# Object Oriented Programming

**Objectives:**

- Define and understand classes and objects.
- Understand encapsulation and how classes support information hiding and implementation independence.
- Understand inheritance and how it promotes software reuse.
- Understand polymorphism and how it enables code generalisation.
- Exclude: method overloading and multiple inheritance

## 1. Class Basics

Classes are blueprints/template for objects. They define the **structure** and **behavior** of objects.

- Python is highly object-oriented.
- But it does not force you to use it until you need to do so.

Creating a new object is called `instantiation` . An **object** of a class is also called an **instance** of that class.

- Multiple objects can be created from same class.

### Everything is an Object

In Python, everything is an object.

- This includes classes (types).
- The `id()` method can be used to get unique ID of an object.

**Question:**

What is the ID of the `str` class, and ID of a str object `a = 'abc'` ?

In [1]:

```python
s = 'abc'
print(id(s))
print(id(type(s)))
print(id(str))
```

```
3046744108272
140722196434160
140722196434160
```

### Class Definition

Classes are defined using the `class` keyword followed by CamelCase name.

- Class instances are created by calling the class as if it is a function.

In [24]:

```
class Vehicle:
    pass

v = Vehicle()
isinstance(v, Vehicle)
```

Out[24]:

```
True
```

When you print an instance, Python shows its class and its memory location.

In [25]:

```
print(Vehicle)
print(v)
```

```
<class '__main__.Vehicle'>
<__main__.Vehicle object at 0x000001AEC7D15B38>
```

## Initializer Method __init__()

Python class has an initializer method, `__init__()` , which will be automatically called to initialize the newly created object.

- `__init__()` is a **dunder** method which generally are used by Python compiler.
- Its definition is similar to function definition except that its first argument is `self` .
- It can take in additional arguments.

#### Instance Attributes

Its common to initialize **Instance Attributes** in the initializer method `__init__()` .

#### Keyword `self`

To access any instance method or instance attribute in the class, you need to prefix it with `self.` .

In [36]:

```
class Vehicle:
    def __init__(self, horsepower, color='black', wheels=4):
        self.horsepower = horsepower
        self.color = color
        self.wheels = wheels

v = Vehicle(100)
v.wheels = 6
print(v.horsepower, v.color, v.wheels)
```

```
100 black 6
```

## Instance Methods

**Methods** are functions defined within a class. **Instance Methods** are functions can be called on objects.

- It defines the **behavior** of objects of the class.
- Methods are called using `instance.method()` .

**Argument `self`**

- The `self` attribute must be the first input parameter for all instance methods.
- The `self` attribute is refer to current object of the class, i.e. the instance calling the method.
  - This is similar to the `this` in C# or Java.
- When a instance method is called, `self` argument is omitted.

In [4]:

```
class Vehicle:
    def __init__(self, horsepower, color='black'):
        self.horsepower = horsepower
        self.color = color

    def engine_power_kw(self):
        return self.horsepower * 0.745699872

v = Vehicle(100, 'Blue')
'{} hp = {} kw'.format(v.horsepower, v.engine_power_kw())
```

Out[4]:

```
'100 hp = 74.5699872 kw'
```

## Implement `__str__()` for Custom Object

By default, our `Vehicle` class inherits `__str__()` method from `Object` class, which print class name and memory location of the object.

In [5]:

```
class Vehicle:
    def __init__(self, plate):
        self.plate = plate

v1 = Vehicle('A1234')

print(str(v1))
```

```
<__main__.Vehicle object at 0x000002662B938EF0>
```

**Exercise:**

For our `Vehicle` class to support `str()` method, we can implement `__str__()` method in the class.

In [6]:

```
class Vehicle:
    def __init__(self, plate):
        self.plate = plate

    def __str__(self):
        return 'Vehicle: {}'.format(self.plate)

v = Vehicle('A1234')
print(str(v))
```

```
Vehicle: A1234
```

# 2. Class Attributes, Static Methods and Class Methods (Optional)

## Class Attributes

Class Attributes are attributes which belong to class instead of a particular object.

- It can be accessed through either class or instance.

We can use class attributes to keep a rolling value which is shared among all instances. For example, we would like to keep track of number of Customers and assign each customer a unique serial number.

In [7]:

```
class Customer:

    next_serial = 1

    def __init__(self):
        self.serial = Customer.next_serial
        Customer.next_serial += 1

## Test
s1 = Customer()
s2 = Customer()
print(s1.serial)
print(s2.serial)
print(Customer.next_serial, s1.next_serial, s2.next_serial)
```

```
1
2
3 3 3
```

## Static Methods

In Python, all instance methods have `self` as their first argument.

Static methods in Python are similar to instance methods, the difference being that a static method is bound to a class rather than the objects for that class.

- A static method is a method which does not has `self` as its first argument.
- It can be called without an object of that class.
- This also means that static methods cannot modify the state of an object as they are not bound to it.

Static method are declared using `@staticmethod` decorator.

- The `@staticmethod` decorator is optional. But static method without `@staticmethod` decorator cannot be called from its instance.

In [ ]:

```
class Calculator:
    @staticmethod
    def add(x, y):
        return x + y

Calculator.add(1,2)
```

### Class Methods

Class methods are much like **static method**. They are methods that are bound to a class rather than its object.

The difference between a static method and a class method is:

- Static method knows nothing about the class and just deals with the parameters.
- Class method works with the class since its parameter is always the class itself.

To create a class method, use `@classmethod` decorator.

In [ ]:

```
class Converter:

    PI = 3.1415926

    @classmethod
    def rad_to_degree(cls, r):
        d = r/cls.PI*180
        return d

Converter.rad_to_degree(3.1415926)
```

## 3. Inheritance (Optional)

Similar to other programming languages, Python allows class inheritance.

In following code sample, both class `B` and `C` inherit from class `A`.

- The special attribute `__base__` returns its 1st base class. To get all base classes, use attribute `__bases__` .
- We can test whether a class is subclass of one or more classes using `issubclass()` method.

In [9]:

```
class A:
    pass

class B(A):
    pass

print(A.__base__)
print(B.__base__)
print(issubclass(B,A))
```

```
<class 'object'>
<class '__main__.A'>
True
```

## Method Overriding

A subclass may override a method defined in its superclass.

**Example:**

- Class B doesnot override `hi()` method in class A
- Class C overrides `hi()` method in class B

In [28]:

```
class A:
    def hi(self):
        print('hi A')

class B(A):
    pass

class C(A):
    def hi(self):
        print('hi C')

b = B()
b.hi()
c = C()
c.hi()
```

```
hi A
hi C
```

## Super Function - super()

With inheritance, the super() function allows us to call a method from the parent class.

In [15]:

```python
class A:

    def hi(self):
        print('hi A' )

class C(A):

    def hi(self):
        super().hi()
        print('hi C')

c = C()
c.hi()
```

```
hi A
hi C
```

## Method Overloading - NOT AVAILABLE

Python doesnot support method overloading. It keeps only the latest definition of the method.

In [16]:

```python
def add(a,b):
    return a+b

def add(a,b,c):
    return a+b+c

add(1,2,3)

## Raise a TypeError
# add(2,3)
```

Out[16]:

```
6
```