Python - Object Oriented

Python has been an object-oriented language since it existed. Because of this, creating and using classes and objects are downright easy. This chapter helps you become an expert in using Python's object-oriented programming support.

If you do not have any previous experience with object-oriented (OO) programming, you may want to consult an introductory course on it or at least a tutorial of some sort so that you have a grasp of the basic concepts.

However, here is small introduction of Object-Oriented Programming (OOP) to bring you at speed −

Overview of OOP Terminology

- **Class** − A user-defined prototype for an object that defines a set of attributes that characterize any object of the class. The attributes are data members (class variables and instance variables) and methods, accessed via dot notation.
- **Class variable** − A variable that is shared by all instances of a class. Class variables are defined within a class but outside any of the class's methods. Class variables are not used as frequently as instance variables are.
- **Data member** − A class variable or instance variable that holds data associated with a class and its objects.
- **Function overloading** − The assignment of more than one behavior to a particular function. The operation performed varies by the types of objects or arguments involved.
- **Instance variable** − A variable that is defined inside a method and belongs only to the current instance of a class.
- **Inheritance** − The transfer of the characteristics of a class to other classes that are derived from it.
- **Instance** − An individual object of a certain class. An object obj that belongs to a class Circle, for example, is an instance of the class Circle.
- **Instantiation** − The creation of an instance of a class.
- **Method** − A special kind of function that is defined in a class definition.
- **Object** − A unique instance of a data structure that's defined by its class. An object comprises both data members (class variables and instance variables) and methods.
- **Operator overloading** − The assignment of more than one function to a particular operator.

Creating Classes

The *class* statement creates a new class definition. The name of the class immediately follows the keyword *class* followed by a colon as follows −

```
class ClassName:
  'Optional class documentation string'
  class_suite
```

- The class has a documentation string, which can be accessed via *ClassName.__doc__*.
- The *class_suite* consists of all the component statements defining class members, data attributes and functions.

Example

Following is the example of a simple Python class −

```
class Employee:

  'Common base class for all employees'

  empCount = 0



  def __init__(self, name, salary):

    self.name = name

    self.salary = salary

    Employee.empCount += 1



  def displayCount(self):

   print "Total Employee %d" % Employee.empCount



  def displayEmployee(self):

    print "Name : ", self.name,  ", Salary: ", self.salary
```

- The variable *empCount* is a class variable whose value is shared among all instances of a this class. This can be accessed as *Employee.empCount* 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.

```
"This would create first object of Employee class"
emp1 = Employee("Zara", 2000)
"This would create second object of Employee class"
emp2 = Employee("Manni", 5000)
```

Accessing Attributes

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

```
emp1.displayEmployee()
emp2.displayEmployee()
print "Total Employee %d" % Employee.empCount
```

Now, putting all the concepts together −

```
#!/usr/bin/python


class Employee:

  'Common base class for all employees'

  empCount = 0


  def __init__(self, name, salary):

    self.name = name

    self.salary = salary

    Employee.empCount += 1


  def displayCount(self):

   print "Total Employee %d" % Employee.empCount
  def displayEmployee(self):

     print "Name : ", self.name,  ", Salary: ", self.salary



"This would create first object of Employee class"

emp1 = Employee("Zara", 2000)

"This would create second object of Employee class"

emp2 = Employee("Manni", 5000)

emp1.displayEmployee()

emp2.displayEmployee()

print "Total Employee %d" % Employee.empCount
```

When the above code is executed, it produces the following result −

```
Name :  Zara ,Salary:  2000
Name :  Manni ,Salary:  5000
Total Employee 2
```

You can add, remove, or modify attributes of classes and objects at any time −

```
emp1.age = 7  # Add an 'age' attribute.
emp1.age = 8  # Modify 'age' attribute.
del emp1.age  # Delete 'age' attribute.
```

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.

For the above class let us try to access all these attributes −

```
#!/usr/bin/python


class Employee:
   'Common base class for all employees'
   empCount = 0


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


   def displayCount(self):
     print "Total Employee %d" % Employee.empCount


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


print "Employee.__doc__:", Employee.__doc__
print "Employee.__name__:", Employee.__name__
print "Employee.__module__:", Employee.__module__
print "Employee.__bases__:", Employee.__bases__

print "Employee.__dict__:", Employee.__dict__
```

When the above code is executed, it produces the following result −

```
Employee.__doc__: Common base class for all employees
Employee.__name__: Employee
Employee.__module__: __main__
Employee.__bases__: ()
Employee.__dict__: {'__module__': '__main__', 'displayCount':
<function displayCount at 0xb7c84994>, 'empCount': 2,
'displayEmployee': <function displayEmployee at 0xb7c8441c>,
'__doc__': 'Common base class for all employees',
'__init__': <function __init__ at 0xb7c846bc>}
```

Destroying Objects (Garbage Collection)

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.

Example

This __del__() destructor prints the class name of an instance that is about to be destroyed −

```
#!/usr/bin/python
```

```
class Point:
   def __init__( self, x=0, y=0):
      self.x = x
      self.y = y
   def __del__(self):
      class_name = self.__class__.__name__
      print class_name, "destroyed"

pt1 = Point()
pt2 = pt1
pt3 = pt1
print id(pt1), id(pt2), id(pt3) # prints the ids of the obejcts
del pt1
del pt2
del pt3
```

When the above code is executed, it produces following result −

```
3083401324 3083401324 3083401324
Point destroyed
```

Note − Ideally, you should define your classes in separate file, then you should import them in your main program file using *import* statement.

Class Inheritance

Instead of starting from scratch, you can create a class by deriving it from a preexisting class by listing the parent class in parentheses after the new class name.

