MODULE 1: CORE PYTHON & DATA

WEEK: 1 LECTURE: 9

DATE: 28/08/2025

INSTRUCTOR: HAMZA SAJID

A NEW WAY OF THINKING: INTRODUCTION TO OBJECT-ORIENTED PROGRAMMING (OOP)

Welcome back! We've mastered functions, which allow us to create reusable blocks of code. But as our programs grow, we face a new challenge: managing complexity. Today, we learn a new paradigm for structuring our code that is the foundation of modern software: **Object-Oriented Programming**. It's not just new syntax; it's a new way to think about problems.

TODAY'S AGENDA

- **The "Why" - From Data to Objects**
  - The Problem with Procedural Code: Managing Complexity
  - The Core Idea of OOP: Bundling Data and Behavior
  - Analogy: The Blueprint and the House (Classes and Objects)
  - Defining a Class: The class Keyword
  - The Constructor: Initializing an Object with __init__()
  - The Magic of self: The Instance Variable
  - Interactive Exercise: Creating Your First Dog Object
- **Bringing Objects to Life with Methods**
  - What are Methods? Functions that Belong to a Class
  - Defining and Calling Methods
  - How Methods Use self to Modify an Object's State
  - Special Methods: Making Your Objects User-Friendly with __str__()
  - *Interactive Exercise: Making the Dog Bark and Sit*

AGENDA CONT.

- **Putting It All Together & The Hands-On Lab**
  - Key Takeaways: Class vs. Object, Attribute vs. Method
  - The Power of Encapsulation
  - **Hands-On Lab: The Car Class**
  - Q&A and Wrap-up

THE "WHY" - THE NEED FOR OOP & THE CLASS BLUEPRINT

THE PROBLEM WITH PROCEDURAL CODE: A DEEPER LOOK

Imagine we're building a simple game with two characters. Using what we know, we might store their data in dictionaries:

```
player1 = {"name": "Gandalf", "hp": 100, "mana": 200}
player2 = {"name": "Aragorn", "hp": 150, "stamina": 120}

def cast_spell(player, cost):
    player["mana"] -= cost
    print(f"{player['name']} casts a spell!")

def swing_sword(player, cost):
    player["stamina"] -= cost
    print(f"{player['name']} swings their sword!")

cast_spell(player1, 30)
# swing_sword(player1, 10) # This would cause an error!
Gandalf has no stamina.
```

THE PROBLEMS:

- **No Connection:** The player data and the take_damage function are completely separate. There's nothing stopping us from accidentally calling a function with the wrong kind of data.
- **Hard to Manage:** What if we add 50 more functions? use_potion, equip_armor, level_up. Our code becomes a long, confusing list of functions and variables.
- **Not Reusable:** If we want to create a new game with a similar "character" concept, we have to copy and paste all this code and hope we don't miss anything.

THE SOLUTION: THE FOUR PILLARS OF OOP

- Object-Oriented Programming solves these problems by allowing us to create "objects" that bundle data and behavior together. It's built on four main principles:
  - **Encapsulation:** Bundling data (attributes) and the methods that operate on that data into a single object. This is the core concept we are learning today. It protects data from outside interference.
  - **Abstraction:** Hiding the complex implementation details and showing only the essential features of the object. (e.g., you press the "on" button on a TV without needing to know how the electronics work).
  - **Inheritance:** Allowing a new class to adopt the properties and methods of an existing class. (e.g., Dog and Cat classes can both inherit from a general Animal class).
  - **Polymorphism:** Allowing objects of different classes to be treated as objects of a common super class. (e.g., you can tell both a Dog and a Cat to make_sound(), and they will respond differently).

OUR FOCUS IS ON ENCAPSULATION.
ANALOGY: THE BLUEPRINT AND THE HOUSE

- A **Class** is like a blueprint. It defines the structure and capabilities of something. A blueprint for a house defines how many rooms it will have (attributes) and what you can do in it, like open doors (methods).
- An **Object** is the actual thing created from the blueprint. It's the physical house you build. You can build many houses (objects) from the same blueprint (class), but each one is a separate, unique instance.

