ĐẠI HỌC ĐÀ NẴNG
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG VIỆT - HÀN
Vietnam - Korea University of Information and Communication Technology

Chapter 3

Object-Oriented Programming in Python

Thu Huong Nguyen, PhD
Si Thin Nguyen, PhD

KHMT
Computer Science

http://vku.udn.vn/

About Authors

Thu Huong Nguyen

PhD in Computer Science at Université Côte d'Azur, France

Email: nthuong@vku.udn.vn

Address: Faculty of Computer Science, VKU

Si Thin Nguyen

PhD in Computer Science at Soongsil University, Korea

Email: nsthin@vku.udn.vn

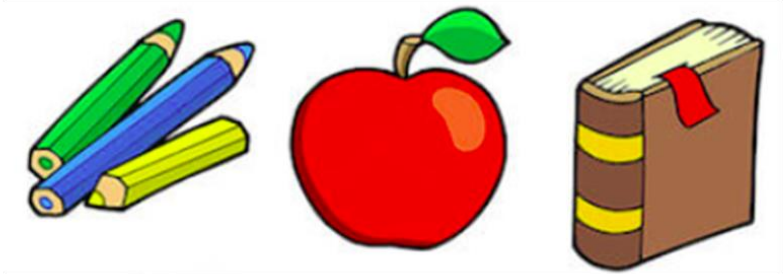
Address: Faculty of Computer Science, VKU

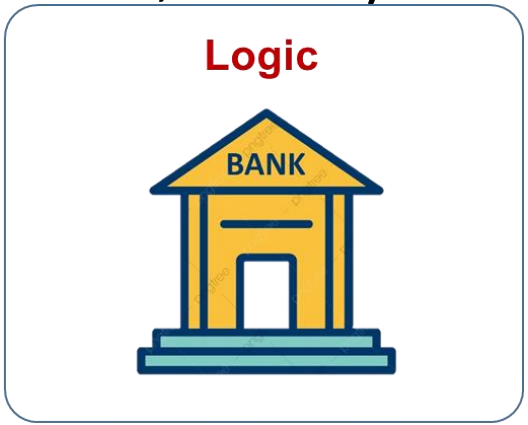
Chapter Content

- Class & Object
- Attributes
- Constructor
- Method
- Inheritance
- Polymorphism
- Abstraction
- Encapsulation

Class & Object

- **Object**: is the entity/instance with “state” and “behavior”
- **Features**:
  - Physical or logic entity
  - Having three characteristics: state, behavior, identity





Class & Object

➢ **Class:**

- Is the template or blueprint from which objects are made
- Can be defined as a collection of object

➢ **Including:**

- Data: attribute/ field/ instance variable
- Methods
- Constructor

.....

Class & Object

➢ **Syntax:**

```
class ClassName:

#statement_suite
```

Example:

```
class Employee:
    id = 10
    name = "Devansh"
    def display (self):
        print(self.id,self.name)
```

Attributes

- **Concept:** The data held by an object is represented by its attributes
- **Types:**

- Class variables : defined within the scope of the class, but outside of any methods

- Instance variables: are tied to the instance (objects) than class

Attributes

- **Example:**

```
class Person:
    # instance_count is class variable
    instance_count = 0
    def __init__(self, name, age):
        Person.instance_count += 1
    # name, age are instance variables
        self.name = name
        self.age = age
```

Constructor

- **Concept:** is a special type of method (function) which is used to initialize the instance members of the class.
- Constructors can be of two types.
  - Parameterized Constructor
  - Non-parameterized Constructor

Constructor

- **Syntax:** `__init__(<parameter>)`
- **Example:**

```
class Employee:
    def __init__(self, name, id):
        self.id = id
        self.name = name

    def display(self):
        print("ID: %d \nName: %s" % (self.id, self.name))
    emp1 = Employee("John", 101)
    emp2 = Employee("David", 102)
```

Method

- Instance method
- Class method
- Static method
- Special method
- Getter, setter method

Instance method

- **Concept**: it is tied to an instance of the class
- **Example**:

```
class Employee:
    id = 0
    name = "Devansh"
    def display (self):
        print(self.id,self.name)
```

- **“Display(seft)”** is a instance method
- “**self**” is used as a reference variable, which refers to the current class object. It is always the first argument in the function definition. However, using self is optional in the function call

Class method

- **Concept**: behaviour that is linked to the class rather than an individual object
- **Example**:

```
class Employee:
    id = 0
    name = "Devansh"
    @classmethod
    def increment_id(cls):
        id+=1
    def display (self):
        print(self.id,self.name)
```

