Deep Dives in OOP

Olivia Lynn
LSST-DA Data Science Fellowship Program Session 21
June 5, 2024

Table of Contents

Quick Vocab

Coupling

- The degree of dependency between classes
- Tight coupling can make maintenance harder

Cohesion

- How well methods and properties relate to the class's goal
- High cohesion = easier maintenance

Association

- Relationship between classes
- Types: one-to-one, one-to-many, many-to-one, many-to-many

Quick Vocab: Class Attribute

```python
class MyClass:
    # Mutable class attribute (list)
    mutable_list = []

    # Immutable class attribute (string)
    immutable_string = "Hello"

# Modifying mutable class attribute
obj1 = MyClass()
obj2 = MyClass()

obj1.mutable_list.append(1)
obj2.mutable_list.append(2)

print(obj1.mutable_list)  # Output: [1]
print(obj2.mutable_list)  # Output: [1, 2]
```

```python
# Attempting to modify immutable class attribute
# (raises AttributeError)
# obj1.immutable_string = "World"
# AttributeError: can't set attribute

# Accessing immutable class attribute
print(obj1.immutable_string)  # Output: Hello
print(obj2.immutable_string)  # Output: Hello
```

Decorators

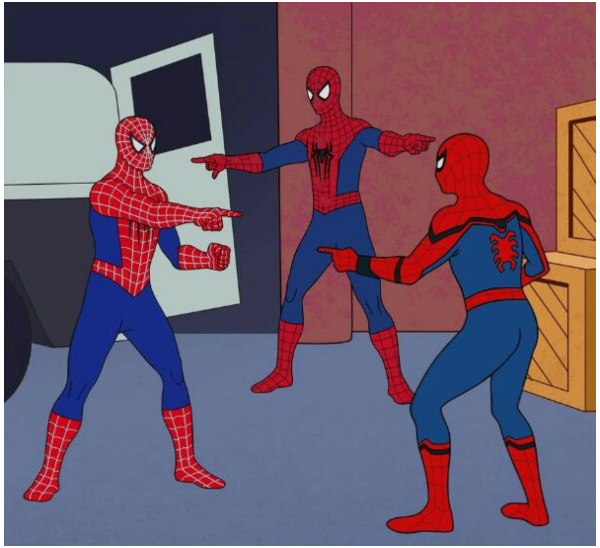
What are decorators?

- Decorators are a feature in Python that modify or enhance the behavior of functions or methods.
- Decorators are applied using the @decorator_name notation

```
@decorator_name
function_name():
    print("Hello")
```

What are decorators?

- They modify the behavior of functions without changing their source code
- Decorators are implemented as functions that take another function as an argument and return a new function
- They can capture variables from their enclosing scope using function closure



Common Use Cases

- Decorators are used for input validation, authentication, logging, and caching.
- They improve code modularity, readability, and maintainability.

(Examples incoming)

Example: Input Validation

```
def validate_input(func):
    def wrapper(*args, **kwargs):
        if any(arg < 0 for arg in args):
            raise ValueError("Input arguments must be
        non-negative")
        return func(*args, **kwargs)
    return wrapper

@validate_input
def divide(a, b):
    return a / b
```

Example: Authentication

```
def authenticate(func):
    def wrapper(*args, **kwargs):
        if not user_authenticated():
            raise PermissionError("User not authenticated")
        return func(*args, **kwargs)
    return wrapper

@authenticate
def delete_account(user_id):
    # Delete user account logic here
    pass
```

Class and Method Decorators

- Decorators can be applied to class methods, too
- They modify the behavior of methods within a class

```
# Class decorator example
def class_decorator(cls):
    class DecoratedClass(cls):
        def decorated_method(self):
            print("Decorated method")

    return DecoratedClass

# Method decorator example
def method_decorator(func):
    def wrapper(self):
        print("Before method ex.")
        func(self)
        print("After method ex.")
    return wrapper
```

```
@class_decorator
class MyClass:
    @method_decorator
    def my_method(self):
        print("Executing my_method")

# Usage
obj = MyClass()
obj.my_method()  # Before method ex.
                 # Executing my_method
                 # After method ex.
obj.decorated_method()
                 # Decorated method
```

Decorators with Arguments

- Decorators can accept arguments, known as decorator factories
- They can be customized with parameters to control their behavior

```
def log_with_level(level):
    def decorator(func):
        def wrapper(*args, **kwargs):
                print(f"[{level}] Calling {func.__name__}")
            return func(*args, **kwargs)
        return wrapper
    return decorator

@log_with_level(level="INFO")
def my_function():
    print("Executing my_function")

my_function()
```

Built-in Decorators and Libraries

Python has built-in decorators like @staticmethod, @classmethod, and @property.

