Tkinter 8.5 reference: a GUI for Python

John W. Shipman

2013-12-31 17:59

Abstract

Describes the *Tkinter* widget set for constructing graphical user interfaces (GUIs) in the Python programming language. Includes coverage of the *ttk* themed widgets.

This publication is available in Web form[1] and also as a PDF document[2]. Please forward any comments to **tcc-doc@nmt.edu**.

Table of Contents

1. A cross-platform graphical user interface builder for Python 3
2. A minimal application 4
3. Definitions 5
4. Layout management 5
    4.1. The `.grid()` method 6
    4.2. Other grid management methods 7
    4.3. Configuring column and row sizes 7
    4.4. Making the root window resizeable 8
5. Standard attributes 9
    5.1. Dimensions 9
    5.2. The coordinate system 10
    5.3. Colors 10
    5.4. Type fonts 10
    5.5. Anchors 12
    5.6. Relief styles 12
    5.7. Bitmaps 12
    5.8. Cursors 13
    5.9. Images 14
    5.10. Geometry strings 15
    5.11. Window names 16
    5.12. Cap and join styles 16
    5.13. Dash patterns 17
    5.14. Matching stipple patterns 17
6. Exception handling 18
7. The `Button` widget 18
8. The `Canvas` widget 20
    8.1. `Canvas` coordinates 22
    8.2. The `Canvas` display list 22
    8.3. `Canvas` object IDs 22

[1] http://www.nmt.edu/tcc/help/pubs/tkinter/
[2] http://www.nmt.edu/tcc/help/pubs/tkinter/tkinter.pdf

8.4. `Canvas` tags 22
8.5. `Canvas` *`tagOrId`* arguments 22
8.6. Methods on `Canvas` widgets 22
8.7. `Canvas` arc objects 28
8.8. `Canvas` bitmap objects 29
8.9. `Canvas` image objects 30
8.10. `Canvas` line objects 30
8.11. `Canvas` oval objects 32
8.12. `Canvas` polygon objects 33
8.13. `Canvas` rectangle objects 35
8.14. `Canvas` text objects 37
8.15. `Canvas` window objects 38
9. The `Checkbutton` widget 38
10. The `Entry` widget 41
10.1. Scrolling an `Entry` widget 45
10.2. Adding validation to an `Entry` widget 45
11. The `Frame` widget 47
12. The `Label` widget 48
13. The `LabelFrame` widget 50
14. The `Listbox` widget 52
14.1. Scrolling a `Listbox` widget 56
15. The `Menu` widget 56
15.1. `Menu` item creation (`coption`) options 59
15.2. Top-level menus 60
16. The `Menubutton` widget 61
17. The `Message` widget 63
18. The `OptionMenu` widget 64
19. The `PanedWindow` widget 65
19.1. `PanedWindow` child configuration options 67
20. The `Radiobutton` widget 68
21. The `Scale` widget 71
22. The `Scrollbar` widget 74
22.1. The `Scrollbar` *`command`* callback 77
22.2. Connecting a `Scrollbar` to another widget 77
23. The `Spinbox` widget 78
24. The `Text` widget 82
24.1. `Text` widget indices 84
24.2. `Text` widget marks 86
24.3. `Text` widget images 86
24.4. `Text` widget windows 87
24.5. `Text` widget tags 87
24.6. Setting tabs in a `Text` widget 87
24.7. The `Text` widget undo/redo stack 88
24.8. Methods on `Text` widgets 88
25. `Toplevel`: Top-level window methods 95
26. Universal widget methods 97
27. Standardizing appearance 105
27.1. How to name a widget class 106
27.2. How to name a widget instance 107
27.3. Resource specification lines 107
27.4. Rules for resource matching 108
28. *ttk*: Themed widgets 108

28.1. Importing *ttk* 109
28.2. The *ttk* widget set 110
29. *ttk*.`Button` 110
30. *ttk*.`Checkbutton` 112
31. *ttk*.`Combobox` 115
32. *ttk*.`Entry` 116
33. *ttk*.`Frame` 118
34. *ttk*.`Label` 119
35. *ttk*.`LabelFrame` 122
36. *ttk*.`Menubutton` 124
37. *ttk*.`Notebook` 126
37.1. Virtual events for the *ttk*.`Notebook` widget 128
38. *ttk*.`PanedWindow` 129
39. *ttk*.`Progressbar` 130
40. *ttk*.`Radiobutton` 131
41. *ttk*.`Scale` 133
42. *ttk*.`Scrollbar` 135
43. *ttk*.`Separator` 137
44. *ttk*.`Sizegrip` 137
45. *ttk*.`Treeview` 137
45.1. Virtual events for the *ttk*.`Treeview` widget 145
46. Methods common to all *ttk* widgets 145
46.1. Specifying widget states in *ttk* 146
47. Customizing and creating *ttk* themes and styles 146
48. Finding and using *ttk* themes 147
49. Using and customizing *ttk* styles 147
50. The *ttk* element layer 149
50.1. *ttk* layouts: Structuring a style 149
50.2. *ttk* style maps: dynamic appearance changes 151
51. Connecting your application logic to the widgets 153
52. Control variables: the values behind the widgets 153
53. Focus: routing keyboard input 155
53.1. Focus in *ttk* widgets 156
54. Events 157
54.1. Levels of binding 157
54.2. Event sequences 158
54.3. Event types 158
54.4. Event modifiers 160
54.5. Key names 160
54.6. Writing your handler: The `Event` class 162
54.7. The extra arguments trick 164
54.8. Virtual events 165
55. Pop-up dialogs 165
55.1. The `tkMessageBox` dialogs module 165
55.2. The `tkFileDialog` module 167
55.3. The `tkColorChooser` module 168

1. A cross-platform graphical user interface builder for Python

Tkinter is a GUI (graphical user interface) widget set for Python. This document was written for Python 2.7 and *Tkinter* 8.5 running in the X Window system under Linux. Your version may vary.

Pertinent references:

- Fredrik Lundh, who wrote *Tkinter*, has two versions of his *An Introduction to Tkinter*: a more complete 1999 version[3] and a 2005 version[4] that presents a few newer features.
- *Python 2.7 quick reference*[5]: general information about the Python language.
- For an example of a sizeable working application (around 1000 lines of code), see *huey: A color and font selection tool*[6]. The design of this application demonstrates how to build your own compound widgets.

We'll start by looking at the visible part of *Tkinter*: creating the widgets and arranging them on the screen. Later we will talk about how to connect the face—the "front panel"—of the application to the logic behind it.

2. A minimal application

Here is a trivial *Tkinter* program containing only a Quit button:

```
#!/usr/bin/env python                    1
import Tkinter as tk                     2

class Application(tk.Frame):                          3
    def __init__(self, master=None):
        tk.Frame.__init__(self, master)               4
        self.grid()                                   5
        self.createWidgets()

    def createWidgets(self):
        self.quitButton = tk.Button(self, text='Quit',
            command=self.quit)                        6
        self.quitButton.grid()                        7

app = Application()                                   8
app.master.title('Sample application')                9
app.mainloop()                                        10
```

1 This line makes the script self-executing, assuming that your system has Python correctly installed.
2 This line imports the *Tkinter* module into your program's namespace, but renames it as `tk`.
3 Your application class must inherit from *Tkinter*'s `Frame` class.
4 Calls the constructor for the parent class, `Frame`.
5 Necessary to make the application actually appear on the screen.
6 Creates a button labeled "Quit".
7 Places the button on the application.
8 The main program starts here by instantiating the `Application` class.
9 This method call sets the title of the window to "Sample application".
10 Starts the application's main loop, waiting for mouse and keyboard events.

[3] http://www.pythonware.com/library/tkinter/introduction/
[4] http://effbot.org/tkinterbook/
[5] http://www.nmt.edu/tcc/help/pubs/python/
[6] http://www.nmt.edu/tcc/help/lang/python/examples/huey/

3. Definitions

Before we proceed, let's define some of the common terms.

window
: This term has different meanings in different contexts, but in general it refers to a rectangular area somewhere on your display screen.

top-level window
: A window that exists independently on your screen. It will be decorated with the standard frame and controls for your system's desktop manager. You can move it around on your desktop. You can generally resize it, although your application can prevent this

widget
: The generic term for any of the building blocks that make up an application in a graphical user interface. Examples of widgets: buttons, radiobuttons, text fields, frames, and text labels.

frame
: In *Tkinter*, the `Frame` widget is the basic unit of organization for complex layouts. A frame is a rectangular area that can contain other widgets.

child, parent
: When any widget is created, a *parent-child* relationship is created. For example, if you place a text label inside a frame, the frame is the parent of the label.

4. Layout management

Later we will discuss the widgets, the building blocks of your GUI application. How do widgets get arranged in a window?

Although there are three different "geometry managers" in *Tkinter*, the author strongly prefers the `.grid()` geometry manager for pretty much everything. This manager treats every window or frame as a table—a gridwork of rows and columns.

- A *cell* is the area at the intersection of one row and one column.
- The width of each column is the width of the widest cell in that column.
- The height of each row is the height of the largest cell in that row.
- For widgets that do not fill the entire cell, you can specify what happens to the extra space. You can either leave the extra space outside the widget, or stretch the widget to fit it, in either the horizontal or vertical dimension.
- You can combine multiple cells into one larger area, a process called *spanning*.

When you create a widget, it does not appear until you register it with a geometry manager. Hence, construction and placing of a widget is a two-step process that goes something like this:

```
    self.thing = tk.Constructor(parent, ...)
    self.thing.grid(...)
```

where *`Constructor`* is one of the widget classes like `Button`, `Frame`, and so on, and `parent` is the parent widget in which this child widget is being constructed. All widgets have a `.grid()` method that you can use to tell the geometry manager where to put it.

4.1. The `.grid()` method

To display a widget *w* on your application screen:

```
    w.grid(option=value, ...)
```

This method registers a widget *w* with the grid geometry manager—if you don't do this, the widget will exist internally, but it will not be visible on the screen. For the options, see Table 1, "Arguments of the `.grid()` geometry manager" (p. 6).

Table 1. Arguments of the `.grid()` geometry manager

`column`	The column number where you want the widget gridded, counting from zero. The default value is zero.
`columnspan`	Normally a widget occupies only one cell in the grid. However, you can grab multiple cells of a row and merge them into one larger cell by setting the `columnspan` option to the number of cells. For example, `w.grid(row=0, column=2, columnspan=3)` would place widget *w* in a cell that spans columns 2, 3, and 4 of row 0.
`in_`	To register *w* as a child of some widget w_2, use `in_=`w_2. The new parent w_2 must be a descendant of the *parent* widget used when *w* was created.
`ipadx`	Internal x padding. This dimension is added inside the widget inside its left and right sides.
`ipady`	Internal y padding. This dimension is added inside the widget inside its top and bottom borders.
`padx`	External x padding. This dimension is added to the left and right outside the widget.
`pady`	External y padding. This dimension is added above and below the widget.
`row`	The row number into which you want to insert the widget, counting from 0. The default is the next higher-numbered unoccupied row.
`rowspan`	Normally a widget occupies only one cell in the grid. You can grab multiple adjacent cells of a column, however, by setting the `rowspan` option to the number of cells to grab. This option can be used in combination with the `columnspan` option to grab a block of cells. For example, `w.grid(row=3, column=2, rowspan=4, columnspan=5)` would place widget *w* in an area formed by merging 20 cells, with row numbers 3–6 and column numbers 2–6.
`sticky`	This option determines how to distribute any extra space within the cell that is not taken up by the widget at its natural size. See below.

- If you do not provide a `sticky` attribute, the default behavior is to center the widget in the cell.
- You can position the widget in a corner of the cell by using `sticky=tk.NE` (top right), `tk.SE` (bottom right), `tk.SW` (bottom left), or `tk.NW` (top left).
- You can position the widget centered against one side of the cell by using `sticky=tk.N` (top center), `tk.E` (right center), `tk.S` (bottom center), or `tk.W` (left center).
- Use `sticky=tk.N+tk.S` to stretch the widget vertically but leave it centered horizontally.
- Use `sticky=tk.E+tk.W` to stretch it horizontally but leave it centered vertically.
- Use `sticky=tk.N+tk.E+tk.S+tk.W` to stretch the widget both horizontally and vertically to fill the cell.

- The other combinations will also work. For example, `sticky=tk.N+tk.S+tk.W` will stretch the widget vertically and place it against the west (left) wall.

4.2. Other grid management methods

These grid-related methods are defined on all widgets:

`w.grid_bbox(column=None, row=None, col2=None, row2=None)`
Returns a 4-tuple describing the bounding box of some or all of the grid system in widget `w`. The first two numbers returned are the `x` and `y` coordinates of the upper left corner of the area, and the second two numbers are the width and height.

If you pass in `column` and `row` arguments, the returned bounding box describes the area of the cell at that column and row. If you also pass in `col2` and `row2` arguments, the returned bounding box describes the area of the grid from columns `column` to `col2` inclusive, and from rows `row` to `row2` inclusive.

For example, `w.grid_bbox(0, 0, 1, 1)` returns the bounding box of four cells, not one.

`w.grid_forget()`
This method makes widget `w` disappear from the screen. It still exists, it just isn't visible. You can use `.grid()` it to make it appear again, but it won't remember its grid options.

`w.grid_info()`
Returns a dictionary whose keys are `w`'s option names, with the corresponding values of those options.

`w.grid_location(x, y)`
Given a coordinates `(x, y)` relative to the containing widget, this method returns a tuple `(col, row)` describing what cell of `w`'s grid system contains that screen coordinate.

`w.grid_propagate()`
Normally, all widgets *propagate* their dimensions, meaning that they adjust to fit the contents. However, sometimes you want to force a widget to be a certain size, regardless of the size of its contents. To do this, call `w.grid_propagate(0)` where `w` is the widget whose size you want to force.

`w.grid_remove()`
This method is like `.grid_forget()`, but its grid options are remembered, so if you `.grid()` it again, it will use the same grid configuration options.

`w.grid_size()`
Returns a 2-tuple containing the number of columns and the number of rows, respectively, in `w`'s grid system.

`w.grid_slaves(row=None, column=None)`
Returns a list of the widgets managed by widget `w`. If no arguments are provided, you will get a list of all the managed widgets. Use the `row=` argument to select only the widgets in one row, or the `column=` argument to select only the widgets in one column.

4.3. Configuring column and row sizes

Unless you take certain measures, the width of a grid column inside a given widget will be equal to the width of its widest cell, and the height of a grid row will be the height of its tallest cell. The `sticky` attribute on a widget controls only where it will be placed if it doesn't completely fill the cell.

If you want to override this automatic sizing of columns and rows, use these methods on the *parent* widget `w` that contains the grid layout:

`w.columnconfigure(N, option=value, ...)`
In the grid layout inside widget `w`, configure column `N` so that the given `option` has the given `value`. For options, see the table below.

`w.rowconfigure(N, option=value, ...)`
In the grid layout inside widget `w`, configure row `N` so that the given `option` has the given `value`. For options, see the table below.

Here are the options used for configuring column and row sizes.

Table 2. Column and row configuration options for the `.grid()` geometry manager

`minsize`	The column or row's minimum size in pixels. If there is nothing in the given column or row, it will not appear, even if you use this option.
`pad`	A number of pixels that will be added to the given column or row, over and above the largest cell in the column or row.

`weight`	To make a column or row stretchable, use this option and supply a value that gives the relative weight of this column or row when distributing the extra space. For example, if a widget `w` contains a grid layout, these lines will distribute three-fourths of the extra space to the first column and one-fourth to the second column:
	`w.columnconfigure(0, weight=3)`
	`w.columnconfigure(1, weight=1)`
	If this option is not used, the column or row will not stretch.

4.4. Making the root window resizeable

Do you want to let the user resize your entire application window, and distribute the extra space among its internal widgets? This requires some operations that are not obvious.

It's necessary to use the techniques for row and column size management, described in Section 4.3, “Configuring column and row sizes” (p. 7), to make your `Application` widget's grid stretchable. However, that alone is not sufficient.

Consider the trivial application discussed in Section 2, “A minimal application” (p. 4), which contains only a *Quit* button. If you run this application, and resize the window, the button stays the same size, centered within the window.

Here is a replacement version of the `.__createWidgets()` method in the minimal application. In this version, the *Quit* button always fills all the available space.

```
    def createWidgets(self):
        top=self.winfo_toplevel()                    1
        top.rowconfigure(0, weight=1)                2
        top.columnconfigure(0, weight=1)             3
        self.rowconfigure(0, weight=1)               4
        self.columnconfigure(0, weight=1)            5
        self.quit = Button(self, text='Quit', command=self.quit)
        self.quit.grid(row=0, column=0,              6
            sticky=tk.N+tk.S+tk.E+tk.W)
```

1 The “top level window” is the outermost window on the screen. However, this window is not your `Application` window—it is the *parent* of the `Application` instance. To get the top-level window,

call the `.winfo_toplevel()` method on any widget in your application; see Section 26, "Universal widget methods" (p. 97).

2. This line makes row 0 of the top level window's grid stretchable.
3. This line makes column 0 of the top level window's grid stretchable.
4. Makes row 0 of the `Application` widget's grid stretchable.
5. Makes column 0 of the `Application` widget's grid stretchable.
6. The argument `sticky=tk.N+tk.S+tk.E+tk.W` makes the button expand to fill its cell of the grid.

There is one more change that must be made. In the constructor, change the second line as shown:

```
    def __init__(self, master=None):
        tk.Frame.__init__(self, master)
        self.grid(sticky=tk.N+tk.S+tk.E+tk.W)
        self.createWidgets()
```

The argument `sticky=tk.N+tk.S+tk.E+tk.aW` to `self.grid()` is necessary so that the `Application` widget will expand to fill its cell of the top-level window's grid.

5. Standard attributes

Before we look at the widgets, let's take a look at how some of their common attributes—such as sizes, colors and fonts—are specified.

- Each widget has a set of *options* that affect its appearance and behavior—attributes such as fonts, colors, sizes, text labels, and such.
- You can specify options when calling the widget's constructor using keyword arguments such as `text='PANIC!'` or `height=20`.
- After you have created a widget, you can later change any option by using the widget's `.config()` method. You can retrieve the current setting of any option by using the widget's `.cget()` method. See Section 26, "Universal widget methods" (p. 97) for more on these methods.

5.1. Dimensions

Various lengths, widths, and other dimensions of widgets can be described in many different units.

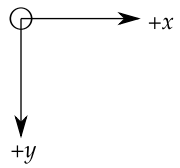
- If you set a dimension to an integer, it is assumed to be in pixels.
- You can specify units by setting a dimension to a string containing a number followed by:

Table 3. Dimensional units

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

5.2. The coordinate system

As in most contemporary display systems, the origin of each coordinate system is at its upper left corner, with the *x* coordinate increasing toward the right, and the *y* coordinate increasing toward the bottom:



The base unit is the pixel, with the top left pixel having coordinates (0,0). Coordinates that you specify as integers are always expressed in pixels, but any coordinate may be specified as a dimensioned quantity; see Section 5.1, "Dimensions" (p. 9).

5.3. Colors

There are two general ways to specify colors in *Tkinter*.

- You can use a string specifying the proportion of red, green, and blue in hexadecimal digits:

`#rgb`	Four bits per color
`#rrggbb`	Eight bits per color
`#rrrgggbbb`	Twelve bits per color

  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. Other names may work, depending on your local installation.

5.4. Type fonts

Depending on your platform, there may be up to three ways to specify type style.

- As a tuple whose first element is the font family, followed by a size (in points if positive, in pixels if negative), optionally followed by a string containing one or more of the style modifiers `bold`, `italic`, `underline`, and `overstrike`.

  Examples: `('Helvetica', '16')` for a 16-point Helvetica regular; `('Times', '24', 'bold italic')` for a 24-point Times bold italic. For a 20-pixel Times bold font, use `('Times', -20, 'bold')`.

- You can create a "font object" by importing the `tkFont` module and using its `Font` class constructor:

```
import tkFont

font = tkFont.Font(option, ...)
```

  where the options include:

`family`	The font family name as a string.

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

For example, to get a 36-point bold Helvetica italic face:

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

- 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 a good fixed-width font for onscreen use. Use the *xfontsel* program to help you select pleasing fonts.

To get a list of all the families of fonts available on your platform, call this function:

```
tkFont.families()
```

The return value is a list of strings. *Note*: You must create your root window before calling this function.

These methods are defined on all `Font` objects:

`.actual(`*`option`*`=None)`
If you pass no arguments, you get back a dictionary of the font's actual attributes, which may differ from the ones you requested. To get back the value of an attribute, pass its name as an argument.

`.cget(`*`option`*`)`
Returns the value of the given *`option`*.

`.configure(`*`option`*`, ...)`
Use this method to change one or more options on a font. For example, if you have a `Font` object called `titleFont`, if you call `titleFont.configure(family='times', size=18)`, that font will change to 18pt Times and any widgets that use that font will change too.

`.copy()`
Returns a copy of a `Font` object.

`.measure(`*`text`*`)`
Pass this method a string, and it will return the number of pixels of width that string will take in the font. Warning: some slanted characters may extend outside this area.

`.metrics(`*`option`*`)`
If you call this method with no arguments, it returns a dictionary of all the *font metrics*. You can retrieve the value of just one metric by passing its name as an argument. Metrics include:

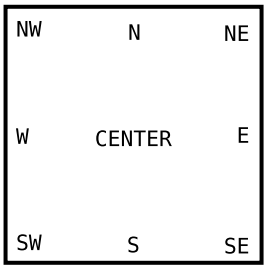
`ascent`	Number of pixels of height between the baseline and the top of the highest ascender.
`descent`	Number of pixels of height between the baseline and the bottom of the lowest ascender.
`fixed`	This value is `0` for a variable-width font and `1` for a monospaced font.
`linespace`	Number of pixels of height total. This is the leading of type set solid in the given font.

5.5. Anchors

The *Tkinter* module defines a number of *anchor* constants that you can use to control where items are positioned relative to their context. For example, anchors can specify where a widget is located inside a frame when the frame is bigger than the widget.

These constants are given as compass points, where north is up and west is to the left. We apologize to our Southern Hemisphere readers for this Northern Hemisphere chauvinism[7].

The anchor constants are shown in this diagram:



For example, if you create a small widget inside a large frame and use the `anchor=tk.SE` option, the widget will be placed in the bottom right corner of the frame. If you used `anchor=tk.N` instead, the widget would be centered along the top edge.

Anchors are also used to define where text is positioned relative to a reference point. For example, if you use `tk.CENTER` as a text anchor, the text will be centered horizontally and vertically around the reference point. Anchor `tk.NW` will position the text so that the reference point coincides with the northwest (top left) corner of the box containing the text. Anchor `tk.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.

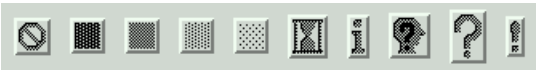
5.6. Relief styles

The *relief style* of a widget refers to certain simulated 3-D effects around the outside of the widget. Here is a screen shot of a row of buttons exhibiting all the possible relief styles:

The width of these borders depends on the `borderwidth` option of the widget. The above graphic shows what they look like with a 5-pixel border; the default border width is 2.

5.7. Bitmaps

For `bitmap` options in widgets, these bitmaps are guaranteed to be available:



[7] http://flourish.org/upsidedownmap/

The graphic above shows `Button` widgets bearing the standard bitmaps. From left to right, they are `'error'`, `'gray75'`, `'gray50'`, `'gray25'`, `'gray12'`, `'hourglass'`, `'info'`, `'questhead'`, `'question'`, and `'warning'`.

You can use your own bitmaps. Any file in `.xbm` (X bit map) format will work. In place of a standard bitmap name, use the string `'@'` followed by the pathname of the `.xbm` file.

5.8. Cursors

There are quite a number of different mouse cursors available. Their names and graphics are shown here. The exact graphic may vary according to your operating system.

Table 4. Values of the `cursor` option

arrow	man
based_arrow_down	middlebutton
based_arrow_up	mouse
boat	pencil
bogosity	pirate
bottom_left_corner	plus
bottom_right_corner	question_arrow
bottom_side	right_ptr
bottom_tee	right_side
box_spiral	right_tee
center_ptr	rightbutton
circle	rtl_logo
clock	sailboat
coffee_mug	sb_down_arrow
cross	sb_h_double_arrow
cross_reverse	sb_left_arrow
crosshair	sb_right_arrow
diamond_cross	sb_up_arrow
dot	sb_v_double_arrow

dotbox	shuttle
double_arrow	sizing
draft_large	spider
draft_small	spraycan
draped_box	star
exchange	target
fleur	tcross
gobbler	top_left_arrow
gumby	top_left_corner
hand1	top_right_corner
hand2	top_side
heart	top_tee
icon	trek
iron_cross	ul_angle
left_ptr	umbrella
left_side	ur_angle
left_tee	watch
leftbutton	xterm
ll_angle	X_cursor
lr_angle	

5.9. Images

There are three general methods for displaying graphic images in your *Tkinter* application.

- To display bitmap (two-color) images in the `.xbm` format, refer to Section 5.9.1, "The `BitmapImage` class" (p. 15).
- To display full-color images in the `.gif`, `.pgm`, or `.ppm` format, see Section 5.9.2, "The `PhotoImage` class" (p. 15).

- The Python Imaging Library (PIL) supports images in a much wider variety of formats. Its `ImageTk` class is specifically designed for displaying images within *Tkinter* applications. See the author's companion document for PIL documentation: *Python Imaging Library (PIL) quick reference*[8].

5.9.1. The `BitmapImage` class

To display a two-color image in the `.xbm` format, you will need this constructor:

```
    tk.BitmapImage(file=f[, background=b][, foreground=c])
```

where *f* is the name of the `.xbm` image file.

Normally, foreground (1) bits in the image will be displayed as black pixels, and background (0) bits in the image will be transparent. To change this behavior, use the optional `background=`*b* option to set the background to color *b*, and the optional `foreground=`*c* option to set the foreground to color *c*. For color specification, see Section 5.3, "Colors" (p. 10).

This constructor returns a value that can be used anywhere *Tkinter* expects an image. For example, to display an image as a label, use a `Label` widget (see Section 12, "The `Label` widget" (p. 48)) and supply the `BitmapImage` object as the value of the `image` option:

```
    logo = tk.BitmapImage('logo.xbm', foreground='red')
    Label(image=logo).grid()
```

5.9.2. The `PhotoImage` class

To display a color image in `.gif`, `.pgm`, or `.ppm` format, you will need this constructor:

```
    tk.PhotoImage(file=f)
```

where *f* is the name of the image file. The constructor returns a value that can be used anywhere *Tkinter* expects an image.

5.10. Geometry strings

A *geometry string* is a standard way of describing the size and location of a top-level window on a desktop.

A geometry string has this general form:

```
    'wxh±x±y'
```

where:

- The *w* and *h* parts give the window width and height in pixels. They are separated by the character `'x'`.
- If the next part has the form `+`*x*, it specifies that the left side of the window should be *x* pixels from the left side of the desktop. If it has the form `-`*x*, the right side of the window is *x* pixels from the right side of the desktop.
- If the next part has the form `+`*y*, it specifies that the top of the window should be *y* pixels below the top of the desktop. If it has the form `-`*y*, the bottom of the window will be *y* pixels above the bottom edge of the desktop.

[8] http://www.nmt.edu/tcc/help/pubs/pil/

For example, a window created with `geometry='120x50-0+20'` would be 120 pixels wide by 50 pixels high, and its top right corner will be along the right edge of the desktop and 20 pixels below the top edge.

5.11. Window names

The term *window* describes a rectangular area on the desktop.

- A *top-level* or *root* window is a window that has an independent existence under the window manager. It is decorated with the window manager's decorations, and can be moved and resized independently. Your application can use any number of top-level windows.
- The term "window" also applies to any widget that is part of a top-level window.

Tkinter names all these windows using a hierarchical *window path name*.

- The root window's name is `'.'`.
- Child windows have names of the form `'.n'`, where *n* is some integer in string form. For example, a window named `'.135932060'` is a child of the root window (`'.'`).
- Child windows within child windows have names of the form `'p.n'` where *p* is the name of the parent window and *n* is some integer. For example, a window named `'.135932060.137304468'` has parent window `'.135932060'`, so it is a grandchild of the root window.
- The *relative name* of a window is the part past the last `'.'` in the path name. To continue the previous example, the grandchild window has a relative name `'137304468'`.

To get the path name for a widget *w*, use `str(w)`.

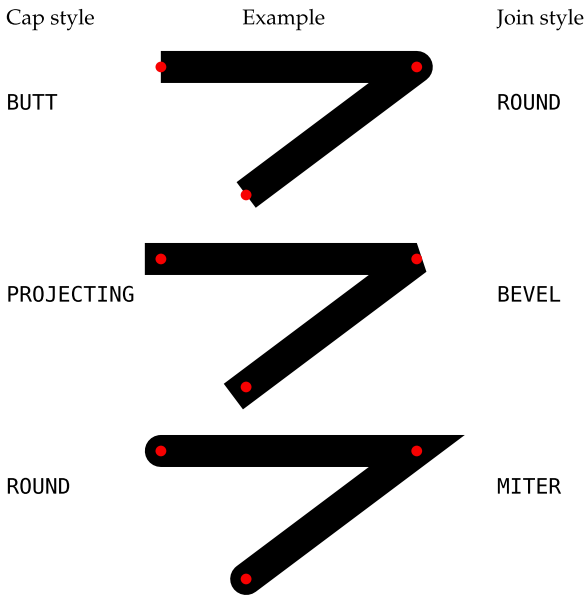
See also Section 26, "Universal widget methods" (p. 97) for methods you can use to operate on window names, especially the `.winfo_name`, `.winfo_parent`, and `.winfo_pathname` methods.

