📘 Python OOP (Object-Oriented Programming) Syntax with Annotations

1. Introduction

Python supports Object-Oriented Programming (OOP) with classes, inheritance, encapsulation, and polymorphism. Alongside, Python provides annotations like type hints and decorators to make OOP code more robust, readable, and maintainable.

---

2. Class Definition

```python
class Dog:
    species: str = "Canis familiaris"   # class attribute with type annotation

    def __init__(self, name: str, age: int) -> None:  # constructor with annotations
        self.name: str = name
        self.age: int = age

    def bark(self) -> str:  # return type annotation
        return f"{self.name} says Woof!"
```

🔹 Annotations used here:

- `name: str`, `age: int` → type hints for instance attributes.
- `-> None` → constructor returns nothing.
- `-> str` → method returns a string.

---

3. Class vs Instance Attributes

```python
class Car:
    wheels: int = 4  # class attribute

    def __init__(self, brand: str, year: int) -> None:
        self.brand: str = brand  # instance attribute
        self.year: int = year
```

---

4. Instance, Class, and Static Methods

```
class MathUtils:
    pi: float = 3.14

    def square(self, num: int) -> int:   # instance method
        return num * num

    @classmethod
    def circle_area(cls, radius: float) -> float:   # class method
        return cls.pi * radius * radius

    @staticmethod
    def add(a: int, b: int) -> int:  # static method
        return a + b
```

◈ Annotations used here:

- `@classmethod` → method bound to the class, not an object.
- `@staticmethod` → method not bound to either class or object.

---

5. Encapsulation

```
class BankAccount:
    def __init__(self, balance: float) -> None:
        self._balance: float = balance     # protected attribute
        self.__pin: str = "1234"          # private attribute

    def get_balance(self) -> float:
        return self._balance
```

---

6. Inheritance & Polymorphism

```
class Animal:
    def speak(self) -> str:
        return "Some sound"

class Dog(Animal):
    def speak(self) -> str:   # overriding method
        return "Woof!"
```

---

7. Abstract Classes & Interfaces

```python
from abc import ABC, abstractmethod

class Shape(ABC):   # abstract base class
    @abstractmethod
    def area(self) -> float:
        pass

class Square(Shape):
    def __init__(self, side: float) -> None:
        self.side = side

    def area(self) -> float:
        return self.side * self.side
```

◈ Annotations used here:

- `ABC` → base class for defining abstract classes.
- `@abstractmethod` → enforces implementation in subclasses.

---

8. Dataclasses (Annotation for Simplicity)

```python
from dataclasses import dataclass

@dataclass
class Book:
    title: str
    author: str
    pages: int
```

✅ `@dataclass` auto-generates `__init__`, `__repr__`, `__eq__` methods.

---

9. Properties with @property Annotation

```python
class Temperature:
    def __init__(self, celsius: float) -> None:
        self._celsius = celsius

    @property
    def celsius(self) -> float:    # getter
        return self._celsius
    @celsius.setter
    def celsius(self, value: float) -> None:   # setter
        if value < -273.15:
            raise ValueError("Temperature below absolute zero!")
        self._celsius = value
```

◈ Annotations used here:

- `@property` → makes a method act like an attribute.
- `@celsius.setter` → defines setter for the property.

---

10. Magic (Dunder) Methods

```
class Vector:
    def __init__(self, x: float, y: float) -> None:
        self.x = x
        self.y = y

    def __str__(self) -> str:
        return f"Vector({self.x}, {self.y})"

    def __add__(self, other: "Vector") -> "Vector":  # operator overloading
        return Vector(self.x + other.x, self.y + other.y)
```

---

11. Commonly Used Annotations in OOP

Annotation	Use Case
`-> type`	Function/method return type hint
`var: type`	Attribute/parameter type hint
`@classmethod`	Define class-level methods
`@staticmethod`	Define static utility methods
`@property`	Define read-only attributes
`@property.setter`	Define setters for properties
`@abstractmethod`	Force subclasses to implement a method
`@dataclass`	Auto-generate class boilerplate
`@overload` (from typing)	Define multiple type signatures for functions

---

12. Best Practices

- Always annotate method parameters and return types.
- Use `@dataclass` for data containers.
- Prefer `@property` over getter/setter methods.
- Use abstract classes for interfaces.
- Use composition over multiple inheritance when possible.

📝 25 Exercises on Python OOP (with Annotations)

Beginner

1. Create a `Student` class with type annotations for name and marks.
2. Write a class `Circle` with radius attribute and annotated area method.
3. Add type hints to a `Car` class with brand and year attributes.
4. Create a `Book` class using `@dataclass`.
5. Define a `Point` class with `__add__` method and annotations.

Intermediate

6. Create a `BankAccount` with annotated deposit and withdraw methods.
7. Use `@classmethod` to create a `Person` object from a birth year.
8. Write a `MathUtils` class with a `@staticmethod` add method.
9. Create a `Rectangle` class with annotated `area()` and `perimeter()`.
10. Add `@property` and `@setter` to control access to a `Temperature` attribute.

Inheritance

11. Create an `Animal` class and subclasses `Dog`, `Cat` with annotated `speak()` methods.
12. Implement a `Shape` abstract class and subclasses `Circle`, `Square`.
13. Use `@abstractmethod` in an `Employee` class requiring `calculate_salary()`.
14. Demonstrate type hints in overridden methods.
15. Show how multiple inheritance affects annotations.

Advanced OOP

16. Implement a `Vector` class with annotated `__add__` and `__str__`.
17. Use `@dataclass` for a `Product` class with name, price, and quantity.
18. Implement a `Zoo` class that stores a list of `Animal` objects with type hints.
19. Create a `Playlist` class with `__len__` and `__str__` methods using annotations.
20. Define a `Factory` class that returns different objects with proper type hints.

Challenge

21. Create a `Logger` singleton class using annotations.
22. Write a `Database` class with connection pooling (hint annotations).
23. Implement an interface using `ABC` for `PaymentGateway`.
24. Write a `ShoppingCart` class with annotated methods for adding/removing items.
25. Build a `Game` class with annotated methods for starting, pausing, and stopping.