(Examples incoming)

@staticmethod

- Declares a method as a static method, which can be called on the class itself without needing an instance.
- Does not have access to the class or instance attributes.

```
class MathUtility:
    @staticmethod
    def add(x, y):
        return x + y

# Usage
result = MathUtility.add(3, 5)
print(result)  # Output: 8
```

@classmethod

- Declares a method as a class method, which receives the class itself as the first argument (cls).
- Can access and modify class-level attributes.

```
class MyClass:
    class_variable = "Hello"

    @classmethod
    def print_class_variable(cls):
        print(cls.class_variable)

# Usage
MyClass.print_class_variable()
   # Output: Hello
```

@property

- Defines a method as a property getter, allowing attribute access via dot notation (obj.property) rather than method invocation (obj.property()).
- Can also define setter and deleter methods for property manipulation.

```
class Circle:
    def __init__(self, radius):
        self._radius = radius

    @property
    def radius(self):
        return self._radius

    @radius.setter
    def radius(self, value):
        if value <= 0:
            raise ValueError("Radius must be
                                         positive")
        self._radius = value

# Usage
circle = Circle(radius=5)
print(circle.radius)  # Output: 5
circle.radius = 10    # Set new radius
print(circle.radius)  # Output: 10
```

@dataclass

- Added in version 3.7
- Automatically generates special methods such as __init__, __repr__, __eq__, and others, based on class variables defined in the class.
- Commonly used to create classes that primarily store data without much additional functionality.

```
from dataclasses import dataclass

@dataclass
class Person:
    name: str
    age: int

# Usage
person = Person(name="Alice", age=30)
print(person)
# Output: Person(name='Alice', age=30)
```

Decorator Chaining and Order of Execution

- Multiple decorators can be applied to a single function.
- The order of decorators affects the order of execution (innermost is applied first)

```
# Decorator for logging
def log_decorator(func):
    def wrapper(*args, **kwargs):
        print(f"Logging: {func.__name__} called with args={args}, kwargs={kwargs}")
        return func(*args, **kwargs)
    return wrapper

class MathUtility:
    @classmethod
    @log_decorator
    def add(cls, x, y):
        return x + y
```

```
# Usage
result = MathUtility.add(3, 5)
print("Result:", result)  # 8
```

Operator overloading: How To & When To

Another type of polymorphism

On Monday, we went over the most common kind of polymorphism in Python:

dynamic polymorphism aka
run-time polymorphism aka
method overriding

```
class SuperClass:
    my_method(self):
        print("Super!")

class SubClassOne(SuperClass):
    # no my_method defined

class SubClassTwo(SuperClass):
    my_method(self):
        print("Sub two!")

my_sub_one = SubClassOne()
my_sub_one.my_method() # "Super!"

my_sub_two = SubClassTwo()
my_sub_two.my_method() # "Sub two!"
```

Operator overloading = static polymorphism

__repr__ knows how to handle some input types to make them readable:

```
my_dict = dict()
my_dict['a'] = 1
my_dict['b'] = 2

repr(my_dict)  # {'a': 1, 'b': 2}
```

Operator overloading = static polymorphism

And other input types, less so:

```
class RandomClass:
    def __init__(self, name):
        self.name = name

my_instance = RandomClass("my_instance")

repr(my_instance)  # <__main__.RandomClass at 0x109386ad0>
```

Operator overloading = static polymorphism

So by adding __repr__ to our class, we are overloading the operator, and adding a new input type that it can handle:

```
class RandomClass:
    def __init__(self, name):
        self.name = name

    def __repr__(self):
        return f"RandomClass named {self.name}"

my_instance = RandomClass("my_instance")

repr(my_instance)  # RandomClass named my_instance
```

Other common operators to overload

__eq__ (==):

- Enables custom comparison of objects for equality.
- Allows you to define what it means for two objects of your class to be considered equal.

__add__ (+):

- Enables addition of objects using the + operator.
- Useful for defining custom behavior when combining objects of your class.

__getitem__, __setitem__, __delitem__ ([] indexing):

- Allows you to define behavior for getting, setting, and deleting items using square bracket notation ([]) on your objects.
- Useful for creating custom data structures that support indexing.

```
my_instance.data[index]

=> my_instance[index]
```

When to overload operators (and when to not)

> Only overload operators if it's the natural, expected thing to do and doesn't have any side effects.

> So if you make a new RomanNumeral class, it makes sense to overload addition and subtraction etc. But don't overload it unless it's natural: it makes no sense to define addition and subtraction for a Car or a Vehicle object.

