```
# ■ OOPS in Python with Shape Class Example (Basic to Advanced)
```

■ 1. What is OOPS?

```
Object-Oriented Programming (OOP) is a programming paradigm based on the
concept of "objects" that contain both data (attributes) and methods
(functions). It helps in organizing complex code by grouping related
variables and functions into objects.
```

■ 4 Pillars of OOPS

```
1. Encapsulation: Wrapping the data and methods into a single unit
(class). 2. Inheritance: Deriving a new class from an existing class. 3.
Polymorphism: Using a single method name but having different
implementations. 4. Abstraction: Hiding unnecessary details and showing
only essential information. ---
```

■ 2. OOPS Concepts Using Shape Class Example

■■ Step 1: Basic Class and Object (Encapsulation)

```
class Shape: def __init__(self, color): self.color = color def
display_color(self): print(f"The color of the shape is {self.color}")
circle = Shape("Red") square = Shape("Blue") circle.display_color()
square.display_color()
```

```
---
```

■ Step 2: Inheritance (Reusing Parent Class)

```
class Shape: def __init__(self, color): self.color = color def
display_color(self): print(f"The color of the shape is {self.color}")
class Circle(Shape): def __init__(self, color, radius):
super().__init__(color) self.radius = radius def area(self): return 3.14 *
self.radius ** 2 circle = Circle("Green", 5) circle.display_color()
print(f"Circle Area: {circle.area()}")
```

```
---
```

■■ Step 3: Polymorphism (Same Method, Different Behavior)

```
class Shape: def area(self): print("Area calculation not defined") class
Circle(Shape): def __init__(self, radius): self.radius = radius def
area(self): return 3.14 * self.radius ** 2 circle = Circle(7)
print(f"Circle Area: {circle.area()}")
```

```
---
```

■ Step 4: Abstraction (Hiding Details)

```
from abc import ABC, abstractmethod class Shape(ABC): @abstractmethod def
area(self): pass class Circle(Shape): def __init__(self, radius):
self.radius = radius def area(self): return 3.14 * self.radius ** 2 circle
= Circle(4) print(f"Circle Area: {circle.area()}")
```

```
---
```

■ 3. Class Method vs Static Method

Class Method Example

```
class Shape: shape_count = 0 def __init__(self, name): self.name = name
Shape.shape_count += 1 @classmethod def display_count(cls): print(f"Total
shapes created: {cls.shape_count}") circle = Shape("Circle") square =
Shape("Square") Shape.display_count()
```

Static Method Example

```
class MathOperations: @staticmethod def add(x, y): return x + y
@staticmethod def multiply(x, y): return x * y
print(MathOperations.add(10, 5)) print(MathOperations.multiply(4, 3))
```

Combining Both Class and Static Methods

```
class Account: interest_rate = 0.05 def __init__(self, balance):
self.balance = balance @classmethod def set_interest_rate(cls, rate):
cls.interest_rate = rate @staticmethod def validate_amount(amount): return
amount > 0 Account.set_interest_rate(0.07) print(Account.interest_rate)
print(Account.validate_amount(100)) print(Account.validate_amount(-50))
```

```
---
```

■ Key Takeaway

```
- `@classmethod` → Bound to the class and can modify class-level data. -
`@staticmethod` → Independent of the class, acts like a regular function
but belongs to the class's namespace.
```