SINGLETON PATTERN - COMPLETE LEARNING GUIDE **Author:** Aayush Tarwey **Date:** December 12, 2025 **Version:** 1.0 --- ## Table of Contents 1. [Quick Definition](#quick-definition) 2. [What is a Metaclass?](#what-is-a-metaclass) 3. [What is type()?](#what-is-type) 4. [How Singleton Works](#how-singleton-works) 5. [Code Walkthrough](#code-walkthrough) 6. [Real-World Examples](#real-world-examples) 7. [Interview Q&A](#interview-qa) 8. [Key Takeaways](#key-takeaways) --- ## Quick Definition ### SINGLETON PATTERN > A creational design pattern that ensures a class has only ONE instance and provides a global point of access to that instance. **Key Point:** No matter how many times you try to create an object of that class, you'll always get the SAME object back! --- ## What is a Metaclass? ### Metaclass Concept - A **metaclass** is a "**class of a class**" - Just like a class defines how objects behave, a metaclass defines how a CLASS behaves - In Python, all classes are instances of `type` by default - `type` is the built-in metaclass in Python ### Simple Analogy ``` type is to class as class is to object type → defines how to create classes class → defines how to create objects ``` ### Example ```python # When you define a class: class MyClass: pass # Python automatically does this behind the scenes: class MyClass(metaclass=type): pass ``` ### Why Use Metaclass for Singleton? When you use a metaclass, you can control **WHAT HAPPENS** when someone tries to create an instance of a class. Perfect for singleton! --- ## What is type()? ### Two Uses of type() #### 1. CHECKING THE TYPE OF AN OBJECT ```python x = 5 print(type(x)) # y = "hello" print(type(y)) # class MyClass: pass obj = MyClass() print(type(obj)) # ``` #### 2. CREATING A CLASS DYNAMICALLY ```python MyClass = type('MyClass', (), {}) # This creates a new class called MyClass! ``` **Format:** `type(name, bases, dict)` - **name:** name of the class (string) - **bases:** tuple of parent classes - **dict:** dictionary of class attributes and methods ### Example of Dynamic Class Creation ```python # Creating a simple class SimpleClass = type('SimpleClass', (), {'method': lambda self: 'Hello'}) # Creating a class with inheritance class Parent: pass ChildClass = type('ChildClass', (Parent,), {'value': 42}) ``` ### The Important Thing The default metaclass is always `type`. When you define a normal class, Python uses `type` as the metaclass automatically. --- ## How Singleton Works ### Step-by-Step Execution #### 1. WE DEFINE A METACLASS 'Singleton' ```python class Singleton(type): # This is a custom metaclass that controls class behavior ``` #### 2. WE DEFINE A DICTIONARY TO STORE INSTANCES ```python _instance = {} # This dictionary remembers all created instances ``` #### 3. WE OVERRIDE __call__ METHOD ```python def __call__(self, *args, **kwds): # This is invoked when you try to create an instance # Example: NetworkDriver() ← This triggers __call__ ``` #### 4. THE __call__ LOGIC ```python if self not in self._instance: # If this class hasn't been created yet, create it! self._instance[self] = super(Singleton, self).__call__(*args, **kwds) return self._instance[self] # Always return the stored instance (whether new or old) ``` #### 5. WE USE THE METACLASS ```python class NetworkDriver(metaclass=Singleton): # This tells Python: "Use Singleton to control how this class is created!" ``` ### Visual Flow #### NORMAL CLASS (WITHOUT SINGLETON) ``` NetworkDriver() → Creates instance 1 (ID: 123) NetworkDriver() → Creates instance 2 (ID: 456) ← Different! NetworkDriver() → Creates instance 3 (ID: 789) ← Different! ``` #### WITH SINGLETON ```

