# 02 OOP Intro

# Python Object-Oriented Programming (OOP)

## Introduction to Object-Oriented Programming

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of "objects," which are instances of classes that encapsulate data (attributes) and behavior (methods). Python fully supports OOP, enabling developers to write modular, reusable, and maintainable code.

### Core Principles of OOP

1. **Encapsulation**: Bundling data and methods that operate on that data within a single unit (class) and restricting direct access to internal components.
2. **Abstraction**: Hiding complex implementation details and exposing only essential features.
3. **Inheritance**: Creating new classes based on existing ones to promote code reuse.
4. **Polymorphism**: Allowing objects of different types to be treated as instances of a common superclass.

### Why Use OOP?

- **Modularity**: Code is organized into logical, self-contained units.
- **Reusability**: Classes can be reused across different parts of an application or in other projects.
- **Maintainability**: Changes to one part of the system have minimal impact on others.
- **Scalability**: Easier to manage and extend large codebases.

---

## Class Definition

A class is a blueprint or template for creating objects. It defines the structure (attributes) and behavior (methods) that its instances will possess.

### Syntax

```python
class ClassName:
    # Class body
    pass
```

### Example

```python
class Dog:
    pass
```

> A class definition does not create an object; it only defines the structure for future objects.

## Objects and Instantiation

An object is an instance of a class. Instantiation is the process of creating an object from a class.

### Creating an Object

PYTHON
```python
my_dog = Dog()
print(type(my_dog))  # <class '__main__.Dog'>
```

Each object has:

- **Identity**: Unique in memory (use `id()` to inspect).
- **State**: Represented by attributes (data).
- **Behavior**: Defined by methods (functions).

---

## The `__init__` Method (Constructor)

The `__init__` method is a special method in Python that initializes a newly created object. It is automatically called during instantiation.

### Example

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

dog1 = Dog("Buddy", "Golden Retriever")
dog2 = Dog("Max", "Labrador")

print(dog1.name)    # Buddy
print(dog2.breed)  # Labrador
```

> The `self` parameter refers to the current instance of the class and must be the first parameter in any instance method.

---

## Instance Variables vs. Class Variables

### Instance Variables

- Defined within methods (typically `__init__`).

- Unique to each instance.
- Accessed via `self.attribute`.

### Class Variables

- Defined at the class level (outside any method).
- Shared among all instances of the class.
- Accessed via `ClassName.attribute` or `self.attribute` (with caution).

### Example

```python
class Dog:
    species = "Canis lupus familiaris"  # Class variable

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

dog1 = Dog("Buddy", "Golden Retriever")
dog2 = Dog("Max", "Labrador")

print(dog1.species)  # Canis lupus familiaris
print(dog2.species)  # Canis lupus familiaris

# Modifying class variable affects all instances
Dog.species = "Domestic Dog"
print(dog1.species)  # Domestic Dog

# Assigning to self creates an instance variable (shadows class variable)
dog1.species = "My Custom Species"
print(dog1.species)  # My Custom Species (instance variable)
print(dog2.species)  # Domestic Dog (class variable)
```

> **Warning**: Assigning to `self.class_var` creates an instance variable that shadows the class variable for that specific object.

---

## Methods in Python Classes

Methods are functions defined inside a class that define the behavior of its instances.

### Types of Methods

1. **Instance Methods**
2. **Class Methods**
3. **Static Methods**

## Instance Methods

- Operate on an instance of the class.
- First parameter is `self`.

PYTHON

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

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

dog = Dog("Buddy")
print(dog.bark())  # Buddy says Woof!
```

## Class Methods

- Decorated with `@classmethod`.
- First parameter is `cls` (refers to the class).
- Used for alternative constructors or modifying class state.

PYTHON

```python
class Dog:
    species = "Canis lupus familiaris"

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

    @classmethod
    def from_string(cls, dog_str):
        name, breed = dog_str.split('-')
        return cls(name, breed)

    @classmethod
    def set_species(cls, new_species):
        cls.species = new_species

# Usage
dog = Dog.from_string("Max-Labrador")
Dog.set_species("Domestic Dog")
```

## Static Methods

- Decorated with `@staticmethod`.
- No `self` or `cls` parameter.

- Behave like regular functions but belong to the class namespace.

```python
class MathUtils:
    @staticmethod
    def add(x, y):
        return x + y

    @staticmethod
    def is_even(n):
        return n % 2 == 0

print(MathUtils.add(3, 5))       # 8
print(MathUtils.is_even(4))      # True
```

> **Use static methods** when the function is related to the class but does not need access to instance or class data.

---

## Encapsulation and Access Control

Encapsulation is the practice of restricting direct access to an object's internal state and requiring all interaction to occur through an object's methods.

### Access Control in Python

Python does not enforce strict access control but uses naming conventions:

- **Public**: `attribute` — accessible from anywhere.
- **Protected**: `_attribute` — intended for internal use (convention only).
- **Private**: `__attribute` — name-mangled to prevent accidental access.

### Name Mangling

Attributes prefixed with double underscores (`__`) are name-mangled to `_ClassName__attribute`, making them harder (but not impossible) to access externally.

