Python Objects and Classes

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 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 a 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**.

Defining a Class in Python

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 of 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 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.

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

print(Person.age)

print(Person.__doc__)
```

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 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.

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

# Output: 10
print(Person.age)

# Output: "This is a person class"
print(Person.__doc__)

p1=Person()
print(p1.age)
```

Methods

```
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()
```

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. 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.

```
# define the Vehicle class
class Vehicle:
    name = ""
    kind = "car"
    color = ""
    value = 100.00
    def description(self):
        desc_str = "%s is a %s %s worth $%.2f." % (self.name, self.color, self.kind, self.value)
        return desc_str

# your code goes here
car1 = Vehicle()
car1.name = "Fer"
car1.color = "red"
car1.kind = "convertible"
car1.value = 60000.00

car2 = Vehicle()
car2.name = "Jump"
car2.color = "blue"
car2.kind = "van"
car2.value = 10000.00

# test code
print(car1.description())
print(car2.description())
```

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.

Types of constructors :

- **default constructor:** The default constructor is a simple constructor which doesn’t accept any arguments. Its definition has only one argument which is a reference to the instance being constructed.
- **parameterized constructor:** constructor with parameters is known as parameterized constructor. The parameterized constructor takes its first argument as a reference to the instance being constructed known as self and the rest of the arguments are provided by the programmer.

```
class str:

# default constructor
     def __init__(self):
          self.strng = "hello"

# a method for printing data members
     def print_str(self):
          print(self.strng)

# creating object of the class
obj = str()

# calling the instance method using the object obj
obj.print_str()
```

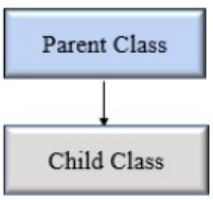
```
class NumberHolder:

   def __init__(self, number):
       self.number = number

   def returnNumber(self):
       return self.number

var = NumberHolder(7)
print(var.returnNumber()) #Prints '7'
```

Inheritance

```
class ParentClass:

      Body of base class

class DerivedClass(ParentClass):
      Body of derived class
```

Single Inheritance

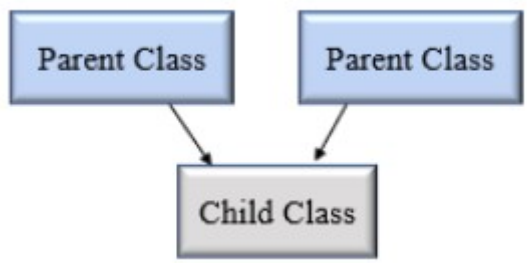
In single inheritance, a child class inherits from a single-parent class. Here is one child class and one parent class.



```
# Base class
class Vehicle:
    def Vehicle_info(self):
        print('Inside Vehicle class')

# Child class
class Car(Vehicle):
    def car_info(self):
        print('Inside Car class')

# Create object of Car
car = Car()

# access Vehicle's info using car object
car.Vehicle_info()
car.car_info()
```

Multiple Inheritance

In multiple inheritance, one child class can inherit from multiple parent classes. So here is one child class and multiple parent classes.

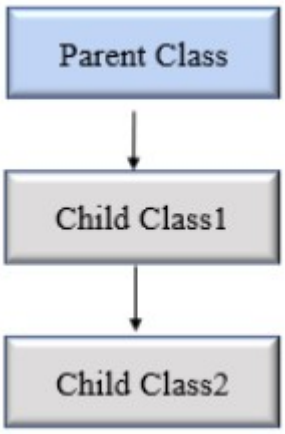



```
# Parent class 1
class Person:
    def person_info(self, name, age):
        print('Inside Person class')
        print('Name:', name, 'Age:', age)

# Parent class 2
class Company:
    def company_info(self, company_name, location):
        print('Inside Company class')
        print('Name:', company_name, 'location:', location)

# Child class
class Employee(Person, Company):
    def Employee_info(self, salary, skill):
        print('Inside Employee class')
        print('Salary:', salary, 'Skill:', skill)

# Create object of Employee
emp = Employee()

# access data
emp.person_info('Jessa', 28)
emp.company_info('Google', 'Atlanta')
```

```
emp.Employee_info(12000, 'Machine Learning')
```