The child class inherits the attributes of its parent class, and you can use those attributes as if they were defined in the child class. A child class can also override data members and methods from the parent.

Syntax

Derived classes are declared much like their parent class; however, a list of base classes to inherit from is given after the class name −

```
class SubClassName (ParentClass1[, ParentClass2, ...]):
   'Optional class documentation string'
   class_suite
```

Example

```
#!/usr/bin/python

class Parent:        # define parent class
   parentAttr = 100
   def __init__(self):
      print "Calling parent constructor"

   def parentMethod(self):
      print 'Calling parent method'

   def setAttr(self, attr):
      Parent.parentAttr = attr

   def getAttr(self):
      print "Parent attribute :", Parent.parentAttr

class Child(Parent): # define child class
   def __init__(self):
      print "Calling child constructor"
```

```
  def childMethod(self):
     print 'Calling child method'


c = Child()          # instance of child
c.childMethod()      # child calls its method
c.parentMethod()     # calls parent's method
c.setAttr(200)       # again call parent's method
c.getAttr()          # again call parent's method
```

When the above code is executed, it produces the following result −

```
Calling child constructor
Calling child method
Calling parent method
Parent attribute : 200
```

Similar way, you can drive a class from multiple parent classes as follows −

```
class A:        # define your class A
.....

class B:         # define your class B
.....

class C(A, B):   # subclass of A and B
.....
```

You can use issubclass() or isinstance() functions to check a relationships of two classes and instances.

- The **issubclass(sub, sup)** boolean function returns true if the given subclass **sub**is indeed a subclass of the superclass **sup**.
- The **isinstance(obj, Class)** boolean function returns true if *obj* is an instance of class *Class* or is an instance of a subclass of Class

Overriding Methods

You can always override your parent class methods. One reason for overriding parent's methods is because you may want special or different functionality in your subclass.

Example

```
#!/usr/bin/python
```

```
class Parent:        # define parent class
   def myMethod(self):
      print 'Calling parent method'

class Child(Parent): # define child class
   def myMethod(self):
      print 'Calling child method'

c = Child()          # instance of child
c.myMethod()         # child calls overridden method
```

When the above code is executed, it produces the following result −

```
Calling child method
```

Base Overloading Methods

Following table lists some generic functionality that you can override in your own classes −

Sr.No.	Method, Description & Sample Call
1	**__init__ (self [,args...])** Constructor (with any optional arguments) Sample Call : *obj = className(args)*
2	**__del__(self)** Destructor, deletes an object Sample Call : *del obj*
3	**__repr__(self)** Evaluable string representation Sample Call : *repr(obj)*

4	**__str__(self)** Printable string representation Sample Call : *str(obj)*
5	**__cmp__ (self, x)** Object comparison Sample Call : *cmp(obj, x)*

Overloading Operators

Suppose you have created a Vector class to represent two-dimensional vectors, what happens when you use the plus operator to add them? Most likely Python will yell at you.

You could, however, define the *__add__* method in your class to perform vector addition and then the plus operator would behave as per expectation −

Example

```
#!/usr/bin/python


class Vector:
   def __init__(self, a, b):
      self.a = a
      self.b = b


   def __str__(self):
      return 'Vector (%d, %d)' % (self.a, self.b)


   def __add__(self,other):
      return Vector(self.a + other.a, self.b + other.b)


v1 = Vector(2,10)
v2 = Vector(5,-2)
```

```
print v1 + v2
```

When the above code is executed, it produces the following result −

```
Vector(7,8)
```

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 are not be directly visible to outsiders.

Example

```
#!/usr/bin/python


class JustCounter:
   __secretCount = 0


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


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

When the above code is executed, it produces the following result −

```
1
2
Traceback (most recent call last):
   File "test.py", line 12, in <module>
      print counter.__secretCount
AttributeError: JustCounter instance 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 −

```
.........................
print counter._JustCounter__secretCount
```

When the above code is executed, it produces the following result −

```
1
2
2
```

Searching Algorithms

Searching is a very basic necessity when you store data in different data structures. The simplest approach is to go across every element in the data structure and match it with the value you are searching for. This is known as Linear search. It is inefficient and rarely used, but creating a program for it gives an idea about how we can implement some advanced search algorithms.

Linear Search

In this type of search, a sequential search is made over all items one by one. Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data structure.

```
def linear_search(values, search_for):
    search_at = 0
    search_res = False

# Match the value with each data element
    while search_at < len(values) and search_res is False:
        if values[search_at] == search_for:
            search_res = True
        else:
            search_at = search_at + 1
    return search_res


l = [64, 34, 25, 12, 22, 11, 90]
print(linear_search(l, 12))
print(linear_search(l, 91))
```

When the above code is executed, it produces the following result −

```
True
False
```

Interpolation Search

This search algorithm works on the probing position of the required value. For this algorithm to work properly, the data collection should be in a sorted form and equally distributed. Initially, the probe position is the position of the middle most item of the collection.If a match occurs, then the index of the item is returned. If the middle item is greater than the item, then the probe position is again calculated in the sub-array to the right of the middle item. Otherwise, the item is searched in the subarray to the left of the middle item. This process continues on the sub-array as well until the size of subarray reduces to zero.

There is a specific formula to calculate the middle position which is indicated in the program below.

```


def intpolsearch(values,x ):
    idx0 = 0
    idxn = (len(values) - 1)


    while idx0 <= idxn and x >= values[idx0] and x <= values[idxn]:


# Find the mid point
            mid = idx0 +\
              int(((float(idxn - idx0)/( values[idxn] - values[idx0]))
                    * ( x - values[idx0])))


# Compare the value at mid point with search value
        if values[mid] == x:
            return "Found "+str(x)+" at index "+str(mid)


        if values[mid] < x:
            idx0 = mid + 1
    return "Searched element not in the list"



l = [2, 6, 11, 19, 27, 31, 45, 121]
print(intpolsearch(l, 2))
```

When the above code is executed, it produces the following result −

```
Found 2 at index 0
```

Sorting Algorithms

Sorting refers to arranging data in a particular format. Sorting algorithm specifies the way to arrange data in a particular order. Most common orders are in numerical or lexicographical order.

