What is OOP

OOP - Python Session 1

Table of content

Functional Programming vs OOP

Procedural Programming vs OOP

- **Procedural programming** là lập trình theo dạng **thủ tục, câu lệnh, step by step**

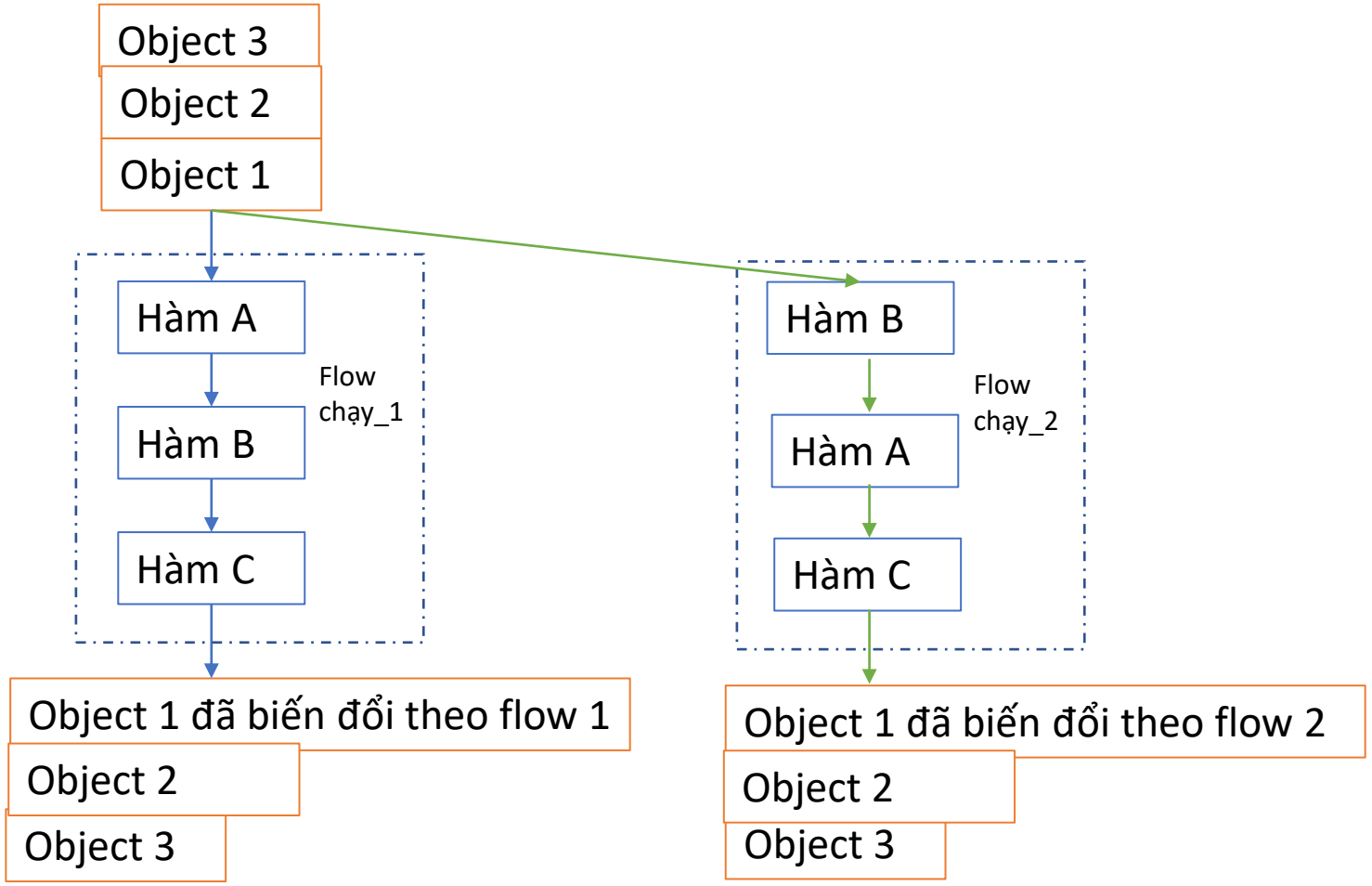
```
def Sum(sample_list):
    total = 0
    for x in sample_list:
       total += x
    return total

list1 = [10, 200, 50, 70]
list2 = [3, 26, 33, 13]
print(Sum(list1))
print(Sum(list2))
```

