Implementing a Python Singleton with Decorators

September 30, 2024

#development #python

Implementing a Python Singleton with Decorators

In Python, a **singleton** is a design pattern that ensures a class has only one instance, and it provides a global point of access to that instance. This is useful when managing shared resources such as database connections, configuration objects, or logging systems where multiple instantiations would lead to inefficiencies or inconsistencies.

In this blog post, we'll discuss a Python implementation of the singleton pattern using a decorator. We'll walk through the code and explain how it works, focusing on the `_SingletonWrapper` class and the `singleton` decorator function.

What is a Decorator?

A **decorator** in Python is a function that modifies or extends the behavior of another function or class. It allows you to "wrap" a function or a class, adding functionality to it without changing its structure.

The Singleton Pattern Explained

The singleton pattern guarantees that a class will have only one instance. When the singleton instance is requested, the existing instance is returned rather than creating a new one. This is particularly important for certain application-level services that need to be shared across the entire program.

The Code Breakdown

Let's break down the code to see how it enforces the singleton behavior.

1. The `_SingletonWrapper` Class

```python
class _SingletonWrapper:
    """
    A singleton wrapper class. Its instances would be created
    for each decorated class.
    """

    def __init__(self, cls):
        self.__wrapped__ = cls
        self._instance = None
```

The `_SingletonWrapper` class serves as a wrapper around the original class that is being decorated.

- The `__init__` method takes a class (`cls`) and stores it in the `__wrapped__` attribute.
- It also initializes `_instance` to `None`. This is the attribute that will store the single instance of the wrapped class.

2. The `__call__` Method

```python
    def __call__(self, *args, **kwargs):
        """Returns a single instance of decorated class"""
        if self._instance is None:
            self._instance = self.__wrapped__(*args, **kwargs)
        return self._instance
```

The `__call__` method is a special method in Python that makes an instance of a class callable. It is triggered when the instance is called like a function. In this case, when the singleton object is called, it checks if an instance already exists.

- If `_instance` is `None` (i.e., no instance has been created yet), it creates a new instance of the decorated class.
- If an instance already exists, it returns the same instance, ensuring that only one instance of the class is ever created.

3. The `singleton` Decorator

```python
def singleton(cls):
    """
    A singleton decorator. Returns a wrapper object. A call on that object
    returns a single instance object of decorated class. Use the __wrapped__
    attribute to access the decorated class directly in unit tests.
    """
    return _SingletonWrapper(cls)
```

This function acts as the actual decorator. When a class is decorated with `@singleton`, it wraps that class in an instance of `_SingletonWrapper`. Now, whenever the decorated class is instantiated, the wrapped version will return a single instance.

Using the Singleton Decorator

Let's look at an example of how this decorator would be used.

```python
@singleton
class Logger:
    def __init__(self):
        self.log = []

    def write_log(self, message):
        self.log.append(message)

    def read_log(self):
        return self.log
```

In this case, the `Logger` class is decorated with the `@singleton` decorator. Now, no matter how many times you try to instantiate `Logger`, you'll always get the same instance:

```python
logger1 = Logger()
logger2 = Logger()

logger1.write_log("Log message 1")
print(logger2.read_log())  # Output: ['Log message 1']

print(logger1 is logger2)  # Output: True
```

As shown above, both `logger1` and `logger2` refer to the same instance, demonstrating the singleton behavior.

Advantages of Using a Singleton

1. **Global Access**: A singleton allows for centralized control and management of an instance across an entire application. This is useful when you have shared resources like database connections or logging services.

2. **Efficiency**: Since the same instance is reused, the singleton pattern can reduce memory usage and speed up object access, especially for resource-heavy objects.

3. **Ease of Testing**: With the `__wrapped__` attribute, you can directly access the original, undecorated class in unit tests. This allows you to test the class independently of the singleton behavior, making the code easier to test and maintain.

When to Avoid Singletons

While the singleton pattern can be useful, it is not always appropriate for every situation. Overusing singletons can lead to tight coupling and make code difficult to test and maintain. They also introduce global state, which can lead to issues in multi-threaded environments or when scaling applications.

Conclusion

The singleton pattern is a powerful tool when applied correctly. Using a decorator like the one we've explored here allows you to create clean, reusable code while enforcing singleton behavior. As with any design pattern, it's important to understand the context in which it's most beneficial and apply it judiciously.

← Previous post

Next post →