Lesson 2a -
OOP (Object Oriented Programming) in Python

- S.P. Chong

Ref: https://realpython.com/python3-object-oriented-programming/

Contents

In this lesson, you will learn the following:

- What is Object Oriented Programming (**OOP**)?
- **Classes** in Python
- **Instances** in Python
- **Class attribute** vs. **instance attribute**
- **Methods**
- **Inheritance**

Although you are not required to use OOP in this elective, you may need to import & use **modules** (such as RPi.GPIO, time, random, spidev, Adafruit_DHT, csv, matplotlib, numpy), most of which are written in the OOP way.

Thus, this lesson gives you some **basic concepts** in OOP, so that you can understand the codes that import & use these modules.

What is Object Oriented Programming?

- **OOP** is a programming paradigm (or model) which provides a means of structuring programs so that **properties** (or **attributes**) and **behaviours** (or **methods**) are bundled into individual **objects**.
- For instance, an object could represent a student with properties such as name, class and hobby, and behaviours such as attending lecture, eating lunch and doing CCA. Or an email with properties such as recipient list, subject and body, and behaviours such as adding attachments and sending.
- Put another way, OOP is an approach for modelling real-world objects.

What is Object Oriented Programming? (cont.)

- Another common programming paradigm is **procedural** programming which structures a program like a recipe in that it provides a set of steps, which flow sequentially in order to complete a task.
- **Python** is a **multi-paradigm** programming language. You can choose to use either OOP or procedural programming or a mix.

Classes in Python

- Each thing or **object** is an **instance** of some **class**.
- The **simple** data structures / **data types** in Python, such as numbers, strings, and lists are designed to represent simple things such as the cost of something, the name of a person, and your favourite fruits, respectively.
- What if you need to represent something much more complicated?
- For instance, you want to track a group of student taking an elective module. The first **property / attribute** can be the student's name, the second attribute can be his/her age, the third attribute can be the school he/she belongs.

Classes in Python (cont.)

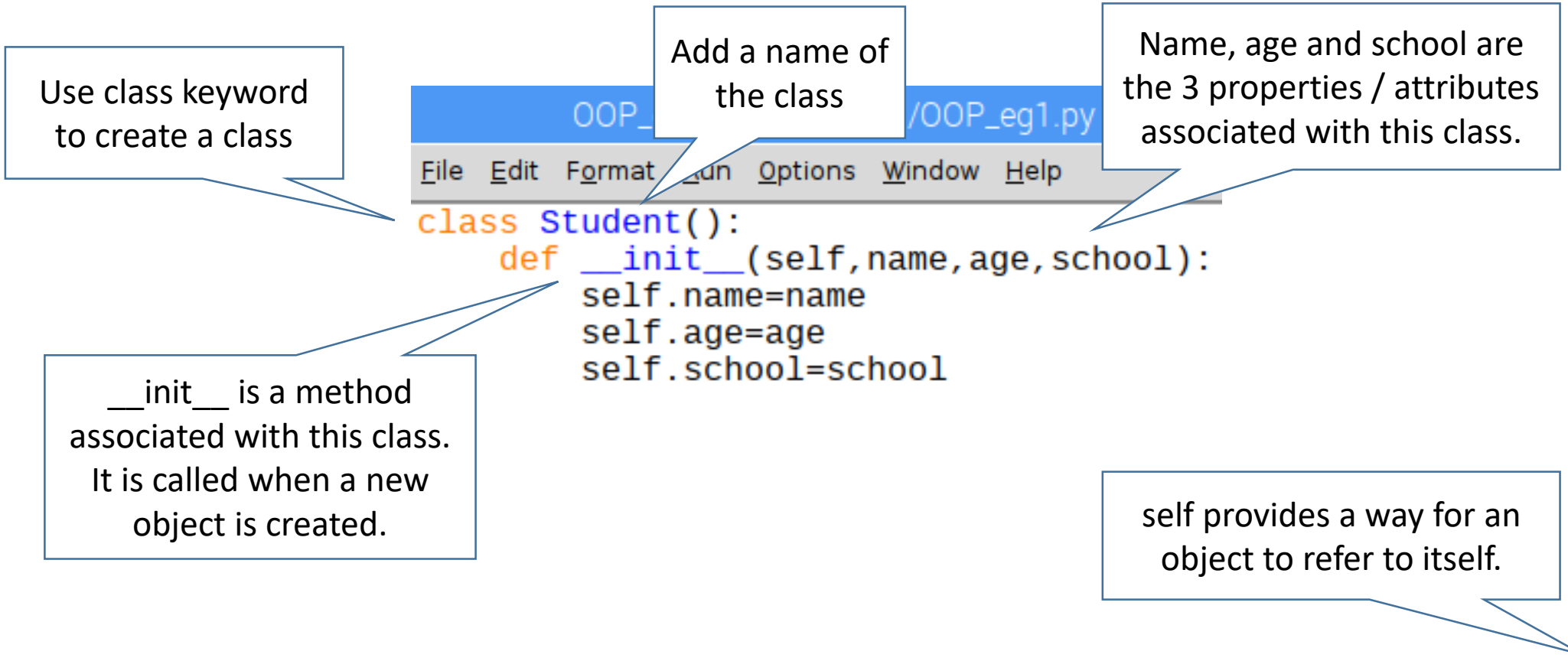
- Classes are used to create new **user-defined data structures** that contain information about something.
- In the case of students taking an elective module, we can create a Student( ) class to track properties about the student, like the name, age and school.
- The following lines of code create such a class:

OOP_eg1.py - /home/pi/OOP_eg1.py (3.5.3)

File Edit Format Run Options Window Help

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

Classes in Python (cont.)



Classes in Python (cont.)

- Note that a class just provides structure – it's a blueprint for how something should be defined, but it doesn't actually provide any real content.
- For instance, the Student ( ) class above may specify that name, age and school are necessary for defining a student taking an elective module, but it doesn't actually state what a specific student's name, age or school is.

Instances in Python

- While the **class** is the blueprint, an **instance** is a copy of the class with actual values. For instance, a student with name “Alice”, age 18, school “CLS”.
- Another way of looking at class & instance is: a class is like a survey form, which defines the needed information. After you fill out the form, your specific copy is an instance of the class.
- **Instantiating** is a term for creating a new, unique instance of a class.

Instances in Python (cont.)

- The following lines of code instantiates 3 instances of the Student class, and assign them to the variables Stu1, Stu2 and Stu3:

OOP_eg1.py - /home/pi/OOP_eg1.py (3.5.3)

File Edit Format Run Options Window Help

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

Stu1=Student('Alice',18,'CLS')
Stu2=Student('Melvyn',19,'EEE')
Stu3=Student('John',18,'ABE')
```

```
Alice from CLS is 18 years old.
```

Instances in Python (cont.)

When Stu1 is instantiated, the __init__ method will execute automatically, assigning 'Alice' to name, 18 to age, and 'CLS' to school for Stu1.

OOP_eg1.py - /home/pi/OOP_eg1.py (3.5.3)

File Edit Format Run Options Window Help

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

Note how arguments are passed in to initialise the attributes.

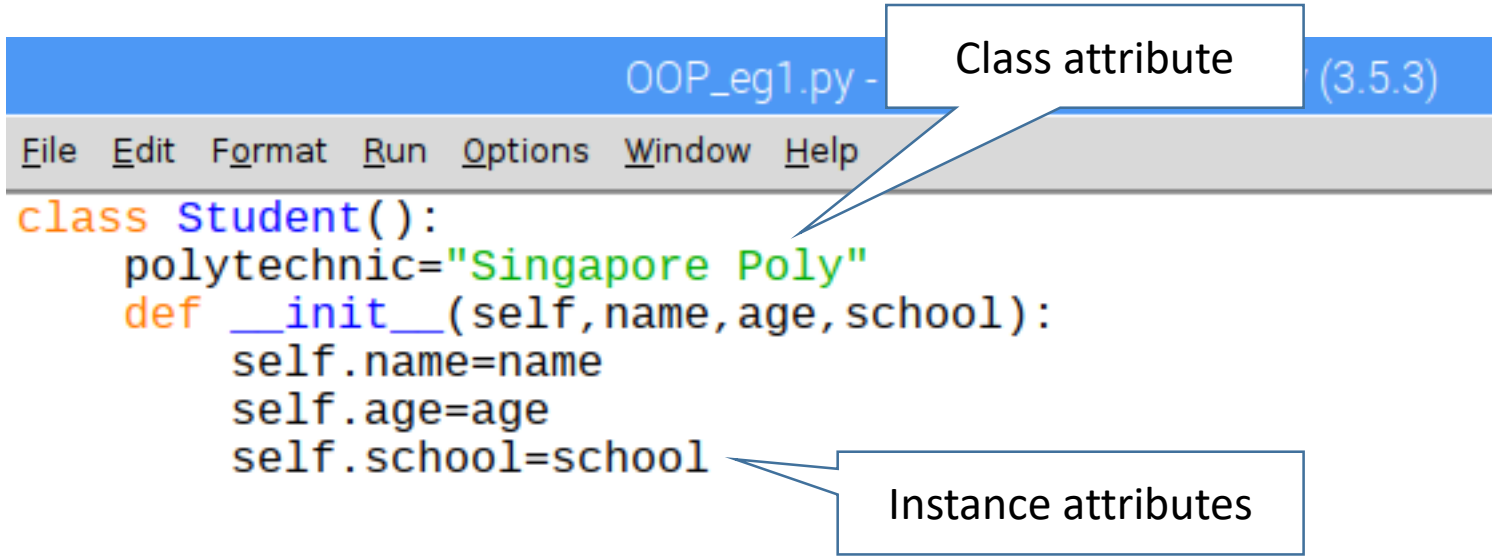
```
Stu1=Student('Alice',18,'CLS')
Stu2=Student('Melvyn',19,'EEE')
Stu3=Student('John',18,'ABE')