Procedural Programming

```
In [15]: class employee():
             def __init__(self,name,age,id,salary):
                 self.name = name
                 self.age = age
                 self.salary = salary
                 self.id = id
         def earn(self):
                 pass

         class childemployee1(employee):

             def earn(self):#Run-time polymorphism
                 print("Hello")

         class childemployee2(employee):

             def earn(self):
                 print("Hello there")

         a = childemployee1
         a.earn(employee)
         b = childemployee2
         b.earn(employee)

         Hello
         Hello there
```

OOP Programming

Procedural Programming vs OOP

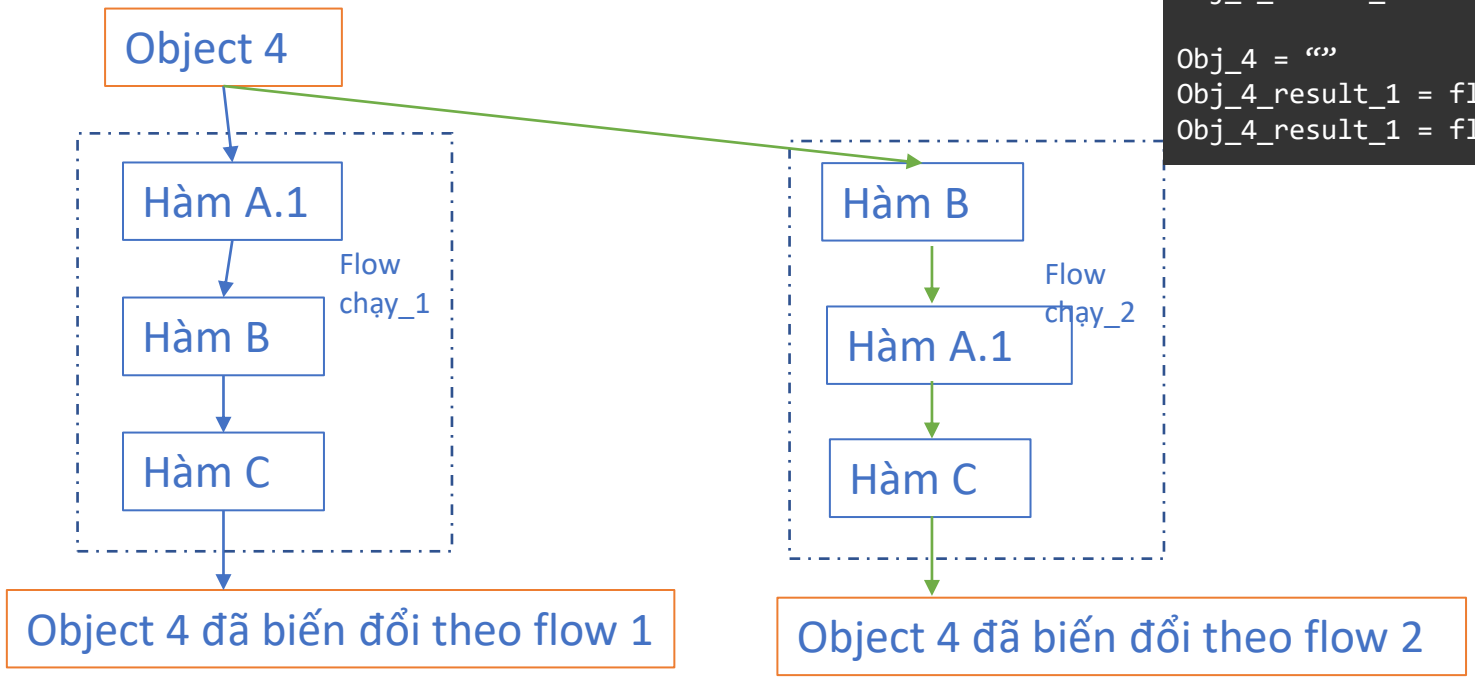



```
def function_a(x):
    x = .....
    return x

def function_b(x):
    x = .....
    return x

def function_c(x):
    x = .....
    return x

def flow_chay_1(x):
    x= function_a(x)
    x= function_b(x)
    x= function_c(x)
    return x

def flow_chay_2(x):
    x= function_b(x)
    x= function_a(x)
    x= function_c(x)
    return x
```

```
Obj_1 = “”
Obj_1_result_1 = flow_chay_1(obj_1)
Obj_1_result_2 = flow_chay_2(obj_1)

Obj_2 = “”
Obj_2_result_1 = flow_chay_1(obj_2)
Obj_2_result_2 = flow_chay_2(obj_2)
…
```

