1. What is the difference between class variable and instance variable? Explain the difference with the example.

Class variable: It is basically a static variable that can be declared anywhere at class level with static. Across different objects, these variables can have only one value. These variables are not tied to any particular object of the class, therefore, can share across all objects of the class.

Instance Variable: It is basically a class variable without a static modifier and is usually shared by all class instances. Across different objects, these variables can have different values. They are tied to a particular object instance of the class, therefore, the contents of an instance variable are totally independent of one object instance to others.

In [1]:
```
class course():
    grade = 'A+'
    def __init__(self, course_name, course_code):
        self.course_name = course_name
        self.course_code = course_code
    def show(self, score):
        print(f'Course Name: {self.course_name}')
        print(f'Course Code: {self.course_code}')
        print(f'Need to get an {self.grade} in this course!')
        print(f'But I got {score} in this course!')
python = course('Python', '542')
python.show(80)
```

```
Course Name: Python
Course Code: 542
Need to get an A+ in this course!
But I got 80 in this course!
```

Here grade is a class variable and course_name & course_code is instance variable

2. What is the significance of the first parameter to each method within the class? Explain with an example.

In the previous example, both of these method definitions (__init **& show) have** "self"__ as the first parameter, and we use this variable inside the method bodies, but we do not appear to pass this parameter in. This is because whenever we call a method on an object, the object itself is automatically passed in as the first parameter. This gives us a way to access the object's properties from inside the object's methods.

In some languages this parameter is implicit – that is, it is not visible in the function signature – and we access it with a special keyword. In Python it is explicitly exposed. It doesn't have to be called self, but this is a very strongly followed convention.

We will want to define our instance methods in a way that anticipates that Python will automatically pass an instance object as the first argument. Thus if we want our method to accept external argument, we should define its signature to have arguments, with the understanding that Python will pass the instance object as the first argument. The accepted convention is to call this first argument self. There is no significance to this name beyond it being the widely-adopted convention among Python users; "self" is meant to indicate that the instance object is passing itself as the first argument of the method.

Below is another example:

In [2]:
```
# demonstrate the use of `self` in instance arguments
class Number:
    def __init__(self, value):
        self.value = value

    def add(self, new_value):
        return self.value + new_value
x = Number(4.0)
x.add(2.0)
```

Out[2]:
```
6.0
```

3. Define Encapsulation, Inheritance and Polymorphism.

Encapsulation: Encapsulation is one of the fundamental concepts in OOP. It describes the idea of restricting access to methods and attributes in a class. This will hide the complex details from the users, and prevent data being modified by accident. In Python, this is achieved by using private methods or attributes using double underscore as prefix, i.e. double "__".

Inheritance: Inheritance allows us to define a class that inherits all the methods and attributes from another class. Convention denotes the new class as child class, and the one that it inherits from is called parent class or superclass. If we refer back to the definition of class structure, we can see the structure for basic inheritance is class ClassName(superclass), which means the new class can access all the attributes and methods from the superclass. Inheritance builds a relationship between the child class and parent class, usually in a way that the parent class is a general type while the child class is a specific type.

Polymorphism: Polymorphism is another fundamental concept in OOP, which means multiple forms. Polymorphism allows us to use a single interface with different underlying forms such as data types or classes. For example, we can have commonly named methods across classes or child classes. These methods might have the same name, but they have different implementation. This ability of using single name with many forms acting differently in different situations greatly reduces our complexities.

4. Define a class called Parent. Initialize the class with name and age. Define three methods — set_name, set_age, print_parent which will set the name, age and print parent's name and age respectively. Instantiate the class. Update name/age and print the details again.

In [3]:
```
class Parent():
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def set_name(self, name):
        self.name = name
    def set_age(self, age):
        self.age = age
    def print_parent(self):
        print(f'Name - {self.name}')
        print(f'Age - {self.age}')
```

In [4]:
```
parent1 = Parent("John", "32")
parent1.print_parent()
```

```
Name - John
Age - 32
```

In [5]:
```
# Updating name and age
parent1.set_name("John Wick")
parent1.set_age(33)
parent1.print_parent()
```

```
Name - John Wick
Age - 33
```

5. What is the output of the below program? Explain the output in less than 2 sentences.

In [6]:
```
class Demo:
    def __init__(self):
        self.x = 1
    def change(self):
        self.x = 10

class Derived(Demo):
    def change(self):
        self.x = self.x+1
        return self.x

obj = Derived()
print(obj.change())
```

```
2
```

As obj is an instance of **Derived** class, **change()** method associated to the **Derived** class will trigger in this case according to the concept of polymorphism. Due to **Demo** being the superclass of **Derived**, obj gets the value of x = 1 and the **change()** method will increase it by 1, thus the output became 2.