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.

Example

```
#!/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__
```

```
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 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.

Destroying Objects

- An object's reference count increases when it's 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>
```

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
```

```
3083401324 3083401324 3083401324
Point destroyed
```

Class Inheritance

- You can create a class by deriving it from a preexisting class by listing the parent class in parentheses after the new class name.

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

- The child class inherits the attributes of its parent class
  - 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.

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
```

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

Multiple Inheritance

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

class B:           # define your calss 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.

```
#!/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
```

```
Calling child method
```

Base Overloading Methods

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

SN	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 : *dell obj*
3	**__repr__(self)** Evaluatable 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

- You could define the __*add*__ method in your class to perform vector addition and then the plus operator would behave as per expectation

```
#!/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
```

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

Data Hiding

- An object's attributes may or may not be visible outside the class definition.
- For these cases, you can name attributes with a double underscore prefix, and those attributes will not be directly visible to outsiders.

```
#!/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
```

```
1
2
Traceback (most recent call last):
  File "test.py", line 12, in <module>
    print counter.__secretCount
AttributeError: JustCounter instance has no attribute '__secretCount'
```

Data Hiding

- 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 would work for you:

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

```
1
2
2
```

Python Modules

- A module allows you to logically organize your Python code.
- Grouping related code into a module makes the code easier to understand and use.
- A module is a Python object with arbitrarily named attributes that you can bind and reference.
- Simply, a module is a file consisting of Python code.
- A module can define functions, classes and variables. A module can also include runnable code.

The *import* Statement

- You can use any Python source file as a module by executing an import statement in some other Python source file.
  The *import* has the following syntax:

```
import module1[, module2[,... moduleN]
```

- When the interpreter encounters an import statement, it imports the module if the module is present in the search path.
- A search path is a list of directories that the interpreter searches before importing a module.

Example

- To import the module hello.py, you need to put the following command at the top of the script:

```
#!/usr/bin/python

# Import module support
import support

# Now you can call defined function that module as follows
support.print_func("Zara")
```

```
Hello : Zara
```

The *from...import* Statement

- Python's *from* statement lets you import specific attributes from a module into the current namespace.
- The *from...import* has the following syntax:

```
from modname import name1[, name2[, ... nameN]]
```

- For example, to import the function fibonacci from the module fib, use the following statement:

```
from fib import fibonacci
```

The *from...import* * Statement:

- It is also possible to import all names from a module into the current namespace by using the following import statement:

```
from modname import *
```

Locating Modules:

- When you import a module, the Python interpreter searches for the module in the following sequences:
  - The current directory.
  - If the module isn't found, Python then searches each directory in the shell variable PYTHONPATH.
  - If all else fails, Python checks the default path.
    - On UNIX, this default path is normally /usr/local/lib/python/.

The *PYTHONPATH* Variable:

- The PYTHONPATH is an environment variable, consisting of a list of directories.
- The syntax of PYTHONPATH is the same as that of the shell variable PATH.
- Here is a typical PYTHONPATH from a Windows system:
  - set PYTHONPATH=c:\python27\lib;
- Here is a typical PYTHONPATH from a UNIX system:
  - set PYTHONPATH=/usr/local/lib/python

Namespaces and Scoping

- Variables are names (identifiers) that map to objects.
- A ***namespace*** is a dictionary of variable names (keys) and their corresponding objects (values).
- A Python statement can access variables in a *local namespace* and in the *global namespace*.
  - If a local and a global variable have the same name, the local variable shadows the global variable.
- Each function has its own local namespace.
  - Class methods follow the same scoping rule as ordinary functions.
- Python makes educated guesses on whether variables are local or global.
  - It assumes that any variable assigned a value in a function is local.