The importance of sorting lies in the fact that data searching can be optimized to a very high level, if data is stored in a sorted manner. Sorting is also used to represent data in more readable formats. Below we see five such implementations of sorting in python.

- Bubble Sort
- Merge Sort

- Insertion Sort
- Shell Sort
- Selection Sort

Bubble Sort

It is a comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order.

```
def bubblesort(list):


# Swap the elements to arrange in order

    for iter_num in range(len(list)-1,0,-1):

        for idx in range(iter_num):

            if list[idx]>list[idx+1]:

                temp = list[idx]

                list[idx] = list[idx+1]

                list[idx+1] = temp




list = [19,2,31,45,6,11,121,27]

bubblesort(list)

print(list)
```

When the above code is executed, it produces the following result –

```
[2, 6, 11, 19, 27, 31, 45, 121]
```

Merge Sort

Merge sort first divides the array into equal halves and then combines them in a sorted manner.

```
def merge_sort(unsorted_list):

    if len(unsorted_list) <= 1:

        return unsorted_list
```

```
# Find the middle point and devide it
    middle = len(unsorted_list) // 2
    left_list = unsorted_list[:middle]
    right_list = unsorted_list[middle:]

    left_list = merge_sort(left_list)
    right_list = merge_sort(right_list)
    return list(merge(left_list, right_list))

# Merge the sorted halves

def merge(left_half,right_half):

    res = []
    while len(left_half) != 0 and len(right_half) != 0:
        if left_half[0] < right_half[0]:
            res.append(left_half[0])
            left_half.remove(left_half[0])
        else:
            res.append(right_half[0])
            right_half.remove(right_half[0])
    if len(left_half) == 0:
        res = res + right_half
    else:
        res = res + left_half
    return res
```

```
unsorted_list = [64, 34, 25, 12, 22, 11, 90]


print(merge_sort(unsorted_list))
```

When the above code is executed, it produces the following result −

```
[11, 12, 22, 25, 34, 64, 90]
```

Insertion Sort

Insertion sort involves finding the right place for a given element in a sorted list. So in beginning we compare the first two elements and sort them by comparing them. Then we pick the third element and find its proper position among the previous two sorted elements. This way we gradually go on adding more elements to the already sorted list by putting them in their proper position.

```
def insertion_sort(InputList):
    for i in range(1, len(InputList)):
        j = i-1
        nxt_element = InputList[i]
# Compare the current element with next one

        while (InputList[j] > nxt_element) and (j >= 0):
            InputList[j+1] = InputList[j]
            j=j-1
        InputList[j+1] = nxt_element

list = [19,2,31,45,30,11,121,27]
insertion_sort(list)
print(list)
```

When the above code is executed, it produces the following result −

```
[2, 11, 19, 27, 30, 31, 45, 121]
```

Shell Sort

Shell Sort involves sorting elements which are away from ech other. We sort a large sublist of a given list and go on reducing the size of the list until all elements are sorted. The below program finds the gap by equating it to half of the length of the list size and then starts sorting all elements in it. Then we keep resetting the gap until the entire list is sorted.

```
def shellSort(input_list):


    gap = len(input_list) // 2
    while gap > 0:


        for i in range(gap, len(input_list)):
            temp = input_list[i]
            j = i
# Sort the sub list for this gap


            while j >= gap and input_list[j - gap] > temp:
                input_list[j] = input_list[j - gap]
                j = j-gap
            input_list[j] = temp


# Reduce the gap for the next element


        gap = gap//2


list = [19,2,31,45,30,11,121,27]


shellSort(list)
print(list)
```

When the above code is executed, it produces the following result −

```
[2, 11, 19, 27, 30, 31, 45, 121]
```

Selection Sort

In selection sort we start by finding the minimum value in a given list and move it to a sorted list. Then we repeat the process for each of the remaining elements in the unsorted list. The next element entering the sorted list is compared with the existing elements and placed at its correct position. So at the end all the elements from the unsorted list are sorted.

```
def selection_sort(input_list):


    for idx in range(len(input_list)):


        min_idx = idx
        for j in range( idx +1, len(input_list)):
            if input_list[min_idx] > input_list[j]:
                min_idx = j
# Swap the minimum value with the compared value


        input_list[idx], input_list[min_idx] = input_list[min_idx], input_list[idx]




l = [19,2,31,45,30,11,121,27]
selection_sort(l)
print(l)
```

When the above code is executed, it produces the following result −

```
[2, 11, 19, 27, 30, 31, 45, 121]
```

Hash Table

Hash tables are a type of data structure in which the address or the index value of the data element is generated from a hash function. That makes accessing the data faster as the index value behaves as a key for the data value. In other words Hash table stores key-value pairs but the key is generated through a hashing function.

So the search and insertion function of a data element becomes much faster as the key values themselves become the index of the array which stores the data.

In Python, the Dictionary data types represent the implementation of hash tables. The Keys in the dictionary satisfy the following requirements.

- The keys of the dictionary are hashable i.e. the are generated by hashing function which generates unique result for each unique value supplied to the hash function.
- The order of data elements in a dictionary is not fixed.

So we see the implementation of hash table by using the dictionary data types as below.

