

Day 3 Session 1:

Object-Oriented Programming Essentials - Deep Dive

This notebook introduces the fundamental concepts of Object-Oriented Programming (OOP) in Python. OOP is a programming paradigm based on the concept of "objects," which can contain data and code to manipulate that data. It is crucial for designing complex, scalable, and maintainable systems, especially in engineering and scientific applications where modeling real-world entities is common.

1. Class, Object, `__init__`, Attributes

At the core of OOP are classes and objects. A class is a blueprint, and an object is an instance of that blueprint.

1.1 What are Classes and Objects?

- **Class:** A blueprint or a template for creating objects. It defines a set of attributes (data) and methods (functions) that the objects created from it will have. Think of it like a design drawing for a satellite or a sensor.
- **Object (Instance):** A concrete realization of a class. When you create an object, you are creating a specific instance of that blueprint. For example, `Sensor_A` and `Sensor_B` could be two different objects created from a `Sensor` class.

1.2 The Constructor (`__init__`)

The `__init__` method is a special method in Python classes. It's automatically called when you create a new instance (object) of a class. It's often used to initialize the object's attributes. `self` is a convention (but not a keyword) that refers to the instance of the class itself.

1.3 Attributes (Instance Variables)

Attributes are variables that belong to an object. They store the data associated with an object. When defined within `__init__` using `self.attribute_name`, they are called **instance variables** because each instance (object) has its own copy of these variables.

Conceptual Approach:

We'll define a **Sensor** class with attributes like **sensor_id**, **sensor_type**, and **current_reading**. The **__init__** method will set these when a **Sensor** object is created. We'll also define a method **update_reading** to change the sensor's state and **get_info** to retrieve its details.

Code Implementation:

```
Python
class Sensor:
    """
    A blueprint for creating sensor objects.
    Each sensor has an ID, type, and can store a current reading.
    """
    def __init__(self, sensor_id: str, sensor_type: str, initial_reading: float
= 0.0):
        """
        The constructor method, called when a new Sensor object is created.
        Initializes the instance attributes.
        """
        self.sensor_id = sensor_id          # Instance attribute
        self.sensor_type = sensor_type      # Instance attribute
        self.current_reading = initial_reading # Instance attribute
        print(f"Sensor {self.sensor_id} ({self.sensor_type}) initialized.")

    def update_reading(self, new_reading: float):
        """
        Updates the current reading of the sensor.
        """
        print(f"Updating {self.sensor_id} reading from
{self.current_reading:.2f} to {new_reading:.2f}.")
        self.current_reading = new_reading

    def get_info(self) -> str:
        """
        Returns a formatted string with the sensor's current information.
        """
        return (f"Sensor ID: {self.sensor_id}, Type: {self.sensor_type}, "
                f"Current Reading: {self.current_reading:.2f}")

# Creating objects (instances) of the Sensor class
print("--- Creating Sensor Objects ---")
```

```

temp_sensor = Sensor(sensor_id="TS-001", sensor_type="Temperature",
initial_reading=25.5)
pressure_sensor = Sensor(sensor_id="PS-002", sensor_type="Pressure",
initial_reading=101.2)
flow_sensor = Sensor("FS-003", "Flow") # initial_reading defaults to 0.0

# Accessing attributes
print(f"\nTemp Sensor ID: {temp_sensor.sensor_id}")
print(f"Pressure Sensor Type: {pressure_sensor.sensor_type}")
print(f"Flow Sensor Reading: {flow_sensor.current_reading}")

# Calling methods
print("\n--- Calling Sensor Methods ---")
temp_sensor.update_reading(26.1)
pressure_sensor.update_reading(101.5)
flow_sensor.update_reading(5.3)

print("\n--- Sensor Information ---")
print(temp_sensor.get_info())
print(pressure_sensor.get_info())
print(flow_sensor.get_info())

# Smaller Example: Basic class and object creation
class Satellite:
    def __init__(self, name):
        self.name = name

    def launch(self):
        print(f"{self.name} launched!")

my_satellite = Satellite("GSAT-19")
my_satellite.launch()

```

Summary of Class, Object, and Attributes

Term	Description	Analogy Example
Class	Blueprint for creating objects.	Satellite Design Plan

Object	An instance of a class; a concrete entity.	Specific Satellite unit (e.g., INSAT-3D)
<code>__init__</code>	Constructor method, initializes object state.	Assembly instructions for a new satellite
Attribute	Data associated with an object (instance variable).	Satellite's ID, current altitude, fuel level
Method	Function that belongs to an object.	Satellite's "deploy_solar_panels()", "transmit_data()"

2. Instance vs. Class Variables

While instance variables belong uniquely to each object, **class variables** are shared among all instances of a class.