NetworkDriver() → Creates instance (ID: 123) → Stores in _instance NetworkDriver() → Returns stored instance (ID: 123) ← SAME! NetworkDriver() → Returns stored instance (ID: 123) ← SAME! ``` --- ## Code Walkthrough ### The Singleton Metaclass ```python class Singleton(type): """ A metaclass that implements the Singleton pattern. This metaclass ensures that only ONE instance of a class exists, regardless of how many times you try to create it. """ _instance = {} # Dictionary to store instances of classes using this metaclass # Key: The class itself # Value: The single instance of that class def __call__(self, *args, **kwds): """ This method is called when you try to create an instance. Example: NetworkDriver() ← This calls __call__ """ # Check if an instance of this class already exists if self not in self._instance: # If NOT, create one using the parent class's __call__ method self._instance[self] = super(Singleton, self).__call__(*args, **kwds) # Return the stored instance (whether newly created or existing) return self._instance[self] ``` ### Using the Singleton Metaclass ```python class NetworkDriver(metaclass=Singleton): """ Example class that uses the Singleton pattern. By specifying metaclass=Singleton, this class will have only ONE instance. """ def log(self): """Display the unique ID of this instance.""" print(f"NetworkDriver instance id: {id(self)}\n") ``` ### Testing the Singleton ```python driver1 = NetworkDriver() # Creates instance, stores it driver2 = NetworkDriver() # Returns same instance driver3 = NetworkDriver() # Returns same instance print(id(driver1) == id(driver2) == id(driver3)) # True! ``` --- ## Real-World Examples ### Example 1: Database Connection Manager ```python class DatabaseConnection(metaclass=Singleton): def __init__(self): print("Creating database connection...") self.connected = True def query(self, sql): if self.connected: return f"Executing: {sql}" # In your application db1 = DatabaseConnection() # Creates connection auth_module_db = DatabaseConnection() # Returns same connection payment_module_db = DatabaseConnection() # Returns same connection # All three point to the SAME connection object assert db1 is auth_module_db is payment_module_db ``` ### Example 2: Logger ```python class Logger(metaclass=Singleton): def __init__(self): self.logs = [] def log(self, message): self.logs.append(message) print(f"[LOG] {message}") # Multiple modules use the same logger logger_auth = Logger() logger_payment = Logger() logger_user = Logger() logger_auth.log("User authenticated") print(logger_payment.logs) # Contains all logs from all modules! ``` ### Example 3: Configuration Manager ```python class Config(metaclass=Singleton): def __init__(self): self.settings = { 'debug': True, 'timeout': 30, 'database_url': 'localhost:5432' } def get(self, key): return self.settings.get(key) # Entire application uses same config config = Config() config_elsewhere = Config() # Same instance! ``` --- ## Interview Q&A ### Q1: What is the Singleton pattern? **A:** It's a creational design pattern that restricts the instantiation of a class to a single object. No matter how many times you try to create an instance, you'll always get the same object. ### Q2: What are real-world use cases for Singleton? **A:** - **Database connections** - You want one global connection pool - **Logger objects** - Write all logs to one centralized place - **Configuration managers** - One global config object - **Thread pools** - One pool for all threads - **Caches** - One cache for entire application - **Session managers** - One session manager across app ### Q3: What is a metaclass? **A:** A metaclass is a class whose instances are classes. It's a way to customize how classes are created. The default metaclass in Python is `type`. ### Q4: Why use a metaclass for Singleton instead of other approaches? **A:** **Pros:** - Clean and elegant implementation - Works seamlessly with inheritance - Automatic enforcement of singleton pattern - Can handle multiple classes **Cons:** - Can be confusing for beginners - More complex to understand than other approaches ### Q5: What's the difference between using `__new__` vs `__call__` for Singleton? **A:** - **`__new__`:** Controls instance creation (object construction) - **`__call__`:** Controls what happens when the class is called (instantiation) Using `__call__` in a metaclass is elegant because it intercepts the call to create instances. ### Q6: Is this Singleton pattern thread-safe? **A:** **NO!** If two threads call `NetworkDriver()` simultaneously, they might both check the dictionary at the same time and create two instances. **Solution:** Use locks for thread-safe singleton: ```python import threading class ThreadSafeSingleton(type): _instances = {} _lock = threading.Lock() def __call__(cls, *args, **kwargs): with cls._lock: if cls not in cls._instances: instance = super().