print("{} from {} is {} years old.".format(Stu1.name,Stu1.school,Stu1.age))
```

What do you think will be printed out by this line?

Dot notation is used to access attributes.

Class attribute vs. instance attribute

- While **instance attributes** (such as name, age and school) can be unique for each instance (i.e. each student), **class attribute** (such as polytechnic) is common to all student i.e. all students are from Singapore Poly.



Methods

- **Methods** defined inside a class can be used to **get** / **modify** the attributes of an instance.

OOP_eg1.py - /home/pi/OOP_eg1.py

File Edit Format Run Options Window Help

```
class Student():
    polytechnic="Singapore Poly"
    def __init__(self,name,age,school):
        self.name=name
        self.age=age
        self.school=school

    def description(self):
        return "{} from {} is {} years old.".format(self.name,self.school,self.age)

    def change_school(self,new_school):
        self.school=new_school
```

This illustrates "Encapsulation", a key concept in OOP: the attributes & methods associated with an object are encapsulated in the class.

Get the attributes

Modify the attribute

Methods (cont.)

- When this program is run, what do you expect to see?

OOP_eg1.py - /home/pi/OOP_eg1.py (3.5.3)

File Edit Format Run Options Window Help

```
class Student():
    polytechnic="Singapore Poly"
    def __init__(self,name,age,school):
        self.name=name
        self.age=age
        self.school=school

    def description(self):
        return "{} from {} is {} years old.".format(self.name,self.school,self.age)

    def change_school(self,new_school):
        self.school=new_school

Stu1=Student('Alice',18,'CLS')
print(Stu1.description())
print(Stu1.name,'is changing school.')
Stu1.change_school('EEE')
print('The new school is:',Stu1.school)
```

```
Alice from CLS is 18 years old.
Alice is changing school.
The new school is: EEE
```

Print out the student's name, school and age.

Change the student's school.

Inheritance

- **Inheritance** is the process by which one class takes on the **attributes** and **methods** of another.
- Newly formed classes are called **child classes**, and the classes that child classes are derived from are called **parent classes**.
- A child class inherits all of its parent class' attributes and methods or behaviours, but can also specify **additional / different** attributes and / or behaviours.

"Inheritance" is another key concept in OOP: a child class inherits the attributes & methods of the parent class.

Inheritance (cont.)

- Let's say for the group of students taking an elective is split into 2 sub-groups, one taught by Mr Chong on Monday morning, and another taught by Mr Joe on Tuesday afternoon.
- To differentiate between these 2 sub-groups, two child classes can be created as follows, with an additional attribute called lesson_time:

OOP_eg1.py - /home/pi/OOP_eg1.py (3.5.3)

File Edit Format Run Options Window Help

```
class Student():
    polytechnic="Singapore Poly"
    def __init__(self,name,age,school):
        self.name=name
        self.age=age
        self.school=school

    def description(self):
        return "{} from {} is {} years old.".format(self.name,self.school,self.age)

    def change_school(self,new_school):
        self.school=new_school

class MrChongs_Student(Student):
    lesson_time="Monday morning"