2.1 Instance Variables (Review)

- Defined inside methods (usually `__init__`) using `self.variable_name`.
- Each instance gets its own copy.
- Changes to an instance variable only affect that specific instance.

2.2 Class Variables

- Defined directly inside the class definition, outside of any methods.
- Shared by all instances of the class.
- Accessed using `ClassName.variable_name` or `self.variable_name` (though the former is preferred for clarity when modifying).
- Changes to a class variable (when accessed via the class) affect all instances.

Conceptual Approach:

We'll extend the **Sensor** class. `sensor.sensor_count` will be a class variable tracking how many sensor objects have been created. `sensor.STANDARD_UNIT_SYSTEM` will represent a default, shared unit system. Each **Sensor** instance will still have its unique `current_reading` (an instance variable).

Code Implementation:

Python

```
class SensorV2:
    """
    Enhanced Sensor class demonstrating instance and class variables.
    """

    # Class Variables (shared among all instances)
    sensor_count = 0 # Tracks the total number of SensorV2 instances created
    STANDARD_UNIT_SYSTEM = "SI" # Default unit system for all sensors

    def __init__(self, sensor_id: str, sensor_type: str, initial_reading: float
= 0.0):
        self.sensor_id = sensor_id
        self.sensor_type = sensor_type
        self.current_reading = initial_reading # Instance variable

        SensorV2.sensor_count += 1 # Increment class variable on each new
instance
        print(f"Sensor {self.sensor_id} ({self.sensor_type}) initialized. Total
sensors: {SensorV2.sensor_count}")

    def update_reading(self, new_reading: float):
        self.current_reading = new_reading

    def get_info(self) -> str:
        return (f"Sensor ID: {self.sensor_id}, Type: {self.sensor_type}, "
                f"Reading: {self.current_reading:.2f}, Unit System:
{self.STANDARD_UNIT_SYSTEM}")

# Creating SensorV2 objects
print("\n--- Demonstrating Instance vs Class Variables ---")
temp_sensor_v2 = SensorV2("TS-V2-001", "Temperature", 22.0)
pressure_sensor_v2 = SensorV2("PS-V2-002", "Pressure", 98.5)

# Accessing instance variables (unique to each object)
print(f"Temp Sensor Reading: {temp_sensor_v2.current_reading}")
print(f"Pressure Sensor Reading: {pressure_sensor_v2.current_reading}")

# Accessing class variables (shared)
print(f"Total sensors (via class): {SensorV2.sensor_count}")
print(f"Temp Sensor Unit System (via instance):
{temp_sensor_v2.STANDARD_UNIT_SYSTEM}")
print(f"Pressure Sensor Unit System (via instance):
```

```

{pressure_sensor_v2.STANDARD_UNIT_SYSTEM}")

# Modifying a class variable via the class name (affects all instances)
SensorV2.STANDARD_UNIT_SYSTEM = "CGS"
print(f"\n--- After changing class variable STANDARD_UNIT_SYSTEM to CGS ---")
print(temp_sensor_v2.get_info())
print(pressure_sensor_v2.get_info())

# Important: Assigning a class variable via an instance creates an instance
variable instead
flow_sensor_v2 = SensorV2("FS-V2-003", "Flow")
flow_sensor_v2.STANDARD_UNIT_SYSTEM = "FPS" # This creates a new INSTANCE
variable
print(f"\n--- After assigning STANDARD_UNIT_SYSTEM via instance for FS-V2-003
---")
print(flow_sensor_v2.get_info()) # Shows FPS
print(temp_sensor_v2.get_info()) # Still shows CGS for other instances
print(f"Class's STANDARD_UNIT_SYSTEM is still:
{SensorV2.STANDARD_UNIT_SYSTEM}") # Still CGS

# Smaller Example: Class variable for configuration
class Component:
    default_manufacturer = "Analog Devices" # Class variable

    def __init__(self, name):
        self.name = name # Instance variable

c1 = Component("OpAmp")
c2 = Component("Resistor")

print(f"\nComponent 1 Manufacturer: {c1.default_manufacturer}")
print(f"Component 2 Manufacturer: {c2.default_manufacturer}")
Component.default_manufacturer = "Linear Technology" # Change class variable
print(f"Component 1 Manufacturer (after change): {c1.default_manufacturer}")

```

Comparison: Instance vs Class Variables

Feature	Instance Variable	Class Variable
Definition	Inside <code>__init__</code> with <code>self.</code> , or in methods.	Directly inside class body.
Ownership	Unique to each instance.	Shared by all instances.
Access	<code>self.variable_name</code>	<code>ClassName.variable_name</code> or <code>self.variable_name</code>
Modification	Affects only that instance.	Affects all instances (if modified via <code>ClassName</code>).
Use Case	Object-specific data (e.g., sensor reading, ID).	Shared constants, counters, or default values.

