Python Dictionaries

Python Dictionaries

- A dictionary in Python is a collection of data that is unordered, indexed, and can be changed.
- A dictionary contains pairs of keys and values so it can have more than one element and holds different data types.
- Its syntax is key : value.
- **Creating a Dictionary :**
  - There are two ways to create a dictionary
  - **The first one** is declaring a variable and assigning to it the desired keys with the desired values and put them into curly brackets{}.

Python Dictionaries

Ex:

```
Fruit = { "kind": "Banana","color": "Yellow","weight": 0.5}
print(Fruit)
```

Output:

```
{'kind': 'Banana', 'color': 'Yellow', 'weight': 0.5}
```

- **The second way** is using the dict() constructor.
- Where the desired values are assigned to their keys using equals instead of colons and they are then passed to the dict() constructor.

Ex :

```
Fruit = dict(kind = "Banana", color="Yellow", weight = 0.5 )
print(Fruit)
```

Output:

```
{'kind': 'Banana', 'color': 'Yellow', 'weight': 0.5}
```

Python Dictionaries

- **Accessing elements of the dictionary:**
  - There are several ways to access variables inside a dictionary.
  - **The first way** is to assign the name of the dictionary followed by square brackets[], where these brackets contain the key to the desired variable.
  - **The second way** to access a dictionary value by using get() function.
  - By assigning the dictionary name attached to it the get() function to a variable, where the key to the desired value is passed as a get() function parameter.

Ex:

```
Fruit = {"kind": "Banana","color": "Yellow","weight": 0.5}
z = Fruit["kind"]  # first way
print(z)
y = Fruit.get("kind") # second way
print(y)
```

Output:

```
Banana
Banana
```

Python Dictionaries

- **Changing a value in a dictionary :**
  - You can change the value of a specific item by referring to its key name:

Ex:

```
Fruit = {"kind": "Banana","color": "Yellow","weight": 0.5}
Fruit["weight"] = 1.5
print(Fruit)
```

Output:

```
{'kind': 'Banana', 'color': 'Yellow', 'weight': 1.5}
```

- **Update Dictionary:**
  - The update() method will update the dictionary with the items from the given argument.
  - The argument must be a dictionary, or an iterable object with key:value pairs.

Ex:

```
Fruit = {"kind": "Banana","color": "Yellow","weight": 0.5}
Fruit.update({"weight": 1.5})
print(Fruit)
```

Output:

```
{'kind': 'Banana', 'color': 'Yellow', 'weight': 1.5}
```

Python Dictionaries

- **Get all keys in a dictionary:**
  - To get all the keys available in a dictionary you can use the keys() function.
  - When the dictionary name is written attached to it the keys() function, it returns a display of all the keys as a list.
  - Another way to get the keys present in a dictionary by looping through the dictionary using for a loop
- **Get all values in a dictionary:**
  - To get all the values in a dictionary you can use the values() function attached to the dictionary name.

Ex:

```
Fruit = {"kind": "Banana","color": "Yellow","weight": 0.5}
print(Fruit.values())
print(Fruit.keys())
```

Output:

```
dict_values(['Banana', 'Yellow', 0.5])
dict_keys(['kind', 'color', 'weight'])
```

Python Dictionaries

- **Removing an element from a dictionary**:
  - There are several ways to remove an element from a dictionary.
  - The first way is using the pop() function.
  - Another way to remove is using popitem() function.
  - The third way to remove an element is by using the del keyword.
  - The clear() method empties the dictionary.

Ex:

```
Fruit = {"kind": "Banana","color": "Yellow","weight": 0.5,"color1":"Red"}
Fruit.pop("color")
print(Fruit)
z = Fruit.popitem()
print("the removed element: ",z)
print("updated dictionary: ",Fruit)
del Fruit["weight"]
print("updated dictionary: ",Fruit)
Fruit.clear()
print(Fruit)
```

Output:

```
{'kind': 'Banana', 'weight': 0.5, 'color1': 'Red'}
the removed element:  ('color1', 'Red')
updated dictionary:  {'kind': 'Banana', 'weight': 0.5}
updated dictionary:  {'kind': 'Banana'}
{}
```