5.12. Cap and join styles

For pleasant and effective rendering of diagrams, sometimes it is a good idea to pay attention to cap and join styles.

- The *cap style* of a line is the shape of the end of the line. Styles are:
  - `tk.BUTT`: The end of the line is cut off square at a line that passes through the endpoint.
  - `tk.PROJECTING`: The end of the line is cut off square, but the cut line projects past the endpoint a distance equal to half the line's width.
  - `tk.ROUND`: The end describes a semicircle centered on the endpoint.
- The *join style* describes the shape where two line segments meet at an angle.
  - `tk.ROUND`: The join is a circle centered on the point where the adjacent line segments meet.
  - `tk.BEVEL`: A flat facet is drawn at an angle intermediate between the angles of the adjacent lines.
  - `tk.MITER`: The edges of the adjacent line segments are continued to meet at a sharp point.

This illustration shows how *Tkinter*'s cap and join options work with a line made of two connected line segments. Small red circles show the location of the points that define this line.



5.13. Dash patterns

A number of widgets allow you to specify a dashed outline. The `dash` and `dashoffset` options give you fine control over the exact pattern of the dashes.

`dash`

This option is specified as a tuple of integers. The first integer specifies how many pixels should be drawn. The second integer specifies how many pixels should be skipped before starting to draw again, and so on. When all the integers in the tuple are exhausted, they are reused in the same order until the border is complete.

For example, `dash=(3,5)` produces alternating 3-pixel dashes separated by 5-pixel gaps. A value of `dash=(7,1,1,1)` produces a dash-and-dot pattern, with the dash seven times as long as the dot or the gaps around the dot. A value of `dash=(5,)` produces alternating five-pixel dashes and five-pixel gaps.

`dashoff`

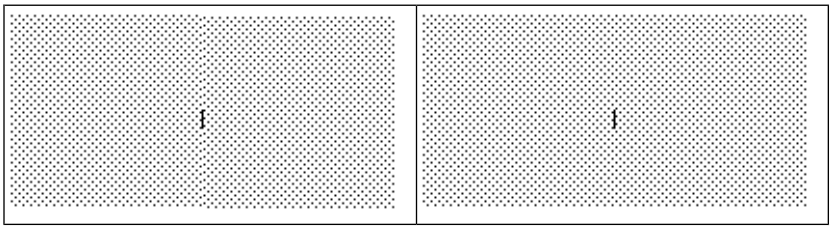
To start the dash pattern in a different point of cycle instead of at the beginning, use an option of `dashoff=`*n*, where *n* is the number of pixels to skip at the beginning of the pattern.

For example, for options `dash=(5, 1, 2, 1)` and `dashoff=3`, the first pattern produced will be: 2 on, 1 off, 2 on, and 1 off. Subsequent patterns will be 5 on, 1 off, 2 on, and 1 off. Here is a screen shot of a line drawn with this combination of options:

5.14. Matching stipple patterns

This may seem like an incredibly picky style point, but if you draw a graphic that has two objects with stippled patterns, a real professional will make sure that the patterns align along their boundary.

Here is an example. The left-hand screen shot shows two adjacent 100×100 squares stippled with the "`gray12`" pattern, but the right-hand square is offset vertically by one pixel. The short black line in the center of the figure is drawn along the boundary of the two figures.



The second screen shot is the same, except that the two 100×100 squares have their stipple patterns lined up.

In practice, this arises in two situations. The alignment of large stippled areas is controlled by an option named `offset`. For figures with stippled outlines, the `outlineoffset` option controls their alignment. Both options have values of one of these forms:

- '*x*,*y*': Offset the stipple patterns by this *x* and *y* value relative to the top-level window or to the canvas's origin.
- '#*x*,*y*': For objects on a canvas, use offset *x* and *y* relative to the top-level window.
- `tk.NE`, `tk.SE`, `tk.SW`, `tk.NW`: Align a corner of the stipple pattern with the corresponding corner of the containing object. For example, `tk.NE` means that the top left corner of the stipple pattern coincides with the top left corner of the area to be stippled.
- `tk.N`, `tk.E`, `tk.S`, `tk.W`: Align the stipple pattern with the center of one side of the containing object. For example, `tk.E` means the center of the stipple pattern will coincide with the center of the right side of the area to be stippled.
- `tk.CENTER`: Align the center of the stipple pattern with the center of the containing object.

6. Exception handling

The exception raised by most programming errors is `tk.TclError`.

7. The `Button` widget

To create a pushbutton in a top-level window or frame named *parent*:

```
w = tk.Button(parent, option=value, ...)
```

The constructor returns the new `Button` widget. Its options include:

Table 5. `Button` widget options

`activebackground`	Background color when the button is under the cursor.
`activeforeground`	Foreground color when the button is under the cursor.
`anchor`	Where the text is positioned on the button. See Section 5.5, "Anchors" (p. 12). For example, `anchor=tk.NE` would position the text at the top right corner of the button.

`bd` or `borderwidth`	Width of the border around the outside of the button; see Section 5.1, "Dimensions" (p. 9). The default is two pixels.
`bg` or `background`	Normal background color.
`bitmap`	Name of one of the standard bitmaps to display on the button (instead of text).
`command`	Function or method to be called when the button is clicked.
`cursor`	Selects the cursor to be shown when the mouse is over the button.
`default`	`tk.NORMAL` is the default; use `tk.DISABLED` if the button is to be initially disabled (grayed out, unresponsive to mouse clicks).
`disabledforeground`	Foreground color used when the button is disabled.
`fg` or `foreground`	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).
`highlightbackground`	Color of the focus highlight when the widget does not have focus.
`highlightcolor`	The color of the focus highlight when the widget has focus.
`highlightthickness`	Thickness of the focus highlight.
`image`	Image to be displayed on the button (instead of text).
`justify`	How to show multiple text lines: `tk.LEFT` to left-justify each line; `tk.CENTER` to center them; or `tk.RIGHT` to right-justify.
`overrelief`	The relief style to be used while the mouse is on the button; default relief is `tk.RAISED`. See Section 5.6, "Relief styles" (p. 12).
`padx`	Additional padding left and right of the text. See Section 5.1, "Dimensions" (p. 9) for the possible values for padding.
`pady`	Additional padding above and below the text.
`relief`	Specifies the relief type for the button (see Section 5.6, "Relief styles" (p. 12)). The default relief is `tk.RAISED`.
`repeatdelay`	See `repeatinterval`, below.
`repeatinterval`	Normally, a button fires only once when the user releases the mouse button. If you want the button to fire at regular intervals as long as the mouse button is held down, set this option to a number of milliseconds to be used between repeats, and set the `repeatdelay` to the number of milliseconds to wait before starting to repeat. For example, if you specify "`repeatdelay=500, repeatinterval=100`" the button will fire after half a second, and every tenth of a second thereafter, until the user releases the mouse button. If the user does not hold the mouse button down at least `repeatdelay` milliseconds, the button will fire normally.
`state`	Set this option to `tk.DISABLED` to gray out the button and make it unresponsive. Has the value `tk.ACTIVE` when the mouse is over it. Default is `tk.NORMAL`.
`takefocus`	Normally, keyboard focus does visit buttons (see Section 53, "Focus: routing keyboard input" (p. 155)), and a *space* character acts as the same as a mouse click, "pushing" the button. You can set the `takefocus` option to zero to prevent focus from visiting the button.

`text`	Text displayed on the button. Use internal newlines to display multiple text lines.
`textvariable`	An instance of `StringVar()` that is associated with the text on this button. If the variable is changed, the new value will be displayed on the button. See Section 52, “Control variables: the values behind the widgets” (p. 153).
`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. For example, `underline=1` would underline the second character of the button's text.
`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. For possible values, see Section 5.1, “Dimensions” (p. 9).

Methods on `Button` objects:

`.flash()`
Causes the button to flash several times between active and normal colors. Leaves the button in the state it was in originally. Ignored if the button is disabled.

`.invoke()`
Calls the button's `command` callback, and returns what that function returns. Has no effect if the button is disabled or there is no callback.

8. The `Canvas` widget

A canvas is a rectangular area intended for drawing pictures or other complex layouts. On it you can place graphics, text, widgets, or frames. See the following sections for methods that create objects on canvases:

- `.create_arc()`: A slice out of an ellipse. See Section 8.7, “`Canvas` arc objects” (p. 28).
- `.create_bitmap()`: An image as a bitmap. See Section 8.8, “`Canvas` bitmap objects” (p. 29).
- `.create_image()`: A graphic image. See Section 8.9, “`Canvas` image objects” (p. 30).
- `.create_line()`: One or more line segments. See Section 8.10, “`Canvas` line objects” (p. 30).
- `.create_oval()`: An ellipse; use this also for drawing circles, which are a special case of an ellipse. See Section 8.11, “`Canvas` oval objects” (p. 32).
- `.create_polygon()`: A polygon. See Section 8.12, “`Canvas` polygon objects” (p. 33).
- `.create_rectangle()`: A rectangle. See Section 8.13, “`Canvas` rectangle objects” (p. 35).
- `.create_text()`: Text annotation. See Section 8.14, “`Canvas` text objects” (p. 37).
- `.create_window()`: A rectangular window. See Section 8.15, “`Canvas` window objects” (p. 38).

To create a `Canvas` object:

```
    w = tk.Canvas(parent, option=value, ...)
```

The constructor returns the new `Canvas` widget. Supported options include:

Table 6. `Canvas` widget options

`bd` or `borderwidth`	Width of the border around the outside of the canvas; see Section 5.1, "Dimensions" (p. 9). The default is two pixels.
`bg` or `background`	Background color of the canvas. Default is a light gray, about `'#E4E4E4'`.
`closeenough`	A `float` that specifies how close the mouse must be to an item to be considered inside it. Default is 1.0.
`confine`	If true (the default), the canvas cannot be scrolled outside of the `scrollregion` (see below).
`cursor`	Cursor used in the canvas. See Section 5.8, "Cursors" (p. 13).
`height`	Size of the canvas in the Y dimension. See Section 5.1, "Dimensions" (p. 9).
`highlightbackground`	Color of the focus highlight when the widget does not have focus. See Section 53, "Focus: routing keyboard input" (p. 155).
`highlightcolor`	Color shown in the focus highlight.
`highlightthickness`	Thickness of the focus highlight. The default value is 1.
`relief`	The relief style of the canvas. Default is `tk.FLAT`. See Section 5.6, "Relief styles" (p. 12).
`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.
`selectbackground`	The background color to use displaying selected items.
`selectborderwidth`	The width of the border to use around selected items.
`selectforeground`	The foreground color to use displaying selected items.
`takefocus`	Normally, focus (see Section 53, "Focus: routing keyboard input" (p. 155)) will cycle through this widget with the tab key only if there are keyboard bindings set for it (see Section 54, "Events" (p. 157) for an overview of keyboard bindings). If you set this option to 1, focus will always visit this widget. Set it to `''` to get the default behavior.
`width`	Size of the canvas in the X dimension. See Section 5.1, "Dimensions" (p. 9).
`xscrollincrement`	Normally, canvases can be scrolled horizontally to any position. You can get this behavior by setting `xscrollincrement` to zero. 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. For more information on scrolling units, see Section 22, "The `Scrollbar` widget" (p. 74).
`xscrollcommand`	If the canvas is scrollable, set this option to the `.set()` method of the horizontal scrollbar.
`yscrollincrement`	Works like `xscrollincrement`, but governs vertical movement.
`yscrollcommand`	If the canvas is scrollable, this option should be the `.set()` method of the vertical scrollbar.

8.1. `Canvas` coordinates

Because the canvas may be larger than the window, and equipped with scrollbars to move the overall canvas around in the window, there are two coordinate systems for each canvas:

- The *window coordinates* of a point are relative to the top left corner of the area on the display where the canvas appears.
- The *canvas coordinates* of a point are relative to the top left corner of the total canvas.

8.2. The `Canvas` display list

The *display list* refers to the sequence of all the objects on the canvas, from background (the "bottom" of the display list) to foreground (the "top").

If two objects overlap, the one *above* the other in the display list means the one closer to the foreground, which will appear in the area of overlap and obscure the one *below*. By default, new objects are always created at the top of the display list (and hence in front of all other objects), but you can re-order the display list.

8.3. `Canvas` object IDs

The *object ID* of an object on the canvas is the value returned by the constructor for that object. All object ID values are simple integers, and the object ID of an object is unique within that canvas.

8.4. `Canvas` tags

A *tag* is a string that you can associate with objects on the canvas.

- A tag can be associated with any number of objects on the canvas, including zero.
- An object can have any number of tags associated with it, including zero.

Tags have many uses. For example, if you are drawing a map on a canvas, and there are text objects for the labels on rivers, you could attach the tag `'riverLabel'` to all those text objects. This would allow you to perform operations on all the objects with that tag, such as changing their color or deleting them.

8.5. `Canvas` *`tagOrId`* arguments

A `tagOrId` argument specifies one or more objects on the canvas.

- If a `tagOrId` argument is an integer, it is treated as an object ID, and it applies only to the unique object with that ID. See Section 8.3, "`Canvas` object IDs" (p. 22).
- If such an argument is a string, it is interpreted as a tag, and selects all the objects that have that tag (if there are any). See Section 8.4, "`Canvas` tags" (p. 22).

8.6. Methods on `Canvas` widgets

All `Canvas` objects support these methods:

`.addtag_above(`*`newTag`*`, `*`tagOrId`*`)`
Attaches a new tag to the object just above the one specified by *`tagOrId`* in the display list. The *`newTag`* argument is the tag you want to attach, as a string.

.addtag_all(*newTag*)
Attaches the given tag *newTag* to all the objects on the canvas.

.addtag_below(*newTag*, *tagOrID*)
Attaches a new tag to the object just below the one specified by *tagOrId* in the display list. The *newTag* argument is a tag string.

.addtag_closest(*newTag*, *x*, *y*, *halo*=None, *start*=None)
Adds a tag to the object closest to screen coordinate (x,y). If there are two or more objects at the same distance, the one higher in the display list is selected.

Use the *halo* argument to increase the effective size of the point. For example, a value of 5 would treat any object within 5 pixels of (x,y) as overlapping.

If an object ID is passed in the *start* argument, this method tags the highest qualifying object that is below *start* in the display list.

.addtag_enclosed(*newTag*, *x1*, *y1*, *x2*, *y2*)
Add tag *newTag* to all objects that occur completely within the rectangle whose top left corner is (*x1*, *y1*) and whose bottom right corner is (*x2*, *y2*).

.addtag_overlapping(*newTag*, *x1*, *y1*, *x2*, *y2*)
Like the previous method, but affects all objects that share at least one point with the given rectangle.

.addtag_withtag(*newTag*, *tagOrId*)
Adds tag *newTag* to the object or objects specified by *tagOrId*.

.bbox(tagOrId=None)
Returns a tuple (x_1, y_1, x_2, y_2) describing a rectangle that encloses all the objects specified by *tagOrId*. If the argument is omitted, returns a rectangle enclosing all objects on the canvas. The top left corner of the rectangle is (x_1, y_1) and the bottom right corner is (x_2, y_2).

.canvasx(*screenx*, gridspacing=None)
Translates a window x coordinate *screenx* to a canvas coordinate. If `gridspacing` is supplied, the canvas coordinate is rounded to the nearest multiple of that value.

.canvasy(*screeny*, gridspacing=None)
Translates a window y coordinate *screeny* to a canvas coordinate. If `gridspacing` is supplied, the canvas coordinate is rounded to the nearest multiple of that value.

.coords(*tagOrId*, *x0*, *y0*, *x1*, *y1*, ..., *xn*, *yn*)
If you pass only the *tagOrId* argument, returns a tuple of the coordinates of the lowest or only object specified by that argument. The number of coordinates depends on the type of object. In most cases it will be a 4-tuple (x_1, y_1, x_2, y_2) describing the bounding box of the object.

You can move an object by passing in new coordinates.

.dchars(*tagOrId*, first=0, last=first)
Deletes characters from a text item or items. Characters between `first` and `last` *inclusive* are deleted, where those values can be integer indices or the string `'end'` to mean the end of the text. For example, for a canvas *C* and an item *I*, `C.dchars(I, 1, 1)` will remove the second character.

.delete(*tagOrId*)
Deletes the object or objects selected by *tagOrId*. It is not considered an error if no items match *tagOrId*.

.dtag(tagOrId, tagToDelete)
Removes the tag specified by *tagToDelete* from the object or objects specified by *tagOrId*.

.find_above(tagOrId)
Returns the ID number of the object just above the object specified by *tagOrId*. If multiple objects match, you get the highest one. Returns an empty tuple if you pass it the object ID of the highest object.

.find_all()
Returns a list of the object ID numbers for all objects on the canvas, from lowest to highest.

.find_below(*tagOrId*)
Returns the object ID of the object just below the one specified by *tagOrId*. If multiple objects match, you get the lowest one. Returns an empty tuple if you pass it the object ID of the lowest object.

.find_closest(*x*, *y*, *halo*=None, *start*=None)
Returns a singleton tuple containing the object ID of the object closest to point (*x*, *y*). If there are no qualifying objects, returns an empty tuple.

Use the *halo* argument to increase the effective size of the point. For example, halo=5 would treat any object within 5 pixels of (*x*, *y*) as overlapping.

If an object ID is passed as the *start* argument, this method returns the highest qualifying object that is below *start* in the display list.

.find_enclosed(*x1*, *y1*, *x2*, *y2*)
Returns a list of the object IDs of all objects that occur completely within the rectangle whose top left corner is (*x1*, *y1*) and bottom right corner is (*x2*, *y2*).

.find_overlapping(*x1*, *y1*, *x2*, *y2*)
Like the previous method, but returns a list of the object IDs of all the objects that share at least one point with the given rectangle.

.find_withtag(*tagOrId*)
Returns a list of the object IDs of the object or objects specified by *tagOrId*.

.focus(*tagOrId*=None)
Moves the focus to the object specified by *tagOrId*. If there are multiple such objects, moves the focus to the first one in the display list that allows an insertion cursor. If there are no qualifying items, or the canvas does not have focus, focus does not move.

If the argument is omitted, returns the ID of the object that has focus, or '' if none of them do.

.gettags(*tagOrId*)
If *tagOrId* is an object ID, returns a list of all the tags associated with that object. If the argument is a tag, returns all the tags for the lowest object that has that tag.

.icursor(*tagOrId*, *index*)
Assuming that the selected item allows text insertion and has the focus, sets the insertion cursor to *index*, which may be either an integer index or the string 'end'. Has no effect otherwise.

.index(*tagOrId*, *specifier*)
Returns the integer index of the given *specifier* in the text item specified by *tagOrId* (the lowest one that, if *tagOrId* specifies multiple objects). The return value is the corresponding position as an integer, with the usual Python convention, where 0 is the position before the first character.

The *specifier* argument may be any of:

- tk.INSERT, to return the current position of the insertion cursor.
- tk.END, to return the position after the last character of the item.
- tk.SEL_FIRST, to return the position of the start of the current text selection. *Tkinter* will raise a tk.TclError exception if the text item does not currently contain the text selection.

- `tk.SEL_LAST`, to return the position after the end of the current text selection, or raise `tk.TclError` if the item does not currently contain the selection.
- A string of the form "`@x,y`", to return the character of the character containing canvas coordinates `(x, y)`. If those coordinates are above or to the left of the text item, the method returns 0; if the coordinates are to the right of or below the item, the method returns the index of the end of the item.

`.insert(tagOrId, specifier, text)`
Inserts the given *`string`* into the object or objects specified by *`tagOrId`*, at the position given by the *`specifier`* argument.

The *`specifier`* values may be:
- Any of the keywords `tk.INSERT`, `tk.END`, `tk.SEL_FIRST`, or `tk.SEL_LAST`. Refer to the description of the `index` method above for the interpretation of these codes.
- The position of the desired insertion, using the normal Python convention for positions in strings.

`.itemcget(tagOrId, option)`
Returns the value of the given configuration *`option`* in the selected object (or the lowest object if *`tagOrId`* specifies more than one). This is similar to the `.cget()` method for *Tkinter* objects.

`.itemconfigure(tagOrId, option, ...)`
If no *`option`* arguments are supplied, returns a dictionary whose keys are the options of the object specified by *`tagOrId`* (the lowest one, if *`tagOrId`* specifies multiple objects).

To change the configuration option of the specified item, supply one or more keyword arguments of the form *`option=value`*.

`.move(tagOrId, xAmount, yAmount)`
Moves the items specified by *`tagOrId`* by adding *`xAmount`* to their x coordinates and *`yAmount`* to their y coordinates.

`.postscript(option, ...)`
Generates an Encapsulated PostScript representation of the canvas's current contents. The options include:

`colormode`	Use `'color'` for color output, `'gray'` for grayscale, or `'mono'` for black and white.
`file`	If supplied, names a file where the PostScript will be written. If this option is not given, the PostScript is returned as a string.
`height`	How much of the Y size of the canvas to print. Default is the entire visible height of the canvas.
`rotate`	If false, the page will be rendered in portrait orientation; if true, in landscape.
`x`	Leftmost canvas coordinate of the area to print.
`y`	Topmost canvas coordinate of the area to print.
`width`	How much of the X size of the canvas to print. Default is the visible width of the canvas.

`.scale(tagOrId, xOffset, yOffset, xScale, yScale)`
Scale all objects according to their distance from a point P=(*`xOffset`*, *`yOffset`*). The scale factors *`xScale`* and *`yScale`* are based on a value of 1.0, which means no scaling. Every point in the objects selected by *`tagOrId`* is moved so that its x distance from P is multiplied by *`xScale`* and its y distance is multiplied by *`yScale`*.

This method will not change the size of a text item, but may move it.

.scan_dragto(*x*, *y*, *gain*=10.0)
See the `.scan_mark()` method below.

.scan_mark(*x*, *y*)
This method is used to implement fast scrolling of a canvas. The intent is that the user will press and hold a mouse button, then move the mouse up to scan (scroll) the canvas horizontally and vertically in that direction at a rate that depends on how far the mouse has moved since the mouse button was depressed.

To implement this feature, bind the mouse's button-down event to a handler that calls `scan_mark(x, y)` where *x* and *y* are the current mouse coordinates. Bind the `<Motion>` event to a handler that, assuming the mouse button is still down, calls `scan_dragto(x, y, gain)` where *x* and *y* are the current mouse coordinates.

The *gain* argument controls the rate of scanning. This argument has a default value of 10.0. Use larger numbers for faster scanning.

.select_adjust(*oid*, *specifier*)
Adjusts the boundaries of the current text selection to include the position given by the *specifier* argument, in the text item with the object ID *oid*.

The current selection anchor is also set to the specified position. For a discussion of the selection anchor, see the canvas `select_from` method below.

For the values of *specifier*, see the canvas `insert` method above.

.select_clear()
Removes the current text selection, if it is set. If there is no current selection, does nothing.

.select_from(*oid*, *specifier*)
This method sets the *selection anchor* to the position given by the *specifier* argument, within the text item whose object ID is given by *oid*.

The currently selected text on a given canvas is specified by three positions: the start position, the end position, and the selection anchor, which may be anywhere within those two positions.

To change the position of the currently selected text, use this method in combination with the `select_adjust`, `select_from`, and `select_to` canvas methods (q.v.).

.select_item()
If there is a current text selection on this canvas, return the object ID of the text item containing the selection. If there is no current selection, this method returns `None`.

.select_to(*oid*, *specifier*)
This method changes the current text selection so that it includes the select anchor and the position given by *specifier* within the text item whose object ID is given by *oid*. For the values of *specifier*, see the canvas `insert` method above.

.tag_bind(*tagOrId*, *sequence*=None, function=None, add=None)
Binds events to objects on the canvas. For the object or objects selected by *tagOrId*, associates the handler *function* with the event *sequence*. If the *add* argument is a string starting with `'+'`, the new binding is added to existing bindings for the given *sequence*, otherwise the new binding replaces that for the given *sequence*.

For general information on event bindings, see Section 54, “Events” (p. 157).

Note that the bindings are applied to items that have this tag at the time of the `tag_bind` method call. If tags are later removed from those items, the bindings will persist on those items. If the tag you specify is later applied to items that did not have that tag when you called `tag_bind`, that binding will *not* be applied to the newly tagged items.

.tag_lower(*tagOrId*, *belowThis*)
Moves the object or objects selected by *tagOrId* within the display list to a position just below the first or only object specied by the tag or ID *belowThis*.

If there are multiple items with tag *tagOrId*, their relative stacking order is preserved.

This method does not affect canvas window items. To change a window item's stacking order, use a `lower` or `lift` method on the window.

.tag_raise(*tagOrId*, *aboveThis*)
Moves the object or objects selected by *tagOrId* within the display list to a position just above the first or only object specied by the tag or ID *aboveThis*.

If there are multiple items with tag *tagOrId*, their relative stacking order is preserved.

This method does not affect canvas window items. To change a window item's stacking order, use a `lower` or `lift` method on the window.

.tag_unbind(*tagOrId*, *sequence*, *funcId*=None)
Removes bindings for handler *funcId* and event *sequence* from the canvas object or objects specified by *tagOrId*. See Section 54, “Events” (p. 157).

.type(*tagOrId*)
Returns the type of the first or only object specified by *tagOrId*. The return value will be one of the strings `'arc'`, `'bitmap'`, `'image'`, `'line'`, `'oval'`, `'polygon'`, `'rectangle'`, `'text'`, or `'window'`.

.xview(tk.MOVETO, *fraction*)
This method scrolls the canvas relative to its image, and is intended for binding to the `command` option of a related scrollbar. The canvas is scrolled horizontally to a position given by *offset*, where 0.0 moves the canvas to its leftmost position and 1.0 to its rightmost position.

.xview(tk.SCROLL, *n*, *what*)
This method moves the canvas left or right: the *what* argument specifies how much to move and can be either `tk.UNITS` or `tk.PAGES`, and *n* tells how many units to move the canvas to the right relative to its image (or left, if negative).

The size of the move for `tk.UNITS` is given by the value of the canvas's `xscrollincrement` option; see Section 22, “The `Scrollbar` widget” (p. 74).

For movements by `tk.PAGES`, *n* is multiplied by nine-tenths of the width of the canvas.

.xview_moveto(*fraction*)
This method scrolls the canvas in the same way as `.xview(tk.MOVETO, fraction)`.

.xview_scroll(*n*, *what*)
Same as `.xview(tk.SCROLL, n, what)`.

.yview(tk.MOVETO, *fraction*)
The vertical scrolling equivalent of `.xview(tk.MOVETO,…)`.

.yview(tk.SCROLL, *n*, *what*)
The vertical scrolling equivalent of `.xview(tk.SCROLL,…)`.

.yview_moveto(*fraction*)
The vertical scrolling equivalent of `.xview()`.

.yview_scroll(*n*, *what*)
The vertical scrolling equivalents of `.xview()`, `.xview_moveto()`, and `.xview_scroll()`.

8.7. `Canvas` arc objects

An *arc object* on a canvas, in its most general form, is a wedge-shaped slice taken out of an ellipse. This includes whole ellipses and circles as special cases. See Section 8.11, "`Canvas` oval objects" (p. 32) for more on the geometry of the ellipse drawn.

To create an arc object on a canvas *C*, use:

```
id = C.create_arc(x0, y0, x1, y1, option, ...)
```

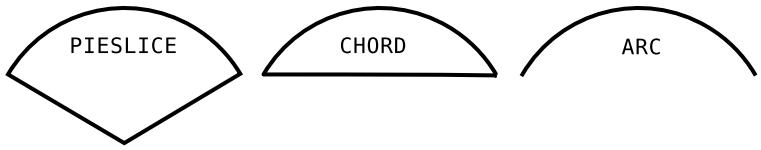
The constructor returns the object ID of the new arc object on canvas *C*.

Point (*x0*,*y0*) is the top left corner and (*x1*,*y1*) the lower right corner of a rectangle into which the ellipse is fit. If this rectangle is square, you get a circle.

The various options include:

Table 7. `Canvas arc` options

`activedash`	These options apply when the arc is in the `tk.ACTIVE` state, that is, when the mouse is over the arc. For example, the `activefill` option specifies the interior color when the arc is active. For option values, see `dash`, `fill`, `outline`, `outlinestipple`, `stipple`, and `width`, respectively.
`activefill`	
`activeoutline`	
`activeoutlinestipple`	
`activestipple`	
`activewidth`	
`dash`	Dash pattern for the outline. See Section 5.13, "Dash patterns" (p. 17).
`dashoffset`	Dash pattern offset for the outline. See Section 5.13, "Dash patterns" (p. 17).
`disableddash`	These options apply when the arc's `state` is `tk.DISABLED`.
`disabledfill`	
`disabledoutline`	
`disabledoutlinestipple`	
`disabledstipple`	
`disabledwidth`	
`extent`	Width of the slice in degrees. The slice starts at the angle given by the *`start`* option and extends counterclockwise for *`extent`* degrees.
`fill`	By default, the interior of an arc is transparent, and `fill=''` will select this behavior. You can also set this option to any color and the interior of the arc will be filled with that color.
`offset`	Stipple pattern offset for the interior of the arc. See Section 5.14, "Matching stipple patterns" (p. 17).
`outline`	The color of the border around the outside of the slice. Default is black.
`outlineoffset`	Stipple pattern offset for the outline. See Section 5.14, "Matching stipple patterns" (p. 17).
`outlinestipple`	If the `outline` option is used, this option specifies a bitmap used to stipple the border. Default is black, and that default can be specified by setting `outlinestipple=''`.

