upGrad

Object-Oriented Programming in Python

Object-Oriented Programming (OOP) is a programming paradigm that organises code using **objects** — entities that bundle **data (attributes)** and **behaviour (methods)**.
It enables building modular, reusable, and maintainable code that mirrors real-world systems.

Why it matters:

- Encourages logical structuring and grouping of related functionality
- Widely used in data pipelines, simulation models, and machine learning APIs

Defining a Class and Creating Objects

A **class** is a blueprint for objects; an **object** is an instance of a class that contains actual data.

```python
class Person:
    def greet(self):
        print("Hello!")

# Creating an object
p = Person()
p.greet()
```

Each object created from a class has access to the methods defined inside the class.

Constructor: `__init__` Method

The `__init__` method is Python's constructor; it runs automatically when a new object is created.

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

It allows you to initialise attributes with custom values and ensures each object starts with a valid state.

Instance Methods and Variables

- **Instance variables** are defined within the `__init__` method and are unique to each object.
- **Instance methods** operate on those variables using the `self` keyword.

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

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

You can create multiple objects, each with its own values and method access.

Class Methods and Class Variables

- **Class variables** are shared across all instances of the class.
- **Class methods** (using `@classmethod`) can access and modify these shared variables via `cls`.

```python
class Car:
    wheels = 4

    def __init__(self, model):
        self.model = model

    @classmethod
    def describe(cls):
        return f"All cars have {cls.wheels} wheels."
```

Useful when maintaining state or behaviour that belongs to the class, not just individual objects.

Static Methods

- A `@staticmethod` doesn't access instance (`self`) or class (`cls`) data.
- It's a regular function logically grouped inside a class.

```python
class Math:
    @staticmethod
    def square(x):
        return x * x
```

Best used for helper functions, like validations, formatters, or computations.

Inheritance: Reuse and Extend

One class can **inherit** from another and reuse or extend its methods and attributes.

```python
class Animal:
    def speak(self):
        print("Some sound")

class Dog(Animal):
    def speak(self):
        print("Woof!")
```

Inheritance supports **code reuse** and helps organise related classes in a hierarchy (e.g., `Vehicle → Car`).

Method Overriding

Child classes can **override** methods of their parent classes by redefining them.

```python
class Cat(Animal):
    def speak(self):
        print("Meow")
```

Use `super()` to call the parent method if needed. Overriding is essential for **customising inherited behaviour**.

upGrad

Core OOP Principles (The 4 Pillars)

Abstraction

- Expose only essential features; hide the internal complexity from the user
- Used in libraries like sklearn,` where models offer a simple interface like `.fit()` or `.predict()`

Encapsulation

- Python does not have *true* encapsulation; rather it relies on the developer to follow convention
- Bundle data and methods inside a class; restrict access using naming conventions (like `__variable`)
- This protects internal states from unintended modification and encourages safe access patterns

```python
class Account:
    def __init__(self):
        self.__balance = 0  # private variable

    def deposit(self, amount):
        self.__balance += amount
```

Inheritance

- Create child classes that inherit attributes and methods from a parent class.
- Enables hierarchical classification and eliminates code repetition.
- Used heavily in model abstraction: `BaseModel` → `LinearModel`, `TreeModel`, etc.

Polymorphism

- Write code that works on objects of different classes through a shared interface.
- Supports flexibility and clean design in APIs and utilities.

```python
for animal in [Dog(), Cat()]:
    animal.speak()  # Same method call → different output
```

Try this out

1. Build a `Book` class with title, author, and a method to display details.
2. Create a `Shape` base class and extend it to `Circle` and `Square` with overridden `area()` methods.
3. Use a class method to keep track of the number of `Employee` objects created.
4. Write a class with a static method to validate email addresses.
5. Demonstrate encapsulation using a class that prevents direct access to its balance.

Ask an LLM

- What's the difference between instance and class variables?
- When should you use a static method over a class method?
- How is encapsulation implemented in Python given that it doesn't enforce access modifiers?
- What role does polymorphism play in code extensibility?
- How does Python handle multiple inheritance?

upGrad

Summary

Concept	Summary
`__init__`	The constructor is used for initialising the object state
`self`	Refers to the object instance
`@classmethod`	Works with class-level attributes via `cls`
`@staticmethod`	Independent utility method, logically grouped within the class
Inheritance	Reuse and extend functionality from base classes
Overriding	Redefine methods in derived classes
Abstraction	Hide implementation, expose interfaces
Encapsulation	Protect internal data, control access via methods
Polymorphism	Same interface, different class behaviours (dynamic method resolution)

upGrad

Lecture Notes:
Python Data Science Libraries

NumPy Arrays

Key Definitions & Concepts

`ndarray`: A multi-dimensional array object with homogeneous data type, the core of NumPy.