Accessing Values in Dictionary

To access dictionary elements, you can use the familiar square brackets along with the key to obtain its value.

```
# Declare a dictionary

dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}



# Accessing the dictionary with its key

print "dict['Name']: ", dict['Name']

print "dict['Age']: ", dict['Age']
```

When the above code is executed, it produces the following result −

```
dict['Name']:  Zara
dict['Age']:  7
```

Updating Dictionary

You can update a dictionary by adding a new entry or a key-value pair, modifying an existing entry, or deleting an existing entry as shown below in the simple example −

```
# Declare a dictionary

dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}

dict['Age'] = 8; # update existing entry

dict['School'] = "DPS School"; # Add new entry

print "dict['Age']: ", dict['Age']

print "dict['School']: ", dict['School']
```

When the above code is executed, it produces the following result −

```
When the above code is executed, it produces the following result −
dict['Age']:  8
dict['School']:  DPS School
```

Delete Dictionary Elements

You can either remove individual dictionary elements or clear the entire contents of a dictionary. You can also delete entire dictionary in a single operation. To explicitly remove an entire dictionary, just use the del statement. −

```
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}

del dict['Name']; # remove entry with key 'Name'

dict.clear();     # remove all entries in dict

del dict ;        # delete entire dictionary



print "dict['Age']: ", dict['Age']

print "dict['School']: ", dict['School']
```

This produces the following result. Note that an exception is raised because after del dict dictionary does not exist any more −

```
dict['Age']:
Traceback (most recent call last):
  File "test.py", line 8, in
    print "dict['Age']: ", dict['Age'];
TypeError: 'type' object is unsubscriptable
```

Matplotlib

Matplotlib is a python library used to create 2D graphs and plots by using python scripts. It has a module named pyplot which makes things easy for plotting by providing feature to control line styles, font properties, formatting axes etc. It supports a very wide variety of graphs and plots namely - histogram, bar charts, power spectra, error charts etc. It is used along with NumPy to provide an environment that is an effective open source alternative for MatLab. It can also be used with graphics toolkits like PyQt and wxPython.

Conventionally, the package is imported into the Python script by adding the following statement −

```
from matplotlib import pyplot as plt
```

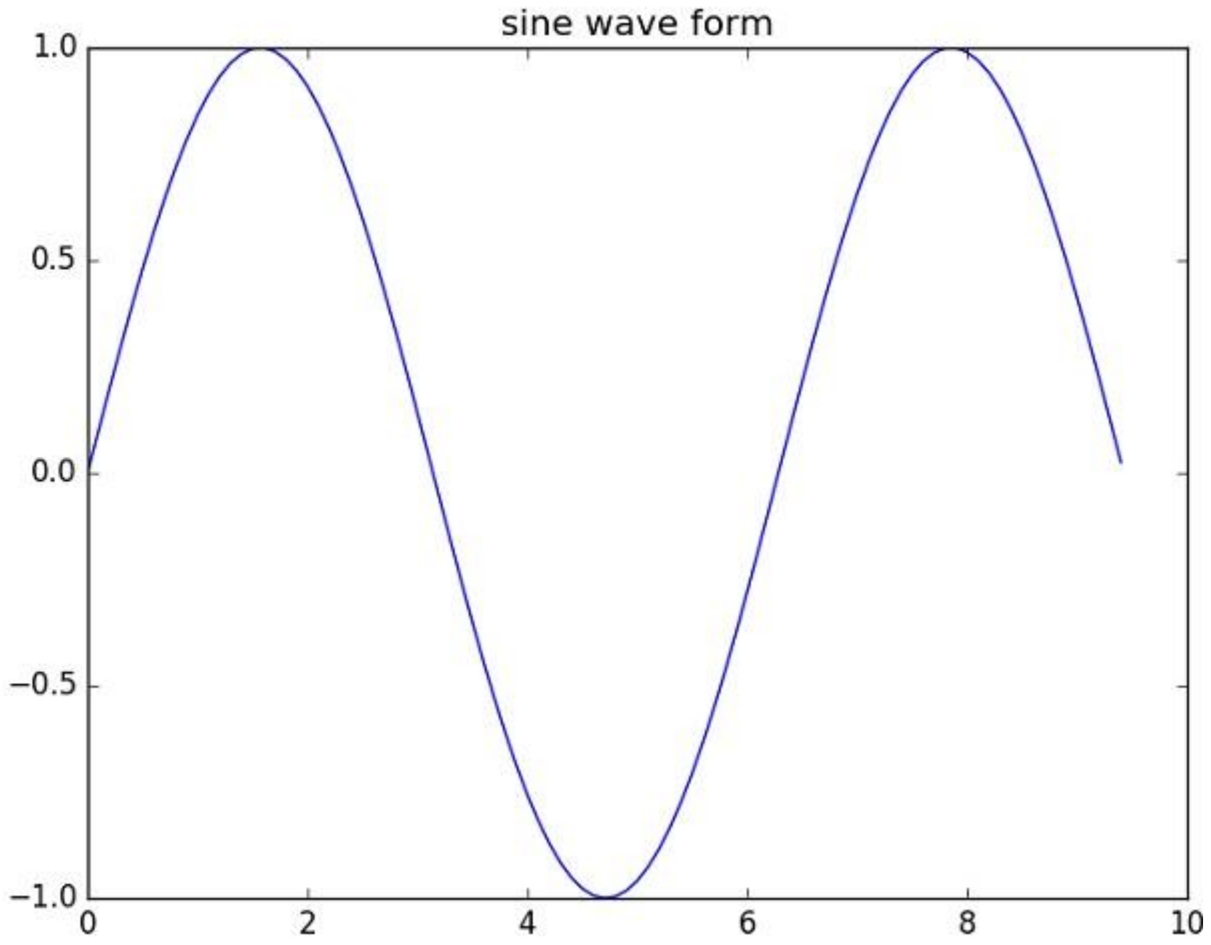
Matplotlib Example

The following script produces the **sine wave plot** using matplotlib.

