CLASSES AND OBJECTS

Python is an object-oriented programming language. Unlike procedure-oriented programming, where the main emphasis is on functions, object-oriented programming stresses on objects.

An object is any real time entity that has a **state (Data/Field)** and **a behavior (Functions/Operations**). Classes describe the state and behavior of the objects. In a nutshell, a class is a blueprint for that object.

Some points on Python class:

- Classes are created by keyword class.
- Attributes are the variables that belong to a class.
- Attributes are always public and can be accessed using the dot (.) operator. E.g.: Myclass.Myattribute

DEFINING CLASSES

CLASS DEFINITION SYNTAX:

```
class ClassName:
    # Statement-1
    .
    .
    .
    # Statement-N
```

Like function definitions begin with the **def** keyword in Python, class definitions begin with a **class** keyword. The first string inside the class is called docstring and has a brief description about the class. Although not mandatory, this is highly recommended. The statement in the definition can be any of these-sequential instructions, decision control statements, loop statements and can even include function definitions. Variables defined in the class are called **class variables** and functions defined in the class are called **class methods**. Together

these two are known as **class members.** These class members can be accessed only by the class objects

A class creates a new **local** namespace where all its attributes(data and functions) are defined. As soon as we define a class, a new class object is created with the same name. This class object allows us to access the different attributes as well as to instantiate new objects of that class.

EXAMPLE:

```
class Person:
    "This is a person class"
    age = 10

    def greet(self):
        print('Hello')

print(Person.age)

print(Person.greet)

print(Person.__doc__)
```

```
10
<function Person.greet at 0x0000023CC1CD0430>
This is a person class
```

CREATING OBJECTS

The object can access the class variables and class methods using the dot operator (.). The syntax to create a class object is

object_name= class_name( )

We can think of class as a sketch (prototype) of a house. It contains all the details about the floors, doors, windows etc. Based on these descriptions we build the house. House is the object. As many houses can be made from a house's blueprint, we can create many objects from a class. An object is also called an instance of a class and the process of creating this object is called **instantiation**.

The syntax for accessing a class member through the class object is

object_name.class_member_name

EXAMPLE:

```
class Person:
    "This is a person class"
    age = 10

    def greet(self):
        print('Hello')


# create a new object of Person class
harry = Person()

# Output: <function Person.greet>
print(Person.greet)

# Output: <bound method Person.greet of <__main__.Person object>>
print(harry.greet)

# Calling object's greet() method
# Output: Hello
harry.greet()
print(harry.age)
```

```
<function Person.greet at 0x0000023CC1CD0AF0>
<bound method Person.greet of <__main__.Person object at 0x0000023CC1CDA7C0>>
Hello
10
```

self KEYWORD

The **self** in Python works in the same way as the **"this"** pointer in C++. Whenever an object calls its method, the object itself is passed as the first argument. So, harry.greet() translates into Person.greet(harry).

Class methods must have an extra first parameter in the method definition. We do not give a value for this parameter when we call the method, Python provides it implicitly.

If we have a method that takes no arguments, then we still have to have one argument.
When we call a method of an object as myobject.method(arg1, arg2), this is automatically converted by Python into MyClass.method(myobject, arg1, arg2) . Since the class methods uses **self**, they require an object or instance of a class to be used. For this reason, they are termed as ***instance methods***.

CONSTRUCTORS IN PYTHON

Class functions that begin with double underscore `__` are called special functions as they have special meaning.
Of one particular interest is the **`__init__()`** function. This special function gets called whenever a new object of that class is instantiated.
This type of function is also called constructors in Object Oriented Programming (OOP). We normally use it to initialize all the variables.

