UNIT – V

1. Operating System Interface

The os module contains plenty of functions for performing operating system stuff like changing directories and removing files, while os.path helps extract directory names, file names, and extensions from a given path. The great thing is that these modules work on any python-supported platform, making your programs much more portable.

Sr. No.	Methods with Description
1	os.access(path,mode) Use the real uid/gid to test for access to path.
2	os.chdir(path) Change the current working directory to path
3	os.chmod(path, mode) Change the mode of path to the numeric mode.
4	os.close(fd) Close file descriptor fd.
5	os.dup(fd) Return a duplicate of file descriptor fd.
6	os.fsync(fd) Force write of file with filedescriptor fd to disk.
7	os.getcwdu() Return a Unicode object representing the current working directory.
8	os.isatty(fd) Return True if the file descriptor fd is open and connected to a tty(-like) device, else False.
9	os.listdir(path) Return a list containing the names of the entries in the directory given by path.
10	os.lseek(fd, pos, how) Set the current position of file descriptor fd to position pos, modified by how.
11	os.makedirs(path[, mode]) Recursive directory creation function.
12	os.open(file, flags[, mode]) Open the file file and set various flags according to flags and possibly its mode according to mode.
13	os.pathconf(path, name) Return system configuration information relevant to a named file.
14	os.pipe() Create a pipe. Return a pair of file descriptors (r, w) usable for reading and writing, respectively.
15	os.popen(command[, mode[, bufsize]]) Open a pipe to or from command.
16	os.remove(path) Remove the file path.
17	os.rename(src, dst) Rename the file or directory src to dst.
18	os.rmdir(path) Remove the directory path
19	os.tmpfile() Return a new file object opened in update mode (w+b).
20	os.tmpnam() Return a unique path name that is reasonable for creating a temporary file.
21	os.ttyname(fd) Return a string which specifies the terminal device associated with file descriptor fd. If fd is not associated with a terminal device, an exception is raised.
22	os.unlink(path) Remove the file path.
23	os.utime(path, times) Set the access and modified times of the file specified by path.
24	os.write(fd, str) Write the string str to file descriptor fd. Return the number of

	bytes actually written.

1. os.access(path,mode) :

This function tests to see that the current process has permission to read, write, or execute a given path. The mode parameter can be combination of os.R_OK(read permission), os.W_OK(write permission), or X_OK(execute permission).

```
>>> os.access("d:\\abc.txt",os.R_OK)

True     # File has read permission

>>> os.access("d:\\abc.txt",os.X_OK)

True    # File has execute permission
```

2. os.chmod(path, mode):

Change the mode of path to the numeric mode. The mode parameter is a number created by adding different octal values listed in Table.

Value	Description
0400	Owner can read the path
0200	Owner can write the path
0100	Owner can executer the file or search the directory
0040	Group members can read the path
0020	Group members can write the path
0010	Group members can execute the file or search the directory
0004	Others can read the path
0002	Others can write the path
0001	Others can execute the file or search the directory

```
>>>os.chmod("d:\\abc.txt",640)
```

It gives the owner read/write permissions, group members read permissions and others no access to a file.

3. os.chdir(path) Change the current working directory to path

```
>>>os.chdir("d:\\hello")
```

4. os.close(fd)

Close file descriptor fd.

```
 >>> f=os.open("d:\\abc.txt", os.R_OK)

>>> os.close(f)
```

5. os.dup(fd) Return a duplicate of file descriptor fd.

```
>>> f=os.open("d:\\abc.txt", os.R_OK)
>>> f1=os.dup(f)
```

6. os.fsync(fd) Force write of file with filedescriptor fd to disk.

```
>>> f=os.open("d:\\abc.txt", os.W_OK)
>>> os.fsync(f)
```

7. os.open(fd[, flags[, mode]])

Creates a file descriptor fd

```
>>> f=os.open("d:\\abc.txt", os.O_CREAT)
>>> os.write(f,"hell0")
```

8. os.getcwd()

Return a Unicode object representing the current working directory.

```
>>>os.getcwd()
'C:\\Users\\DELL\\AppData\\Local\\Programs\\Python\\Python35-32'
```

9. os.isatty(fd)

Return True if the file descriptor fd is open and connected to a tty(-like) device, else False.

```
>>> os.isatty(f)
False
```

10. os.listdir(path) Return a list containing the names of the entries in the directory given by path.

```
>>> os.listdir("d:\\")
['$RECYCLE.BIN', 'A1.class', 'abc.html', 'abc.txt', 'hello', 'java.docx', 'java.pdf', 'jit.txt', 'klm.txt',
'Project', 'Project.zip', 'SourceCode', 'System Volume Information', 'Telugu Movies']
```

2. String Pattern Matching

A regular expression is a special sequence of characters that helps you match or find other strings or sets of strings, using a specialized syntax held in a pattern. Regular expressions are widely used in UNIX world. The module re provides full support for Perl-like regular expressions in Python. The re module raises the exception re.error if an error occurs while compiling or using a regular expression. Two important functions, which would be used to handle regular expressions. But a small thing first: There are various characters, which would have special meaning when they are used in regular expression. To avoid any confusion while dealing with regular expressions, we would use Raw Strings as r'expression'.

The match Function

This function attempts to match RE pattern to string with optional flags. Here is the syntax for this function:

re.match(pattern, string, flags=0)

Here is the description of the parameters:

Parameter	Description
pattern	This is the regular expression to be matched.
string	This is the string, which would be searched to match the pattern at the beginning of string.
flags	You can specify different flags using bitwise OR (|). These are modifiers, which are listed in the table below.

The re.match function returns a match object on success, none on failure. We usegroup(num) or groups() function of match object to get matched expression.

Match Object Methods	**Description**
group(num=0)	This method returns entire match (or specific subgroup num).
groups()	This method returns all matching subgroups in a tuple (empty if there weren't any).

Example

```
import re
line = "Cats are smarter than dogs"
matchObj = re.match( r'(.*) are (.*?) .*', line, re.M|re.I)
if matchObj:
        print "matchObj.group() : ", matchObj.group()
        print "matchObj.group(1) : ", matchObj.group(1)
        print "matchObj.group(2) : ", matchObj.group(2)
else:
        print "No match!!"
```

When the above code is executed, it produces following result:
matchObj.group() : Cats are smarter than dogs
matchObj.group(1) : Cats
matchObj.group(2) : smarter

The search Function

This function searches for first occurrence of RE pattern within string with optional flags.
Here is the syntax for this function:
re.search(pattern, string, flags=0)
Here is the description of the parameters:

Parameter	Description
pattern	This is the regular expression to be matched.
string	This is the string, which is searched to match the pattern anywhere in the string.

Flags	You can specify different flags using bitwise OR (|). These are modifiers, which are listed in the table below.

The re.search function returns a match object on success, none on failure. We use group(num) or groups() function of match object to get matched expression.

Match Object Methods	Description
group(num=0)	This method returns entire match (or specific subgroup num).
groups()	This method returns all matching subgroups in a tuple (empty if there weren't any).

Example

```
#!/usr/bin/python
import re
line = "Cats are smarter than dogs";
searchObj = re.search( r'(.*) are (.*?) .*', line, re.M|re.I)
if searchObj:
        print "searchObj.group() : ", searchObj.group()
        print "searchObj.group(1) : ", searchObj.group(1)
        print "searchObj.group(2) : ", searchObj.group(2)
else:
        print "Nothing found!!"
```

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

```
matchObj.group() : Cats are smarter than dogs
matchObj.group(1) : Cats
matchObj.group(2) : smarter
```

Matching Versus Searchings

Python offers two different primitive operations based on regular expressions: match checks for a match only at the beginning of the string, while search checks for a match anywhere in the string.

Example

```
import re
line = "Cats are smarter than dogs";
matchObj = re.match( r'dogs', line, re.M|re.I)
if matchObj:
        print "match --> matchObj.group() : ", matchObj.group()
else:
        print "No match!!"
searchObj = re.search( r'dogs', line, re.M|re.I)
if searchObj:
        print "search --> searchObj.group() : ", searchObj.group()
else:
        print "Nothing found!!"
```

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

No match!!

search --> matchObj.group() : dogs

Search and Replace

One of the most important re methods that use regular expressions is sub.

Syntax re.sub(pattern, repl, string, max=0)

This method replaces all occurrences of the RE pattern in string with repl, substituting all occurrences unless max provided. This method returns modified string.

Example

```
import re
phone = "2004-959-559 # This is Phone Number"
# Delete Python-style comments
num = re.sub(r'#.*$', "", phone)
print "Phone Num : ", num
# Remove anything other than digits
num = re.sub(r'\D', "", phone)
print "Phone Num : ", num
```

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

```
Phone Num : 2004-959-559
Phone Num : 2004959559
```

Regular-Expression Modifiers: Option Flags

Regular expression literals may include an optional modifier to control various aspects of matching. The modifiers are specified as an optional flag. You can provide multiple modifiers using exclusive OR (|), as shown previously and may be represented by one of these:

Modifier	Description
re.I	Performs case-insensitive matching.
re.L	Interprets words according to the current locale. This interpretation affects the alphabetic group (\w and \W), as well as word boundary behavior (\b and \B).
re.M	Makes $ match the end of a line (not just the end of the string) and makes ^ match the start of any line (not just the start of the string).
re.S	Makes a period (dot) match any character, including a newline.
re.U	Interprets letters according to the Unicode character set. This flag affects the behavior of \w, \W, \b, \B.
re.X	Permits "cuter" regular expression syntax. It ignores whitespace (except inside a set [] or when escaped by a backslash) and treats unescaped # as a comment marker.

Regular- Expression Patterns

Except for control characters, (+ ? . * ^ $ ( ) [ ] { } | \), all characters match themselves. You can escape a control character by preceding it with a backslash.

Following table lists the regular expression syntax that is available in Python:

Pattern	Description
^	Matches beginning of line.
$	Matches end of line.
.	Matches any single character except newline. Using m option allows it to match newline as well.
[...]	Matches any single character in brackets.
[^...]	Matches any single character not in brackets
re*	Matches 0 or more occurrences of preceding expression.
re+	Matches 1 or more occurrence of preceding expression.
re?	Matches 0 or 1 occurrence of preceding expression.
re{ n}	Matches exactly n number of occurrences of preceding expression.
re{ n,}	Matches n or more occurrences of preceding expression.
re{ n, m}	Matches at least n and at most m occurrences of preceding expression.
a| b	Matches either a or b.
(re)	Groups regular expressions and remembers matched text.
(?imx)	Temporarily toggles on i, m, or x options within a regular expression. If in parentheses, only that area is affected.
(?-imx)	Temporarily toggles off i, m, or x options within a regular expression. If in parentheses, only that area is affected.
(?: re)	Groups regular expressions without remembering matched text.
(?imx: re)	Temporarily toggles on i, m, or x options within parentheses.
(?-imx: re)	Temporarily toggles off i, m, or x options within parentheses.
(?#...)	Comment.
(?= re)	Specifies position using a pattern. Doesn't have a range.
(?! re)	Specifies position using pattern negation. Does not have a range.
(?> re)	Matches independent pattern without backtracking.
\w	Matches word characters.
\W	Matches non-word characters.
\s	Matches whitespace. Equivalent to [\t\n\r\f].
\S	Matches non-whitespace.
\d	Matches digits. Equivalent to [0-9].
\D	Matches non-digits.
\A	Matches beginning of string.
\Z	Matches end of string. If a newline exists, it matches just before newline.
\z	Matches end of string.
\G	Matches point where last match finished.
\b	Matches word boundaries when outside brackets. Matches backspace (0x08) when inside brackets.
\B	Matches non-word boundaries.

\n, \t, etc.	Matches newlines, carriage returns, tabs, etc.
\1...\9	Matches nth grouped subexpression.
\10	Matches nth grouped subexpression if it matched already. Otherwise refers to the octal representation of a character code.

Regular-Expression Examples

Literal characters

Example	Description
python	Match "python".

Character classes

Example	Description
[Pp]ython	Match "Python" or "python"
rub[ye]	Match "ruby" or "rube"
[aeiou]	Match any one lowercase vowel
[0-9]	Match any digit; same as [0123456789]
[a-z]	Match any lowercase ASCII letter
[A-Z]	Match any uppercase ASCII letter
[a-zA-Z0-9]	Match any of the above
[^aeiou]	Match anything other than a lowercase vowel
[^0-9]	Match anything other than a digit

Special Character Classes

Example

Example	Description
.	Match any character except newline
\d	Match a digit: [0-9]
\D	Match a non-digit: [^0-9]
\s	Match a whitespace character: [\t\r\n\f]
\S	Match non-whitespace: [^ \t\r\n\f]
\w	Match a single word character: [A-Za-z0-9_]
\W	Match a non-word character: [^A-Za-z0-9_]

Repetition Cases

Example	Description
ruby?	Match "rub" or "ruby": the y is optional.
ruby*	Match "rub" plus 0 or more ys.
ruby+	Match "rub" plus 1 or more ys.
\d{3}	Match exactly 3 digits.
\d{3,}	Match 3 or more digits.
\d{3,5}	Match 3, 4, or 5 digits.

Nongreedy repetition

This matches the smallest number of repetitions:

Example	Description
<.*>	Greedy repetition: matches "<python>perl>".
<.*?>	Nongreedy: matches "<python>" in "<python>perl>".

Grouping with Parentheses

Example	Description
\D\d+	No group: + repeats \d.
(\D\d)+	Grouped: + repeats \D\d pair.
([Pp]ython(,)?)+	Match "Python", "Python, python, python", etc.

Anchors

This needs to specify match position.

Example	Description
^Python	Match "Python" at the start of a string or internal line.
Python$	Match "Python" at the end of a string or line.
\APython	Match "Python" at the start of a string.
Python\Z	Match "Python" at the end of a string.
\bPython\b	Match "Python" at a word boundary.
\brub\B	\B is non-word boundary: match "rub" in "rube" and "ruby" but not alone.
Python(?=!)	Match "Python", if followed by an exclamation point.
Python(?!!)	Match "Python", if not followed by an exclamation point.

3. Mathematics

Number Type Conversion

Python converts numbers internally in an expression containing mixed types to a common type for evaluation. But sometimes, you need to coerce a number explicitly from one type to another to satisfy the requirements of an operator or function parameter.

1. Type **int(x)** to convert x to a plain integer.
2. Type **long(x)** to convert x to a long integer.
3. Type **float(x)** to convert x to a floating-point number.
4. Type **complex(x)** to convert x to a complex number with real part x and imaginary part zero.
5. Type **complex(x, y)** to convert x and y to a complex number with real part x and imaginary part y. x and y are numeric expressions

Mathematical Functions

Python includes following functions that perform mathematical calculations.

Function	**Returns (description)**

abs(x)	The absolute value of x: the (positive) distance between x and zero.
ceil(x)	The ceiling of x: the smallest integer not less than x
cmp(x, y)	-1 if x < y, 0 if x == y, or 1 if x > y
exp(x)	The exponential of x: ex
fabs(x)	The absolute value of x.
floor(x)	The floor of x: the largest integer not greater than x
log(x)	The natural logarithm of x, for x> 0
log10(x)	The base-10 logarithm of x for x> 0.
max(x1, x2,...)	The largest of its arguments: the value closest to positive infinity
min(x1, x2,...)	The smallest of its arguments: the value closest to negative infinity
modf(x)	The fractional and integer parts of x in a two-item tuple. Both parts have the same sign as x. The integer part is returned as a float.
pow(x, y)	The value of x**y.
round(x [,n])	x rounded to n digits from the decimal point. Python rounds away from zero as a tie-breaker: round(0.5) is 1.0 and round(-0.5) is -1.0.
sqrt(x)	The square root of x for x > 0

Random Number Functions

Random numbers are used for games, simulations, testing, security, and privacy applications. Python includes following functions that are commonly used.

Function	**Description**
choice(seq)	A random item from a list, tuple, or string.
randrange ([start,] stop [,step])	A randomly selected element from range(start, stop, step)
random()	A random float r, such that 0 is less than or equal to r and r is less than 1
seed([x])	Sets the integer starting value used in generating random numbers. Call this function before calling any other random module function. Returns None.
shuffle(lst)	Randomizes the items of a list in place. Returns None.
uniform(x, y)	A random float r, such that x is less than or equal to

	r and r is less than y

Trigonometric Functions: Python includes following functions that perform trigonometric calculations.

Function	**Description**
acos(x)	Return the arc cosine of x, in radians.
asin(x)	Return the arc sine of x, in radians.
atan(x)	Return the arc tangent of x, in radians.
atan2(y, x)	Return atan(y / x), in radians.
cos(x)	Return the cosine of x radians.
hypot(x, y)	Return the Euclidean norm, sqrt(x*x + y*y).
sin(x)	Return the sine of x radians.
tan(x)	Return the tangent of x radians.
degrees(x)	Converts angle x from radians to degrees.
radians(x)	Converts angle x from degrees to radians.

Mathematical Constants:

The module also defines two mathematical constants:

Constants	Description
pi	The mathematical constant pi.
e	The mathematical constant e.

4. Internet Access

Python provides two levels of access to network services. At a low level, you can access the basic socket support in the underlying operating system, which allows you to implement clients and servers for both connection-oriented and connectionless protocols. Python also has libraries that provide higher-level access to specific application-level network protocols, such as FTP, HTTP, and so on.

Socket Programming.

What is Sockets?

Sockets are the endpoints of a bidirectional communications channel. Sockets may communicate within a process, between processes on the same machine, or between processes on different continents. Sockets may be implemented over a number of different channel types: Unix domain sockets, TCP, UDP, and so on. The socket library provides specific classes for handling the common transports as well as a generic interface for handling the rest.

Sockets have their own vocabulary:

Term	Description
domain	The family of protocols that is used as the transport mechanism. These values are constants such as AF_INET, PF_INET, PF_UNIX, PF_X25, and so on.
type	The type of communications between the two endpoints, typically SOCK_STREAM for connection-oriented protocols and SOCK_DGRAM for connectionless protocols.
Protocol	Typically zero, this may be used to identify a variant of a protocol within a domain and type.
Hostname	The identifier of a network interface:

	A string, which can be a host name, a dotted-quad address, or an IPV6 address in colon andpossiblydot notation A string "<broadcast>", which specifies an INADDR_BROADCAST address. A zero-length string, which specifies INADDR_ANY, or An Integer, interpreted as a binary address in host byte order.
Port	Each server listens for clients calling on one or more ports. A port may be a Fixnum port number, a string containing a port number, or the name of a service.

The socket Module

To create a socket, you must use the socket.socket function available in socket module, which has the general syntax –

s = socket.socket (socket_family, socket_type, protocol=0)

Here is the description of the parameters −

socket_family: This is either AF_UNIX or AF_INET, as explained earlier.

socket_type: This is either SOCK_STREAM or SOCK_DGRAM.

protocol: This is usually left out, defaulting to 0.