Axis: Indicates the dimension along which an operation is applied. For 2D arrays:

- `axis=0`: down columns
- `axis=1`: across rows

Vectorised Operations: Operations applied directly to entire arrays without the need for explicit loops. These are optimised for performance and are a key feature of NumPy.

Universal Functions (ufuncs): Element-wise operations that are vectorised across arrays — e.g., `np.exp()`,`np.sqrt()`

Broadcasting: Allows arrays of different shapes to be combined in arithmetic operations without explicit looping.

Copy vs View: Slices often return views (not copies), meaning modifying the slice may alter the original array.

Core Ideas & Theoretical Intuition

- NumPy leverages **vectorisation** and **memory-efficient storage** to outperform Python lists in numerical tasks.
- Think of **broadcasting** as an implicit replication mechanism. NumPy aligns shapes without needing loops or `.repeat()`.
- Reshaping and transposition are **non-destructive operations** that offer powerful ways to prepare arrays for computation.

- Images and tabular data can be treated as **2D or 3D tensors**, making NumPy the go-to tool for low-level data prep.

Mathematical Foundation

Let:

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}, \quad B = \begin{bmatrix} 10 & 20 \end{bmatrix}$$

- **Broadcasted Addition**:

$$A + B = \begin{bmatrix} 1+10 & 2+20 \\ 3+10 & 4+20 \end{bmatrix} = \begin{bmatrix} 11 & 22 \\ 13 & 24 \end{bmatrix}$$

- **Dot Product**:

$$\text{If } x = [1, 2], \quad y = [3, 4], \quad \text{then } x \cdot y = 1 * 3 + 2 * 4 = 11$$

Implementation

Array Creation

Python

```
np.array([1, 2, 3])                    # From list
np.zeros((2, 3))                       # 2x3 zeros
np.ones((3,))                          # 1D ones
np.arange(0, 10, 2)                    # [0, 2, 4, 6, 8]
np.linspace(0, 1, 5)                   # [0. , 0.25, ..., 1.]
np.eye(3)                              # 3x3 identity matrix
```

upGrad

Indexing and Slicing

```python
a = np.array([[10, 20, 30], [40, 50, 60]])

a[1, 2]         # 60

a[:, 1]         # [20, 50]

a[0:2, 0:2]     # subarray

a[-1, :]        # last row
```

Reshaping & Flattening

```python
a = np.arange(12)

a.reshape(3, 4)          # new shape

a.flatten()              # copy as 1D

a.ravel()                # view as 1D
```

Aggregations

```python
a = np.array([[1, 2], [3, 4]])

a.sum(), a.mean(), a.std()

a.sum(axis=0)              # column-wise

a.max(axis=1)              # row-wise
```

Array Manipulations

`np.append()`: Add elements to an array

Adds values to the end along the specified axis or flattens if `axis=None`

```python
a = np.array([[1, 2], [3, 4]])

np.append(a, [[5, 6]], axis=0)   # Append row → shape (3, 2)

np.append(a, [[7], [8]], axis=1) # Append column → shape (2, 3)
```

Returns a **new array**, and the original is not modified.

`np.delete()`: Remove elements along an axis

Removes rows/columns by index

```python
a = np.array([[1, 2, 3], [4, 5, 6]])

# Remove 2nd row → shape (1, 3)

np.delete(a, 1, axis=0)

# Remove 1st and 3rd columns → shape (2, 1)

np.delete(a, [0, 2], axis=1)
```

Like `append`, it returns a **copy**, not in-place modification.

