All the IPython Notebooks in ***Python Object Class*** *lecture series by* [*Dr. Milaan Parmar*](https://www.linkedin.com/in/milaanparmar/) *are available @* [*GitHub*](https://github.com/milaan9/06_Python_Object_Class)

Python Classes and Objects

In this class, you will learn about the core functionality of Python objects and classes. You'll learn what a class is, how to create it and use it in your program.

Python Classes and Objects ¶

Python is an object oriented programming language. Everything in Python is an object, with its properties and methods. A number, string, list, dictionary, tuple, set etc. used in a program is an object of a corresponding built-in class. We create class to create an object. Unlike procedure oriented programming, where the main emphasis is on functions, object oriented programming stresses on objects.

An object is simply a collection of data (variables) and methods (functions) that act on those data. Similarly, a class is a blueprint for that object.

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**. We instantiate a class to create an object. The class defines attributes and the behavior of the object, while the object, on the other hand, represents the class.

We have been working with classes and objects right from the beginning of this challenge unknowingly. Every element in a Python program is an object of a class. Let us check if everything in python is a class:

In [1]:

```
num = 10
print("➞ 10 belongs to",type(num))           # <class 'int'>

string = 'string'
print("➞ 'string' belongs to",type(string))  # <class 'str'>

boolean = True
print("➞ True belongs to",type(boolean))     # <class 'bool'>

list_1 = [1,2,3]
print("➞ [1,2,3] belongs to",type(list_1))   # <class 'list'>

tuple_1 = (1,2,3)
print("➞ (1,2,3) belongs to",type(tuple_1))  # <class 'tuple'>

set_1 = {1,2,3}
print("➞ {1,2,3} belongs to",type(set_1))    # <class 'set'>

dict_1 = {1:'A', 2:'B'}
print("➞ {1:'A', 2:'B'} belongs to",type(dict_1))  # <class 'dict'>
```

```
➞ 10 belongs to <class 'int'>
➞ 'string' belongs to <class 'str'>
➞ True belongs to <class 'bool'>
➞ [1,2,3] belongs to <class 'list'>
➞ (1,2,3) belongs to <class 'tuple'>
➞ {1,2,3} belongs to <class 'set'>
➞ {1:'A', 2:'B'} belongs to <class 'dict'>
```

Defining a Class in Python

Like function definitions begin with the [def](https://github.com/milaan9/01_Python_Introduction/blob/main/Python_Keywords_List.ipynb) keyword in Python, class definitions begin with a [class](https://github.com/milaan9/01_Python_Introduction/blob/main/Python_Keywords_List.ipynb) 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.

Here is a simple class definition.

```
class MyNewClass:
    '''This is a docstring. I have created a new class'''
    pass
```

A class creates a new local [namespace](https://github.com/milaan9/01_Python_Introduction/blob/main/013_Python_Namespace_and_Scope.i) where all its attributes are defined. Attributes may be data or functions.

There are also special attributes in it that begins with double underscores `__` . For example, **`__doc__`** gives us the docstring of that class.

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.

In [2]:
```
# Example 1: Creating Class and Object in Python

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

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

print(Person.age)      # Output: 10

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

print(Person.__doc__)  # Output: 'This is my second class'
```

```
10
<function Person.greet at 0x000001D8AEC3AC10>
This is a person class
```

Creating an Object in Python

We saw that the class object could be used to access different attributes.

It can also be used to create new object instances (instantiation) of that class. The procedure to create an object is similar to a **function** (https://github.com/milaan9/04_Python_Functions/blob/main/001_Python_Functions.ipynb) call.

```
harry = Person()
```

This will create a new object instance named **harry** . We can access the attributes of objects using the object name prefix.

Attributes may be data or method. Methods of an object are corresponding functions of that class.

This means to say, since **Person.greet** is a function object (attribute of class), **Person.greet** will be a method object.

In [3]:
```
class Person:
    "This is a person class"
    age = 10

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

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

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

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

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

