Python OOPS Concepts - Notes with Examples

Inheritance

Inheritance allows a class (child) to acquire properties and methods from another class (parent). Types of inheritance in Python: 1. Single Inheritance - One child class inherits from one parent class. 2. Multiple Inheritance - A class inherits from more than one parent class. 3. Multilevel Inheritance - A class is derived from another derived class. 4. Hierarchical Inheritance - Multiple child classes inherit from one parent class. 5. Hybrid Inheritance - Combination of multiple inheritance types.

```
# Single Inheritance
class Parent:
    def greet(self):
        print("Hello from Parent")

class Child(Parent):
    def greet_child(self):
        print("Hello from Child")

c = Child()
c.greet()

# Multiple Inheritance
class Mother:
    def speak(self):
        print("Speaking like Mother")

class Father:
    def speak(self):
        print("Speaking like Father")

class Child(Mother, Father):
    pass

c = Child()
c.speak()  # Method Resolution Order applies

# Multilevel Inheritance
class Grandparent:
    def role(self):
        print("I am Grandparent")

class Parent(Grandparent):
    pass

class Child(Parent):
    pass

c = Child()
c.role()

# Hierarchical Inheritance
class Parent:
```

```
    def greet(self):
        print("Hello from Parent")

class Child1(Parent):
    pass

class Child2(Parent):
    pass

c1 = Child1()
c2 = Child2()
c1.greet()
c2.greet()
```

Polymorphism

Polymorphism means 'many forms'. It allows methods to perform different tasks depending on the object. Types of Polymorphism: 1. Compile-time polymorphism (not directly supported in Python, achieved using default arguments or function overloading). 2. Run-time polymorphism (achieved using method overriding).

```
# Run-time Polymorphism using method overriding
class Animal:
    def sound(self):
        print("Some generic sound")

class Dog(Animal):
    def sound(self):
        print("Bark")

class Cat(Animal):
    def sound(self):
        print("Meow")

for animal in [Dog(), Cat()]:
    animal.sound()

# Function Overloading (achieved with default arguments)
def add(a, b=0, c=0):
    return a + b + c

print(add(2))
print(add(2, 3))
print(add(2, 3, 4))
```

Encapsulation and Data Hiding

Encapsulation is the bundling of data (variables) and methods into a single unit (class). Data hiding restricts direct access to variables and is achieved using private members (prefix __).

```
class BankAccount:
    def __init__(self, balance):
        self.__balance = balance  # private variable
```

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

    def get_balance(self):
        return self.__balance

account = BankAccount(1000)
account.deposit(500)
print(account.get_balance())
# print(account.__balance)  # This will cause error
```

Abstraction

Abstraction means hiding the implementation details and showing only the essential features. In Python, abstraction is achieved using abstract base classes (ABC) and the @abstractmethod decorator.

```
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 * self.radius

c = Circle(5)
print("Circle area:", c.area())
```

Getter and Setter

Getters and Setters are methods to access and update private attributes in a controlled way.

```
class Student:
    def __init__(self, name):
        self.__name = name

    def get_name(self):  # Getter
        return self.__name

    def set_name(self, new_name):  # Setter
        if new_name != "":
            self.__name = new_name

s = Student("John")
print(s.get_name())
s.set_name("Mike")
print(s.get_name())
```