- **“increment_id(cls)”** is a instance method
- Is decorated with “**@classmethod**” keyword and take a first parameter with **“cls”**

Static method

- **Concept**: is defined within a class but are not tied to either the class nor any instance of the class

- **Example**:

```
class Employee:
    id = 0
    name = "Devansh"
    @staticmethod
    def static_function():
        print(“Static method”)
```

- is decorated with the **@staticmethod** decorator
- the same as free standing functions but are defined within a class

Special method

- Start and end with a double underbars ('__').
- You should never name one of your own methods or functions __<something>__ unless you intend to (re)define some default behaviour.
- **Example**: __init__(),
  __str__()
  __dict__()
  __doc__()
  __module__()

Getter and setter methods

- **Concept**: used to access the values of objects
- **Getter methods:** decorated with the **@property** decorator
- **Setter methods:** decorated with the **@attribute_name.setter** decorator

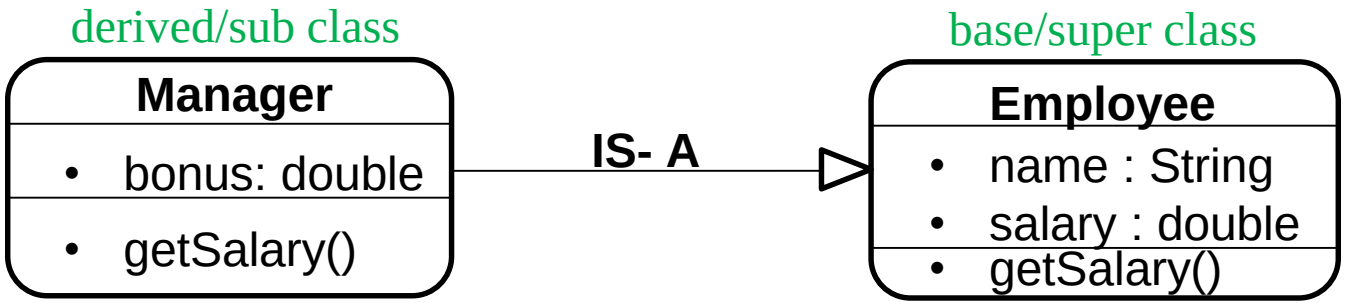
Getter and setter methods

- **Example:**

```
class Example:
    # Attribute
    __domain = ''
    # Getter
    @property
    def domain(self):
        return self.__domain
    # Setter
    @domain.setter
    def domain(self, domain):
        self.__domain = domain
```

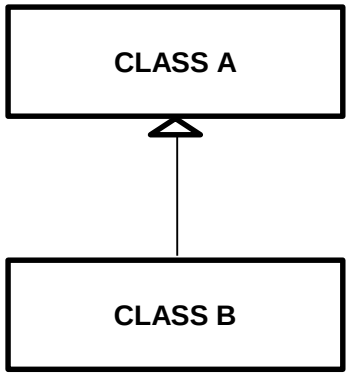
Inheritance

- **Concept**: the child class acquires the properties and can access all the data members and functions defined in the parent class
- **Note**:
  - Reuse code
  - Method Overriding
  - **Syntax**: **class** derived-**class**(base **class**):
    <derived**class**-body>