Multilevel inheritance

In multilevel inheritance, a class inherits from a child class or derived class. Suppose three classes A, B, C. A is the superclass, B is the child class of A, C is the child class of B. In other words, we can say a **chain of classes** is **called multilevel inheritance.**



Python `super()` function

When a class inherits all properties and behavior from the parent class is called inheritance. In such a case, the inherited class is a subclass and the latter class is the parent class.

In child class, we can refer to parent class by using the `super()` function. The super function returns a temporary object of the parent class that allows us to call a parent class method inside a child class method.

Benefits of using the `super()` function.

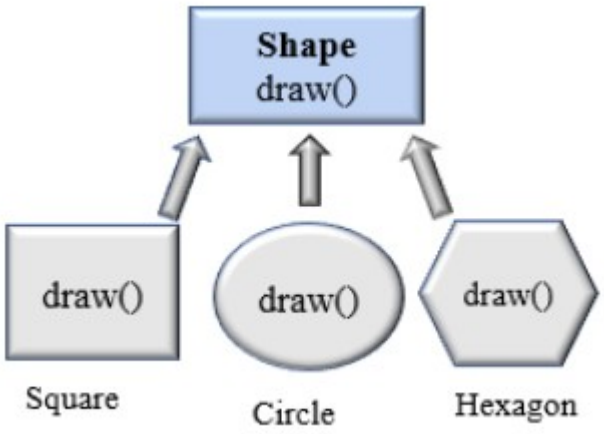
1. We are not required to remember or specify the parent `class` name to access its methods.
2. We can use the `super()` function in both **single** and **multiple inheritances**.
3. The `super()` function support code **reusability** as there is no need to write the entire function

Method Overriding

In inheritance, all members available in the parent class are by default available in the child class. If the child class does not satisfy with parent class implementation, then the child class is allowed to redefine that method by extending additional functions in the child class. This concept is called method overriding.

When a child class method has the same name, same parameters, and same return type as a method in its superclass, then the method in the child is said to **override** the method in the parent class.

When a child class method has the same name, same parameters, and same return type as a method in its superclass, then the method in the child is said to **override** the method in the parent class.



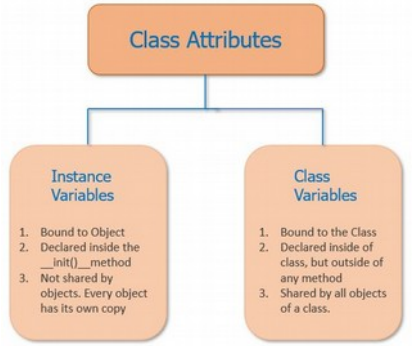

```
class Vehicle:
    def max_speed(self):
        print("max speed is 100 Km/Hour")

class Car(Vehicle):
    # overridden the implementation of Vehicle class
    def max_speed(self):
        print("max speed is 200 Km/Hour")

# Creating object of Car class
car = Car()
car.max_speed()
```

Class Attributes

When we design a class, we use instance variables and class variables.

In Class, attributes can be defined into two parts:

- **Instance variables**: The instance variables are attributes attached to an instance of a class. We define instance variables in the constructor (the `__init__()` method of a class).
- **Class Variables**: A class variable is a variable that is declared inside of class, but outside of any instance method or `__init__()`method.



Objects do not share instance attributes. Instead, every object has its copy of the instance attribute and is unique to each object.

All instances of a class share the class variables. However, unlike instance variables, the value of a class variable is not varied from object to object.

Only one copy of the static variable will be created and shared between all objects of the class.

