Python OOP Practice Tasks

Here are some practice tasks based on the Python OOP examples you provided. Each task corresponds to the concept shown in the respective example number.

1. **Simple Class:** Create a `Book` class. The constructor (`__init__`) should accept `title` and `author` as arguments and store them as instance attributes. Add a method called `get_info()` that returns a formatted string, for example: "The Great Gatsby by F. Scott Fitzgerald". Create an instance of your `Book` class and print the result of calling the `get_info()` method.

2. **Inheritance:** Create a `FictionBook` class that inherits from the `Book` class you created in Task 1. Add a `genre` attribute to the `FictionBook` class (e.g., "Science Fiction", "Mystery"). Override the `get_info()` method in `FictionBook` to include the genre in the returned string, for example: "Dune by Frank Herbert (Science Fiction)". Create an instance of `FictionBook` and test the overridden `get_info()` method.

3. **`super()` Function:** Create an `EBook` class that inherits from the `Book` class (Task 1). The `EBook` constructor should accept `title`, `author`, and `file_format` (e.g., "PDF", "EPUB"). Use the `super()` function within the `EBook`'s `__init__` method to call the parent `Book` class's constructor to initialize the `title` and `author`. Store the `file_format` as an instance attribute. Create an instance of `EBook` and verify its attributes.

4. **Properties:** Create a `Temperature` class that stores temperature internally in Celsius. Implement a property named `fahrenheit`. The getter for `fahrenheit` should calculate and return the temperature in Fahrenheit (F = C * 9/5 + 32). The setter for `fahrenheit` should take a Fahrenheit value, convert it back to Celsius (C = (F - 32) * 5/9), and store it. Add validation in the Celsius setter (or the Fahrenheit setter's conversion) to ensure the temperature cannot be set below absolute zero (-273.15 °C).

5. **Encapsulation:** Create an `Account` class representing a bank account. It should have a public attribute `account_holder` and a private attribute `__balance` initialized in the constructor. Implement public methods `deposit(amount)` and `withdraw(amount)` that modify the private `__balance`. Ensure withdrawal doesn't allow the balance to go below zero. Add a public method `get_balance()` that returns the value of the private `__balance`. Avoid direct access to `__balance` from outside the class.

6. **Polymorphism:** Write a function `display_area(shape)` that accepts any object which has an `calculate_area()` method and prints a message like "The area is: [area]". Create two different classes, `Rectangle` (with `width` and `height`) and `Circle` (with `radius`), both implementing a `calculate_area()` method. Instantiate both classes and pass the objects to your `display_area` function to demonstrate polymorphism.

7. **Abstract Base Classes (ABC):** Define an Abstract Base Class named `Vehicle` using the `abc` module. It should have an abstract method `start_engine()`. Create two concrete subclasses, `Car` and `Motorcycle`, that inherit from `Vehicle` and provide their own implementation for the `start_engine()` method (e.g., print "Car engine started Vroom Vroom" or "Motorcycle engine started Rrrrumble"). Verify that you cannot create an instance of the `Vehicle` ABC directly, but you can create instances of `Car` and `Motorcycle` and call their `start_engine()` methods.

8. **Class Methods and Static Methods:** Create a `StringUtils` class. Add a static method `is_palindrome(text)` that returns `True` if the given string `text` is a palindrome (reads the same forwards and backward), ignoring case and spaces, and `False` otherwise. Add a class method `get_info(cls)` that returns a string describing the class, like "This is a utility class: StringUtils". Test both methods by calling them directly on the class (`StringUtils.is_palindrome(...)`, `StringUtils.get_info()`).

9. **Operator Overloading:** Create a `Vector2D` class representing a 2D vector with `x` and `y` coordinates. Overload the addition operator (`+`) using the `__add__` special method so that adding two `Vector2D` objects results in a new `Vector2D` object representing their sum. Also, overload the multiplication operator (`*`) using `__mul__` so that multiplying a `Vector2D` object by a scalar (number) results in a new `Vector2D` object with scaled coordinates.

10. **String Representations:** Enhance the `Book` class from Task 1 or Task 2. Implement the `__str__` special method to return a user-friendly string representation (you can reuse the `get_info()` logic). Implement the `__repr__` special method to return an unambiguous string representation that could ideally be used to recreate the object, such as `Book('The Great Gatsby', 'F. Scott Fitzgerald')`.

11. **Composition:** Create an `Engine` class with attributes like `horsepower` and `type` (e.g., "Petrol", "Electric"). Create a `Car` class (you can reuse the one from Task 7 or make a new one). Instead of inheriting, give the `Car` class an `engine` attribute which is an instance of the `Engine` class (passed during `Car`'s initialization). Add a

method to the `Car` class, like `display_engine_specs()`, that accesses and displays information about its `engine` attribute.

12. **Multiple Inheritance:** Create a `Swimmer` class with a method `swim()` that prints "Swimming...". Create a `Flyer` class with a method `fly()` that prints "Flying...". Now, create an `AmphibiousPlane` class that inherits from both `Swimmer` and `Flyer`. Create an instance of `AmphibiousPlane` and call both its `swim()` and `fly()` methods to show it has capabilities from both parent classes.
13. **Decorators within Classes:** Revisit the `StringUtils` class from Task 8. Ensure the `is_palindrome` method is decorated with `@staticmethod` and `get_info` is decorated with `@classmethod`. Add a regular instance method `reverse_string(self, text)` that returns the reversed string. Create an instance of `StringUtils` and call the instance method. Verify that the static and class methods can still be called on the class itself.
14. **Singleton Pattern:** Implement a `Logger` class as a singleton. This class should manage logging operations (you can just have it store log messages in a list for simplicity). Ensure that no matter how many times you try to create an instance of `Logger`, you always get the same instance. Add a method `log(message)` to add a message to the log and a method `show_log()` to print all logged messages. Test by getting the instance multiple times, logging messages through different variables referencing the instance, and showing the log to confirm all messages are there and only one logger exists.
15. **Mixin Classes:** Create a `TimestampMixin` class. This mixin should provide a method `get_creation_timestamp()` that returns the time the object was created (hint: store `datetime.now()` in the `__init__` of the classes using the mixin, or within the mixin itself if designed carefully). Create two unrelated classes, `Document` (with a `title` attribute) and `User` (with a `username` attribute). Have both `Document` and `User` inherit from `TimestampMixin` (and `object`). Instantiate both `Document` and `User` and demonstrate that both have access to the `get_creation_timestamp()` method provided by the mixin.