3. Inheritance and Method Overriding

Inheritance is a fundamental OOP concept that allows a class (child/subclass) to inherit attributes and methods from another class (parent/superclass). This promotes code reusability and establishes an "is-a" relationship (e.g., a `TemperatureSensor` is a `Sensor`).

3.1 Inheritance

- **Parent Class (Base Class / Superclass):** The class being inherited from.
- **Child Class (Derived Class / Subclass):** The class that inherits from the parent. It gains all attributes and methods of the parent, and can also define its own unique attributes/methods.
- **Syntax:** `class ChildClass(ParentClass):`

3.2 Method Overriding

A subclass can provide its own implementation of a method that is already defined in its parent class. This is called **method overriding**. When the method is called on an object of the subclass, the subclass's version is executed.

Conceptual Approach:

We'll define a base **Sensor** class. Then, create two child classes: **TemperatureSensor** and **PressureSensor**.

- **TemperatureSensor** will inherit from **Sensor**, add a **unit** attribute specific to temperature (e.g., Celsius), and override the **get_info** method to include the unit.
- **PressureSensor** will also inherit from **Sensor**, add a pressure-specific **pressure_range_kpa** attribute, and override **get_info**.
- We'll use **super().__init__()** to call the parent class's constructor, ensuring base attributes are initialized.

Code Implementation:

```
Python
from typing import Tuple

class BaseSensor:
    """
    A generic base class for all sensors.
    Demonstrates inheritance.
    """
    def __init__(self, sensor_id: str, location: str, manufacturer: str =
"Generic"):
        self.sensor_id = sensor_id
        self.location = location
        self.manufacturer = manufacturer
        self.is_active = True
        print(f"BaseSensor '{self.sensor_id}' initialized at {self.location}.")

    def activate(self):
        """Activates the sensor."""
        self.is_active = True
        print(f"Sensor {self.sensor_id} activated.")

    def deactivate(self):
        """Deactivates the sensor."""
        self.is_active = False
        print(f"Sensor {self.sensor_id} deactivated.")

    def get_status(self) -> str:
        """Returns the operational status of the sensor."""
```



```

        return "Active" if self.is_active else "Inactive"

    def get_info(self) -> str:
        """Returns generic information about the sensor."""
        return (f"ID: {self.sensor_id}, Location: {self.location}, "
                f"Manufacturer: {self.manufacturer}, Status: "
                f"{self.get_status()}")

class TemperatureSensor(BaseSensor):
    """
    A specialized temperature sensor, inheriting from BaseSensor.
    Adds a temperature unit and overrides get_info.
    """
    def __init__(self, sensor_id: str, location: str, unit: str = "Celsius",
                 manufacturer: str = "Generic"):
        super().__init__(sensor_id, location, manufacturer)
        self.unit = unit
        print(f"TemperatureSensor '{self.sensor_id}' initialized with unit "
              f"{self.unit}.")

    def read_temperature(self) -> float:
        """Simulates reading a temperature."""
        import random
        temp = random.uniform(20.0, 30.0)
        print(f"Sensor {self.sensor_id} reads {temp:.2f} {self.unit}.")
        return temp

    def get_info(self) -> str:
        """Returns detailed information about the temperature sensor, including
        unit."""
        base_info = super().get_info()
        return f"{base_info}, Unit: {self.unit}"

class PressureSensor(BaseSensor):
    """
    A specialized pressure sensor, inheriting from BaseSensor.
    Adds a pressure range and overrides get_info.
    """
    def __init__(self, sensor_id: str, location: str, pressure_range_kpa:
                 Tuple[float, float], manufacturer: str = "Generic"):
        super().__init__(sensor_id, location, manufacturer)
        self.pressure_range_kpa = pressure_range_kpa
        print(f"PressureSensor '{self.sensor_id}' initialized with range "
              f"{self.pressure_range_kpa}.")