Python Dictionaries

- **convert list to dictionary :**
  - **Method 1 – Iterating over the list**

Ex:

```
def convert(x):
    dic={}
    for i in range(0,len(x),2):
        dic[x[i]]=x[i+1]
    return dic
a1=[1,'Python',2,'Java',3,'C',4,'C++']
print(convert(a1))
```

Output:

```
{1: 'Python', 2: 'Java', 3: 'C', 4: 'C++'}
```

  - **Method 2 – Using zip() :**

Ex:

```
def convert(x):
    i=iter(x)
    dic=dict(zip(i,i))
    return dic

a1=[1,'Python',2,'Java',3,'C',4,'C++']
print(convert(a1))
```

Output:

```
{1: 'Python', 2: 'Java', 3: 'C', 4: 'C++'}
```

Python Classes and Objects

- Python is an object oriented programming language.
- Almost everything in Python is an object, with its properties and methods.
- A Class is like an object constructor, or a "blueprint" for creating objects.
- **Create a Class:**
  - To create a class, use the keyword class:

  Ex:

```
class MyClass:
    x = 5
```

- **Create Object:**
  - <object-name> = <class-name>(<arguments>)

  Ex:

```
p1 = MyClass()
print(p1.x)
```

Python Classes and Objects

- The __init__() Function:
  - In python classes, "__init__" method is reserved.
  - It is automatically called when you create an object using a class and is used to initialize the variables of the class.
  - It is equivalent to a constructor.
  - Like any other method, init method starts with the keyword "def"
  - "self" is the first parameter in this method just like any other method although in case of init, "self" refers to a newly created object unlike other methods where it refers to the current object or instance associated with that particular method.
  - Additional parameters can be added.
  - The __init__() function is called automatically every time the class is being used to create a new object.

Python Classes and Objects

Ex:

```
class A:
    def __init__(self, course):
        self.course = course
    def display(self):
        print(self.course)
object1 = A("Python")
object1.display()
```

Output:

Python

Ex:

```
class A:
    def __init__(a, course):
        a.course = course
    def display(x):
        print(x.course)
object1 = A("Python")
object1.display()
```

Output:

Python

- The self Parameter:
  - The self parameter is a reference to the current instance of the class, and is used to access variables that belongs to the class.
  - It does not have to be named self , you can call it whatever you like, but it has to be the first parameter of any function in the class.

Python Classes and Objects

Ex:

```
class Employee:
    id = 15
    name = "Riya"
    def display (self):
        print("ID: %d \nName: %s"%(self.id,self.name))
# Creating a emp instance of Employee class
emp = Employee()
emp.display()
```

Output:

```
ID: 15
Name: Riya
```

Python Classes and Objects

- Accessing Function:
  - To access a function inside of an object you use notation similar to accessing a variable.

Ex:

```
class MyClass:
    variable = "python"
    def function(self):
        print("Function call inside class")
myobjectx = MyClass()
myobjectx.function()
```

Output:

```
Function call inside class
```

- The pass Statement:
  - class definitions cannot be empty, but if you for some reason have a class definition with no content, put in the pass statement to avoid getting an error.

Ex:

```
class Person:
 pass
```

Python Constructor

- A constructor is a special type of method (function) which is used to initialize the instance members of the class.
- In C++ or Java, the constructor has the same name as its class, but it treats constructor differently in Python. It is used to create an object.
- Constructors can be of two types.
  - Parameterized Constructor
  - Non-parameterized Constructor

Python Default Constructor

- When we do not include the constructor in the class or forget to declare it, then that becomes the default constructor.
- It does not perform any task but initializes the objects.

Ex:

```
class Student:
    roll_num = 101
    name = "Riya"

    def display(self):
        print(self.roll_num,self.name)

st = Student()
st.display()
```

Output:

101 Riya

Python Constructor

- **Creating the constructor in python:**
  - In Python, the method the __init__() simulates the constructor of the class.
  - This method is called when the class is instantiated.
  - It accepts the self-keyword as a first argument which allows accessing the attributes or method of the class.
  - We can pass any number of arguments at the time of creating the class object, depending upon the __init__() definition.
  - It is mostly used to initialize the class attributes.
  - Every class must have a constructor, even if it simply relies on the default constructor.