Once you have socket object, then you can use required functions to create your client or server program. Following is the list of functions required −

Server Socket Methods

Method	Description
s.bind	This method binds address hostname, portnumberpair to socket.
s.listen	This method sets up and start TCP listener.
s.accept	This passively accept TCP client connection, waiting until connection arrives blocking.

Client Socket Methods

Method	Description
s.connect	This method actively initiates TCP server connection.

General Socket Methods

Method	Description
s.recv	This method receives TCP message
s.send	This method transmits TCP message
s.recvfrom	This method receives UDP message
s.sendto	This method transmits UDP message
s.close	This method closes socket
socket.gethostname	Returns the hostname.

A Simple Server

To write Internet servers, we use the socket function available in socket module to create a socket object. A socket object is then used to call other functions to setup a socket server. Now call bindhostname, port function to specify a port for your service on the given host. Next, call the

accept method of the returned object. This method waits until a client connects to the port you specified, and then returns a connection object that represents the connection to that client.

```
s = socket.socket()             # Create a socket object
host = socket.gethostnam e()  # Get local m achine nam e
port = 12345                    # Reserve a port for your service.
s.bind((host, port))             # Bind to the port
s.listen(5)                     # Now wait for client connection.
while True:
c, addr = s.accept()            # Establish connection with client.
print 'Got connection from ', addr
c.send('Thank you for connecting')
c.close()                       # Close the connection
```

A Simple Client

Let us write a very simple client program which opens a connection to a given port 12345 and given host. This is very simple to create a socket client using Python's socket module function. The socket.connecthosname, port opens a TCP connection to hostname on the port. Once you have a socket open, you can read from it like any IO object. When done, remember to close it, as you would close a file. The following code is a very simple client that connects to a given host and port, reads any available data from the socket, and then exits −

```
import socket                   # Im port socket m odule
s = socket.socket()             # Create a socket object
host = socket.gethostname()   # Get local m achine nam e
port = 12345                    # Reserve a port for your service.
s.connect((host, port))
print s.recv(1024)
s.close                         # Close the socket when done
```

Now run this server.py in background and then run above client.py to see the result.

```
# Following would start a server in background.
$ python server.py &

# Once server is started run client as follows:
$ python client.py
```

This would produce following result −

```
Got connection from ('127.0.0.1', 48437)
Thank you for connecting
```

Python Internet modules

A list of some important modules in Python Network/Internet programming.

Protocol	Common function	Port No	Python module
HTTP	Web pages	80	httplib, urllib, xmlrpclib
NNTP	Usenet news	119	nntplib

FTP	File transfers	20	ftplib, urllib
SMTP	Sending email	25	smtplib
POP3	Fetching email	110	poplib
IMAP4	Fetching email	143	imaplib
Telnet	Command lines	23	telnetlib
Gopher	Document transfers	70	gopherlib, urllib

5. Dates and Times

A Python program can handle date and time in several ways. Converting between date formats is a common chore for computers. Python's time and calendar modules help track dates and times.

What is Tick?

Time intervals are floating-point numbers in units of seconds. Particular instants in time are expressed in seconds since 12:00am, January 1, 1970(epoch).

There is a popular time module available in Python which provides functions for working with times and for converting between representations. The function time.time() returns the current system time in ticks since 12:00am, January 1, 1970(epoch).

Example

```
#!/usr/bin/python
import time;    # This is required to include time module.
ticks = time.time()
print "Number of ticks since 12:00am, January 1, 1970:", ticks
```

This would produce a result something as follows:

```
Number of ticks since 12:00am, January 1, 1970: 7186862.73399
```

Date arithmetic is easy to do with ticks. However, dates before the epoch cannot be represented in this form. Dates in the far future also cannot be represented this way - the cutoff point is sometime in 2038 for UNIX and Windows.

What is TimeTuple?

Many of Python's time functions handle time as a tuple of 9 numbers, as shown below:

Index	**Field**	**Values**
0	4-digit year	2008
1	Month	1 to 12
2	Day	1 to 31
3	Hour	0 to 23
4	Minute	0 to 59
5	Second	0 to 61 (60 or 61 are leap-seconds)
6	Day of Week	0 to 6 (0 is Monday)
7	Day of year	1 to 366 (Julian day)
8	Daylight savings	-1, 0, 1, -1 means library determines DST

The above tuple is equivalent to struct_time structure. This structure has following attributes:

Index	**Attributes**	**Values**
0	tm_year	2008
1	tm_mon	1 to 12
2	tm_mday	1 to 31
3	tm_hour	0 to 23
4	tm_min	0 to 59
5	tm_sec	0 to 61 (60 or 61 are leap-

		seconds)
6	tm_wday	0 to 6 (0 is Monday)
7	tm_yday	1 to 366 (Julian day)
8	tm_isdst	-1, 0, 1, -1 means library determines DST

Getting Current Time

To translate a time instant from a seconds since the epoch floating-point value into a time-tuple, pass the floating-point value to a function (For example, localtime) that returns a time-tuple with all nine items valid.

```
#!/usr/bin/python
import time;
localtime = time.localtime(time.time())
print "Local current time :", localtime
```

This would produce the following result, which could be formatted in any other presentable form:

```
Local current time : time.struct_time(tm_year=2013, tm_mon=7,
tm_mday=17, tm_hour=21, tm_min=26, tm_sec=3, tm_wday=2, tm_yday=198, tm_isdst=0)
```

Getting Formatted Time

You can format any time as per your requirement, but simple method to get time in readable format is asctime():

```
#!/usr/bin/python
import time;
localtime = time.asctime( time.localtime(time.time()) )
print "Local current time :", localtime
```

This would produce the following result:

```
Local current time : Tue Jan 13 10:17:09 2009
```

Getting Calendar for a Month

The calendar module gives a wide range of methods to play with yearly and monthly calendars. Here, we print a calendar for a given month (Jan 2008):

```
#!/usr/bin/python
import calendar
cal = calendar.month(2008, 1)
print "Here is the calendar:"
print cal;
```

This would produce the following result:

```
Here is the calendar:
January 2008
Mo      Tu      We      Th      Fr      Sa      Su
        1       2       3       4       5       6
7       8       9       10      11      12      13
14      15      16      17      18      19      20
21      22      23      24      25      26      27
28      29      30      31
```

The time Module:

There is a popular time module available in Python which provides functions for working with times and for converting between representations. Here is the list of all available methods:

Sr. No.	Function with Description
1	time.altzone The offset of the local DST timezone, in seconds west of UTC, if one is defined. This is negative if the local DST timezone is east of UTC (as in Western Europe, including the UK). Only use this if daylight is nonzero.
2	time.asctime([tupletime]) Accepts a time-tuple and returns a readable 24-character string such as 'Tue Dec 11 18:07:14 2008'.
3	time.clock( ) Returns the current CPU time as a floating-point number of seconds. To measure computational costs of different approaches, the value of time.clock is more useful than that of time.time().
4	time.ctime([secs]) Like asctime(localtime(secs)) and without arguments is like asctime( )
5	time.gmtime([secs]) Accepts an instant expressed in seconds since the epoch and returns a time-tuple t with the UTC time. Note : t.tm_isdst is always 0
6	time.localtime([secs]) Accepts an instant expressed in seconds since the epoch and returns a time-tuple t with the local time (t.tm_isdst is 0 or 1, depending on whether DST applies to instant secs by local rules).
7	time.mktime(tupletime) Accepts an instant expressed as a time-tuple in local time and returns a floating-point value with the instant expressed in seconds since the epoch.
8	time.sleep(secs) Suspends the calling thread for secs seconds.
9	time.strftime(fmt[,tupletime]) Accepts an instant expressed as a time-tuple in local time and returns a string representing the instant as specified by string fmt.
10	time.strptime(str,fmt='%a %b %d %H:%M:%S %Y') Parses str according to format string fmt and returns the instant in time-tuple format.
11	time.time( ) Returns the current time instant, a floating-point number of seconds since the epoch.
12	time.tzset() Resets the time conversion rules used by the library routines. The environment variable TZ specifies how this is done.

time.altzone

The method altzone() is the attribute of the time module. This returns the offset of the local DST timezone, in seconds west of UTC, if one is defined. This is negative if the local DST timezone is east of UTC (as in Western Europe, including the UK). Only use this if daylight is nonzero.

Syntax
Following is the syntax for altzone() method:
time.altzone
Return Value- This method returns the offset of the local DST timezone, in seconds west of UTC, if one is defined.
Example
The following example shows the usage of altzone() method.

```
#!/usr/bin/python
import time
print "time.altzone %d " % time.altzone
```

When we run above program, it produces following result:
time.altzone() 25200

time.actime([tupletime])

The method asctime() converts a tuple or struct_time representing a time as returned by gmtime() or localtime() to a 24-character string of the following form: 'Tue Feb 17 23:21:05 2009'.
Syntax
Following is the syntax for asctime() method:
time.asctime([t]))
Parameters
t -- This is a tuple of 9 elements or struct_time representing a time as returned by gmtime() or localtime() function.
Return Value-This method returns 24-character string of the following form: 'Tue Feb 17 23:21:05 2009'.
Example
The following example shows the usage of asctime() method.

```
#!/usr/bin/python
import time
t = time.localtime()
print "time.asctime(t): %s " % time.asctime(t)
```

When we run above program, it produces following result:
time.asctime(t): Tue Feb 17 09:42:58 2009

time.clock( )

The method clock() returns the current processor time as a floating point number expressed in seconds on Unix. The precision depends on that of the C function of the same name, but in any case, this is the function to use for benchmarking Python or timing algorithms.
On Windows, this function returns wall-clock seconds elapsed since the first call to this function, as a floating point number, based on the Win32 function QueryPerformanceCounter.
Syntax
Following is the syntax for clock() method:
time.clock()
Return Value - This method returns the current processor time as a floating point number expressed in seconds on Unix and in Windows it returns wall-clock seconds elapsed since the first call to this function, as a floating point number.
Example
The following example shows the usage of clock() method.

```
#!/usr/bin/python
import time
def procedure():
        time.sleep(2.5)
        # measure process time
        t0 = time.clock()
procedure()
print time.clock() - t0, "seconds process time"
# measure wall time
t0 = time.time()
procedure()
print time.time() - t0, "seconds wall time"
```

When we run above program, it produces following result:

```
0.0 seconds process time
2.50023603439 seconds wall time
```

Note: Not all systems can measure the true process time. On such systems (including Windows), clock usually measures the wall time since the program was started.

time.ctime([secs])

The method ctime() converts a time expressed in seconds since the epoch to a string representing local time. If secs is not provided or None, the current time as returned by time() is used. This function is equivalent to asctime(localtime(secs)). Locale information is not used by ctime().

Syntax

Following is the syntax for ctime() method:

```
time.ctime([ sec ])
```

Parameters

- sec-- These are the number of seconds to be converted into string representation.

Return Value- This method does not return any value.

Example

The following example shows the usage of ctime() method.

```
#!/usr/bin/python
import time
print "time.ctime() : %s" % time.ctime()
```

When we run above program, it produces following result:

```
time.ctime() : Tue Feb 17 10:00:18 2009
```

time.gmtime([secs])

The method gmtime() converts a time expressed in seconds since the epoch to a struct_time in UTC in which the dst flag is always zero. If secs is not provided or None, the current time as returned by time() is used.

Syntax

Following is the syntax for gmtime() method:

```
time.gmtime([ sec ])
```

Parameters

sec -- These are the number of seconds to be converted into structure struct_time representation.

Return Value- This method does not return any value.

Example

The following example shows the usage of gmtime() method.

```
#!/usr/bin/python
import time
print "time.gmtime() : %s" % time.gmtime()
```
When we run above program, it produces following result:
```
time.gmtime() : (2009, 2, 17, 17, 3, 38, 1, 48, 0)
```

time.localtime([secs])

The method localtime() is similar to gmtime() but it converts number of seconds to local time. If secs is not provided or None, the current time as returned by time() is used. The dst flag is set to 1 when DST applies to the given time.

Syntax

Following is the syntax for localtime() method:
```
time.localtime([ sec ])
```
Parameters

sec -- These are the number of seconds to be converted into structure struct_time representation.

Return Value- This method does not return any value.

Example

The following example shows the usage of localtime() method.
```
#!/usr/bin/python
import time
print "time.localtime() : %s" % time.localtime()
```
When we run above program, it produces following result:
```
time.localtime() : (2009, 2, 17, 17, 3, 38, 1, 48, 0)
```

time.mktime(tupletime)

The method mktime() is the inverse function of localtime(). Its argument is the struct_time or full 9-tuple and it returns a floating point number, for compatibility with time(). If the input value cannot be represented as a valid time, either OverflowError or ValueErrorwill be raised.

Syntax

Following is the syntax for mktime() method:
```
time.mktime(t)
```
Parameters- t -- This is the struct_time or full 9-tuple.

Return Value-This method returns a floating point number, for compatibility with time().

Example

The following example shows the usage of mktime() method.
```
#!/usr/bin/python
import time
t = (2009, 2, 17, 17, 3, 38, 1, 48, 0)
secs = time.mktime( t )
print "time.mktime(t) : %f" % secs
print "asctime(localtime(secs)): %s" % time.asctime(time.localtime(secs))
```
When we run above program, it produces following result:
```
time.mktime(t) : 1234915418.000000
asctime(localtime(secs)): Tue Feb 17 17:03:38 2009
```

time.sleep(secs)

The method sleep() suspends execution for the given number of seconds. The argument may be a floating point number to indicate a more precise sleep time.

The actual suspension time may be less than that requested because any caught signal will terminate the sleep() following execution of that signal's catching routine.
Syntax
Following is the syntax for sleep() method:
time.sleep(t)
Parameters
t -- This is the number of seconds execution to be suspended.
Example
The following example shows the usage of sleep() method.

```
#!/usr/bin/python
import time
print "Start : %s" % time.ctime()
time.sleep( 5 )
print "End : %s" % time.ctime()
```

When we run above program, it produces following result:

```
Start : Tue Feb 17 10:19:18 2009
End : Tue Feb 17 10:19:23 2009
```

time.strftime(fmt[,tupletime])

The method strftime() converts a tuple or struct_time representing a time as returned by gmtime() or localtime() to a string as specified by the format argument.
If t is not provided, the current time as returned by localtime() is used. format must be a string. An exception ValueError is raised if any field in t is outside of the allowed range.
Syntax
Following is the syntax for strftime() method:
time.strftime(format[, t])
Parameters
t -- This is the time in number of seconds to be formatted.
format -- This is the directive which would be used to format given time. The following directives can be embedded in the format string:
Directive
%a - abbreviated weekday name
%A - full weekday name
%b - abbreviated month name
%B - full month name
%c - preferred date and time representation
%C - century number (the year divided by 100, range 00 to 99)
%d - day of the month (01 to 31)
Example
The following example shows the usage of strftime() method.

```
#!/usr/bin/python
import time
t = (2009, 2, 17, 17, 3, 38, 1, 48, 0)
t = time.mktime(t)
print time.strftime("%b %d %Y %H:%M:%S", time.gmtime(t))
```

When we run above program, it produces following result:

```
Feb 18 2009 00:03:38
```

time.time( )

The method time() returns the time as a floating point number expressed in seconds since the epoch, in UTC.
Note: Even though the time is always returned as a floating point number, not all systems provide time with a better precision than 1 second. While this function normally returns non-decreasing values, it can return a lower value than a previous call if the system clock has been set back between the two calls.
Following is the syntax for time() method:
time.time()
Return Value
This method returns the time as a floating point number expressed in seconds since the epoch, in UTC.
Example
The following example shows the usage of time() method.

```
#!/usr/bin/python
import time
print "time.time(): %f " % time.time()
print time.localtime( time.time() )
print time.asctime( time.localtime(time.time()) )
```

When we run above program, it produces following result:

```
time.time(): 1234892919.655932
(2009, 2, 17, 10, 48, 39, 1, 48, 0)
Tue Feb 17 10:48:39 2009
```

time.tzset()

The method tzset() resets the time conversion rules used by the library routines. The environment variable TZ specifies how this is done.
The standard format of the TZ environment variable is (whitespace added for clarity):
std offset [dst [offset [,start[/time], end[/time]]]]
- std and dst: Three or more alphanumerics giving the timezone abbreviations. These will be propagated into time.tzname.
- offset: The offset has the form: .hh[:mm[:ss]]. This indicates the value added the local time to arrive at UTC. If preceded by a '-', the timezone is east of the Prime Meridian; otherwise, it is west. If no offset follows dst, summer time is assumed to be one hour ahead of standard time.
- start[/time], end[/time]: Indicates when to change to and back from DST. The format of the start and end dates are one of the following:

o Jn: The Julian day n (1 <= n <= 365). Leap days are not counted, so in all years February 28 is day 59 and March 1 is day 60.
o n: The zero-based Julian day (0 <= n <= 365). Leap days are counted, and it is possible to refer to February 29.
Mm.n.d: The d'th day (0 <= d <= 6) or week n of month m of the year (1 <= n <= 5, 1 <= m <= 12, where week 5 means 'the last d day in month m' which may occur in either the fourth or the fifth week). Week 1 is the first week in which the d'th day occurs. Day zero is Sunday.
o time: This has the same format as offset except that no leading sign ('-' or '+') is allowed. The default, if time is not given, is 02:00:00.
Syntax
Following is the syntax for tzset() method:

time.tzset()
Example
The following example shows the usage of tzset() method.

```
#!/usr/bin/python
import time
import os
os.environ['TZ'] = 'EST+05EDT,M4.1.0,M10.5.0'
time.tzset()
print time.strftime('%X %x %Z')
os.environ['TZ'] = 'AEST-10AEDT-11,M10.5.0,M3.5.0'
time.tzset()
print time.strftime('%X %x %Z')
```

When we run above program, it produces following result

```
13:00:40 02/17/09 EST
05:00:40 02/18/09 AEDT
```

There are following two important attributes available with time module:

1. time.timezone:

Attribute time.timezone is the offset in seconds of the local time zone (without DST) from UTC (>0 in the Americas; <=0 in most of Europe, Asia, Africa).

2. time.tzname

Attribute time.tzname is a pair of locale-dependent strings, which are the names of the local time zone without and with DST, respectively.

The calendar Module

The calendar module supplies calendar-related functions, including functions to print a text calendar for a given month or year.

By default, calendar takes Monday as the first day of the week and Sunday as the last one. To change this, call calendar.setfirstweekday() function.

Here is a list of functions available with the calendar module: Sr. No. Function with Description

1. calendar.calendar(year,w=2,l=1,c=6)

Returns a multiline string with a calendar for year formatted into three columns separated by c spaces. w is the width in characters of each date; each line has length 21*w+18+2*c. l is the number of lines for each week.