```
<function Person.greet at 0x000001D8AEC7E4C0>
<bound method Person.greet of <__main__.Person object at 0x000001D8AEC44EB0>>
Hello
```

Explanation:

You may have noticed the `self` parameter in function definition inside the class but we called the method simply as `harry.greet()` without any **arguments** (https://github.com/milaan9/04_Python_Functions/blob/main/004_Python_Function_Arguments.ipynb). It still worked.

This is because, whenever an object calls its method, the object itself is passed as the first argument. So, `harry.greet()` translates into `Person.greet(harry)` .

In general, calling a method with a list of n arguments is equivalent to calling the corresponding function with an argument list that is created by inserting the method's object before the first argument.

For these reasons, the first argument of the function in class must be the object itself. This is conventionally called self. It can be named otherwise but we highly recommend to follow the convention.

Now you must be familiar with class object, instance object, function object, method object and their

Constructors in Python

In Python, if a class functions that begin with double underscore `__` are called special functions as they have special meaning.

A class without a constructor is not really useful in real applications. A constructor is a special type of method used in Python to initialize the object of a Class. The constructor will be executed automatically when the object is created. If we create three objects, the constructor is called three times and initialize each object.

The main purpose of the constructor is to declare and initialize instance variables. It can take at least one argument that is `self` . The `__init__()` method is called the constructor in Python. In other words, the name of the constructor should be `__init__(self)` .

A constructor is optional, and if we do not provide any constructor, then Python provides the default constructor. Every class in Python has a constructor, but it's not required to define it.

In [4]:
```
# Example 1:

class ComplexNumber:
    def __init__(self,r=0,i=1):
        self.real=r;
        self.imag=i;

    def getData(self):
        print('{0}+{1}j'.format(self.real,self.imag))


c1=ComplexNumber(5,6)
c1.getData()
```

```
5+6j
```

In [5]:

```python
# Example 2:

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')

# Create a new ComplexNumber object
num1 = ComplexNumber(2, 3)

# Call get_data() method

num1.get_data()   # Output: 2+3j

# Create another ComplexNumber object and create a new attribute 'attr'
num2 = ComplexNumber(5)
num2.attr = 10

print((num2.real, num2.imag, num2.attr))  # Output: (5, 0, 10)

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

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

---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-5-05908a030c74> in <module>
     23
     24 # but c1 object doesn't have attribute 'attr'
---> 25 print(num1.attr)  # AttributeError: 'ComplexNumber' object has no attribut
e 'attr'

AttributeError: 'ComplexNumber' object has no attribute 'attr'
```

Explanation:

In the above 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.

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`** .

Types of Constructors

We have two types of constructors in Python:

- **Non-parameterized:** The constructors in Python which have no parameter is known as a non parameterized constructor. The non-parameterized constructor uses when we do not want to manipulate the value or the constructor that has only self as an argument.
- **Parameterized Constructor:** The constructor with parameters is known as a parameterized constructor. The parameterized constructor has multiple parameters along with the self.

In [6]:

```python
# Example 1: Constructor without parameters (Non-parameterized)

class Test:
    # Constructor - non parameterized
    def __init__(self):
        print("This is non parametrized constructor")

    def show(self, name):
        print("Hello", name)

# creating object of the class
t = Test()          # Output:This is non parametrized constructor

# calling the instance method
t.show("Arthur")    # Output Hello Arthur
```

```
This is non parametrized constructor
Hello Arthur
```

In [7]:

```python
# Example 2: Constructor with parameters

class Fruit:
    # parameterized constructor
    def __init__(self, name, color):
        print("This is parametrized constructor")
        self.name = name
        self.color = color

    def show(self):
        print("Fruit is", self.name, "and Color is", self.color)

# creating object of the class
# this will invoke parameterized constructor
obj = Fruit("Apple", "red")   # Output This is parametrized constructor

# calling the instance method using the object
obj.show()                    # Output Fruit is Apple and Color is red
```

```
This is parametrized constructor
Fruit is Apple and Color is red
```

Explanation:

In the above example, we create a **parameterized constructor** with parameters `name` and `color`. When we create an object of `Test` class called `obj`, the parameterized constructor will be executed automatically.

In [8]:

```
# Example 3:

class Student:
    # Constructor - parameterized
    def __init__(self, name):
        print("This is parametrized constructor")
        self.name = name

    def show(self):
        print("Hello",self.name)

student = Student("World")
student.show()
```

```
This is parametrized constructor
Hello World
```

In [9]:

```
# Example 4: Creating Class and Object in Python

class MasterStudentClass:
    # class attribute
    species = "students"

    # instance attribute
    def __init__(self, name, age):
        self.name = name
        self.age = age

# instantiate the Parrot class
jane = MasterStudentClass("Jane", 18)
bella = MasterStudentClass("Bella", 19)
candy = MasterStudentClass("Candy", 17)
lucia = MasterStudentClass("Lucia", 18)
ran = MasterStudentClass("Ran", 20)

# access the class attributes
print("Jane is a {}".format(jane.__class__.species))         # Jane is a students
print("Bella is also a {}".format(bella.__class__.species))  # Bella is also a stu
```

```
Jane is a students
Bella is also a students
```

In [10]:

```
# Example 5:

class Student:

    # class attribute
    'Common base class for all students'
    student_count=0

    def __init__(self, name, id):  # check the number of underscore '_' used
        self.name = name
        self.id = id
        Student.student_count+=1

    def printStudentData(self):
        print ("Name : ", self.name, ", Id : ", self.id)

s=Student("Mark",101)
s.printStudentData()  # Name :  Mark , Id :  101
```

```
Name :  Mark , Id :  101
```

Explanation:

- The variable **`student_count`** is a class variable whose value is shared among all the instances of a in this class. This can be accessed as **`Student.student_count`** from inside the class or outside the class.
- The first method **`__init__()`** is a special method, which is called class **constructor** or **initialization** method that Python calls when you create a new instance of this class.
- You declare other class methods like normal functions with the exception that the first argument to each method is self. Python adds the self argument to the list for you; you do not need to include it when you call the methods.

Creating Instance Objects

To create instances of a class, you call the class using class name and pass in whatever arguments its **`__init__`** method accepts. Lets Create Studnet class object of above example :

```
std=Student('Vijay','102')
```

Accessing Attributes with `self` Parameter

The **`self`** is used to represent the instance of the class. It is the default variable that is always pointing to the current object.

By using **`self`** , we can access the instance variable and instance method of the object. While defining constructor and instance method, the **`self`** is their first parameter.

It's not required the first parameter named to be **`self`** , we can give any name whatever we like, but it has to be the first parameter of any function in the class.

You access the object's attributes using the dot operator with object. Class variable would be accessed using class name as follows:

In [11]:
```
# Example 1:

class Employee:
    def __init__(self, id, name):
        # instance variable
        self.id = id
        self.name = name

    # instance method
    def info(self):
        print("Employee ID is ", self.id, "and name is", self.name)

    # instance method
    def department(self):
        print("Employee of IT department")

emp = Employee(19116, "Amy", )
emp.info()          # Output Employee ID is 19116 and name is Amy

emp.department()    # Output Employee of IT department
```

```
Employee ID is  19116 and name is Amy
Employee of IT department
```

Explanation:

In the above example, all methods including **__init__** have a self parameter. We created two instance variables **id** and **name** . Then we create an object of a class **Employee** called **emp** and accessed instance methods of the class called **info()** and **department()** respectively.

In [12]:
```
# Example 2:

class Student:
    'Common base class for all students'
    student_count=0

    def __init__(self, name, id):
        self.name = name
        self.id = id
        Student.student_count+=1

    def printStudentData(self):
        print ("Name : ", self.name, ", Id : ", self.id)

std1=Student("Milan",101)
std2=Student("Vijay",102)
std3=Student("Chirag",103)

print("Total Student : ",Student.student_count)
std1.printStudentData()
std2.printStudentData()
std3.printStudentData()
```

```
Total Student :  3
Name :  Milan , Id :  101
Name :  Vijay , Id :  102
Name :  Chirag , Id :  103
```

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

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

Example:

```
hasattr(std1, 'id') # Returns true if 'id' attribute exists
getattr(std1, 'id') # Returns value of 'id' attribute
setattr(std1, 'id', 104) # Set attribute 'id' 104
delattr(std1, 'id') # Delete attribute 'id'
```

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.

let us try to access all these attributes

In [13]:
```
# Example 1:

class Student:
    'Common base class for all students'
    student_count=0

    def __init__(self, name, id):
        self.name = name
        self.id = id
        Student.student_count+=1

    def printStudentData(self):
        print ("Name : ", self.name, ", Id : ", self.id)

std1=Student("Milan",101)
std2=Student("Vijay",102)
std3=Student("Chirag",103)

print("Total Student : ",Student.student_count)
print ("Student.__doc__:", Student.__doc__)
print ("StudentStudent.__name__:", Student.__name__)
print ("Student.__module__:", Student.__module__)
print ("Student.__bases__:", Student.__bases__)
print ("Student.__dict__:", Student.__dict__)
```

```
Total Student :  3
Student.__doc__: Common base class for all students
StudentStudent.__name__: Student
Student.__module__: __main__
Student.__bases__: (<class 'object'>,)
Student.__dict__: {'__module__': '__main__', '__doc__': 'Common base class for all
students', 'student_count': 3, '__init__': <function Student.__init__ at 0x000001D
8AEC7E1F0>, 'printStudentData': <function Student.printStudentData at 0x000001D8AE
C7E5E0>, '__dict__': <attribute '__dict__' of 'Student' objects>, '__weakref__': <
attribute '__weakref__' of 'Student' objects>}
```

Object Properties

Every object has properties with it. In other words, we can say that object property is an association between **name** and **value**.

For example, a car is an object, and its properties are car color, sunroof, price, manufacture, model, engine, and so on. Here, color is the name and red is the **value**.

Modify Object Properties

Every object has properties associated with them. We can set or modify the object's properties after object initialization by calling the property directly using the dot operator.

In [14]:
```
# Example 1:

class Fruit:
    def __init__(self, name, color):
        self.name = name
        self.color = color

    def show(self):
        print("Fruit is", self.name, "and Color is", self.color)