Namespaces and Scoping

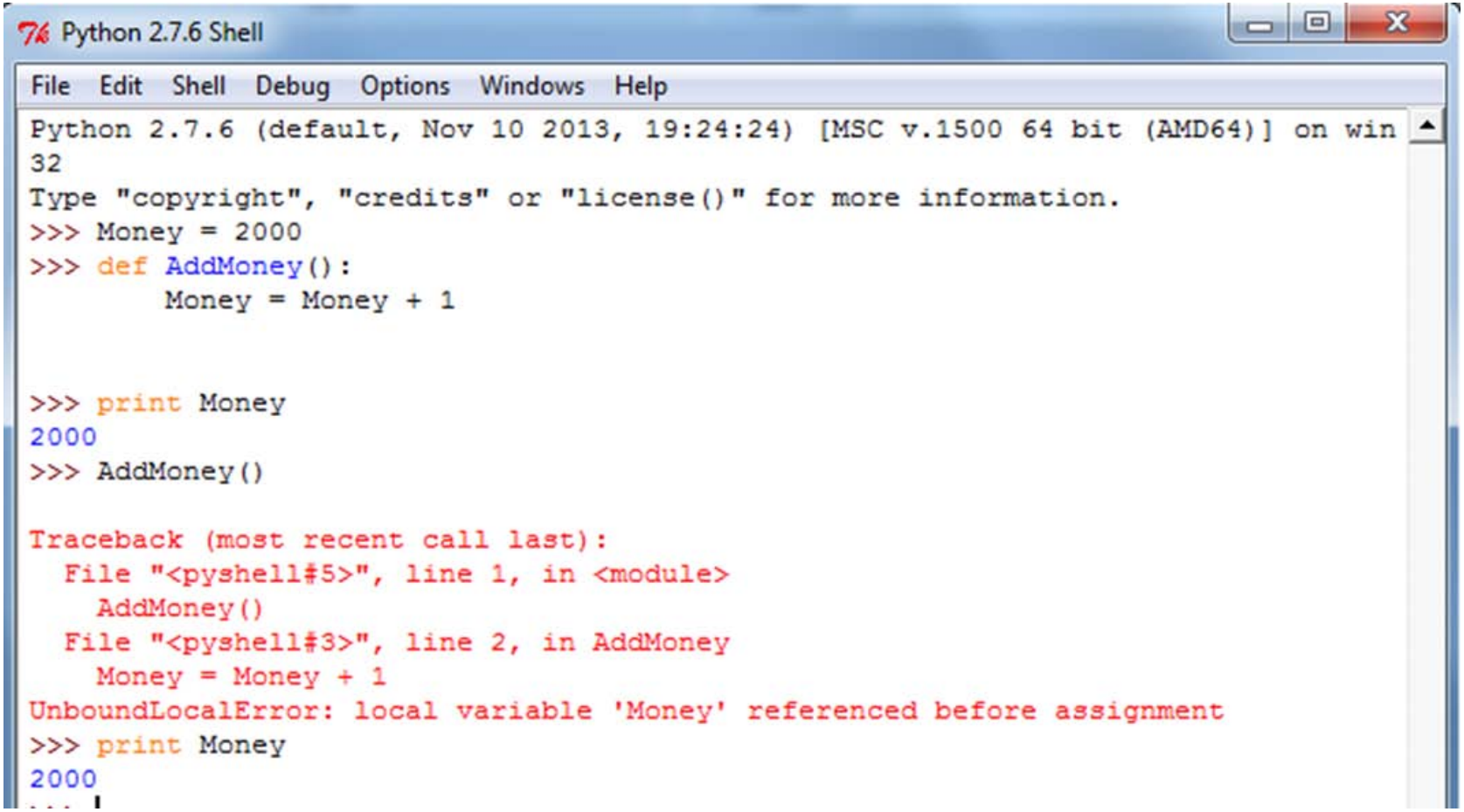
- Therefore, in order to assign a value to a global variable within a function, you must first use the global statement.
- The statement *global VarName* tells Python that VarName is a global variable.
  - Python stops searching the local namespace for the variable.

```
#!/usr/bin/python

Money = 2000
def AddMoney():
   # Uncomment the following line to fix the code:
   # global Money
   Money = Money + 1

print Money
AddMoney()
print Money
```

Results


Python 2.7.6 Shell
File Edit Shell Debug Options Windows Help
Python 2.7.6 (default, Nov 10 2013, 19:24:24) [MSC v.1500 64 bit (AMD64)] on win
32
Type "copyright", "credits" or "license()" for more information.
>>> Money = 2000
>>> def AddMoney():
Money = Money + 1
>>> print Money
2000
>>> AddMoney()
Traceback (most recent call last):
File "<pyshell#5>", line 1, in <module>
AddMoney()
File "<pyshell#3>", line 2, in AddMoney
Money = Money + 1
UnboundLocalError: local variable 'Money' referenced before assignment
>>> print Money
2000

Example

```
# sample.py
myGlobal = 5

def func1():
    myGlobal = 42

def func2():
    print myGlobal

func1()
func2()
```

5

```
def func1():
    global myGlobal
    myGlobal = 42
```

42

The dir( ) Function

- The dir() built-in function returns a sorted list of strings containing the names defined by a module.
- The list contains the names of all the modules, variables and functions that are defined in a module.
- Here, the special string variable *__name__* is the module's name, and *__file__* is the filename from which the module was loaded.

```
#!/usr/bin/python

# Import built-in module math
import math

content = dir(math)

print content;
```

```
['__doc__', '__file__', '__name__', 'acos', 'asin', 'atan',
'atan2', 'ceil', 'cos', 'cosh', 'degrees', 'e', 'exp',
'fabs', 'floor', 'fmod', 'frexp', 'hypot', 'ldexp', 'log',
'log10', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh',
'sqrt', 'tan', 'tanh']
```

The *globals()* and *locals()* Functions

- The *globals()* and *locals()* functions can be used to return the names in the global and local namespaces depending on the location from where they are called.
- If locals() is called from within a function, it will return all the names that can be accessed locally from that function.
- If globals() is called from within a function, it will return all the names that can be accessed globally from that function.
- The return type of both these functions is dictionary. Therefore, names can be extracted using the keys() function.