2. calendar.firstweekday( )

Returns the current setting for the weekday that starts each week. By default, when calendar is first imported, this is 0, meaning Monday.

3. calendar.isleap(year)

Returns True if year is a leap year; otherwise, False.

4. calendar.leapdays(y1,y2)

Returns the total number of leap days in the years within range(y1,y2).

5. calendar.month(year,month,w=2,l=1)

Returns a multiline string with a calendar for month of year, one line per week plus two header lines. w is the width in characters of each date; each line has length 7*w+6. l is the number of lines for each week.

6. calendar.monthcalendar(year,month)

Returns a list of lists of ints. Each sublist denotes a week. Days outside month of year are set to 0; days within the month are set to their day-of-month, 1 and up.

7. calendar.monthrange(year,month)

Returns two integers. The first one is the code of the weekday for the first day of the month month in year; the second one is the number of days in the month. Weekday codes are 0 (Monday) to 6 (Sunday); month numbers are 1 to 12.

8. calendar.prcal(year, w=2, l=1, c=6)

Like print calendar.calendar(year, w, l, c).

9. calendar.prmonth(year, month, w=2, l=1)

Like print calendar.month(year, month, w, l).

10. calendar.setfirstweekday(weekday)

Sets the first day of each week to weekday code weekday. Weekday codes are 0 (Monday) to 6 (Sunday).

11. calendar.timegm(tupletime)

The inverse of time.gmtime: accepts a time instant in time-tuple form and returns the same instant as a floating-point number of seconds since the epoch.

12. calendar.weekday(year,month,day)

Returns the weekday code for the given date. Weekday codes are 0 (Monday) to 6 (Sunday); month numbers are 1 (January) to 12 (December).

6. Multi Threading

Running several threads is similar to running several different programs concurrently, but with the following benefits:

- Multiple threads within a process share the same data space with the main thread and can therefore share information or communicate with each other more easily than if they were separate processes.
- Threads sometimes called light-weight processes and they do not require much memory overhead; they care cheaper than processes.

A thread has a beginning, an execution sequence, and a conclusion. It has an instruction pointer that keeps track of where within its context it is currently running.

- It can be pre-empted (interrupted)
- It can temporarily be put on hold (also known as sleeping) while other threads are running - this is called yielding.

6.1 Starting a New Thread

To spawn another thread, you need to call following method available in thread module:

thread.start_new_thread (function, args[, kwargs])

This method call enables a fast and efficient way to create new threads in both Linux and Windows.

The method call returns immediately and the child thread starts and calls function with the passed list of agrs. When function returns, the thread terminates.

Here, args is a tuple of arguments; use an empty tuple to call function without passing any arguments. kwargs is an optional dictionary of keyword arguments.

Example

```
import thread
import time
def print_time( threadName, delay):
    count = 0
    while count < 5:
                    time.sleep(delay)
                    count += 1
```

```
                print( "%s: %s" % ( threadName, time.ctime(time.time()) ))
# Create two threads as follows
try:
    thread.start_new_thread( print_time, ("Thread-1", 2, ) )
    thread.start_new_thread( print_time, ("Thread-2", 4, ) )
except:
    print ("Error: unable to start thread")
while 1:
    pass
```

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

```
Thread-1: Thu Jan 22 15:42:17 2009
Thread-1: Thu Jan 22 15:42:19 2009
Thread-2: Thu Jan 22 15:42:19 2009
Thread-1: Thu Jan 22 15:42:21 2009
Thread-2: Thu Jan 22 15:42:23 2009
Thread-1: Thu Jan 22 15:42:23 2009
Thread-1: Thu Jan 22 15:42:25 2009
Thread-2: Thu Jan 22 15:42:27 2009
Thread-2: Thu Jan 22 15:42:31 2009
Thread-2: Thu Jan 22 15:42:35 2009
```

Although it is very effective for low-level threading, but the thread module is very limited compared to the newer threading module.

6.2 The Threading Module:

The newer threading module included with Python 2.4 provides much more powerful, high-level support for threads than the thread module discussed in the previous section.

The threading module exposes all the methods of the thread module and provides some additional methods:

- threading.activeCount(): Returns the number of thread objects that are active.
- threading.currentThread(): Returnsthe number of thread objects in the caller's thread control.
- threading.enumerate(): Returns a list of all thread objects that are currently active.

In addition to the methods, the threading module has the Thread class that implements threading. The methods provided by the Thread class are as follows:

- run(): The run() method is the entry point for a thread.
- start(): The start() method starts a thread by calling the run method.
- join([time]): The join() waits for threads to terminate.
- isAlive(): TheisAlive() method checks whether a thread is still executing.
- getName(): The getName() method returns the name of a thread.
- setName(): The setName() method sets the name of a thread.

Creating Thread Using Module:

To implement a new thread using the threading module, you have to do the following:

- Define a new subclass of the Thread class.
- Override the __init_(self [,args]) method to add additional arguments.
- Then, override the run(self [,args]) method to implement what the thread should do when started.

Once you have created the new Thread subclass, you can create an instance of it and then start a new thread by invoking the start(), which in turn calls run() method.

Example

```
#!/usr/bin/python
import threading
import time
exitFlag = 0
class myThread (threading.Thread):
    def __init__(self, threadID, name, counter):
        threading.Thread.__init__(self)
        self.threadID = threadID
        self.name = name
        self.counter = counter
    def run(self):
        print ("Starting " + self.name)
        printtime(self.name, self.counter, 5)
        print ("Exiting " + self.name)

def printtime(threadName, delay, counter):
    while counter:
        if exitFlag:
            thread.exit()
        time.sleep(delay)
        print ("%s: %s" % (threadName, time.ctime(time.time())))
        counter -= 1
# Create new threads
thread1 = myThread(1, "Thread-1", 1)
thread2 = myThread(2, "Thread-2", 2)
# Start new Threads
thread1.start()
thread2.start()
print( "Exiting Main Thread")
```

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

```
Starting Thread-1
Starting Thread-2
Exiting Main Thread
Thread-1: Thu Mar 21 09:10:03 2013
Thread-1: Thu Mar 21 09:10:04 2013
Thread-2: Thu Mar 21 09:10:04 2013
Thread-1: Thu Mar 21 09:10:05 2013
Thread-1: Thu Mar 21 09:10:06 2013
Thread-2: Thu Mar 21 09:10:06 2013
Thread-1: Thu Mar 21 09:10:07 2013
Exiting Thread-1
Thread-2: Thu Mar 21 09:10:08 2013
Thread-2: Thu Mar 21 09:10:10 2013
Thread-2: Thu Mar 21 09:10:12 2013
Exiting Thread-2
```

6.3 Synchronizing Threads

The threading module provided with Python includes a simple-to-implement locking mechanism that allows you to synchronize threads. A new lock is created by calling theLock() method, which returns the new lock.

The acquire(blocking) method of the new lock object is used to force threads to run synchronously. The optional blocking parameter enables you to control whether the thread waits to acquire the lock.

If blocking is set to 0, the thread returns immediately with a 0 value if the lock cannot be acquired and with a 1 if the lock was acquired. If blocking is set to 1, the thread blocks and wait for the lock to be released.

The release() method of the new lock object is used to release the lock when it is no longer required.

Example

```
#!/usr/bin/python
import threading
import time
class myThread (threading.Thread):
    def __init__(self, threadID, name, counter):
        threading.Thread.__init__(self)
        self.threadID = threadID
        self.name = name
        self.counter = counter

    def run(self):
        print( "Starting " + self.name)
        # Get lock to synchronize threads
        threadLock.acquire()
        print_time(self.name, self.counter, 3)
        # Free lock to release next thread
        threadLock.release()

def print_time(threadName, delay, counter):
    while counter:
        time.sleep(delay)
        print( "%s: %s" % (threadName, time.ctime(time.time())))
        counter -= 1

threadLock = threading.Lock()
threads = []
# Create new threads
thread1 = myThread(1, "Thread-1", 1)
thread2 = myThread(2, "Thread-2", 2)
# Start new Threads
thread1.start()
thread2.start()
# Add threads to thread list
threads.append(thread1)
threads.append(thread2)
```

```
# Wait for all threads to complete
for t in threads:
    t.join()
print ("Exiting Main Thread")
```

OUTPUT:

```
Starting Thread-1
Starting Thread-2
Thread-1: Thu Mar 21 09:11:28 2013
Thread-1: Thu Mar 21 09:11:29 2013
Thread-1: Thu Mar 21 09:11:30 2013
Thread-2: Thu Mar 21 09:11:32 2013
Thread-2: Thu Mar 21 09:11:34 2013
Thread-2: Thu Mar 21 09:11:36 2013
Exiting Main Thread
```

7. Turtle

Graphical User Interfaces (GUI's) provide a rich environment in which information can be exchanged between a user and the computer. GUI's are not limited to simply displaying text and reading text from the keyboard. GUI's enable users to control the behavior of a program by performing actions such as using the mouse to drag or click on graphical objects. GUI's can make using programs much more intuitive and easier to learn since they provide users with immediate visual feedback that shows the effects of their actions.

There are many Python packages that can be used to create graphics and GUI's. Two graphics modules, called turtle and tkinter, come as a part of Python's standard library. tkinter is primarily designed for creating GUI's. In fact, IDLE is built using tkinter. However, we will focus on the turtle module that is primarily used as a simple graphics package but can also be used to create simple GUI's.

The turtle module is an implementation of turtle graphics and uses tkinter for the creation of the underlying graphics. Turtle graphics dates back to the 1960's and was part of the Logo programming language. This chapter provides an introduction to using the graphics capabilities of the turtle module and demonstrates the creation of simple images and simple GUI's for games and applications. In addition to helping you gain practical programming skills, learning to use turtle graphics is fun and it enables you to use Python to be visually creative!

Turtle Basics

Among other things, the methods in the turtle module allow us to draw images. The idea behind the turtle part of "turtle graphics" is based on a metaphor. Imagine you have a turtle on a canvas that is holding a pen. The pen can be either up (not touching the canvas) or down (touching the canvas). Now think of the turtle as a robot that you can control by issuing commands. When the pen it holds is down, the turtle leaves a trail when you tell it to move to a new location. When the pen is up, the turtle moves to a new position but no trail is left. In addition to position, the turtle also has a heading, i.e., a direction, of forward movement. The turtle module provides commands that can set the turtle's position and heading, control its forward and backward movement, specify the type of pen it is holding, etc. By controlling the movement and orientation of the turtle as well as the pen it is holding, you can create drawings from the trails the turtle leaves.

Importing Turtle Graphics

In order to start using turtle graphics, we need to import the turtle module. Start Python/IDLE and

type the following:

Importing the turtle module.

```
>>> import turtle as t
```

This imports the turtle module using the identifier t. By importing the module this way we access the methods within the module using t.<object> instead of turtle.<object>. To ensure that the module was properly imported, use the dir() function as shown in Listing

Listing :Using dir() to view the turtle module's methods and attributes.

```
1 >>> dir(t)
2 ['Canvas', 'Pen', 'RawPen', 'RawTurtle', 'Screen', 'ScrolledCanvas',
3 'Shape', 'TK', 'TNavigator', 'TPen', 'Tbuffer', 'Terminator',
4 'Turtle', 'TurtleGraphicsError', 'TurtleScreen', 'TurtleScreenBase',
5 'Vec2D', '_CFG', '_LANGUAGE', '_Root', '_Screen', '_TurtleImage',
6 '__all__', '__builtins__', '__cached__', '__doc__', '__file__',
7 ... <<MANY LINES OF OUTPUT DELETED>>
8 'window_width', 'write', 'write_docstringdict', 'xcor', 'ycor']
```

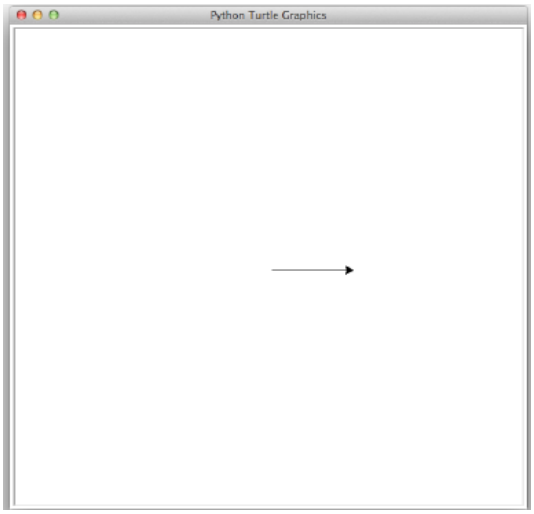
The list returned by the dir() function in Listing has been truncated—in all, there are 173 items in this list. To learn more about any of these attributes or methods, you can use the help() function. For example, to learn about the forward() method, enter the statement shown in line 1 of Listing

```
>>> help(t.forward)
```

Your First Drawing

Let's begin by telling our turtle to draw a line. Try entering the command shown.For drawing a line with a length of 100 units.

```
>>> t.fd(100)
```



As shown in figure, a graphics window should appear in which you see a small arrow 100 units to the right of the center of the window. A thin black line is drawn from the center of the window to the tail of the arrow. The arrow represents our "turtle" and the direction the arrow is pointing indicates the current heading. The fd() method is a shorthand for the forward() method—the two

methods are identical. fd() takes one integer argument that specifies the number of units you want to move the turtle forward in the direction of the current heading. If you provide a negative argument, the turtle moves backwards the specified amount. Alternatively, to move backward one can call either backward(), back(), or bk().

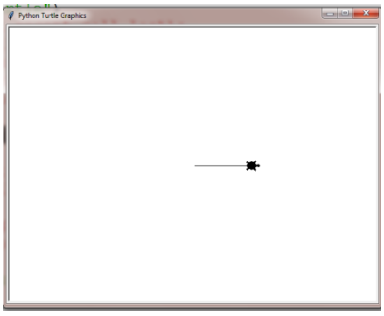
The default shape for our turtle is an arrow but if we wanted to have it look like a turtle we

could type the command shown in Listing

Listing :Changing the shape of the turtle.

```
>>> t.shape("turtle")
```

This replaces the arrow with a small turtle. We can change the shape of our turtle to a number of other built in shapes using the shape() method.



Even though our turtle's shape appears on the graphics window, the turtle is not truly part of our drawing. The shape of the turtle is there to help you see the turtle's current position and heading, but you need to issue other commands, such as fd(), to create a drawing. If you have created a masterpiece and you no longer want to see the turtle in your graphics window, you can enter the command shown. Command to hide the turtle's shape from the screen.

```
>>> t.hideturtle()
```

This hides the image that currently represents the turtle. In fact, you can continue to create lines even when the turtle's shape is hidden, but you will not be able to see the turtle's current position nor its heading. If you want to see the turtle again, simply issue the command shown. Making the turtle visible.

```
>>> t.showturtle()
```

The turtle's heading can be controlled using one of three methods: left(), right(), and setheading(); or the shorter aliases of lt(), rt(), and seth(), respectively. left() and right() turn the turtle either to the left or right, respectively, by the number of degrees given as the argument. These turns are relative to the turtle's current heading. So, for example, left(45) causes the turtle to turn 45 degrees to the left. On the other hand, setheading() and seth() set the absolute heading of the turtle. A heading of 0 is horizontally to the right (i.e., east), 90 is up (i.e., north), 135 is up and to the left (i.e., northwest), and so on. After doing this you should see a square drawn in the graphics window as shown in

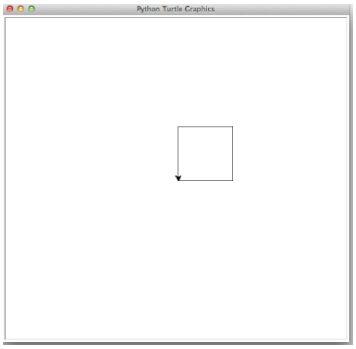


Figure: A square box.

Commands to change the turtle's heading and draw a square box.

```
import turtle as t

t.fd(100)
t.left(90)
t.fd(100)
t.left(90)
t.fd(100)
t.left(90)
t.fd(100)
```

What if we want to change the location of the turtle without generating a line? We can accomplish this by calling the method penup() before we enter commands to move the turtle. To re-enable drawing, we call the method pendown().

We can also move the turtle to a specific position within the graphics window by using the setposition() method (or its aliases setpos() and goto()). setposition()'s arguments are the desired x and y values. The change of position does not affect the turtle's heading. If the pen is down, when you call setposition() (or its aliases), a straight line is drawn from that starting point to the position specified by the arguments. To demonstrate the use of penup(), pendown(), and setposition(), issue the commands shown in Listing. The resulting image is shown.

Listing : Using penup() and setposition() to move the turtle without making a line.

```
import turtle as t
t.fd(100)
t.left(90)
t.fd(100)
t.left(90)
t.fd(100)
t.left(90)
t.fd(100)
t.left(90)
t.fd(100)
t.penup()
t.setposition(100, -100)
t.pendown()
t.fd(130)
```

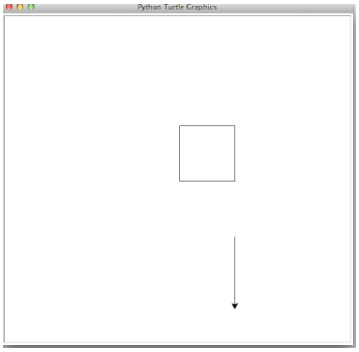


Figure : Result of moving the turtle without drawing a line and then, once at the new location, drawing a line

If we want to erase everything that we previously drew, we can use either the clear() or reset() methods. clear() clears the drawing from the graphics window but it leaves the turtle in its current position with its current heading. reset() clears the drawing and also returns the turtle to its starting position in the center of the screen. To illustrate the behavior of clear(), enter the statement shown. Using the clear() method to clear the image.

```
>>> t.clear()
```

You should now see that the drawing that our turtle generated has been cleared from the screen but the turtle is still in the state that you last specified. To demonstrate what reset() does, enter the command shown.

```
>>> t.reset()
```

The turtle is moved back to the center of the screen with its original heading. Note that we do not need to call clear() before we call reset(). reset() will also clear the drawing—in the above example they were done sequentially solely for demonstrating their behavior.