`start`	Starting angle for the slice, in degrees, measured from +x direction. If omitted, you get the entire ellipse.
`state`	This option is `tk.NORMAL` by default. It may be set to `tk.HIDDEN` to make the arc invisible or to `tk.DISABLED` to gray out the arc and make it unresponsive to events.
`stipple`	A bitmap indicating how the interior fill of the arc will be stippled. Default is `stipple=''` (solid). You'll probably want something like `stipple='gray25'`. Has no effect unless *`fill`* has been set to some color.
`style`	The default is to draw the whole arc; use `style=tk.PIESLICE` for this style. To draw only the circular arc at the edge of the slice, use `style=tk.ARC`. To draw the circular arc and the chord (a straight line connecting the endpoints of the arc), use `style=tk.CHORD`. 
`tags`	If a single string, the arc is tagged with that string. Use a tuple of strings to tag the arc with multiple tags. See Section 8.4, "`Canvas` tags" (p. 22).
`width`	Width of the border around the outside of the arc. Default is 1 pixel.

8.8. `Canvas` bitmap objects

A bitmap object on a canvas is shown as two colors, the background color (for 0 data values) and the foreground color (for 1 values).

To create a bitmap object on a canvas *`C`*, use:

```
    id = C.create_bitmap(x, y, *options ...)
```

which returns the integer ID number of the image object for that canvas.

The *`x`* and *`y`* values are the reference point that specifies where the bitmap is placed.

Options include:

Table 8. `Canvas` bitmap options

`activebackground`	These options specify the `background`, `bitmap`, and `foreground` values when the bitmap is active, that is, when the mouse is over the bitmap.
`activebitmap`	
`activeforeground`	
`anchor`	The bitmap is positioned relative to point (*`x`*, *`y`*). The default is anchor=`tk.CENTER`, meaning that the bitmap is centered on the (*`x`*, *`y`*) position. See Section 5.5, "Anchors" (p. 12) for the various `anchor` option values. For example, if you specify `anchor=tk.NE`, the bitmap will be positioned so that point (*`x`*, *`y`*) is located at the northeast (top right) corner of the bitmap.
`background`	The color that will appear where there are 0 values in the bitmap. The default is `background=''`, meaning transparent.

`bitmap`	The bitmap to be displayed; see Section 5.7, “Bitmaps” (p. 12).
`disabledbackground`	These options specify the background, bitmap, and foreground to be used when the bitmap's `state` is `tk.DISABLED`.
`disabledbitmap`	
`disabledforeground`	
`foreground`	The color that will appear where there are 1 values in the bitmap. The default is `foreground='black'`.
`state`	By default, items are created with `state=tk.NORMAL`. Use `tk.DISABLED` to make the item grayed out and unresponsive to events; use `tk.HIDDEN` to make the item invisible.
`tags`	If a single string, the bitmap is tagged with that string. Use a tuple of strings to tag the bitmap with multiple tags. See Section 8.4, “`Canvas` tags” (p. 22).

8.9. `Canvas` image objects

To display a graphics image on a canvas *C*, use:

```
id = C.create_image(x, y, option, ...)
```

This constructor returns the integer ID number of the image object for that canvas.

The image is positioned relative to point (*x*, *y*). Options include:

Table 9. `Canvas image` options

`activeimage`	Image to be displayed when the mouse is over the item. For option values, see `image` below.
`anchor`	The default is `anchor=tk.CENTER`, meaning that the image is centered on the (*x*, *y*) position. See Section 5.5, “Anchors” (p. 12) for the possible values of this option. For example, if you specify `anchor=tk.S`, the image will be positioned so that point (*x*, *y*) is located at the center of the bottom (south) edge of the image.
`disabledimage`	Image to be displayed when the item is inactive. For option values, see `image` below.
`image`	The image to be displayed. See Section 5.9, “Images” (p. 14), above, for information about how to create images that can be loaded onto canvases.
`state`	Normally, image objects are created in state `tk.NORMAL`. Set this value to `tk.DISABLED` to make it grayed-out and unresponsive to the mouse. If you set it to `tk.HIDDEN`, the item is invisible.
`tags`	If a single string, the image is tagged with that string. Use a tuple of strings to tag the image with multiple tags. See Section 8.4, “`Canvas` tags” (p. 22).

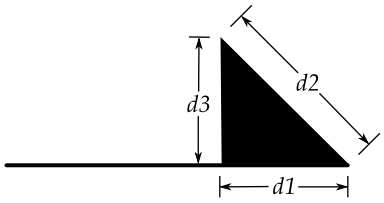
8.10. `Canvas` line objects

In general, a line can consist of any number of segments connected end to end, and each segment can be straight or curved. To create a canvas line object on a canvas *C*, use:

```
id = C.create_line(x0, y0, x1, y1, ..., xn, yn, option, ...)
```

The line goes through the series of points (*x0*, *y0*), (*x1*, *y1*), … (*xn*, *yn*). Options include:

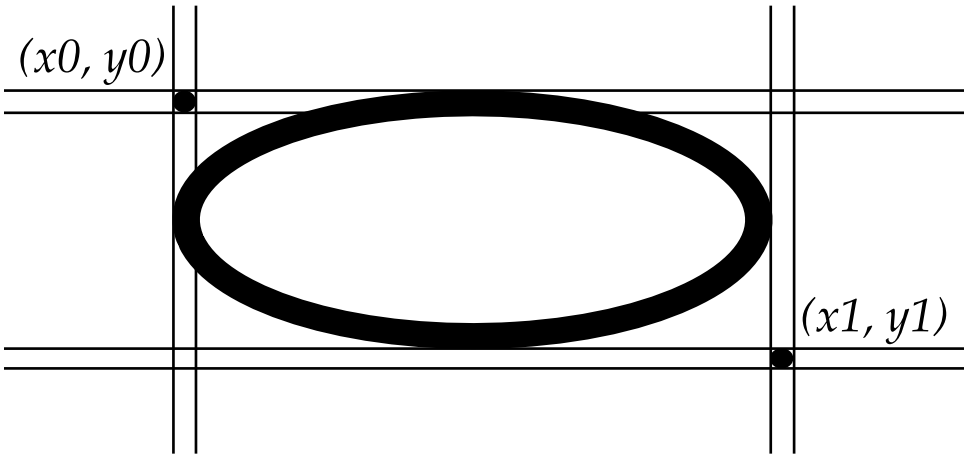
Table 10. `Canvas` line options

`activedash`	These options specify the `dash`, `fill`, `stipple`, and `width` values to be used when the line is active, that is, when the mouse is over it.
`activefill`	
`activestipple`	
`activewidth`	
`arrow`	The default is for the line to have no arrowheads. Use `arrow=tk.FIRST` to get an arrowhead at the (*x0*, *y0*) end of the line. Use `arrow=tk.LAST` to get an arrowhead at the far end. Use `arrow=tk.BOTH` for arrowheads at both ends.
`arrowshape`	A tuple `(d1, d2, d3)` that describes the shape of the arrowheads added by the `arrow` option. Default is `(8,10,3)`. 
`capstyle`	You can specify the shape of the ends of the line with this option; see Section 5.12, “Cap and join styles” (p. 16). The default option is `tk.BUTT`.
`dash`	To produce a dashed line, specify this option; see Section 5.13, “Dash patterns” (p. 17). The default appearance is a solid line.
`dashoffset`	If you specify a `dash` pattern, the default is to start the specified pattern at the beginning of the line. The `dashoffset` option allows you to specify that the start of the dash pattern occurs at a given distance after the start of the line. See Section 5.13, “Dash patterns” (p. 17).
`disableddash`	The `dash`, `fill`, `stipple`, and `width` values to be used when the item is in the `tk.DISABLED` state.
`disabledfill`	
`disabledstipple`	
`disabledwidth`	
`fill`	The color to use in drawing the line. Default is `fill='black'`.
`joinstyle`	For lines that are made up of more than one line segment, this option controls the appearance of the junction between segments. For more details, see Section 5.12, “Cap and join styles” (p. 16). The default style is `ROUND`
`offset`	For stippled lines, the purpose of this option is to match the item's stippling pattern with those of adjacent objects. See Section 5.14, “Matching stipple patterns” (p. 17)..
`smooth`	If true, the line is drawn as a series of parabolic splines fitting the point set. Default is false, which renders the line as a set of straight segments.
`splinesteps`	If the `smooth` option is true, each spline is rendered as a number of straight line segments. The `splinesteps` option specifies the number of segments used to approximate each section of the line; the default is `splinesteps=12`.

`state`	Normally, line items are created in state `tk.NORMAL`. Set this option to `tk.HIDDEN` to make the line invisible; set it to `tk.DISABLED` to make it unresponsive to the mouse.
`stipple`	To draw a stippled line, set this option to a bitmap that specifies the stippling pattern, such as `stipple='gray25'`. See Section 5.7, "Bitmaps" (p. 12) for the possible values.
`tags`	If a single string, the line is tagged with that string. Use a tuple of strings to tag the line with multiple tags. See Section 8.4, "`Canvas` tags" (p. 22).
`width`	The line's width. Default is 1 pixel. See Section 5.1, "Dimensions" (p. 9) for possible values.

8.11. `Canvas` oval objects

Ovals, mathematically, are ellipses, including circles as a special case. The ellipse is fit into a rectangle defined by the coordinates (*`x0`*, *`y0`*) of the top left corner and the coordinates (*`x1`*, *`y1`*) of a point just *outside of* the bottom right corner.



The oval will coincide with the top and left-hand lines of this box, but will fit just inside the bottom and right-hand sides.

To create an ellipse on a canvas *`C`*, use:

```
id = C.create_oval(x0, y0, x1, y1, option, ...)
```

which returns the object ID of the new oval object on canvas *`C`*.

Options for ovals:

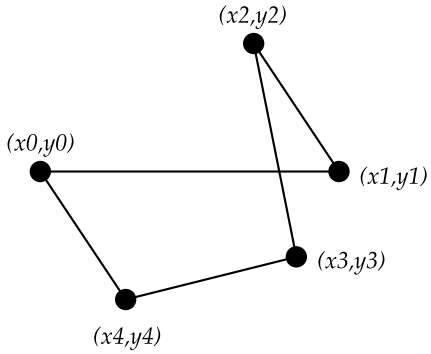
Table 11. `Canvas oval` options

`activedash`	These options specify the dash pattern, fill color, outline color, outline stipple pattern, interior stipple pattern, and outline width values to be used when the oval is in the `tk.ACTIVE` state, that is, when the mouse is over the oval. For option values, see `dash`, `fill`, `outline`, `outlinestipple`, `stipple`, and `width`.
`activefill`	
`activeoutline`	
`activeoutlinestipple`	

activestipple	
activewidth	
dash	To produce a dashed border around the oval, set this option to a dash pattern; see Section 5.13, “Dash patterns” (p. 17)
dashoffset	When using the dash option, the dashoffset option is used to change the alignment of the border's dash pattern relative to the oval. See Section 5.14, “Matching stipple patterns” (p. 17).
disableddash	These options specify the appearance of the oval when the item's state is tk.DISABLED.
disabledfill	
disabledoutline	
disabledoutlinestipple	
disabledstipple	
disabledwidth	
fill	The default appearance of an oval's interior is transparent, and a value of fill='' will select this behavior. You can also set this option to any color and the interior of the ellipse will be filled with that color; see Section 5.3, “Colors” (p. 10).
offset	Stipple pattern offset of the interior. See Section 5.14, “Matching stipple patterns” (p. 17).
outline	The color of the border around the outside of the ellipse. Default is outline='black'.
outlineoffset	Stipple pattern offset of the border. See Section 5.14, “Matching stipple patterns” (p. 17).
stipple	A bitmap indicating how the interior of the ellipse will be stippled. Default is stipple='', which means a solid color. A typical value would be stipple='gray25'. Has no effect unless the fill has been set to some color. See Section 5.7, “Bitmaps” (p. 12).
outlinestipple	Stipple pattern to be used for the border. For option values, see stipple below.
state	By default, oval items are created in state tk.NORMAL. Set this option to tk.DISABLED to make the oval unresponsive to mouse actions. Set it to tk.HIDDEN to make the item invisible.
tags	If a single string, the oval is tagged with that string. Use a tuple of strings to tag the oval with multiple tags. See Section 8.4, “Canvas tags” (p. 22).
width	Width of the border around the outside of the ellipse. Default is 1 pixel; see Section 5.1, “Dimensions” (p. 9) for possible values. If you set this to zero, the border will not appear. If you set this to zero and make the fill transparent, you can make the entire oval disappear.

8.12. Canvas polygon objects

As displayed, a polygon has two parts: its outline and its interior. Its geometry is specified as a series of vertices [(x0, y0), (x1, y1), … (xn, yn)], but the actual perimeter includes one more segment from (xn, yn) back to (x0, y0). In this example, there are five vertices:



To create a new polygon object on a canvas *C*:

```
id = C.create_polygon(x0, y0, x1, y1, ..., option, ...)
```

The constructor returns the object ID for that object. Options:

Table 12. `Canvas polygon` options

`activedash`	These options specify the appearance of the polygon when it is in the `tk.ACTIVE` state, that is, when the mouse is over it. For option values, see `dash`, `fill`, `outline`, `outlinestipple`, `stipple`, and `width`.
`activefill`	
`activeoutline`	
`activeoutlinestipple`	
`activestipple`	
`activewidth`	
`dash`	Use this option to produce a dashed border around the polygon. See Section 5.13, “Dash patterns” (p. 17).
`dashoffset`	Use this option to start the dash pattern at some point in its cycle other than the beginning. See Section 5.13, “Dash patterns” (p. 17).
`disableddash`	These options specify the appearance of the polygon when its `state` is `tk.DISABLED`.
`disabledfill`	
`disabledoutline`	
`disabledoutlinestipple`	
`disabledstipple`	
`disabledwidth`	
`fill`	You can color the interior by setting this option to a color. The default appearance for the interior of a polygon is transparent, and you can set `fill=''` to get this behavior. See Section 5.3, “Colors” (p. 10).
`joinstyle`	This option controls the appearance of the intersections between adjacent sides of the polygon. See Section 5.12, “Cap and join styles” (p. 16).
`offset`	Offset of the stipple pattern in the interior of the polygon. See Section 5.14, “Matching stipple patterns” (p. 17).
`outline`	Color of the outline; defaults to `outline=''`, which makes the outline transparent.

`outlineoffset`	Stipple offset for the border. See Section 5.14, “Matching stipple patterns” (p. 17).
`outlinestipple`	Use this option to get a stippled border around the polygon. The option value must be a bitmap; see Section 5.7, “Bitmaps” (p. 12).
`smooth`	The default outline uses straight lines to connect the vertices; use `smooth=0` to get that behavior. If you use `smooth=1`, you get a continuous spline curve. Moreover, if you set `smooth=1`, you can make any segment straight by duplicating the coordinates at each end of that segment.
`splinesteps`	If the `smooth` option is true, each spline is rendered as a number of straight line segments. The `splinesteps` option specifies the number of segments used to approximate each section of the line; the default is `splinesteps=12`.
`state`	By default, polygons are created in the `tk.NORMAL` state. Set this option to `tk.HIDDEN` to make the polygon invisible, or set it to `tk.DISABLED` to make it unresponsive to the mouse.
`stipple`	A bitmap indicating how the interior of the polygon will be stippled. Default is `stipple=''`, which means a solid color. A typical value would be `stipple='gray25'`. Has no effect unless the `fill` has been set to some color. See Section 5.7, “Bitmaps” (p. 12).
`tags`	If a single string, the polygon is tagged with that string. Use a tuple of strings to tag the polygon with multiple tags. See Section 8.4, “`Canvas` tags” (p. 22).
`width`	Width of the outline; defaults to 1. See Section 5.1, “Dimensions” (p. 9).

8.13. `Canvas` rectangle objects

Each rectangle is specified as two points: (*`x0`*, *`y0`*) is the top left corner, and (*`x1`*, *`y1`*) is the location of the pixel just *outside* of the bottom right corner.

For example, the rectangle specified by top left corner (100,100) and bottom right corner (102,102) is a square two pixels by two pixels, including pixel (101,101) but *not* including (102,102).

Rectangles are drawn in two parts:

- The outline lies inside the rectangle on its top and left sides, but *outside* the rectangle on its bottom and right side. The default appearance is a one-pixel-wide black border.

  For example, consider a rectangle with top left corner (10,10) and bottom right corner (11,11). If you request no border (`width=0`) and green fill (`fill='green'`), you will get one green pixel at (10,10). However, if you request the same options with a black border (`width=1`), you will get four black pixels at (10,10), (10,11), (11,10), and (11,11).

- The fill is the area inside the outline. Its default appearance is transparent.

To create a rectangle object on canvas *`C`*:

```
    id = C.create_rectangle(x0, y0, x1, y1, option, ...)
```

This constructor returns the object ID of the rectangle on that canvas. Options include:

Table 13. `Canvas rectangle` options

`activedash`	These options specify the appearance of the rectangle when its `state` is `tk.ACTIVE`, that is, when the mouse is on top of the rectangle. For option values, refer to `dash`, `fill`, `outline`, `outlinestipple`, `stipple`, and `width` below.
`activefill`	
`activeoutline`	
`activeoutlinestipple`	
`activestipple`	
`activewidth`	
`dash`	To produce a dashed border around the rectangle, use this option to specify a dash pattern. See Section 5.13, “Dash patterns” (p. 17).
`dashoffset`	Use this option to start the border's dash pattern at a different point in the cycle; see Section 5.13, “Dash patterns” (p. 17).
`disableddash`	These options specify the appearance of the rectangle when its `state` is `tk.DISABLED`.
`disabledfill`	
`disabledoutline`	
`disabledoutlinestipple`	
`disabledstipple`	
`disabledwidth`	
`fill`	By default, the interior of a rectangle is empty, and you can get this behavior with `fill=''`. You can also set the option to a color; see Section 5.3, “Colors” (p. 10).
`offset`	Use this option to change the offset of the interior stipple pattern. See Section 5.14, “Matching stipple patterns” (p. 17).
`outline`	The color of the border. Default is `outline='black'`.
`outlineoffset`	Use this option to adjust the offset of the stipple pattern in the outline; see Section 5.14, “Matching stipple patterns” (p. 17).
`outlinestipple`	Use this option to produce a stippled outline. The pattern is specified by a bitmap; see Section 5.7, “Bitmaps” (p. 12).
`state`	By default, rectangles are created in the `tk.NORMAL` state. The state is `tk.ACTIVE` when the mouse is over the rectangle. Set this option to `tk.DISABLED` to gray out the rectangle and make it unresponsive to mouse events.
`stipple`	A bitmap indicating how the interior of the rectangle will be stippled. Default is `stipple=''`, which means a solid color. A typical value would be `stipple='gray25'`. Has no effect unless the `fill` has been set to some color. See Section 5.7, “Bitmaps” (p. 12).
`tags`	If a single string, the rectangle is tagged with that string. Use a tuple of strings to tag the rectangle with multiple tags. See Section 8.4, “`Canvas` tags” (p. 22).
`width`	Width of the border. Default is 1 pixel. Use `width=0` to make the border invisible. See Section 5.1, “Dimensions” (p. 9).

8.14. `Canvas` text objects

You can display one or more lines of text on a canvas *C* by creating a text object:

```
id = C.create_text(x, y, option, ...)
```

This returns the object ID of the text object on canvas *C*. Options include:

Table 14. `Canvas text` options

`activefill`	The text color to be used when the text is active, that is, when the mouse is over it. For option values, see `fill` below.
`activestipple`	The stipple pattern to be used when the text is active. For option values, see `stipple` below.
`anchor`	The default is `anchor=tk.CENTER`, meaning that the text is centered vertically and horizontally around position (*x*, *y*). See Section 5.5, "Anchors" (p. 12) for possible values. For example, if you specify `anchor=tk.SW`, the text will be positioned so its lower left corner is at point (*x*, *y*).
`disabledfill`	The text color to be used when the text object's `state` is `tk.DISABLED`. For option values, see `fill` below.
`disabledstipple`	The stipple pattern to be used when the text is disabled. For option values, see `stipple` below.
`fill`	The default text color is black, but you can render it in any color by setting the `fill` option to that color. See Section 5.3, "Colors" (p. 10).
`font`	If you don't like the default font, set this option to any font value. See Section 5.4, "Type fonts" (p. 10).
`justify`	For multi-line textual displays, this option controls how the lines are justified: `tk.LEFT` (the default), `tk.CENTER`, or `tk.RIGHT`.
`offset`	The stipple offset to be used in rendering the text. For more information, see Section 5.14, "Matching stipple patterns" (p. 17).
`state`	By default, the text item's state is `tk.NORMAL`. Set this option to `tk.DISABLED` to make in unresponsive to mouse events, or set it to `tk.HIDDEN` to make it invisible.
`stipple`	A bitmap indicating how the text will be stippled. Default is `stipple=''`, which means solid. A typical value would be `stipple='gray25'`. See Section 5.7, "Bitmaps" (p. 12).
`tags`	If a single string, the text object is tagged with that string. Use a tuple of strings to tag the object with multiple tags. See Section 8.4, "`Canvas` tags" (p. 22).
`text`	The text to be displayed in the object, as a string. Use newline characters (`'\n'`) to force line breaks.
`width`	If you don't specify a `width` option, the text will be set inside a rectangle as long as the longest line. However, you can also set the `width` option to a dimension, and each line of the text will be broken into shorter lines, if necessary, or even broken within words, to fit within the specified width. See Section 5.1, "Dimensions" (p. 9).

You can change the text displayed in a text item.

- To retrieve the text from an item with object ID *I* on a canvas *C*, call `C.itemcget(I, 'text')`.

- To replace the text in an item with object ID *I* on a canvas *C* with the text from a string *S*, call `C.itemconfigure(I, text=S)`.

A number of canvas methods allow you to manipulate text items. See Section 8.6, "Methods on `Canvas` widgets" (p. 22), especially `dchars`, `focus`, `icursor`, `index`, and `insert`.

8.15. `Canvas` window objects

You can place any *Tkinter* widget onto a canvas by using a *canvas window* object. A window is a rectangular area that can hold one *Tkinter* widget. The widget must be the child of the same top-level window as the canvas, or the child of some widget located in the same top-level window.

If you want to put complex multi-widget objects on a canvas, you can use this method to place a `Frame` widget on the canvas, and then place other widgets inside that frame.

To create a new canvas window object on a canvas *C*:

```
id = C.create_window(x, y, option, ...)
```

This returns the object ID for the window object. Options include:

Table 15. `Canvas window` options

`anchor`	The default is `anchor=tk.CENTER`, meaning that the window is centered on the (*x*, *y*) position. See Section 5.5, "Anchors" (p. 12) for the possible values. For example, if you specify `anchor=tk.E`, the window will be positioned so that point (*x*, *y*) is on the midpoint of its right-hand (east) edge.
`height`	The height of the area reserved for the window. If omitted, the window will be sized to fit the height of the contained widget. See Section 5.1, "Dimensions" (p. 9) for possible values.
`state`	By default, window items are in the `tk.NORMAL` state. Set this option to `tk.DISABLED` to make the window unresponsive to mouse input, or to `tk.HIDDEN` to make it invisible.
`tags`	If a single string, the window is tagged with that string. Use a tuple of strings to tag the window with multiple tags. See Section 8.4, "`Canvas` tags" (p. 22).
`width`	The width of the area reserved for the window. If omitted, the window will be sized to fit the width of the contained widget.
`window`	Use `window=w` where *w* is the widget you want to place onto the canvas. If this is omitted initially, you can later call `C.itemconfigure (id, window=w)` to place the widget *w* onto the canvas, where *id* is the window's object ID..

9. The `Checkbutton` widget

The purpose of a checkbutton widget (sometimes called "checkbox") is to allow the user to read and select a two-way choice. The graphic above shows how checkbuttons look in the off (0) and on (1) state in one implementation: this is a screen shot of two checkbuttons using 24-point Times font.

The *indicator* is the part of the checkbutton that shows its state, and the *label* is the text that appears beside it.

- You will need to create a control variable, an instance of the `IntVar` class, so your program can query and set the state of the checkbutton. See Section 52, "Control variables: the values behind the widgets" (p. 153), below.
- You can also use event bindings to react to user actions on the checkbutton; see Section 54, "Events" (p. 157), below.
- You can disable a checkbutton. This changes its appearance to "grayed out" and makes it unresponsive to the mouse.
- You can get rid of the checkbutton indicator and make the whole widget a "push-push" button that looks recessed when it is set, and looks raised when it is cleared.

To create a checkbutton in an existing parent window or frame *`parent`*:

```
w = tk.Checkbutton(parent, option, ...)
```

The constructor returns a new `Checkbutton` widget. Options include:

Table 16. `Checkbutton` widget options

`activebackground`	Background color when the checkbutton is under the cursor. See Section 5.3, "Colors" (p. 10).
`activeforeground`	Foreground color when the checkbutton is under the cursor.
`anchor`	If the widget inhabits a space larger than it needs, this option specifies where the checkbutton will sit in that space. The default is `anchor=tk.CENTER`. See Section 5.5, "Anchors" (p. 12) for the allowable values. For example, if you use `anchor=NW`, the widget will be placed in the upper left corner of the space.
`bg` or `background`	The normal background color displayed behind the label and indicator. See Section 5.3, "Colors" (p. 10). For the `bitmap` option, this specifies the color displayed for 0-bits in the bitmap.
`bitmap`	To display a monochrome image on a button, set this option to a bitmap; see Section 5.7, "Bitmaps" (p. 12).
`bd` or `borderwidth`	The size of the border around the indicator. Default is two pixels. For possible values, see Section 5.1, "Dimensions" (p. 9).
`command`	A procedure to be called every time the user changes the state of this checkbutton.
`compound`	Use this option to display both text and a graphic, which may be either a bitmap or an image, on the button. Allowable values describe the position of the graphic relative to the text, and may be any of `tk.BOTTOM`, `tk.TOP`, `tk.LEFT`, `tk.RIGHT`, or `tk.CENTER`. For example, `compound=tk.LEFT` would position the graphic to the left of the text.
`cursor`	If you set this option to a cursor name (see Section 5.8, "Cursors" (p. 13)), 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`*. See Section 5.4, "Type fonts" (p. 10).
`fg` or `foreground`	The color used to render the *`text`*. For the `bitmap` option, this specifies the color displayed for 1-bits in the bitmap.
`height`	The number of lines of text on the checkbutton. Default is 1.

highlightbackground	The color of the focus highlight when the checkbutton does not have focus. See Section 53, “Focus: routing keyboard input” (p. 155).
highlightcolor	The color of the focus highlight when the checkbutton has the focus.
highlightthickness	The thickness of the focus highlight. Default is 1. Set to 0 to suppress display of the focus highlight.
image	To display a graphic image on the button, set this option to an image object. See Section 5.9, “Images” (p. 14).
indicatoron	Normally a checkbutton displays as its indicator a box that shows whether the checkbutton is set or not. You can get this behavior by setting indicatoron=1. However, if you set indicatoron=0, the indicator disappears, and the entire widget becomes a push-push button that looks raised when it is cleared and sunken when it is set. You may want to increase the borderwidth value to make it easier to see the state of such a control.
justify	If the *text* contains multiple lines, this option controls how the text is justified: tk.CENTER, tk.LEFT, or tk.RIGHT.
offrelief	By default, checkbuttons use the tk.RAISED relief style when the button is off (cleared); use this option to specify a different relief style to be displayed when the button is off. See Section 5.6, “Relief styles” (p. 12) for values.
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.
overrelief	Use this option to specify a relief style to be displayed when the mouse is over the checkbutton; see Section 5.6, “Relief styles” (p. 12).
padx	How much space to leave to the left and right of the checkbutton and text. Default is 1 pixel. For possible values, see Section 5.1, “Dimensions” (p. 9).
pady	How much space to leave above and below the checkbutton and text. Default is 1 pixel.
relief	With the default value, relief=tk.FLAT, the checkbutton does not stand out from its background. You may set this option to any of the other styles (see Section 5.6, “Relief styles” (p. 12)), or use relief=tk.SOLID, which gives you a solid black frame around it.
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. See Section 5.9, “Images” (p. 14).
state	The default is state=tk.NORMAL, but you can use state=tk.DISABLED to gray out the control and make it unresponsive. If the cursor is currently over the checkbutton, the state is tk.ACTIVE.
takefocus	The default is that the input focus (see Section 53, “Focus: routing keyboard input” (p. 155)) will pass through a checkbutton. If you set takefocus=0, focus will not pass through it.
text	The label displayed next to the checkbutton. Use newlines ('\n') to display multiple lines of text.