Packages in Python

- A package is a hierarchical file directory structure that defines a single Python application environment that consists of modules and subpackages and sub-subpackages, and so on.
- Consider a file *Pots.py* available in *Phone* directory
- We have another two files having different functions with the same name as above:
  - *Phone/Isdn.py* file having function Isdn()
  - *Phone/G3.py* file having function G3()
- Now, create one more file __init__.py in *Phone* directory:
  - Phone/__init__.py

Packages in Python

- To make all of your functions available when you've imported Phone, you need to put explicit import statements in __init__.py as follows:
  - from Pots import Pots
  - from Isdn import Isdn
  - from G3 import G3

```
#!/usr/bin/python

# Now import your Phone Package.
import Phone

Phone.Pots()
Phone.Isdn()
Phone.G3()
```

```
I'm Pots Phone
I'm 3G Phone
I'm ISDN Phone
```

Python GUI Programming (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.
- **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.

Tkinter Programming

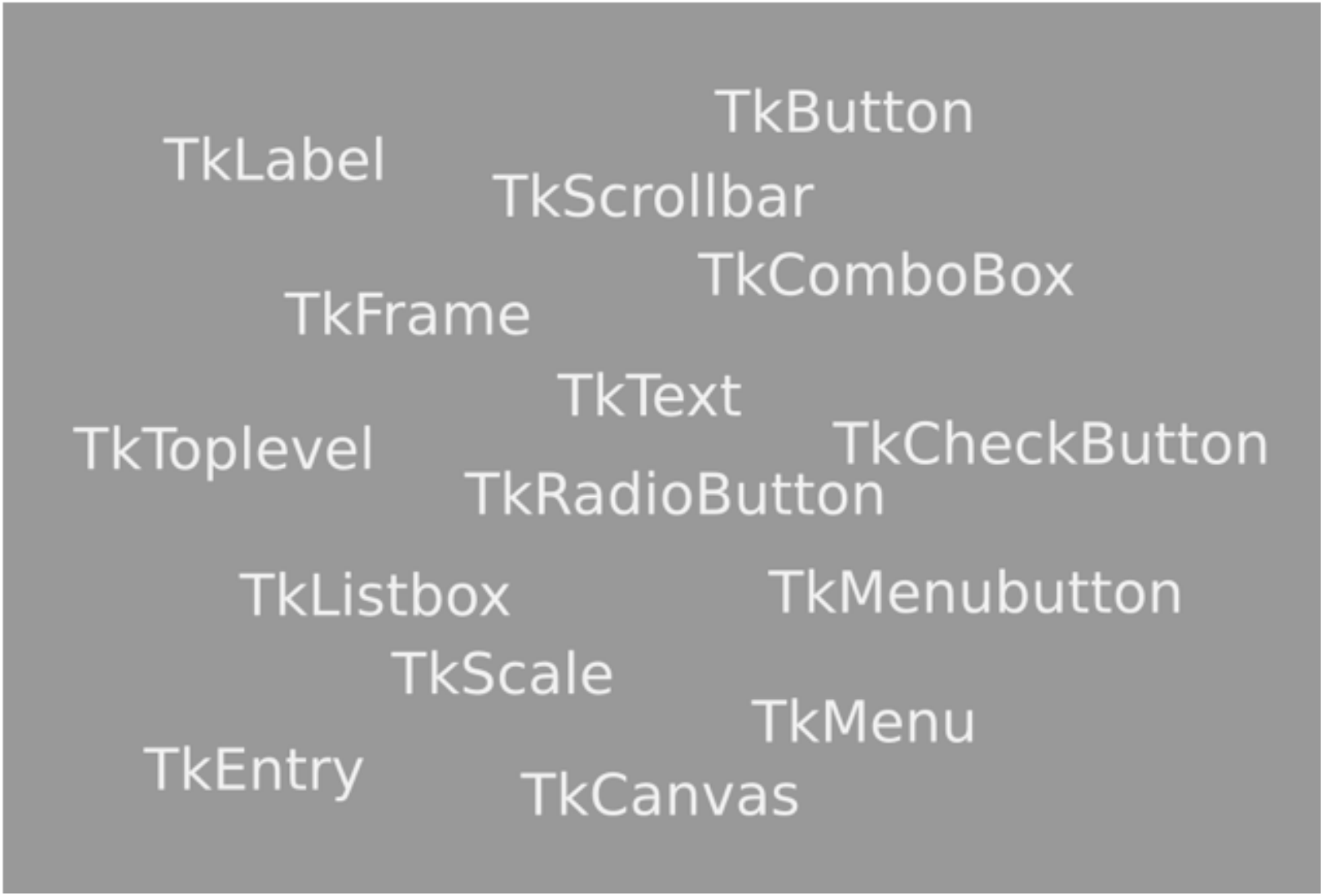
- Tkinter is the standard GUI library for Python.
- Python when combined with Tkinter provides a fast and easy way to create GUI applications.
- Tkinter provides a powerful object-oriented interface to the Tk GUI toolkit.
- All you need to do is perform the following steps:
  - Import the *Tkinter* module.
  - 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()
```

Tkinter


TkButton
TkLabel
TkScrollbar
TkComboBox
TkFrame
TkText
TkToplevel
TkCheckButton
TkRadioButton
TkListbox
TkMenubutton
TkScale
TkMenu
TkEntry
TkCanvas

Operator	Description
Button	The Button widget is used to display buttons in your application.
Canvas	The Canvas widget is used to draw shapes, such as lines, ovals, polygons and rectangles, in your application.
Checkbutton	The Checkbutton widget is used to display a number of options as checkboxes. The user can select multiple options at a time.
Entry	The Entry widget is used to display a single-line text field for accepting values from a user.
Frame	The Frame widget is used as a container widget to organize other widgets.
Label	The Label widget is used to provide a single-line caption for other widgets. It can also contain images.
Listbox	The Listbox widget is used to provide a list of options to a user.
Menubutton	The Menubutton widget is used to display menus in your application.
Menu	The Menu widget is used to provide various commands to a user. These commands are contained inside Menubutton.
Message	The Message widget is used to display multiline text fields for accepting values from a user.
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.
Scale	The Scale widget is used to provide a slider widget.
Scrollbar	The Scrollbar widget is used to add scrolling capability to various widgets, such as list boxes.
Text	The Text widget is used to display text in multiple lines.
Toplevel	The Toplevel widget is used to provide a separate window container.
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.
PanedWindow	A PanedWindow is a container widget that may contain any number of panes, arranged horizontally or vertically.
LabelFrame	A labelframe is a simple container widget. Its primary purpose is to act as a spacer or container for complex window layouts.
tkMessageBox	This module is used to display message boxes in your applications.

Standard Attributes

- Let's 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

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.

Example

- Import Tkinter *
- root = Tk()
- frame = Frame(root)
- frame.pack()
- bottomframe = Frame(root)
- bottomframe.pack(side = BOTTOM)
- redbutton = Button(frame, text="Red", fg="red")
  redbutton.pack(side = LEFT)
- greenbutton = Button(frame, text="Brown", fg="brown")
  greenbutton.pack(side = LEFT)

Example

- bluebutton = Button(frame, text="Blue", fg="blue")
- bluebutton.pack(side = LEFT)
- blackbutton = Button(bottomframe, text="Black", fg="black")
- blackbutton.pack(side = BOTTOM) root.mainloop()

Example