EXAMPLE:

In the below example, we defined a new class to represent complex numbers. It has two functions, `__init__()` to initialize the variables (defaults to zero) and `get_data()` to display the number properly.

```
class ComplexNumber:
    def __init__(self, r=0, i=0):
        self.real = r
        self.imag = i

    def get_data(self):
        print(f'{self.real}+{self.imag}j')


num1 = ComplexNumber(2, 3)
num1.get_data()

num2 = ComplexNumber(5)
#Creating a new attribute on the fly
num2.attr = 10

print((num2.real, num2.imag, num2.attr))

# but c1 object doesn't have attribute 'attr'
# AttributeError: 'ComplexNumber' object has no attribute 'attr'
print(num1.attr)
```

```
2+3j
(5, 0, 10)

---------------------------------------------------------------------------
AttributeError                            Traceback (most recent
```

An interesting thing to note in the above step is that attributes of an object can be created on the fly. We created a new attribute `attr` for object `num2` and read it as well. But this does not create that attribute for object `num1`.

CLASS VARIABLES AND OBJECT VARIABLES

The variables defined inside the class are of two types- class variables and object variables. Class variables are owned by the class and object variables are owned by each object.

So, if a class has n objects then there will be n separate copies of the object variable as each object will have its own object variable. The object variable will not be shared between the objects. So, a change made to the object variable by one object will not be reflected in other objects.

However, if a class has one class variable, then there will be only one copy of that variable. All the objects of the class will share the same variable. Since there exists a single copy of the class variable, any change made to the class variable by an object will be reflected in all objects.

Class variables are usually used to keep a count of number of objects created from a class. Another important use is to define constants associated with a particular class or provide default attribute values.

Class variables are accessed using the class name followed by the dot operator.

```
class Employee:
   'Common base class for all employees'
   empCount = 0 #Class variable

   def __init__(self, name, salary):
      self.name = name #object variable
      self.salary = salary #object variable
      Employee.empCount += 1

   def displayCount(self):
     print(f"Total Employee{Employee.empCount}")

   def displayEmployee(self):
      print("Name : ", self.name,  ", Salary: ", self.salary)

emp1 = Employee("Zara", 2000)
emp2 = Employee("Rachel", 1000)
print (Employee.__doc__)
emp1.displayEmployee()
emp2.displayEmployee()
print (f"Total Employee:{Employee.empCount}")
```

```
Common base class for all employees
Name :  Zara , Salary:  2000
Name :  Rachel , Salary:  1000
Total Employee:2
```

THE __del__() METHOD

As __init__( ) initializes the object of the class at the time of creation, we have the __del__( ) which does the opposite work. IT is called automatically called

when an object goes out of scope. We can also explicitly delete the object and clean up the resources occupied by that object using he **del** keyword.

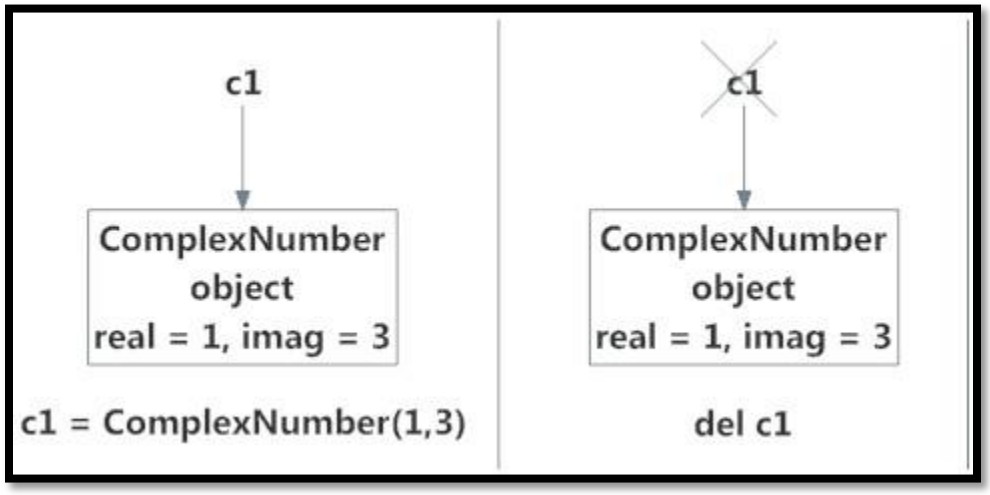
DELETE OBJECT PROPERTIES

You can delete properties on objects by using the `del` keyword: **del e1.salary**

