Section A3 Attributes and Methods

- **Class:** Classes are used to create user-defined data structures. A class is a blueprint for how something should be defined. It doesn't actually contain any data. In Python, we use the keyword ***class*** to create a class. The general class definition looks as follows.

```
class ClassName:
        '''Docstring - An optional documentation string used to describe the class'''
        statement_1
        statement_2
        ...
        statement_n
```

  - The docstring of a class is accessed using **ClassName.__doc__**
- **Object:** An object is a variable of class type. An object of a class is also known as an instance of a class. All the members of a class are accessed using an object of that class. The following syntax is used to create an object.

```
object_name = ClassName(arguments)
```

- **Self:** The self parameter is used to easily access all the instances defined within a class, including its methods and attributes. Every method of a class must have the first parameter as self. What self parameter does is ask to refer to the current object being used to call that method.
  - It does not have to be named self, we can call it whatever we like, for example this, but it has to be the first parameter of any function in the class.
  - However, it is highly recommended to stick with the usage of self.

Introduction to the Python __init__() Method
The attributes of the classes are placed inside a special method called **__init__()** and all methods are declared within the class level.

When you create a new object of a class, Python automatically calls the __init__() method to initialise the attributes of the object. The double underscores(i.e., __) at both sides indicate that Python will use this method internally. Therefore, you do not need to explicitly call this method, since Python is automatically calling the __init__() method immediately after you create a new object. You can use the __init__() method to initialise the object's attributes.

The __init__() method is called the **constructor** since the task of constructors is to initialise (assign values) when an object of the class is being created. There are two types of constructors.

- **Default constructor:** This is a simple constructor which does not accept any arguments. Its definition has only one argument which is

the self parameter, denoting a reference to the instance being created.
- **Parameterised constructor:** This is a constructor with parameters. The parameterised constructor takes its first argument the self parameter, and the remaining arguments (e.g., *name*, *position*) are provided by you. The "self" parameter should always come first in the function parameters.

Class Variables/Attributes vs Instance Variables/Attributes
- class variables are shared by all instances of the class, they will generally have the same value for every instance
- Class variables are defined outside of all the methods. Usually, they are placed right below the class header and before the constructor method and other methods.
- Instance variables are defined within methods.

Revisiting Accessing and Modifying Attributes

the constructor must declare all the attributes of the object and initialise their values within the body, which means attributes of a class must always appear within the constructor.

```
class Customer:
    #initialization function --- it is very important ---
    def __init__(self,name,account_no):
        #the word 'self' refers to the specific instance of the class
        self.name = name
        self.account_no = account_no
        self.balance = 0
        self.debit_card_no = None

    def withdraw_money(self,amount):
        self.balance -= amount
        return self.balance

    def deposit_money(self,amount):
        self.balance += amount
        return self.balance
```

Access Modifiers
there are certain scenarios where need to hide the attributes and/or methods from being accessible outside. An example of this is the balance attribute in our fictional **Bank of Adelaide** Customer class. Although we need the instance variable balance to be visible and modifiable within the class definition, we shouldn't allow access to balance through the object. In such cases, we declare the attributes to be 'private'. Private attributes can only be accessed from within the scope of the class.

```python
class Customer:
    customer_counter = 0

    def __init__(self, name):
        #the word 'self' refers to the specific instance of the class
        self.name = name
        self.account_no = Customer.customer_counter+1
        Customer.customer_counter += 1
        self.__balance = 0
        self.debit_card_no = None

    def withdraw_money(self, amount):
        self.__balance -= amount
        return self.__balance

    def deposit_money(self, amount):
        self.__balance += amount
        return self.__balance

    def get_balance(self):
        return self.__balance

    def set_balance(self, amount):
        if amount > 0:
            self.__balance = amount
```

```python
customer_1 = Customer('John Doe')
customer_2 = Customer('Jane Doe')

customer_1.deposit_money(1000)
customer_2.deposit_money(2000)

print(f'The customer {customer_1.name} has a balance of {customer_1.get_balance()}$')
```

```
The customer John Doe has a balance of 1000$
```

here are three types of access modifiers for a class in Python. They are Public, Private, and Protected.

Magic Methods (Dunder Methods)

Magic methods fall into several categories, as summarised below.

- Construction and Initialisation (e.g., __init__)
- Comparison (e.g., __lt__,__eq__)
- Arithmetic (e.g., __add__,__sub__)
- Representation (e.g., __repr__, __str__)
- Controlling attribute access (e.g., __getattr__)

Let us look at the use of the representation magic method __str__ to format objects for printing.

```
class Customer:
    customer_counter=0
    def __init__(self, name):

        self.name=name
        self.account_no=Customer.customer_counter+1
        Customer.customer_counter+=1
        self.__balance=0
        self.debit_card_no=None

    def __str__(self):
        return(f'The customer {self.name} with the account number {self.account_no} has a balance of {self.__balance}')

    def withdraw_money(self,amount):
        self.__balance-=amount
        return self.__balance

    def deposit_money(self,amount):
        self.__balance+=amount
        return self.__balance

    def get_balance(self):
        return self.__balance

    def set_balance(self,amount):
        if amount>0:
            self.__balance=amount

customer_1 = Customer('John Doe')
print(str(customer_1))
```

```
The customer John Doe with account number 1 has a balance of 1000
```

The above example illustrates how we could use the __str__ magic function to define the string representation for all objects of the Customer class. Similarly, by defining the behaviour for comparison, we could allow our objects to be sorted by using one of the comparison magic methods (usually __lt__, __gt__, and __eq__ are used frequently). Let's look at examples of __lt__ and __gt__.

```
class Weight:
    def __init__(self, weight):
        self. weight= weight

    def __lt__(self,other):
        return self.weight<other.weight

    def __gt__(self,other):
        return self.weight>other.weight

a = Weight(50)
b = Weight(60)
c = Weight(70)

print(a<b and b>c)
```

```
False
```