Flow 1 Procedural Programming

Procedural Programming vs OOP




```
Obj_1 = “”
Obj_1_result_1 = flow_chay_1(obj_1)
Obj_1_result_2 = flow_chay_2(obj_1)

Obj_4 = “”
Obj_4_result_1 = flow_chay_1.1(obj_4)
Obj_4_result_1 = flow_chay_2.1(obj_4)
```

Object 4 muốn chạy qua **hàm A.1 thay vì hàm A** như object 1, 2, 3. Nhưng vẫn muốn **giữ 2 flow chạy**
Thì sẽ như thế nào?

Flow 2 Procedural Programming

```
def function_a(x):
    x = .....
    return x

def function_a.1(x):
    x = .....
    return x

def function_b(x):
    x = .....
    return x

def function_c(x):
    x = .....
    return x

def flow_chay_1(x):
    x= function_a(x)
    x= function_b(x)
    x= function_c(x)
    return x

def flow_chay_2(x):
    x= function_b(x)
    x= function_a(x)
    x= function_c(x)
    return x

def flow_chay_1.1(x):
    x= function_a.1(x)
    x= function_b(x)
    x= function_c(x)
    return x

def flow_chay_2.1(x):
    x= function_b(x)
    x= function_a.1(x)
    x= function_c(x)
    return x
```

Procedural Programming vs OOP

```
class Flow:
    def __init__(self,x):
        self.x = x
  def function_a(self):
        self.x = ....
  def function_b(self):
        self.x = ....
  def function_c(self):
        self.x = ....

 def run_flow_1(self):
        function_a()
        function_b()
        function_c()
  def run_flow_2(self):
        function_b()
        function_a()
        function_c()
```

```
obj_1 = "..."
obj_1_flow = Flow(obj_1)
Obj_1_result_1 = obj_1_flow.run_flow_1()
Obj_1_result_2 = obj_1_flow.run_flow_2()
…

obj_4 = "..."
obj_4_flow = Flow_1(obj_4)
Obj_4_result_1 = obj_4_flow.run_flow_1()
Obj_4_result_2 = obj_4_flow.run_flow_2()
```

```
class Flow_1(Flow):
    def __init__(self,x):
        super(self,x)
    def function_a(self):
      self.x =.... # thay đổi mà object 4 muốn
```

Flow_1 kế thừa từ Flow 2 hàm run_flow nên không cần phải tạo lại hàm này

```
def function_a(x):
    x = .....
    return x

def function_a.1(x):
    x = .....
    return x

def function_b(x):
    x = .....
    return x

def function_c(x):
    x = .....
    return x

def flow_chay_1(x):
    x= function_a(x)
    x= function_b(x)
    x= function_c(x)
    return x

def flow_chay_2(x):
    x= function_b(x)
    x= function_a(x)
    x= function_c(x)
    return x

def flow_chay_1.1(x):
    x= function_a.1(x)
    x= function_b(x)
    x= function_c(x)
    return x

def flow_chay_2.1(x):
    x= function_b(x)
    x= function_a.1(x)
    x= function_c(x)
    return x
```