Example

```
import numpy as np

import matplotlib.pyplot as plt


# Compute the x and y coordinates for points on a sine curve

x = np.arange(0, 3 * np.pi, 0.1)

y = np.sin(x)

plt.title("sine wave form")


# Plot the points using matplotlib

plt.plot(x, y)

plt.show()
```

Its **output** is as follows −



We will see lots of examples on using Matplotlib library of python in Data science work in the next chapters.

Plotting Using PyLab

PyLab is a Python standard library module that provides many of the facilities of MATLAB, "a high- level technical computing language and interactive environment for algorithm development, data visualization, data analysis, and numeric computation."

1.A simple example

(1) Simple

Let's start with a simple example that uses **pylab.plot** to produce two plots. Executing:

```
import pylab
pylab.figure(1)
pylab.plot([1,2,3,4,5],[1,7,3,5,12])
pylab.show()
```

will cause a window to appear on your computer monitor. Its exact appearance may depend on your Python environment, but it will look similar to Figure 11.1 (which was produced using **Anaconda**). If you run this code with the default parameter settings of most installations of PyLab, the line will probably not be as thick as the line in Figure 11.1. We have used nonstandard default values for line width and font sizes so that the figures will reproduce better in black and white. We discuss how this is done later in this section.

1. Sequences of the same length

The bar at the top of the plot contains the name of the window, in this case "Figure 1." The middle section of the window contains the plot generated by the invocation of pylab.plot. The two parameters of pylab.**plot must be sequences of the same length**. The first specifies the x-coordinates of the points to be plotted, and the second specifies the y-coordinates. Together, they provide a sequence of four <x, y> coordinate pairs,[(1,1), (2,7), (3,3), (4,5)]. These are plotted in order. As each point is plotted, a line is drawn connecting it to the previous point.

2. The problem of pylab.show()

The final line of code, **pylab.show()**, causes the window to appear on the computer screen. In some Python environments, if that line were not present, the figure would still have been produced, but it would not have been displayed. This is not as silly as it at first sounds, since one might well choose to write a figure directly to a file, as we will do later, rather than display it on the screen.

3. Introduction of the Buttons:

The bar at the top of the window contains a number of push buttons. The rightmost button pops up a window with options that can be used to adjust various aspects of the figure. The next button to the left is used to write the plot to a file. The button to the left of that is used to adjust the appearance of the plot in the window. The next two buttons are used for zooming and panning. The two buttons that look like arrows are used to see previous views (like the forward and backward arrows of a Web browser). And the button on the extreme left is used to restore the figure to its original appearance after you are done playing with other buttons.

(2) The multiple figures

It is possible to produce multiple figures and to write them to files. These files can have any name you like, but they will all have the file extension .png. The file extension .png indicates that the file is in the Portable Networks Graphics format. This is a public domain standard for representing images.

multiple figures

```
pylab.figure(1) #create figure 1
pylab.plot([1,2,3,4], [1,2,3,4]) #draw on figure 1
pylab.figure(2) #create figure 2
pylab.plot([1,4,2,3], [5,6,7,8]) #draw on figure 2
pylab.savefig('Figure-Addie') #save figure 2
pylab.figure(1) #go back to working on figure 1
pylab.plot([5,6,10,3]) #draw again on figure 1
pylab.savefig('Figure-Jane') #save figure 1
```

1. One argument

Observe that the last call to **pylab.plot** is passed only **one argument**. This argument supplies the y values. ==The corresponding x values default to the sequence yielded by range(len([5, 6, 10, 3])), which is why they range from 0 to 3 in this case.

2. "Current figure."

PyLab has a notion of **"current figure."** Executing pylab.figure(x) sets the current figure to the figure numbered x. Subsequently executed calls of plotting functions implicitly refer to that figure until another invocation of pylab.figure occurs. This explains why the figure written to the file Figure-Addie.png was the second figure created.

2. Compound interest

(1) Simple

values.append(principal

******Let's look at another example. The code:**

```
principal = 10000 #initial investment
interestRate = 0.05
years = 20
values = []
for i in range(years + 1):
    values.append(principal)
    principal += principal*interestRate
pylab.plot(values)
pylab.savefig('FigurePlottingCompoundGrowth')
```

produces the plot on the left in Figure 11.3.

(2) informative Titles

If we look at the code, we can deduce that this is a plot showing the growth of an initial investment of $10,000 at an annually compounded interest rate of 5%. However, this cannot be easily inferred by looking only at the plot itself. That’s a bad thing. All plots should have informative titles, and all axes should be labeled. If we add to the end of our code the lines

The xlabel,ylabel,title

```
pylab.title('5% Growth, Compounded Annually')
pylab.xlabel('Years of Compounding')
pylab.ylabel('Value of Principal ($)')
```

we get the plot on the right in Figure 11.3.