DEFINING A CLASS: THE CLASS KEYWORD

- We use the class keyword to create the blueprint. Class names, by convention, use **PascalCase** (or CapWords), where each word is capitalized.

```
class Dog:
    pass # The 'pass' keyword means we're leaving the body empty for now
```

THE CONSTRUCTOR: __INIT__()

- How do we give our dog a name and an age when we create it? We use a special method called __init__, the **initializer** or **constructor**. This method runs automatically *one time* when you create a new object from the class.
- Methods that start and end with double underscores (__) are special in Python. They are often called "dunder" methods.

EXAMPLE

```
class Dog:
    # The initializer method
    def __init__(self, name, age):
        # We are creating the attributes for our object here
        self.name = name
        self.age = age
        print(f"A new dog named {self.name} has been created!")
```

THE MAGIC OF SELF

- What is self? It's the most important concept in OOP.
- self refers to the **specific object instance** that is being created or worked on. When you create a Dog named Fido, self.name = name means "take this specific dog, Fido, and set *its* name attribute to the value that was passed in."

EXAMPLE

```
# Creating two different Dog objects from the same Dog class blueprint
dog1 = Dog("Fido", 4)
dog2 = Dog("Lucy", 2)

# Accessing their attributes using dot notation
print(f"{dog1.name} is {dog1.age} years old.") # -> Fido is 4 years old.
print(f"{dog2.name} is {dog2.age} years old.") # -> Lucy is 2 years old.
```

IN-CLASS EXERCISE: MODELING A STUDENT

- Create a class named Student.
- In the __init__ method, accept name and student_id as parameters.
- Create the attributes self.name, self.student_id.
- Add another attribute self.courses and initialize it as an empty list [].
- Create two different Student objects and print their names and IDs.

BRINGING OBJECTS TO LIFE WITH METHODS

WHAT ARE METHODS?

- Methods are simply functions that are defined inside a class. They represent the behaviors of an object. The first parameter of every method is always self, which gives the method access to the object's attributes.

```
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age
        self.is_sitting = False # Add a new attribute to track state

    # A method to represent a behavior
    def bark(self):
        print(f"{self.name} says: Woof!")

    # A method that changes the object's internal state
    def sit(self):
        print(f"{self.name} is now sitting.")
        self.is_sitting = True

    def stand(self):
        print(f"{self.name} is now standing.")
        self.is_sitting = False
```

CALLING METHODS

- You call methods on an object instance using dot notation.

```
my_dog = Dog("Rex", 5)
my_dog.bark() # -> Rex says: Woof!


print(f"Is Rex sitting? {my_dog.is_sitting}") # -> Is Rex sitting? False
my_dog.sit()
print(f"Is Rex sitting? {my_dog.is_sitting}") # -> Is Rex sitting? True
```

SPECIAL METHODS: __STR__()

- What happens if you try to print an object directly?
- print(my_dog) -> <__main__.Dog object at 0x...> (Not very useful!)
- The __str__ method allows you to define a user-friendly string representation of your object.

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

    def bark(self):
        print(f"{self.name} says: Woof!")

    # Define the string representation
    def __str__(self):
        return f"Dog(Name: {self.name}, Age: {self.age})"

my_dog = Dog("Rex", 5)
print(my_dog) # Now this will print our custom string! -> Dog(Name: Rex, Age: 5)
```

IN-CLASS EXERCISE: BANK ACCOUNT

- Create a BankAccount class.
- The __init__ method should take an owner_name and an initial balance (defaulting to 0).
- Create a deposit(self, amount) method that adds to the balance.
- Create a withdraw(self, amount) method that subtracts from the balance. Add a check to prevent withdrawing more money than is in the account.
- Create a __str__ method that returns a string like "Account for John Doe with balance: $500".

PUTTING IT ALL TOGETHER & THE HANDS-ON LAB
KEY TAKEAWAYS

- **Class:** The blueprint (e.g., class Car:).
- **Object (Instance):** The real thing created from the blueprint (e.g., my_blue_toyota = Car(...)).
- **Attribute:** A variable that belongs to an object (data, e.g., my_blue_toyota.color). Defined in __init__.
- **Method:** A function that belongs to an object (behavior, e.g., my_blue_toyota.start()).

THE POWER OF ENCAPSULATION

- This core OOP principle means bundling the data (attributes) and the methods that operate on that data into a single, self-contained unit (the object). The BankAccount object *manages its own balance*. You don't have separate functions floating around; you just tell the object what to do (my_account.deposit(100)), and it handles its own internal state. This makes code safer, more organized, and easier to reason about.

HANDS-ON LAB: THE CAR CLASS

- **Goal:** Define a Car class that models a real-world car with its properties and actions.
- **Step-by-Step Instructions:**
  - Create a new file car_class.py

PART 1: THE BLUEPRINT (__INIT__)

- Define a class named Car.
- Create the __init__ method. It should accept brand and color as parameters.
- Inside __init__, create the attributes self.brand, self.color.
- Add a new attribute called self.is_running and initialize it to False. This will track the car's state.

PART 2: ADDING BEHAVIORS (METHODS)

- Define a method called start().
  - This method should check if self.is_running is already True. If it is, print a message like "The Toyota is already running."
  - If it's False, it should set self.is_running to True and print a message like "The Toyota starts. Vroom vroom!"
  - (Use f-strings to include the car's brand in the messages!)
- Define a method called stop().
  - This method should check if the car is running. If it is, set self.is_running to False and print a message like "The Toyota engine is turned off."
  - If it's already off, print "The Toyota is already off."

PART 3: TESTING YOUR CLASS

- Below your class definition, create two different Car objects (instances) with different brands and colors. For example:

