COP 3035

Intro Programming in Python

Summer 2024

Lecture 19 – part 1

Homework 6 – 07/19/24
Lab 9
Lab 10, Exam 4

Lecture 19 – part 2

Review

Review

Object Oriented Programming

Polymorphism

Class variables

Composition

Aggregation

Class methods

Static methods

Polymorphism

- Polymorphism is an OOP principle that allows objects of different classes to be treated as objects of a common superclass.
- In python polymorphism refers to the way in which different object classes can **share the same method** name, and those methods can be called from the same place even though a variety of different objects might be passed in.
- Polymorphism is achieved through methods that have the **same name but possibly act differently** based on which object calls them.

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

    def speak(self):
        return self.name+' says Woof!'

class Cat:
    def __init__(self, name):
        self.name = name

    def speak(self):
        return self.name+' says Meow!'

niko = Dog('Niko')
felix = Cat('Felix')

print(niko.speak())
print(felix.speak())
```

```
Niko says Woof!
Felix says Meow!
```

Class Variables

- Defined within a class but outside any instance methods
- Shared across all instances of the class
- Accessed using the class name as well as by instance references
- Ideal for storing constants and default values

```
class Circle:
    pi = 3.14

    # Circle gets instantiated with a radius (default is 1)
    def __init__(self, radius=1):
        self.radius = radius
        self.area = radius * radius * Circle.pi

    # Method for resetting Radius
    def setRadius(self, new_radius):
        self.radius = new_radius
        self.area = new_radius * new_radius * self.pi

    # Method for getting Circumference
    def getCircumference(self):
        return self.radius * self.pi * 2


c = Circle()
```

Class variable

Instance variable

Composition

Composition:

- A "has-a" relationship where a class is made up of components of another class.
- The composed object cannot exist independently of the owning class.
- Lifecycle dependency: When the owning class is destroyed, its components are also destroyed.

- Example:
  - A Car class containing a instance of a Engine class. If the car ceases to exist, the engine associated with it also cease to exist.

- Use it when you need a strong association between the container object and the contained object(s).

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

    def stop(self):
        print("Engine stopping.")

# Composition example
class Car:
    def __init__(self):
        self.engine = Engine()  # Car has-a Engine

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

    def stop(self):
        self.engine.stop()

myCar = Car()
myCar.start()
```

```
Engine starting.
```

Aggregation

Aggregation:

- A "has-a" relationship that represents ownership between two classes, but with less tightly coupled lifecycles.
- The aggregated object can exist independently of the owning class.
- Lifecycle dependency: When the owning class is destroyed, its aggregated objects can continue to exist.

- Example:
  - A Department class containing instances of a Professor class. Professors can exist without the department.
- Use it when you want to maintain a relationship between objects without enforcing a strong lifecycle dependency.

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

    def teach(self):
        return "{} is teaching".format(self.name)

class Department:
    # Aggregation
    def __init__(self, name):
        self.name = name
        self.professors = []  # Department "has-a" Professor,
                              # but Professors can exist independently

    def add_professor(self, professor):
        self.professors.append(professor)

    def get_professors(self):
        return [professor.name for professor in self.professors]

# Aggregation example
math_department = Department("Mathematics")
prof_john = Professor("John Doe")
math_department.add_professor(prof_john)

print(math_department.get_professors())
print(prof_john.teach())
```

```
['John Doe']
John Doe is teaching
```

Class Methods and Static Methods

Class Methods:

- Bound to the class rather than its object.
- Can modify the class state that applies across all instances.
- Defined with the **`@classmethod`** decorator.
- Automatically takes the class (**cls**) as the first argument.

Static Methods:

- Behave like regular functions but belong to the class's namespace.
- Do not have access to cls or self unless explicitly passed.
- Defined with the **`@staticmethod`** decorator.
- Useful for utility functions that don't access class or instance state.

```python
class MyClass:
    count = 0

    def __init__(self):
        MyClass.count += 1

    @classmethod
    def get_count(cls):
        return f"There are {cls.count} instances of MyClass."

    @staticmethod
    def utility_function(value):
        return value ** 2

# Using class method
print(MyClass.get_count())
instance = MyClass()
print(MyClass.get_count())

# Using static method
print(MyClass.utility_function(5))  # Output: 25
```

```
There are 0 instances of MyClass.
There are 1 instances of MyClass.
25
```

Lecture 19 – part 4

More on OOP

Encapsulation

- In encapsulation, the variables of a class are hidden from other classes and can be accessed only through the methods of their current class. Therefore, it is also known as data hiding.
- It promotes more secure code. It ensure data is not changed in unexpected ways.
- Python does not have strict enforcement of access modifiers like private or protected as in other languages. The convention is respected by users and enforced by the Python interpreter.

- How? - **Prefix attributes or methods with a double underscore __ to make them private.**

```python
class BankAccount:
    def __init__(self, initial_balance):
        self.__balance = initial_balance

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount
        else:
            raise ValueError("Deposit amount must be positive.")

    def withdraw(self, amount):
        if 0 < amount <= self.__balance:
            self.__balance -= amount
        else:
            raise ValueError("Insufficient balance.")

    def get_balance(self):
        return self.__balance
```

Note: it merely obfuscates their names to discourage direct access (name mangling).

ac1._BankAccount__balance = 2000000

Lecture 19 – part 4

Integration Exercise

Simplified social media model

1. Base User Class: Define **`User`** with private **`username`** and **`email`**, a class variable **`total_users`**, methods for username access, and a static method for email validation.
2. User Class Extension: Create **`PersonalAccount`** and **`BusinessAccount`** from **`User`**, adding specific attributes (**`birth_date`** for personal and **`business_name`** for business) and polymorphically overriding the post method.
3. Post Classes: Implement a general **`Post`** class with **`content`**, **`author`**, and **`likes`**. Derive **`PersonalPost`** and **`BusinessPost`** for specific post types, adding **`privacy_level`** and **`category`**, respectively.
4. Feed Class: Develop a **`Feed`** class to collect and display posts.
5. Integration and Testing: Instantiate personal and business accounts, create posts, add to feed, and display, ensuring all components integrate well.