Basic Shapes and Using Iteration to Generate Graphics

The commands we used to draw a square box would be rather cumbersome to type repeatedly if we wanted to draw more than one box. Observe that we are typing the same commands four times. As you already know, this sort of iteration can be accomplished in a much simpler way using a for-loop. Let's create a function called square() that draws a square using a for-loop. The function takes one argument which is the length of the square's sides. Enter the commands in Listing that define this function and then call it three times, each time with a different length. The result should be the squares that are shown in fig. A function that draws a square using a for-loop.

```
import turtle as t
def square(length):
        for i in range(4):
                t.fd(length)
                t.left(90)
square(60)
square(100)
square(200)
```

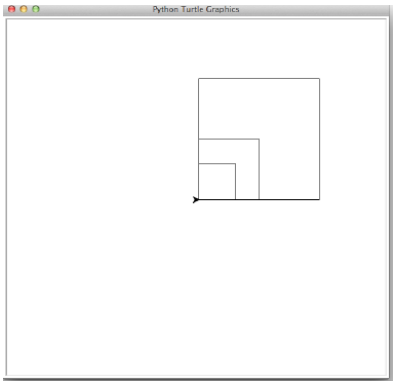


Figure: Drawing multiple boxes

Building a square from straight lines is relatively straightforward, but what if we want to draw circles? Turtle provides a method called circle() that can be used to tell the turtle to draw a complete circle or only a part of a circle, i.e., an arc. The circle() method has one mandatory argument which is the radius of the circle. Optional arguments specify the "extent," which is the degrees of arc that are drawn, and the "steps," which are the number of straight-line segments used to approximate the circle. If the radius is positive, the circle is drawn (starting from the current position) by turning to the left (counterclockwise). If the radius is negative, the circle is drawn (starting from the current position) by turning to the right (clockwise). To demonstrate this, enter the commands shown. After issuing these commands, the graphics window should appear as shown in Fig.

Drawing circles using the circle() method.

```
import turtle as t
t.reset()       # Remove previous drawings and reset turtle.
t.circle(100)   # Draw circle counterclockwise with radius
100.
t.circle(-50)   # Draw circle clockwise with radius 50.
```

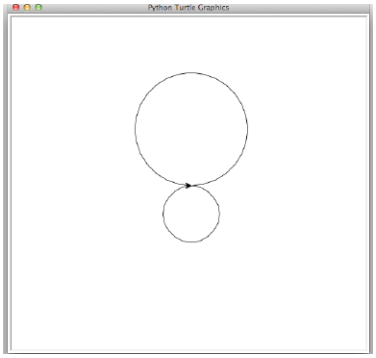


Figure: The circles drawn by the code shown left side

Controlling the Turtle's Animation Speed

If you have been watching the turtle move in the graphics window as you issue commands, you will have noticed that the turtle goes through its motions relatively slowly. Sometimes watching the turtle move can be helpful but there are ways that you can speed up the movement (and hence speed up the drawing process). There is a speed() method for which the argument specifies the speed as an integer between 0 and 10. A value of 1 is the slowest speed. Speeds generally increase with increasing arguments so that 2 is faster than 1, three is faster than 2, and so on. However, rather than 10 being be fastest speed, it is actually the second to the fastest speed—0 is the fastest speed. The default speed is 3. To demonstrate this, issue the commands shown.

Speeding up the animation by using the speed() method.

```
import turtle as t
t.reset()       # Remove previous drawings and reset turtle.
t.speed(1)
t.circle(100)   # Draw circle counterclockwise with radius
100.
t.circle(-50)   # Draw circle clockwise with radius 50.
```

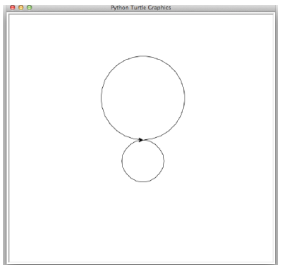


Figure: The circles drawn by the code shown left side

The image should be the same as, but it should render noticeably faster than it did previously. However, even though supplying the speed() method with an argument of 0 makes the animation faster, it is still quite slow if we want to draw a more complex drawing. In order to make our drawing appear almost immediately we can make use of the tracer() method. tracer() takes two arguments. One controls how often screens should be updated and the other controls the delay between these update. To obtain the fastest possible rendering, both these arguments should be set to zero as shown.

Turning off the animation with tracer().

```
>>> tracer(0, 0)
```

By calling tracer() with bother arguments set to zero, we are essentially turning off all animation and our drawings will be drawn "immediately." However, if we turn the animation off in this way we need to explicitly update the image with the update() method after we are doneissuing drawing commands.

If you want to reset tracer() to its original settings, its arguments should be 1 and 10, as shown.

Restoring animation to the default settings.

```
>>> tracer(1, 10)
```

Colors and Filled Shapes

Now that we know how to create some basic shapes, let's explore how we can create more complex images. In order to change the color of our turtle's pen we can use the color() method.

Changing the color of the turtle's pen.

```
1 >>> t.reset()

2 >>> t.color("blue")
```

This change the turtle's pen to blue. There are actually numerous ways of specifying a color in Python's implementation of turtle graphics. We may specify a color via a string. Alternatively, we can specify a color by providing numeric values the specify the amount of red, green, and blue. To learn more about the use of colors, use the help() function to read about t.color or t.pencolor.

We can also change the thickness of the turtle's pen by using the pensize() method and passing it an integer argument that specifies the thickness. After issuing following commands you will see that a thick blue line has been drawn in the graphics window as shown in Fig

```
import turtle as t
t.reset()
t.color("blue")

t.pensize(100)
t.fd(100)
```

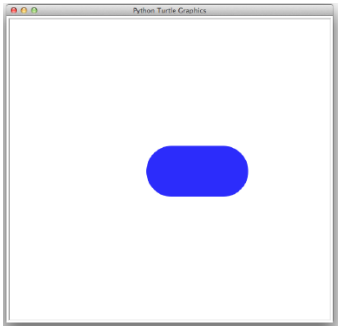


Figure: A blue line with a thickness and width of 100

We can also change the background color of the graphics window. To demonstrate this, you should enter the following commands: Changing the background color of the window.

```
>>> t.bgcolor("blue")          # Change the background to blue.

>>> t.bgcolor("white")                 # Change it back to white.
```

Let's take what we have learned so far and draw a more complex image. Enter the following commands shown Listing 13.20. The for-loop that starts in line 3 sets the heading to angles

between 0 and 345 degrees, inclusive, in 15 degree increments. For each of these headings it draws a circle with a radius of 100. (Note that a heading of 360 is the same as a heading of 0, so that why a circle is not draw with a heading of 360 degrees.) The resulting image is shown in Fig. Creating a red flower.

```
import turtle as t
t.reset()
t.color("red")
for angle in range(0, 360,
15):
    t.seth(angle)
    t.circle(100)
```

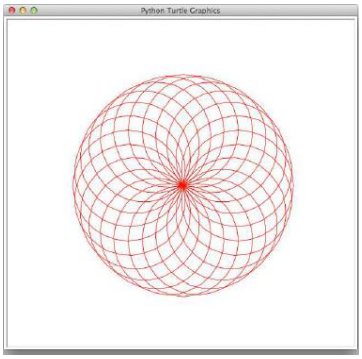


Figure: A red flower using circle()

We can also use iteration to change the color and thickness of the turtle's pen more dynamically. As an example, try typing the commands shown. This should result in the image shown.

```
import turtle as t
colors = ["blue", "green", "purple", "cyan",
"magenta", "violet"]
t.reset()
t.tracer(0, 0)
for i in range(45):
    t.color(colors[i % 6])
    t.pendown()
    t.fd(2 + i * 5)
    t.left(45)
    t.width(i)
    t.penup()

t.update()
```

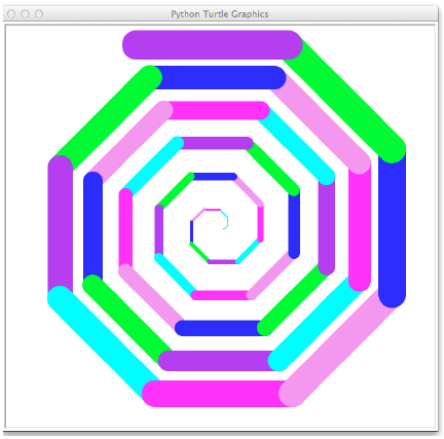


Figure: A colorful spiral.

Filled Shapes

We can also fill the shapes we draw by using the methods begin fill() and endfill(). To fill a shape, we must call first begin fill(), then issue commands to draw the desired shape, and finally call end fill(). When we call end fill() the shape will be filled with the currently set color. To demonstrate this, try entering the commands as shown. After entering these commands you should see a green filled box in the graphics window as shown.

```
import turtle as t
t.reset()
t.color("green")
t.begin_fill()
t.fd(100)
t.left(90)
t.fd(100)
t.left(90)
t.fd(100)
t.left(90)
t.fd(100)
t.end_fill()
```

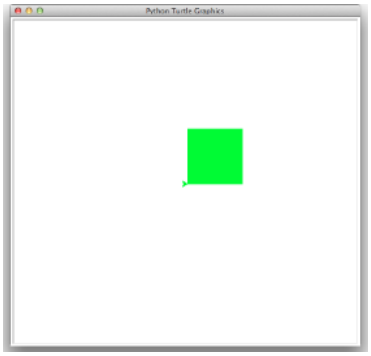


Figure : A filled green box

8. GUI PROGRAMMING

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.

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

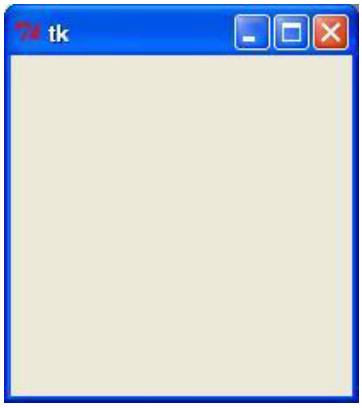
```
#!/usr/bin/python

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:

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

1. Button

The Button widget is used to add buttons in a Python application. These buttons can display text or images that convey the purpose of the buttons. You can attach a function or a method to a button which is called automatically when you click the button.

Syntax

Here is the simple syntax to create this widget:

w = Button (master, option=value, ...)

Parameters

 master: This represents the parent window.

 options: Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

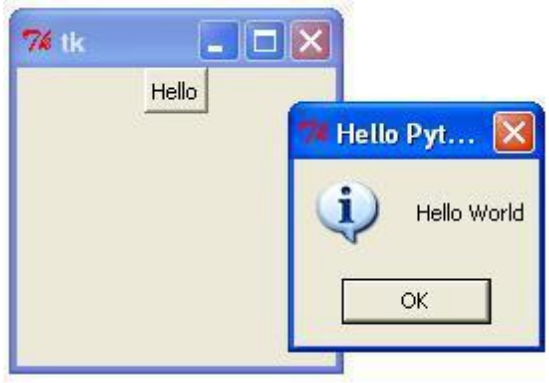
Option	Description
activebackground	Background color when the button is under the cursor.
activeforeground	Foreground color when the button is under the cursor.
bd	Border width in pixels. Default is 2.
bg	Normal background color.
command	Function or method to be called when the button is clicked.
fg	Normal foreground (text) color.
font	Text font to be used for the button's label.
height	Height of the button in text lines (for textual buttons) or pixels (for images).
highlightcolor	The color of the focus highlight when the widget has focus.
image	Image to be displayed on the button (instead of text).
justify	How to show multiple text lines: LEFT to left-justify each line; CENTER to center them; or RIGHT to right-justify.
padx	Additional padding left and right of the text.
pady	Additional padding above and below the text.
relief	Relief specifies the type of the border. Some of the values are SUNKEN, RAISED, GROOVE, and RIDGE.
state	Set this option to DISABLED to gray out the button and make it unresponsive. Has the value ACTIVE when the mouse is over it. Default is NORMAL.
underline	Default is -1, meaning that no character of the text on the button will be underlined. If nonnegative, the corresponding text character will be underlined.
width	Width of the button in letters (if displaying text) or pixels (if

	displaying an image).
wraplength	If this value is set to a positive number, the text lines will be wrapped to fit within this length.

Example

```
import tkinter
from tkinter import messagebox
top = tkinter.Tk()
def helloCallBack():
    messagebox.showinfo( "Hello Python", "Hello World")
    B = tkinter.Button(top, text ="Hello", command = helloCallBack)
    B.pack()
    top.mainloop()
helloCallBack()
```

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



2. Canvas

The Canvas is a rectangular area intended for drawing pictures or other complex layouts. You can place graphics, text, widgets2. or frames on a Canvas.

Syntax

Here is the simple syntax to create this widget:

w = Canvas (master, option=value, ...)

Parameters

- **master:** This represents the parent window.
- **options:** Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Option	Description
bd	Border width in pixels. Default is 2.
bg	Normal background color.
confine	If true (the default), the canvas cannot be scrolled outside of the scrollregion.
cursor	Cursor used in the canvas like *arrow, circle, dot etc.*
height	Size of the canvas in the Y dimension.
highlightcolor	Color shown in the focus highlight.
relief	Relief specifies the type of the border. Some of the values are SUNKEN, RAISED, GROOVE, and RIDGE.
scrollregion	A tuple (w, n, e, s) that defines over how large an area the canvas can be scrolled, where w is the left side, n the top, e the right side, and s the bottom.

width	Size of the canvas in the X dimension.
xscrollincrement	If you set this option to some positive dimension, the canvas can be positioned only on multiples of that distance, and the value will be used for scrolling by scrolling units, such as when the user clicks on the arrows at the ends of a scrollbar.
xscrollcommand	If the canvas is scrollable, this attribute should be the .set() method of the horizontal scrollbar.
yscrollincrement	Works like xscrollincrement, but governs vertical movement.
yscrollcommand	If the canvas is scrollable, this attribute should be the .set() method of the vertical scrollbar.

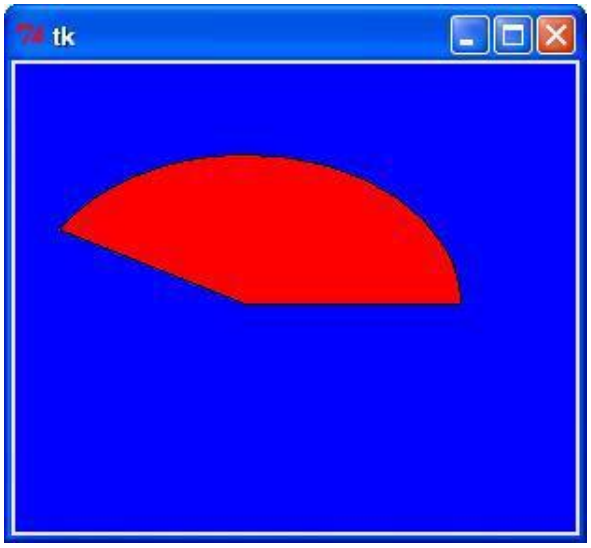
The Canvas widget can support the following standard items:
arc : Creates an arc item, which can be a chord, a pieslice or a simple arc.
coord = 10, 50, 240, 210
arc = Canvas.create_arc(coord, start=0, extent=150, fill="blue")

image : Creates an image item, which can be an instance of either the BitmapImage or the PhotoImage classes.
filename = PhotoImage(file = "sunshine.gif")
image = canvas.create_image(50, 50, anchor=NE, image=filename)

line Creates a line item.
line = Canvas.create_line(x0, y0, x1, y1, ..., xn, yn, options)
oval **:**Creates a circle or an ellipse at the given coordinates. It takes two pairs of coordinates; the top left and bottom right corners of the bounding rectangle for the oval.
oval = Canvas.create_oval(x0, y0, x1, y1, options)
polygon Creates a polygon item that must have at least three vertices.
oval = Canvas.create_polygon(x0, y0, x1, y1,...xn, yn, options)
Example
Try the following example yourself:

```
import tkinter
from tkinter import messagebox
top = tkinter.Tk()
C = tkinter.Canvas(top, bg="blue", height=250, width=300)
coord = 10, 50, 240, 210
arc = C.create_arc(coord, start=0, extent=150, fill="red")
C.pack()
top.mainloop()
```

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



3. Checkbutton

The Checkbutton widget is used to display a number of options to a user as toggle buttons. The user can then select one or more options by clicking the button corresponding to each option. You can also display images in place of text.

Syntax

Here is the simple syntax to create this widget:

w = Checkbutton (master, option, ...)

Parameters

 master: This represents the parent window.

options: Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Option	Description
activebackground	Background color when the checkbutton is under the cursor.
activeforeground	Foreground color when the checkbutton is under the cursor.
bg	The normal background color displayed behind the label and indicator.
bitmap	To display a monochrome image on a button.
bd	The size of the border around the indicator. Default is 2 pixels.
command	A procedure to be called every time the user changes the state of this checkbutton.
cursor	If you set this option to a cursor name (*arrow, dot etc.), the mouse cursor will change to that pattern when it is over the checkbutton.*
disabledforeground	The foreground color used to render the text of a disabled checkbutton. The default is a stippled version of the default foreground color.
font	The font used for the text.
fg	The color used to render the text.
height	The number of lines of text on the checkbutton. Default is 1.
highlightcolor	The color of the focus highlight when the checkbutton has the focus.

image	To display a graphic image on the button.
justify	If the text contains multiple lines, this option controls how the text is justified: CENTER, LEFT, or RIGHT.
offvalue	Normally, a checkbutton's associated control variable will be set to 0 when it is cleared (off). You can supply an alternate value for the off state by setting offvalue to that value.
onvalue	Normally, a checkbutton's associated control variable will be set to 1 when it is set (on). You can supply an alternate value for the on state by setting onvalue to that value.
padx	How much space to leave to the left and right of the checkbutton and text. Default is 1 pixel.
pady	How much space to leave above and below the checkbutton and text. Default is 1 pixel.
relief	With the default value, relief=FLAT, the checkbutton does not stand out from its background. You may set this option to any of the other styles
selectcolor	The color of the checkbutton when it is set. Default is selectcolor="red".
selectimage	If you set this option to an image, that image will appear in the checkbutton when it is set.
state	The default is state=NORMAL, but you can use state=DISABLED to gray out the control and make it unresponsive. If the cursor is currently over the checkbutton, the state is ACTIVE.
text	The label displayed next to the checkbutton. Use newlines ("\n") to display multiple lines of text.
underline	With the default value of -1, none of the characters of the text label are underlined. Set this option to the index of a character in the text (counting from zero) to underline that character.
variable	The control variable that tracks the current state of the checkbutton. Normally this variable is an IntVar, and 0 means cleared and 1 means set, but see the offvalue and onvalue options above.
width	The default width of a checkbutton is determined by the size of the displayed image or text. You can set this option to a number of characters and the checkbutton will always have room for that many characters.
wraplength	Normally, lines are not wrapped. You can set this option to a number of characters and all lines will be broken into pieces no longer than that number.

Following are commonly used methods for this widget: **Medthod**

Methods	**Description**

deselect()	Clears (turns off) the checkbutton.
flash()	Flashes the checkbutton a few times between its active and normal colors, but leaves it the way it started.
invoke()	You can call this method to get the same actions that would occur if the user clicked on the checkbutton to change its state.
select()	Sets (turns on) the checkbutton.
toggle()	Clears the checkbutton if set, sets it if cleared.

Example

Try the following example yourself:

```
from tkinter import *
from tkinter import messagebox
import tkinter
top = tkinter.Tk()
CheckVar1 = IntVar()
CheckVar2 = IntVar()
C1 = Checkbutton(top, text = "Music", variable = CheckVar1, onvalue = 1, offvalue = 0, height=5,
width = 20)
C2 = Checkbutton(top, text = "Video", variable = CheckVar2, onvalue = 1, offvalue = 0, height=5,
width = 20)
C1.pack()
C2.pack()
top.mainloop()
```

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

4. Entry

The Entry widget is used to accept single-line text strings from a user.

- If you want to display multiple lines of text that canbe edited, then you should use the *Text* widget.
- If you want to display one or more lines of text that cannot be modified by the user, then you should use the *Label* widget.

Syntax

Here is the simple syntax to create this widget:

```
w = Entry( master, option, ... )
```

Parameters

- **master:** This represents the parent window.

options: Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Option	Description
bg	The normal background color displayed behind the label and indicator.
bd	The size of the border around the indicator. Default is 2 pixels.
command	A procedure to be called every time the user changes the state of this checkbutton.
cursor	If you set this option to a cursor name (*arrow, dot etc.*), the mouse cursor will change to that pattern when it is over the checkbutton.
font	The font used for the text.
exportselection	By default, if you select text within an Entry widget, it is automatically exported to the clipboard. To avoid this exportation, use exportselection=0.
fg	The color used to render the text.
highlightcolor	The color of the focus highlight when the checkbutton has the focus.
justify	If the text contains multiple lines, this option controls how the text is justified: CENTER, LEFT, or RIGHT.
relief	With the default value, relief=FLAT, the checkbutton does not stand out from its background. You may set this option to any of the other styles
selectbackground	The background color to use displaying selected text.
selectborderwidth	The width of the border to use around selected text. The default is one pixel.
selectforeground	The foreground (text) color of selected text.
show	Normally, the characters that the user types appear in the entry. To make a .password. entry that echoes each character as an asterisk, set show="*".
state	The default is state=NORMAL, but you can use state=DISABLED to gray out the control and make it unresponsive. If the cursor is currently over the checkbutton, the state is ACTIVE.
textvariable	In order to be able to retrieve the current text from your entry widget, you must set this option to an instance of the StringVar class.
width	The default width of a checkbutton is determined by the size of the displayed image or text. You can set this option to a number of characters and the checkbutton will always have room for that many characters.
xscrollcommand	If you expect that users will often enter more text than the onscreen size of the widget, you can link your entry widget to a scrollbar.

Methods : Following are commonly used methods for this widget:

Medthod	Description
delete (first, last=None)	Deletes characters from the widget, starting with the one at index first, up to but not including the character at position last. If the second argument is omitted, only the single character at position first is deleted.
get()	Returns the entry's current text as a string.
icursor (index)	Set the insertion cursor just before the character at the given

	index.
index (index)	Shift the contents of the entry so that the character at the given index is the leftmost visible character. Has no effect if the text fits entirely within the entry.
insert (index, s)	Inserts string s before the character at the given index.
select_adjust (index)	This method is used to make sure that the selection includes the character at the specified index.
select_clear()	Clears the selection. If there isn't currently a selection, has no effect.
select_from (index)	Sets the ANCHOR index position to the character selected by index, and selects that character.
select_present()	If there is a selection, returns true, else returns false.
select_range (start, end)	Sets the selection under program control. Selects the text starting at the start index, up to but not including the character at the end index. The start position must be before the end position.
select_to (index)	Selects all the text from the ANCHOR position up to but not including the character at the given index.
xview (index)	This method is useful in linking the Entry widget to a horizontal scrollbar.
xview_scroll (number, what)	Used to scroll the entry horizontally. The what argument must be either UNITS, to scroll by character widths, or PAGES, to scroll by chunks the size of the entry widget. The number is positive to scroll left to right, negative to scroll right to left.

Example

Try the following example yourself:

```
from tkinter import *
top = Tk()
L1 = Label(top, text="User Name")
L1.pack( side = LEFT)
E1 = Entry(top, bd =5)
E1.pack(side = RIGHT)
top.mainloop()
```

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

Frame:

The Frame widget is very important for the process of grouping and organizing other widgets in a somehow friendly way. It works like a container, which is responsible for arranging the position of other widgets. It uses rectangular areas in the screen to organize the layout and to provide padding of these widgets. A frame can also be used as a foundation class to implement complex widgets.

Syntax

Here is the simple syntax to create this widget:
w = Frame (master, option, ...)
Parameters:
 master: This represents the parent window.
 options: Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Option	Description
bg	The normal background color displayed behind the label and indicator.
bd	The size of the border around the indicator. Default is 2 pixels.
cursor	If you set this option to a cursor name (*arrow, dot etc.*), the mouse cursor will change to that pattern when it is over the checkbutton.
height	The vertical dimension of the new frame.
highlightbackground	Color of the focus highlight when the frame does not have focus.
highlightcolor	Color shown in the focus highlight when the frame has the focus.
highlightthickness	Thickness of the focus highlight.
relief	With the default value, relief=FLAT, the checkbutton does not stand out from its background. You may set this option to any of the other styles
width	The default width of a checkbutton is determined by the size of the displayed image or text. You can set this option to a number of characters and the checkbutton will always have room for that many characters.

Try the following example yourself:

```
from tkinter import *
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 )
bluebutton = Button(frame, text="Blue", fg="blue")
bluebutton.pack( side = LEFT )
blackbutton = Button(bottomframe, text="Black", fg="black")
blackbutton.pack( side = BOTTOM)
root.mainloop()
```

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

6. Label

This widget implements a display box where you can place text or images. The text displayed by this widget can be updated at any time you want. It is also possible to underline part of the text (like to identify a keyboard shortcut) and span the text across multiple lines.

Syntax

Here is the simple syntax to create this widget:

w = Label (master, option, ...)

Parameters

- **master:** This represents the parent window.
- **options:** Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Option	Description
anchor	This options controls where the text is positioned if the widget has more space than the text needs. The default is anchor=CENTER, which centers the text in the available space.
bg	The normal background color displayed behind the label and indicator.
bitmap	Set this option equal to a bitmap or image object and the label will display that graphic.
bd	The size of the border around the indicator. Default is 2 pixels.
cursor	If you set this option to a cursor name (*arrow*, *dot etc.*), the mouse cursor will change to that pattern when it is over the checkbutton.
font	If you are displaying text in this label (with the text or textvariable option, the font option specifies in what font that text will be displayed.
fg	If you are displaying text or a bitmap in this label, this option specifies the color of the text. If you are displaying a bitmap, this is the color that will appear at the position of the 1-bits in the bitmap.
height	The vertical dimension of the new frame.
image	To display a static image in the label widget, set this option to an image object.
justify	Specifies how multiple lines of text will be aligned with respect to each other: LEFT for flush left, CENTER for centered (the default), or RIGHT for right-justified.
padx	Extra space added to the left and right of the text within the widget. Default is 1.
pady	Extra space added above and below the text within the widget. Default is 1.
relief	Specifies the appearance of a decorative border around the label. The default is FLAT; for other values.
text	To display one or more lines of text in a label widget, set this option to a string containing the text. Internal newlines ("\n") will force a line break.
textvariable	To slave the text displayed in a label widget to a control variable of class StringVar, set this option to that variable.
underline	You can display an underline (_) below the nth letter of the text, counting from 0, by setting this option to n. The default is underline=-1, which means no

	underlining.
width	Width of the label in characters (not pixels!). If this option is not set, the label will be sized to fit its contents.
wraplength	You can limit the number of characters in each line by setting this option to the desired number. The default value, 0, means that lines will be broken only at newlines.

Example

Try the following example yourself:

```
from tkinter import *
root = Tk()
var = StringVar()
label = Label( root, textvariable=var, relief=RAISED )
var.set("Hey!? How are you doing?")
label.pack()
root.mainloop()
```

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

7. Listbox

The Listbox widget is used to display a list of items from which a user can select a number of items

Syntax

Here is the simple syntax to create this widget:

```
w = Listbox ( master, option, ... )
```

Parameters

- **master:** This represents the parent window.
- **options:** Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Option	Description
bg	The normal background color displayed behind the label and indicator.
bd	The size of the border around the indicator. Default is 2 pixels.
cursor	The cursor that appears when the mouse is over the listbox.
font	The font used for the text in the listbox.
fg	The color used for the text in the listbox.
height	Number of lines (not pixels!) shown in the listbox. Default is 10.
highlightcolor	Color shown in the focus highlight when the widget has the focus.
highlightthickness	Thickness of the focus highlight.
relief	Selects three-dimensional border shading effects. The default is SUNKEN.
selectbackgroun	The background color to use displaying selected text.

d	
selectmode	Determines how many items can be selected, and how mouse drags affect the selection: BROWSE: Normally, you can only select one line out of a listbox. If you click on an item and then drag to a different line, the selection will follow the mouse. This is the default. SINGLE: You can only select one line, and you can't drag the mouse.wherever you click button 1, that line is selected. MULTIPLE: You can select any number of lines at once. Clicking on any line toggles whether or not it is selected. EXTENDED: You can select any adjacent group of lines at once by clicking on the first line and dragging to the last line.
width	The width of the widget in characters. The default is 20.
xscrollcommand	If you want to allow the user to scroll the listbox horizontally, you can link your listbox widget to a horizontal scrollbar.
yscrollcommand	If you want to allow the user to scroll the listbox vertically, you can link your listbox widget to a vertical scrollbar.

Methods on listbox objects include: Option activate (index)

Methods :	**Description**
activate (index)	Selects the line specifies by the given index.
curselection()	Returns a tuple containing the line numbers of the selected element or elements, counting from 0. If nothing is selected, returns an empty tuple.
delete (first, last=None)	Deletes the lines whose indices are in the range [first, last]. If the second argument is omitted, the single line with index first is deleted.
get (first, last=None)	Returns a tuple containing the text of the lines with indices from first to last, inclusive. If the second argument is omitted, returns the text of the line closest to first.
index (i)	If possible, positions the visible part of the listbox so that the line containing index i is at the top of the widget.
insert (index, *elements)	Insert one or more new lines into the listbox before the line specified by index. Use END as the first argument if you want to add new lines to the end of the listbox.
nearest (y)	Return the index of the visible line closest to the y-coordinate y relative to the listbox widget.
see (index)	Adjust the position of the listbox so that the line referred to by index is visible.

size()	Returns the number of lines in the listbox.
xview()	To make the listbox horizontally scrollable, set the command option of the associated horizontal scrollbar to this method.
xview_moveto (fraction)	Scroll the listbox so that the leftmost fraction of the width of its longest line is outside the left side of the listbox. Fraction is in the range [0,1].
xview_scroll (number, what)	Scrolls the listbox horizontally. For the what argument, use either UNITS to scroll by characters, or PAGES to scroll by pages, that is, by the width of the listbox. The number argument tells how many to scroll.
yview()	To make the listbox vertically scrollable, set the command option of the associated vertical scrollbar to this method.
yview_moveto (fraction)	Scroll the listbox so that the top fraction of the width of its longest line is outside the left side of the listbox. Fraction is in the range [0,1].
yview_scroll (number, what)	Scrolls the listbox vertically. For the what argument, use either UNITS to scroll by lines, or PAGES to scroll by pages, that is, by the height of the listbox. The number argument tells how many to scroll

Example

Try the following example yourself:

```
from tkinter import *
from tkinter import messagebox
import tkinter
top = Tk()
Lb1 = Listbox(top)
Lb1.insert(1, "Python")
Lb1.insert(2, "Perl")
Lb1.insert(3, "C")
Lb1.insert(4, "PHP")
Lb1.insert(5, "JSP")
Lb1.insert(6, "Ruby")
Lb1.pack()
top.mainloop()
```

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

8. MenuButton

A menubutton is the part of a drop-down menu that stays on the screen all the time. Every menubutton is associated with a Menu widget that can display the choices for that menubutton when the user clicks on it.

Syntax

Here is the simple syntax to create this widget:

w = Menubutton (master, option, ...)

Parameters

- **master:** This represents the parent window.
- **options:** Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Option	Description
activebackground	The background color when the mouse is over the menubutton.
activeforeground	The foreground color when the mouse is over the menubutton.
anchor	This options controls where the text is positioned if the widget has more space than the text needs. The default is anchor=CENTER, which centers the text.
bg	The normal background color displayed behind the label and indicator.
bitmap	To display a bitmap on the menubutton, set this option to a bitmap name.
bd	The size of the border around the indicator. Default is 2 pixels.
cursor	The cursor that appears when the mouse is over this menubutton.
direction	Set direction=LEFT to display the menu to the left of the button; use direction=RIGHT to display the menu to the right of the button; or use direction='above' to place the menu above the button.
disabledforeground	The foreground color shown on this menubutton when it is disabled.
fg	The foreground color when the mouse is not over the menubutton.
height	The height of the menubutton in lines of text (not pixels!). The default is to fit the menubutton's size to its contents.
highlightcolor	Color shown in the focus highlight when the widget has the focus.
image	To display an image on this menubutton,
justify	This option controls where the text is located when the text doesn't fill the menubutton: use justify=LEFT to left-justify the text (this is the default); use justify=CENTER to center it, or justify=RIGHT to right-justify.
menu	To associate the menubutton with a set of choices, set this option to the Menu object containing those choices. That menu object must have been created by passing the associated menubutton to the constructor as its first argument.
padx	How much space to leave to the left and right of the text of the menubutton. Default is 1.
pady	How much space to leave above and below the text of the menubutton. Default is 1.
relief	Selects three-dimensional border shading effects. The default is RAISED.
state	Normally, menubuttons respond to the mouse. Set state=DISABLED to gray out the menubutton and make it unresponsive.

text	To display text on the menubutton, set this option to the string containing the desired text. Newlines ("\n") within the string will cause line breaks.
textvariable	You can associate a control variable of class StringVar with this menubutton. Setting that control variable will change the displayed text.
underline	Normally, no underline appears under the text on the menubutton. To underline one of the characters, set this option to the index of that character.
width	The width of the widget in characters. The default is 20.
wraplength	Normally, lines are not wrapped. You can set this option to a number of characters and all lines will be broken into pieces no longer than that number.

Example
Try the following example yourself:

```
from tkinter import *
from tkinter import messagebox
import tkinter
top = Tk()
mb= Menubutton ( top, text="condiments", relief=RAISED )
mb.grid()
mb.menu = Menu ( mb, tearoff = 0 )
mb["menu"] = mb.menu
mayoVar = IntVar()
ketchVar = IntVar()
mb.menu.add_checkbutton ( label="mayo",
variable=mayoVar )
mb.menu.add_checkbutton ( label="ketchup",
variable=ketchVar )
mb.pack()
top.mainloop()
```

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

9. Menu
The goal of this widget is to allow us to create all kinds of menus that can be used by our applications. The core functionality provides ways to create three menu types: pop-up, toplevel and pull-down.

It is also possible to use other extended widgets to implement new types of menus, such as the *OptionMenu* widget, which implements a special type that generates a pop-up list of items within a selection.

Syntax

Here is the simple syntax to create this widget:

w = Menu (master, option, ...)

Parameters

 master: This represents the parent window.

 options: Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Option	Description
activebackground	The background color that will appear on a choice when it is under the mouse.
activeborderwidth	Specifies the width of a border drawn around a choice when it is under the mouse. Default is 1 pixel.
activeforeground	The foreground color that will appear on a choice when it is under the mouse.
bg	The background color for choices not under the mouse.
bd	The width of the border around all the choices. Default is 1.
cursor	The cursor that appears when the mouse is over the choices, but only when the menu has been torn off.
disabledforeground	The color of the text for items whose state is DISABLED.
font	The default font for textual choices.
fg	The foreground color used for choices not under the mouse.
postcommand	You can set this option to a procedure, and that procedure will be called every time someone brings up this menu.
relief	The default 3-D effect for menus is relief=RAISED.
image	To display an image on this menubutton.
selectcolor	Specifies the color displayed in checkbuttons and radiobuttons when they are selected.
tearoff	Normally, a menu can be torn off, the first position (position 0) in the list of choices is occupied by the tear-off element, and the additional choices are added starting at position 1. If you set tearoff=0, the menu will not have a tear-off feature, and choices will be added starting at position 0.
title	Normally, the title of a tear-off menu window will be the same as the text of the menubutton or cascade that lead to this menu. If you want to change the title of that window, set the title option to that string.

Methods: These methods are available on Menu objects

Option	Description
add_command (options)	Adds a menu item to the menu.
add_radiobutton(options)	Creates a radio button menu item.

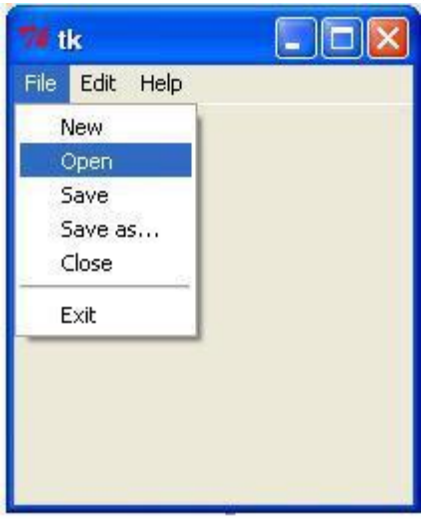
add_checkbutton(options)	Creates a check button menu item.
add_cascade(options)	Creates a new hierarchical menu by associating a given menu to a parent menu
add_separator()	Adds a separator line to the menu.
add(type, options)	Adds a specific type of menu item to the menu.
delete(startindex [, endindex])	Deletes the menu items ranging from startindex to endindex.
entryconfig(index, options)	Allows you to modify a menu item, which is identified by the index, and change its options.
index(item)	Returns the index number of the given menu item label.
insert_separator (index)	Insert a new separator at the position specified by index.
invoke (index)	Calls the command callback associated with the choice at position index. If a checkbutton, its state is toggled between set and cleared; if a radiobutton, that choice is set.
type (index)	Returns the type of the choice specified by index: either "cascade", "checkbutton", "command", "radiobutton", "separator", or "tearoff".

Example
Try the following example yourself:

```
from tkinter import *
def donothing():
      filewin = Toplevel(root)
      button = Button(filewin, text="Do nothing button")
      button.pack()
root = Tk()
menubar = Menu(root)
filemenu = Menu(menubar, tearoff=0)
filemenu.add_command(label="New", command=donothing)
filemenu.add_command(label="Open", command=donothing)
filemenu.add_command(label="Save", command=donothing)
filemenu.add_command(label="Save as...", command=donothing)
filemenu.add_command(label="Close", command=donothing)
filemenu.add_separator()
filemenu.add_command(label="Exit", command=root.quit)
menubar.add_cascade(label="File", menu=filemenu)
editmenu = Menu(menubar, tearoff=0)
editmenu.add_command(label="Undo", command=donothing)
editmenu.add_separator()
editmenu.add_command(label="Cut", command=donothing)
editmenu.add_command(label="Copy", command=donothing)
editmenu.add_command(label="Paste", command=donothing)
editmenu.add_command(label="Delete", command=donothing)
```