Thank you @Peter from StackOverflow

Inheritance, and Why Composition May Be Preferred

Inheritance: Building on the Foundation

- **Definition:** the mechanism by which one class can inherit properties and behavior from another class
- **Benefits:** code reuse, extensibility, and promoting a hierarchical structure
- **Example:** a Car class inheriting from a Vehicle class

Composition: Assembling Objects Piece by Piece

- **Definition:** the concept of creating complex objects by combining simpler objects
- **Benefits:** flexibility, encapsulation, and avoiding the pitfalls of deep inheritance hierarchies
- **Example:** a Car class composed of Engine, Wheels, and Body objects

```python
class Engine:
    def start(self):
        print("Engine started")

class Wheels:
    def rotate(self):
        print("Wheels rotating")

class Car:
    def __init__(self, engine, wheels):
        self.engine = engine
        self.wheels = wheels

    def start(self):
        self.engine.start()

    def drive(self):
        self.wheels.rotate()
```

```python
# Creating components
car_engine = Engine()
car_wheels = Wheels()
car_body = Body(color="red")

# Creating a car using composition
my_car = Car(
        engine=car_engine,
        wheels=car_wheels
)

# Using the car
my_car.start()  # prints “Engine started”
my_car.drive()  #prints “Wheels rotating”
```

Choosing the Right Tool

Inheritance and composition are tools for achieving different goals.

- Think of containment as a **has a** relationship. A car "has an" engine, a person "has a" name, etc.
- Think of inheritance as an **is a** relationship. A car "is a" vehicle, a person "is a" mammal, etc.

Finding Balance

- There’s a saying “prefer composition over inheritance”
- Best to follow a balanced approach, where both inheritance and composition are used judiciously based on the specific requirements of each project

Note: Mixin Classes

These are technically multiple inheritance, so C is-a A and C is-a MixinB:

```
class C(A, MixinB):
    # ...

issubclass(C, A)   # True
issubclass(C, MixinB)   # True
```

Method Resolution Order (MRO)

The MRO operator returns a list of types the class is derived from, in the order they are searched for methods.

```
class A(object): pass

A.__mro__
# (<class '__main__.A'>, <type 'object'>)

class B(A): pass

B.__mro__
# (<class '__main__.B'>, <class '__main__.A'>, <type 'object'>)
```

SOLID Principles

SOLID Overview

- **Single Responsibility** Principle
- **Open/Closed** Principle
- **Liskov Substitution** Principle
- **Interface Segregation** Principle
- **Dependency Inversion** Principle

Single Responsibility Principle (SRP)

Definition: A class should have only one reason to change, meaning it should have only one responsibility or job

Benefits: improved code readability, easier maintenance, and reduced coupling

Example: A Car class responsible for managing its own state and behavior, separate from classes responsible for logging or persistence

Open/Closed Principle (OCP)

Definition: Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification

Benefits: promoting code reuse, reducing the risk of introducing bugs, and facilitating easier maintenance

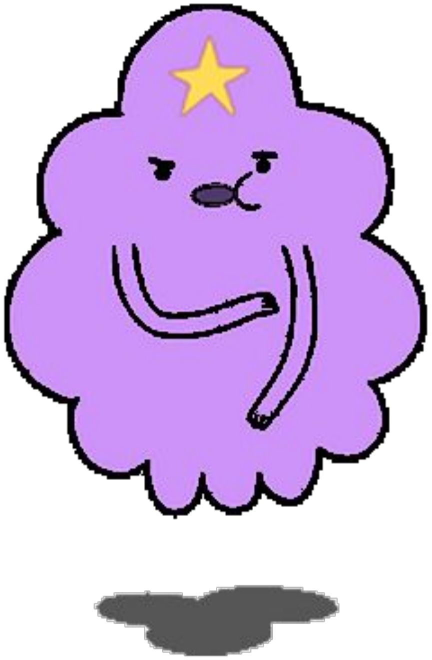
Example: Extending a rental car management system to support trucks and motorcycles by defining a Vehicle base class and creating subclasses for each vehicle type, allowing for easy extension without modifying existing code.

Liskov Substitution Principle (LSP)

Definition: Subtypes should be substitutable for their base types without altering the correctness of the program

Benefits: promoting interoperability and allowing for polymorphic behavior

Example: A Rectangle class should be substitutable for a Shape base class without breaking the behavior expected from a Shape



Interface Segregation Principle (ISP)

Definition: Clients should not be forced to depend on interfaces they do not use

Benefits: reducing the complexity of dependencies, promoting cohesion, and preventing interface bloat

Example: Designing a messaging application with separate interfaces for email, SMS, and push notifications, ensuring that clients only depend on the interfaces relevant to them and promoting interface segregation.

Dependency Inversion Principle (DIP)

Definition:

- High-level modules should not depend on low-level modules; both should depend on abstractions.
- Abstractions should not depend on details; details should depend on abstractions.

Benefits: promoting decoupling, enabling easier testing and mocking, and facilitating dependency injection

Example: Implementing a PaymentService class that depends on a PaymentGateway interface rather than concrete payment gateway classes, allowing for easier testing and flexibility in swapping out different payment gateways.

SOLID Review

- **Single Responsibility Principle**:
    - A class should have only one reason to change.
- **Open/Closed Principle**:
    - Classes should be open for extension, but closed for modification.
- **Liskov Substitution Principle**:
    - Subtypes must be substitutable for their base types.
- **Interface Segregation Principle**:
    - No client should be forced to depend on methods it does not use.
- **Dependency Inversion Principle**:
    - Depend on abstractions, not on concretions.

Questions