```
my_car = Car("Toyota", "blue")
your_car = Car("Ford", "red")
```

- Print the attributes of each car to make sure they were created correctly.
- Call the start() and stop() methods on your car objects. Call them multiple times in a row to test the logic that checks if the car is already on or off.

CHALLENGE / BONUS FEATURES:

- **Add a __str__ method** to your Car class to give it a nice string representation, like "A blue Toyota".
- **Add more attributes** in __init__, such as model and year. Update the __str__ method to include them.
- **Create a state-aware method:** Add a method drive(self, distance). This method should only work if self.is_running is True. If the car is off, it should print a message telling the user to start the car first. If it's on, it should print f"Driving {distance} miles.".

THE POWER OF INHERITANCE

- This core OOP principle allows a new class (child) to reuse and extend the functionality of an existing class (parent). Instead of duplicating code, you place the common attributes and methods in the parent and let child classes inherit them. For example, a Dog and a Cat both share behaviors like eat() and sleep() from an Animal class but can also have their own unique methods like bark() or meow(). This makes code more organized, reusable, and easier to maintain.

THE POWER OF POLYMORPHISM

- Polymorphism means “many forms” — the same method name can behave differently depending on the object. It allows us to write code that can handle different types of objects in a uniform way. For example, both a Circle and a Rectangle can have an area() method, but each calculates it differently. This makes code more flexible and scalable.

HANDS-ON LAB: THE SHAPE CLASS

- Goal: Define a Shape base class and create child classes like Circle and Rectangle so the same method name (area) behaves differently across types.
- Step-by-Step Instructions :
  - Create a new file shape_class.py

PART 1 — THE BLUEPRINT (__INIT__)

- Define a class named Shape.
- Inside Shape, define a method area() that contains only pass (placeholder).

PART 2 — ADDING BEHAVIORS (METHODS)

- Define a class Circle that inherits from Shape.
- Add an __init__ that accepts radius and sets self.radius.
- Implement area() in Circle to return 3.14 * self.radius * self.radius.
- Define a class Rectangle that inherits from Shape.
- Add an __init__ that accepts width and height and sets self.width, self.height.
- Implement area() in Rectangle to return self.width * self.height.

PART 3 — TESTING YOUR CLASSES

- Create a list shapes = [Circle(5), Rectangle(4, 6)].
- Loop over shapes and call shape.area() for each item.
- Observe that each object responds to area() in its own way.

CHALLENGE / BONUS FEATURES:

- Add a Triangle class with base and height; implement area() as 0.5 * base * height.
- Implement __str__ for each shape to return a friendly description.
- Write total_area(shapes) that sums area() for any list of Shape objects.

THE POWER OF ABSTRACTION

- Abstraction is about hiding unnecessary details and exposing only what's essential. In Python, this is often done with abstract classes that define a contract. Subclasses must implement the abstract methods, but each can do so in their own way. For example, a PaymentMethod class may define pay(), but how the payment happens depends on whether it's CreditCard, PayPal, or Cash. This makes systems more organized and extensible.

HANDS-ON LAB: THE PAYMENT SYSTEM

- Goal: Define a PaymentMethod abstract base class that specifies a contract, then implement concrete payment classes that follow it.
- Step-by-Step Instructions :
  - Create a new file payment_system.py

PART 1 — THE BLUEPRINT (ABSTRACT CLASS)

- Import ABC and abstractmethod from the abc module.
- Define a class PaymentMethod that inherits from ABC.
- Inside it, declare an abstract method pay(self, amount) using @abstractmethod.

PART 2 — ADDING BEHAVIORS (METHODS)

- Define a class CreditCard that inherits from PaymentMethod.
- Implement pay(self, amount) to print "Paid $<amount> using Credit Card".
- Define a class PayPal that inherits from PaymentMethod.
- Implement pay(self, amount) to print "Paid $<amount> using PayPal".

PART 3 — TESTING YOUR CLASSES

- Create a list methods = [CreditCard(), PayPal()].
- Loop for m in methods: m.pay(100) and observe each implementation run.
- Note that you cannot instantiate PaymentMethod directly — it enforces the contract.

CHALLENGE / BONUS FEATURES:

- Create a Cash class that implements pay() (no external API required).
- Add a new abstract method refund(self, amount) to PaymentMethod and implement it in all subclasses.
- Add simple validation (e.g., raise ValueError for negative amounts).
- Extend to support currencies (add a currency attribute and format outputs).

CLASS ATTRIBUTES VS INSTANCE ATTRIBUTES

- So far, we've only seen instance attributes — data unique to each object. But sometimes, we need information that is shared by all objects of a class. These are called class attributes.
- Instance attributes → unique per object (e.g., each student has their own name, age, ID).Class attributes → shared across all objects (e.g., all students belong to the same school).This distinction helps us track data at both the individual and collective level.

CODE EXAMPLE:

- class Dog:
-     species = "Canis familiaris"
-     count = 0
-     def __init__(self, name):
-         self.name = name
-         Dog.count += 1

- print(Dog("Buddy").name, Dog.count) # Buddy 1

MODIFYING & DELETING OBJECT PROPERTIES

- Python objects are dynamic. After creating an object, you can:
- Modify existing attributes
- Add new attributes
- Delete attributes
- This flexibility makes Python objects behave like “living” entities that can evolve over time.

CODE EXAMPLE:

- class Car:
- def __init__(self): self.color="blue"
- c = Car(); c.color="red"   # modify
- c.mileage=50000   # add
- del c.mileage   # delete
- print(getattr(c,"mileage","N/A")) # N/A

PYTHON NAMING CONVENTIONS & PRIVACY

- Python does not enforce access control like private/protected in Java or C++. Instead, it uses naming conventions to signal intent:
- Public (default):
  - Accessible by everyone.
- Protected (_single_underscore):
  - Internal use only, but still accessible.
- Private (__double_underscore):
  - Python applies name mangling to discourage direct access.This helps developers respect boundaries without restricting flexibility.

CODE EXAMPLE:

- class Account:
-     def __init__(self):
-         self.owner="Alice"   # public
-         self._bal=1000   # protected
-         self.__pin=1234   # private

- a=Account()
- print(a.owner, a._bal, a._Account__pin)

HANDS-ON LAB: THE ANIMAL CLASS

- **Goal:** To apply all the core OOP concepts—Encapsulation, Inheritance, Polymorphism, and Abstraction—by creating a system of related classes. This lab will also reinforce the use of class attributes and naming conventions.
- **File:** Create a new file named zoo.py.

PART 1: THE ABSTRACT BLUEPRINT (ABSTRACTION)

- **Concept:** We'll start by defining a "contract." We know every animal in our zoo needs to eat and make a sound, but we don't know *how* yet. This is a perfect use case for an abstract base class.
- **Instructions:**
  - Import ABC and abstractmethod from the abc module.
  - Define a class named Animal that inherits from ABC.
  - In the __init__ method, it should accept a name and species. Create self.name and self.species as **instance attributes**.
  - Define a class attribute zoo_name = "The Python Zoo". Class attributes are shared by all instances.
  - Declare an abstract method make_sound(self) using the @abstractmethod decorator. It should only contain pass.
  - Declare another abstract method eat(self). It should also only contain pass.

PART 2: CREATING CONCRETE CLASSES (INHERITANCE)

- **Concept:** Now we'll create specific types of animals. These "child" classes will inherit the common attributes (name, species, zoo_name) from the Animal "parent" class and provide their own concrete implementations for the abstract methods.
- **Instructions:**
- **Create a Lion class that inherits from Animal.**
  - In its __init__, it should call the parent's __init__ method using super().__init__(name, species="Lion").
  - Implement the make_sound() method to print "Roar!".
- **Create a Monkey class that inherits from Animal.**
  - In its __init__, use super().__init__(name, species="Monkey").
  - Implement make_sound() to print "Ooh ooh aah aah!".
  - Implement eat() to print f"{self.name} peels and eats a banana.".
- **Create a Zookeeper class (this one does *not* inherit from Animal).**
  - Its __init__ should take a name.
  - Add a **protected** instance attribute self._employee_id and set it to a random number (e.g., random.randint(1000, 9999)—don't forget to import random).
  - Add a **private** instance attribute self.__salary and set it to a value like 45000.

PART 3: BRINGING THE ZOO TO LIFE (POLYMORPHISM & CLASS ATTRIBUTES)

- Concept: The power of polymorphism is that we can treat different objects in a uniform way. We can ask any Animal to make_sound(), and it will respond correctly based on its specific class. We will also see how class attributes work.
- Instructions:
- Create a list called animals and populate it with instances of your concrete classes:
- code