`textvariable`	If you need to change the label on a checkbutton during execution, create a StringVar (see Section 52, "Control variables: the values behind the widgets" (p. 153)) to manage the current value, and set this option to that control variable. Whenever the control variable's value changes, the checkbutton's annotation will automatically change as well.
`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; see Section 52, "Control variables: the values behind the widgets" (p. 153). 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.

Methods on checkbuttons include:

`.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.

10. The `Entry` widget

The purpose of an `Entry` widget is to let the user see and modify a *single* line of text.

- If you want to display *multiple* lines of text that can be edited, see Section 24, "The `Text` widget" (p. 82).
- If you want to display one or more lines of text that *cannot* be modified by the user, see Section 12, "The `Label` widget" (p. 48).

Some definitions:

- The *selection* is a highlighted region of the text in an `Entry` widget, if there is one.

  Typically the selection is made by the user with the mouse, and selected text is copied to the system's clipboard. However, *Tkinter* allows you to control whether or not selected text gets copied to the clipboard. You can also select text in an `Entry` under program control.

- The *insertion cursor* shows where new text will be inserted. It is displayed only when the user clicks the mouse somewhere in the widget. It usually appears as a blinking vertical line inside the widget. You can customize its appearance in several ways.
- Positions within the widget's displayed text are given as an *index*. There are several ways to specify an index:
  - As normal Python indexes, starting from 0.
  - The constant `tk.END` refers to the position after the existing text.
  - The constant `tk.INSERT` refers to the current position of the insertion cursor.
  - The constant `tk.ANCHOR` refers to the first character of the selection, if there is a selection.
  - You may need to figure out which character position in the widget corresponds to a given mouse position. To simplify that process, you can use as an index a string of the form `'@`*n*`'`, where *n* is the horizontal distance in pixels between the left edge of the `Entry` widget and the mouse. Such an index will specify the character at that horizontal mouse position.

To create a new `Entry` widget in a root window or frame named *parent*:

```
    w = tk.Entry(parent, option, ...)
```

This constructor returns the new `Entry` widget. Options include:

Table 17. `Entry` widget options

`bg` or `background`	The background color inside the entry area. Default is a light gray.
`bd` or `borderwidth`	The width of the border around the entry area; see Section 5.1, "Dimensions" (p. 9). The default is two pixels.
`cursor`	The cursor used when the mouse is within the entry widget; see Section 5.8, "Cursors" (p. 13).
`disabledbackground`	The background color to be displayed when the widget is in the `tk.DISABLED` state. For option values, see `bg` above.
`disabledforeground`	The foreground color to be displayed when the widget is in the `tk.DISABLED` state. For option values, see `fg` below.
`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` or `foreground`	The color used to render the text. Default is black.
`font`	The font used for text entered in the widget by the user. See Section 5.4, "Type fonts" (p. 10).
`highlightbackground`	Color of the focus highlight when the widget does not have focus. See Section 53, "Focus: routing keyboard input" (p. 155).
`highlightcolor`	Color shown in the focus highlight when the widget has the focus.
`highlightthickness`	Thickness of the focus highlight.
`insertbackground`	By default, the insertion cursor (which shows the point within the text where new keyboard input will be inserted) is black. To get a different color of insertion cursor, set `insertbackground` to any color; see Section 5.3, "Colors" (p. 10).

`insertborderwidth`	By default, the insertion cursor is a simple rectangle. You can get the cursor with the `tk.RAISED` relief effect (see Section 5.6, "Relief styles" (p. 12)) by setting `insertborderwidth` to the dimension of the 3-d border. If you do, make sure that the `insertwidth` option is at least twice that value.
`insertofftime`	By default, the insertion cursor blinks. You can set `insertofftime` to a value in milliseconds to specify how much time the insertion cursor spends off. Default is 300. If you use `insertofftime=0`, the insertion cursor won't blink at all.
`insertontime`	Similar to `insertofftime`, this option specifies how much time the cursor spends on per blink. Default is 600 (milliseconds).
`insertwidth`	By default, the insertion cursor is 2 pixels wide. You can adjust this by setting `insertwidth` to any dimension.
`justify`	This option controls how the text is justified when the text doesn't fill the widget's width. The value can be `tk.LEFT` (the default), `tk.CENTER`, or `tk.RIGHT`.
`readonlybackground`	The background color to be displayed when the widget's `state` option is `'readonly'`.
`relief`	Selects three-dimensional shading effects around the text entry. See Section 5.6, "Relief styles" (p. 12). The default is `relief=tk.SUNKEN`.
`selectbackground`	The background color to use displaying selected text. See Section 5.3, "Colors" (p. 10).
`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`	Use this option to disable the `Entry` widget so that the user can't type anything into it. Use `state=tk.DISABLED` to disable the widget, `state=tk.NORMAL` to allow user input again. Your program can also find out whether the cursor is currently over the widget by interrogating this option; it will have the value `tk.ACTIVE` when the mouse is over it. You can also set this option to `'disabled'`, which is like the `tk.DISABLED` state, but the contents of the widget can still be selected or copied.
`takefocus`	By default, the focus will tab through entry widgets. Set this option to 0 to take the widget out of the sequence. For a discussion of focus, see Section 53, "Focus: routing keyboard input" (p. 155).
`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; see Section 52, "Control variables: the values behind the widgets" (p. 153). You can retrieve the text using `v.get()`, or set it using `v.set()`, where *v* is the associated control variable.
`validate`	You can use this option to set up the widget so that its contents are checked by a validation function at certain times. See Section 10.2, "Adding validation to an `Entry` widget" (p. 45).
`validatecommand`	A callback that validates the text of the widget. See Section 10.2, "Adding validation to an `Entry` widget" (p. 45).

`width`	The size of the entry in characters. The default is 20. For proportional fonts, the physical length of the widget will be based on the average width of a character times the value of the `width` option.
`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. Set this option to the `.set` method of the scrollbar. For more information, see Section 10.1, "Scrolling an `Entry` widget" (p. 45).

Methods on `Entry` objects include:

`.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`*.

`.scan_dragto(`*`x`*`)`
See the `scan_mark` method below.

`.scan_mark(`*`x`*`)`
Use this option to set up fast scanning of the contents of the `Entry` widget that has a scrollbar that supports horizontal scrolling.

To implement this feature, bind the mouse's button-down event to a handler that calls `scan_mark(`*`x`*`)`, where *`x`* is the current mouse *`x`* position. Then bind the `<Motion>` event to a handler that calls `scan_dragto(`*`x`*`)`, where *`x`* is the current mouse *`x`* position. The `scan_dragto` method scrolls the contents of the `Entry` widget continuously at a rate proportional to the horizontal distance between the position at the time of the `scan_mark` call and the current position.

`.select_adjust(`*`index`*`)`
This method is used to make sure that the selection includes the character at the specified *`index`*. If the selection already includes that character, nothing happens. If not, the selection is expanded from its current position (if any) to include position *`index`*.

`.select_clear()`
Clears the selection. If there isn't currently a selection, has no effect.

`.select_from(`*`index`*`)`
Sets the `tk.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.

To select all the text in an entry widget `e`, use `e.select_range(0, tk.END)`.

`.select_to(`*`index`*`)`
Selects all the text from the `tk.ANCHOR` position up to but not including the character at the given *`index`*.

`.xview(`*`index`*`)`
Same as `.xview()`. This method is useful in linking the `Entry` widget to a horizontal scrollbar. See Section 10.1, "Scrolling an `Entry` widget" (p. 45).

`.xview_moveto(`*`f`*`)`
Positions the text in the entry so that the character at position *`f`*, relative to the entire text, is positioned at the left edge of the window. The *`f`* argument must be in the range [0,1], where 0 means the left end of the text and 1 the right end.

`.xview_scroll(`*`number`*`, `*`what`*`)`
Used to scroll the entry horizontally. The *`what`* argument must be either `tk.UNITS`, to scroll by character widths, or `tk.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. For example, for an entry widget `e`, `e.xview_scroll(-1, tk.PAGES)` would move the text one "page" to the right, and `e.xview_scroll(4, tk.UNITS)` would move the text four characters to the left.

10.1. Scrolling an `Entry` widget

Making an `Entry` widget scrollable requires a little extra code on your part to adapt the `Scrollbar` widget's callback to the methods available on the `Entry` widget. Here are some code fragments illustrating the setup. First, the creation and linking of the `Entry` and `Scrollbar` widgets:

```
    self.entry = tk.Entry(self, width=10)
    self.entry.grid(row=0, sticky=tk.E+tk.W)

    self.entryScroll = tk.Scrollbar(self, orient=tk.HORIZONTAL,
        command=self.__scrollHandler)
    self.entryScroll.grid(row=1, sticky=tk.E+tk.W)
    self.entry['xscrollcommand'] = self.entryScroll.set
```

Here's the adapter function referred to above:

```
    def __scrollHandler(self, *L):
        op, howMany = L[0], L[1]

        if op == 'scroll':
            units = L[2]
            self.entry.xview_scroll(howMany, units)
        elif op == 'moveto':
            self.entry.xview_moveto(howMany)
```

10.2. Adding validation to an `Entry` widget

In some applications, you will want to check the contents of an `Entry` widget to make sure they are valid according to some rule that your application must enforce. You define what is valid by writing a callback function that checks the contents and signals whether it is valid or not.

Here is the procedure for setting up validation on a widget.

1. Write a callback function that checks the text in the `Entry` and returns `True` if the text is valid, or `False` if not. If the callback returns `False`, the user's attempt to edit the text will be refused, and the text will be unchanged.
2. Register the callback function. In this step, you will produce a Tcl wrapper around a Python function.

   Suppose your callback function is a function named `isOkay`. To register this function, use the universal widget method `.register(isOkay)`. This method returns a character string that *Tkinter* can use to call your function.
3. When you call the `Entry` constructor, use the `validatecommand` option in the `Entry` constructor to specify your callback, and use the `validate` option to specify when the callback will be called to validate the text in the callback. The values of these options are discussed in more detail below.

Here are the values of the `validate` option and what they mean.

`'focus'`
: Validate whenever the `Entry` widget gets or loses focus (see Section 53, "Focus: routing keyboard input" (p. 155)).

`'focusin'`
: Validate whenever the widget gets focus.

`'focusout'`
: Validate whenever the widget loses focus.

`'key'`
: Validate whenever any keystroke changes the widget's contents.

`'all'`
: Validate in all the above situations.

`'none'`
: Turn off validation. This is the default option value. Note that this is the string `'none'`, not the special Python value `None`.

The value of the `validatecommand` option depends on what arguments you would like your callback to receive.

- Perhaps the only thing the callback needs to know is what text currently appears in the `Entry`. If that is the case, it can use the `.get()` method of the `textvariable` associated with the widget to retrieve that text.

  In this case, all you need is the option "`validatecommand=`*`f`*", where *`f`* is the name of your callback function.
- *Tkinter* can also provide a number of items of information to the callback. If you would like to use some of these items, when you call the `Entry` constructor, use the option `validatecommand=(`*`f`*`,` s_1`, ` s_2`, ...)`, where *`f`* is the name of your callback function, and each additional s_i is a substitution code. For each substitution code that you provide, the callback will receive a positional argument containing the appropriate value.

Here are the substitution codes.

Table 18. Callback substitution codes

`'%d'`	Action code: 0 for an attempted deletion, 1 for an attempted insertion, or -1 if the callback was called for focus in, focus out, or a change to the `textvariable`.

`'%i'`	When the user attempts to insert or delete text, this argument will be the index of the beginning of the insertion or deletion. If the callback was due to focus in, focus out, or a change to the `textvariable`, the argument will be `-1`.
`'%P'`	The value that the text will have if the change is allowed.
`'%s'`	The text in the entry before the change.
`'%S'`	If the call was due to an insertion or deletion, this argument will be the text being inserted or deleted.
`'%v'`	The current value of the widget's `validate` option.
`'%V'`	The reason for this callback: one of `'focusin'`, `'focusout'`, `'key'`, or `'forced'` if the `textvariable` was changed.
`'%W'`	The name of the widget.

Here is a small example. Suppose you want your callback to receive the `'%d'` to find out why it was called; `'%i'` to find out where the insertion or deletion would occur; and `'%S'` to find out what is to be inserted or deleted. Your method might look like this:

```
    def isOkay(self, why, where, what):
        ...
```

Next you use the universal `.register()` method to wrap this function. We assume that `self` is some widget.

```
    okayCommand = self.register(isOkay)
```

To set up this callback, you would use these two options in the `Entry` constructor:

```
    self.w = Entry(self, validate='all',
         validatecommand=(okayCommand, '%d', '%i', '%S'), ...)
```

Suppose that the `Entry` currently contains the string `'abcdefg'`, and the user selects `'cde'` and then presses *Backspace*. This would result in a call `isOkay(0, 2, 'cde')`: 0 for deletion, 2 for the position before `'c'`, and `'cde'` for the string to be deleted. If `isOkay()` returns `True`, the new text will be `'abfg'`; if it returns `False`, the text will not change.

The `Entry` widget also supports an `invalidcommand` option that specifies a callback function that is called whenever the `validatecommand` returns `False`. This command may modify the text in the widget by using the `.set()` method on the widget's associated `textvariable`. Setting up this option works the same as setting up the `validatecommand`. You must use the `.register()` method to wrap your Python function; this method returns the name of the wrapped function as a string. Then you will pass as the value of the `invalidcommand` option either that string, or as the first element of a tuple containing substitution codes.

11. The `Frame` widget

A frame is basically just a container for other widgets.

- Your application's root window is basically a frame.
- Each frame has its own grid layout, so the gridding of widgets within each frame works independently.
- Frame widgets are a valuable tool in making your application modular. You can group a set of related widgets into a compound widget by putting them into a frame. Better yet, you can declare a new

class that inherits from `Frame`, adding your own interface to it. This is a good way to hide the details of interactions within a group of related widgets from the outside world.

To create a new frame widget in a root window or frame named *`parent`*:

```
w = Frame(parent, option, ...)
```

The constructor returns the new `Frame` widget. Options:

Table 19. `Frame` widget options

`bg` or `background`	The frame's background color. See Section 5.3, "Colors" (p. 10).
`bd` or `borderwidth`	Width of the frame's border. The default is 0 (no border). For permitted values, see Section 5.1, "Dimensions" (p. 9).
`cursor`	The cursor used when the mouse is within the frame widget; see Section 5.8, "Cursors" (p. 13).
`height`	The vertical dimension of the new frame. This will be ignored unless you also call `.grid_propagate(0)` on the frame; see Section 4.2, "Other grid management methods" (p. 7).
`highlightbackground`	Color of the focus highlight when the frame does not have focus. See Section 53, "Focus: routing keyboard input" (p. 155).
`highlightcolor`	Color shown in the focus highlight when the frame has the focus.
`highlightthickness`	Thickness of the focus highlight.
`padx`	Normally, a `Frame` fits tightly around its contents. To add *`N`* pixels of horizontal space inside the frame, set `padx=`*`N`*.
`pady`	Used to add vertical space inside a frame. See `padx` above.
`relief`	The default relief for a frame is `tk.FLAT`, which means the frame will blend in with its surroundings. To put a border around a frame, set its `borderwidth` to a positive value and set its relief to one of the standard relief types; see Section 5.6, "Relief styles" (p. 12).
`takefocus`	Normally, frame widgets are not visited by input focus (see Section 53, "Focus: routing keyboard input" (p. 155) for an overview of this topic). However, you can set `takefocus=1` if you want the frame to receive keyboard input. To handle such input, you will need to create bindings for keyboard events; see Section 54, "Events" (p. 157) for more on events and bindings.
`width`	The horizontal dimension of the new frame. See Section 5.1, "Dimensions" (p. 9). This value be ignored unless you also call `.grid_propagate(0)` on the frame; see Section 4.2, "Other grid management methods" (p. 7).

12. The `Label` widget

Label widgets can display one or more lines of text in the same style, or a bitmap or image. To create a label widget in a root window or frame *`parent`*:

```
w = tk.Label(parent, option, ...)
```

The constructor returns the new `Label` widget. Options include:

Table 20. `Label` widget options

`activebackground`	Background color to be displayed when the mouse is over the widget.
`activeforeground`	Foreground color to be displayed when the mouse is over the widget.
`anchor`	This options controls where the text is positioned if the widget has more space than the text needs. The default is `anchor=tk.CENTER`, which centers the text in the available space. For other values, see Section 5.5, "Anchors" (p. 12). For example, if you use `anchor=tk.NW`, the text would be positioned in the upper left-hand corner of the available space.
`bg` or `background`	The background color of the label area. See Section 5.3, "Colors" (p. 10).
`bitmap`	Set this option equal to a bitmap or image object and the label will display that graphic. See Section 5.7, "Bitmaps" (p. 12) and Section 5.9, "Images" (p. 14).
`bd` or `borderwidth`	Width of the border around the label; see Section 5.1, "Dimensions" (p. 9). The default value is two pixels.
`compound`	If you would like the `Label` widget to display both text and a graphic (either a bitmap or an image), the `compound` option specifies the relative orientation of the graphic relative to the text. Values may be any of `tk.LEFT`, `tk.RIGHT`, `tk.CENTER`, `tk.BOTTOM`, or `tk.TOP`. For example, if you specify `compound=BOTTOM`, the graphic will be displayed below the text.
`cursor`	Cursor that appears when the mouse is over this label. See Section 5.8, "Cursors" (p. 13).
`disabledforeground`	The foreground color to be displayed when the widget's `state` is `tk.DISABLED`.
`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. See Section 5.4, "Type fonts" (p. 10).
`fg` or `foreground`	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. See Section 5.3, "Colors" (p. 10).
`height`	Height of the label in *lines* (not pixels!). If this option is not set, the label will be sized to fit its contents.
`highlightbackground`	Color of the focus highlight when the widget does not have focus.
`highlightcolor`	The color of the focus highlight when the widget has focus.
`highlightthickness`	Thickness of the focus highlight.
`image`	To display a static image in the label widget, set this option to an image object. See Section 5.9, "Images" (p. 14).
`justify`	Specifies how multiple lines of text will be aligned with respect to each other: `tk.LEFT` for flush left, `tk.CENTER` for centered (the default), or `tk.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 `tk.FLAT`; for other values, see Section 5.6, "Relief styles" (p. 12).
`state`	By default, an `Entry` widget is in the `tk.NORMAL` state. Set this option to `tk.DISABLED` to make it unresponsive to mouse events. The state will be `tk.ACTIVE` when the mouse is over the widget.
`takefocus`	Normally, focus does not cycle through `Label` widgets; see Section 53, "Focus: routing keyboard input" (p. 155). If you want this widget to be visited by the focus, set `takefocus=1`.
`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. SeeSection 52, "Control variables: the values behind the widgets" (p. 153).
`underline`	You can display an underline (_) below the *n*th 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.

There are no special methods for label widgets other than the common ones (see Section 26, "Universal widget methods" (p. 97)).

13. The `LabelFrame` widget

The `LabelFrame` widget, like the `Frame` widget, is a spatial container—a rectangular area that can contain other widgets. However, unlike the `Frame` widget, the `LabelFrame` widget allows you to display a label as part of the border around the area.

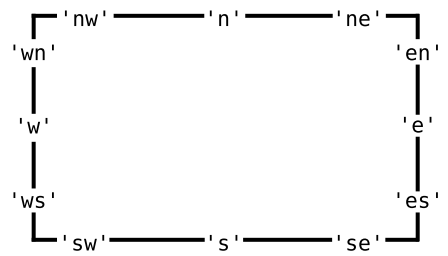
Here is an example of a `LabelFrame` widget containing two `Button` widgets. Note that the label "Important controls" interrupts the border. This widget illustrates the default `GROOVE` relief (see Section 5.6, "Relief styles" (p. 12)) and the default `'nw'` label anchor, which positions the label at the left side of the top of the frame.

To create a new `LabelFrame` widget inside a root window or frame *`parent`*:

```
    w = tk.LabelFrame(parent, option, ...)
```

This constructor returns the new `LabelFrame` widget. Options:

Table 21. `LabelFrame` widget options

`bg` or `background`	The background color to be displayed inside the widget; see Section 5.3, "Colors" (p. 10).
`bd` or `borderwidth`	Width of the border drawn around the perimeter of the widget; see Section 5.1, "Dimensions" (p. 9). The default value is two pixels.
`cursor`	Selects the cursor that appears when the mouse is over the widget; see Section 5.8, "Cursors" (p. 13).
`fg` or `foreground`	Color to be used for the label text.
`height`	The vertical dimension of the new frame. This will be ignored unless you also call `.grid_propagate(0)` on the frame; see Section 4.2, "Other grid management methods" (p. 7).
`highlightbackground`	Color of the focus highlight when the widget does not have focus.
`highlightcolor`	The color of the focus highlight when the widget has focus.
`highlightthickness`	Thickness of the focus highlight.
`labelanchor`	Use this option to specify the position of the label on the widget's border. The default position is `'nw'`, which places the label at the left end of the top border. For the nine possible label positions, refer to this diagram: 
`labelwidget`	Instead of a text label, you can use any widget as the label by passing that widget as the value of this option. If you supply both `labelwidget` and `text` options, the `text` option is ignored.
`padx`	Use this option to add additional padding inside the left and right sides of the widget's frame. The value is in pixels.
`pady`	Use this option to add additional padding inside the top and bottom of the widget's frame. The value is in pixels.
`relief`	This option controls the appearance of the border around the outside of the widget. The default style is `tk.GROOVE`; for other values, see Section 5.6, "Relief styles" (p. 12).
`takefocus`	Normally, the widget will not receive focus; supply a `True` value to this option to make the widget part of the focus traversal sequence. For more information, see Section 53, "Focus: routing keyboard input" (p. 155).
`text`	Text of the label.
`width`	The horizontal dimension of the new frame. This will be ignored unless you also call `.grid_propagate(0)` on the frame; see Section 4.2, "Other grid management methods" (p. 7).

14. The `Listbox` widget

The purpose of a listbox widget is to display a set of lines of text. Generally they are intended to allow the user to select one or more items from a list. All the lines of text use the same font. If you need something more like a text editor, see Section 24, "The `Text` widget" (p. 82).

To create a new listbox widget inside a root window or frame *`parent`*:

```
    w = tk.Listbox(parent, option, ...)
```

This constructor returns the new `Listbox` widget. Options:

Table 22. `Listbox` widget options

`activestyle`	This option specifies the appearance of the active line. It may have any of these values: **`'underline'`** The active line is underlined. This is the default option. **`'dotbox'`** The active line is enclosed in a dotted line on all four sides. **`'none'`** The active line is given no special appearance.
`bg` or `background`	The background color in the listbox.
`bd` or `borderwidth`	The width of the border around the listbox. Default is two pixels. For possible values, see Section 5.1, "Dimensions" (p. 9).
`cursor`	The cursor that appears when the mouse is over the listbox. See Section 5.8, "Cursors" (p. 13).
`disabledforeground`	The color of the text in the listbox when its `state` is `tk.DISABLED`.
`exportselection`	By default, the user may select text with the mouse, and the selected text will be exported to the clipboard. To disable this behavior, use `exportselection=0`.
`font`	The font used for the text in the listbox. See Section 5.4, "Type fonts" (p. 10).
`fg` or `foreground`	The color used for the text in the listbox. See Section 5.3, "Colors" (p. 10).
`height`	Number of *lines* (not pixels!) shown in the listbox. Default is 10.
`highlightbackground`	Color of the focus highlight when the widget does not have focus. See Section 53, "Focus: routing keyboard input" (p. 155).
`highlightcolor`	Color shown in the focus highlight when the widget has the focus.
`highlightthickness`	Thickness of the focus highlight.
`listvariable`	A `StringVar` that is connected to the complete list of values in the listbox (see Section 52, "Control variables: the values behind the widgets" (p. 153). If you call the `.get()` method of the `listvariable`, you will get back a *string* of the form `"('`v_0`', '`v_1`', ...)"`, where each v_i is the contents of one line of the listbox. To change the entire set of lines in the listbox at once, call `.set(`*`s`*`)` on the `listvariable`, where *`s`* is a string containing the line values with spaces between them.

	For example, if `listCon` is a `StringVar` associated with a listbox's `listvariable` option, this call would set the listbox to contain three lines:
	`listCon.set('ant bee cicada')`
	This call would return the string `"('ant', 'bee', 'cicada')"`:
	`listCon.get()`

`relief`	Selects three-dimensional border shading effects. The default is `tk.SUNKEN`. For other values, see Section 5.6, "Relief styles" (p. 12).
`selectbackground`	The background color to use displaying selected text.
`selectborderwidth`	The width of the border to use around selected text. The default is that the selected item is shown in a solid block of color `selectbackground`; if you increase the `selectborderwidth`, the entries are moved farther apart and the selected entry shows `tk.RAISED` relief (see Section 5.6, "Relief styles" (p. 12)).
`selectforeground`	The foreground color to use displaying selected text.
`selectmode`	Determines how many items can be selected, and how mouse drags affect the selection: • `tk.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. • `tk.SINGLE`: You can only select one line, and you can't drag the mouse—wherever you click button 1, that line is selected. • `tk.MULTIPLE`: You can select any number of lines at once. Clicking on any line toggles whether or not it is selected. • `tk.EXTENDED`: You can select any adjacent group of lines at once by clicking on the first line and dragging to the last line.
`state`	By default, a listbox is in the `tk.NORMAL` state. To make the listbox unresponsive to mouse events, set this option to `tk.DISABLED`.
`takefocus`	Normally, the focus will tab through listbox widgets. Set this option to 0 to take the widget out of the sequence. See Section 53, "Focus: routing keyboard input" (p. 155).
`width`	The width of the widget in *characters* (not pixels!). The width is based on an average character, so some strings of this length in proportional fonts may not fit. 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. Set this option to the `.set` method of the scrollbar. See Section 14.1, "Scrolling a `Listbox` widget" (p. 56) for more on scrollable listbox widgets.
`yscrollcommand`	If you want to allow the user to scroll the listbox vertically, you can link your listbox widget to a vertical scrollbar. Set this option to the `.set` method of the scrollbar. See Section 14.1, "Scrolling a `Listbox` widget" (p. 56).

A special set of index forms is used for many of the methods on listbox objects:

- If you specify an index as an integer, it refers to the line in the listbox with that index, counting from 0.
- Index `tk.END` refers to the last line in the listbox.
- Index `tk.ACTIVE` refers to the selected line. If the listbox allows multiple selections, it refers to the line that was last selected.
- An index string of the form `'@x,y'` refers to the line closest to coordinate (*x*,*y*) relative to the widget's upper left corner.

Methods on `Listbox` objects include:

`.activate(`*`index`*`)`
Selects the line specifies by the given *`index`*.

`.bbox(`*`index`*`)`
Returns the bounding box of the line specified by *`index`* as a 4-tuple `(`*`xoffset`*`, `*`yoffset`*`, `*`width`*`, `*`height`*`)`, where the upper left pixel of the box is at `(`*`xoffset`*`, `*`yoffset`*`)` and the *`width`* and *`height`* are given in pixels. The returned *`width`* value includes only the part of the line occupied by text.

If the line specified by the *`index`* argument is not visible, this method returns `None`. If it is partially visible, the returned bounding box may extend outside the visible area.

`.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`*], **inclusive** (contrary to the usual Python idiom, where deletion stops short of the last index), counting from 0. 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 `tk.END` as the first argument if you want to add new lines to the end of the listbox.

`.itemcget(`*`index`*`, `*`option`*`)`
Retrieves one of the option values for a specific line in the listbox. For option values, see `itemconfig` below. If the given option has not been set for the given line, the returned value will be an empty string.

`.itemconfig(`*`index`*`, `*`option`*`=`*`value`*`, ...)`
Change a configuration option for the line specified by *`index`*. Option names include:

`background`
The background color of the given line.

`foreground`
The text color of the given line.

selectbackground
The background color of the given line when it is selected.

selectforeground
The text color of the given line when it is selected.

.nearest(*y*)
Return the index of the visible line closest to the y-coordinate *y* relative to the listbox widget.

.scan_dragto(*x*, *y*)
See `scan_mark` below.

.scan_mark(*x*, *y*)
Use this method to implement scanning—fast steady scrolling—of a listbox. To get this feature, bind some mouse button event to a handler that calls `scan_mark` with the current mouse position. Then bind the `<Motion>` event to a handler that calls `scan_dragto` with the current mouse position, and the listbox will be scrolled at a rate proportional to the distance between the position recorded by `scan_mark` and the current position.

.see(*index*)
Adjust the position of the listbox so that the line referred to by *index* is visible.

.selection_anchor(*index*)
Place the "selection anchor" on the line selected by the *index* argument. Once this anchor has been placed, you can refer to it with the special index form `tk.ANCHOR`.

For example, for a listbox named `lbox`, this sequence would select lines 3, 4, and 5:

```
lbox.selection_anchor(3)
lbox.selection_set(tk.ANCHOR,5)
```

.selection_clear(*first*, *last*=None)
Unselects all of the lines between indices *first* and *last*, inclusive. If the second argument is omitted, unselects the line with index *first*.

.selection_includes(*index*)
Returns 1 if the line with the given *index* is selected, else returns 0.