```
Import Tkinter
class GUIDemo(Frame):  # (inherit)  Tkinter Frame
    def __init__(self, master=None):
        Frame.__init__(self, master)
        self.grid()
        self.createWidgets()

    def createWidgets(self):
        self.inputText = Label(self)
        self.inputText["text"] = "Input:"
        self.inputText.grid(row=0, column=0)
        self.inputField = Entry(self)
        self.inputField["width"] = 50
        self.inputField.grid(row=0, column=1, columnspan=6)

        self.outputText = Label(self)
        self.outputText["text"] = "Output:"
        self.outputText.grid(row=1, column=0)
        self.outputField = Entry(self)
        self.outputField["width"] = 50
        self.outputField.grid(row=1, column=1, columnspan=6)
```

```
self.new = Button(self)
        self.new["text"] = "New"
        self.new.grid(row=2, column=0)
        self.load = Button(self)
        self.load["text"] = "Load"
        self.load.grid(row=2, column=1)
        self.save = Button(self)
        self.save["text"] = "Save"
        self.save.grid(row=2, column=2)
        self.encode = Button(self)
        self.encode["text"] = "Encode"
        self.encode.grid(row=2, column=3)
        self.decode = Button(self)
        self.decode["text"] = "Decode"
        self.decode.grid(row=2, column=4)
        self.clear = Button(self)
        self.clear["text"] = "Clear"
        self.clear.grid(row=2, column=5)
        self.copy = Button(self)
        self.copy["text"] = "Copy"
        self.copy.grid(row=2, column=6)
```

- self.displayText = Label(self)
- self.displayText["text"] = "something happened"
- self.displayText.grid(row=3, column=0, columnspan=7)
-
- if __name__ == '__main__':
- root = Tk()
- app = GUIDemo(master=root)
- app.mainloop()