```
editmenu.add_command(label="Select All", command=donothing)
menubar.add_cascade(label="Edit", menu=editmenu)
helpmenu = Menu(menubar, tearoff=0)
helpmenu.add_command(label="Help Index", command=donothing)
helpmenu.add_command(label="About...", command=donothing)
menubar.add_cascade(label="Help", menu=helpmenu)
root.config(menu=menubar)
root.mainloop()
```

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



10. Message

This widget provides a multiline and noneditable object that displays texts, automatically breaking lines and justifying their contents.

Its functionality is very similar to the one provided by the Label widget, except that it can also automatically wrap the text, maintaining a given width or aspect ratio.

Syntax

Here is the simple syntax to create this widget:

w = Message (master, option, ...)

Parameters

- **master:** This represents the parent window.
- **options:** Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Option	Description
anchor	This options controls where the text is positioned if the widget has more space than the text needs. The default is anchor=CENTER, which centers the text in the available space.
bg	The normal background color displayed behind the label and indicator.
bitmap	Set this option equal to a bitmap or image object and the label will display that graphic.
bd	The size of the border around the indicator. Default is 2 pixels.

cursor	If you set this option to a cursor name (*arrow, dot etc.*), the mouse cursor will change to that pattern when it is over the checkbutton.
font	If you are displaying text in this label (with the text or textvariable option, the font option specifies in what font that text will be displayed.
fg	If you are displaying text or a bitmap in this label, this option specifies the color of the text. If you are displaying a bitmap, this is the color that will appear at the position of the 1-bits in the bitmap.
height	The vertical dimension of the new frame.
image	To display a static image in the label widget, set this option to an image object.
justify	Specifies how multiple lines of text will be aligned with respect to each other: LEFT for flush left, CENTER for centered (the default), or RIGHT for right-justified.
padx	Extra space added to the left and right of the text within the widget. Default is 1.
pady	Extra space added above and below the text within the widget. Default is 1.
relief	Specifies the appearance of a decorative border around the label. The default is FLAT; for other values.
text	To display one or more lines of text in a label widget, set this option to a string containing the text. Internal newlines ("\n") will force a line break.
textvariable	To slave the text displayed in a label widget to a control variable of class StringVar, set this option to that variable.
underline	You can display an underline (_) below the nth letter of the text, counting from 0, by setting this option to n. The default is underline=-1, which means no underlining.
width	Width of the label in characters (not pixels!). If this option is not set, the label will be sized to fit its contents.
wraplength	You can limit the number of characters in each line by setting this option to the desired number. The default value, 0, means that lines will be broken only at newlines.

Example

Try the following example yourself:

```
from tkinter import *
root = Tk()
var = StringVar()
label = Message( root, textvariable=var, relief=RAISED )
var.set("Hey!? How are you doing?")
label.pack()
root.mainloop()
```

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

11. Radiobutton

This widget implements a multiple-choice button, which is a way to offer many possible selections to the user and lets user choose only one of them.

In order to implement this functionality, each group of radiobuttons must be associated to the same variable and each one of the buttons must symbolize a single value. You can use the Tab key to switch from one radionbutton to another.

Syntax

Here is the simple syntax to create this widget:

Radiobutton (master, option, ...)

Parameters

 master: This represents the parent window.

 options: Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Option	Description
activebackground	The background color when the mouse is over the radiobutton.
activeforeground	The foreground color when the mouse is over the radiobutton.
anchor	If the widget inhabits a space larger than it needs, this option specifies where the radiobutton will sit in that space. The default is anchor=CENTER.
bg	The normal background color behind the indicator and label.
bitmap	To display a monochrome image on a radiobutton, set this option to a bitmap.
borderwidth	The size of the border around the indicator part itself. Default is 2 pixels.
command	A procedure to be called every time the user changes the state of this radiobutton.
cursor	If you set this option to a cursor name (arrow, dot etc.), the mouse cursor will change to that pattern when it is over the radiobutton.
font	The font used for the text.
fg	The color used to render the text.
height	The number of lines (not pixels) of text on the radiobutton. Default is 1.
highlightbackground	The color of the focus highlight when the radiobutton does not have focus.
highlightcolor	The color of the focus highlight when the radiobutton has the focus.
image	To display a graphic image instead of text for this radiobutton, set this option to an image object.
justify	If the text contains multiple lines, this option controls how the text is justified: CENTER (the default), LEFT, or RIGHT.
padx	How much space to leave to the left and right of the radiobutton and text. Default is 1.
pady	How much space to leave above and below the radiobutton and text. Default is 1.
relief	Specifies the appearance of a decorative border around the label. The default is FLAT; for other values.
selectcolor	The color of the radiobutton when it is set. Default is red.
selectimage	If you are using the image option to display a graphic instead of text when the radiobutton is cleared, you can set the selectimage option to a different image that will be displayed when the radiobutton is set.
state	The default is state=NORMAL, but you can set state=DISABLED to gray out the control and make it unresponsive. If the cursor is currently over

	the radiobutton, the state is ACTIVE.
text	The label displayed next to the radiobutton. Use newlines ("\n") to display multiple lines of text.
textvariable	To slave the text displayed in a label widget to a control variable of class StringVar, set this option to that variable.
underline	You can display an underline (_) below the nth letter of the text, counting from 0, by setting this option to n. The default is underline=-1, which means no underlining.
value	When a radiobutton is turned on by the user, its control variable is set to its current value option. If the control variable is an IntVar, give each radiobutton in the group a different integer value option. If the control variable is a StringVar, give each radiobutton a different string value option.
variable	The control variable that this radiobutton shares with the other radiobuttons in the group. This can be either an IntVar or a StringVar.
width	Width of the label in characters (not pixels!). If this option is not set, the label will be sized to fit its contents.
wraplength	You can limit the number of characters in each line by setting this option to the desired number. The default value, 0, means that lines will be broken only at newlines.

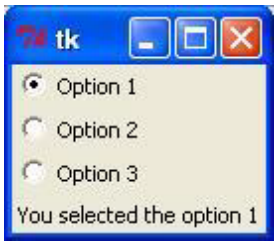
Methods Methods	**Description**
deselect()	Clears (turns off) the radiobutton.
flash()	Flashes the radiobutton a few times between its active and normal colors, but leaves it the way it started.
invoke()	You can call this method to get the same actions that would occur if the user clicked on the radiobutton to change its state.
select()	Sets (turns on) the radiobutton.

Example:
Try the following example yourself:

```
from tkinter import *
def sel():
      selection = "You selected the option " + str(var.get())
      label.config(text = selection)
root = Tk()
var = IntVar()
R1 = Radiobutton(root, text="Option 1", variable=var, value=1,
command=sel)
R1.pack( anchor = W )
R2 = Radiobutton(root, text="Option 2", variable=var, value=2,
command=sel)
R2.pack( anchor = W )
```

```
R3 = Radiobutton(root, text="Option 3", variable=var, value=3,
command=sel)
R3.pack( anchor = W)
label = Label(root)
label.pack()
root.mainloop()
```

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



12. Scale

The Scale widget provides a graphical slider object that allows you to select values from a specific scale.

Syntax

Here is the simple syntax to create this widget:

w = Scale (master, option, ...)

Parameters

- **master:** This represents the parent window.
- **options:** Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Option	Description
activebackground	The background color when the mouse is over the scale.
bg	The background color of the parts of the widget that are outside the trough.
bd	Width of the 3-d border around the trough and slider. Default is 2 pixels.
command	A procedure to be called every time the slider is moved. This procedure will be passed one argument, the new scale value. If the slider is moved rapidly, you may not get a callback for every possible position, but you'll certainly get a callback when it settles.
cursor	If you set this option to a cursor name (*arrow, dot etc.*), the mouse cursor will change to that pattern when it is over the scale.
digits	The way your program reads the current value shown in a scale widget is through a control variable. The control variable for a scale can be an IntVar, a DoubleVar (float), or a StringVar. If it is a string variable, the digits option controls how many digits to use when the numeric scale value is converted to a string.
font	The font used for the label and annotations.

fg	The color of the text used for the label and annotations.
from_	A float or integer value that defines one end of the scale's range.
highlightbackground	The color of the focus highlight when the scale does not have focus.
highlightcolor	The color of the focus highlight when the scale has the focus.
label	You can display a label within the scale widget by setting this option to the label's text. The label appears in the top left corner if the scale is horizontal, or the top right corner if vertical. The default is no label.
length	The length of the scale widget. This is the x dimension if the scale is horizontal, or the y dimension if vertical. The default is 100 pixels.
orient	Set orient=HORIZONTAL if you want the scale to run along the x dimension, or orient=VERTICAL to run parallel to the y-axis. Default is horizontal.
relief	Specifies the appearance of a decorative border around the label. The default is FLAT; for other values.
repeatdelay	This option controls how long button 1 has to be held down in the trough before the slider starts moving in that direction repeatedly. Default is repeatdelay=300, and the units are milliseconds.
resolution	Normally, the user will only be able to change the scale in whole units. Set this option to some other value to change the smallest increment of the scale's value. For example, if from_=-1.0 and to=1.0, and you set resolution=0.5, the scale will have 5 possible values: -1.0, -0.5, 0.0, +0.5, and +1.0.
showvalue	Normally, the current value of the scale is displayed in text form by the slider (above it for horizontal scales, to the left for vertical scales). Set this option to 0 to suppress that label.
sliderlength	Normally the slider is 30 pixels along the length of the scale. You can change that length by setting the sliderlength option to your desired length.
state	Normally, scale widgets respond to mouse events, and when they have the focus, also keyboard events. Set state=DISABLED to make the widget unresponsive.
takefocus	Normally, the focus will cycle through scale widgets. Set this option to 0 if you don't want this behavior.
tickinterval	To display periodic scale values, set this option to a number, and ticks will be displayed on multiples of that value. For example, if from_=0.0, to=1.0, and tickinterval=0.25, labels will be displayed along the scale at values 0.0, 0.25, 0.50, 0.75, and 1.00. These labels appear below the scale if horizontal, to its left if vertical. Default is 0, which suppresses display of ticks.
to	A float or integer value that defines one end of the scale's range; the other end is defined by the from_ option, discussed above. The to value can be either greater than or less than the from_ value. For vertical scales, the to value defines the bottom of the scale; for horizontal scales, the right end.

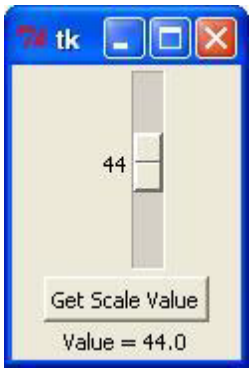
troughcolor	The color of the trough.
variable	The control variable for this scale, if any. Control variables may be from class IntVar, DoubleVar (float), or StringVar. In the latter case, the numerical value will be converted to a string.
width	The width of the trough part of the widget. This is the x dimension for vertical scales and the y dimension if the scale has orient=HORIZONTAL. Default is 15 pixels.

Methods : Scale objects have these methods

Methods	**Description**
get()	This method returns the current value of the scale.
set (value)	Sets the scale's value.

```
from tkinter import *
def sel():
      selection = "Value = " + str(var.get())
      label.config(text = selection)
root = Tk()
var = DoubleVar()
scale = Scale( root, variable = var )
scale.pack(anchor=CENTER)
button = Button(root, text="Get Scale Value", command=sel)
button.pack(anchor=CENTER)
label = Label(root)
label.pack()
root.mainloop()
```

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



13. Scrollbar

This widget provides a slide controller that is used to implement vertical scrolled widgets, such as Listbox, Text and Canvas. Note that you can also create horizontal scrollbars on Entry widgets.

Syntax:

Here is the simple syntax to create this widget:

w = Scrollbar (master, option, ...)

Parameters

 master: This represents the parent window.

 options: Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Option	Description
activebackground	The color of the slider and arrowheads when the mouse is over them.
bg	The color of the slider and arrowheads when the mouse is not over them.
bd	The width of the 3-d borders around the entire perimeter of the trough, and also the width of the 3-d effects on the arrowheads and slider. Default is no border around the trough, and a 2-pixel border around the arrowheads and slider.
command	A procedure to be called whenever the scrollbar is moved.
cursor	The cursor that appears when the mouse is over the scrollbar.
elementborderwidth	The width of the borders around the arrowheads and slider. The default is elementborderwidth=-1, which means to use the value of the borderwidth option.
highlightbackground	The color of the focus highlight when the scrollbar does not have focus.
highlightcolor	The color of the focus highlight when the scrollbar has the focus.
highlightthickness	The thickness of the focus highlight. Default is 1. Set to 0 to suppress display of the focus highlight.
jump	This option controls what happens when a user drags the slider. Normally (jump=0), every small drag of the slider causes the command callback to be called. If you set this option to 1, the callback isn't called until the user releases the mouse button.
orient	Set orient=HORIZONTAL for a horizontal scrollbar, orient=VERTICAL for a vertical one.
repeatdelay	This option controls how long button 1 has to be held down in the trough before the slider starts moving in that direction repeatedly. Default is repeatdelay=300, and the units are milliseconds.
repeatinterval	repeatinterval
takefocus	Normally, you can tab the focus through a scrollbar widget. Set takefocus=0 if you don't want this behavior.
troughcolor	The color of the trough.
width	Width of the scrollbar (its y dimension if horizontal, and its x dimension if vertical). Default is 16.

Methods Scrollbar objects have these methods:

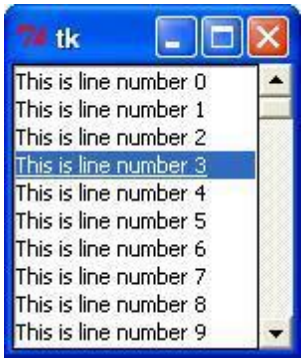
Methods	Description
get()	Returns two numbers (a, b) describing the current position of the slider. The a value gives the position of the left or top edge of the slider, for horizontal and vertical scrollbars respectively; the b value gives the position of the right or bottom edge.

set (first, last)	To connect a scrollbar to another widget w, set w's xscrollcommand or yscrollcommand to the scrollbar's set() method. The arguments have the same meaning as the values returned by the get() method.

Example
Try the following example yourself:

```
from tkinter import *
root = Tk()
scrollbar = Scrollbar(root)
scrollbar.pack( side = RIGHT, fill=Y )
mylist = Listbox(root, yscrollcommand = scrollbar.set )
for line in range(100):
      mylist.insert(END, "This is line number " + str(line))
mylist.pack( side = LEFT, fill = BOTH )
scrollbar.config( command = mylist.yview )
mainloop()
```

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



14. Text
Text widgets provide advanced capabilities that allow you to edit a multiline text and format the way it has to be displayed, such as changing its color and font. You can also use elegant structures like tabs and marks to locate specific sections of the text, and apply changes to those areas. Moreover, you can embed windows and images in the text because this widget was designed to handle both plain and formatted text.

Syntax
Here is the simple syntax to create this widget:
w = Text (master, option, ...)

Parameters
master: This represents the parent window.
options: Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Option	Description
bg	The default background color of the text widget.
bd	The width of the border around the text widget. Default is 2 pixels.
cursor	The cursor that will appear when the mouse is over the text widget.

exportselection	Normally, text selected within a text widget is exported to be the selection in the window manager. Set exportselection=0 if you don't want that behavior.
font	The default font for text inserted into the widget.
fg	The color used for text (and bitmaps) within the widget. You can change the color for tagged regions; this option is just the default.
height	The height of the widget in lines (not pixels!), measured according to the current font size.
highlightbackground	The color of the focus highlight when the text widget does not have focus.
highlightcolor	The color of the focus highlight when the text widget has the focus.
highlightthickness	The thickness of the focus highlight. Default is 1. Set highlightthickness=0 to suppress display of the focus highlight.
insertbackground	The color of the insertion cursor. Default is black.
insertborderwidth	Size of the 3-D border around the insertion cursor. Default is 0.
insertofftime	The number of milliseconds the insertion cursor is off during its blink cycle. Set this option to zero to suppress blinking. Default is 300.
insertontime	The number of milliseconds the insertion cursor is on during its blink cycle. Default is 600.
insertwidth	Width of the insertion cursor (its height is determined by the tallest item in its line). Default is 2 pixels.
padx	The size of the internal padding added to the left and right of the text area. Default is one pixel.
pady	The size of the internal padding added above and below the text area. Default is one pixel.
relief	The 3-D appearance of the text widget. Default is relief=SUNKEN.
selectbackground	The background color to use displaying selected text.
selectborderwidth	The width of the border to use around selected text.
spacing1	This option specifies how much extra vertical space is put above each line of text. If a line wraps, this space is added only before the first line it occupies on the display. Default is 0.
spacing2	This option specifies how much extra vertical space to add between displayed lines of text when a logical line wraps. Default is 0.
spacing3	This option specifies how much extra vertical space is added below each line of text. If a line wraps, this space is added only after the last line it occupies on the display. Default is 0.
state	Normally, text widgets respond to keyboard and mouse events; set state=NORMAL to get this behavior. If you set state=DISABLED, the text widget will not respond, and you won't be able to modify its contents programmatically either.
tabs	This option controls how tab characters position text.
width	The width of the widget in characters (not pixels!), measured according to the current font size.
wrap	This option controls the display of lines that are too wide. Set wrap=WORD and it will break the line after the last word that will fit. With the default behavior, wrap=CHAR, any line that gets too

	long will be broken at any character.
xscrollcommand	To make the text widget horizontally scrollable, set this option to the set() method of the horizontal scrollbar.
yscrollcommand	To make the text widget vertically scrollable, set this option to the set() method of the vertical scrollbar.

Methods Text objects have these methods:

Methods & Description
1. delete(startindex [,endindex]) This method deletes a specific character or a range of text.
2. get(startindex [,endindex]) This method returns a specific character or a range of text.
3. index(index) Returns the absolute value of an index based on the given index.
4. insert(index [,string]...) This method inserts strings at the specified index location.
5. See(index) This method returns true if the text located at the index position is visible.

Text widgets support three distinct helper structures: Marks, Tabs, and Indexes:
Marks are used to bookmark positions between two characters within a given text. We have the following methods available when handling marks:

Methods & Description
1. index(mark) Returns the line and column location of a specific mark.
2. mark_gravity(mark [,gravity]) Returns the gravity of the given mark. If the second argument is provided, the gravity is set for the given mark.
3. mark_names() Returns all marks from the Text widget.
4. mark_set(mark, index) Informs a new position to the given mark.
5. mark_unset(mark) Removes the given mark from the Text widget.

Tags are used to associate names to regions of text which makes easy the task of modifying the display settings of specific text areas. Tags are also used to bind event callbacks to specific ranges of text. Following are the available methods for handling tabs:

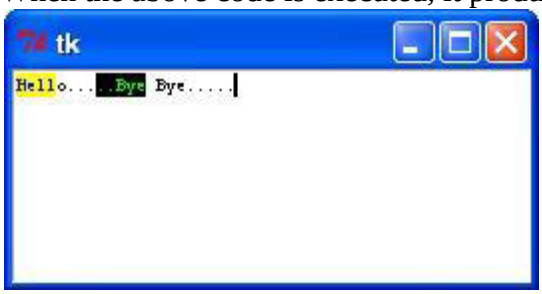
Methods and Description
1. tag_add(tagname, startindex[,endindex] ...) This method tags either the position defined by startindex, or a range delimited by the positions startindex and endindex.
2. tag_config You can use this method to configure the tag properties, which include, justify(center, left, or right), tabs(this property has the same functionality of the Text widget tabs's property), and underline(used to underline the tagged text).
3. tag_delete(tagname) This method is used to delete and remove a given tag.
4. tag_remove(tagname [,startindex[.endindex]] ...) After applying this method, the given tag is removed from the provided area without deleting the actual tag definition.

Example
Try the following example yourself:

```
from Tkinter import *
def onclick():
pass
root = Tk()
text = Text(root)
text.insert(INSERT, "Hello.....")
```

```
text.insert(END, "Bye Bye.....")
text.pack()
text.tag_add("here", "1.0", "1.4")
text.tag_add("start", "1.8", "1.13")
text.tag_config("here", background="yellow", foreground="blue")
text.tag_config("start", background="black", foreground="green")
root.mainloop()
```

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



15. TopLevel

Toplevel widgets work as windows that are directly managed by the window manager. They do not necessarily have a parent widget on top of them. Your application can use any number of top-level windows.

Syntax

Here is the simple syntax to create this widget:

w = Toplevel (option, ...)

Parameters:

 options: Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Option	Description
bg	The background color of the window.
bd	Border width in pixels; default is 0.
cursor	The cursor that appears when the mouse is in this window.
class_	Normally, text selected within a text widget is exported to be the selection in the window manager. Set exportselection=0 if you don't want that behavior.
font	The default font for text inserted into the widget.
fg	The color used for text (and bitmaps) within the widget. You can change the color for tagged regions; this option is just the default.
height	Window height.
relief	Normally, a top-level window will have no 3-d borders around it. To get a shaded border, set the bd option larger that its default value of zero, and set the relief option to one of the constants.
width	The desired width of the window.

Methods : Toplevel objects have these methods

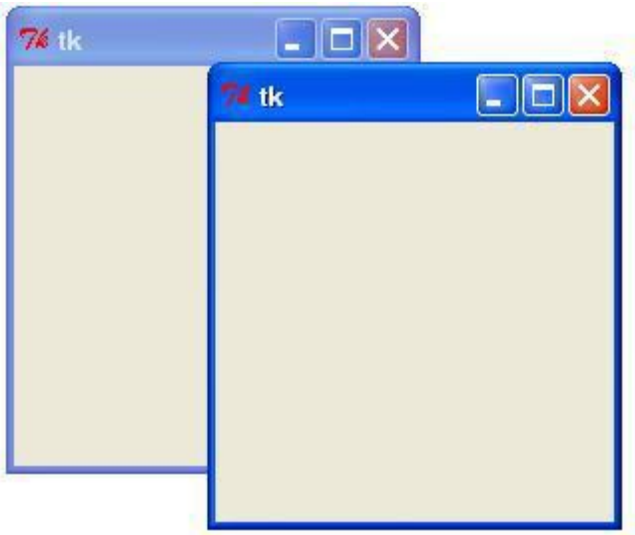
Methods and Description
deiconify() Displays the window, after using either the iconify or the withdraw

methods.
frame() Returns a system-specific window identifier.
group(window) Adds the window to the window group administered by the given window.
iconify() Turns the window into an icon, without destroying it.
protocol(name, function) Registers a function as a callback which will be called for the given protocol.
iconify() Turns the window into an icon, without destroying it.
state() Returns the current state of the window. Possible values are normal, iconic, withdrawn and icon.
transient([master]) Turns the window into a temporary(transient) window for the given master or to the window's parent, when no argument is given.
withdraw() Removes the window from the screen, without destroying it.
maxsize(width, height) Defines the maximum size for this window.
minsize(width, height) Defines the minimum size for this window.
positionfrom(who) Defines the position controller.
resizable(width, height) Defines the resize flags, which control whether the window can be resized.
sizefrom(who) Defines the size controller.
title(string) Defines the window title.

Example
Try following example yourself:

```
from tkinter import *
root = Tk()
top = Toplevel()
top.mainloop()
```

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



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.

Syntax

Here is the simple syntax to create this widget:

w = Spinbox(master, option, ...)

Parameters

- master: This represents the parent window.
- options: Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Option	Description
activebackground	The color of the slider and arrowheads when the mouse is over them.
bg	The color of the slider and arrowheads when the mouse is not over them.
bd	The width of the 3-d borders around the entire perimeter of the trough, and also the width of the 3-d effects on the arrowheads and slider. Default is no border around the trough, and a 2-pixel border around the arrowheads and slider.
command	A procedure to be called whenever the scrollbar is moved.
cursor	The cursor that appears when the mouse is over the scrollbar.
disabledbackground	The background color to use when the widget is disabled.
disabledforeground	The text color to use when the widget is disabled.
fg	Text color.
font	The font to use in this widget.
format	Format string. No default value.
from_	The minimum value. Used together with to to limit the spinbox range.
justify	Default is LEFT
relief	Default is SUNKEN.

repeatdelay	Together with repeatinterval, this option controls button auto-repeat. Both values are given in milliseconds.
repeatinterval	See repeatdelay.
state	One of NORMAL, DISABLED, or "readonly". Default is NORMAL.
textvariable	No default value.
to	See from.
validate	Validation mode. Default is NONE.
validatecommand	Validation callback. No default value.
values	A tuple containing valid values for this widget. Overrides from/to/increment.
vcmd	Same as validatecommand.
width	Widget width, in character units. Default is 20.
wrap	If true, the up and down buttons will wrap around.
xscrollcommand	Used to connect a spinbox field to a horizontal scrollbar. This option should be set to the set method of the corresponding scrollbar.

Methods : Spinbox objects have these methods:

Methods and Description
delete(startindex [,endindex]) This method deletes a specific character or a range of text.
get(startindex [,endindex]) This method returns a specific character or a range of text.
identify(x, y)
Identifies the widget element at the given location.
index(index) Returns the absolute value of an index based on the given index.
insert(index [,string]...) This method inserts strings at the specified index location.
invoke(element) Invokes a spinbox button.

Example

Try the following example yourself:

```
from tkinter import *
master = Tk()
w = Spinbox(master, from_=0, to=10)
w.pack()
mainloop()
```

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

17. PanelWindow

A PanedWindow is a container widget that may contain any number of panes, arranged horizontally or vertically. Each pane contains one widget and each pair of panes is separated by a moveable (via mouse movements) sash. Moving a sash causes the widgets on either side of the sash to be resized.

Syntax

Here is the simple syntax to create this widget:

w = PanedWindow(master, option, ...)

- **master:** This represents the parent window.
- **options**: Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Option	Description
bg	The color of the slider and arrowheads when the mouse is not over them.
bd	The width of the 3-d borders around the entire perimeter of the trough, and also the width of the 3-d effects on the arrowheads and slider. Default is no border around the trough, and a 2-pixel border around the arrowheads and slider.
borderwidth	Default is 2.
cursor	The cursor that appears when the mouse is over the window.
handlepad	Default is 8.
handlesize	Default is 8.
height	No default value.
orient	Default is HORIZONTAL.
relief	Default is FLAT.
sashcursor	No default value.
sashrelief	Default is RAISED.
sashwidth	Default is 2.
showhandle	No default value
width	No default value.

Methods : PanedWindow objects have these methods

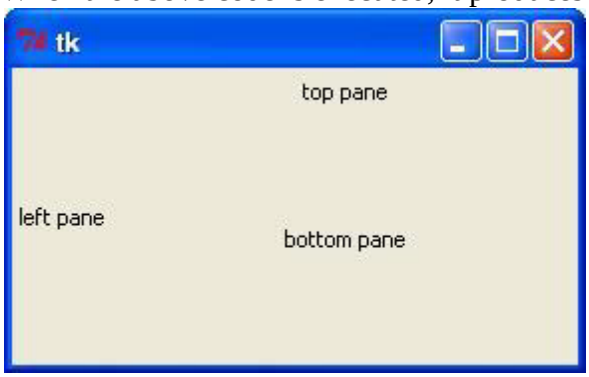
Methods and Description
add(child, options) Adds a child window to the paned window.
get(startindex [,endindex]) This method returns a specific character or a range of text.
config(options) Modifies one or more widget options. If no options are given, the method returns a dictionary containing all current option values.

Example

Try the following example yourself. Here's how to create a 3-pane widget:

```
from tkinter import *
m1 = PanedWindow()
m1.pack(fill=BOTH, expand=1)
left = Label(m1, text="left pane")
m1.add(left)
m2 = PanedWindow(m1, orient=VERTICAL)
m1.add(m2)
top = Label(m2, text="top pane")
m2.add(top)
bottom = Label(m2, text="bottom pane")
m2.add(bottom)
mainloop()
```

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



20. LabelFrame

A labelframe is a simple container widget. Its primary purpose is to act as a spacer or container for complex window layouts. This widget has the features of a frame plus the ability to display a label.

Syntax

Here is the simple syntax to create this widget:

w = LabelFrame(master, option, ...)

Parameters

- **master:** This represents the parent window**.**
- **options:** Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Option	**Description**
bg	The normal background color displayed behind the label and indicator.
bd	The size of the border around the indicator. Default is 2 pixels.
cursor	If you set this option to a cursor name (*arrow, dot etc.*), the mouse cursor will change to that pattern when it is over the checkbutton.
font	The vertical dimension of the new frame.
height	The vertical dimension of the new frame.
labelAnchor	Specifies where to place the label.
highlightbackground	Color of the focus highlight when the frame does not have focus.
highlightcolor	Color shown in the focus highlight when the frame has the focus.

highlightthickness	Thickness of the focus highlight.
relief	With the default value, relief=FLAT, the checkbutton does not stand out from its background. You may set this option to any of the other styles
text	Specifies a string to be displayed inside the widget.
width	Specifies the desired width for the window.

Example
Try the following example yourself. Here is how to create a labelframe widget:

```
from tkinter import *
root = Tk()
labelframe = LabelFrame(root, text="This is a LabelFrame")
labelframe.pack(fill="both", expand="yes")
left = Label(labelframe, text="Inside the LabelFrame")
left.pack()
root.mainloop()
```

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

23.messagebox
The **messagebox** module is used to display message boxes in your applications. This module provides a number of functions that you can use to display an appropriate message.
Some of these functions are showinfo, showwarning, showerror, askquestion, askokcancel, askyesno, and askretryignore.
Syntax:
Here is the simple syntax to create this widget:
messagebox.FunctionName(title, message [, options])
Parameters
FunctionName: This is the name of the appropriate message box function.
title: This is the text to be displayed in the title bar of a message box.
message: This is the text to be displayed as a message.
options: options are alternative choices that you may use to tailor a standard message box. Some of the options that you can use are default and parent. The default option is used to specify the default button, such as ABORT, RETRY, or IGNORE in the message box. The parent option is used to specify the window on top of which the message box is to be displayed.
You could use one of the following functions with dialogue box:

- **showinfo()**
- **showwarning()**
- **showerror ()**

- **askquestion()**
- **askokcancel()**
- **askyesno ()**
- **askretrycancel ()**

Try the following example yourself:

```
import tkinter
from tkinter import messagebox
top = tkinter.Tk()
def hello():
      messagebox.showinfo("Say Hello", "Hello World")

B1 = tkinter.Button(top, text = "Say Hello", command = hello)
B1.pack()
top.mainloop()
```

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

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

1. Dimensions
2. Colors
3. Fonts
4. Anchors
5. Relief styles
6. Bitmaps
7. Cursors

Let us study them briefly:

Dimensions

Various lengths, widths, and other dimensions of widgets can be described in many different units. If you set a dimension to an integer, it is assumed to be in pixels. You can specify units by setting a dimension to a string containing a number followed by**.**

Character	Description
c	Centimeters
i	Inches
m	Millimeters
p	Printer's points (about 1/72")

Length options: Tkinter expresses a length as an integer number of pixels. Here is the list of common length options: 1. 1. **borderwidth**: Width of the border which gives a three-dimensional look to the widget.
2. **highlightthickness**: Width of the highlight rectangle when the widget has focus.
3. **padX padY:** Extra space the widget requests from its layout manager beyond the minimum the widget needs to display its contents in the x and y directions.
4. **selectborderwidth**: Width of the three-dimentional border around selected items of the widget.
5. **wraplength**: Maximum line length for widgets that perform word wrapping.
6. **height**: Desired height of the widget; must be greater than or equal to 1.
7. **underline**: Index of the character to underline in the widget's text (0 is the first character, 1 the second one, and so on).
8. **width**: Desired width of the widget.

Colors

Tkinter represents colors with strings. There are two general ways to specify colors in Tkinter: We can use a string specifying the proportion of red, green and blue in hexadecimal digits. For example, "#fff" is white, "#000000" is black, "#000fff000" is pure green, and "#00ffff" is pure cyan (green plus blue). You can also use any locally defined standard color name. The colors "white", "black", "red", "green", "blue", "cyan", "yellow", and "magenta" will always be available.

Color options

The common color options are:
1. **activebackground**: Background color for the widget when the widget is active.
2. **activeforeground**: Foreground color for the widget when the widget is active.
3. **background**: Background color for the widget. This can also be represented as *bg*.
4. **disabledforeground**: Foreground color for the widget when the widget is disabled.
5. **foreground**: Foreground color for the widget. This can also be represented as *fg*.
6. **highlightbackground**: Background color of the highlight region when the widget has focus.
7. **highlightcolor**: Foreground color of the highlight region when the widget has focus.
8. **selectbackground**: Background color for the selected items of the widget.
9. **selectforeground**: Foreground color for the selected items of the widget.

Fonts : There may be up to three ways to specify type style.
Simple Tuple Fonts : As a tuple whose first element is the font family, followed by a size in points, optionally followed by a string containing one or more of the style modifiers bold, italic, underline and overstrike.
Example
("Helvetica", "16") for a 16-point Helvetica regular.
("Times", "24", "bold italic") for a 24-point Times bold italic**.**

Font object Fonts

You can create a "font object" by importing the tkFont module and using its Font class constructor:
import tkFont
font = tkFont.Font (option, ...)
Here is the list of options:
1. **family**: The font family name as a string.

2. **size**: The font height as an integer in points. To get a font n pixels high, use -n.
3. **weight**: "bold" for boldface, "normal" for regular weight.
4. **slant**: "italic" for italic, "roman" for unslanted.
5. underline: 1 for underlined text, 0 for normal.
6. **overstrike**: 1 for overstruck text, 0 for normal**.**

Example

helv36 = tkFont.Font(family="Helvetica",size=36,weight="bold")

X Window Fonts

If you are running under the X Window System, you can use any of the X font names.
For example, the font named "-*-lucidatypewriter-medium-r-*-*-*-140-*-*-*-*-*-*" is the author's favorite fixed-width font for onscreen use. Use the *xfontsel* program to help you select pleasing fonts.

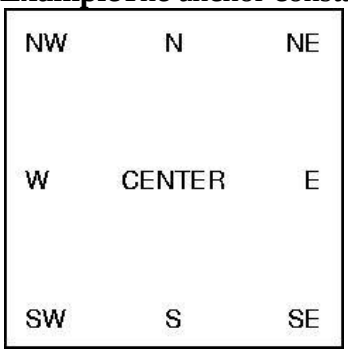
Anchors :

Anchors are used to define where text is positioned relative to a reference point. Here is list of possible constants, which can be used for Anchor attribute.

NW N NE W CENTER E SW S SE

For example, if you use CENTER as a text anchor, the text will be centered horizontally and vertically around the reference point. Anchor NW will position the text so that the reference point coincides with the northwest (top left) corner of the box containing the text. Anchor W will center the text vertically around the reference point, with the left edge of the text box passing through that point, and so on. If you create a small widget inside a large frame and use the anchor=SE option, the widget will be placed in the bottom right corner of the frame. If you used anchor=N instead, the widget would be centered along the top edge.

ExampleThe anchor constants are shown in this diagram:



9. Data compression

Since the time necessary to transmit data over the network is often more significant than the time your CPU spends preparing the data for transmission, it is often worthwhile to compress data before sending it. The popular HTTP protocol lets a client and server figure out whether they can both support compression. An interesting fact about the most ubiquitous form of compression, the GNU zlib facility (For applications that require data compression, the functions in this module allow compression and decompression, using the zlib library) that is available through the Python Standard Library, is that it is self-framing. If you start feeding it a compressed stream of data, then it can tell you when the compressed data has ended and further, uncompressed data has arrived past its end.

Most protocols choose to do their own framing and then, if desired, pass the resulting block to zlib for decompression. But you could conceivably promise yourself that you would always tack a bit of uncompressed data onto the end of each zlib compressed string—here, we will use a single '.' byte— and watch for your compression object to split out that "extra data" as the signal that you are done. Consider this combination of two compressed data streams:

```
>>> import zlib
>>> data = zlib.compress('sparse') + '.' + zlib.compress('flat') + '.'
>>> data
'x\x9c+.H,*N\x05\x00\t\r\x02\x8f.x\x9cK\xcbI,\x01\x00\x04\x16\x01\xa8.'
>>> len(data)
```

Imagine that these 28 bytes arrive at their destination in 8-byte packets. After processing the first packet, we will find the
decompression object's unused_data slot still empty, which tells us that there is still more data coming, so we would recv()
on our socket again:

```
>>> dobj = zlib.decompressobj()
>>> dobj.decompress(data[0:8]), dobj.unused_data
('spars', '')
```

But the second block of eight characters, when fed to our decompress object, both finishes out the compressed data we
were waiting for (since the final 'e' completes the string 'sparse') and also finally has a non-empty unused_data value that
shows us that we finally received our '.' byte:

```
>>> dobj.decompress(data[8:16]), dobj.unused_data
('e', '.x')
```

If another stream of compressed data is coming, then we have to provide everything past the '.'—in this case, the
character 'x'—to our new decompress object, then start feeding it the remaining "packets":