Python Constructor

Ex:

```
class Employee:
    def __init__(self, name, id):
        self.id = id
        self.name = name

    def display(self):
        print("ID: %d \nName: %s" % (self.id, self.name))

emp1 = Employee("Riya", 101)
emp2 = Employee("Mona", 102)
emp1.display()
emp2.display()
```

Output:

```
    ID: 101
    Name: Riya
    ID: 102
    Name: Mona
```

Python Non-Parameterized Constructor

- The default constructor is simple constructor which doesn't accept any arguments.
- It's definition has only one argument which is a reference to the instance being constructed.

Ex:

```
class Student:


   def __init__(self):
       print("Non parametrized constructor")
    def show(self,name):
       print("Hello",name)
student = Student()
student.show("India")
```

Output:

```
Non parametrized constructor
Hello India
```

Python Parameterized Constructor

- constructor with parameters is known as parameterized constructor.
- The parameterized constructor take 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.
- The constructor overloading is not allowed in Python.

Ex:

```
class Student:

    def __init__(self, name):
        print("parametrized constructor")
        self.name = name
    def show(self):
        print("Hello",self.name)
student = Student("Aman")
student.show()
```

Output:

```
parametrized constructor
Hello Aman
```

Python Parameterized Constructor

Ex:

```
class Addition:
        a = 0
        b = 0
        sum = 0
    def __init__(self, x, y):
            self.a = x
            self.b = y
      def display(self):
            print("First number = " + str(self.a))
            print("Second number = " + str(self.b))
            print("Addition of two numbers = " + str(self.sum))
      def calculate(self):
            self.sum = self.a + self.b
    obj = Addition(10, 20)
    obj.calculate()
    obj.display()
```

Output:

```
First number = 10
Second number = 20
Addition of two numbers = 30
```

Python Inheritance

- Inheritance is the capability of one class to derive or inherit the properties from another class.
- The child class acquires the properties and can access all the data members and functions defined in the parent class.
- A child class can also provide its specific implementation to the functions of the parent class.
- **Parent class** is the class being inherited from, also called base class.
- **Child class** is the class that inherits from another class, also called derived class.
- In python, a derived class can inherit base class by just mentioning the base in the bracket after the derived class name.

  Ex:

  class derived-class(base class):

  <class-suite>

- A class can inherit multiple classes by mentioning all of them inside the bracket.

  Ex:

  class derive-class(<base class 1>, <base class 2>, ... <base class n>):

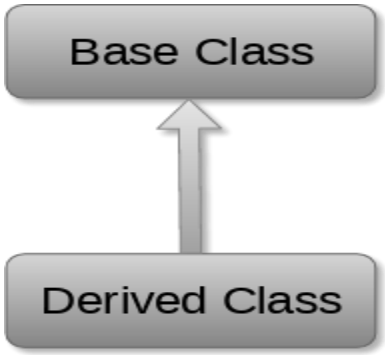
  <class - suite>

Single Inheritance

- When a child class inherits from only one parent class, it is called single inheritance.

Ex:

```
class A:
    def display(self):
        print("class A")
#child class B inherits the base class A
class B(A):
    def show(self):
        print("class B")
obj = B()
obj.show()
obj.display()
```



Output:

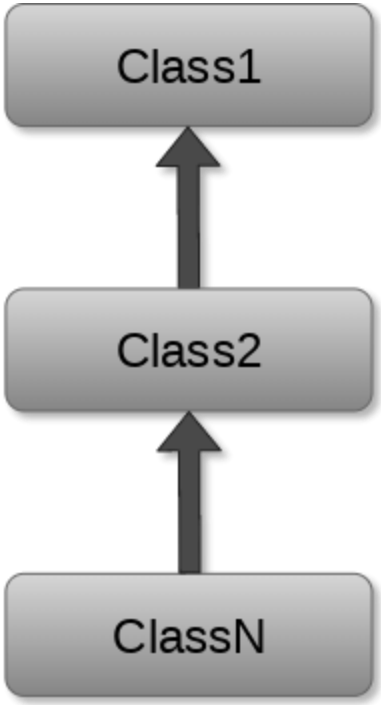
```
class B
class A
```

Multi-Level Inheritance

- Multi-level inheritance is archived when a derived class inherits another derived class.
- There is no limit on the number of levels up to which, the multi-level inheritance is archived in python.

