LECTURER: TAI LE QUY

OBJECT ORIENTED AND FUNCTIONAL PROGRAMMING WITH PYTHON

Thanks Prof. Dr. Max Pumperla for his contribution

TOPIC OUTLINE

Object oriented programming	1 +2
Functional programming	3
Projects and testing in Python	4
Working with Database in Python	5
Documenting a project	6

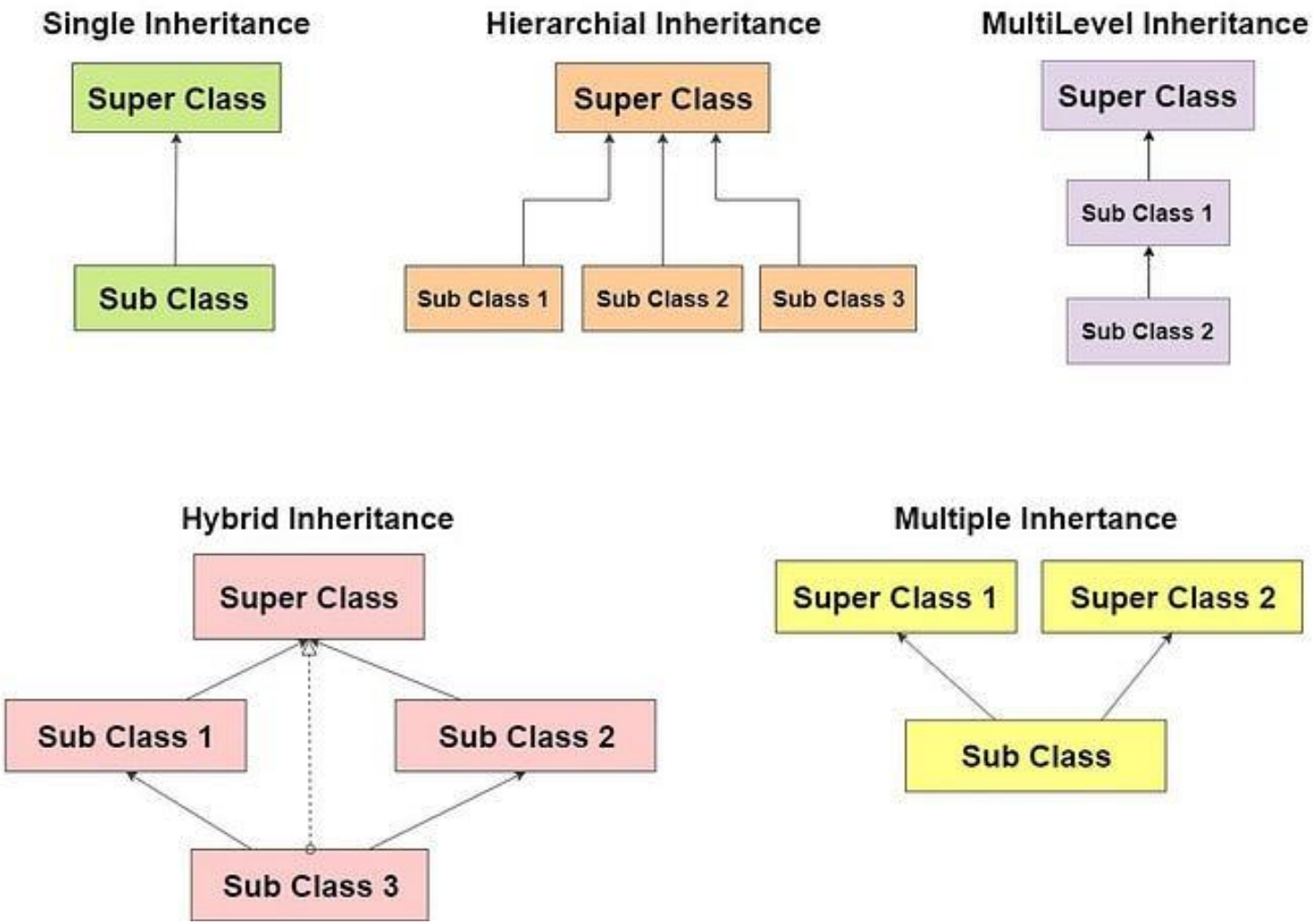
UNIT 2

OBJECT ORIENTED PROGRAMMING

INHERITANCE RECAP

```python
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        print("This animal speaks.")

class Dog(Animal):
    def __init__(self, name):
        super().__init__(name)

    def speak(self):
        print("Woof!")

dog = Dog("Pelle")
animal = Animal("Generic")

dog.speak() # Output: Woof!
animal.speak() # Output: This animal speaks.
```