DELETE OBJECTS

Delete the e1 object: **del e1**

Actually, it is more complicated than that. When we do `c1 = ComplexNumber(1,3)`, a new instance object is created in memory and the name *c1* binds with it.
On the command `del c1`, this binding is removed and the name *c1* is deleted from the corresponding namespace. The object however continues to exist in memory and if no other name is bound to it, it is later automatically destroyed.
This automatic destruction of unreferenced objects in Python is also called garbage collection.



Deleting objects in Python removes the name binding

GARBAGE COLLECTION (DESTROYING OBJECTS)

Python deletes unneeded objects (built-in types or class instances) automatically to free the memory space. The process by which Python

periodically reclaims blocks of memory that no longer are in use is termed Garbage Collection.

Python's garbage collector runs during program execution and is triggered when an object's reference count reaches zero. An object's reference count changes as the number of aliases that point to it changes.

An object's reference count increases when it is **assigned a new name** or **placed in a container** (list, tuple, or dictionary). The object's reference count decreases when it's **deleted with *del*, its reference is reassigned**, or its **reference goes out of scope.** When an object's reference count reaches zero, Python collects it automatically.

```
a = 40      # Create object <40>
b = a       # Increase ref. count  of <40>
c = [b]     # Increase ref. count  of <40>


del a       # Decrease ref. count  of <40>
b = 100     # Decrease ref. count  of <40>
c[0] = -1   # Decrease ref. count  of <40>
```

You normally will not notice when the garbage collector destroys an orphaned instance and reclaims its space. But a class can implement the special method *__del__()*, called a destructor, that is invoked when the instance is about to be destroyed. This method might be used to clean up any non-memory resources used by an instance.

ACCESSING CLASS ATTRIBUTES

Instead of using the normal statements to access attributes, you can use the following functions –

- The **getattr(obj, name[, default])** – to access the attribute of object.
- The **hasattr(obj,name)** – to check if an attribute exists or not.
- The **setattr(obj,name,value)** – to set an attribute. If attribute does not exist, then it would be created.
- The **delattr(obj, name)** – to delete an attribute.

```
hasattr(emp1, 'age')    # Returns true if 'age' attribute exists
getattr(emp1, 'age')    # Returns value of 'age' attribute
setattr(emp1, 'age', 8) # Set attribute 'age' at 8
delattr(empl, 'age')    # Delete attribute 'age'
```

BUILT IN CLASS ATTRIBUTES

Every Python class keeps following built-in attributes and they can be accessed using dot operator like any other attribute –

- **__dict__** – Dictionary containing the class's namespace.
- **__doc__** – Class documentation string or none, if undefined.
- **__name__** – Class name.
- **__module__** – Module name in which the class is defined. This attribute is "__main__" in interactive mode.
- **__bases__** – A possibly empty tuple containing the base classes, in the order of their occurrence in the base class list.

DATA ABSTRACTION AND HIDING THROUGH CLASSES

Data encapsulation also called as data hiding, organizes the data and methods into a structure that prevents data access by any function(or method) that is not specified in the class. This ensures the integrity of the data contained in the object.

Encapsulation defines different access levels for data variabes and member functions of the class. These access levels specifies the access rights. For instance,

- Any data or function with access level **public** can be accessed by any function belonging to **any** class. This is the lowest level of data protection
- Any data or functions with access level private can be accesses only by the class in which it is declared. This is the highest level of data protection. In Python, private variables are prefixed with a double underscore (__). For instance, __var is a private variable of the class. We can initialise it by **self.__var=10**

A private member can be accesses using the following syntax:
objectname._classname__privatevariable
We can also have private methods to discourage people from accessing parts of a class that have implementation details.