.selection_set(*first*, *last*=None)
Selects all of the lines between indices *first* and *last*, inclusive. If the second argument is omitted, selects the line with index *first*.

.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. See Section 14.1, "Scrolling a `Listbox` widget" (p. 56).

.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 `tk.UNITS` to scroll by characters, or `tk.PAGES` to scroll by pages, that is, by the width of the listbox. The *number* argument tells how many to scroll; negative values move the text to the right within the listbox, positive values leftward.

.yview()
To make the listbox vertically scrollable, set the *command* option of the associated vertical scrollbar to this method. See Section 14.1, "Scrolling a Listbox widget" (p. 56).

.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 tk.UNITS to scroll by lines, or tk.PAGES to scroll by pages, that is, by the height of the listbox. The *number* argument tells how many to scroll; negative values move the text downward inside the listbox, and positive values move the text up.

14.1. Scrolling a Listbox widget

Here is a code fragment illustrating the creation and linking of a listbox to both a horizontal and a vertical scrollbar.

```
    self.yScroll = tk.Scrollbar(self, orient=tk.VERTICAL)
    self.yScroll.grid(row=0, column=1, sticky=tk.N+tk.S)

    self.xScroll = tk.Scrollbar(self, orient=tk.HORIZONTAL)
    self.xScroll.grid(row=1, column=0, sticky=tk.E+tk.W)

    self.listbox = tk.Listbox(self,
         xscrollcommand=self.xScroll.set,
         yscrollcommand=self.yScroll.set)
    self.listbox.grid(row=0, column=0, sticky=tk.N+tk.S+tk.E+tk.W)
    self.xScroll['command'] = self.listbox.xview
    self.yScroll['command'] = self.listbox.yview
```

15. The Menu widget

"Drop-down" menus are a popular way to present the user with a number of choices, yet take up minimal space on the face of the application when the user is not making a choice.

- A *menubutton* is the part that always appears on the application.
- A *menu* is the list of choices that appears only after the user clicks on the menubutton.
- To select a choice, the user can drag the mouse from the menubutton down onto one of the choices. Alternatively, they can click and release the menubutton: the choices will appear and stay until the user clicks one of them.
- The Unix version of *Tkinter* (at least) supports "tear-off menus." If you as the designer wish it, a dotted line will appear above the choices. The user can click on this line to "tear off" the menu: a new, separate, independent window appears containing the choices.

Refer to Section 16, "The Menubutton widget" (p. 61), below, to see how to create a menubutton and connect it to a menu widget. First let's look at the Menu widget, which displays the list of choices.

The choices displayed on a menu may be any of these things:

- A simple command: a text string (or image) that the user can select to perform some operation.

- A *cascade*: a text string or image that the user can select to show another whole menu of choices.
- A checkbutton (see Section 9, "The `Checkbutton` widget" (p. 38)).
- A group of radiobuttons (see Section 20, "The `Radiobutton` widget" (p. 68)).

To create a menu widget, you must first have created a `Menubutton`, which we will call `mb`:

```
    w = tk.Menu(mb, option, ...)
```

This constructor returns the new `Menu` widget. Options include:

Table 23. `Menu` widget options

`activebackground`	The background color that will appear on a choice when it is under the mouse. See Section 5.3, "Colors" (p. 10).
`activeborderwidth`	Specifies the width of a border drawn around a choice when it is under the mouse. Default is 1 pixel. For possible values, see Section 5.1, "Dimensions" (p. 9).
`activeforeground`	The foreground color that will appear on a choice when it is under the mouse.
`bg` or `background`	The background color for choices not under the mouse.
`bd` or `borderwidth`	The width of the border around all the choices; see Section 5.1, "Dimensions" (p. 9). The default is one pixel.
`cursor`	The cursor that appears when the mouse is over the choices, but only when the menu has been torn off. See Section 5.8, "Cursors" (p. 13).
`disabledforeground`	The color of the text for items whose `state` is `tk.DISABLED`.
`font`	The default font for textual choices. See Section 5.4, "Type fonts" (p. 10).
`fg` or `foreground`	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=tk.RAISED`. For other options, see Section 5.6, "Relief styles" (p. 12).
`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.
`tearoffcommand`	If you would like your program to be notified when the user clicks on the tear-off entry in a menu, set this option to your procedure. It will be called with two arguments: the window ID of the parent window, and the window ID of the new tear-off menu's root window.
`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.

These methods are available on `Menu` objects. The ones that create choices on the menu have their own particular options; see Section 15.1, "`Menu` item creation (`coption`) options" (p. 59).

.add(*kind*, *coption*, ...)
Add a new element of the given *kind* as the next available choice in this menu. The *kind* argument may be any of 'cascade', 'checkbutton', 'command', 'radiobutton', or 'separator'. Depending on the *kind* argument, this method is equivalent to .add_cascade(), .add_checkbutton(), and so on; refer to those methods below for details.

.add_cascade(*coption*, ...)
Add a new cascade element as the next available choice in this menu. Use the menu option in this call to connect the cascade to the next level's menu, an object of type Menu.

.add_checkbutton(*coption*, ...)
Add a new checkbutton as the next available choice in self. The options allow you to set up the checkbutton much the same way as you would set up a Checkbutton object; see Section 15.1, "Menu item creation (coption) options" (p. 59).

.add_command(*coption*, ...)
Add a new command as the next available choice in self. Use the label, bitmap, or image option to place text or an image on the menu; use the command option to connect this choice to a procedure that will be called when this choice is picked.

.add_radiobutton(*coption*, ...)
Add a new radiobutton as the next available choice in self. The options allow you to set up the radiobutton in much the same way as you would set up a Radiobutton object; see Section 20, "The Radiobutton widget" (p. 68).

.add_separator()
Add a separator after the last currently defined option. This is just a ruled horizontal line you can use to set off groups of choices. Separators are counted as choices, so if you already have three choices, and you add a separator, the separator will occupy position 3 (counting from 0).

.delete(*index1*, *index2*=None)
This method deletes the choices numbered from *index1* through *index2*, inclusive. To delete one choice, omit the *index2* argument. You can't use this method to delete a tear-off choice, but you can do that by setting the menu object's tearoff option to 0.

.entrycget(*index*, *coption*)
To retrieve the current value of some coption for a choice, call this method with *index* set to the index of that choice and *coption* set to the name of the desired option.

.entryconfigure(*index*, *coption*, ...)
To change the current value of some *coption* for a choice, call this method with *index* set to the index of that choice and one or more *coption=value* arguments.

.index(*i*)
Returns the position of the choice specified by index *i*. For example, you can use .index(tk.END) to find the index of the last choice (or None if there are no choices).

.insert_cascade(*index*, *coption*, ...)
Inserts a new cascade at the position given by *index*, counting from 0. Any choices after that position move down one. The options are the same as for .add_cascade(), above.

.insert_checkbutton(*index*, *coption*, ...)
Insert a new checkbutton at the position specified by *index*. Options are the same as for .add_checkbutton(), above.

.insert_command(*index*, *coption*, ...)
Insert a new command at position *index*. Options are the same as for .add_command(), above.

.insert_radiobutton(*index*, *coption*, ...)
Insert a new radiobutton at position *index*. Options are the same as for .add_radiobutton(), above.

.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.

.post(*x*, *y*)
Display this menu at position (*x*, *y*) relative to the root window.

.type(*index*)
Returns the type of the choice specified by *index*: either tk.CASCADE, tk.CHECKBUTTON, tk.COMMAND, tk.RADIOBUTTON, tk.SEPARATOR, or tk.TEAROFF.

.yposition(*n*)
For the *n*th menu choice, return the vertical offset in pixels relative to the menu's top. The purpose of this method is to allow you to place a popup menu precisely relative to the current mouse position.

15.1. Menu item creation (coption) options

Wherever the menu methods described above allow a coption, you may apply a value to any of the option names below by using the option name as a keyword argument with the desired value. For example, to make a command's text appear with red letters, use "foreground='red'" as an option to the add_command method call.

Table 24. Menu item coption values

accelerator	To display an "accelerator" keystroke combination on the right side of a menu choice, use the option "accelerator=*s*" where *s* is a string containing the characters to be displayed. For example, to indicate that a command has *Control-X* as its accelerator, use the option "accelerator='^X'". Note that this option does not actually implement the accelerator; use a keystroke binding to do that.
activeback-ground	The background color used for choices when they are under the mouse.
activefore-ground	The foreground color used for choices when they are under the mouse.
background	The background color used for choices when they are *not* under the mouse. Note that this *cannot* be abbreviated as bg.
bitmap	Display a bitmap for this choice; see Section 5.7, "Bitmaps" (p. 12).
columnbreak	Normally all the choices are displayed in one long column. If you set column-break=1, this choice will start a new column to the right of the one containing the previous choice.
columnbreak	Use option "columnbreak=True" to start a new column of choices with this choice.
command	A procedure to be called when this choice is activated.
compound	If you want to display both text and a graphic (either a bitmap or an image) on a menu choice, use this coption to specify the location of the graphic relative to the text. Values may be any of tk.LEFT, tk.RIGHT, tk.TOP, tk.BOTTOM, tk.CENTER,

	or `tk.NONE`. For example, a value of "`compound=tk.TOP`" would position the graphic above the text.
`font`	The font used to render the `label` text. See Section 5.4, "Type fonts" (p. 10)
`foreground`	The foreground color used for choices when they are *not* under the mouse. Note that this *cannot* be abbreviated as `fg`.
`hidemargin`	By default, a small margin separates adjacent choices in a menu. Use the `coption` "`hidemargin=True`" to suppress this margin. For example, if your choices are color swatches on a palette, this option will make the swatches touch without any other intervening color.
`image`	Display an image for this choice; see Section 5.9, "Images" (p. 14).
`label`	The text string to appear for this choice.
`menu`	This option is used only for cascade choices. Set it to a `Menu` object that displays the next level of choices.
`offvalue`	Normally, the control variable for a checkbutton is set to `0` when the checkbutton is off. You can change the off value by setting this option to the desired value. See Section 52, "Control variables: the values behind the widgets" (p. 153).
`onvalue`	Normally, the control variable for a checkbutton is set to `1` when the checkbutton is on. You can change the on value by setting this option to the desired value.
`selectcolor`	Normally, the color displayed in a set checkbutton or radiobutton is red. Change that color by setting this option to the color you want; see Section 5.3, "Colors" (p. 10).
`selectimage`	If you are using the `image` option to display a graphic instead of text on a menu radiobutton or checkbutton, if you use `selectimage=`*`I`*, image *`I`* will be displayed when the item is selected.
`state`	Normally, all choices react to mouse clicks, but you can set `state=tk.DISABLED` to gray it out and make it unresponsive. This `coption` will be `tk.ACTIVE` when the mouse is over the choice.
`underline`	Normally none of the letters in the `label` are underlined. Set this option to the index of a letter to underline that letter.
`value`	Specifies the value of the associated control variable (see Section 52, "Control variables: the values behind the widgets" (p. 153)) for a radiobutton. This can be an integer if the control variable is an `IntVar`, or a string if the control variable is a `StringVar`.
`variable`	For checkbuttons or radiobuttons, this option should be set to the control variable associated with the checkbutton or group of radiobuttons. See Section 52, "Control variables: the values behind the widgets" (p. 153).

15.2. Top-level menus

Especially under MacOS, it is sometimes desirable to create menus that are shown as part of the top-level window. To do this, follow these steps.

1. Using any widget *`W`*, obtain the top-level window by using the *`W`*`.winfo_toplevel()` method.
2. Create a `Menu` widget, using the top-level window as the first argument.
3. Items added to this `Menu` widget will be displayed across the top of the application.

Here is a brief example. Assume that `self` is the application instance, an instance of a class that inherits from `Frame`. This code would create a top-level menu choice named "*Help*" with one choice named "*About*" that calls a handler named `self.__aboutHandler`:

```
    top = self.winfo_toplevel()
    self.menuBar = tk.Menu(top)
    top['menu'] = self.menuBar

    self.subMenu = tk.Menu(self.menuBar)
    self.menuBar.add_cascade(label='Help', menu=self.subMenu)
    self.subMenu.add_command(label='About', command=self.__aboutHandler)
```

There is some variation in behavior depending on your platform.

- Under Windows or Unix systems, the top-level menu choices appear at the top of your application's main window.
- Under MacOS X, the top-level menu choices appear at the top of the screen when the application is active, right where Mac users expect to see them.

  You must use the `.add_cascade()` method for all the items you want on the top menu bar. Calls to `.add_checkbutton()`, `.add_command()`, or `.add_radiobutton()` will be ignored.

16. The `Menubutton` widget

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 (see above) that can display the choices for that menubutton when the user clicks on it.

To create a menubutton within a root window or frame *`parent`*:

```
    w = tk.Menubutton(parent, option, ...)
```

The constructor returns the new `Menubutton` widget. Options:

Table 25. `Menubutton` widget options

`activebackground`	The background color when the mouse is over the menubutton. See Section 5.3, "Colors" (p. 10).
`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=tk.CENTER`, which centers the text. For other options, see Section 5.5, "Anchors" (p. 12). For example, if you use `anchor=tk.W`, the text would be centered against the left side of the widget.
`bg` or `background`	The background color when the mouse is not over the menubutton.
`bitmap`	To display a bitmap on the menubutton, set this option to a bitmap name; see Section 5.7, "Bitmaps" (p. 12).
`bd` or `borderwidth`	Width of the border around the menubutton. Default is two pixels. For possible values, see Section 5.1, "Dimensions" (p. 9).
`compound`	If you specify both text and a graphic (either a bitmap or an image), this option specifies where the graphic appears relative to the text. Possible values are `tk.NONE` (the default value), `tk.TOP`, `tk.BOTTOM`, `tk.LEFT`,

	`tk.RIGHT`, and `tk.CENTER`. For example, `compound=tk.RIGHT` would position the graphic to the right of the text. If you specify `compound=tk.NONE`, the graphic is displayed but the `text` (if any) is not.
`cursor`	The cursor that appears when the mouse is over this menubutton. See Section 5.8, "Cursors" (p. 13).
`direction`	Normally, the menu will appear below the menubutton. Set `direction=tk.LEFT` to display the menu to the left of the button; use `direction=tk.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` or `foreground`	The foreground color when the mouse is not over the menubutton.
`font`	Specifies the font used to display the `text`; see Section 5.4, "Type fonts" (p. 10).
`height`	The height of the menubutton in *lines* of text (not pixels!). The default is to fit the menubutton's size to its contents.
`highlightbackground`	Color of the focus highlight when the widget does not have focus. See Section 53, "Focus: routing keyboard input" (p. 155).
`highlightcolor`	Color shown in the focus highlight when the widget has the focus.
`highlightthickness`	Thickness of the focus highlight.
`image`	To display an image on this menubutton, set this option to the image object. See Section 5.9, "Images" (p. 14).
`justify`	This option controls where the text is located when the text doesn't fill the menubutton: use `justify=tk.LEFT` to left-justify the text (this is the default); use `justify=tk.CENTER` to center it, or `justify=tk.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. See below for an example showing how to associate a menubutton and menu.
`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`	Normally, menubuttons will have `tk.RAISED` appearance. For other 3-d effects, see Section 5.6, "Relief styles" (p. 12).
`state`	Normally, menubuttons respond to the mouse. Set `state=tk.DISABLED` to gray out the menubutton and make it unresponsive.
`takefocus`	Normally, menubuttons do not take keyboard focus (see Section 53, "Focus: routing keyboard input" (p. 155)). Use `takefocus=True` to add the menubutton to the focus traversal order.
`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. See Section 52, "Control variables: the values behind the widgets" (p. 153).

`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`	Width of the menubutton in *characters* (not pixels!). If this option is not set, the label will be sized to fit its contents.
`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.

Here is a brief example showing the creation of a menubutton and its associated menu with two checkboxes:

```
    self.mb = tk.Menubutton(self, text='condiments',
                            relief=RAISED)
    self.mb.grid()

    self.mb.menu = tk.Menu(self.mb, tearoff=0)
    self.mb['menu'] = self.mb.menu

    self.mayoVar  = tk.IntVar()
    self.ketchVar = tk.IntVar()
    self.mb.menu.add_checkbutton(label='mayo',
        variable=self.mayoVar)
    self.mb.menu.add_checkbutton(label='ketchup',
        variable=self.ketchVar)
```

This example creates a menubutton labeled `condiments`. When clicked, two checkbuttons labeled `mayo` and `ketchup` will drop down.

17. The `Message` widget

This widget is similar to the `Label` widget (see Section 12, "The `Label` widget" (p. 48)), but it is intended for displaying messages over multiple lines. All the text will be displayed in the same font; if you need to display text with more than one font, see Section 24, "The `Text` widget" (p. 82).

To create a new `Message` widget as the child of a root window or frame named *`parent`*:

```
    w = tk.Message(parent, option, ...)
```

This constructor returns the new `Message` widget. Options may be any of these:

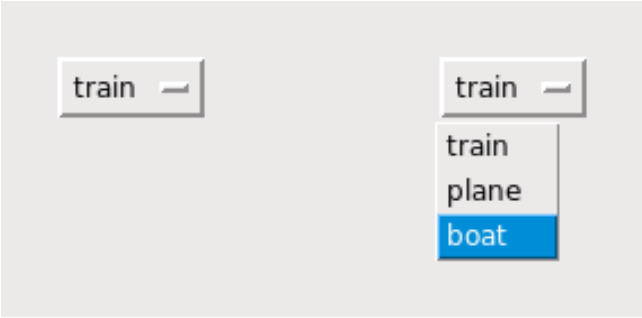
Table 26. `Message` widget options

`aspect`	Use this option to specify the ratio of width to height as a percentage. For example, `aspect=100` would give you a text message fit into a square; with `aspect=200`, the text area would be twice as wide as high. The default value is 150, that is, the text will be fit into a box 50% wider than it is high.
`bg` or `background`	The background color behind the text; see Section 5.3, "Colors" (p. 10).
`bd` or `borderwidth`	Width of the border around the widget; see Section 5.1, "Dimensions" (p. 9). The default is two pixels. This option is visible only when the `relief` option is not `tk.FLAT`.
`cursor`	Specifies the cursor that appears when the mouse is over the widget; see Section 5.8, "Cursors" (p. 13).

`font`	Specifies the font used to display the text in the widget; see Section 5.4, "Type fonts" (p. 10).
`fg` or `foreground`	Specifies the text color; see Section 5.3, "Colors" (p. 10).
`highlightbackground`	Color of the focus highlight when the widget does not have focus. See Section 53, "Focus: routing keyboard input" (p. 155).
`highlightcolor`	Color shown in the focus highlight when the widget has the focus.
`highlightthickness`	Thickness of the focus highlight.
`justify`	Use this option to specify how multiple lines of text are aligned. Use `justify=tk.LEFT` to get a straight left margin; `justify=tk.CENTER` to center each line; and `justify=tk.RIGHT` to get a straight right margin.
`padx`	Use this option to add extra space inside the widget to the left and right of the text. The value is in pixels.
`pady`	Use this option to add extra space inside the widget above and below the text. The value is in pixels.
`relief`	This option specifies the appearance of the border around the outside of the widget; see Section 5.6, "Relief styles" (p. 12). The default style is `tk.FLAT`.
`takefocus`	Normally, a `Message` widget will not acquire focus (see Section 53, "Focus: routing keyboard input" (p. 155)). Use `takefocus=True` to add the widget to the focus traversal list.
`text`	The value of this option is the text to be displayed inside the widget.
`textvariable`	If you would like to be able to change the message under program control, associate this option with a `StringVar` instance (see Section 52, "Control variables: the values behind the widgets" (p. 153)). The value of this variable is the text to be displayed. If you specify both `text` and `textvariable` options, the `text` option is ignored.
`width`	Use this option to specify the width of the text area in the widget, in pixels. The default width depends on the displayed text and the value of the `aspect` option.

18. The `OptionMenu` widget

The purpose of this widget is to offer a fixed set of choices to the user in a drop-down menu.



The illustrations above shows an `OptionMenu` in two states. The left-hand example shows the widget in its initial form. The right-hand example shows how it looks when the mouse has clicked on it and dragged down to the `'boat'` choice.

To create a new `OptionMenu` widget as the child of a root window or frame named *`parent`*:

```
w = tk.OptionMenu(parent, variable, choice1, choice2, ...)
```

This constructor returns the new `OptionMenu` widget. The *`variable`* is a `StringVar` instance (see Section 52, "Control variables: the values behind the widgets" (p. 153)) that is associated with the widget, and the remaining arguments are the choices to be displayed in the widget as strings.

The illustration above was created with this code snippet:

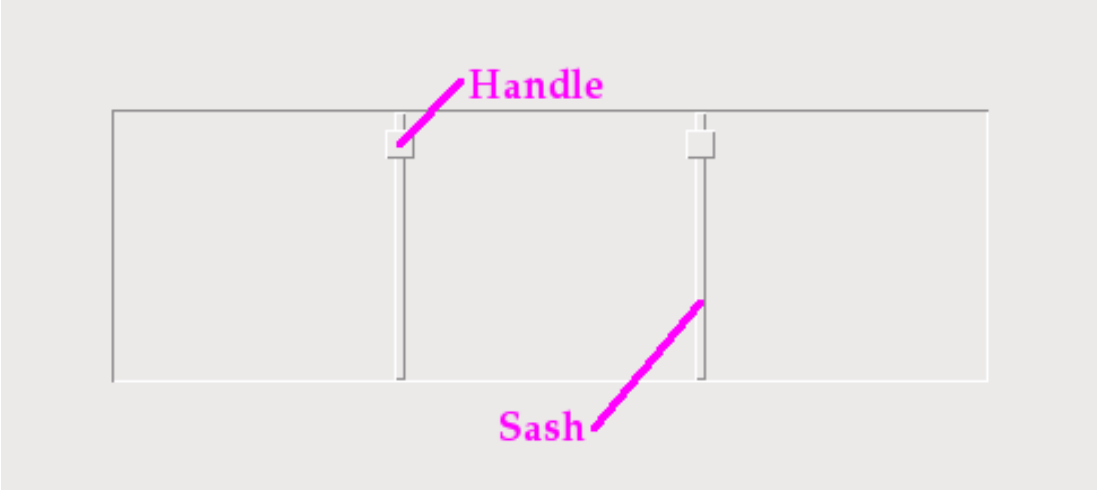
```
optionList = ('train', 'plane', 'boat')
self.v = tk.StringVar()
self.v.set(optionList[0])
self.om = tk.OptionMenu(self, self.v, *optionList)
```

To find out which choice is currently selected in an `OptionMenu` widget, the `.get()` method on the associated control variable will return that choice as a string.

19. The `PanedWindow` widget

The purpose of the `PanedWindow` widget is to give the application's user some control over how space is divided up within the application.

A `PanedWindow` is somewhat like a `Frame`: it is a container for *child widgets*. Each `PanedWindow` widget contains a horizontal or vertical stack of child widgets. Using the mouse, the user can drag the boundaries between the child widgets back and forth.



- You may choose to display *handles* within the widget. A handle is a small square that the user can drag with the mouse.
- You may choose to make *sashes* visible. A sash is a bar placed between the child widgets.
- A *pane* is the area occupied by one child widget.

To create a new `PanedWindow` widget as the child of a root window or frame named *`parent`*:

```
w = tk.PanedWindow(parent, option, ...)
```

This constructor returns the new `PanedWindow` widget. Here are the options:

Table 27. `PanedWindow` widget options

`bg` or `background`	The background color displayed behind the child widgets; see Section 5.3, "Colors" (p. 10).
`bd` or `borderwidth`	Width of the border around the outside of the widget; see Section 5.1, "Dimensions" (p. 9). The default is two pixels.
`cursor`	The cursor to be displayed when the mouse is over the widget; see Section 5.8, "Cursors" (p. 13).
`handlepad`	Use this option to specify the distance between the handle and the end of the sash. For `orient=tk.VERTICAL`, this is the distance between the left end of the sash and the handle; for `orient=tk.HORIZONTAL`, it is the distance between the top of the sash and the handle. The default value is eight pixels; for other values, see Section 5.1, "Dimensions" (p. 9).
`handlesize`	Use this option to specify the size of the handle, which is always a square; see Section 5.1, "Dimensions" (p. 9). The default value is eight pixels.
`height`	Specifies the height of the widget; see Section 5.1, "Dimensions" (p. 9). If you don't specify this option, the height is determined by the height of the child widgets.
`opaqueresize`	This option controls how a resizing operation works. For the default value, `opaqueresize=True`, the resizing is done continuously as the sash is dragged. If this option is set to `False`, the sash (and adjacent child widgets) stays put until the user releases the mouse button, and then it jumps to the new position.
`orient`	To stack child widgets side by side, use `orient=tk.HORIZONTAL`. To stack them top to bottom, use `orient=tk.VERTICAL`.
`relief`	Selects the relief style of the border around the widget; see Section 5.6, "Relief styles" (p. 12). The default is `tk.FLAT`.
`sashpad`	Use this option to allocate extra space on either side of each sash. The default is zero; for other values, see Section 5.1, "Dimensions" (p. 9).
`sashrelief`	This option specifies the relief style used to render the sashes; see Section 5.6, "Relief styles" (p. 12). The default style is `tk.FLAT`.
`sashwidth`	Specifies the width of the sash; see Section 5.1, "Dimensions" (p. 9). The default width is two pixels.
`showhandle`	Use `showhandle=True` to display the handles. For the default value, `False`, the user can still use the mouse to move the sashes. The handle is simply a visual cue.
`width`	Width of the widget; see Section 5.1, "Dimensions" (p. 9). If you don't specify a value, the width will be determined by the sizes of the child widgets.

To add child widgets to a `PanedWindow`, create the child widgets as children of the parent `PanedWindow`, but rather than using the `.grid()` method to register them, use the `.add()` method on the `PanedWindow`.

Here are the methods on `PanedWindow` widgets.

`.add(`*`child`*`[, `*`option`*`=`*`value`*`] ...)`

Use this method to add the given *`child`* widget as the next child of this `PanedWindow`. First create the *`child`* widget with the `PanedWindow` as its parent widget, but do not call the `.grid()` method to register it. Then call `.add(`*`child`*`)` and the child will appear inside the `PanedWindow` in the next available position.

Associated with each child is a set of configuration options that control its position and appearance. See Section 19.1, "`PanedWindow` child configuration options" (p. 67). You can supply these configuration options as keyword arguments to the `.add()` method. You can also set or change their values anytime with the `.paneconfig()` method, or retrieve the current value of any of these options using the `.panecget()` method; these methods are described below.

`.forget(`*`child`*`)`
Removes a child widget.

`.identify(`*`x, y`*`)`
For a given location `(`*`x, y`*`)` in window coordinates, this method returns a value that describes the feature at that location.

- If the feature is a child window, the method returns an empty string.
- If the feature is a sash, the method returns a tuple `(`*`n`*`, 'sash')` where *n* is 0 for the first sash, 1 for the second, and so on.
- If the feature is a handle, the method returns a tuple `(`*`n`*`, 'handle')` where *n* is 0 for the first handle, 1 for the second, and so on.

`.panecget(`*`child, option`*`)`
This method retrieves the value of a child widget configuration option, where *`child`* is the child widget and `option` is the name of the option as a string. For the list of child widget configuration options, see Section 19.1, "`PanedWindow` child configuration options" (p. 67).

`.paneconfig(`*`child, option=value`*`, ...)`
Use this method to configure options for child widgets. The options are described in Section 19.1, "`PanedWindow` child configuration options" (p. 67).

`.panes()`
This method returns a list of the child widgets, in order from left to right (for `orient=tk.HORIZONTAL`) or top to bottom (for `orient=tk.VERTICAL`).

`.remove(`*`child`*`)`
Removes the given *`child`*; this is the same action as the `.forget()` method.

`.sash_coord(`*`index`*`)`
This method returns the location of a sash. The *`index`* argument selects the sash: 0 for the sash between the first two children, 1 for the sash between the second and third child, and so forth. The result is a tuple `(`*`x, y`*`)` containing the coordinates of the upper left corner of the sash.

`.sash_place(`*`index, x, y`*`)`
Use this method to reposition the sash selected by *`index`* (0 for the first sash, and so on). The *`x`* and *`y`* coordinates specify the desired new position of the upper left corner of the sash. *Tkinter* ignores the coordinate orthogonal to the orientation of the widget: use the *`x`* value to reposition the sash for `orient=tk.HORIZONTAL`, and use the *`y`* coordinate to move the sash for option `orient=tk.VERTICAL`.

19.1. `PanedWindow` child configuration options

Each child of a `PanedWindow` has a set of configuration options that control its position and appearance. These options can be provided when a child is added with the `.add()` method, or set with the `.paneconfig()` method, or queried with the `.panecget()` methods described above.