__call__(*args, **kwargs) cls._instances[cls] = instance return cls._instances[cls] ``` ### Q7: What happens if I modify a Singleton instance? **A:** Since all references point to the same object, any changes are global and visible to everyone. ```python driver1 = NetworkDriver() driver2 = NetworkDriver() driver1.status = "connected" print(driver2.status) # Outputs: "connected" (same object!) ``` ### Q8: Can multiple classes use the same Singleton metaclass? **A:** **YES!** The dictionary stores instances by class name, so each class gets its own single instance. ```python class Database(metaclass=Singleton): pass class Logger(metaclass=Singleton): pass db1 = Database() db2 = Database() log1 = Logger() log2 = Logger() assert db1 is db2 # True assert log1 is log2 # True assert db1 is not log1 # True - different classes! ``` ### Q9: What is `super(Singleton, self).__call__(*args, **kwds)`? **A:** This calls the parent class's `__call__` method (from `type`). It actually creates the instance. We then store that created instance in our dictionary. ### Q10: Why use `_instance` as a dictionary instead of just a variable? **A:** Using a dictionary allows the metaclass to handle **multiple classes** that use this metaclass. Each class gets its own entry in the dictionary. ### Q11: How do I debug a Singleton issue? **A:** Print the `id()` of instances to verify they're the same: ```python driver1 = NetworkDriver() driver2 = NetworkDriver() print(f"driver1 id: {id(driver1)}") print(f"driver2 id: {id(driver2)}") print(f"Are they same? {driver1 is driver2}") ``` ### Q12: What are alternative ways to implement Singleton? **1. Using `__new__` method:** ```python class Singleton: _instance = None def __new__(cls): if cls._instance is None: cls._instance = super().__new__(cls) return cls._instance ``` **2. Using a decorator:** ```python def singleton(cls): instances = {} def get_instance(*args, **kwargs): if cls not in instances: instances[cls] = cls(*args, **kwargs) return instances[cls] return get_instance @singleton class MyClass: pass ``` **3. Using a module-level instance:** ```python # singleton.py class _Singleton: pass instance = _Singleton() ``` --- ## Key Takeaways ### ✅ What You Should Remember 1. **Singleton ensures only ONE instance** of a class exists 2. **Metaclass controls class behavior** (like how a class controls object behavior) 3. **`type` is the default metaclass** in Python 4. **`__call__` method** intercepts instance creation in a metaclass 5. **`_instance` dictionary** stores the single instance 6. **This implementation is NOT thread-safe** -

use locks for multi-threading 7. **Dictionary allows multiple classes** to use the same metaclass 8. **Useful for:** databases, loggers, config managers, caches, etc. ### 📌 Common Mistakes to Avoid - ❌ Forgetting to use `metaclass=Singleton` when defining the class - ❌ Thinking this is thread-safe (it's not!) - ❌ Modifying a singleton instance without realizing it affects all users - ❌ Using singleton when you actually need multiple instances - ❌ Confusing metaclass with inheritance ### 💡 Interview Tips - **Explain with analogies:** "It's like a bank having only one vault..." - **Mention real-world use cases:** Database connections, loggers, config - **Discuss trade-offs:** Simplicity vs thread-safety - **Know alternatives:** Decorator, `__new__`, module-level instance - **Code example ready:** Be prepared to code it on a whiteboard - **Discuss limitations:** Thread-safety, testing difficulties --- ## Quick Reference ### Creating a Singleton ```python class Singleton(type): _instance = {} def __call__(self, *args, **kwds): if self not in self._instance: self._instance[self] = super().__call__(*args, **kwds) return self._instance[self] class MyClass(metaclass=Singleton): pass ``` ### Using It ```python obj1 = MyClass() obj2 = MyClass() assert obj1 is obj2 # True! ``` ### Verifying It Works ```python print(id(obj1) == id(obj2)) # True print(obj1 is obj2) # True ``` --- **Happy Learning! 🚀 **