name = db_name
      self.connection = sqlite3.connect(db_name)
      self.cursor = self.connection.cursor()
```

```
# Create a table if it doesn't exist
db1.cursor.execute(
  "CREATE TABLE IF NOT EXISTS example_table "
  "(id INTEGER PRIMARY KEY, value TEXT)")
db1.cursor.execute("INSERT INTO example_table (value) VALUES (?)", (10,))
db1.connection.commit()

# Select data from the table
db2.cursor.execute("SELECT * FROM example_table")
print("# Fetch all rows")
rows = db2.cursor.fetchall()

# Iterate over the rows and print the values
for row in rows:
  print(row)
db1.cursor.close()
db1.connection.close()
```

Monostate (Borg) Singleton pattern

An alternative to having only one object is to have objects that share the same state.

In Python, the `__dict__` attribute is used to store the state of every object of a class. We can assign the `__dict__` attribute to the `_shared_state` class variable.

When we create two instances, we get two different objects. However, the object states `__dict__` are the same.
We can act on the `__init__` method, or we can override the `__new__` method.

The `__new__` method is responsible for creating a new instance of the class. We can override the default `__new__` method to ensure that all class instances share the same state, stored in `_shared_state`.

Test the Borg class

`Borg1` uses the `__init__` method while Borg2 uses the `__new__` method to ensure that all class instances share the same state.

```
# use Borg1 or Borg2
b1_a = Borg1(10)   # Borg2(10)
b1_b = Borg1(20)   # Borg2(20)

print("x:", b1_a.x, b1_b.x, "object address:", hex(id(b1_a)), hex(id(b1_b)))
print("__dict__ address:", hex(id(b1_a.__dict__)), hex(id(b1_b.__dict__)))
```

```
Output
--------------------
x: 20 20 object address: 0x1052aec10 0x1052aebb0
__dict__ address: 0x1050421c0 0x1050421c0
```

Solution

```
class Borg1:

  _shared_state = {}  # Class attribute to hold shared state

  def __init__(self, x):
    self.__dict__ = self._shared_state
    # or ... = self.__class__._shared_state
    # all the instances share the same state!
    self.x = x


class Borg2:

  _shared_state = {}  # Class attribute to hold shared state

  def __new__(cls, *args, **kwargs):
    obj = super().__new__(cls)
    obj.__dict__ = cls._shared_state
    return obj


  def __init__(self, x):
    self.x = x
```

Singleton in Python

Implementing the Singleton design pattern in Python is often not needed.

A simplest solution is to use **modules**.

We can create a module containing global variables and objects, and simply import them where needed.

Initialization can be performed inside the module itself.