Procedural Programming vs OOP

Procedural Programming	OOP Programming
Mindset câu lệnh và hàm	Mindset khung xương (class) và đối tượng
Hàm nên không có tính kế thừa và mở rộng	Khung xương có tính kế thừa và mở rộng
Không cần phải bảo mật data	Có thể bảo mật data
Tập trung vào sự tách biệt chức năng, dễ kiểm tra, gỡ lỗi	Tập trung vào sự mở rộng mà không làm thay đổi những biến trước đó

OOP là **phiên bản nâng cấp** của Procedural Programming, người dung có thể **dung syntax OOP với mindset Procedural Programming** được, nhưng **ngược lại** thì **không**

4 Principle of OOP

4 Principle of OOP

- - Tính kế thừa
- - Tính đa hình
- - Tính đóng gói
- - Tính trừu tượng

} **Tính mở rộng**

Tính mở rộng

- Tính mở rộng được kết hợp từ **kế thừa** và **đa hình**.
- Ví dụ: mở rộng them cho lớp cha (Person) một lớp con (Student) nữa.

```python
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def print_info(self):
        print(f"Info: {self.name} is {self.age}“)
```

```python
class Student(Person):
    def __init__(self,name, age, classroom):
        super(self,name, age)
        self.classroom = classroom
    def print_info(self):
        print(f"Info: {self.name} is {self.age} who studying in class {self.classroom}")
```

```python
person_1 = Person("A", "20")
person_2 = Student("B", "21", "A1")

person_1.print_info()
person_2.print_info()
```

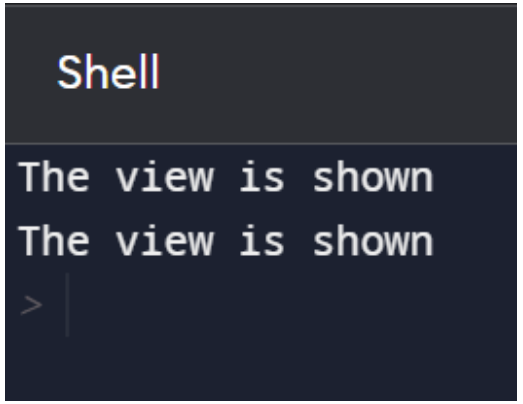
Kế thừa: lớp Student thừa hưởng hàm khởi tạo init và hàm print_info của lớp Person

Đa hình: Cùng là 2 function print_info(self) nhưng ở 2 lớp khác nhau, thì Python sẽ hiểu là dung hàm nào

Tính kế thừa

- Là tính **thừa hưởng** những **thuộc tính** (attribute) và **hàm** (method) cho lớp con mà **lớp cha sở hữu**.

```
class MainView():
    def __init__(self):
      self.resolution = "1920x1080"
      self.name = "view"

    def show_view(self):
        return f"{self.name} is shown"

class ChildView(MainView):
    def __init__(self):
        super().__init__()

main_view = MainView()
print(main_view.show_view())
child_view = ChildView()
print(child_view.show_view())
```



Tính đa hình

- Tính đa hình là khi class con được **kế thừa** từ lớp cha, và lớp con **muốn thay đổi attribute hoặc method** của lớp cha cho lớp con

```
class MainView():
    def __init__(self):
      self.resolution = "1920x1080"
      self.name = "view"

    def show_view(self):
        return f"{self.name} is shown"

class ChildView(MainView):
    def __init__(self):
        super().__init__()
    def show_view(self):
        return f"Child {self.name} is shown"

main_view = MainView()
print(main_view.show_view())
child_view = ChildView()
print(child_view.show_view())
```

```
The view is shown
Child view is shown
>
```

Tính năng thay đổi method của lớp cha này có tên gọi là : **“overriding method”**

Tính đóng gói (bảo mật)

- Bảo mật này sẽ có **2 level**:
  - **private/protected**: được truy cập trong class và class con
  - **public**: được truy cập ở khắp nơi

```
class Person:
    def __init__(self):
        self.bien_public = "Public"
        self.__bien_private = "Private"

    def truy_cap_bien_private(self):
        print("Truy cap private: ",self.__bien_private)

class Student(Person):
    def __init__(self):
        super().__init__()

student_1 = Student()
print(student_1.bien_public)
student_1.truy_cap_bien_private()
print(student_1.__bien_private)
```

```
ERROR!
Public
Truy cap private:  Private
Traceback (most recent call last):
  File "<string>", line 16, in <module>
AttributeError: 'Student' object has no attribute '__bien_private'
>
```

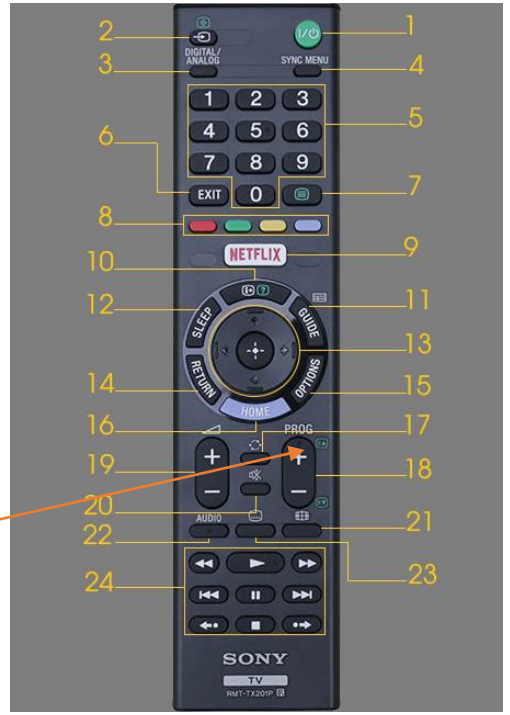
Tính trừu tượng

Nếu ta chỉ cho người dung access vào "**abstractClassName**" source code thì họ không biết được ta sẽ viết hàm đó như thế nào

```python
from abc import ABC, abstractmethod

class abstractClassName(ABC):
    @abstractmethod
    def print_info(self):
        pass

class Person(abstractClassName):
    def __init__(self):
        self.name = "A"
        self.age = 20
    def print_info(self):
        print(f"{self.name} is {self.age}")

person = Person()
person.print_info()
```

Che giấu thông tin thực thi

Tuy nhiên, abstraction chỉ mang tính **che bớt thông tin chi tiết và giảm độ phức tạp** cho người dung cơ bản (người **không cần quan tâm how it work?**), **không có nghĩa là bảo mật thông tin**. Cho nên nếu chỉ áp dụng abstraction thì thông tin vẫn có thể bị chỉnh sửa



Level type in OOP

Level type in OOP

- **Class level attribute** : Là attribute của class đó.
- **Instance level attribute**: Là attribute của từng object tạo ra từ class đó.

```
class Person:
    bien_class_level = "class-level"
    def __init__(self, bien_instance_level):
        self.bien_instance_level = bien_instance_level

person_1 = Person("instance_1")
person_2 = Person("instance_2")

print(person_1.bien_class_level,",", person_1.bien_instance_level)
print(person_2.bien_class_level,",", person_2.bien_instance_level)
```

```
class-level , instance_1
class-level , instance_2
>
```

Thuộc tính: Class level

- Khi khởi tạo **class-level attribute** thì tất cả các object của class đó đều có cùng giá trị.
- **Class-level** dung cho khi khởi tạo một biến.

```
class Animal:
    name = "animal"
    def __init__(self):
        pass
    def set_name(self,name):
       self.name = name

animal_1 = Animal()
print(animal_1.name)
animal_1.set_name("new_name")
print(animal_1.name)
```

==

```
class Animal:
    def __init__(self):
        self.name="animal"

    def set_name(self,name):
        self.name = name
```

