### **What is a Class?**

A Class is a logical grouping of data and functions. It gives the freedom to create data structures that contains arbitrary content and hence easily accessible.

For example, for any bank employee who want to fetch the customer details online would go to **customer class**, where all its attributes like transaction details, withdrawal and deposit details, outstanding debt, etc. would be listed out.

## **How to define Python classes**

To define class you need to consider following points

**Step 1)** In Python, classes are defined by the **"Class"**keyword

class myClass():

**Step 2)** Inside classes, you can define functions or methods that are part of this class

def method1 (self):

print "Guru99"

def method2 (self,someString):

print "Software Testing:" + someString

* Here we have defined method1 that prints "Guru99."
* Another method we have defined is method2 that prints "Software Testing"+ SomeString. SomeString is the variable supplied by the calling method

**Step 3)** Everything in a class is indented, just like the code in the function, loop, if statement, etc. Anything not indented is not in the class

**NOTE**: About using "self" in Python

* The self-argument refers to the object itself. Hence the use of the word self. So inside this method, self will refer to the specific instance of this object that's being operated on.
* Self is the name preferred by convention by Pythons to indicate the first parameter of instance methods in Python. It is part of the Python syntax to access members of objects

**Step 4)** To make an object of the class

c = myClass()

**Step 5)** To call a method in a class

c.method1()

c.method2(" Testing is fun")

* Notice that when we call the method1 or method2, we don't have to supply the self-keyword. That's automatically handled for us by the Python runtime.
* Python runtime will pass "self" value when you call an instance method on in instance, whether you provide it deliberately or not
* You just have to care about the non-self arguments

**Step 6)**Here is the complete code

# Example file for working with classes

class myClass():

def method1(self):

print("Guru99")

def method2(self,someString):

print("Software Testing:" + someString)

def main():

# exercise the class methods

c = myClass ()

c.method1()

c.method2(" Testing is fun")

if \_\_name\_\_== "\_\_main\_\_":

main()

### **Instance Objects**

Now what can we do with instance objects? The only operations understood by instance objects are attribute references. There are two kinds of valid attribute names: data attributes and methods.

data attributes correspond to “instance variables” in Smalltalk, and to “data members” in C++. Data attributes need not be declared; like local variables, they spring into existence when they are first assigned to. For example, if x is the instance of MyClass created above, the following piece of code will print the value 16, without leaving a trace:

x.counter = 1

**while** x.counter < 10:

x.counter = x.counter \* 2

print(x.counter)

**del** x.counter

The other kind of instance attribute reference is a method. A method is a function that “belongs to” an object. (In Python, the term method is not unique to class instances: other object types can have methods as well. For example, list objects have methods called append, insert, remove, sort, and so on. However, in the following discussion, we’ll use the term method exclusively to mean methods of class instance objects, unless explicitly stated otherwise.)

Valid method names of an instance object depend on its class. By definition, all attributes of a class that are function objects define corresponding methods of its instances. So in our example, x.f is a valid method reference, since MyClass.f is a function, but x.i is not, since MyClass.i is not. But x.fis not the same thing as MyClass.f — it is a method object, not a function object.

### **Class and Instance Variables**

Generally speaking, instance variables are for data unique to each instance and class variables are for attributes and methods shared by all instances of the class:

**class** **Dog**:

kind = 'canine' *# class variable shared by all instances*

**def** \_\_init\_\_(self, name):

self.name = name *# instance variable unique to each instance*

>>> d = Dog('Fido')

>>> e = Dog('Buddy')

>>> d.kind *# shared by all dogs*

'canine'

>>> e.kind *# shared by all dogs*

'canine'

>>> d.name *# unique to d*

'Fido'

>>> e.name *# unique to e*

'Buddy'

As discussed in [A Word About Names and Objects](https://docs.python.org/3/tutorial/classes.html#tut-object), shared data can have possibly surprising effects with involving [mutable](https://docs.python.org/3/glossary.html#term-mutable) objects such as lists and dictionaries. For example, the tricks list in the following code should not be used as a class variable because just a single list would be shared by all Doginstances:

**class** **Dog**:

tricks = [] *# mistaken use of a class variable*

**def** \_\_init\_\_(self, name):

self.name = name

**def** add\_trick(self, trick):

self.tricks.append(trick)

>>> d = Dog('Fido')

>>> e = Dog('Buddy')

>>> d.add\_trick('roll over')

>>> e.add\_trick('play dead')

>>> d.tricks *# unexpectedly shared by all dogs*

['roll over', 'play dead']

Correct design of the class should use an instance variable instead:

**class** **Dog**:

**def** \_\_init\_\_(self, name):

self.name = name

self.tricks = [] *# creates a new empty list for each dog*

**def** add\_trick(self, trick):

self.tricks.append(trick)

>>> d = Dog('Fido')

>>> e = Dog('Buddy')

>>> d.add\_trick('roll over')

>>> e.add\_trick('play dead')

>>> d.tricks

['roll over']

>>> e.tricks

['play dead']