PYTHON
```python
class BankAccount:
    def __init__(self, owner, balance=0):
        self.owner = owner
        self._balance = balance      # Protected (by convention)
        self.__pin = "1234"          # Private (name-mangled)

account = BankAccount("Alice", 1000)
print(account.owner)               # Alice
print(account._balance)            # 1000 (discouraged but allowed)
# print(account.__pin)             # AttributeError
print(account._BankAccount__pin)   # 1234 (possible but violates encapsulation)
```

> **Best Practice**: Avoid accessing private attributes directly. Use public methods or properties instead.

---

## Properties: Pythonic Encapsulation

The `@property` decorator provides a clean way to implement getters, setters, and deleters without breaking the public API.

## Basic Property

PYTHON

```python
class BankAccount:
    def __init__(self, owner, balance=0):
        self.owner = owner
        self._balance = balance

    @property
    def balance(self):
        return self._balance

    @balance.setter
    def balance(self, value):
        if value < 0:
            raise ValueError("Balance cannot be negative.")
        self._balance = value

    @balance.deleter
    def balance(self):
        del self._balance

# Usage
account = BankAccount("Bob", 500)
print(account.balance)    # 500
account.balance = 700     # Uses setter
# account.balance = -100 # Raises ValueError
```

**Advantages of `@property`:**

- Maintains a clean interface (`obj.attr` instead of `obj.get_attr()`).
- Allows validation and logic in setters.
- Backward-compatible if you later need to add logic.

## Comprehensive Example: Library Book System

PYTHON

```python
class LibraryBook:
    total_books = 0  # Class variable

    def __init__(self, title, author, isbn):
        self.title = title
        self.author = author
        self.__isbn = isbn
        self.__is_checked_out = False
        LibraryBook.total_books += 1

    @property
    def isbn(self):
        return self.__isbn

    @property
    def is_checked_out(self):
        return self.__is_checked_out

    def check_out(self):
        if not self.__is_checked_out:
            self.__is_checked_out = True
            return f"'{self.title}' checked out successfully."
        return f"'{self.title}' is already checked out."

    def return_book(self):
        if self.__is_checked_out:
            self.__is_checked_out = False
            return f"'{self.title}' returned successfully."
        return f"'{self.title}' was not checked out."

    @classmethod
    def get_total_books(cls):
        return cls.total_books

    @staticmethod
    def validate_isbn(isbn):
        return len(str(isbn)) == 13 and str(isbn).isdigit()

    def __str__(self):
        status = "Checked Out" if self.__is_checked_out else "Available"
        return f"'{self.title}' by {self.author} ({status})"

# Usage
```

```python
book1 = LibraryBook("1984", "George Orwell", 9780451524935)
print(book1.check_out())
print(LibraryBook.get_total_books())
```

This example demonstrates:

- Class and instance variables
- Private attributes with name mangling
- Properties for controlled access
- Instance, class, and static methods
- String representation via `__str__`

---

## Best Practices for Python OOP

1. **Use `@property` for attribute access control** instead of explicit getter/setter methods.
2. **Prefer composition over inheritance** when modeling "has-a" relationships.
3. **Use class methods for alternative constructors** (e.g., `from_json`, `from_dict`).
4. **Reserve static methods for utility functions** that are logically related to the class but do not require instance or class data.
5. **Follow naming conventions**:
   - Public: `name`
   - Protected: `_name`
   - Private: `__name`
6. **Avoid exposing internal implementation details**; design clear public interfaces.
7. **Validate input in setters and constructors** to maintain object integrity.

---

## Summary of Key Concepts

| Concept | Description |
|---|---|
| **Class** | Blueprint for creating objects. |
| **Object** | Instance of a class. |
| **Instantiation** | Process of creating an object using `ClassName()`. |
| **Instance Variable** | Data unique to each object (`self.attribute`). |
| **Class Variable** | Data shared by all instances (`ClassName.attribute`). |
| **Instance Method** | Method that operates on an instance (`self` as first parameter). |
| **Class Method** | Method that operates on the class (`@classmethod`, `cls` as first parameter). |
| **Static Method** | Utility method unrelated to instance/class state (`@staticmethod`). |
| **Encapsulation** | Bundling data and methods; controlling access via properties/conventions. |
| **Access Control** | Public (`attr`), Protected (`_attr`), Private (`__attr`) — convention-based. |

## Exercises

### Exercise 1: Implement a `Car` Class

Create a `Car` class with:

- Attributes: `make`, `model`, `year`, and private `__mileage`
- Class variable: `total_cars` (incremented on each instantiation)
- Methods:
  - `drive(miles)`: Increases mileage if miles > 0
  - Property for `mileage` with validation (non-negative)
  - Class method `get_total_cars()`

**Solution**:

```python
class Car:
    total_cars = 0

    def __init__(self, make, model, year, mileage=0):
        self.make = make
        self.model = model
        self.year = year
        self.__mileage = mileage
        Car.total_cars += 1

    @property
    def mileage(self):
        return self.__mileage

    @mileage.setter
    def mileage(self, value):
        if value < 0:
            raise ValueError("Mileage cannot be negative.")
        self.__mileage = value

    def drive(self, miles):
        if miles > 0:
            self.mileage += miles
            print(f"Drove {miles} miles. Total: {self.mileage}")
        else:
            print("Invalid miles.")

    @classmethod
    def get_total_cars(cls):
        return cls.total_cars
```

## Exercise 2: Extend with Inheritance

Create an `ElectricCar` subclass of `Car` that:

- Adds a `battery_capacity` attribute
- Overrides `drive()` to include battery consumption logic
- Adds a `charge()` method

This will be covered in detail in the next section on **Inheritance**.

---