Five Key Concepts of Object-Oriented Programming (OOP)

Encapsulation:
Definition: Encapsulation is the bundling of data and methods that operate on the data into a single unit or class. It restricts direct access to some of an object’s components.
Example: Using private attributes and public methods to control access.

Abstraction:
Definition: Abstraction is the concept of hiding the complex implementation details and showing only the necessary features of an object.
Example: Abstract base classes and abstract methods.

Inheritance:
Definition: Inheritance allows a class to inherit attributes and methods from another class, promoting code reuse and establishing a hierarchical relationship.
Example: A Dog class inheriting from an Animal class.

Polymorphism:
Definition: Polymorphism allows objects of different classes to be treated as objects of a common superclass, typically by using method overriding or method overloading.
Example: Different classes implementing the same method name with different behaviors.

Composition:
Definition: Composition is a design principle where a class is composed of one or more objects from other classes to achieve a more complex functionality.
Example: A Car class containing Engine and Wheel objects.

2. Python Class for a Car

python

```
class Car:
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year

    def display_info(self):
        print(f"{self.year} {self.make} {self.model}")

# Example usage
my_car = Car("Toyota", "Corolla", 2020)
my_car.display_info()
```

3. Instance Methods vs. Class Methods

Instance Methods:

Definition: Methods that operate on an instance of the class (i.e., they require an instance of the class to be called).

Example:

python

```
class MyClass:
    def instance_method(self):
        return "Instance method called"

obj = MyClass()
print(obj.instance_method())
```

Class Methods:

Definition: Methods that operate on the class itself rather than an instance. They are defined with the @classmethod decorator and take cls as the first parameter.

Example:

python

```
class MyClass:
    @classmethod
    def class_method(cls):
        return "Class method called"

print(MyClass.class_method())
```

4. Method Overloading in Python

Python does not support method overloading in the traditional sense. Instead, you can use default arguments or variable-length argument lists to achieve similar functionality.

Example with Default Arguments:

python

```
class Calculator:
    def add(self, a, b, c=0):
        return a + b + c

calc = Calculator()
print(calc.add(2, 3))      # Output: 5
print(calc.add(2, 3, 4))   # Output: 9
```

5. Access Modifiers in Python

Public: Accessible from anywhere. Denoted by no leading underscores.
Protected: Intended to be accessed only within the class and its subclasses. Denoted by a single leading underscore (_).
Private: Not accessible from outside the class. Denoted by a double leading underscore (__).

6. Types of Inheritance in Python

Single Inheritance: A class inherits from one base class.

Example:

python

```
class Animal:
    pass

class Dog(Animal):
    pass
```

Multiple Inheritance: A class inherits from more than one base class.

Example:

python

```
class Engine:
    pass

class Wheels:
    pass

class Car(Engine, Wheels):
    pass
```

Multilevel Inheritance: A class inherits from a base class, and another class inherits from that derived class.

Example:

python

```
class Animal:
    pass

class Mammal(Animal):
    pass

class Dog(Mammal):
    pass
```

Hierarchical Inheritance: Multiple classes inherit from a single base class.

Example:

python

```
class Animal:
    pass

class Dog(Animal):
    pass

class Cat(Animal):
    pass
```

Hybrid Inheritance: A combination of two or more types of inheritance.

Example:

python

```
class Animal:
    pass

class Mammal(Animal):
    pass

class Bird(Animal):
    pass

class Bat(Mammal, Bird):
    pass
```

7. Method Resolution Order (MRO) in Python

Definition: The order in which Python looks up methods in a hierarchy of classes. It is crucial in multiple inheritance scenarios.

Retrieve Programmatically:

python

```
class A:
    pass

class B(A):
    pass

class C(A):
    pass

class D(B, C):
    pass

print(D.mro())
```

8. Abstract Base Class Example

python

```
from abc import ABC, abstractmethod
import math

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

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return math.pi * self.radius * self.radius

class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

# Example usage
circle = Circle(5)
rectangle = Rectangle(4, 6)
print(f"Circle Area: {circle.area()}")
print(f"Rectangle Area: {rectangle.area()}")
```

9. Polymorphism Example

python

```
def print_area(shape):
    print(f"Area: {shape.area()}")

print_area(Circle(5))
print_area(Rectangle(4, 6))
```

10. Encapsulation in a BankAccount Class

python

```
class BankAccount:
    def __init__(self, account_number, balance):
        self.__account_number = account_number
        self.__balance = balance

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount

    def withdraw(self, amount):
        if 0 < amount <= self.__balance:
            self.__balance -= amount

    def get_balance(self):
        return self.__balance

# Example usage
account = BankAccount("123456789", 1000)
account.deposit(500)
account.withdraw(200)
print(f"Balance: {account.get_balance()}")
```

11. Overriding __str__ and __add__ Methods

python

```
class Item:
    def __init__(self, name, price):
        self.name = name
        self.price = price

    def __str__(self):
        return f"Item(name: {self.name}, price: {self.price})"

    def __add__(self, other):
        return self.price + other.price

item1 = Item("Book", 20)
item2 = Item("Pen", 5)
print(item1)                    # Output: Item(name: Book, price: 20)
print(item1 + item2)             # Output: 25
```

12. Decorator for Measuring Execution Time

python

```
import time

def timing_decorator(func):
    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        print(f"Execution time: {end_time - start_time:.4f} seconds")
        return result
    return wrapper

@timing_decorator
def example_function(n):
    time.sleep(n)
    return "Done"

example_function(2)
```

13. Diamond Problem in Multiple Inheritance

Definition: The Diamond Problem occurs in multiple inheritance when a class inherits from two classes that have a common base class.
Python Resolution: Python uses C3 linearization (MRO) to handle the Diamond Problem by providing a consistent method resolution order.

14. Class Method to Track Instance Count

python

```
class MyClass:
    instance_count = 0

    def __init__(self):
        MyClass.instance_count += 1
    @classmethod
    def get_instance_count(cls):
        return cls.instance_count

# Example usage
a = MyClass()
b = MyClass()
print(MyClass.get_instance_count())  # Output: 2
```

15. Static Method to Check Leap Year

python

```
class Utils:
    @staticmethod
    def is_leap_year(year):
        return (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0)

# Example usage
print(Utils.is_leap_year(2020))  # Output: True
print(Utils.is_leap_year(2021))  # Output: False
```

This comprehensive guide covers