Figure 11.3

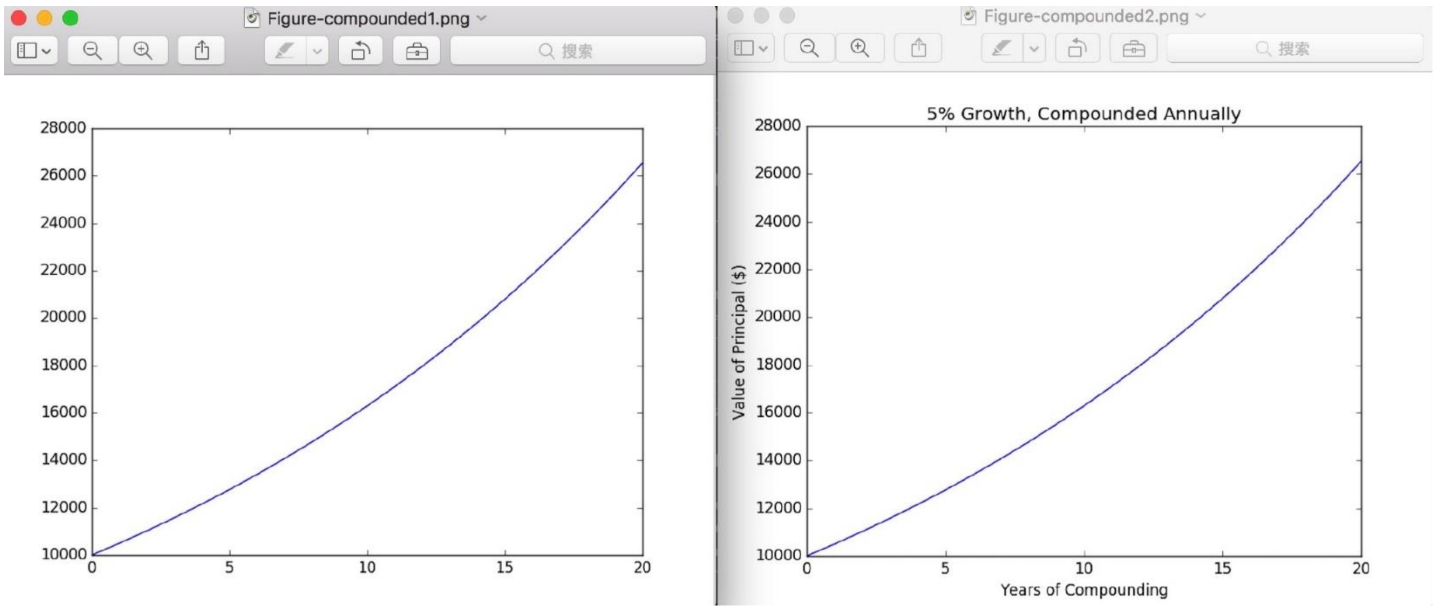


Figure 11.3

(3) Optional argument

For every plotted curve, there is an optional argument that is a format string indicating the color and line type of the plot. The letters and symbols of the format string are derived from those used in MATLAB, and are composed of a color indicator followed by an optional line-style indicator. The default format string is **'b-'**, which produces a solid blue line. To plot the the growth in principal with black circles, one would replace the call

pylab.plot(values) by pylab.plot(values, 'ko',)

which produces the plot in Figure 11.4. For a complete list of color and line-style indicators, see