```
>>> dobj2 = zlib.decompressobj()
>>> dobj2.decompress('x'), dobj2.unused_data
('', '')
>>> dobj2.decompress(data[16:24]), dobj2.unused_data
('flat', '')
>>> dobj2.decompress(data[24:]), dobj2.unused_data
('', '.')
```

At this point, unused_data is again non-empty, meaning that we have read past the end of this second bout of compressed
data and can examine its content.

zlib

The zlib module supports data compression by providing access to the zlib library.

1. adler32(string [, value])

Computes the Adler-32 checksum of string. value is used as the starting value (which can be used to compute a checksum over the concatenation of several strings). Otherwise, a fixed default value is used.

2. compress(string [, level])

Compresses the data in string, where level is an integer from 1 to 9 controlling the level of compression. 1 is the least (fastest) compression, and 9 is the best (slowest) compression. The default value is 6. Returns a string containing the compressed data or raises error if an error occurs.

3. compressobj([level])
Returns a compression object. level has the same meaning as in the compress() function.
4. crc32(string [, value])
Computes a CRC checksum of string. If value is present, it's used as the starting value of the checksum. Otherwise, a fixed value is used.
5. decompress(string [, wbits [, buffsize]])
Decompresses the data in string. wbits controls the size of the window buffer, and buffsize is the initial size of the output buffer. Raises error if an error occurs.
6. decompressobj([wbits])
Returns a compression object.The wbits parameter controls the size of the window buffer.
A compression object, c, has the following methods:
7. c.compress(string)
Compresses string. Returns a string containing compressed data for at least part of the data in string.This data should be concatenated to the output produced by earlier calls to c.compress() to create the output stream. Some input data may be stored in internal buffers for later processing.
8. c.flush([mode])
Compresses all pending input and returns a string containing the remaining compressed output. mode is Z_SYNC_FLUSH, Z_FULL_FLUSH, or Z_FINISH (the default). Z_SYNC_FLUSH and Z_FULL_FLUSH allow further compression and are used to allow partial error recovery on decompression. Z_FINISH terminates the compression stream. A decompression object, d, has the following methods and attributes:
9. d.decompress(string [,max_length])
Decompresses string and returns a string containing uncompressed data for at least part of the data in string.This data should be concatenated with data produced by earlier calls to decompress() to form the output stream. Some input data may be stored in internal buffers for later processing. max_length specifies the maximum size of returned data. If exceeded, unprocessed data will be placed in the d.unconsumed_tail attribute.
10. d.flush()
All pending input is processed, and a string containing the remaining uncompressed output is returned.The decompression object cannot be used again after this call.
11. d.unconsumed_tail
String containing data not yet processed by the last decompress() call.This would contain data if decompression needs to be performed in stages due to buffer size limitations. In this case, this variable would be passed to subsequent decompress() calls.

10. Testing

10.1 Why Testing is required?

It's important to ask the question, "Why is testing a valuable use of my time?" It's a fair question, and it's the question those unfamiliar with testing code often ask. After all, testing takes time that could otherwise be spend writing code, and isn't that the most productive thing to be doing?
There are a number of valid answers to this question. I'll list a few here:
1. Testing makes sure your code works properly under a given set of conditions. Testing assures correctness under a basic set of conditions. Syntax errors will almost certainly be caught by running tests, and the basic logic of a unit of code can be tested to ensure correctness under certain conditions. Again, it's not about proving the code is correct under any set of conditions. We're simply aiming for a reasonably complete set of possible conditions.
2. Testing allows one to ensure that changes to the code did not break existing functionality. This is especially helpful when refactoring1 code. Without tests in place, you have no assurances that

your code changes did not break things that were previously working fine. If you want to be able to change or rewrite your code and know you didn't break anything, proper unit testing is imperative.
3. Testing forces one to think about the code under unusual conditions, possibly revealing logical errors. Writing tests forces you to think about the non-normal conditions your code may encounter. In the example above, my_addition_function adds two numbers. A simple test of basic correctness would call my_addition_function(2, 2) and assert that the result was 4. Further tests, however, might test that the function works correctly with floats by running my_addition_function(2.0, 2.0). Defensive coding principles suggest that your code should be able to gracefully fail on invalid input, so testing that an exception is properly raised when strings are passed as arguments to the function.
4. Good testing requires modular, decoupled code, which is a hallmark of good system design. The whole practice of unit testing is made much easier by code that is loosely coupled2. If your application code has direct database calls, for example, testing the logic of your application depends on having a valid database connection available and test data to be present in the database. Code that isolates external resources, on the other hand, can easily replace them during testing using mock objects. Applications designed with test-ability in mind usually end up being modular and loosely coupled out of necessity.

Three Approaches to Choosing Test Data

Testing appears to be a matter of choosing inputs that will show, to our satisfaction, that a program produces the correct outputs. How do we do this? There are three basic approaches to testing: haphazard, black box, and white box.

1. Haphazard Testing

The haphazard approach is the one you might be tempted to use late at night the day before a program is due. Just bang on the program with a few inputs until it breaks, fix the bugs that show up, and call it correct. Considering that the possible combinations of inputs for a complex program can run into the billions and beyond, randomly trying out a few of these is not going to be very effective.

2 Black-Box Testing

In black-box testing, we try to be more organized in your choice of inputs. Consider the simple example of a payroll program that computes pay differently for regular hours and overtime hours. Although there are many possible values for hours worked, you do not need to try all of them to feel confident that the program works correctly. When constructing the test data, you observe that all hours between 0 and 40 are in some sense equivalent, as are all hours over 40. So you might decide to test the program with just the inputs 30 and 50. Generally, inputs can be partitioned into clusters of equivalent data, such that if a program works correctly on one set of values from a cluster, it works equally well for allother values in the same cluster. Just to be on the safe side, you should also test the program for values on the boundaries between clusters.
For the payroll problem, this means adding the values 0 and 40 to our test data. Finally, we should consider data that you know are unreasonable. For the payroll problem, we could include –15, 3045, and “3ax6” in the test data. There are difficulties with black-box testing. It is easy to overlook some clusters, and worse, the number of clusters can be so large that we cannot possibly consider them all. It is important to note that the construction of the test data is made without consideration or knowledge of the program’s internal workings.

3. White-Box Testing

In white-box testing, you attempt to concoct test data that exercise all parts of our program. To do so, you examine the code closely and then formulate the test data, but the task can be difficult. Imagine a program consisting of a dozen if-else statements following each other in sequence.

When testing this program, prudence recommends using test data that cause each branch of each if-else statement to be executed at least once. This is called code coverage and perhaps could be achieved with as few as two sets of inputs. The first set might exercise all of the if clauses, while the second set exercises all the else clauses. However, such an approach is woefully inadequate. What you really need are test data that exercise every possible combination of if clauses and else clauses, of which there are 212, or 4096. That is, the test data should exercise every possible path through the program. Keep in mind that this is just a simple example. A typical program might contain an enormous number of paths. Unfortunately, the fact that every path through a program has been tested tells you nothing about whether or not the program's logic takes into account all the different combinations of inputs. Thus, you should combine black-box testing with white-box testing.

10.2 Unit testing

Unit testing, specifically tests a single "unit" of code in isolation. A unit could be an entire module, a single class or function, or almost anything in between. What's important, however, is that the code is isolated from other code we're not testing (which itself could have errors and would thus confuse test results). Consider the following example:

```
def is_prime(number):
    """Return True if *number* is prime."""
    for element in range(number):
        if number % element == 0:
            return False
    return True

def print_next_prime(number):
    """Print the closest prime number larger than *number*."""
    index = number
    while True:
        index += 1
        if is_prime(index):
            print(index)
```

We have two functions, is_prime and print_next_prime. If we wanted to test print_next_prime, we would need to be sure that is_prime is correct, as print_next_prime makes use of it. In this case, the function print_next_prime is one unit, and is_prime is another. Since unit tests test only a single unit at a time, we would need to think carefully about how we could accurately test print_next_prime .So what does test code look like? If the previous example is stored in a file named primes.py, we may write test code in a file named test_primes.py. Here are the minimal contents of test_primes.py, with an example test:

```
import unittest
from primes import is_prime
class PrimesTestCase(unittest.TestCase):
    """Tests for `primes.py`."""

    def test_is_five_prime(self):
        """Is five successfully determined to be prime?"""
        self.assertTrue(is_prime(5))

if __name__ == '__main__':
```

```
    unittest.main()
```

The file creates a unit test with a single test case: test_is_five_prime. Using Python's built-in unittest framework, any member function whose name begins with test in a class deriving from unittest.TestCase will be run, and its assertions checked, when unittest.main() is called. If we "run the tests" by running python test_primes.py, we'll see the output of the unittest framework printed on the console:

```
$ python test_primes.py
E
======================================================================
ERROR: test_is_five_prime (__main__.PrimesTestCase)
----------------------------------------------------------------------
Traceback (most recent call last):
File "test_primes.py", line 8, in test_is_five_prime
    self.assertTrue(is_prime(5))
File "/home/jknupp/code/github_code/blug_private/primes.py", line 4, in is_prime
    if number % element == 0:
ZeroDivisionError: integer division or modulo by zero
----------------------------------------------------------------------
Ran 1 test in 0.000s
```

The single "E" represents the results of our single test (if it was successful, a "." would have been printed). We can see that our test failed, the line that caused the failure, and any exceptions raised.

The "unittest" unit testing framework was originally inspired by JUnit and has a similar flavor as major unit testing frameworks in other languages. It supports test automation, sharing of setup and shutdown code for tests, aggregation of tests into collections, and independence of the tests from the reporting framework. To achieve this, unittest supports some important concepts in an object-oriented way:

1. test fixture: A *test fixture* represents the preparation needed to perform one or more tests, and any associate cleanup actions. This may involve, for example, creating temporary or proxy databases, directories, or starting a server process.

2. test case: A *test case* is the individual unit of testing. It checks for a specific response to a particular set of inputs. unittest provides a base class,TestCase, which may be used to create new test cases.

3. test suite: A *test suite* is a collection of test cases, test suites, or both. It is used to aggregate tests that should be executed together.

4. test runner: A *test runner* is a component which orchestrates the execution of tests and provides the outcome to the user. The runner may use a graphical interface, a textual interface, or return a special value to indicate the results of executing the tests.

Basic example

The unittest module provides a rich set of tools for constructing and running tests. This section demonstrates that a small subset of the tools suffice to meet the needs of most users. Here is a short script to test three string methods:

```
import unittest
class TestStringMethods(unittest.TestCase):
    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')
    def test_isupper(self):
        self.assertTrue('FOO'.isupper())
```

```
        self.assertFalse('Foo'.isupper())
    def test_split(self):
        s = 'hello world'
        self.assertEqual(s.split(), ['hello', 'world'])
        # check that s.split fails when the separator is not a string
        with self.assertRaises(TypeError):
            s.split(2)
if __name__ == '__main__':
    unittest.main()
```

A testcase is created by subclassing unittest.TestCase. The three individual tests are defined with methods whose names start with the letterstest. This naming convention informs the test runner about which methods represent tests.
The final block shows a simple way to run the tests. unittest.main() provides a command-line interface to the test script. When run from the command line, the above script produces an output that looks like this:

```
...
----------------------------------------------------------------------
Ran 3 tests in 0.000s
OK
```

Passing the -v option to your test script will instruct unittest.main() to enable a higher level of verbosity, and produce the following output:

```
test_isupper (__main__.TestStringMethods) ... ok
test_split (__main__.TestStringMethods) ... ok
test_upper (__main__.TestStringMethods) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.001s
OK
```

The above examples show the most commonly used unittest features which are sufficient to meet many everyday testing needs.

Organizing test code

The basic building blocks of unit testing are *test cases* — single scenarios that must be set up and checked for correctness. In unittest, test cases are represented by unittest.TestCase instances. To make your own test cases you must write subclasses of TestCase or use FunctionTestCase. The testing code of a TestCase instance should be entirely self contained, such that it can be run either in isolation or in arbitrary combination with any number of other test cases.The simplest TestCase subclass will simply implement a test method (i.e. a method whose name starts with test) in order to perform specific testing code:

```
import unittest

class DefaultWidgetSizeTestCase(unittest.TestCase):
    def test_default_widget_size(self):
        widget = Widget('The widget')
        self.assertEqual(widget.size(), (50, 50))
```

Note that in order to test something, we use one of the assert*() methods provided by the TestCase base class. If the test fails, an exception will be raised, and unittest will identify the

test case as a *failure*. Any other exceptions will be treated as *errors*. Tests can be numerous, and their set-up can be repetitive. Luckily, we can factor out set-up code by implementing a method called setUp(), which the testing framework will automatically call for every single test we run:

```
import unittest

class WidgetTestCase(unittest.TestCase):
    def setUp(self):
        self.widget = Widget('The widget')

    def test_default_widget_size(self):
        self.assertEqual(self.widget.size(), (50,50),  'incorrect default size')

    def test_widget_resize(self):
        self.widget.resize(100,150)
        self.assertEqual(self.widget.size(), (100,150), 'wrong size after resize')
```

Note: The order in which the various tests will be run is determined by sorting the test method names with respect to the built-in ordering for strings. If the setUp() method raises an exception while the test is running, the framework will consider the test to have suffered an error, and the test method will not be executed. Similarly, we can provide a tearDown() method that tidies up after the test method has been run:

```
import unittest

class WidgetTestCase(unittest.TestCase):
    def setUp(self):
        self.widget = Widget('The widget')

    def tearDown(self):
        self.widget.dispose()
```

If setUp() succeeded, tearDown() will be run whether the test method succeeded or not. Such a working environment for the testing code is called a *fixture*. Test case instances are grouped together according to the features they test. unittest provides a mechanism for this: the *test suite*, represented by unittest's TestSuite class. In most cases, calling unittest.main() will do the right thing and collect all the module's test cases for you, and then execute them.

However, should you want to customize the building of your test suite, you can do it yourself:

```
def suite():
    suite = unittest.TestSuite()
    suite.addTest(WidgetTestCase('test_default_size'))
    suite.addTest(WidgetTestCase('test_resize'))
    return suite
```

You can place the definitions of test cases and test suites in the same modules as the code they are to test (such as widget.py), but there are several advantages to placing the test code in a separate module, such as test_widget.py:

- The test module can be run standalone from the command line.
- The test code can more easily be separated from shipped code.
- There is less temptation to change test code to fit the code it tests without a good reason.
- Test code should be modified much less frequently than the code it tests.

- Tested code can be refactored more easily.
- Tests for modules written in C must be in separate modules anyway, so why not be consistent?
- If the testing strategy changes, there is no need to change the source code.

Re-using old test code:

Some users will find that they have existing test code that they would like to run from unittest, without converting every old test function to a TestCase subclass. For this reason, unittest provides a FunctionTestCase class. This subclass of TestCase can be used to wrap an existing test function. Set-up and tear-down functions can also be provided.

Given the following test function:

```
def testSomething():
    something = makeSomething()
    assert something.name is not None
    # ...
```

one can create an equivalent test case instance as follows, with optional set-up and tear-down methods:

```
testcase = unittest.FunctionTestCase(testSomething,
                                     setUp=makeSomethingDB,
                                     tearDown=deleteSomethingDB)
```

Note: Even though FunctionTestCase can be used to quickly convert an existing test base over to a unittest-based system, this approach is not recommended. Taking the time to set up proper TestCase subclasses will make future test refactorings infinitely easier.

In some cases, the existing tests may have been written using the doctest module. If so, doctest provides a DocTestSuite class that can automatically build unittest.TestSuite instances from the existing doctest-based tests.

Test cases

Instances of the TestCase class represent the logical test units in the unittest universe. This class is intended to be used as a base class, with specific tests being implemented by concrete subclasses. This class implements the interface needed by the test runner to allow it to drive the tests, and methods that the test code can use to check for and report various kinds of failure.

class unittest.**TestCase**(*methodName='runTest'*)

Each instance of TestCase will run a single base method: the method named *methodName*. In most uses of TestCase, you will neither change the *methodName* nor reimplement the default runTest() method.

Changed in version 3.2: TestCase can be instantiated successfully without providing a *methodName*. This makes it easier to experiment with TestCase from the interactive interpreter.

TestCase instances provide three groups of methods: one group used to run the test, another used by the test implementation to check conditions and report failures, and some inquiry methods allowing information about the test itself to be gathered.

Methods in the first group (running the test) are:

1. setUp(): Method called to prepare the test fixture. This is called immediately before calling the test method; other than AssertionError or SkipTest, any exception raised by this method will be considered an error rather than a test failure. The default implementation does nothing.

2. tearDown() :Method called immediately after the test method has been called and the result recorded. This is called even if the test method raised an exception, so the implementation in subclasses may need to be particularly careful about checking internal state. Any exception, other thanAssertionError or SkipTest, raised by this method will be considered an additional error rather

than a test failure (thus increasing the total number of reported errors). This method will only be called if the setUp() succeeds, regardless of the outcome of the test method. The default implementation does nothing.

3. setUpClass() : A class method called before tests in an individual class run. setUpClass is called with the class as the only argument and must be decorated as a classmethod():

```
@classmethod
def setUpClass(cls):
    ...
```

4. tearDownClass() : A class method called after tests in an individual class have run. tearDownClass is called with the class as the only argument and must be decorated as a classmethod():

```
@classmethod
def tearDownClass(cls):
    ...
```

5. run(*result=None*) : Run the test, collecting the result into the TestResult object passed as *result*. If *result* is omitted or None, a temporary result object is created (by calling the defaultTestResult() method) and used. The result object is returned to run()'s caller. The same effect may be had by simply calling the TestCase instance.

6. skipTest(*reason*)

Calling this during a test method or setUp() skips the current test. See Skipping tests and expected failures for more information.

7. subTest(*msg=None*, ***params*) : Return a context manager which executes the enclosed code block as a subtest. *msg* and *params* are optional, arbitrary values which are displayed whenever a subtest fails, allowing you to identify them clearly. A test case can contain any number of subtest declarations, and they can be arbitrarily nested.

8. **debug**() : Run the test without collecting the result. This allows exceptions raised by the test to be propagated to the caller, and can be used to support running tests under a debugger. The TestCase class provides several assert methods to check for and report failures. The following table lists the most commonly used methods:

Method	**Checks that**	**New in**
assertEqual(a, b)	a == b	
assertNotEqual(a, b)	a != b	
assertTrue(x)	bool(x) is True	
assertFalse(x)	bool(x) is False	
assertIs(a, b)	a is b	3.1
assertIsNot(a, b)	a is not b	3.1
assertIsNone(x)	x is None	3.1
assertIsNotNone(x)	x is not None	3.1
assertIn(a, b)	a in b	3.1
assertNotIn(a, b)	a not in b	3.1

Method	**Checks that**	**New in**
assertIsInstance(a, b)	isinstance(a, b)	3.2
assertNotIsInstance(a, b)	not isinstance(a, b)	3.2