DESIGN PATTERNS - CHAPITRE 2

Singleton en Python

Mémoriser une identité partagée et mesurer les limites de sa portée.

Par Ali ZAINOUL · Cours gratuit

Ce que vous allez apprendre

- Mémoriser une identité partagée et mesurer les limites de sa portée.
- Distinguer instance partagée, initialisation et état global.

Prérequis. Savoir utiliser classes, héritage, composition, listes, dictionnaires et exceptions Python. Les mécanismes spécifiques sont expliqués avant leur emploi.

Singleton : intention et expérience

Singleton : contrôler l'identité, avec des limites

Un Singleton cherche à fournir une seule instance accessible par un point commun. `__new__` crée normalement l'objet ; cette version mémorise celui du premier appel. `is` compare l'identité, pas seulement les valeurs.

Ces variantes sont des démonstrations séquentielles dans un processus. Elles ne garantissent aucune unicité entre processus ni protection contre les accès concurrents. Un état global caché complique les tests : transmettre explicitement une configuration déjà créée est souvent plus simple.

La variante avec `__new__` suppose ici que la classe n'est pas sous-classée. Un cache hérité peut sinon retourner une instance de la classe de base. Elle ne protège pas non plus contre un appel explicite à un autre mécanisme de construction.

Python

```python
# example.py
class Singleton:
    _instance = None  # Private attribute to hold the single instance

    def __new__(cls, *args, **kwargs):
        if cls._instance is None:
            cls._instance = super().__new__(cls)
            # Initialization logic can be placed here
            # For example: cls._instance.setup(config)
        return cls._instance

    # You can add methods to the Singleton class
    def some_business_logic(self):
        print("Singleton is doing something!")

# Example usage:
s1 = Singleton()
s2 = Singleton()

print(s1 is s2)  # Output: True, both variables point to the same instance
s1.some_business_logic() # Output: Singleton is doing something!
```

Comparer les mécanismes d'instance partagée

Cette variante expose `get_instance()`. Le premier appel crée une configuration ; les suivants retrouvent le même dictionnaire. L'appel direct au constructeur après création lève une erreur : ce contrat diffère donc de la variante avec `__new__`.

Python

```python
# example.py
class ConfigurationManager:
    _instance = None  # Private class attribute to hold the singleton instance

    def __init__(self):
        # Ensure that the class cannot be directly instantiated after the first instance
        if ConfigurationManager._instance is not None:
            raise Exception("This class is a singleton!")
        else:
            ConfigurationManager._instance = self
            self.config = {}  # Dictionary to store configuration parameters

    @staticmethod
    def get_instance():
        # Static method to provide access to the singleton instance
        if ConfigurationManager._instance is None:
            ConfigurationManager()  # Create the instance if it doesn't already exist
        return ConfigurationManager._instance

    def set_config(self, key, value):
        # Method to set a configuration parameter
        self.config[key] = value

    def get_config(self, key):
        # Method to retrieve a configuration parameter
        return self.config.get(key)

# Example usage of the ConfigurationManager singleton
config_manager = ConfigurationManager.get_instance()
config_manager.set_config("database_url", "localhost:5432")
print(config_manager.get_config("database_url"))

config_manager2 = ConfigurationManager.get_instance()
print(config_manager2.get_config("database_url")) # Accessing the same instance as config_manager
```

```
print(config_manager is config_manager2) # Output: True, proving that it's the same instance
```

Un décorateur Python est un objet appelable qui reçoit une fonction ou une classe et retourne son remplacement. La syntaxe `@singleton` applique cette transformation à la classe juste après sa définition. Ici, le nom `Logger` désigne ensuite une fonction d'accès à l'instance : ce remplacement peut surprendre du code qui attend une classe, notamment avec `isinstance`.

`wraps`, importé de `functools`, conserve notamment le nom et la documentation de l'objet enveloppé ; il ne transforme pas cette fonction en classe.

Python

```python
# example.py
from functools import wraps

def singleton(cls):
    # Decorator function to implement the Singleton pattern
    instances = {}  # Dictionary to store the singleton instance
    @wraps(cls)
    def get_instance(*args, **kwargs):
        # Check if an instance already exists
        if cls not in instances:
            instances[cls] = cls(*args, **kwargs)  # Create the instance if it
doesn't exist
        return instances[cls]  # Return the stored instance
    return get_instance

@singleton
class Logger:
    # Example Logger class using the singleton decorator
    def __init__(self):
        self.logs = []  # List to store log messages

    def log(self, message):
        # Method to add a log message
        self.logs.append(message)
        print(f"Log: {message}")

# Example usage of the Logger singleton
logger1 = Logger()
logger1.log("First log message")

logger2 = Logger()  # This will return the same instance as logger1
logger2.log("Second log message")

print(logger1 is logger2) # Output: True, indicating that logger1 and logger2
are the same instance
```

Une métaclasse est la classe qui construit une classe. En redéfinissant son `__call__`, on intercepte ici l'appel de construction d'une instance. Ce mécanisme avancé sera détaillé plus loin. L'exemple affiche des messages : il n'ouvre aucune base de données.

Essayez deux URL distinctes : la seconde est ignorée puisque l'instance est déjà créée. Ce comportement rend ce choix inadapté à la gestion générale de connexions ; le cycle de vie, la fermeture et les transactions restent à concevoir.

Python

```python
# example.py
class Singleton(type):
    # Metaclass to implement the Singleton pattern
    _instances = {}  # Dictionary to store instances of the singleton classes

    def __call__(cls, *args, **kwargs):
        # Override the __call__ method to control instance creation
        if cls not in cls._instances:
            cls._instances[cls] = super(Singleton, cls).__call__(*args,
**kwargs)  # Create a new instance if it doesn't exist
        return cls._instances[cls]  # Return the existing instance

class DatabaseConnection(metaclass=Singleton):
    # Example DatabaseConnection class using the Singleton metaclass
    def __init__(self, database_url):
        self.database_url = database_url
        print(f"Connecting to database: {self.database_url}")

    def execute_query(self, query):
        print(f"Executing query: {query} on {self.database_url}")

# Example usage of the DatabaseConnection singleton
db_connection1 = DatabaseConnection("memory://training")
db_connection1.execute_query("users")

db_connection2 = DatabaseConnection("another_url") # This will return the same
instance as db_connection1
db_connection2.execute_query("products")

print(db_connection1 is db_connection2) # Output: True, verifying that both
variables point to the same instance
```

Vérifier et interpréter

Construisez deux accès et modifiez la configuration par le premier. Le second voit-il la modification ? Identité partagée signifie aussi état partagé. Comparez cette dépendance implicite à deux fonctions recevant explicitement le même objet : pour une configuration simple, le passage d'argument peut suffire.

Version web : https://www.clearcode.fr/courses/programming/design-patterns/creation/singleton