Object Oriented Programming - Revisit (part 1)

Scope:

- Class and Instance
- Instnace Methods
- The `self` Object
- Instance Variables & Class Variables
- Class Methods and Static Methods

1. Introduction to OOP

Procedural Programming

We used to group blocks of statements together into `functions`. We call these functions in sequential order. Such way of structuring a program is called **Procedural Programming**.

Generally states of the program is defined outside of the function as variables.

Such programming paradigm works well for small programs, which is easy to understand and maintain.

In [1]:

```
def rectangle_area(width, length):
    return width * length

w = 10
l = 20
area  = rectangle_area(w, l)
print(area)
```

```
200
```

Another Example

We would like to keep track of a list of Students and lessons taken by respective student.

We can use list to keep multiple properties of each student. The code is simple but not very readable. And the code will become more ugly when as the number of students grow.

In [2]:

```
s1 = ['Alan', []]
s2 = ['Bob', ['Maths']]
s3 = ['Chalie', ['Maths', 'Physics']]

# To add lesson to a student
s1[1].append('Maths')
```

Object Oriented Programming

Object Oriented Programming is another paradigm which bundles properties and behaviors together. It models real-world entities as software object.

In [3]:

```
class Rectangle:

    def __init__(self,width, length):
        self.width = width
        self.length = length

    def calc_area(self):
        return self.width * self.length

r = Rectangle(10, 20)
print(r.calc_area())
```

```
200
```

The example on students and lessons can be rewriten as following:

In [4]:

```
class Student:

    def __init__(self, name, lessons=None):
        self.name = name
        self.lessons = [] if lessons is None else lessons

    def attend(self, lesson):
        self.lessons.append(lesson)

s1 = Student('Alan')
s2 = Student('Bob', ['Maths'])
s3 = Student('Chalie', ['Maths', 'Physics'])

s1.attend('Maths')
```

2. Class and Instance

Classes

A class is a blueprint for the object.

- It contains all the details about such a object, e.g. properties and behaviors.
- To define a class, `class` keyword is used.

The simplest form of Class is a class without any property or method.

In [5]:

```
class Person:
    pass
```

Instances

A class creates a new type where objects are instances of the class.

Recall that an integer can be created from its constructor function.

In [6]:

```
a = int(123)
print(a, type(a))
```

```
123 <class 'int'>
```

Rectangle objects can also be constructed in similar way.

In [7]:

```
p1 = Person()
p2 = Person()
print(id(p1), id(p2), p1 == p2)
```

```
2367047852944 2367047852888 False
```

Methods

Methods are functions which are added to a class to define its behaviors.

In [8]:

```
class Person:

    def say_hi(self):
        print('Hi')


p = Person()
p.say_hi()
```

```
Hi
```

The `self`

The `self` parameter is a variable refers to the object itself. By convention, it is always given the name `self`.

- Parameter `self` must be the 1st parameter.
- Using `self`, you can access other attributes in the object.

Passing Arguments to Method

- When you call the method, you do NOT need to give a value for `self` parameter. Python provides it automatically.

In [9]:

```
class Person:

    def say_hi(self):
        print('Hi')

    def say_hi_twice(self):
        self.say_hi()
        self.say_hi()

p = Person()
p.say_hi_twice()
```

```
Hi
Hi
```

The `__init__()` Method

Remember that attributes can be dynamically added to Python object.

- Attributes added to one instance does not exists in another instance.

In [10]:

```
p1 = Person()
p1.name = 'Alan'
p1.age = 18

print(p1.name, p1.age)
```

```
Alan 18
```

Python class can implement a `__init__()` method to initialize a common set of variables for all instances.

- The `__init__()` takes in a `self` parameter too.

In [11]:

```python
class Person:

    def __init__(self, name, age):
        self.name = name
        self.age = age

p1 = Person('Alan', 18)
p2 = Person('Bob', 18)

print(p1.name, p1.age)
print(p2.name, p2.age)
```

```
Alan 18
Bob 18
```

Instance Variables

Instance variables are owned by each individual object/instance of the class. In this case, each object has its own copy of the fields.

- Instance variables are not shared among instances although they have same name.

In above exmaple, the `name` and `age` variables contains different value in `p1` and `p2` .

Class Variable

Class variables are shared by all instances of that class.

- There is only one copy of the class variable
- When any one object makes a change to a class variable, all the other instances all see the change

In [12]:

```python
class Person:

    population = 0

    def __init__(self):
        Person.population += 1

    def die(self):
        Person.population -= 1

p1 = Person()
p2 = Person()
print(Person.population)
p1.die()
print(Person.population)
```

```
2
1
```

Class Variable vs Instance Variable (confusing)

Class variables and instance variables belong to different namespaces, i.e. they are different variable objects.

In [13]:

```
class Car:

    name = 'SUV'

    def __init__(self):
        self.name  = 'my suv'

car = Car()
print(Car.name, car.name)
print(id(Car.name) == id(car.name))
```

```
SUV my suv
False
```

A class variable can be access by class or by its instance, provided there is no instance variable of same name.

In [14]:

```
class Car:

    name = 'SUV'

    def __init__(self):
        pass


car = Car()
print(Car.name, car.name)
print(id(car.name) == id(Car.name))
```

```
SUV SUV
True
```

When a value is assigned to a non-existence instance variable, such instance variable will be created immediately.

In [15]:

```
print(Car.name, car.name)
car.name = 'little'
print(Car.name, car.name)
print(id(car.name) == id(Car.name))
```

```
SUV SUV
SUV little
False
```

3. Class Methods and Static Methods

Instance Methods

We have seen instance methods, they are the methods with 1st parameter `self` , which will be passed in by Python automatically.

Example: BMI Calculator

Considering following class `BMI` which is used to calculate BMI value.

- It initializes its `height` and `weight` values in `__init__()` function.
- Its `get_bmi()` function returns BMI value calculated from `height` and `weight` .

In this example, `get_bmi()` must be an instance method because it needs to access to `self` object.

- Instance methods are accessed from instance object.

In [16]:

```
class BMI:

    def __init__(self, weight_kg, height_m):
        self.weight = weight_kg
        self.height = height_m

    def get_bmi(self):
        bmi = self.weight / (self.height**2)
        return bmi
```

In [17]:

```
bmi = BMI(70, 1.7)
bmi.get_bmi()
```

Out[17]:

```
24.221453287197235
```

Static Methods

BMI Calculator Version 2

If we modify the `get_bmi()` method to take in `weight` and `height` as parameters, then the function doesn't need to access to `self` object anymore.

In fact, it doesn't need to use any other data from class `BMI` . Such a method is called **static method**.

In [18]:

```
class BMI:

    @staticmethod
    def get_bmi(weight_kg, height_m):
        bmi = weight_kg / (height_m**2)
        return bmi

BMI.get_bmi(70,1.7)
```

Out[18]:

```
24.221453287197235
```

Class Methods

We would like to add another method `analyze()` , which take a BMI value as input and give simple analysis.

We defined 2 class variables, `LOWER` and `UPPER` , representing lower bound and upper bound of healthy BMI value.

For `analyze()` method to access to the 2 class variables, it needs to be defined as **class method**.

- A class method has `class` object as its 1st parameter, which is passed in automatically by Python.

In [19]:

```
class BMI:

    LOWER = 18.5
    UPPER = 24.9

    @classmethod
    def analyze(cls, bmi):
        if (bmi > cls.UPPER):
            return 'over-weight'
        elif (bmi < cls.LOWER):
            return 'under-weight'
        else:
            return 'perfect'

BMI.analyze(25)
```

Out[19]:

```
'over-weight'
```