Figure 11.4 Another plot of compound growth

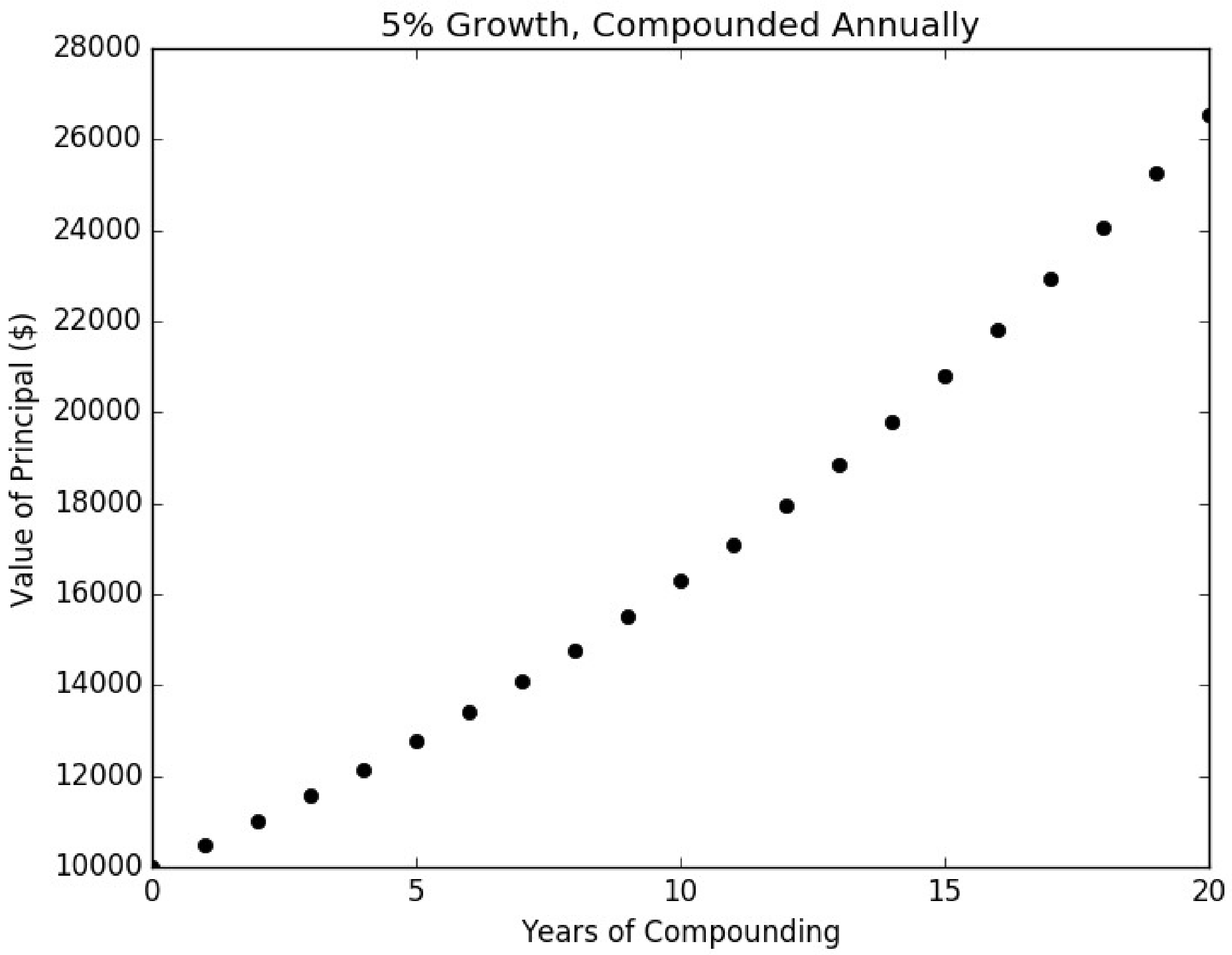


Figure 11.5 Strange-looking plot

(4) Change the type size and line width

It is also possible to change the type size and line width used in plots. This can be done using keyword arguments in individual calls to functions. E.g., the code:

```
principal = 10000 #initial
investment interestRate = 0.05
years = 20
values = []
for i in range(years + 1):
    values.append(principal)
    principal += principal*interestRate
pylab.plot(values, linewidth = 30)
pylab.title('5% Growth, Compounded Annually', fontsize = 'xx-large')
pylab.xlabel('Years of Compounding', fontsize = 'x- small')
pylab.ylabel('Value of Principal ($)')
```

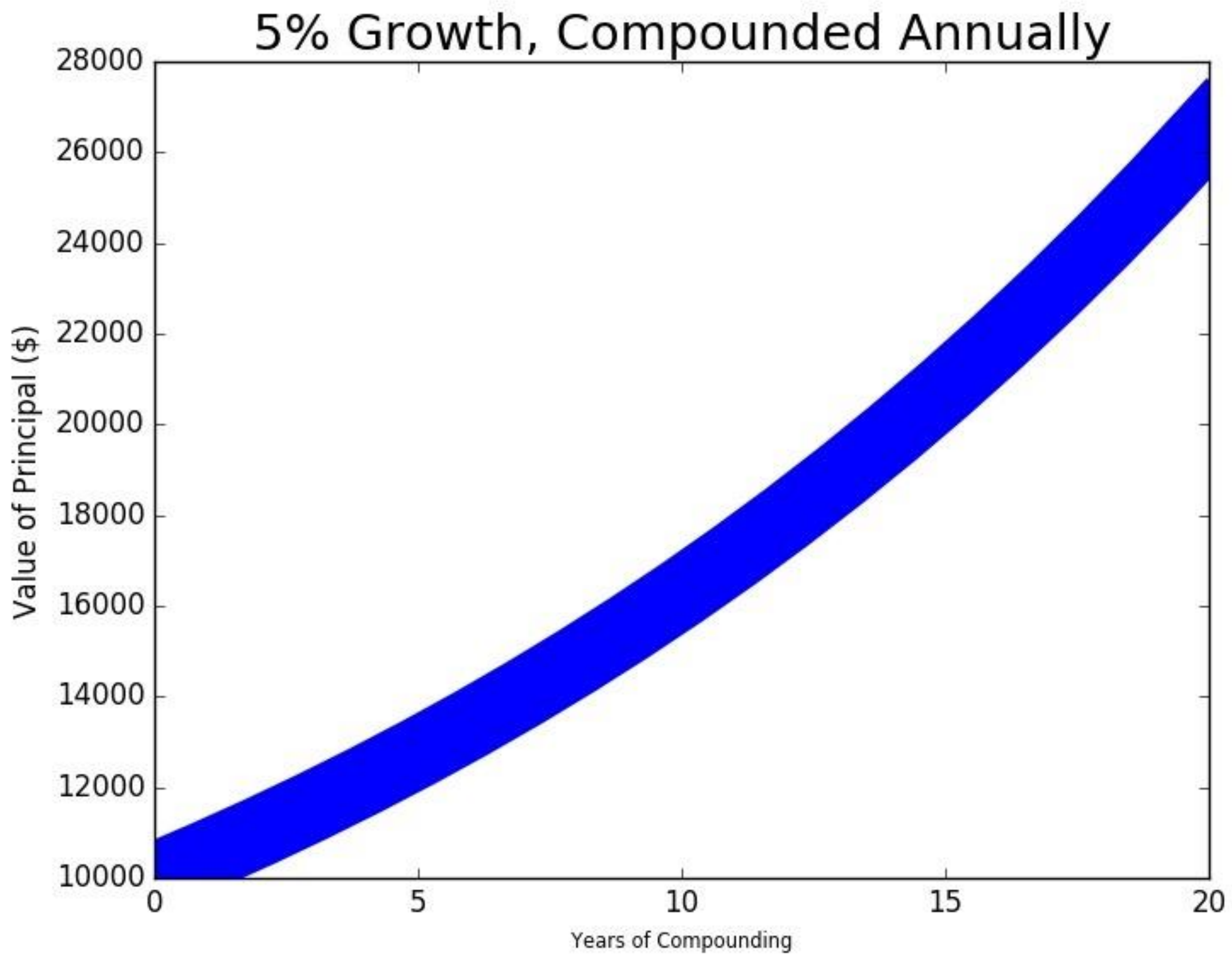


Figure 11.5 Strange-looking plot

3. Change the default values:“rc settings.”

It is also possible to change the default values, which are known as **“rc settings.”** (The name “rc” is derived from the .rc file extension used for runtime configuration files in Unix.) These values are stored in a dictionary-like variable that can be accessed via the name pylab.rcParams. So, for example, you can set the default line width to 6 points by executing the code

pylab.rcParams['lines.linewidth'] = 6.

