Python oops concept:

1. Classes and Objects

Concept

A **class** is a blueprint for creating objects, and an **object** is an instance of a class.

Example

```
class Dog:
    species = "Canis familiaris"  # Class attribute

    def __init__(self, name, age):
        self.name = name  # Instance attribute
        self.age = age

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

Explanation

- `Dog` is a class with a shared attribute `species`.
- `__init__` initializes an object with `name` and `age`.
- `bark` is a method that returns a string.

2. Inheritance

Concept

Inheritance allows a class to inherit attributes and methods from another class.

Example

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

class Cat(Pet):  # Cat inherits from Pet
    def speak(self):
        return "Meow!"
```

Explanation

- `Cat` inherits from `Pet`, so it has `name` and `age` attributes without redefining them.
- `speak` is specific to `Cat`.

3. Encapsulation

Concept

Encapsulation restricts direct access to object data and requires methods for interaction.

Example

```python
class BankAccount:
    def __init__(self, owner, balance=0):
        self.owner = owner
        self.__balance = balance  # Private attribute

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

    def get_balance(self):
        return self.__balance
```

Explanation

- `__balance` is private and can't be accessed directly.
- `get_balance` is a public method to retrieve the balance.

4. Polymorphism

Concept

Polymorphism allows methods in different classes to have the same name but different behaviors.

Example

```python
class Animal:
    def speak(self):
        raise NotImplementedError("Subclass must implement this

class Dog(Animal):
    def speak(self):
        return "Bark"

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

Explanation

- Both `Dog` and `Cat` have a `speak` method but behave differently.
- `Animal` enforces implementation of `speak` in subclasses.

5. Abstraction

Concept

Abstraction hides implementation details while exposing essential functionalities.

Example

```
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
```

Explanation

- `Shape` is an abstract class; `Circle` must implement `area`.
- This enforces a consistent interface.

6. Properties and Getters/Setters

Concept

Properties allow controlled attribute access.

Example

```
class Temperature:
    def __init__(self, celsius):
        self._celsius = celsius

    @property
    def fahrenheit(self):
        return self._celsius * 9/5 + 32
```

Explanation

- `fahrenheit` acts like an attribute but calculates a value dynamically.

7. Class and Static Methods

Concept

- **Class methods** operate on class variables.
- **Static methods** don't modify class or instance state.

Example

```
class MathOperations:
    pi = 3.14159

    @classmethod
    def circle_area(cls, radius):
        return cls.pi * radius ** 2

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

Explanation

- `circle_area` uses `pi`, a class variable.
- `is_even` doesn't use class attributes or methods.

8. Magic Methods (Dunder Methods)

Concept

Magic methods customize object behavior, e.g., `__str__`, `__add__`.

Example

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)
```

Explanation

- `__add__` allows `+` operator to work on `Vector` instances.

9. Metaclasses

Concept

A **metaclass** defines how classes are created.

Example

```
class SingletonMeta(type):
    _instances = {}
    def __call__(cls, *args, **kwargs):
        if cls not in cls._instances:
            cls._instances[cls] = super().__call__(*args, **kwa
        return cls._instances[cls]
```

Explanation

- Ensures only one instance of a class is created.

10. Design Patterns

Example: Singleton

```
class Singleton:
    _instance = None
    def __new__(cls):
        if cls._instance is None:
            cls._instance = super().__new__(cls)
        return cls._instance
```

Explanation

- Ensures a class has only one instance.