Ex:

```
class A:
    def display(self):
        print("class A")
class B(A):  #child class B inherits the base class A
    def show(self):
        print("class B")
class C(B):   #child class c inherits the base class A
    def printmsg(self):
        print("class c")
obj = C()
obj.printmsg()
obj.show()
obj.display()
```



Output:

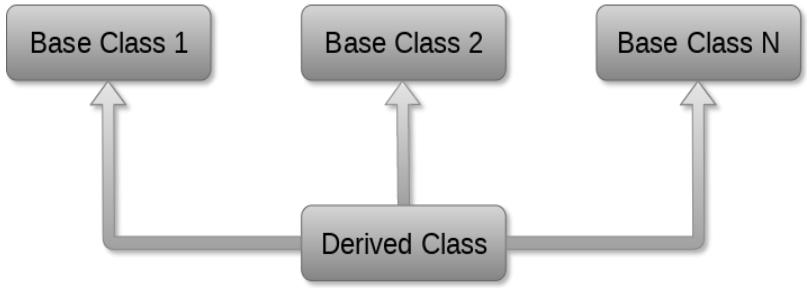
```
class c
class B
class A
```

Multiple Inheritance

- Python provides us the flexibility to inherit multiple base classes in the child class.

Ex:

```
class A:
        def display(self):
            print("class A")
    class B:
        def show(self):
            print("class B")
     #child class c inherits the base class A ,class B
    class C(A,B):
        def printmsg(self):
            print("class C")
    obj = C()
    obj.display()
    obj.show()
    obj.printmsg()
```



Output:

```
class A
class B
class C
```

- We have child class, that inherits all the properties of parent class
- Add def __init__() function to the child class.
- This function call automatically every time the class being used to create a new object.

Ex:

```
class Student(Person):
  def __init__(self, fname, lname):
```

- **Note :** child function override the __init__(), add a call to the parents's __init__() function.

Ex:

```
class Student(Person):
  def __init__(self, fname, lname):
    Person.__init__(self, fname, lname)
```

- **Use the super() Function**
- Python also has the super() function that will make the child class inherit all the method and properties from the parent:
- By using super function (), you do not have to use the name of the parent element, it will automatically inherit the method and properties from its parent.
- **Add Properties**
- Add a property to the student class

Ex:

```
class Student(Person):
  def __init__(self, fname, lname):
    super().__init__(fname, lname)
    self.graduationyear = 2019
```

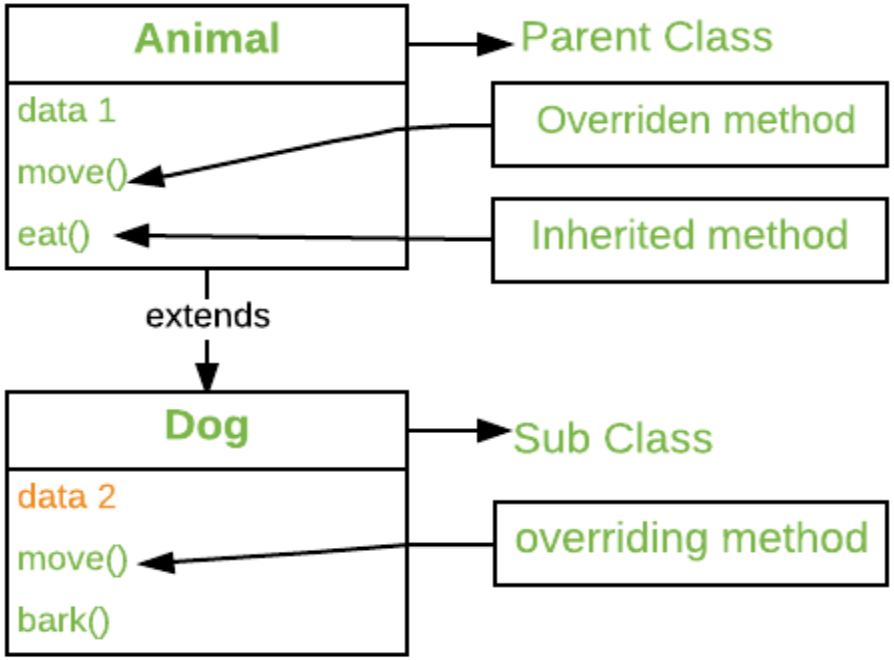
Ex:

```
class Person:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname
  def printname(self):
    print(self.firstname, self.lastname)

class Student(Person):
  def __init__(self, fname, lname):
    super().__init__(fname, lname)
    self.graduationyear = 2019

x = Student("Mike", "Olsen")
print(x.graduationyear)
```

Output:

2019

Method Overriding in Python



- When a method in a subclass has the same name, same parameters or signature and same return type(or sub-type) as a method in its super-class, then the method in the subclass is said to **override** the method in the super-class.