Table 28. `PanedWindow` child widget options

`after`	Normally, when you `.add()` a new child to a `PanedWindow`, the new child is added after any existing child widgets. You may instead use the `after=w` option to insert the new widget at a position just after an existing child widget *w*.
`before`	When used as option `before=w` in a call to the `.add()` method, places the new widget at a position just before an existing child widget *w*.
`height`	This option specifies the desired height of the child widget; see Section 5.1, "Dimensions" (p. 9).
`minsize`	Use this option to specify a minimum size for the child widget in the direction of the `PanedWindow`'s orientation. For `orient=tk.HORIZONTAL`, this is the minimum width; for `orient=tk.VERTICAL`, it is the minimum height. For permissible values, see Section 5.1, "Dimensions" (p. 9).
`padx`	The amount of extra space to be added to the left and right of the child widget; see Section 5.1, "Dimensions" (p. 9).
`pady`	The amount of extra space to be added above and below the child widget; see Section 5.1, "Dimensions" (p. 9).
`sticky`	This option functions like the `sticky` argument to the `.grid()` method; see Section 4.1, "The `.grid()` method" (p. 6). It specifies how to position a child widget if the pane is larger than the widget. For example, `sticky=tk.NW` would position the widget in the upper left ("northwest") corner of the pane.
`width`	Desired width of the child widget; see Section 5.1, "Dimensions" (p. 9).

20. The `Radiobutton` widget

Radiobuttons are sets of related widgets that allow the user to select only one of a set of choices. Each radiobutton consists of two parts, the *indicator* and the *label*:

- The indicator is the diamond-shaped part that turns red in the selected item.
- The label is the text, although you can use an image or bitmap as the label.
- If you prefer, you can dispense with the indicator. This makes the radiobuttons look like "push-push" buttons, with the selected entry appearing sunken and the rest appearing raised.
- To form several radiobuttons into a functional group, create a single control variable (see Section 52, "Control variables: the values behind the widgets" (p. 153), below), and set the `variable` option of each radiobutton to that variable.

  The control variable can be either an `IntVar` or a `StringVar`. If two or more radiobuttons share the same control variable, setting any of them will clear the others.
- Each radiobutton in a group must have a unique `value` option of the same type as the control variable. For example, a group of three radiobuttons might share an `IntVar` and have values of 0, 1, and 99. Or you can use a `StringVar` control variable and give the radiobuttons `value` options like `'too hot'`, `'too cold'`, and `'just right'`.

To create a new radiobutton widget as the child of a root window or frame named *parent*:

```
w = tk.Radiobutton(parent, option, ...)
```

This constructor returns the new `Radiobutton` widget. Options:

Table 29. `Radiobutton` widget options

`activebackground`	The background color when the mouse is over the radiobutton. See Section 5.3, "Colors" (p. 10).
`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=tk.CENTER`. For other positioning options, see Section 5.5, "Anchors" (p. 12). For example, if you set `anchor=tk.NE`, the radiobutton will be placed in the top right corner of the available space.
`bg` or `background`	The normal background color behind the indicator and label.
`bitmap`	To display a monochrome image on a radiobutton, set this option to a bitmap; see Section 5.7, "Bitmaps" (p. 12).
`bd` or `borderwidth`	The size of the border around the indicator part itself. Default is two pixels. For possible values, see Section 5.1, "Dimensions" (p. 9).
`command`	A procedure to be called every time the user changes the state of this radiobutton.
`compound`	If you specify both text and a graphic (either a bitmap or an image), this option specifies where the graphic appears relative to the text. Possible values are `tk.NONE` (the default value), `tk.TOP`, `tk.BOTTOM`, `tk.LEFT`, `tk.RIGHT`, and `tk.CENTER`. For example, `compound=tk.BOTTOM` would position the graphic below the text. If you specify `compound=tk.NONE`, the graphic is displayed but the `text` (if any) is not.
`cursor`	If you set this option to a cursor name (see Section 5.8, "Cursors" (p. 13)), the mouse cursor will change to that pattern when it is over the radiobutton.
`disabledforeground`	The foreground color used to render the text of a disabled radiobutton. The default is a stippled version of the default foreground color.
`font`	The font used for the `text`. See Section 5.4, "Type fonts" (p. 10).
`fg` or `foreground`	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. See Section 53, "Focus: routing keyboard input" (p. 155).
`highlightcolor`	The color of the focus highlight when the radiobutton has the focus.
`highlightthickness`	The thickness of the focus highlight. Default is `1`. Set `highlightthickness=0` to suppress display of the focus highlight.
`image`	To display a graphic image instead of text for this radiobutton, set this option to an image object. See Section 5.9, "Images" (p. 14). The image appears when the radiobutton is *not* selected; compare `selectimage`, below.
`indicatoron`	Normally a radiobutton displays its indicator. If you set this option to zero, the indicator disappears, and the entire widget becomes a "push-push" button that looks raised when it is cleared and sunken when it is set. You

	may want to increase the `borderwidth` value to make it easier to see the state of such a control.
`justify`	If the `text` contains multiple lines, this option controls how the text is justified: `tk.CENTER` (the default), `tk.LEFT`, or `tk.RIGHT`.
`offrelief`	If you suppress the indicator by asserting `indicatoron=False`, the `offrelief` option specifies the relief style to be displayed when the radiobutton is not selected. The default values is `tk.RAISED`.
`overrelief`	Specifies the relief style to be displayed when the mouse is over the radiobutton.
`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`	By default, a radiobutton will have `tk.FLAT` relief, so it doesn't stand out from its background. See Section 5.6, “Relief styles” (p. 12) for more 3-d effect options. You can also use `relief=tk.SOLID`, which displays a solid black frame around the radiobutton.
`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. See Section 5.9, “Images” (p. 14).
`state`	The default is `state=tk.NORMAL`, but you can set `state=tk.DISABLED` to gray out the control and make it unresponsive. If the cursor is currently over the radiobutton, the state is `tk.ACTIVE`.
`takefocus`	By default, the input focus (see Section 53, “Focus: routing keyboard input” (p. 155)) will pass through a radiobutton. If you set `takefocus=0`, focus will not visit this radiobutton.
`text`	The label displayed next to the radiobutton. Use newlines (`'\n'`) to display multiple lines of text.
`textvariable`	If you need to change the label on a radiobutton during execution, create a `StringVar` (see Section 52, “Control variables: the values behind the widgets” (p. 153)) to manage the current value, and set this option to that control variable. Whenever the control variable's value changes, the radiobutton's annotation will automatically change to that text as well.
`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.
`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; see Section 52, “Control variables: the values behind the widgets” (p. 153). This can be either an `IntVar` or a `StringVar`.

`width`	The default width of a radiobutton is determined by the size of the displayed image or text. You can set this option to a number of characters (*not* pixels) and the radiobutton 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.

Methods on radiobutton objects include:

`.deselect()`
Clears (turns off) the radiobutton.

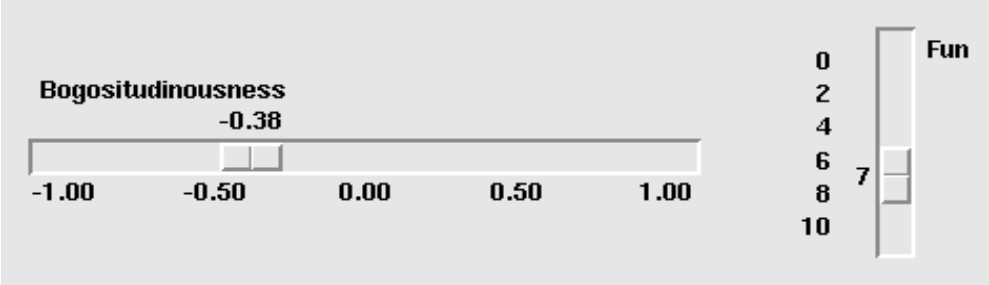
`.flash()`
Flashes the radiobutton a few times between its active and normal colors, but leaves it the way it started.

`.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.

21. The `Scale` widget

The purpose of a scale widget is to allow the user to set some `int` or `float` value within a specified range. Here are two scale widgets, one horizontal and one vertical:



Each scale displays a *slider* that the user can drag along a *trough* to change the value. In the figure, the first slider is currently at -0.38 and the second at 7.

- You can drag the slider to a new value with mouse button 1.
- If you click button 1 in the trough, the slider will move one increment in that direction per click. Holding down button 1 in the trough will, after a delay, start to auto-repeat its function.
- If the scale has keyboard focus, left arrow and up arrow keystrokes will move the slider up (for vertical scales) or left (for horizontal scales). Right arrow and down arrow keystrokes will move the slider down or to the right.

To create a new scale widget as the child of a root window or frame named *`parent`*:

```
    w = tk.Scale(parent, option, ...)
```

The constructor returns the new `Scale` widget. Options:

Table 30. `Scale` widget options

`activebackground`	The color of the slider when the mouse is over it. See Section 5.3, "Colors" (p. 10).
`bg` or `background`	The background color of the parts of the widget that are outside the trough.
`bd` or `borderwidth`	Width of the 3-d border around the trough and slider. Default is two pixels. For acceptable values, see Section 5.1, "Dimensions" (p. 9).
`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`	The cursor that appears when the mouse is over the scale. See Section 5.8, "Cursors" (p. 13).
`digits`	The way your program reads the current value shown in a scale widget is through a control variable; see Section 52, "Control variables: the values behind the widgets" (p. 153). The control variable for a scale can be an `IntVar`, a `DoubleVar` (for type `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. See Section 5.4, "Type fonts" (p. 10).
`fg` or `foreground`	The color of the text used for the label and annotations.
`from_`	A `float` value that defines one end of the scale's range. For vertical scales, this is the top end; for horizontal scales, the left end. The underbar (_) is not a typo: because `from` is a reserved word in Python, this option is spelled `from_`. The default is 0.0. See the `to` option, below, for the other end of the range.
`highlightbackground`	The color of the focus highlight when the scale does not have focus. See Section 53, "Focus: routing keyboard input" (p. 155).
`highlightcolor`	The color of the focus highlight when the scale has the focus.
`highlightthickness`	The thickness of the focus highlight. Default is 1. Set `highlightthickness=0` to suppress display of the focus highlight.
`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. For allowable values, see Section 5.1, "Dimensions" (p. 9).
`orient`	Set `orient=tk.HORIZONTAL` if you want the scale to run along the *x* dimension, or `orient=tk.VERTICAL` to run parallel to the *y*-axis. Default is vertical.
`relief`	With the default `relief=tk.FLAT`, the scale does not stand out from its background. You may also use `relief=tk.SOLID` to get a solid black frame around the scale, or any of the other relief types described in Section 5.6, "Relief styles" (p. 12).

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	This option controls how often the slider jumps once button 1 has been held down in the trough for at least repeatdelay milliseconds. For example, repeatinterval=100 would jump the slider every 100 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. All smaller movements will be ignored. Use resolution=-1 to disable any rounding of values.
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; see Section 5.1, "Dimensions" (p. 9).
sliderrelief	By default, the slider is displayed with a tk.RAISED relief style. For other relief styles, set this option to any of the values described in Section 5.6, "Relief styles" (p. 12).
state	Normally, scale widgets respond to mouse events, and when they have the focus, also keyboard events. Set state=tk.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. See Section 53, "Focus: routing keyboard input" (p. 155).
tickinterval	Normally, no "ticks" are displayed along the scale. 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 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. The default value is 100.0.
troughcolor	The color of the trough.
variable	The control variable for this scale, if any; see Section 52, "Control variables: the values behind the widgets" (p. 153). Control variables may be from class IntVar, DoubleVar (for type float), or StringVar. In the latter case, the numerical value will be converted to a string. See the the digits option, above, for more information on this conversion.
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=tk.HORIZONTAL. Default is 15 pixels.

Scale objects have these methods:

`.coords(value=None)`
Returns the coordinates, relative to the upper left corner of the widget, corresponding to a given value of the scale. For `value=None`, you get the coordinates of the center of the slider at its current position. To find where the slider would be if the scale's value were set to some value *`x`*, use `value=`*`x`*.

`.get()`
This method returns the current value of the scale.

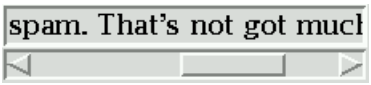
`.identify(`*`x, y`*`)`
Given a pair of coordinates (*`x, y`*) relative to the top left corner of the widget, this method returns a string identifying what functional part of the widget is at that location. The return value may be any of these:

`'slider'`	The slider.
`'trough1'`	For horizontal scales, to the left of the slider; for vertical scales, above the slider.
`'trough2'`	For horizontal scales, to the right of the slider; for vertical scales, below the slider.
`''`	Position (*`x, y`*) is not on any of the above parts.

`.set(`*`value`*`)`
Sets the scale's value.

22. The `Scrollbar` widget

A number of widgets, such as listboxes and canvases, can act like sliding windows into a larger virtual area. You can connect scrollbar widgets to them to give the user a way to slide the view around relative to the contents. Here's a screen shot of an entry widget with an associated scrollbar widget:



- Scrollbars can be horizontal, like the one shown above, or vertical. A widget that has two scrollable dimensions, such as a canvas or listbox, can have both a horizontal and a vertical scrollbar.
- The *slider*, or *scroll thumb*, is the raised-looking rectangle that shows the current scroll position.
- The two triangular *arrowheads* at each end are used for moving the position by small steps. The one on the left or top is called `arrow1`, and the one on the right or bottom is called `arrow2`.
- The *trough* is the sunken-looking area visible behind the arrowheads and slider. The trough is divided into two areas named `trough1` (above or to the left of the slider) and `trough2` (below or to the right of the slider).
- The slider's size and position, relative to the length of the entire widget, show the size and position of the view relative to its total size. For example, if a vertical scrollbar is associated with a listbox, and its slider extends from 50% to 75% of the height of the scrollbar, that means that the visible part of the listbox shows that portion of the overall list starting at the halfway mark and ending at the three-quarter mark.
- In a horizontal scrollbar, clicking B1 (button 1) on the left arrowhead moves the view by a small amount to the left. Clicking B1 on the right arrowhead moves the view by that amount to the right. For a vertical scrollbar, clicking the upward- and downward-pointing arrowheads moves the view

small amounts up or down. Refer to the discussion of the associated widget to find out the exact amount that these actions move the view.

- The user can drag the slider with B1 or B2 (the middle button) to move the view.
- For a horizontal scrollbar, clicking B1 in the trough to the left of the slider moves the view left by a page, and clicking B1 in the trough to the right of the slider moves the view a page to the right. For a vertical scrollbar, the corresponding actions move the view a page up or down.
- Clicking B2 anywhere along the trough moves the slider so that its left or top end is at the mouse, or as close to it as possible.

The *normalized position* of the scrollbar refers to a number in the closed interval [0.0, 1.0] that defines the slider's position. For vertical scrollbars, position 0.0 is at the top and 1.0 at the bottom; for horizontal scrollbars, position 0.0 is at the left end and 1.0 at the right.

To create a new `Scrollbar` widget as the child of a root window or frame *`parent`*:

```
w = tk.Scrollbar(parent, option, ...)
```

The constructor returns the new `Scrollbar` widget. Options for scrollbars include:

Table 31. `Scrollbar` widget options

`activebackground`	The color of the slider and arrowheads when the mouse is over them. See Section 5.3, “Colors” (p. 10).
`activerelief`	By default, the slider is shown with the `tk.RAISED` relief style. To display the slider with a different relief style when the mouse is over the slider.
`bg` or `background`	The color of the slider and arrowheads when the mouse is not over them.
`bd` or `borderwidth`	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 two-pixel border around the arrowheads and slider. For possible values, see Section 5.1, “Dimensions” (p. 9).
`command`	A procedure to be called whenever the scrollbar is moved. For a discussion of the calling sequence, see Section 22.1, “The `Scrollbar` *`command`* callback” (p. 77).
`cursor`	The cursor that appears when the mouse is over the scrollbar. See Section 5.8, “Cursors” (p. 13).
`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. See Section 53, “Focus: routing keyboard input” (p. 155).
`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=tk.HORIZONTAL` for a horizontal scrollbar, `orient=tk.VERTICAL` for a vertical one (the default orientation).

`relief`	Controls the relief style of the widget; the default style is `tk.SUNKEN`. This option has no effect in Windows.
`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`	This option controls how often slider movement will repeat when button 1 is held down in the trough. Default is `repeatinterval=100`, and the units are milliseconds.
`takefocus`	Normally, you can tab the focus through a scrollbar widget; see Section 53, "Focus: routing keyboard input" (p. 155). Set `takefocus=0` if you don't want this behavior. The default key bindings for scrollbars allow the user to use the ← and → arrow keys to move horizontal scrollbars, and they can use the ↑ and ↓ keys to move vertical scrollbars.
`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. For possible values, see Section 5.1, "Dimensions" (p. 9).

Methods on scrollbar objects include:

`.activate(element=None)`

If no argument is provided, this method returns one of the strings `'arrow1'`, `'arrow2'`, `'slider'`, or `''`, depending on where the mouse is. For example, the method returns `'slider'` if the mouse is on the slider. The empty string is returned if the mouse is not currently on any of these three controls.

To highlight one of the controls (using its `activerelief` relief style and its `activebackground` color), call this method and pass a string identifying the control you want to highlight, one of `'arrow1'`, `'arrow2'`, or `'slider'`.

`.delta(`*`dx`*`, `*`dy`*`)`

Given a mouse movement of `(`*`dx`*`, `*`dy`*`)` in pixels, this method returns the `float` value that should be added to the current slider position to achieve that same movement. The value must be in the closed interval [-1.0, 1.0].

`.fraction(`*`x`*`, `*`y`*`)`

Given a pixel location `(`*`x`*`, `*`y`*`)`, this method returns the corresponding normalized slider position in the interval [0.0, 1.0] that is closest to that location.

`.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. Each value is in the interval [0.0, 1.0] where 0.0 is the leftmost or top position and 1.0 is the rightmost or bottom position. For example, if the slider extends from halfway to three-quarters of the way along the trough, you might get back the tuple (0.5,0.75).

`.identify(`*`x`*`, `*`y`*`)`

This method returns a string indicating which (if any) of the components of the scrollbar are under the given `(`*`x`*`, `*`y`*`)` coordinates. The return value is one of `'arrow1'`, `'trough1'`, `'slider'`, `'trough2'`, `'arrow2'`, or the empty string `''` if that location is not on any of the scrollbar components.

.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. Please note that moving the scrollbar's slider does *not* move the corresponding widget.

22.1. The `Scrollbar` *`command`* callback

When the user manipulates a scrollbar, the scrollbar calls its `command` callback. The arguments to this call depend on what the user does:

- When the user requests a movement of one "unit" left or up, for example by clicking button B1 on the left or top arrowhead, the arguments to the callback look like:

```
    command(tk.SCROLL, -1, tk.UNITS)
```

- When the user requests a movement of one unit right or down, the arguments are:

```
    command(tk.SCROLL, 1, tk.UNITS)
```

- When the user requests a movement of one page left or up:

```
    command(tk.SCROLL, -1, tk.PAGES)
```

- When the user requests a movement of one page right or down:

```
    command(tk.SCROLL, 1, tk.PAGES)
```

- When the user drags the slider to a value *f* in the range [0,1], where 0 means all the way left or up and 1 means all the way right or down, the call is:

```
    command(tk.MOVETO, f)
```

These calling sequences match the arguments expected by the `.xview()` and `.yview()` methods of canvases, listboxes, and text widgets. The `Entry` widget does not have an `.xview()` method. See Section 10.1, "Scrolling an `Entry` widget" (p. 45).

22.2. Connecting a `Scrollbar` to another widget

Here is a code fragment showing the creation of a canvas with horizontal and vertical scrollbars. In this fragment, `self` is assumed to be a `Frame` widget.

```
    self.canv = tk.Canvas(self, width=600, height=400,
        scrollregion=(0, 0, 1200, 800))
    self.canv.grid(row=0, column=0)

    self.scrollY = tk.Scrollbar(self, orient=tk.VERTICAL,
        command=self.canv.yview)
    self.scrollY.grid(row=0, column=1, sticky=tk.N+tk.S)

    self.scrollX = tk.Scrollbar(self, orient=tk.HORIZONTAL,
        command=self.canv.xview)
    self.scrollX.grid(row=1, column=0, sticky=tk.E+tk.W)
```

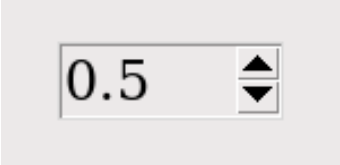
```
    self.canv['xscrollcommand'] = self.scrollX.set
    self.canv['yscrollcommand'] = self.scrollY.set
```

Notes:

- The connection goes both ways. The canvas's `xscrollcommand` option has to be connected to the horizontal scrollbar's `.set` method, and the scrollbar's `command` option has to be connected to the canvas's `.xview` method. The vertical scrollbar and canvas must have the same mutual connection.
- The `sticky` options on the `.grid()` method calls for the scrollbars force them to stretch just enough to fit the corresponding dimension of the canvas.

23. The `Spinbox` widget

The `Spinbox` widget allows the user to select values from a given set. The values may be a range of numbers, or a fixed set of strings.



On the screen, a `Spinbox` has an area for displaying the current values, and a pair of arrowheads.

- The user can click the upward-pointing arrowhead to advance the value to the next higher value in sequence. If the value is already at maximum, you can set up the widget, if you wish, so that the new value will wrap around to the lowest value.
- The user can click the downward-pointing arrowhead to advance the value to the next lower value in sequence. This arrow may also be configured to wrap around, so that if the current value is the lowest, clicking on the down-arrow will display the highest value.
- The user can also enter values directly, treating the widget as if it were an `Entry`. The user can move the focus to the widget (see Section 53, "Focus: routing keyboard input" (p. 155)), either by clicking on it or by using *tab* or *shift-tab*, and then edit the displayed value.

To create a new `Spinbox` widget as the child of a root window or frame *`parent`*:

```
    w = tk.Spinbox(parent, option, ...)
```

The constructor returns the new `Spinbox` widget. Options include:

Table 32. `Spinbox` widget options

`activebackground`	Background color when the cursor is over the widget; see Section 5.3, "Colors" (p. 10).
`bg` or `background`	Background color of the widget.
`bd` or `borderwidth`	Width of the border around the widget; see Section 5.1, "Dimensions" (p. 9). The default value is one pixel.
`buttonbackground`	The background color displayed on the arrowheads. The default is gray.

buttoncursor	The cursor to be displayed when the mouse is over the arrowheads; see Section 5.8, “Cursors” (p. 13).
buttondownrelief	The relief style for the downward-pointing arrowhead; see Section 5.6, “Relief styles” (p. 12). The default style is tk.RAISED.
buttonup	The relief style for the upward-pointing arrowhead; see Section 5.6, “Relief styles” (p. 12). The default style is tk.RAISED.
command	Use this option to specify a function or method to be called whenever the user clicks on one of the arrowheads. Note that the callback is *not* called when the user edits the value directly as if it were an Entry.
cursor	Selects the cursor that is displayed when the mouse is over the entry part of the widget; see Section 5.8, “Cursors” (p. 13).
disabledbackground	These options select the background and foreground colors displayed when the widget's state is tk.DISABLED.
disabledforeground	
exportselection	Normally, the text in the entry portion of a Spinbox can be cut and pasted. To prohibit this behavior, set the exportselection option to True.
font	Use this option to select a different typeface for the entry text; see Section 5.4, “Type fonts” (p. 10).
fg or foreground	This option selects the color used to display the text in the entry part of the widget, and the color of the arrowheads.
format	Use this option to control the formatting of numeric values in combination with the from_ and to options. For example, format='%10.4f' would display the value as a ten-character field, with four digits after the decimal.
from_	Use this option in combination with the to option (described below) to constrain the values to a numeric range. For example, from_=1 and to=9 would allow only values between 1 and 9 inclusive. See also the increment option below.
highlightbackground	The color of the focus highlight when the Spinbox does not have focus. See Section 53, “Focus: routing keyboard input” (p. 155).
highlightcolor	The color of the focus highlight when the Spinbox has the focus.
highlightthickness	The thickness of the focus highlight. Default is 1. Set to 0 to suppress display of the focus highlight.
increment	When you constrain the values with the from_ and to options, you can use the increment option to specify how much the value increases or decreases when the user clicks on an arrowhead. For example, with options from_=0.0, to=2.0, and increment=0.5, the up-arrowhead will step through values 0.0, 0.5, 1.0, 1.5, and 2.0.
insertbackground	Selects the color of the insertion cursor displayed in the entry part of the widget.
insertborderwidth	This option controls the width of the border around the insertion cursor. Normally, the insertion cursor will have no border. If this option is set to a nonzero value, the insertion cursor will be displayed in the tk.RAISED relief style.
insertofftime	These two options control the blink cycle of the insertion cursor: the amount of time it spends off and on, respectively, in milliseconds. For example, with options insertofftime=200 and insertontime=400, the cursor would blink off for 0.2 seconds and then on for 0.4 seconds.
insertontime	

`insertwidth`	Use this option to specify the width of the insertion cursor; for possible values, see Section 5.1, “Dimensions” (p. 9). The default width is two pixels.
`justify`	This option controls the position of the text in the entry part of the widget. Values may be `tk.LEFT` to left-justify the text; `tk.CENTER` to center it; or `RIGHT` to right-justify the text.
`readonlybackground`	This option specifies the background color that will be displayed when the widget's `state` is `'readonly'`; see Section 5.3, “Colors” (p. 10).
`relief`	Use this option to select a relief style for the widget; see Section 5.6, “Relief styles” (p. 12). The default style is `tk.SUNKEN`.
`repeatdelay`	These options specify the auto-repeat behavior of mouse clicks on the arrowheads; values are in milliseconds. The `repeatdelay` value specifies how long the mouse button must be held down before it repeats, and `repeatinterval` specifies how often the function repeats. Default values are 400 and 100 milliseconds, respectively.
`repeatinterval`	
`selectbackground`	The background color to use displaying selected items.
`selectborderwidth`	The width of the border to display around selected items.
`selectforeground`	The foreground color to use displaying selected items.
`state`	Normally, a `Spinbox` widget is created in the `tk.NORMAL` state. Set this option to `tk.DISABLED` to make the widget unresponsive to mouse or keyboard actions. If you set it to `'readonly'`, the value in the entry part of the widget cannot be modified with keystrokes, but the value can still be copied to the clipboard, and the widget still responds to clicks on the arrowheads.
`takefocus`	Normally, the entry part of a `Spinbox` widget can have focus (see Section 53, “Focus: routing keyboard input” (p. 155)). To remove the widget from the focus traversal sequence, set `takefocus=False`.
`textvariable`	If you want to retrieve the current value of the widget, you can use the `.get()` method below, or you can associate a control variable with the widget by passing that control variable as the value of this option. See Section 52, “Control variables: the values behind the widgets” (p. 153).
`to`	This option specifies the upper limit of a range values. See the `from_` option, above, and also the `increment` option.
`values`	There are two ways to specify the possible values of the widget. One way is to provide a tuple of strings as the value of the `values` option. For example, `values=('red', 'blue', 'green')` would allow only those three strings as values. To configure the widget to accept a range of numeric values, see the `from_` option above.
`width`	Use this option to specify the number of characters allowed in the entry part of the widget. The default value is 20.
`wrap`	Normally, when the widget is at its highest value, the up-arrowhead does nothing, and when the widget is at its lowest value, the down-arrowhead does nothing. If you select `wrap=True`, the up-arrowhead will advance from the highest value back to the lowest, and the down-arrowhead will advance from the lowest value back to the highest.

`xscrollcommand`	Use this option to connect a scrollbar to the entry part of the widget. For details, see Section 22.2, "Connecting a `Scrollbar` to another widget" (p. 77).

These methods are available on `Spinbox` widgets:

`.bbox(`*`index`*`)`
This method returns the bounding box of the character at position *`index`* in the entry part of the widget. The result is a tuple `(`*`x`*`, `*`y`*`, `*`w`*`, `*`h`*`)`, where the values are the *`x`* and *`y`* coordinates of the upper left corner, and the character's width and height in pixels, in that order.

`.delete(first, last=None)`
This method deletes characters from the entry part of the `Spinbox`. The values of `first` and `last` are interpreted in the standard way for Python slices.

`.get()`
This method returns the value of the `Spinbox`. The value is always returned as a string, even if the widget is set up to contain a number.

`.icursor(`*`index`*`)`
Use this method to position the insertion cursor at the location specified by *`index`*, using the standard Python convention for positions.

`.identify(`*`x`*`, `*`y`*`)`
Given a position `(`*`x`*`, `*`y`*`)` within the widget, this method returns a string describing what is at that location. Values may be any of:
- `'entry'` for the entry area.
- `'buttonup'` for the upward-pointing arrowhead.
- `'buttondown'` for the downward-pointing arrowhead.
- `''` (an empty string) if these coordinates are not within the widget.

`.index(`*`i`*`)`
This method returns the numerical position of an index *`i`*. Arguments may be any of:
- `tk.END` to get the position after the last character of the entry.
- `tk.INSERT` to get the position of the insertion cursor.
- `tk.ANCHOR` to get the position of the selection anchor.
- `tk.SEL_FIRST'` to get the position of the start of the selection. If the selection is not within the widget, this method raises a `tk.TclError` exception.
- `tk.SEL_LAST` to get the position just past the end of the selection. If the selection is not within the widget, this method raises a `tk.TclError` exception.
- A string of the form "`@`*`x`*" denotes an *`x`*-coordinate within the widget. The return value is the position of the character containing that coordinate. If the coordinate is outside the widget altogether, the return value will be the position of the character closest to that position.

`.insert(`*`index`*`, `*`text`*`)`
This method inserts characters from the string *`text`* at the position specified by *`index`*. For the possible index values, see the `.index()` method above.

