Got it ✅
I'll create a **well-structured, beginner-friendly note on OOP in Python** with **clear explanations** and **line-by-line code descriptions** so that anyone can understand it easily.

📘 Object-Oriented Programming (OOP) in Python – Complete Notes

1. Introduction

Object-Oriented Programming (OOP) is a programming paradigm that organizes code into **objects** — entities that combine **data** (attributes) and **behavior** (methods).
Python fully supports OOP, making it easier to write **modular, reusable, and maintainable** code.

2. Four Pillars of OOP

- **Encapsulation** – Binding data and methods together, restricting direct access to data.
- **Inheritance** – Reusing code by creating new classes from existing ones.
- **Polymorphism** – Same method name, different behavior.
- **Abstraction** – Hiding implementation details and showing only the necessary features.

3. OOP in Python – Step-by-Step Example

We'll create a **single program** that demonstrates **all OOP concepts**.

```
# Importing ABC for abstraction
from abc import ABC, abstractmethod

# ------------------------------
# 1. Abstraction
# ------------------------------
class Shape(ABC):  # Abstract class
    @abstractmethod
    def area(self):
        pass  # Abstract method (must be implemented in child classes)

# ------------------------------
# 2. Encapsulation
# ------------------------------
class Rectangle(Shape):
    def __init__(self, length, width):
        self.__length = length  # Private attribute
        self.__width = width    # Private attribute
```

```
    # Getter method for length
    def get_length(self):
        return self.__length

    # Setter method for length
    def set_length(self, length):
        if length > 0:
            self.__length = length
        else:
            print("Length must be positive.")

    # Method to calculate area (implementation of abstract method)
    def area(self):
        return self.__length * self.__width

# --------------------------------
# 3. Inheritance
# --------------------------------
class Square(Rectangle):  # Inherits from Rectangle
    def __init__(self, side):
        super().__init__(side, side)  # Call parent constructor

# --------------------------------
# 4. Polymorphism
# --------------------------------
shapes = [Rectangle(4, 5), Square(6)]

for shape in shapes:
    print(f"Area: {shape.area()}")  # Same method name, different behavior
```

4. Line-by-Line Explanation

Imports

```
from abc import ABC, abstractmethod
```

- `ABC`

  : Base class for defining abstract classes.

- `abstractmethod`

  : Decorator to define methods that must be implemented in subclasses.

Abstraction

```
class Shape(ABC):
    @abstractmethod
    def area(self):
        pass
```

- `Shape`

  is an **abstract class** — cannot be instantiated directly.

- `area()`

  is an **abstract method** — must be implemented in child classes.

Encapsulation

```
class Rectangle(Shape):
    def __init__(self, length, width):
        self.__length = length
        self.__width = width
```

- `__length`

  and

  `__width`

  are **private attributes** (cannot be accessed directly outside the class).

- This protects data from accidental modification.

```
def get_length(self):
    return self.__length
```

- **Getter method** to access private data.

```
def set_length(self, length):
    if length > 0:
        self.__length = length
    else:
        print("Length must be positive.")
```

- **Setter method** to update private data with validation.

```
def area(self):
    return self.__length * self.__width
```

- Implements the abstract method from `Shape`.
- Calculates the area of a rectangle.

Inheritance

```
class Square(Rectangle):
    def __init__(self, side):
        super().__init__(side, side)
```

- `Square` inherits from `Rectangle`.
- Uses `super()`

to call the parent constructor with equal length and width.

Polymorphism

```
shapes = [Rectangle(4, 5), Square(6)]
for shape in shapes:
    print(f"Area: {shape.area()}")
```

- Both

  `Rectangle`

  and

  `Square`

  have an

  `area()`

  method.
- The same method name works differently depending on the object type.

5. Output

```
Area: 20
Area: 36
```

6. Key Takeaways

- **Encapsulation** protects data using private attributes and getter/setter methods.
- **Inheritance** allows code reuse and hierarchy creation.
- **Polymorphism** enables the same method name to work differently for different objects.
- **Abstraction** hides unnecessary details and enforces method implementation in subclasses.

If you want, I can now prepare a **beautifully formatted PDF note** with diagrams explaining **OOP in Python** so it's even easier for beginners to understand visually.
Do you want me to make that?