`.T` or `np.transpose()`: Transpose array dimensions

```python
a = np.array([[1, 2], [3, 4]])

a.T                    # [[1, 3], [2, 4]]

np.transpose(a)        # Same as a.T for 2D
```

For 3D+, use the `axes` argument:

```python
a = np.ones((2, 3, 4))

np.transpose(a, (1, 0, 2))  # Rearranges axes
```

`np.concatenate()`: Join arrays along an axis

Concatenates arrays with compatible dimensions

```python
a = np.array([[1, 2], [3, 4]])

b = np.array([[5, 6]])


np.concatenate((a, b), axis=0)  # Vertical → shape (3, 2)

np.concatenate((a, a), axis=1)  # Horizontal → shape (2, 4)
```

Dimensions must **match on all axes except the one concatenated.**

Axis Swapping

Python

```
a = np.array([[1, 2], [3, 4]])

a.T                    # transpose: [[1, 3], [2, 4]]

a.swapaxes(0, 1)       # swap dimensions
```

Stacking & Splitting

Python

```
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])

np.stack((a, b))          # shape: (2, 3)
np.hstack((a, b))         # shape: (6,)
np.vstack((a, b))         # shape: (2, 3)

c = np.arange(9).reshape(3, 3)
np.hsplit(c, 3)           # three column arrays
```

Element-wise Operations (Broadcasting)

Python

```
a = np.array([1, 2, 3])

a + 10                    # [11, 12, 13]

a * 2                     # [2, 4, 6]

a ** 2                    # [1, 4, 9]

np.exp(a)                 # [2.71, 7.39, 20.09]
```

Logical Operations & Boolean Indexing

```python
a = np.array([1, 2, 3, 4, 5])
a[a > 3]                    # [4, 5]
np.where(a % 2 == 0, 0, 1) # Replace even with 0
```

Image Manipulation

```python
from PIL import Image
img = Image.open("image.jpg").convert("L")
arr = np.array(img)

# Invert grayscale
inverted = 255 - arr

# Brighten image
bright = np.clip(arr + 30, 0, 255)
```

Real-world Use Cases

- **Data Preprocessing**: Clean and transform numeric datasets before modelling
- **Scientific Computation**: Perform fast simulations, e.g., particle systems, numerical integration
- **Image Analysis**: Pixel-wise filters, transformations, masks
- **Feature Engineering**: Compute means, deltas, differences, or normalisations across axes

Pitfalls and Misconceptions

- **Shape mismatch in broadcasting**: Ensure trailing dimensions are compatible
- **Slicing returns views, not copies**: Use `.copy()` if needed
- **`reshape()` errors**: Total size must remain constant
- **Misuse of axis**: Confusing axis 0 vs 1 is a common source of bugs

NumPy Functions and Operations

NumPy Functions	Description
`np.array([1,2,3])`	Creates a 1D array of shape **1x3** with values *1, 2, 3*
`np.array([(1,2,3), (4,5,6)])`	Creates 2D array of shape **2x3** with values *1,2,3,4,5,6*
`np.array([[(1,2,3), (4,5,6)], [(7,8,9), (10,11,12)]])`	Creates a 3D array with shape **2x2x3**
`np.zeros(3,4)`	Creates a **3x4** array of zeros
`np.arrange(1,60,5)`	Creates a 1D array of values 1 through 60 at steps of 5
`arr.reshape(2,3,4)`	Reshapes array `arr` into an array of shape **2x3x4**
`arr.shape`	To get the shape of the array `arr`

NumPy Operations	Description
`a[0]`	Gets the 0th element in 1D array
`b[0,0]`	Gets the 0th element in 2D array
`c[0,0,0]`	Gets the 0th element in 3D array
`c[9,0,0]`	Gets the element which is the 0th element in the 0th row in the 9th depth
1D Arrays	
`a[:]`	Selects everything
`a[2:5]`	Selects the 2nd through the 4th rows (does not include the 5th row)
2D Arrays	

NumPy Operations	Description
`b[:,:]`	Selects all rows and all columns
`b[:,0]`	Selects all rows, and the zeroth column
`b[0,:]`	Selects the zeroth row, and all columns in that row
`b[0:2,:]`	Selects the zeroth and first row, but NOT the second row
`b[0:2,0:2]`	Selects the zeroth and first row, and the zeroth and first column
3D Arrays	
`c[:,:,:]`	Selects all rows and columns on all depths
`c[0,:,:]`	Selects the everything in the first depth
`c[:,0,:]`	Selects the first row of each depth
`c[:,:,0]`	Selects the first column of each depth

Try this out

1. Write NumPy code to create a 5×5 matrix with values 1 to 25.
2. Given `a = np.array([1, 2, 3])`, what is `a[:, np.newaxis] + a`?
3. How do you extract the diagonal of a 2D array?
4. How would you stack two 1D arrays column-wise?
5. What is the difference between `flatten()` and `ravel()`?