`.invoke(`*`element`*`)`
Call this method to get the same effect as the user clicking on an arrowhead. The *`element`* argument is `'buttonup'` for the up-arrowhead, and `'buttondown'` for the down-arrowhead.

`.scan_dragto(`*`x`*`)`
This method works the same as the `.scan_dragto()` method described in Section 10, "The `Entry` widget" (p. 41).

.scan_mark(*x*)
This method works the same as the `.scan_mark()` method described in Section 10, "The `Entry` widget" (p. 41).

.selection('from', *index*)
Sets the selection anchor in the widget to the position specified by the *`index`*. For the possible values of *`index`*, see the `.index()` method above. The initial value of the selection anchor is 0.

.selection('to', *index*)
Selects the text between the selection anchor and the given *`index`*.

.selection('range', *start*, *end*)
Select the text between the *`start`* and *`end`* indices. For allowable index values, see the `.index()` method above.

.selection_clear()
Clears the selection.

.selection_get()
Returns the selected text. If there is currently no selection, this method will raise a `tk.TclError` exception.

24. The `Text` widget

Text widgets are a much more generalized method for handling multiple lines of text than the `Label` widget. Text widgets are pretty much a complete text editor in a window:

- You can mix text with different fonts, colors, and backgrounds.
- You can intersperse embedded images with text. An image is treated as a single character. See Section 24.3, "`Text` widget images" (p. 86).
- An *index* is a way of describing a specific position between two characters of a text widget. See Section 24.1, "`Text` widget indices" (p. 84).
- A text widget may contain invisible *mark* objects between character positions. See Section 24.2, "`Text` widget marks" (p. 86).
- Text widgets allow you to define names for regions of the text called *tags*. You can change the appearance of a tagged region, changing its font, foreground and background colors, and other option. See Section 24.5, "`Text` widget tags" (p. 87).
- You can bind events to a tagged region. See Section 54, "Events" (p. 157).
- You can even embed a text widget in a "window" containing any *Tkinter* widget—even a frame widget containing other widgets. A window is also treated as a single character. See Section 24.4, "`Text` widget windows" (p. 87).

To create a text widget as the child of a root window or frame named *`parent`*:

```
    w = tk.Text(parent, option, ...)
```

The constructor returns the new `Text` widget. Options include:

Table 33. `Text` widget options

`autoseparators`	If the `undo` option is set, the `autoseparators` option controls whether separators are automatically added to the undo stack after each insertion or deletion (if `autoseparators=True`) or not (if `autoseparat-`

	ors=False). For an overview of the undo mechanism, see Section 24.7, "The Text widget undo/redo stack" (p. 88).
bg or background	The default background color of the text widget. See Section 5.3, "Colors" (p. 10).
bd or borderwidth	The width of the border around the text widget; see Section 5.1, "Dimensions" (p. 9). The default is two pixels.
cursor	The cursor that will appear when the mouse is over the text widget. See Section 5.8, "Cursors" (p. 13).
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. Note that you can have multiple fonts in the widgets by using tags to change the properties of some text. See Section 5.4, "Type fonts" (p. 10).
fg or foreground	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. See Section 53, "Focus: routing keyboard input" (p. 155).
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.
maxundo	This option sets the maximum number of operations retained on the undo stack. For an overview of the undo mechanism, see Section 24.7, "The Text widget undo/redo stack" (p. 88). Set this option to -1 to specify an unlimited number of entries in the undo stack.
padx	The size of the internal padding added to the left and right of the text area. Default is one pixel. For possible values, see Section 5.1, "Dimensions" (p. 9).
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=tk.SUNKEN; for other values, see Section 5.6, "Relief styles" (p. 12).
selectbackground	The background color to use displaying selected text.
selectborderwidth	The width of the border to use around selected text.
selectforeground	The foreground color to use displaying 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=tk.NORMAL` to get this behavior. If you set `state=tk.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. See Section 24.6, "Setting tabs in a `Text` widget" (p. 87).
`takefocus`	Normally, focus will visit a text widget (see Section 53, "Focus: routing keyboard input" (p. 155)). Set `takefocus=0` if you do not want focus in the widget.
`undo`	Set this option to `True` to enable the undo mechanism, or `False` to disable it. See Section 24.7, "The `Text` widget undo/redo stack" (p. 88).
`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. • With the default behavior, `wrap=tk.CHAR`, any line that gets too long will be broken at any character. • Set `wrap=tk.WORD` and it will break the line after the last word that will fit. • If you want to be able to create lines that are too long to fit in the window, set `wrap=tk.NONE` and provide a horizontal scrollbar.
`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.

24.1. `Text` widget indices

An *index* is a general method of specifying a position in the content of a text widget. An index is a string with one of these forms:

`'`*`line`*`.`*`column`*`'`
: The position just before the given *`column`* (counting from zero) on the given *`line`* (counting from one). Examples: `'1.0'` is the position of the beginning of the text; `'2.3'` is the position before the fourth character of the second line.

`'`*`line`*`.end'`
: The position just before the newline at the end of the given line (counting from one). So, for example, index `'10.end'` is the position at the end of the tenth line.

tk.INSERT
The position of the insertion cursor in the text widget. This constant is equal to the string `'insert'`.

tk.CURRENT
The position of the character closest to the mouse pointer. This constant is equal to the string `'current'`.

tk.END
The position after the last character of the text. This constant is equal to the string `'end'`.

tk.SEL_FIRST
If some of the text in the widget is currently selection (as by dragging the mouse over it), this is the position before the start of the selection. If you try to use this index and nothing is selected, a `tk.TclError` exception will be raised. This constant is equal to the string `'sel.first'`.

tk.SEL_LAST
The position after the end of the selection, if any. As with `SEL_FIRST`, you'll get a `tk.TclError` exception if you use such an index and there is no selection. This constant is equal to the string `'sel.last'`.

'*markname*'
You can use a mark as an index; just pass its name where an index is expected. See Section 24.2, "`Text` widget marks" (p. 86).

'*tag*.first'
The position before the first character of the region tagged with name *tag*; see Section 24.5, "`Text` widget tags" (p. 87).

'*tag*.last'
The position after the last character of a tagged region.

'@*x*,*y*'
The position before the character closest to the coordinate (*x*, *y*).

embedded-object
If you have an image or window embedded in the text widget, you can use the `PhotoImage`, `BitmapImage`, or embedded widget as an index. See Section 24.3, "`Text` widget images" (p. 86) and Section 24.4, "`Text` widget windows" (p. 87).

In addition to the basic index options above, you can build arbitrary complex expressions by adding any of these suffixes to a basic index or index expression:

+ *n* chars
From the given index, move forward *n* characters. This operation will cross line boundaries.

For example, suppose the first line looks like this:

```
abcdef
```

The index expression "`1.0 + 5 chars`" refers to the position between `e` and `f`. You can omit blanks and abbreviate keywords in these expressions if the result is unambiguous. This example could be abbreviated "`1.0+5c`".

- *n* chars
Similar to the previous form, but the position moves *backwards* *n* characters.

+ *n* lines
Moves *n* lines past the given index. *Tkinter* tries to leave the new position in the same column as it was on the line it left, but if the line at the new position is shorter, the new position will be at the end of the line.

- *n* lines
Moves *n* lines before the given index.

linestart
Moves to the position before the first character of the given index. For example, position "`current linestart`" refers to the beginning of the line closest to the mouse pointer.

lineend
Moves to the position after the last character of the given index. For example, position "`sel.last lineend`" refers to the end of the line containing the end of the current selection.

wordstart
The position before the beginning of the word containing the given index. For example, index "`11.44 wordstart`" refers to the position before the word containing position 44 on line 11.

For the purposes of this operation, a word is either a string of consecutive letter, digit, or underbar (_) characters, or a single character that is none of these types.

24.2. `Text` widget marks

A *mark* represents a floating position somewhere in the contents of a text widget.

- You handle each mark by giving it a name. This name can be any string that doesn't include whitespace or periods.
- There are two special marks. `tk.INSERT` is the current position of the insertion cursor, and `tk.CURRENT` is the position closest to the mouse cursor.
- Marks float along with the adjacent content. If you modify text somewhere away from a mark, the mark stays at the same position relative to its immediate neighbors.
- Marks have a property called *gravity* that controls what happens when you insert text at a mark. The default gravity is `tk.RIGHT`, which means that when new text is inserted at that mark, the mark stays after the end of the new text. If you set the gravity of a mark to `tk.LEFT` (using the text widget's `.mark_gravity()` method), the mark will stay at a position just before text newly inserted at that mark.
- Deleting the text all around a mark does not remove the mark. If you want to remove a mark, use the `.mark_unset()` method on the text widget.

Refer to Section 24.8, "Methods on `Text` widgets" (p. 88), below, to see how to use marks.

24.3. `Text` widget images

You can put an image or bitmap into a text widget. It is treated as a single character whose size is the natural size of the object. See Section 5.9, "Images" (p. 14) andSection 5.7, "Bitmaps" (p. 12).

Images are placed into the text widget by calling that widget's `.image_create()` method. See below for the calling sequence and other methods for image manipulation.

Images are manipulated by passing their name to methods on the text widget. You can give *Tkinter* a name for an image, or you can just let *Tkinter* generate a default name for that image.

An image may appear any number of times within the same `Text` widget. Each instance will carry a unique name. This names can be used as an index.

24.4. `Text` widget windows

You can put any *Tkinter* widget—even a frame containing other widgets—into a text widget. For example, you can put a fully functional button or a set of radiobuttons into a text widget.

Use the `.window_create()` method on the text widget to add the embedded widget. For the calling sequence and related methods, see Section 24.8, "Methods on `Text` widgets" (p. 88).

24.5. `Text` widget tags

There are lots of ways to change both the appearance and functionality of the items in a text widget. For text, you can change the font, size, and color. Also, you can make text, widgets, or embedded images respond to keyboard or mouse actions.

To control these appearance and functional features, you associate each feature with a *tag*. You can then associate a tag with any number of pieces of text in the widget.

- The name of a tag can be any string that does not contain white space or periods.
- There is one special predefined tag called `SEL`. This is the region currently selected, if any.
- Since any character may be part of more than one tag, there is a *tag stack* that orders all the tags. Entries are added at the end of the tag list, and later entries have priority over earlier entries.

  So, for example, if there is a character c that is part of two tagged regions t_1 and t_2, and t_1 is deeper in the tag stack than t_2, and t_1 wants the text to be green and t_2 wants it to be blue, c will be rendered in blue because t_2 has precedence over t_1.
- You can change the ordering of tags in the tag stack.

Tags are created by using the `.tag_add()` method on the text widget. See Section 24.8, "Methods on `Text` widgets" (p. 88), below, for information on this and related methods.

24.6. Setting tabs in a `Text` widget

The `tabs` option for `Text` widgets gives you a number of ways to set tab stops within the widget.

- The default is to place tabs every eight characters.
- To set specific tab stops, set this option to a sequence of one or more distances. For example, setting `tabs=('3c', '5c', '12c')` would put tab stops 3, 5, and 12cm from the left side. Past the last tab you set, tabs have the same width as the distance between the last two existing tab stops. So, continuing our example, because `12c-5c` is 7 cm, if the user keeps pressing the *Tab* key, the cursor would be positioned at 19cm, 26cm, 33cm, and so on.
- Normally, text after a tab character is aligned with its left edge on the tab stop, but you can include any of the keywords `tk.LEFT`, `tk.RIGHT`, `tk.CENTER`, or `tk.NUMERIC` in the list after a distance, and that will change the positioning of the text after each tab.
  - A `tk.LEFT` tab stop has the default behavior.
  - A `tk.RIGHT` tab stop will position the text so its right edge is on the stop.
  - A `tk.CENTER` tab will center the following text on the tab stop.
  - A `tk.NUMERIC` tab stop will place following text to the left of the stop up until the first period (`'.'`) in the text—after that, the period will be centered on the stop, and the rest of the text will positioned to its right.

For example, setting `tabs=('0.5i', '0.8i', tk.RIGHT, '1.2i', tk.CENTER, '2i', tk.NUMERIC)` would set four tab stops: a left-aligned tab stop half an inch from the left side, a right-aligned tab stop 0.8″ from the left side, a center-aligned tab stop 1.2″ from the left, and a numeric-aligned tab stop 2″ from the left.

24.7. The `Text` widget undo/redo stack

The `Text` widget has a built-in mechanism that allows you to implement undo and redo operations that can cancel or reinstate changes to the text within the widget.

Here is how the undo/redo stack works:

- Every change to the content is recorded by pushing entries onto the stack that describe the change, whether an insertion or a deletion. These entries record the old state of the contents as well as the new state: if a deletion, the deleted text is recorded; if an insertion, the inserted text is recorded, along with a description of the location and whether it was an insertion or a deletion.
- Your program may also push a special record called a *separator* onto the stack.
- An *undo* operation changes the contents of the widget to what they were at some previous point. It does this by reversing all the changes pushed onto the undo/redo stack until it reaches a separator or until it runs out of stack.

  However, note that *Tkinter* also remembers how much of the stack was reversed in the undo operation, until some other editing operation changes the contents of the widget.
- A *redo* operation works only if no editing operation has occurred since the last undo operation. It re-applies all the undone operations.

For the methods used to implement the undo/redo stack, see the `.edit_redo`, `.edit_reset`, `.edit_separator`, and `.edit_undo` methods in Section 24.8, "Methods on `Text` widgets" (p. 88). The undo mechanism is not enabled by default; you must set the `undo` option in the widget.

24.8. Methods on `Text` widgets

These methods are available on all text widgets:

`.bbox(`*`index`*`)`
Returns the bounding box for the character at the given index, a 4-tuple `(`*`x`*`, `*`y`*`, `*`width`*`, `*`height`*`)`. If the character is not visible, returns `None`. Note that this method may not return an accurate value unless you call the `.update_idletasks()` method (see Section 26, "Universal widget methods" (p. 97)).

`.compare(`*`index1`*`, `*`op`*`, `*`index2`*`)`
Compares the positions of two indices in the text widget, and returns true if the relational *`op`* holds between *`index1`* and *`index2`*. The *`op`* specifies what comparison to use, one of: `'<'`, `'<='`, `'=='`, `'!='`, `'>='`, or `'>'`.

For example, for a text widget `t`, `t.compare('2.0', '<=', END)` returns true if the beginning of the second line is before or at the end of the text in `t`.

`.delete(`*`index1`*`, `*`index2`*`=None)`
Deletes text starting just after *`index1`*. If the second argument is omitted, only one character is deleted. If a second index is given, deletion proceeds up to, but not including, the character after *`index2`*. Recall that indices sit *between* characters.

.dlineinfo(*index*)
Returns a bounding box for the line that contains the given *index*. For the form of the bounding box, and a caution about updating idle tasks, see the definition of the `.bbox` method above.

.edit_modified(arg=None)
Queries, sets, or clears the *modified flag*. This flag is used to track whether the contents of the widget have been changed. For example, if you are implementing a text editor in a `Text` widget, you might use the modified flag to determine whether the contents have changed since you last saved the contents to a file.

When called with no argument, this method returns `True` if the modified flag has been set, `False` if it is clear. You can also explicitly set the modified flag by passing a `True` value to this method, or clear it by passing a `False` value.

Any operation that inserts or deletes text, whether by program actions or user actions, or an undo or redo operation, will set the modified flag.

.edit_redo()
Performs a redo operation. For details, see Section 24.7, "The `Text` widget undo/redo stack" (p. 88).

.edit_reset()
Clears the undo stack.

.edit_separator()
Pushes a separator onto the undo stack. This separator limits the scope of a future undo operation to include only the changes pushed since the separator was pushed. For details, see Section 24.7, "The `Text` widget undo/redo stack" (p. 88).

.edit_undo()
Reverses all changes to the widget's contents made since the last separator was pushed on the undo stack, or all the way to the bottom of the stack if the stack contains no separators. For details, see Section 24.7, "The `Text` widget undo/redo stack" (p. 88). It is an error if the undo stack is empty.

.image_create(*index*[, *option=value*, ...])
This method inserts an image into the widget. The image is treated as just another character, whose size is the image's natural size.

The options for this method are shown in the table below. You may pass either a series of *option=value* arguments, or a dictionary of option names and values.

`align`	This option specifies how the image is to be aligned vertically if its height is less than the height of its containing line. Values may be `top` to align it at the top of its space; `center` to center it; `bottom` to place it at the bottom; or `baseline` to line up the bottom of the image with the text baseline.
`image`	The image to be used. See Section 5.9, "Images" (p. 14).
`name`	You can assign a name to this instance of the image. If you omit this option, *Tkinter* will generate a unique name. If you create multiple instances of an image in the same `Text` widget, *Tkinter* will generate a unique name by appending a "`#`" followed by a number.
`padx`	If supplied, this option is a number of pixels of extra space to be added on both sides of the image.
`pady`	If supplied, this option is a number of pixels of extra space to be added above and below the image.

.get(*index1*, *index2*=None)
Use this method to retrieve the current text from the widget. Retrieval starts at index *index1*. If the second argument is omitted, you get the character after *index1*. If you provide a second index,

you get the text between those two indices. Embedded images and windows (widgets) are ignored. If the range includes multiple lines, they are separated by newline (`'\n'`) characters.

`.image_cget(`*`index`*`, `*`option`*`)`
To retrieve the current value of an option set on an embedded image, call this method with an index pointing to the image and the name of the option.

`.image_configure(`*`index`*`, `*`option`*`, ...)`
To set one or more options on an embedded image, call this method with an index pointing to the image as the first argument, and one or more *`option=value`* pairs.

If you specify no options, you will get back a dictionary defining all the options on the image, and the corresponding values.

`.image_names()`
This method returns a tuple of the names of all the text widget's embedded images.

`.index(`*`i`*`)`
For an index *`i`*, this method returns the equivalent position in the form *`'line.char'`*.

`.insert(`*`index`*`, `*`text`*`, `*`tags`*`=None)`
Inserts the given *`text`* at the given *`index`*.

If you omit the `tags` argument, the newly inserted text will be tagged with any tags that apply to the characters *both* before and after the insertion point.

If you want to apply one or more tags to the text you are inserting, provide as a third argument a tuple of tag strings. Any tags that apply to existing characters around the insertion point are ignored. *Note:* The third argument must be a tuple. If you supply a list argument, *Tkinter* will silently fail to apply the tags. If you supply a string, each character will be treated as a tag.

`.mark_gravity(`*`mark`*`, `*`gravity`*`=None)`
Changes or queries the gravity of an existing mark; see Section 24.2, "`Text` widget marks" (p. 86), above, for an explanation of gravity.

To set the gravity, pass in the name of the mark, followed by either `tk.LEFT` or `tk.RIGHT`. To find the gravity of an existing mark, omit the second argument and the method returns `tk.LEFT` or `tk.RIGHT`.

`.mark_names()`
Returns a sequence of the names of all the marks in the window, including `tk.INSERT` and `tk.CURRENT`.

`.mark_next(`*`index`*`)`
Returns the name of the mark following the given *`index`*; if there are no following marks, the method returns an empty string.

If the *`index`* is in numeric form, the method returns the first mark at that position. If the *`index`* is a mark, the method returns the next mark following that mark, which may be at the same numerical position.

`.mark_previous(`*`index`*`)`
Returns the name of the mark preceding the given *`index`*. If there are no preceding marks, the method returns an empty string.

If the *`index`* is in numeric form, the method returns returns the last mark at that position. If the *`index`* is a mark, the method returns the preceding mark, which may be at the same numerical position.

.mark_set(*mark*, *index*)
If no mark with name *mark* exists, one is created with tk.RIGHT gravity and placed where *index* points. If the mark already exists, it is moved to the new location.

This method may change the position of the tk.INSERT or tk.CURRENT indices.

.mark_unset(*mark*)
Removes the named mark. This method cannot be used to remove the tk.INSERT or tk.CURRENT marks.

.scan_dragto(*x*, *y*)
See .scan_mark, below.

.scan_mark(*x*, *y*)
This method is used to implement fast scrolling of a Text widget. Typically, a user presses and holds a mouse button at some position in the widget, and then moves the mouse in the desired direction, and the widget moves in that direction at a rate proportional to the distance the mouse has moved since the button was depressed. The motion may be any combination of vertical or horizontal scrolling.

To implement this feature, bind a mouse button down event to a handler that calls .scan_mark(*x*, *y*), where *x* and *y* are the current mouse position. Then bind the <Motion> event to a handler that calls .scan_dragto(*x*, *y*), where *x* and *y* are the new mouse position.

.search(*pattern*, *index*, *option*, ...)
Searches for *pattern* (which can be either a string or a regular expression) in the buffer starting at the given *index*. If it succeeds, it returns an index of the '*line*.*char*' form; if it fails, it returns an empty string.

The allowable options for this method are:

backwards	Set this option to True to search backwards from the index. Default is forwards.
count	If you set this option to an IntVar control variable, when there is a match you can retrieve the length of the text that matched by using the .get() method on that variable after the method returns.
exact	Set this option to True to search for text that exactly matches the *pattern*. This is the default option. Compare the regexp option below.
forwards	Set this option to True to search forwards from the index. This is the default option.
regexp	Set this option to True to interpret the *pattern* as a Tcl-style regular expression. The default is to look for an exact match to *pattern*. Tcl regular expressions are a subset of Python regular expressions, supporting these features: . ^ [c_1…] (…) * + ? e_1|e_2
nocase	Set this option to 1 to ignore case. The default is a case-sensitive search.
stopindex	To limit the search, set this option to the index beyond which the search should not go.

.see(*index*)
If the text containing the given index is not visible, scroll the text until that text is visible.

.tag_add(*tagName*, *index1*, *index2*=None)
This method associates the tag named *tagName* with a region of the contents starting just after index *index1* and extending up to index *index2*. If you omit *index2*, only the character after *index1* is tagged.

.tag_bind(*tagName*, *sequence*, *func*, *add*=None)

This method binds an event to all the text tagged with *tagName*. See Section 54, “Events” (p. 157), below, for more information on event bindings.

To create a new binding for tagged text, use the first three arguments: *sequence* identifies the event, and *func* is the function you want it to call when that event happens.

To add another binding to an existing tag, pass the same first three arguments and `'+'` as the fourth argument.

To find out what bindings exist for a given sequence on a tag, pass only the first two arguments; the method returns the associated function.

To find all the bindings for a given tag, pass only the first argument; the method returns a list of all the tag's *sequence* arguments.

.tag_cget(*tagName*, *option*)

Use this method to retrieve the value of the given *option* for the given *tagName*.

.tag_config(*tagName*, *option*, ...)

To change the value of options for the tag named *tagName*, pass in one or more *option*=*value* pairs.

If you pass only one argument, you will get back a dictionary defining all the options and their values currently in force for the named tag.

Here are the options for tag configuration:

`background`	The background color for text with this tag. Note that you can't use `bg` as an abbreviation.
`bgstipple`	To make the background appear grayish, set this option to one of the standard bitmap names (see Section 5.7, “Bitmaps” (p. 12)). This has no effect unless you also specify a `background`.
`borderwidth`	Width of the border around text with this tag. Default is `0`. Note that you can't use `bd` as an abbreviation.
`fgstipple`	To make the text appear grayish, set this option a bitmap name.
`font`	The font used to display text with this tag. See Section 5.4, “Type fonts” (p. 10).
`foreground`	The color used for text with this tag. Note that you can't use the `fg` abbreviation here.
`justify`	The `justify` option set on the first character of each line determines how that line is justified: `tk.LEFT` (the default), `tk.CENTER`, or `tk.RIGHT`.
`lmargin1`	How much to indent the first line of a chunk of text that has this tag. The default is `0`. See Section 5.1, “Dimensions” (p. 9)for allowable values.
`lmargin2`	How much to indent successive lines of a chunk of text that has this tag. The default is `0`.
`offset`	How much to raise (positive values) or lower (negative values) text with this tag relative to the baseline. Use this to get superscripts or subscripts, for example. For allowable values, see Section 5.1, “Dimensions” (p. 9).
`overstrike`	Set `overstrike=1` to draw a horizontal line through the center of text with this tag.
`relief`	Which 3-D effect to use for text with this tag. The default is `relief=tk.FLAT`; for other possible values see Section 5.6, “Relief styles” (p. 12).

`rmargin`	Size of the right margin for chunks of text with this tag. Default is `0`.
`spacing1`	This option specifies how much extra vertical space is put above each line of text with this tag. 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 with this tag when a logical line wraps. Default is `0`.
`spacing3`	This option specifies how much extra vertical space is added below each line of text with this tag. If a line wraps, this space is added only after the last line it occupies on the display. Default is `0`.
`tabs`	How tabs are expanded on lines with this tag. See Section 24.6, "Setting tabs in a `Text` widget" (p. 87).
`underline`	Set `underline=1` to underline text with this tag.
`wrap`	How long lines are wrapped in text with this tag. See the description of the `wrap` option for text widgets, above.

`.tag_delete(`*`tagName`*`, ...)`
To delete one or more tags, pass their names to this method. Their options and bindings go away, and the tags are removed from all regions of text.

`.tag_lower(`*`tagName`*`, `*`belowThis`*`=None)`
Use this method to change the order of tags in the tag stack (see Section 24.5, "`Text` widget tags" (p. 87), above, for an explanation of the tag stack). If you pass two arguments, the tag with name *`tagName`* is moved to a position just below the tag with name *`belowThis`*. If you pass only one argument, that tag is moved to the bottom of the tag stack.

`.tag_names(`*`index`*`=None)`
If you pass an index argument, this method returns a sequence of all the tag names that are associated with the character after that index. If you pass no argument, you get a sequence of all the tag names defined in the text widget.

`.tag_nextrange(`*`tagName`*`, `*`index1`*`, `*`index2`*`=None)`
This method searches a given region for places where a tag named *`tagName`* starts. The region searched starts at index *`index1`* and ends at index *`index2`*. If the *`index2`* argument is omitted, the search goes all the way to the end of the text.

If there is a place in the given region where that tag starts, the method returns a sequence `[`*`i0`*`, i1]`, where *`i0`* is the index of the first tagged character and *`i1`* is the index of the position just after the last tagged character.

If no tag starts are found in the region, the method returns an empty string.

`.tag_prevrange(`*`tagName`*`, `*`index1`*`, `*`index2`*`=None)`
This method searches a given region for places where a tag named *`tagName`* starts. The region searched starts *before* index *`index1`* and ends at index *`index2`*. If the *`index2`* argument is omitted, the search goes all the way to the end of the text.

The return values are as in `.tag_nextrange()`.

`.tag_raise(tagName, `*`aboveThis`*`=None)`
Use this method to change the order of tags in the tag stack (see Section 24.5, "`Text` widget tags" (p. 87), above, for an explanation of the tag stack). If you pass two arguments, the tag with name *`tagName`* is moved to a position just above the tag with name *`aboveThis`*. If you pass only one argument, that tag is moved to the top of the tag stack.

.tag_ranges(*tagName*)
This method finds all the ranges of text in the widget that are tagged with name *tagName*, and returns a sequence [s_0, e_0, s_1, e_1, …], where each s_i is the index just before the first character of the range and e_i is the index just after the last character of the range.

.tag_remove(*tagName*, *index1*, *index2*=None)
Removes the tag named *tagName* from all characters between *index1* and *index2*. If *index2* is omitted, the tag is removed from the single character after *index1*.

.tag_unbind(*tagName*, *sequence*, *funcid*=None)
Remove the event binding for the given *sequence* from the tag named *tagName*. If there are multiple handlers for this sequence and tag, you can remove only one handler by passing it as the third argument.

.window_cget(*index*, *option*)
Returns the value of the given *option* for the embedded widget at the given *index.*

.window_configure(*index*, *option*)
To change the value of options for embedded widget at the given *index*, pass in one or more *option=value* pairs.

If you pass only one argument, you will get back a dictionary defining all the options and their values currently in force for the given widget.

.window_create(*index*, *option*, ...)
This method creates a window where a widget can be embedded within a text widget. There are two ways to provide the embedded widget:

a. you can use pass the widget to the `window` option in this method, or

b. you can define a procedure that will create the widget and pass that procedure as a callback to the `create` option.

Options for `.window_create()` are:

`align`	Specifies how to position the embedded widget vertically in its line, if it isn't as tall as the text on the line. Values include: `align=tk.CENTER` (the default), which centers the widget vertically within the line; `align=tk.TOP`, which places the top of the image at the top of the line; `align=tk.BOTTOM`, which places the bottom of the image at the bottom of the line; and `align=tk.BASELINE`, which aligns the bottom of the image with the text baseline.
`create`	A procedure that will create the embedded widget on demand. This procedure takes no arguments and must create the widget as a child of the text widget and return the widget as its result.
`padx`	Extra space added to the left and right of the widget within the text line. Default is `0`.
`pady`	Extra space added above and below the widget within the text line. Default is `0`.
`stretch`	This option controls what happens when the line is higher than the embedded widget. Normally this option is `0`, meaning that the embedded widget is left at its natural size. If you set `stretch=1`, the widget is stretched vertically to fill the height of the line, and the `align` option is ignored.
`window`	The widget to be embedded. This widget must be a child of the text widget.

.window_names()
Returns a sequence containing the names of all embedded widgets.