- # Python program to demonstrate method overriding

- # Defining parent class

```
class Parent():

    # Constructor
    def __init__(self):
     self.value = "Inside Parent"
     # Parent's show method
    def show(self):
            print(self.value)
```

```
# Defining child class
class Child(Parent):

    # Constructor
    def __init__(self):
            self.value = "Inside Child"

    # Child's show method
    def show(self):
            print(self.value)
```

```
# Driver's code
obj1 = Parent()
obj2 = Child()

obj1.show()
obj2.show()
```

Method Overloading Python

- python does not support method overloading by default.
- Same name function defined in same class with different number of argument and with different types of argument.
- In case, you overload methods then you will be able to access the last defined method.

Ex:

```
def product(a, b):
        p = a * b
        print(p)
def product(a, b, c):
        p = a * b*c
        print(p)
product(4, 5, 5)
```

Output:

```
100
```

Duck typing in python

- Duck Typing is a type system used in dynamic languages.
- For example, Python, Perl, Ruby, PHP, Javascript, etc. where the type or the class of an object is less important than the method it defines.
- Using Duck Typing, we do not check types at all. Instead, we check for the presence of a given method or attribute.
- The name Duck Typing comes from the phrase:
- This polymorphic behaviour is a core idea behind Python which is also a dynamically typed language.
- This means that it performs type checking at run-time as opposed to statically typed languages (such as Java) that perform it during compile-time.

Ex:

```
x = 12000
print(type(x))

x = 'Dynamic Typing'
print(type(x))

x = [1, 2, 3, 4]
print(type(x))
```

Output:

```
<class 'int'>
<class 'str'>
<class 'list'>
```

Operator overloading

Python Operator Overloading:

- Python operators work for built-in classes. But the same operator behaves differently with different types. For example, the + operator will perform arithmetic addition on two numbers, merge two lists, or concatenate two strings.
- This feature in Python that allows the same operator to have different meaning according to the context is called operator overloading.
- Operator Overloading means giving extended meaning beyond their predefined operational meaning.
- For example operator + is used to add two integers as well as join two strings and merge two lists.
- It is achievable because '+' operator is overloaded by int class and str class.
- You might have noticed that the same built-in operator or function shows different behaviour for objects of different classes, this is called Operator Overloading.

- # Python program to show use of + operator for different purposes.

Ex:

```
print(1 + 2)

# concatenate two strings
print("Geeks"+"For")

# Product two numbers
print(3 * 4)

# Repeat the String
print("Geeks"*4)
```

Output:

```
3
GeeksFor
12
GeeksGeeksGeeksGeeks
```

- **How to overload the operators in Python?**
  - Consider that we have two objects which are a physical representation of a class (user-defined data type) and we have to add two objects with binary '+' operator it throws an error, because compiler don't know how to add two objects.
  - So we define a method for an operator and that process is called operator overloading.
  - We can overload all existing operators but we can't create a new operator. To perform operator overloading, Python provides some special function or magic function that is automatically invoked when it is associated with that particular operator.
  - For example, when we use + operator, the magic method __add__ is automatically invoked in which the operation for + operator is defined.

- **Overloading binary + operator in Python :**
  - When we use an operator on user defined data types then automatically a special function or magic function associated with that operator is invoked.
  - Changing the behavior of operator is as simple as changing the behavior of method or function. You define methods in your class and operators work according to that behavior defined in methods.
  - When we use + operator, the magic method __add__ is automatically invoked in which the operation for + operator is defined.
  - There by changing this magic method’s code, we can give extra meaning to the + operator.

Python Program illustrate how to overload an binary + operator

Ex:

```
class A:
        def __init__(self, a):
                self.a = a

        # adding two objects
        def __add__(self, o):
                return self.a + o.a
ob1 = A(1)
ob2 = A(2)
ob3 = A("Geeks")
ob4 = A("For")

print(ob1 + ob2)
print(ob3 + ob4)
```

Output:

```
3
GeeksFor
```