Attributes and Transformers

October 9, 2025

Instance Attributes

Objective: To understand how to create and use instance attributes that are unique to each object of a class.

Theory: **Attributes** are variables that belong to a class. **Instance attributes** are specific to an instance (object) of a class. They store data that defines the unique state of each object. You define them within the __init__ constructor method using the self keyword. Each time a new object is created, these attributes are initialized with the values passed during instantiation.

Program:

```
# A class representing a `Dog`.
class Dog:
    # The __init__ method is the constructor.
    def __init__(self, name, age):
        # Instance attributes, unique to each dog object.
        self.name = name
        self.age = age

    # A method to display the dog's information.
    def get_info(self):
        return f"Name: {self.name}, Age: {self.age}"

# Create two separate dog objects with different attributes.
dog1 = Dog("Buddy", 3)
dog2 = Dog("Lucy", 5)

# Access and print the instance attributes for each object.
print(f"Dog 1: {dog1.get_info()}")
print(f"Dog 2: {dog2.get_info()}")
```

Output:

```
Dog 1: Name: Buddy, Age: 3
Dog 2: Name: Lucy, Age: 5
```

Class Attributes

Objective: To understand how to define and use class attributes that are shared by all objects of a class.

Theory: Class attributes are attributes that are common to all instances of a class. They are defined directly inside the class body but outside of any methods. They are useful for storing data that does not change from one object to another, such as constants or default values. You can access a class attribute using either the class name (ClassName.attribute) or an instance of the class (object.attribute).

Program:

```
class Planet:
    # A class attribute representing the constant gravitational acceleration.
    gravitational_constant = 9.8  # meters/second^2

    def __init__(self, name, mass):
        self.name = name
        self.mass = mass

    # A method to display planet information.
    def get_info(self):
        return f"Planet: {self.name}, Mass: {self.mass} kg"

# Create two planet objects.
earth = Planet("Earth", 5.97e24)
mars = Planet("Mars", 6.39e23)

# Accessing the class attribute via the class name.
```

```
print(f"Gravitational constant for all planets: {Planet.gravitational_constant}")


# Accessing the class attribute via an object.
print(f"Gravitational constant for Earth: {earth.gravitational_constant}")
print(f"Gravitational constant for Mars: {mars.gravitational_constant}")
```

Output:

```
Gravitational constant for all planets: 9.8
Gravitational constant for Earth: 9.8
Gravitational constant for Mars: 9.8
```

Property Decorators as Transformers

Objective: To learn how to use Python's @property decorator to create "smart" attributes that act as both attributes and methods.

Theory: A **property** is a special kind of attribute that is managed by a method. It allows you to add getter, setter, and deleter functionality to a class attribute without changing the way you access it. This is a powerful form of data encapsulation, as it gives you control over how an attribute's value is retrieved and modified. The @property decorator turns a method into a getter, and @<property_name>.setter turns another method into a setter.

Program:

```
class Circle:
    def __init__(self, radius):
        self._radius = radius  # A "protected" attribute with a leading underscore


    # The @property decorator turns this method into a getter for 'radius'.
    @property
    def radius(self):
        print("Getting value...")
        return self._radius
    # The @radius.setter decorator defines the setter method for 'radius'.
    @radius.setter
    def radius(self, value):
        print("Setting value...")
        if value < 0:
            raise ValueError("Radius cannot be negative.")
        self._radius = value

# Create a Circle object.
my_circle = Circle(5)

# Access the property (calls the getter).
print(f"Initial radius: {my_circle.radius}")

# Set the property (calls the setter).
my_circle.radius = 10
print(f"New radius: {my_circle.radius}")

# This will raise an error because of the setter's validation.
try:
    my_circle.radius = -1
except ValueError as e:
    print(f"Error: {e}")
```

Output:

```
Getting value...
Initial radius: 5
Setting value...
Getting value...
```

```
New radius: 10
Setting value...
Error: Radius cannot be negative.
```

@classmethod and @staticmethod as Transformers

Objective: To understand the difference between instance methods, class methods, and static methods and how they transform a method's behavior.

Theory:

- An **instance method** takes self as its first parameter and operates on the instance's attributes.
- A **class method** takes cls (the class itself) as its first parameter. It is created using the @classmethod decorator and is used for methods that operate on class attributes or need to create a new instance of the class in a specific way.
- A **static method** takes neither self nor cls as its first parameter. It is created using the @staticmethod decorator and is essentially a regular function that is logically part of the class but does not depend on the class's state or the instance's state.

Program:

```
class MathOperations:
  @staticmethod
  def add(x, y):
    # A static method. It doesn't need 'self' or 'cls'.
    return x + y

  @classmethod
  def create_from_tuple(cls, numbers):
    # A class method. It takes the class 'cls' as a parameter.
    # It can create and return a new instance.
    return cls(numbers[0], numbers[1])

  def __init__(self, num1, num2):
    self.num1 = num1
```

```
        self.num2 = num2


    def multiply(self):
        # An instance method. It operates on the instance's data.
        return self.num1 * self.num2


# Calling the static method via the class.
sum_result = MathOperations.add(5, 10)
print(f"Static method result: {sum_result}")


# Calling the class method to create a new instance.
# This is a common pattern for factory methods.
op_instance = MathOperations.create_from_tuple((8, 3))


# Calling the instance method.
product_result = op_instance.multiply()
print(f"Instance method result: {product_result}")
```

Output:

```
Static method result: 15
Instance method result: 24
```