Additional Reading

- Official NumPy Docs
- Visual Guide to Broadcasting
- NumPy 100 Practice Problems

upGrad

Pandas Series

Key Definitions & Concepts

`Series`: A one-dimensional labelled array, capable of holding any data type (int, float, str, datetime, etc.). Think of it as a hybrid of a list and a dictionary.

Index: The labels associated with each element in a Series. Allows fast access and alignment.

Vectorised Operations: Just like NumPy arrays, Series supports element-wise operations (addition, comparison, logical operations).

Method Chaining: The practice of applying multiple Series/DataFrame methods in a single statement. Promotes readable, fluent code.

`.str` **accessor**: Enables vectorised string methods.

`.dt` **accessor**: Provides datetime-specific operations after converting a Series to a datetime format.

Core Ideas

- A Pandas Series behaves like a **dictionary-backed vector**: values are like a NumPy array; labels (index) are like dictionary keys.
- Operations respect the index: this makes Pandas ideal for labelled time series or named features.
- **Method chaining** encourages writing "pipelines", stepwise transformations applied in sequence.
- Series enables **clean, readable, high-level operations** without the verbosity of loops.

Implementation

Creating Series

```python
import pandas as pd

# From a list
pd.Series([10, 20, 30])

# With custom index
pd.Series([10, 20, 30], index=['a', 'b', 'c'])

# From a dictionary
pd.Series({'x': 1, 'y': 2, 'z': 3})
```

Indexing and Slicing

```python
s = pd.Series([10, 20, 30], index=['a', 'b', 'c'])


s['a']              # 10

s[0:2]              # First two elements

s[['a', 'c']]       # Specific labels

s.loc['b']          # Access by label

s.iloc[1]           # Access by position
```

Operations on Series

```python
s = pd.Series([1, 2, 3, 4])

s + 10                      # [11, 12, 13, 14]
s * 2                       # [2, 4, 6, 8]
s.mean(), s.max(), s.std()

# Method chaining
s.add(1).mul(10).clip(0, 25)

# Remove duplicates
pd.Series([1, 2, 2, 3]).drop_duplicates()

# Apply/map
s.apply(lambda x: x**2)
s.map({1: 'A', 2: 'B', 3: 'C'})  # Mapping values
```

Working with Series (Boolean Logic & Filtering)

```python
s = pd.Series([100, 200, 300, 400], index=['a', 'b', 'c', 'd'])


s[s > 200]                # Filter by value

s.isnull()                # Check for missing values

s.fillna(0)               # Fill missing values
```

String Operations with `.str`

Python

```
names = pd.Series(['alice', 'BOB', 'Charlie'])


names.str.upper()                  # ['ALICE', 'BOB', 'CHARLIE']

names.str.len()                    # [5, 3, 7]

names.str.contains('li')          # [True, False, True]

names.str.split('a')               # [['', 'lice'], ['BOB'], ['Ch',
'rlie']]
```

Datetime Operations with `.dt`

Python

```
dates = pd.Series(['2023-01-01', '2023-05-15'])

dates = pd.to_datetime(dates)


dates.dt.year          # [2023, 2023]

dates.dt.month         # [1, 5]

dates.dt.weekday       # [6, 0]
```

Summary

Operation	Method	Description
Create Series	`pd.Series(data)`	From list, dict, array
Indexing	`s['a']`, `s.iloc[0]`	By label or position
Math ops	`s + 10`, `s * 2`	Element-wise operations
Filtering	`s[s > 100]`	Boolean condition
Map/Apply	`s.map()`, `s.apply()`	Transform values
String ops	`s.str.upper()`, `s.str.len()`	Vectorised string handling
Datetime ops	`s.dt.year`, `s.dt.month`	Extract date components

Real-world Use Cases

- **Feature Engineering**: Extract year/month from timestamps
- **Data Cleaning**: Lowercase names, split strings, remove duplicates
- **Time Series Analysis**: Weekly sales, monthly trends
- **Categorical Encoding**: Mapping string values to codes

Pitfalls and Misconceptions

- `.apply()` vs `.map()`:
  `map()` is for value substitution; `apply()` is more general (can apply any function)
- `.str` accessor only works on string-typed Series; convert if needed
- `to_datetime()` is required before using `.dt` accessor

Try this out

1. Create a Series of 5 cities with population values.
2. Convert `['2022-01-01', '2023-01-01']` into a datetime Series and extract the year.
3. How would you replace all lowercase names in a Series with uppercase versions?
4. Filter a Series of integers to only keep even numbers.
5. Explain the difference between `.apply()` and `.map()` with a small example.