```

```

    def read_pressure(self) -> float:
        """Simulates reading pressure within its defined range."""
        import random
        pressure = random.uniform(self.pressure_range_kpa[0],
self.pressure_range_kpa[1])
        print(f"Sensor {self.sensor_id} reads {pressure:.2f} kPa.")
        return pressure

    def get_info(self) -> str:
        """Returns detailed information about the pressure sensor, including
its range."""
        base_info = super().get_info()
        return f"{base_info}, Range:
{self.pressure_range_kpa[0]}-{self.pressure_range_kpa[1]} kPa"

print("--- Demonstrating Inheritance and Method Overriding ---")

# Create instances of subclasses
temp_sensor_lab = TemperatureSensor("T_ENG_001", "Engine Bay",
unit="Fahrenheit", manufacturer="ThermoCorp")
pressure_sensor_lab = PressureSensor("P_HYD_001", "Hydraulic System",
pressure_range_kpa=(80.0, 120.0))

print("\n--- Sensor Status and Info ---")
print(temp_sensor_lab.get_info())
print(pressure_sensor_lab.get_info())

# Call methods inherited from the base class
temp_sensor_lab.deactivate()
print(temp_sensor_lab.get_info())

# Call specific methods
temp_sensor_lab.read_temperature()
pressure_sensor_lab.read_pressure()

# Demonstrate 'is-a' relationship
print(f"\nIs temp_sensor_lab a BaseSensor? {isinstance(temp_sensor_lab,
BaseSensor)}")
print(f"Is temp_sensor_lab a TemperatureSensor? {isinstance(temp_sensor_lab,
TemperatureSensor)}")
print(f"Is pressure_sensor_lab a TemperatureSensor?
{isinstance(pressure_sensor_lab, TemperatureSensor)}")

# Smaller Example: Inheritance with method overriding
class Instrument:
    def measure(self):

```

```

        print("Measuring generic data.")

class Voltmeter(Instrument):
    def measure(self):
        print("Measuring voltage in Volts.")

class Ammeter(Instrument):
    def measure(self):
        print("Measuring current in Amperes.")

gen_instrument = Instrument()
voltmeter = Voltmeter()
ammeter = Ammeter()

print("\n--- Smaller Inheritance Example ---")
gen_instrument.measure()
voltmeter.measure()
ammeter.measure()

```

Summary of Inheritance and Method Overriding

Concept	Description	Benefit
Inheritance	A class (subclass) derives attributes and methods from another class (superclass).	Code reusability, forms "is-a" hierarchy, reduces duplication.
Parent/Base Class	The class being inherited from.	Defines common interface and default behavior.
Child/Subclass	The class that inherits.	Specializes/extends the parent, adds unique features.
<code>super().__init__()</code>	Calls the constructor of the parent class from the child's <code>__init__</code> .	Ensures parent attributes are properly initialized.

Method Overriding	A subclass provides its own implementation of a method from its superclass.	Allows specialized behavior for specific types of objects.
--------------------------	---	--

4. Polymorphism

Polymorphism, meaning "many forms," is a core OOP concept that allows objects of different classes to be treated as objects of a common type. It enables a single interface to represent different underlying forms, leading to more flexible and extensible code.

4.1 Understanding Polymorphism

Conceptual Approach:

Polymorphism in Python often relies on method overriding (from inheritance) and a concept called "Duck Typing." The key idea is that if different objects share a common method name or set of methods, they can be processed by the same piece of code, even if their internal implementations differ.

Key aspects of Polymorphism:

- **Same Interface, Different Implementations:** Different objects (instances of different classes) can respond to the same method call in their own specific way.
- **Flexibility:** Code becomes more generic, able to work with various object types without explicit type checks.
- **Extensibility:** New classes can be added later that conform to an existing interface, and existing code will work with them automatically.

4.2 Polymorphism through Inheritance (Method Overriding Revisited)

Conceptual Approach:

When a child class overrides a method from its parent, objects of both the parent and child classes can be processed polymorphically. The calling code doesn't need to know the exact type of object; it just needs to know that the object has the expected method.

Code Implementation:

Python

```
# Reusing BaseSensor, TemperatureSensor, PressureSensor from Section 3.1

def describe_sensor(sensor_obj: BaseSensor):
    """
    A polymorphic function that can describe any sensor object that has a
    'get_info' method.
    It doesn't need to know the exact type of sensor.
    """
    print(f"Description: {sensor_obj.get_info()}")

print("\n--- Polymorphism with Inheritance (describe_sensor) ---")
# Create new instances for this example
temp_s = TemperatureSensor("PT100-A", "Furnace", unit="Kelvin")
pres_s = PressureSensor("BP-500", "Vacuum Chamber", pressure_range_kpa=(0.1,
50.0))
base_s = BaseSensor("GEN-001", "Control Room")

describe_sensor(temp_s)
describe_sensor(pres_s)
describe_sensor(base_s)

# Smaller Example: Polymorphic action with inherited methods
class Vehicle:
    def move(self):
        pass # Placeholder for generic move

class Car(Vehicle):
    def move(self):
        print("Car drives on road.")

class Boat(Vehicle):
    def move(self):
        print("Boat sails on water.")

def make_it_move(vehicle):
    vehicle.move()

print("\n--- Smaller Polymorphism Example (Vehicle) ---")
my_car = Car()
my_boat = Boat()
make_it_move(my_car)
make_it_move(my_boat)
```