Accessing properties and assigning values

- An instance attribute can be accessed or modified by using the dot notation: `instance_name.attribute_name`.
- A class variable is accessed or modified using the class name

Destructors are called when an object gets destroyed. In Python, destructors are not needed as much as in C++ because Python has a garbage collector that handles memory management automatically.
The **__del__()** method is a known as a destructor method in Python. It is called when all references to the object have been deleted i.e when an object is garbage collected.

Set

Set in python is a data structure or data type that allows storing lots of mutable data into a single variable. The elements of the sets are unordered, there is no index number associated with them. Therefore, accessing the set elements with the help of index numbers is not possible.

Set are mutable just like a list which means, once a set is defined we can modify and update it later. we already know that set returns an element without duplicate.

```
Blog = {"hello", "Iterathon", "I", "am", "There",”am”}
for i in Blog:
      print(i)

#Output
There
Iterathon
I
am
hello
```

A set has the following characteristics:

- **Unique values.** You cannot have duplicates in a set.
- **Unordered collection.** There is no order in the set.
- **Mutable.** You can modify the set by adding and removing values.
- **Supports mathematical set operations,** such as `union`, `difference`, `intersection`.

Set add() Method :

The set add () method adds a given item to a set. Here is the syntax –
set.add(item)

Note: If the element already exists, then it doesn't add any item

clear() Method :

The name itself self-explanatory. The clear() method clears/removes all items from the set.

difference() Method

```
#difference() method in Python

A = {1, 2, 3, 4, 5}
B = {3, 4, 5, 6, 7, 8}

# Equivalent to A-B
A.difference(B)
set([1, 2])

# Equivalent to B-A
B.difference(A)
set([8, 6, 7])

# No argument provided -> Original set
A.difference()
set([1, 2, 3, 4, 5])
```

discard() Method :

In this method, it deletes an argument item from the set if it exists, or else it returns nothing.

intersection() Method :

This intersection() method returns a new set of items that are common of all sets.

```
A = {1, 2, 3, 4, 5}
B = {3, 4, 5, 6, 7, 8}
A.intersection(B)
#Output
set([3, 4, 5])

B.intersection(A)
Output
set([3, 4, 5])
```

remove() Method :

In remove() method, it will search for the item passed to the argument in the set and removes it.

Set union() Method :

In the union() method, it returns a new set that contains distinct items from all the sets.

Frozen Sets in Python

Frozen sets in Python are sets that cannot be changed after creation. In addition, they cannot contain duplicate values.

A frozen set in Python is like a combination of tuples and sets. A frozen set cannot be modified and does not accept duplicates. It can be used to carry out mathematical set operations.

A frozen set has the following characteristics:

- **Unique collection.** No duplicate values are allowed.
- **Unordered.** The notion of order is meaningless in a frozen set.
- **Immutable.** You cannot change a frozen set after creation.
- **Supports (immutable) set operations**, such as `union`, `difference`, `intersection`.

To create a frozen set, wrap a set around a `frozenset()` call:

constants = frozenset({3.141, 2.712, 1.414})
names = frozenset({"Alice", "Bob", "Charlie"})

Python’s frozen set is like a combination of a tuple and a set.

Set methods can be used in frozen set

Frozen Set Conversions in Python

You can convert other types into frozen sets in Python.

List to Frozenset

You can convert other types into frozen sets in Python.

List to Frozenset

```
nums = [1, 2, 2, 3, 3, 3, 6]
A = frozenset(nums)

print(A)
```

Tuple to Frozenset

Similar to converting a list to a `frozenset`, you can convert a tuple to one.

For instance:

```
nums = (1, 2, 2, 3, 3, 3, 6)
A = frozenset(nums)

print(A)
```

Dictionary to Frozenset

It is possible to convert a dictionary to a `frozenset` as well. Notice that doing this removes all the important value information from the dictionary.

```
person = {
    "name": "Jack",
    "married": False,
    "age": 33
}

A = frozenset(person)

print(A)
```

output

```
frozenset({'name', 'age', 'married'})
```

Display single item

```
A = frozenset({1, 2, 3, 6})

for number in A:
```

```
print(number)
```