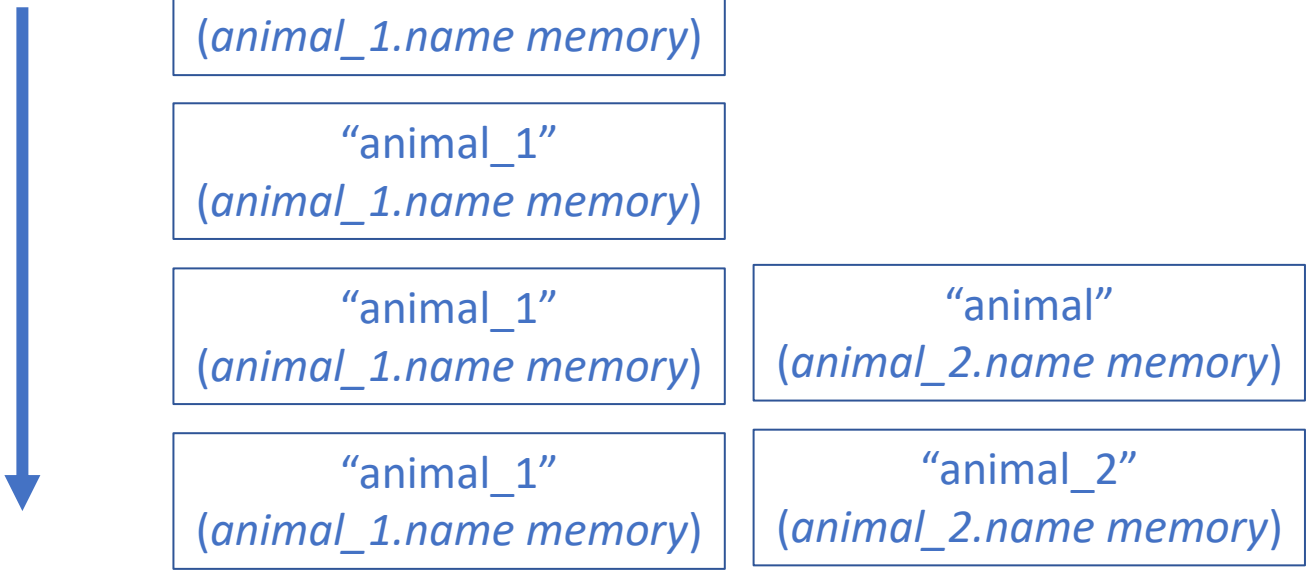
Ở đây cho ra **kết quả giống nhau**, ta chỉ chuyển sự khởi tạo từ hàm **__init__()** sang **class-level attribute** thôi

Thuộc tính: Instance level

- Là **attribute riêng biệt** của từng object chứ nó **không sharing** attribute đó cho nhau.
- Khi khởi tạo instance attribute, class sẽ tạo ra **một vùng memory** của biến đó cho từng object.

```
class Animal:
    def __init__(self):
        self.name="animal"

    def set_name(self,name):
        self.name = name
```

```
animal_1 = Animal()
animal_1.set_name(“animal_1")

animal_2 = Animal()
animal_2.set_name(“animal_2")
```



Tham số “self”

- Vì có **rất nhiều vùng nhớ** cho cùng **một tên biến của class**, nên hàm của OOP, yêu cầu người dung truyền vào object đó để truy cập tới **vùng nhớ của object đó.**
- **Self chính là object mà minh đã khởi tạo**. Do đó có thể không dung từ self mà dung từ khác cũng được

```
class Animal:
    def __init__(self):
        self.name="animal"

    def get_self(self):
        print(self)

animal_1 = Animal()

print(animal_1)
animal_1.get_self()
Animal.get_self(animal_1)
```

```
<__main__.Animal object at 0x7f85ad2fe610>
<__main__.Animal object at 0x7f85ad2fe610>
<__main__.Animal object at 0x7f85ad2fe610>
>
```

Tham số self

```
class Animal:
    def __init__(self):
        self.name="animal"

    def get_self(self):
        print(self)

    def set_name(self,name):
        self.name = name

animal_1 = Animal()

animal_1.set_name("new_name_1")
print(animal_1.name)

Animal.set_name(animal_1,"new_name_2")
print(animal_1.name)
```

```
new_name_1
new_name_2
>
```

Sẽ có gọi hàm set_name ở **2 cách như trên**, đều cho ra **cùng một kết quả.**

Bài tập

- Diễn giải **3 chủ đề** đã sharing dưới **văn phong của minh** và **cho ví dụ**:
  - Functional vs OOP Programming
  - 4 Principle of OOP
  - Level type of OOP