4.3 Polymorphism through Duck Typing

Conceptual Approach:

Python is a dynamically typed language and heavily relies on **Duck Typing**. The principle is: "If it walks like a duck and quacks like a duck, then it's a duck." This means that Python doesn't care about the *actual type* of an object, but rather about what *methods it has* (its interface). If an object has all the methods required by a piece of code, that code will work with it, regardless of its class hierarchy.

This is a very powerful and common form of polymorphism in Python.

Code Implementation:

```
Python
# No inheritance required, just a common method
class DataLogger:
    def write_data(self, data):
        print(f"DataLogger: Writing {data} to disk.")

class NetworkTransmitter:
    def write_data(self, data):
        print(f"NetworkTransmitter: Sending {data} over network.")

class DisplayModule:
    def write_data(self, data):
        print(f"DisplayModule: Displaying {data} on screen.")

def process_and_output(device_list, sensor_value):
    """
    Processes a sensor value and outputs it using any device
    that has a 'write_data' method.
    This function is polymorphic via duck typing.
    """
    print(f"\nProcessing sensor value: {sensor_value}")
    for device in device_list:
        device.write_data(f"Sensor Value: {sensor_value}")

print("--- Polymorphism with Duck Typing ---")
logger = DataLogger()
transmitter = NetworkTransmitter()
display = DisplayModule()

output_devices = [logger, transmitter, display]
process_and_output(output_devices, 28.75)
```

```

process_and_output(output_devices, 101.3)

# Another "duck" that can "write_data" - it will work!
class CustomHandler:
    def write_data(self, msg):
        print(f"CustomHandler: !!! {msg} !!!")

custom_handler = CustomHandler()
process_and_output([custom_handler], "CRITICAL_ALERT")

# Smaller Example: Duck Typing for a 'start' method
class Motor:
    def start(self):
        print("Motor starting up.")

class Pump:
    def start(self):
        print("Pump initiating flow.")

def turn_on(device):
    device.start()

print("\n--- Smaller Duck Typing Example ---")
my_motor = Motor()
my_pump = Pump()
turn_on(my_motor)
turn_on(my_pump)

```

Summary of Polymorphism Concepts

Concept	Description	How it applies in Python
Polymorphism (General)	"Many forms" - single interface for different types of objects.	Achieved through inheritance (method overriding) and duck typing.
Method Overriding	Subclass re-implements a method from its superclass.	Enables objects in an inheritance hierarchy to respond differently to the same method call.

Duck Typing	An object's suitability is determined by the presence of certain methods/attributes, not its explicit type.	If an object has the required methods, Python will use it, regardless of its class. Very Pythonic.
-------------	---	--




5. Key Takeaways

This session introduced you to the foundational concepts of Object-Oriented Programming in Python, which are essential for building organized, scalable, and reusable code for complex engineering and scientific systems.

- **Classes and Objects:** Understood classes as blueprints and objects as instances, forming the core of OOP.
- **`__init__` and Instance Attributes:** Learned how the constructor (`__init__`) is used to initialize unique instance-specific data (attributes).
- **Instance vs. Class Variables:** Differentiated between instance variables (unique per object) and class variables (shared across all objects of a class), understanding their appropriate use cases and interaction.
- **Inheritance:** Mastered the concept of inheritance, allowing child classes to inherit properties and behaviors from parent classes, promoting code reuse and establishing "is-a" relationships.
- **Method Overriding:** Learned how subclasses can customize the behavior of inherited methods by providing their own implementation, enabling specialized behavior.
- **Polymorphism:** Gained a deep understanding of polymorphism, both through method overriding in inheritance and the crucial Pythonic concept of Duck Typing, enabling flexible and extensible code that works with diverse object types.

By grasping these OOP essentials, you are now equipped to design and implement more structured and maintainable Python applications for modeling real-world entities and systems.

ISRO URSC – Python Training | Analog Data | Rajath Kumar

 For queries: rajath@analogdata.ai |  [\(+91\) 96633 53992](tel:+919663353992) |  <https://analogdata.ai>

This material is part of the ISRO URSC Python Training Program conducted by Analog Data (June 2025). For educational use only. © 2025 AnalogData.
