Bootcamp 134 | Python

Course 09 | OOP

Amir Hossein Chegouniyan

Head of the Technical Team at Dariche Tejarat

Lecturer of Python – Django at Maktab Sharif

Amirhossein-chegounian

Content

Object Lifecycle

- Object Creation and Destruction:
  - Explain the lifecycle of an object, from creation to destruction.
- Constructors and Destructors:
  - Review __init__ for initialization and introduce __del__ for cleanup (with examples).
- Practical Use Cases:
  - When to use destructors, e.g., for closing files or network connections.

Data Hiding and Encapsulation

- Private and Protected Attributes:
  - Using underscores to indicate private (_) and protected (__) attributes in Python.
- Encapsulation:
  - Importance of controlling access to data within a class, with examples of safe data modification.
- Getters and Setters:
  - Introduction to getter and setter methods for controlled access to private attributes.

Data Hiding and Encapsulation | How to Works

- Data Hiding:
  - The variables (attributes) are kept private or protected, meaning they are not accessible directly from outside the class. Instead, they can only be accessed or modified through the methods.
- Access through Methods:
  - Methods act as the interface through which external code interacts with the data stored in the variables. For instance, getters and setters are common methods used to retrieve and update the value of a private variable.
- Control and Security:
  - By encapsulating the variables and only allowing their manipulation via methods, the class can enforce rules on how the variables are accessed or modified, thus maintaining control and security over the data.

Data Hiding and Encapsulation | Example 1

```
class Public:
    def __init__(self):
        self.name = "John"  # Public attribute
    def display_name(self):
        print(self.name)  # Public method

obj = Public()
obj.display_name()  # Accessible
print(obj.name)  # Accessible
```

Data Hiding and Encapsulation | Example 2

```
class Protected:
    def __init__(self):
        self._age = 30  # Protected attribute

class Subclass(Protected):
    def display_age(self):
        print(self._age)  # Accessible in subclass

obj = Subclass()
obj.display_age()
```

Data Hiding and Encapsulation | Example 3

```
class Private:
    def __init__(self):
        self.__salary = 50000  # Private attribute

    def salary(self):
        return self.__salary  # Access through public method

obj = Private()
print(obj.salary())  # Works
print(obj.__salary)  # Raises AttributeError
```

Data Hiding and Encapsulation | Example 4

```
class Celsius:
    def __init__(self, temperature=0):
        self.set_temperature(temperature)

def get_temperature(self):  # getter method
        return self._temperature

def set_temperature(self, value):  # setter method
        if value < -273.15:
            raise ValueError("Temperature below -273.15 is not possible.")
        self._temperature = value
```

Properties and the @property Decorator

- Using Properties:
  - How to create properties to encapsulate getter, setter, and deleter functionality.
- @property Decorator:
  - Demonstrate how to use the @property decorator to define properties for cleaner, safer access to attributes.
- Advantages of Properties:
  - When and why to use properties instead of direct attribute access.

Properties and the @property Decorator | Example 1

```
class Celsius:
    def __init__(self, temperature=0):
        self.temperature = temperature

def get_temperature(self): # getter
    return self._temperature

def set_temperature(self, value): # setter
    if value < -273.15:
            raise ValueError("Temperature below -273.15 is not possible")
        self._temperature = value

temperature = property(get_temperature, set_temperature) # creating a property object
```

Properties and the @property Decorator | Example 2

```
class Celsius:
    def __init__(self, temperature=0):
        self.temperature = temperature

    @property
    def temperature(self):
        return self._temperature

    @temperature.setter
    def temperature(self, value):
        if value < -273.15:
            raise ValueError("Temperature below -273.15°C is not possible")
        self._temperature = value
```

Properties and the @property Decorator | Example 3

```
class Celsius:
    def __init__(self, temperature=0):
        self.temperature = temperature

    @property
    def temperature(self):
        return self._temperature

    @temperature.setter
    def temperature(self, value):
        self._temperature = value

@temperature.deleter
    def temperature(self, value):
        del self._temperature
```

Magic (Dunder) Methods & Operator Overloading

- Magic Methods Overview:
  - Explain what magic methods are and why they're useful.
- Common Magic Methods:
  - Key methods like __str__, __repr__, __len__, __eq__, and __add__.
- Operator Overloading:
  - Show how magic methods allow operators like + or == to work with custom objects, making classes more intuitive and flexible.
- Practical Examples:
  - Simple use cases, such as customizing object string representations or adding objects.

Inheritance and Multiple Inheritance

- Review of Inheritance:
  - Quick recap of single inheritance and its benefits.
- Multiple Inheritance:
  - Introduce the concept of multiple inheritance with a focus on practical applications.
- Method Resolution Order (MRO):
  - Explain MRO and how Python determines which method to call when there are multiple base classes.
- Diamond Problem:
  - Briefly discuss the diamond problem and how Python's MRO handles it.

Inheritance and Multiple Inheritance | Exmaple 1

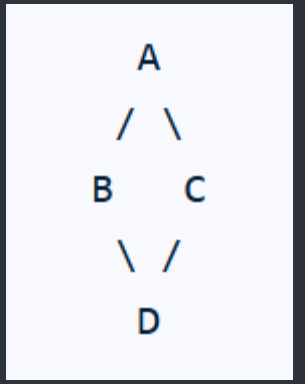
```
class A:
    def greet(self):
        print("Hello from A")


class B(A):
    def greet(self):
        print("Hello from B")
```

```
class C(A):
    def greet(self):
        print("Hello from C")


class D(B, C):
    pass


d_instance = D()
d_instance.greet()  # Output:
Hello from B
print(D.__mro__)
```



Composition vs. Inheritance

- Composition:
  - Explain composition as an alternative to inheritance, using a “has-a” relationship instead of “is-a.”
- When to Use Composition vs. Inheritance:
  - Practical examples to demonstrate when each approach is appropriate.

Abstraction

- What is Abstraction?:
  - Introduce abstraction as a concept for hiding unnecessary details.
- Abstract Classes and Methods:
  - Use abc module to demonstrate abstract classes and methods.
- Benefits of Abstraction:
  - Practical use cases for designing modular, reusable code.

Abstraction | Example 1

```
from abc import ABC, abstractmethod
class BaseClass(ABC):
    @abstractmethod
    def method_1(self):
        #empty body
        pass
```

Mixins

- What are Mixins?:
  - Introduce mixins as a lightweight, flexible way to add specific functionality to classes.
- Using Mixins in Python:
  - Examples of common mixin functionality (e.g., logging, timestamping) and when to use them.
- Advantages of Mixins:
  - Emphasize the usefulness of mixins in creating reusable and modular components without complex inheritance hierarchies.

Any question?

Next course

- Introduction to Clean Code Principles
- Basic Clean Code Practices
- Advanced Clean Code Tips
- Introduction to Virtual Environments
- Best Practices for Virtual Environments and Dependency Management