class MrJoes_Student(Student):
    lesson_time="Tueday afternoon"
    def description(self):
        return "{} from Mr Joe's class is {} years old.".format(self.name,self.age)
```

Parent class.

Child class.

Additional attribute.

Different behaviour.

Inheritance (cont.)

- Let's see how a child class inherits its parent class' attributes and behaviours, and how additional attribute & different method can be possible:

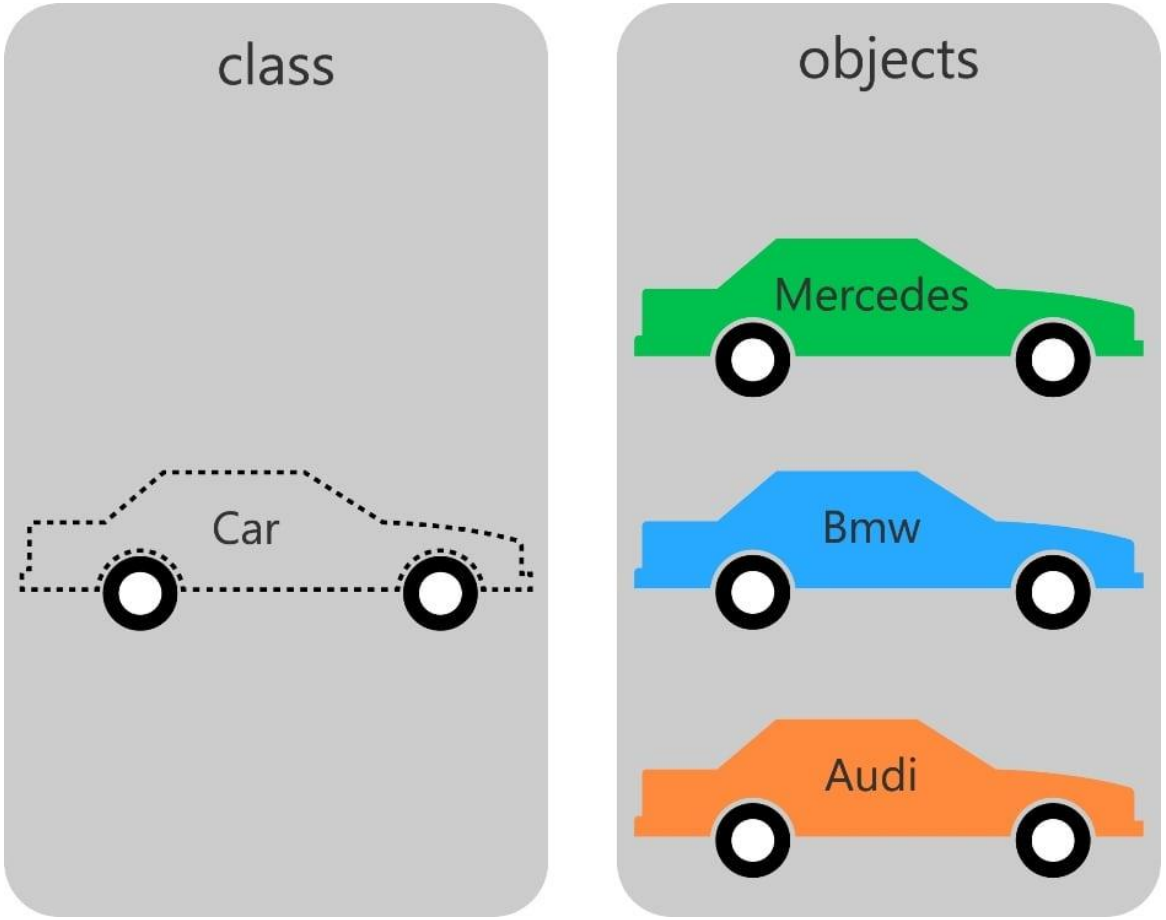
OOP_eg1.py - /home/pi/

File Edit Format Run Options Window Help

```
class Student():
    polytechnic="Singapore Poly"
    def __init__(self,name,age,school):
        self.name=name
        self.age=age
        self.school=school

    def description(self): #a method to get attributes
        return "{} from {} is {} years old.".format(self.name,self.school,self.age)

    def change_school(self,new_school):
        self.school=new_school

class MrChongs_Student(Student):
    lesson_time="Monday morning"

class MrJoes_Student(Student):
    lesson_time="Tueday afternoon"
    def description(self):
        return "{} from Mr Joe's class is {} years old.".format(self.name,self.age)

Stu4=MrChongs_Student('Lisa',17,'SB')
print(Stu4.description())
print('The lesson time is:',Stu4.lesson_time)

Stu5=MrJoes_Student('Ahmad',18,'SMA')
print(Stu5.description())
print('The lesson time is:',Stu5.lesson_time)
```

```
Lisa from SB is 17 years old.
The lesson time is: Monday morning
Ahmad from Mr Joe's class is 18 years old.
The lesson time is: Tueday afternoon
```

Summary…



- This brief lesson introduces you to some OOP concepts, such as class, instance, attributes, methods, **encapsulation**, **inheritance**.
- Other than encapsulation and inheritance, **abstraction** and **polymorphism** are frequently mentioned, when people talk about OOP.
- This elective does not require you to stick to OOP programming paradigm, but one day, you may need to use a module / library to get something done, and OOP concepts will come in handy.