```
Python
animals = [
    Lion("Leo"),
    Monkey("Momo")
]
```

- Create an instance of your Zookeeper class: zookeeper = Zookeeper("Hamza").
- Demonstrate Polymorphism:
- Write a for loop that iterates through the animals list.
- Inside the loop, call animal.make_sound() and animal.eat() for each animal.
- Observe how each object responds to the same method call in its own unique way.
- Demonstrate Class Attributes:
- For each animal in your loop, print f"{animal.name} lives at {animal.zoo_name}".
- Notice that you can access zoo_name from both the instance (leo.zoo_name) and the class itself (Lion.zoo_name).

CHALLENGE / BONUS FEATURES (APPLYING ALL CONCEPTS)

- **Modify an Object:** After creating your Lion object, add a new attribute to it directly: leo.favorite_toy = "Big Red Ball". Print this new attribute to show that Python objects are dynamic.
- **Naming Conventions & Privacy:**
  - Try to print the zookeeper's protected ID: print(zookeeper._employee_id). Observe that it works, but the underscore signals that you *shouldn't* do this.
  - Try to print the zookeeper's private salary: print(zookeeper.__salary). Observe that this causes an AttributeError.
  - Now, access the "mangled" name: print(zookeeper._Zookeeper__salary). This demonstrates how name mangling works to discourage direct access.

- **Extend with a New Class:**
  - Create a Penguin class that inherits from Animal.
  - Implement its make_sound() and eat() methods.
  - Add a new method unique to penguins, like slide(), that prints f"{self.name} slides on its belly!".
  - Add a penguin to your animals list and re-run your loop.
- **Create a feed_animals Method:**
  - Add a method to your Zookeeper class called feed_animals(self, animals_list).
  - This method should take a list of Animal objects as a parameter.
  - It should loop through the list and call the eat() method on each animal, demonstrating how different classes can interact. Call this method with your list of animals.