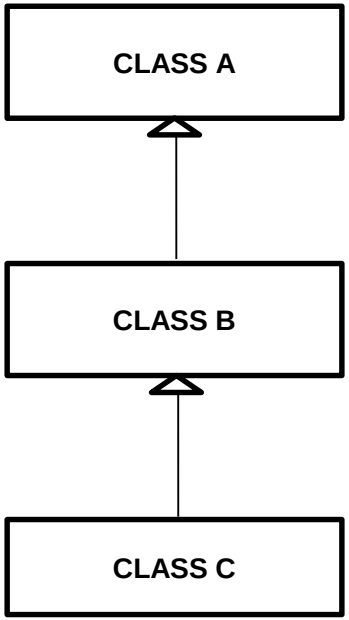


Inheritance

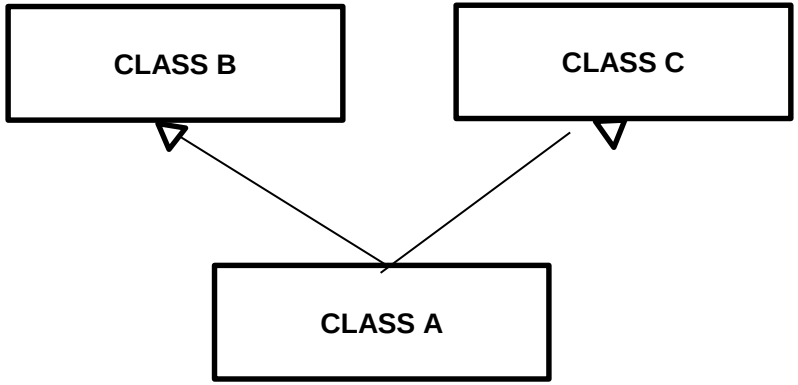
Single



Multi-level inheritance



Multiple - inheritance



Inheritance

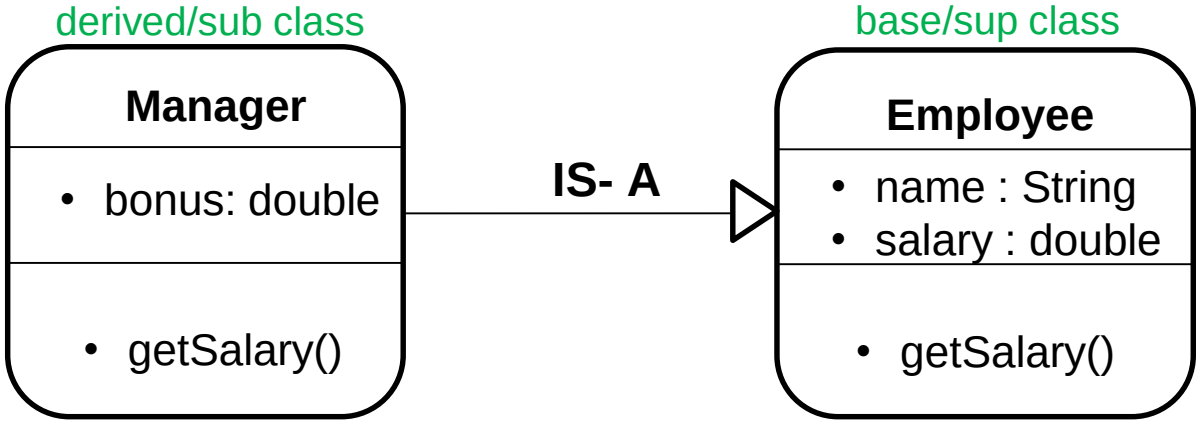
- **"super()":** is used to call method and constructor of parent class

```
class Manager(Employee):
    # three private attributes: __name, __salary and __bonus
    # Constructor
    def __init__(self, name, salary, bonus):
        super().__init__(name, salary)
        self.__bonus=bonus
    # Override "get_salary" method
    def get_salary(self):
        # return self.__salary + self.__bonus
        # return self.get_salary()+self.__bonus
        return super().get_salary()+ self.__bonus
```

```
class Employee:
    # two attributes: __name and __salary
    # Constructor
    def __init__(self, name, salary):
        self.__name=name
        self.__salary=salary
    # method get_salary
    def get_salary(self):
        return self.__salary
    # How to overide "get_salary" method on Manager class
    # return self.__salary + self.__bonus      --> Error
    # return self.get_salary()+self.__bonus    --> Error
```

Polymorphism

- **Polymorphism:** one task can be performed in different ways
- **Note:** Runtime polymorphism: Method Overriding



Method Overriding

- **Concept:** subclass (child class) has the same method as declared in the parent class, it is known as method overriding
- **Note:** - Same name and the number of parameters
  - Runtime polymorphism
  - The prefer in the order: left to right, up to down

```
class Manager(Employee):
    # three private attributes: __name, __salary and __bonus
    # Constructor
    def __init__(self, name, salary, bonus):
        super().__init__(name, salary)
        self.__bonus=bonus
    # Override "get_salary" method
    def get_salary(self):
        # return self.__salary + self.__bonus
        # return self.get_salary()+self.__bonus
        return super().get_salary()+ self.__bonus
```

```
class Employee:
    # two attributes: __name and __salary
    # Constructor
    def __init__(self, name, salary):
        self.__name=name
        self.__salary=salary
    # method get_salary
    def get_salary(self):
        return self.__salary
    # How to overide "get_salary" method on Manager class
    # return self.__salary + self.__bonus      --> Error
    # return self.get_salary()+self.__bonus    --> Error
```