# creating object of the class
obj = Fruit("Apple", "red")

# Modifying Object Properties
obj.name = "strawberry"

# calling the instance method using the object obj
obj.show()    # Output Fruit is strawberry and Color is red
```

```
Fruit is strawberry and Color is red
```

Delete object properties

We can delete the object property by using the **`del`** keyword. After deleting it, if we try to access it, we will get an error.

In [15]:
```
# Example 1:

class Fruit:
    def __init__(self, name, color):
        self.name = name
        self.color = color

    def show(self):
        print("Fruit is", self.name, "and Color is", self.color)

# creating object of the class
obj = Fruit("Apple", "red")

# Deleting Object Properties
del obj.name

# Accessing object properties after deleting
print(obj.name)    # Output: AttributeError: 'Fruit' object has no attribute 'name'
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-15-250fe64180f3> in <module>
     16
     17 # Accessing object properties after deleting
---> 18 print(obj.name)    # Output: AttributeError: 'Fruit' object has no attribut
e 'name'

AttributeError: 'Fruit' object has no attribute 'name'
```

Delete Objects

In Python, we can also delete the object by using a **`del`** keyword. An object can be anything like, class object, list, tuple, set, etc.

Syntax:

```
del object_name
```

In [16]:

```
# Example 1:

class Employee:
    depatment = "IT"

    def show(self):
        print("Department is ", self.depatment)

emp = Employee()
emp.show()

# delete object
del emp

# Accessing after delete object
emp.show()  # Output : NameError: name 'emp' is not defined
```

```
Department is  IT

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-16-4340020aef34> in <module>
     14
     15 # Accessing after delete object
---> 16 emp.show()  # Output : NameError: name 'emp' is not defined

NameError: name 'emp' is not defined
```

Explanation:

In the above example, we create the object **emp** of the class **Employee** . After that, using the **del** keyword, we deleted that object.

In [17]:

```
# Example 2:

num1 = ComplexNumber(2,3)
del num1.imag
num1.get_data()
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-17-86635ab2cc7e> in <module>
      3 num1 = ComplexNumber(2,3)
      4 del num1.imag
----> 5 num1.get_data()

<ipython-input-5-05908a030c74> in get_data(self)
      7
      8     def get_data(self):
----> 9         print(f'{self.real}+{self.imag}j')
     10
     11 # Create a new ComplexNumber object

AttributeError: 'ComplexNumber' object has no attribute 'imag'
```

In [18]:
```
# Example 3:

del ComplexNumber.get_data
num1.get_data()
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-18-b8eab4452d4b> in <module>
      2
      3 del ComplexNumber.get_data
----> 4 num1.get_data()

AttributeError: 'ComplexNumber' object has no attribute 'get_data'
```

We can even delete the object itself, using the del statement!

In [19]:
```
# Example 4:

c1 = ComplexNumber(1,3)
del c1
c1
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-19-45d582f31548> in <module>
      3 c1 = ComplexNumber(1,3)
      4 del c1
----> 5 c1

NameError: name 'c1' is not defined
```

Explanation:

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.

Data Hiding

An object's attributes may or may not be visible outside the class definition. You need to name attributes with a double underscore prefix, and those attributes then will not be directly **visible** to outsiders.

In [20]:

```
class JustCounter:
    __secretCount = 0  # private attribute

    def count(self):
        self.__secretCount += 1
        print (self.__secretCount)

counter = JustCounter()
counter.count()
counter.count()
print (counter.__secretCount)
```

```
1
2

---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-20-59b7b8cc9b5b> in <module>
      9 counter.count()
     10 counter.count()
---> 11 print (counter.__secretCount)

AttributeError: 'JustCounter' object has no attribute '__secretCount'
```

Python protects those members by internally changing the name to include the class name. You can access such attributes as **object._className__attrName** . If you would replace your last line as following, then it works for you −

In [21]:

```
class JustCounter:
    __secretCount = 0

    def count(self):
        self.__secretCount += 1
        print (self.__secretCount)

counter = JustCounter()
counter.count()
counter.count()
# print (counter.__secretCount)  # This wont work
print (counter._JustCounter__secretCount)
```

```
1
2
2
```

In [22]:

```
# convert_dictionary_to_python_object

class obj(object):
    def __init__(self, d):
        for x, y in d.items():
            setattr(self, x, obj(y) if isinstance(y, dict) else y)
data = {'a':5,'b':7,'c':{'d':8}}
ob = obj(data)
print(data)
```

```
{'a': 5, 'b': 7, 'c': {'d': 8}}
```

In [ ]: