Advanced Python OOP: Custom Descriptors Workbook

This workbook explores the power of the Descriptor Protocol (`__get__` , `__set__` , `__delete__` , and `__set_name__`). Descriptors are the underlying technology behind `property` , `classmethod` , and `staticmethod` , offering a way to generalize attribute access logic across multiple classes.

Exercise 1: The Typed Attribute Validator

Descriptor/Problem: Create a descriptor called `Typed` that enforces a specific data type for an attribute.

1. It should accept a `type` (e.g., `int` , `str`) during initialization.
2. It must use `__set_name__` to automatically discover the name of the attribute it is assigned to in the managed class.
3. If a value of the wrong type is assigned, it should raise a `TypeError` with a descriptive message.
4. Store the data in the instance's `__dict__` to avoid memory leaks associated with weak references or global dictionaries.

Solution:

```
class Typed:
    def __init__(self, expected_type):
        self.expected_type = expected_type
        self.storage_name = None

    def __set_name__(self, owner, name):
        # Automatically called when the class is created
        self.storage_name = name

    def __set__(self, instance, value):
        if not isinstance(value, self.expected_type):
            raise TypeError(
                f"Expected {self.expected_type.__name__} for '{self.storage_name}
                f"got {type(value).__name__}."
            )
        # Store directly in instance __dict__ to avoid recursion
        instance.__dict__[self.storage_name] = value

    def __get__(self, instance, owner):
        if instance is None:
            return self
        return instance.__dict__.get(self.storage_name)

# Usage
class Profile:
    name = Typed(str)
    age = Typed(int)

p = Profile()
p.name = "Alice"  # Works
```

```
# p.age = "25"   # Raises TypeError
```

Conceptual Notes:

- `__set_name__`: Introduced in Python 3.6, this eliminates the need to pass the attribute name string (like `"name"`) manually to the descriptor's constructor.
- **Instance Storage**: By using `instance.__dict__`, we ensure that the data lives with the object instance, not on the descriptor itself (which is a class-level singleton).

Exercise 2: The Non-Negative Range Bounder

Descriptor/Problem: Design a descriptor `BoundedNumber` that ensures a numeric value stays within a range `[min_val, max_val]`.

1. The descriptor should be reusable for multiple attributes in the same class.
2. If the user attempts to set a value outside the range, the descriptor should "clamp" the value to the nearest boundary (e.g., if max is 100 and user sets 110, it becomes 100) instead of raising an error.
3. Provide a way to retrieve the "raw" value attempted if needed (optional design choice).

Solution:

```
class BoundedNumber:
    def __init__(self, min_val, max_val):
        self.min_val = min_val
        self.max_val = max_val
        self.name = None

    def __set_name__(self, owner, name):
        self.name = name

    def __get__(self, instance, owner):
        if instance is None: return self
        return instance.__dict__.get(self.name, self.min_val)

    def __set__(self, instance, value):
        if not isinstance(value, (int, float)):
            raise TypeError("Value must be a number")

        # Clamping logic
        clamped = max(self.min_val, min(value, self.max_val))
        instance.__dict__[self.name] = clamped

class GameCharacter:
    health = BoundedNumber(0, 100)
    mana = BoundedNumber(0, 50)

hero = GameCharacter()
hero.health = 150  # Sets to 100
hero.mana = -10    # Sets to 0
print(f"Health: {hero.health}, Mana: {hero.mana}")
```

Conceptual Notes:

- **Clamping vs. Validating**: This pattern shows how descriptors can transform input data silently to maintain object integrity, a common pattern in game development and UI sliders.

Exercise 3: Lazy Computed Attribute (Cached Property)

Descriptor/Problem: Implement a `LazyProperty` descriptor.

1. It should take a function (method) as an argument.
2. The first time the attribute is accessed, it should run the function, store the result on the instance, and return it.
3. Subsequent accesses should return the cached value immediately without re-running the function.
4. It must work like a read-only attribute until the cache is invalidated.

Solution:

```
import time

class LazyProperty:
    def __init__(self, function):
        self.function = function
        self.name = function.__name__

    def __get__(self, instance, owner):
        if instance is None:
            return self

        # Calculate the value
        value = self.function(instance)
        # Overwrite the descriptor on the instance with the actual value
        # This works because instance attributes take precedence over
        # non-data descriptors (descriptors without __set__)
        instance.__dict__[self.name] = value
        return value

class DeepAnalysis:
    @LazyProperty
    def heavy_computation(self):
        print("Running expensive calculation...")
        time.sleep(2)
        return 42

analysis = DeepAnalysis()
print(analysis.heavy_computation) # Takes 2 seconds
print(analysis.heavy_computation) # Instant (cached)
```

Conceptual Notes:

- **Non-Data Descriptors**: If a descriptor does *not* define `__set__`, it is a "non-data descriptor." Python's attribute lookup priority means an entry in `instance.__dict__` will override the descriptor's `__get__` for future calls.

Exercise 4: Validated Cross-Dependency (The Logic Gate)

Descriptor/Problem: Create a system where one descriptor depends on another. Implement a `Constant` descriptor and a `Verified` descriptor.

1. `Constant` attributes can only be set once.
2. `Verified` attributes require a specific "Admin Key" (another attribute in the class) to be set to a specific value before they can be modified.
3. If the Admin Key is incorrect, modification should raise a `PermissionError`.

Solution:

```
class Constant:
    def __init__(self):
        self.name = None

    def __set_name__(self, owner, name):
        self.name = name

    def __set__(self, instance, value):
        if self.name in instance.__dict__:
            raise AttributeError(f"'{self.name}' is constant and cannot be reset.
        instance.__dict__[self.name] = value

class Verified:
    def __init__(self, required_key):
        self.required_key = required_key
        self.name = None

    def __set_name__(self, owner, name):
        self.name = name

    def __set__(self, instance, value):
        # Accessing another attribute on the instance
        current_key = getattr(instance, 'admin_key', None)
        if current_key != self.required_key:
            raise PermissionError("Invalid Admin Key. Access Denied.")
        instance.__dict__[self.name] = value

class SecureSystem:
    admin_key = Constant()
    database_url = Verified(required_key="SECRET_123")

sys = SecureSystem()
sys.admin_key = "SECRET_123"
# sys.admin_key = "NEW_KEY" # Raises AttributeError
sys.database_url = "[https://db.prod.com](https://db.prod.com)" # Success
```

Conceptual Notes:

- **Interspectivity**: Descriptors can use `getattr` or `instance.__dict__` to check the state of *other* attributes on the same instance, allowing for complex state-machine-like behavior.

Exercise 5: The Observer/Callback Descriptor

Descriptor/Problem: Develop an `Observable` descriptor that executes a list of callback functions whenever its value changes.

1. The descriptor should allow users to "subscribe" new functions to an attribute.
2. When `__set__` is called, it should pass the `instance`, the `old_value`, and the `new_value` to all registered callbacks.
3. Handle the case where the attribute is being set for the first time (`old_value` is `None`).

Solution:

```
class Observable:
    def __init__(self):
        self.name = None
        self.callbacks = []

    def __set_name__(self, owner, name):
        self.name = name

    def subscribe(self, callback):
        self.callbacks.append(callback)

    def __get__(self, instance, owner):
        if instance is None: return self
        return instance.__dict__.get(self.name)

    def __set__(self, instance, value):
        old_value = instance.__dict__.get(self.name)
        instance.__dict__[self.name] = value

        for cb in self.callbacks:
            cb(instance, old_value, value)

# Usage
def log_change(obj, old, new):
    print(f"Update: {old} -> {new}")

class Configuration:
    port = Observable()

# Setup
cfg = Configuration()
# Accessing the descriptor via the class to call subscribe
Configuration.port.subscribe(log_change)

cfg.port = 8080 # Output: Update: None -> 8080
cfg.port = 443  # Output: Update: 8080 -> 443
```

Conceptual Notes:

- **Class-Level Registry**: Since the descriptor is defined at the class level, `self.callbacks` is shared across all instances of `Configuration`. If per-instance callbacks were required, the `callbacks` list would need to be stored inside `instance.__dict__`.
- **The `instance is None` Check**: Always include this in `__get__` so you can interact with the descriptor object itself (to call `subscribe`) rather than its managed value.