Abstraction

Concept: main goal is to handle complexity by hiding unnecessary details from the user

Note:

- We know “what it does” but we don’t know “how it does”
- Abstract Base Classes (ABCs)

Remote

Sending Message

Abstract class and abstract method

- **Abstract class:** is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class)
- **Note:**
  - Having at least one abstract method
  - Can not be instantiated themselves
  - **Abstract method:** can only be used in an abstract class, and it does not have a body. The body is provided by the subclass (inherited from).
- **Note:**
  - Is declared in subclass

Abstract Class

- **Abstract Base Classes (ABCs) :**

  - Can be used to define generic (potentially abstract) behaviour that can be mixed into other Python classes and act as an abstract root of a class hierarchy.

  - There are many built-in ABCs in Python including (but not limited to): IO, numbers, collection,…modules

- **Declared an Abstract Class**

  - Step 1 :import ABCs, abstract method

  - Step 2: Declared an Abstract Class inheritance from ABC class in step 1

  - Step 3: Declared Abstract Methods

Abstract Class

- **Example 1:**

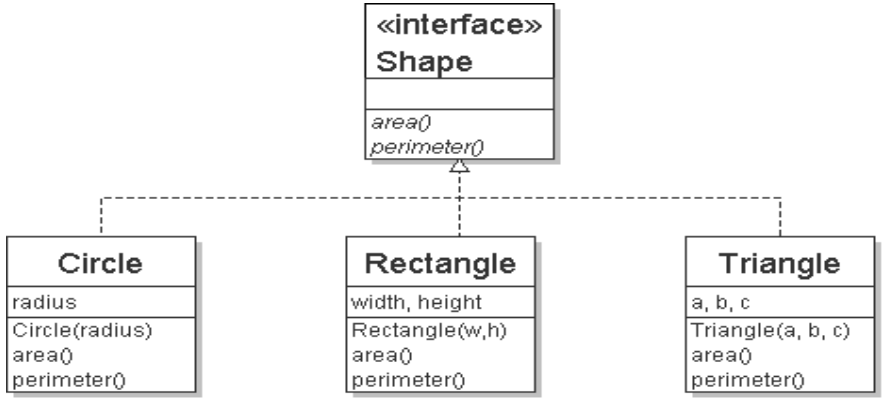
```
from abc import ABC, abstractmethod
class Vidu(ABC):
  @abstractmethod
  def methodName(self):
    pass
```

- **Example 2:**

```
from collections import MutableSequence. abstractmethod
class Bag(MutableSequence):
  @abstractmethod
  def methodName(self):
    pass
```

Interface

- **Interface:** this is a contract between the implementors of an interface and the user of the implementation guaranteeing that certain facilities will be provided. Python does not explicitly have the concept of an interface contract (note here interface refers to the interface between a class and the code that utilizes that class).
- **Example:** Create an “interface” Shape and subclasses: Circle, Rectangle, Triangle



Interface

- "Interface" Shape

```
from abc import ABC, abstractmethod
class Shape(ABC):
    @abstractmethod
    def area(self):
        pass
    @abstractmethod
    def perimeter(self):
        pass
```

- Circle Class

```
class Circle(Shape):
    def __init__(self, radius) -> None:
        super().__init__()
        self.radius=radius
    def area(self):
        return 3.14*self.radius*self.radius
    def perimeter(self):
        return 2*3.14*self.radius
```

- Rectangle Class

```
class Rectangle(Shape):
    def __init__(self, width, height) -> None:
        super().__init__()
        self.width=height
        self.height=height
    def area(self):
        return self.width*self.height
    def perimeter(self):
        return (self.width+self.height)*2
```

Encapsulation

- **Concept:** It is used to restrict access to methods and variables. In encapsulation, code and data are wrapped together within a single unit from being modified by accident.
- **Note:**
  - Private Attributes
  - Using getter and setter methods to access data

```
class Student:
    def __init__(self, univer):
        self.__univer=univer
    # getter
    @property
    def univer(self):
        return self.__univer
    # Missing setter method
    # only read data
```

```
a= Student("VKU")
a.__univer="VKU University"
print(a.univer)
# result is not changed: "VKU"
```

```
class Student:
    def __init__(self, univer):
        self.__univer=univer
    # setter
    @univer.setter
    def univer(self,univer):
        self.__univer=univer
    # Missing getter method
    # Only write data
```

```
a= Student("VKU")
a.univer="VKU University"
print(a.__univer)
# Cannot print
```