TYPES OF INHERITANCE



Source of the image: https://python.plainenglish.io/python-inheritance-45df7ab2d524

Unit 1

TRANSFER TASK

TRANSFER TASK

- Write a Python program to create a **People** class with:
  - Instance attributes: name, date of birth, nationality
  - Method: show the information
- Create a Student class that inherits from the **People** class with
  - New attributes: major, university
  - Method: show the information

TRANSFER TASK
PRESENTATION OF THE RESULTS

Please present your results.

The results will be discussed in plenary.

STUDY GOALS

- Inheritance in Python
- Polymorphism
- Encapsulation
- Abstraction
- More examples

POLYMORPHISM BASICS

```python
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        print("This animal speaks.")

class Dog(Animal):
    def __init__(self, name):
        super().__init__(name)

    def speak(self):
        print("Woof!")

class Cat(Animal):
    def __init__(self, name):
        super().__init__(name)

    def speak(self):
        print("Meow!")

dog.speak() # Output: Woof!
animal.speak() # Output: This animal speaks.
```

POLYMORPHISM BASICS

- Different objects inheriting from a common base class
- Same method names, different behaviors (polymorph)
- Promotes code reusability
- “If it works with the base class, it works with all children”
- Often increases readability (but: more abstraction)

```
def make_animal_speak(animal):
    animal.speak()

# Create instances of different animals
dog = Dog("Fido")
cat = Cat("Whiskers")

# Call the make_animal_speak function with different animals
make_animal_speak(dog)     # Output: Woof!
make_animal_speak(cat)     # Output: Meow!
```

ENCAPSULATION BASICS

- Classes “encapsulate” data and methods in one entity
- Hide internal state from outside access
- Data integrity and control
- promotes modularity and separation of concerns
- Prevent data modifications (public, protected, private members)
- Allows internal refactoring while keeping the interface stable

ENCAPSULATION EXAMPLES

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

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

    def withdraw(self, amount):
        if amount <= self.__balance:
            self.__balance -= amount
        else:
            print("Insufficient balance.")

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

```python
# Create an instance of the BankAccount class
account = BankAccount("1234567890", 1000)

# Accessing private attributes causes an AttributeError
print(account.__balance)  # Raises an AttributeError

# Accessing private attributes using getter method
print(account.get_balance())  # Output: 1000

# Depositing and withdrawing funds
account.deposit(500)
account.withdraw(200)

# Checking the updated balance
print(account.get_balance())  # Output: 1300
```

ABSTRACTION BASICS

- Hide non-essentials, represent essentials
- Create “**abstract**” classes defining characteristics and behaviour
- No implementation details
- Battles complexity (think about “clean” interfaces)
- Work with concepts, not specific intricacies
- Separate what (interface) and how (implementation)

ABSTRACTION EXAMPLES

```python
from abc import ABC, abstractmethod

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

    @abstractmethod
    def perimeter(self):
        pass

class Rectangle(Shape):
    def __init__(self, length, width):
        self._length = length
        self._width = width

    def area(self):
        return self._length * self._width

    def perimeter(self):
        return 2 * (self._length + self._width)

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

    def area(self):
        return 3.14 * self.__radius ** 2

    def perimeter(self):
        return 2 * 3.14 * self.__radius
```

ABSTRACTION EXAMPLES

- Circle and Rectangle exhibit polymorphism
- Classes encapsulate data (protected & private)
- We use abstract base classes as interfaces
- Circle and Rectangle implement the interface

```
# Creating objects of different shapes
rectangle = Rectangle(5, 3)
circle = Circle(4)

# Calling the abstract methods on the objects
print(rectangle.area())  # Output: 15
print(rectangle.perimeter())  # Output: 16
print(circle.area())  # Output: 50.24
print(circle.perimeter())  # Output: 25.12
```

Unit 2

TRANSFER TASK

TRANSFER TASK

- Write a Python program to create an abstract **People** class with:
  - @abtractmethod
- Create a Student class that inherits from the **People** class with
  - Attributes: name, university, country
  - Method: show the information
- Create a Customer class that inherits from the **People** class with
  - Attributes: name, product
  - Method: show the information

TRANSFER TASK
PRESENTATION OF THE RESULTS

Please present your results.

The results will be discussed in plenary.