There are an enormous number rcParams settings. A complete list can be found at

If you don’t want to worry about customizing individual parameters,there are pre-defined style sheets. A description of these can be found at

The values used in most of the remaining examples in this book were set with the code

```
#set line width
pylab.rcParams['lines.linewidth'] = 4
```

```
#set font size for titles
pylab.rcParams['axes.titlesize'] = 20
#set font size for labels on axes
pylab.rcParams['axes.labelsize'] = 20
#set size of numbers on x-axis
pylab.rcParams['xtick.labelsize'] = 16
#set size of numbers on y-axis
pylab.rcParams['ytick.labelsize'] = 16
#set size of ticks on x-axis
pylab.rcParams['xtick.major.size'] = 7
#set size of ticks on y-axis
pylab.rcParams['ytick.major.size'] = 7
#set size of markers, e.g., circles representing points
pylab.rcParams['lines.markersize'] = 10
#set number of times marker is shown when displaying legend
pylab.rcParams['legend.numpoints'] = 1
```

If you are viewing plots on a color display, you will have little reason to customize these settings. We customized the settings we used so that it would be easier to read the plots when we shrank them and converted them to black and white.

Tkinter

Python provides various options for developing graphical user interfaces (GUIs). Most important are listed below.

- **Tkinter** − Tkinter is the Python interface to the Tk GUI toolkit shipped with Python. We would look this option in this chapter.
- **wxPython** − This is an open-source Python interface for wxWindows http://wxpython.org.
- **JPython** − JPython is a Python port for Java which gives Python scripts seamless access to Java class libraries on the local machine http://www.jython.org.

There are many other interfaces available, which you can find them on the net.

Tkinter Programming

Tkinter is the standard GUI library for Python. Python when combined with Tkinter provides a fast and easy way to create GUI applications. Tkinter provides a powerful object-oriented interface to the Tk GUI toolkit.

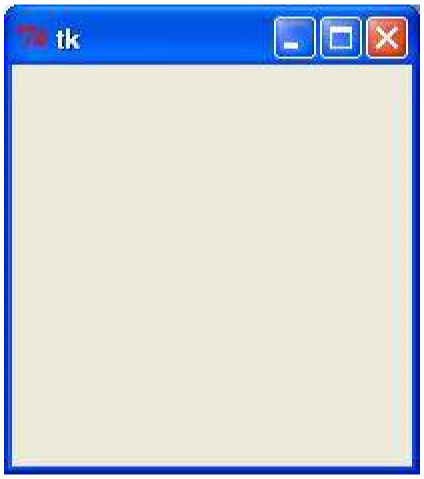
Creating a GUI application using Tkinter is an easy task. All you need to do is perform the following steps −

- Import the *Tkinter* module.
- Create the GUI application main window.
- Add one or more of the above-mentioned widgets to the GUI application.
- Enter the main event loop to take action against each event triggered by the user.

Example

```
#!/usr/bin/python

import Tkinter
top = Tkinter.Tk()
# Code to add widgets will go here...
top.mainloop()
```

This would create a following window −



Tkinter Widgets

Tkinter provides various controls, such as buttons, labels and text boxes used in a GUI application. These controls are commonly called widgets.

There are currently 15 types of widgets in Tkinter. We present these widgets as well as a brief description in the following table −

Sr.No.	Operator & Description
1	**Button** The Button widget is used to display buttons in your application.
2	**Canvas**

	The Canvas widget is used to draw shapes, such as lines, ovals, polygons and rectangles, in your application.
3	**Checkbutton** The Checkbutton widget is used to display a number of options as checkboxes. The user can select multiple options at a time.
4	**Entry** The Entry widget is used to display a single-line text field for accepting values from a user.
5	**Frame** The Frame widget is used as a container widget to organize other widgets.
6	**Label** The Label widget is used to provide a single-line caption for other widgets. It can also contain images.
7	**Listbox** The Listbox widget is used to provide a list of options to a user.
8	**Menubutton** The Menubutton widget is used to display menus in your application.
9	**Menu** The Menu widget is used to provide various commands to a user. These commands are contained inside Menubutton.
10	**Message** The Message widget is used to display multiline text fields for accepting values from a user.
11	**Radiobutton** The Radiobutton widget is used to display a number of options as radio buttons. The user can select only one option at a time.

12	**Scale** The Scale widget is used to provide a slider widget.
13	**Scrollbar** The Scrollbar widget is used to add scrolling capability to various widgets, such as list boxes.
14	**Text** The Text widget is used to display text in multiple lines.
15	**Toplevel** The Toplevel widget is used to provide a separate window container.
16	**Spinbox** The Spinbox widget is a variant of the standard Tkinter Entry widget, which can be used to select from a fixed number of values.
17	**PanedWindow** A PanedWindow is a container widget that may contain any number of panes, arranged horizontally or vertically.
18	**LabelFrame** A labelframe is a simple container widget. Its primary purpose is to act as a spacer or container for complex window layouts.
19	**tkMessageBox** This module is used to display message boxes in your applications.

Let us study these widgets in detail −

Standard attributes

Let us take a look at how some of their common attributes.such as sizes, colors and fonts are specified.

- Dimensions
- Colors

- Fonts
- Anchors
- Relief styles
- Bitmaps
- Cursors

Let us study them briefly −

Geometry Management

All Tkinter widgets have access to specific geometry management methods, which have the purpose of organizing widgets throughout the parent widget area. Tkinter exposes the following geometry manager classes: pack, grid, and place.

- The *pack()* Method − This geometry manager organizes widgets in blocks before placing them in the parent widget.
- The *grid()* Method − This geometry manager organizes widgets in a table-like structure in the parent widget.
- The *place()* Method − This geometry manager organizes widgets by placing them in a specific position in the parent widget.