.xview(tk.MOVETO, *fraction*)
This method scrolls the text widget horizontally, and is intended for binding to the command option of a related horizontal scrollbar.

This method can be called in two different ways. The first call positions the text at a value given by *fraction*, where 0.0 moves the text to its leftmost position and 1.0 to its rightmost position.

.xview(tk.SCROLL, *n*, *what*)
The second call moves the text left or right: the *what* argument specifies how much to move and can be either `tk.UNITS` or `tk.PAGES`, and *n* tells how many characters or pages to move the text to the right relative to its image (or left, if negative).

.xview_moveto(*fraction*)
This method scrolls the text in the same way as `.xview(tk.MOVETO, fraction)`.

.xview_scroll(*n*, *what*)
Same as `.xview(tk.SCROLL, n, what)`.

.yview(tk.MOVETO, *fraction*)
The vertical scrolling equivalent of `.xview(tk.MOVETO,…)`.

.yview(tk.SCROLL, *n*, *what*)
The vertical scrolling equivalent of `.xview(tk.SCROLL,…)`. When scrolling vertically by `tk.UNITS`, the units are lines.

.yview_moveto(*fraction*)
The vertical scrolling equivalent of `.xview_moveto()`.

.yview_scroll(*n*, *what*)
The vertical scrolling equivalent of `.xview_scroll()`.

25. `Toplevel`: Top-level window methods

A *top-level window* is a window that has an independent existence under the window manager. It is decorated with the window manager's decorations, and can be moved and resized independently. Your application can use any number of top-level windows.

For any widget *w*, you can get to its top-level widget using `w.winfo_toplevel()`.

To create a new top-level window:

```
    w = tk.Toplevel(option, ...)
```

Options include:

Table 34. `Toplevel` window methods

`bg` or `background`	The background color of the window. See Section 5.3, “Colors” (p. 10).
`bd` or `borderwidth`	Border width in pixels; default is `0`. For possible values, see Section 5.1, “Dimensions” (p. 9). See also the `relief` option, below.
`class_`	You can give a `Toplevel` window a “class” name. Such names are matched against the option database, so your application can pick up the user's configuration preferences (such as colors) by class name. For example, you might design a series of pop-ups called “screamers,” and set them all up with `class_='Screamer'`. Then you can put a line in your option database like this:

	`*Screamer*background: red` and then, if you use the `.option_readfile()` method to read your option database, all widgets with that class name will default to a red background. This option is named `class_` because `class` is a reserved word in Python.
`cursor`	The cursor that appears when the mouse is in this window. See Section 5.8, “Cursors” (p. 13).
`height`	Window height; see Section 5.1, “Dimensions” (p. 9).
`highlightbackground`	The color of the focus highlight when the window does not have focus. See Section 53, “Focus: routing keyboard input” (p. 155).
`highlightcolor`	The color of the focus highlight when the window has the focus.
`highlightthickness`	The thickness of the focus highlight. Default is `1`. Set `highlightthickness=0` to suppress display of the focus highlight.
`menu`	To provide this window with a top-level menubar, supply a `Menu` widget as the value of this option. Under MacOS, this menu will appear at the top of the screen when the window is active. Under Windows or Unix, it will appear at the top of the application.
`padx`	Use this option to provide extra space on the left and right sides of the window. The value is a number of pixels.
`pady`	Use this option to provide extra space on the top and bottom sides of the window. The value is a number of pixels.
`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 discussed under Section 5.6, “Relief styles” (p. 12).
`takefocus`	Normally, a top-level window does not get focus. Use `takefocus=True` if you want it to be able to take focus; see Section 53, “Focus: routing keyboard input” (p. 155).
`width`	The desired width of the window; see Section 5.1, “Dimensions” (p. 9).

These methods are available for top-level windows:

`.aspect(`*n_{min}*`, `*d_{min}*`, `*n_{max}*`, `*d_{max}*`)`
Constrain the root window's width:length ratio to the range [$n_{min} / d_{min}, n_{max} / d_{max}$].

`.deiconify()`
If this window is iconified, expand it.

`.geometry(`*`newGeometry`*`=None)`
Set the window geometry. For the form of the argument, see Section 5.10, “Geometry strings” (p. 15). If the argument is omitted, the current geometry string is returned.

`.iconify()`
Iconify the window.

`.lift(aboveThis=None)`
To raise this window to the top of the stacking order in the window manager, call this method with no arguments. You can also raise it to a position in the stacking order just above another `Toplevel` window by passing that window as an argument.

.lower(belowThis=None)
If the argument is omitted, moves the window to the bottom of the stacking order in the window manager. You can also move the window to a position just under some other top-level window by passing that `Toplevel` widget as an argument.

.maxsize(width=None, height=None)
Set the maximum window size. If the arguments are omitted, returns the current `(width, height)`.

.minsize(width=None, height=None)
Set the minimum window size. If the arguments are omitted, returns the current minima as a 2-tuple.

.overrideredirect(flag=None)
If called with a `True` argument, this method sets the override redirect flag, which removes all window manager decorations from the window, so that it cannot be moved, resized, iconified, or closed. If called with a `False` argument, window manager decorations are restored and the override redirect flag is cleared. If called with no argument, it returns the current state of the override redirect flag.

Be sure to call the `.update_idletasks()` method (see Section 26, "Universal widget methods" (p. 97)) before setting this flag. If you call it before entering the main loop, your window will be disabled before it ever appears.

This method may not work on some Unix and MacOS platforms.

.resizable(width=None, height=None)
If is true, allow horizontal resizing. If `height` is true, allow vertical resizing. If the arguments are omitted, returns the current size as a 2-tuple.

.state(newstate=None)
Returns the window's current state, one of:
- `'normal'`: Displayed normally.
- `'iconic'`: Iconified with the `.iconify()` method.
- `'withdrawn'`: Hidden; see the `.withdraw()` method below.

To change the window's state, pass one of the strings above as an argument to the method. For example, to iconify a `Toplevel` instance *T*, use "*T*`.state('iconify') `".

.title(text=None)
Set the window title. If the argument is omitted, returns the current title.

.transient(parent=None)
Make this window a transient window for some *parent* window; the default parent window is this window's parent.

This method is useful for short-lived pop-up dialog windows. A transient window always appears in front of its parent. If the parent window is iconified, the transient is iconified as well.

.withdraw()
Hides the window. Restore it with `.deiconify()` or `.iconify()`.

26. Universal widget methods

The methods are defined below on all widgets. In the descriptions, *w* can be any widget of any type.

w.after(*delay_ms*, callback=None, **args*)
Requests *Tkinter* to call function `callback` with arguments *`args`* after a delay of at least `delay_ms` milliseconds. There is no upper limit to how long it will actually take, but your callback won't be called sooner than you request, and it will be called only once.

This method returns an integer "after identifier" that can be passed to the `.after_cancel()` method if you want to cancel the callback.

If you do not pass a `callback` argument, this method waits `delay_ms` milliseconds, as in the `.sleep()` function of the standard Python `time` module[9].

w.after_cancel(*id*)
Cancels a request for callback set up earlier `.after()`. The *`id`* argument is the result returned by the original *`.after()`* call.

w.after_idle(*func*, **args*)
Requests that *Tkinter* call function *`func`* with arguments *`args`* next time the system is idle, that is, next time there are no events to be processed. The callback will be called only once. If you want your callback to be called again, you must call the `.after_idle` method again.

w.bell()
Makes a noise, usually a beep.

w.bind(*sequence*=None, *func*=None, *add*=None)
This method is used to attach an event binding to a widget. See Section 54, "Events" (p. 157) for the overview of event bindings.

The *`sequence`* argument describes what event we expect, and the *`func`* argument is a function to be called when that event happens to the widget. If there was already a binding for that event for this widget, normally the old callback is replaced with *`func`*, but you can preserve both callbacks by passing `add='+'`.

w.bind_all(*sequence*=None, *func*=None, *add*=None)
Like `.bind()`, but applies to all widgets in the entire application.

w.bind_class(*className*, *sequence*=None, *func*=None, add=None)
Like `.bind()`, but applies to all widgets named *`className`* (e.g., `'Button'`).

w.bindtags(*tagList*=None)
If you call this method, it will return the "binding tags" for the widget as a sequence of strings. A binding tag is the name of a window (starting with `'.'`) or the name of a class (e.g., `'Listbox'`).

You can change the order in which binding levels are called by passing as an argument the sequence of binding tags you want the widget to use.

See Section 54, "Events" (p. 157) for a discussion of binding levels and their relationship to tags.

w.cget(*option*)
Returns the current value of *`option`* as a string. You can also get the value of an option for widget *`w`* as *`w`*`[`*`option`*`]`.

w.clipboard_append(*text*)
Appends the given *`text`* string to the display's clipboard, where cut and pasted strings are stored for all that display's applications.

w.clipboard_clear()
Clears the display's clipboard (see `.clipboard_append()` above).

[9] http://docs.python.org/library/time.html

`w.column_configure()`
See Section 4.2, “Other grid management methods” (p. 7).

`w.config(option=value, ...)`
Same as `.configure()`.

`w.configure(option=value, ...)`
Set the values of one or more options. For the options whose names are Python reserved words (`class`, `from`, `in`), use a trailing underbar: `'class_'`, `'from_'`, `'in_'`.

You can also set the value of an option for widget *w* with the statement

```
w[option] = value
```

If you call the `.config()` method on a widget with no arguments, you'll get a dictionary of all the widget's current options. The keys are the option names (including aliases like `bd` for `borderwidth`). The value for each key is:

- for most entries, a five-tuple: (option name, option database key, option database class, default value, current value); or,
- for alias names (like `'fg'`), a two-tuple: (alias name, equivalent standard name).

`w.destroy()`
Calling *w*`.destroy()` on a widget *w* destroys *w* and all its children.

`w.event_add(virtual, *sequences)`
This method creates a virtual event whose name is given by the *`virtual`* string argument. Each additional argument describes one *sequence*, that is, the description of a physical event. When that event occurs, the new virtual event is triggered.

See Section 54, “Events” (p. 157) for a general description of virtual events.

`w.event_delete(virtual, *sequences)`
Deletes physical events from the virtual event whose name is given by the string *`virtual`*. If all the physical events are removed from a given virtual event, that virtual event won't happen anymore.

`w.event_generate(sequence, **kw)`
This method causes an event to trigger without any external stimulus. The handling of the event is the same as if it had been triggered by an external stimulus. The *`sequence`* argument describes the event to be triggered. You can set values for selected fields in the `Event` object by providing `keyword=`*`value`* arguments, where the *`keyword`* specifies the name of a field in the `Event` object.

See Section 54, “Events” (p. 157) for a full discussion of events.

`w.event_info(virtual=None)`
If you call this method without an argument, you'll get back a sequence of all the currently defined virtual event names.

To retrieve the physical events associated with a virtual event, pass this method the name of the virtual event and you will get back a sequence of the physical *`sequence`* names, or `None` if the given virtual event has never been defined.

`w.focus_displayof()`
Returns the name of the window that currently has input focus on the same display as the widget. If no such window has input focus, returns `None`.

See Section 53, “Focus: routing keyboard input” (p. 155) for a general description of input focus.

w.focus_force()
Force the input focus to the widget. This is impolite. It's better to wait for the window manager to give you the focus. See also `.grab_set_global()` below.

w.focus_get()
Returns the widget that has focus in this application, if any—otherwise returns `None`.

w.focus_lastfor()
This method retrieves the name of the widget that last had the input focus in the top-level window that contains *w*. If none of this top-level's widgets have ever had input focus, it returns the name of the top-level widget. If this application doesn't have the input focus, `.focus_lastfor()` will return the name of the widget that will get the focus next time it comes back to this application.

w.focus_set()
If *w*'s application has the input focus, the focus will jump to *w*. If *w*'s application doesn't have focus, Tk will remember to give it to *w* next the application gets focus.

w.grab_current()
If there is a grab in force for *w*'s display, return its identifier, otherwise return `None`. Refer to Section 54, "Events" (p. 157) for a discussion of grabs.

w.grab_release()
If *w* has a grab in force, release it.

w.grab_set()
Widget *w* grabs all events for *w*'s application. If there was another grab in force, it goes away. See Section 54, "Events" (p. 157) for a discussion of grabs.

w.grab_set_global()
Widget *w* grabs all events for the entire screen. This is considered impolite and should be used only in great need. Any other grab in force goes away. Try to use this awesome power only for the forces of good, and never for the forces of evil, okay?

w.grab_status()
If there is a local grab in force (set by `.grab_set()`), this method returns the string `'local'`. If there is a global grab in force (from `.grab_set_global()`), it returns `'global'`. If no grab is in force, it returns `None`.

w.grid_forget()
See Section 4.2, "Other grid management methods" (p. 7).

w.grid_propagate()
See Section 4.2, "Other grid management methods" (p. 7).

w.grid_remove()
See Section 4.2, "Other grid management methods" (p. 7).

w.image_names()
Returns the names of all the images in *w*'s application as a sequence of strings.

w.keys()
Returns the option names for the widget as a sequence of strings.

w.lift(aboveThis=None)
If the argument is `None`, the window containing *w* is moved to the top of the window stacking order. To move the window just above some `Toplevel` window *w*, pass *w* as an argument.

w.lower(belowThis=None)
If the argument is `None`, the window containing *w* is moved to the bottom of the window stacking order. To move the window just below some `Toplevel` window *w*, pass *w* as an argument.

w.mainloop()

This method must be called, generally after all the static widgets are created, to start processing events. You can leave the main loop with the `.quit()` method (below). You can also call this method inside an event handler to resume the main loop.

w.nametowidget(*name*)

This method returns the actual widget whose path name is *name*. See Section 5.11, "Window names" (p. 16). If the *name* is unknown, this method will raise `KeyError`.

w.option_add(*pattern*, *value*, *priority*=None)

This method adds default option values to the *Tkinter* option database. The *pattern* is a string that specifies a default *value* for options of one or more widgets. The *priority* values are one of:

20	For global default properties of widgets.
40	For default properties of specific applications.
60	For options that come from user files such as their `.Xdefaults` file.
80	For options that are set after the application starts up. This is the default priority level.

Higher-level priorities take precedence over lower-level ones. See Section 27, "Standardizing appearance" (p. 105) for an overview of the option database. The syntax of the *pattern* argument to `.option_add()` is the same as the *option-pattern* part of the resource specification line.

For example, to get the effect of this resource specification line:

```
*Button*font: times 24 bold
```

your application (`self` in this example) might include these lines:

```
    self.bigFont = tkFont.Font(family='times', size=24,
                               weight='bold')
    self.option_add('*Button*font', self.bigFont)
```

Any `Button` widgets created after executing these lines would default to bold Times 24 font (unless overriden by a `font` option to the `Button` constructor).

w.option_clear()

This method removes all options from the *Tkinter* option database. This has the effect of going back to all the default values.

w.option_get(*name*, *classname*)

Use this method to retrieve the current value of an option from the *Tkinter* option database. The first argument is the instance key and the second argument is the class key. If there are any matches, it returns the value of the option that best matches. If there are no matches, it returns `''`.

Refer to Section 27, "Standardizing appearance" (p. 105) for more about how keys are matched with options.

w.option_readfile(*fileName*, *priority*=None)

As a convenience for user configuration, you can designate a named file where users can put their preferred options, using the same format as the `.Xdefaults` file. Then, when your application is initializing, you can pass that file's name to this method, and the options from that file will be added to the database. If the file doesn't exist, or its format is invalid, this method will raise `tk.TclError`.

Refer to Section 27, "Standardizing appearance" (p. 105) for an introduction to the options database and the format of option files.

w.register(*function*)
This method creates a Tcl wrapper around a Python *function*, and returns the Tcl wrapper name as a string. For an example of the usage of this method, see Section 10.2, "Adding validation to an `Entry` widget" (p. 45).

w.quit()
This method exits the main loop. See `.mainloop()`, above, for a discussion of main loops.

w.rowconfigure()
See Section 4.2, "Other grid management methods" (p. 7).

w.selection_clear()
If *w* currently has a selection (such as a highlighted segment of text in an entry widget), clear that selection.

w.selection_get()
If *w* currently has a selection, this method returns the selected text. If there is no selection, it raises `tk.TclError`.

w.selection_own()
Make *w* the owner of the selection in *w*'s display, stealing it from the previous owner, if any.

w.selection_own_get()
Returns the widget that currently owns the selection in *w*'s display. Raises `tk.TclError` if there is no such selection.

w.tk_focusFollowsMouse()
Normally, the input focus cycles through a sequence of widgets determined by their hierarchy and creation order; see Section 53, "Focus: routing keyboard input" (p. 155). You can, instead, tell *Tkinter* to force the focus to be wherever the mouse is; just call this method. There is no easy way to undo it, however.

w.tk_focusNext()
Returns the widget that follows *w* in the focus traversal sequence. Refer to Section 53, "Focus: routing keyboard input" (p. 155) for a discussion of focus traversal.

w.tk_focusPrev()
Returns the widget that precedes *w* in the focus traversal sequence.

w.unbind(*sequence*, *funcid*=None)
This method deletes bindings on *w* for the event described by *sequence*. If the second argument is a callback bound to that sequence, that callback is removed and the rest, if any, are left in place. If the second argument is omitted, all bindings are deleted.

See Section 54, "Events" (p. 157), below, for a general discussion of event bindings.

w.unbind_all(*sequence*)
Deletes all event bindings throughout the application for the event described by the given *sequence*.

w.unbind_class(*className*, *sequence*)
Like `.unbind()`, but applies to all widgets named *className* (e.g., `'Entry'` or `'Listbox'`).

w.update()
This method forces the updating of the display. It should be used only if you know what you're doing, since it can lead to unpredictable behavior or looping. It should never be called from an event callback or a function that is called from an event callback.

`w.update_idletasks()`
Some tasks in updating the display, such as resizing and redrawing widgets, are called *idle tasks* because they are usually deferred until the application has finished handling events and has gone back to the main loop to wait for new events.

If you want to force the display to be updated before the application next idles, call the `w.update_idletasks()` method on any widget.

`w.wait_variable(v)`
Waits until the value of variable `v` is set, even if the value does not change. This method enters a local wait loop, so it does not block the rest of the application.

`w.wait_visibility(w)`
Wait until widget `w` (typically a `Toplevel`) is visible.

`w.wait_window(w)`
Wait until window `w` is destroyed.

`w.winfo_children()`
Returns a list of all `w`'s children, in their stacking order from lowest (bottom) to highest (top).

`w.winfo_class()`
Returns `w`'s class name (e.g., `'Button'`).

`w.winfo_containing(rootX, rootY, displayof=0)`
This method is used to find the window that contains point (`rootX`, `rootY`). If the `displayof` option is false, the coordinates are relative to the application's root window; if true, the coordinates are treated as relative to the top-level window that contains `w`. If the specified point is in one of the application's top-level window, this method returns that window; otherwise it returns `None`.

`w.winfo_depth()`
Returns the number of bits per pixel in `w`'s display.

`w.winfo_fpixels(number)`
For any dimension `number` (see Section 5.1, "Dimensions" (p. 9)), this method returns that distance in pixels on `w`'s display, as a number of type `float`.

`w.winfo_geometry()`
Returns the geometry string describing the size and on-screen location of `w`. See Section 5.10, "Geometry strings" (p. 15).

Warning

The geometry is not accurate until the application has updated its idle tasks. In particular, all geometries are initially `'1x1+0+0'` until the widgets and geometry manager have negotiated their sizes and positions. See the `.update_idletasks()` method, above, in this section to see how to insure that the widget's geometry is up to date.

`w.winfo_height()`
Returns the current height of `w` in pixels. See the remarks on geometry updating under `.winfo_geometry()`, above. You may prefer to use `.winfo_reqheight()`, described below, which is always up to date.

`w.winfo_id()`
Returns an integer that uniquely identifies `w` within its top-level window. You will need this for the `.winfo_pathname()` method, below.

w.winfo_ismapped()
This method returns true if *w* is mapped, false otherwise. A widget is mapped if it has been gridded (or placed or packed, if you are using one of the other geometry managers) into its parent, and if its parent is mapped, and so on up to the top-level window.

w.winfo_manager()
If *w* has not been gridded (or placed via one of the other geometry managers), this method returns an empty string. If *w* has been gridded or otherwise placed, it returns a string naming the geometry manager for *w*: this value will be one of `'grid'`, `'pack'`, `'place'`, `'canvas'`, or `'text'`.

w.winfo_name()
This method returns *w*'s name relative to its parent. See Section 5.11, "Window names" (p. 16). Also see `.winfo_pathname()`, below, to find out how to obtain a widget's path name.

w.winfo_parent()
Returns *w*'s parent's path name, or an empty string if *w* is a top-level window. See Section 5.11, "Window names" (p. 16) above, for more on widget path names.

w.winfo_pathname(*id*, *displayof*=0)
If the *displayof* argument is false, returns the window path name of the widget with unique identifier *id* in the application's main window. If *displayof* is true, the *id* number specifies a widget in the same top-level window as *w*. See Section 5.11, "Window names" (p. 16) for a discussion of widget path names.

w.winfo_pixels(number)
For any dimension *number* (see Dimensions, above), this method returns that distance in pixels on *w*'s display, as an integer.

w.winfo_pointerx()
Returns the same value as the *x* coordinate returned by `.winfo_pointerxy()`.

w.winfo_pointerxy()
Returns a tuple `(x, y)` containing the coordinates of the mouse pointer relative to *w*'s root window. If the mouse pointer isn't on the same screen, returns `(-1, -1)`.

w.winfo_pointery()
Returns the same value as the *y* coordinate returned by `.winfo_pointerxy()`.

w.winfo_reqheight()
These methods return the requested height of widget *w*. This is the minimum height necessary so that all of *w*'s contents have the room they need. The actual height may be different due to negotiations with the geometry manager.

w.winfo_reqwidth()
Returns the requested width of widget *w*, the minimum width necessary to contain *w*. As with `.winfo_reqheight()`, the actual width may be different due to negotiations with the geometry manager.

w.winfo_rgb(color)
For any given color, this method returns the equivalent red-green-blue color specification as a 3-tuple `(r, g, b)`, where each number is an integer in the range [0, 65536). For example, if the `color` is `'green'`, this method returns the 3-tuple `(0, 65535, 0)`.

For more on specifying colors, see Section 5.3, "Colors" (p. 10).

w.winfo_rootx()
Returns the *x* coordinates of the left-hand side of *w*'s root window relative to *w*'s parent.

If *w* has a border, this is the outer edge of the border.

w.winfo_rooty()
Returns the *y* coordinate of the top side of *w*'s root window relative to *w*'s parent.

If *w* has a border, this is the top edge of the border.

w.winfo_screenheight()
Returns the height of the screen in pixels.

w.winfo_screenmmheight()
Returns the height of the screen in millimeters.

w.winfo_screenmmwidth()
Returns the width of the screen in millimeters.

w.winfo_screenvisual()
Returns a string that describes the display's method of color rendition. This is usually `'truecolor'` for 16- or 24-bit displays, `'pseudocolor'` for 256-color displays.

w.winfo_screenwidth()
Returns the width of the screen in pixels.

w.winfo_toplevel()
Returns the top-level window containing *w*. That window supports all the methods on `Toplevel` widgets; see Section 25, “`Toplevel`: Top-level window methods” (p. 95).

w.winfo_viewable()
A predicate that returns a `True` value if *w* is viewable, that is, if it and all its ancestors in the same `Toplevel` are mapped.

w.winfo_width()
Returns the current width of *w* in pixels. See the remarks on geometry updating under `.winfo_geometry()`, above. You may prefer to use the `.winfo_reqwidth()` method, described above; it is always up to date.

w.winfo_x()
Returns the *x* coordinate of the left side of *w* relative to its parent. If *w* has a border, this is the outer edge of the border.

w.winfo_y()
Returns the *y* coordinate of the top side of *w* relative to its parent. If *w* has a border, this is the outer edge of the border.

27. Standardizing appearance and the option database

It's easy to apply colors, fonts, and other options to the widgets when you create them. However,

- if you want a lot of widgets to have the same background color or font, it's tedious to specify each option each time, and
- it's nice to let the user override your choices with their favorite color schemes, fonts, and other choices.

Accordingly, we use the idea of an *option database* to set up default option values.

- Your application can specify a file (such as the standard `.Xdefaults` file used by the X Window System) that contains the user's preferences. You can set up your application to read the file and tell *Tkinter* to use those defaults. See the section on the `.option_readfile()` method, above, in the section on Section 26, “Universal widget methods” (p. 97), for the structure of this file.

- Your application can directly specify defaults for one or many types of widgets by using the `.option_add()` method; see this method under Section 26, "Universal widget methods" (p. 97).

Before we discuss how options are set, consider the problem of customizing the appearance of GUIs in general. We could give every widget in the application a name, and then ask the user to specify every property of every name. But this is cumbersome, and would also make the application hard to reconfigure—if the designer adds new widgets, the user would have to describe every property of every new widget.

So, the option database allows the programmer and the user to specify *general patterns* describing which widgets to configure.

These patterns operate on the names of the widgets, but widgets are named using *two* parallel naming schemes:

a. Every widget has a *class name*. By default, the class name is the same as the class constructor: `'Button'` for buttons, `'Frame'` for a frame, and so on. However, you can create new classes of widgets, usually inheriting from the `Frame` class, and give them new names of your own creation. See Section 27.1, "How to name a widget class" (p. 106) for details.

b. You can also give any widget an *instance name*. The default name of a widget is usually a meaningless number (see Section 5.11, "Window names" (p. 16)). However, as with widget classes, you can assign a name to any widget. See the section Section 27.2, "How to name a widget instance" (p. 107) for details.

Every widget in every application therefore has two hierarchies of names—the class name hierarchy and the instance name hierarchy. For example, a button embedded in a text widget which is itself embedded in a frame would have the class hierarchy `Frame.Text.Button`. It might also have an instance hierarchy something like `.mainFrame.messageText.panicButton` if you so named all the instances. The initial dot stands for the root window; see Section 5.11, "Window names" (p. 16) for more information on window path names.

The option database mechanism can make use of either class names or instance names in defining options, so you can make options apply to whole classes (e.g., all buttons have a blue background) or to specific instances (e.g., the Panic Button has red letters on it). After we look at how to name classes and instances, in Section 27.3, "Resource specification lines" (p. 107), we'll discuss how the options database really works.

27.1. How to name a widget class

For example, suppose that `Jukebox` is a new widget class that you have created. It's probably best to have new widget classes inherit from the `Frame` class, so to *Tkinter* it acts like a frame, and you can arrange other widgets such as labels, entries, and buttons inside it.

You set the new widget's class name by passing the name as the `class_` option to the parent constructor in your new class's constructor. Here is a fragment of the code that defines the new class:

```
class Jukebox(tk.Frame):
    def __init__(self, master):
        '''Constructor for the Jukebox class
        '''
        tk.Frame.__init__(self, master, class_='Jukebox')
        self.__createWidgets()
        ...
```

27.2. How to name a widget instance

To give an instance name to a specific widget in your application, set that widget's `name` option to a string containing the name.

Here's an example of an instance name. Suppose you are creating several buttons in an application, and you want one of the buttons to have an instance name of `panicButton`. Your call to the constructor might look like this:

```
    self.panic = tk.Button(self, name='panicButton', text='Panic', ...)
```

27.3. Resource specification lines

Each line in an option file specifies the value of one or more options in one or more applications and has one of these formats:

```
app option-pattern: value
option-pattern: value
```

The first form sets options only when the name of the application matches *app*; the second form sets options for all applications.

For example, if your application is called *xparrot*, a line of the form

```
xparrot*background: LimeGreen
```

sets all `background` options in the *xparrot* application to lime green. (Use the `-name` option on the command line when launching your application to set the name to `'xparrot'`.)

The *option-pattern* part has this syntax:

```
{{*|.}name}...option
```

That is, each *option-pattern* is a list of zero or more names, each of which is preceded by an asterisk or period. The last name in the series is the name of the option you are setting. Each of the rest of the names can be either:

- the name of a widget *class* (capitalized), or
- the name of an *instance* (lowercased).

The way the option patterns work is a little complicated. Let's start with a simple example:

```
*font: times 24
```

This line says that all `font` options should default to 24-point Times. The * is called the *loose binding* symbol, and means that this option pattern applies to any `font` option anywhere in any application. Compare this example:

```
*Listbox.font: lucidatypewriter 14
```

The period between `Listbox` and `font` is called the *tight binding* symbol, and it means that this rule applies only to `font` options for widgets in class `Listbox`.

As another example, suppose your *xparrot* application has instances of widgets of class `Jukebox`. In order to set up a default background color for all widgets of that class `Jukebox`, you could put a line in your options file like this:

```
xparrot*Jukebox*background: PapayaWhip
```

The loose-binding (*) symbol between `Jukebox` and `background` makes this rule apply to any `background` option of any widget anywhere inside a `Jukebox`. Compare this option line:

```
xparrot*Jukebox.background: NavajoWhite
```

This rule will apply to the frame constituting the `Jukebox` widget itself, but because of the tight-binding symbol it will not apply to widgets that are inside the `Jukebox` widget.

In the next section we'll talk about how *Tkinter* figures out exactly which option value to use if there are multiple resource specification lines that apply.

27.4. Rules for resource matching

When you are creating a widget, and you don't specify a value for some option, and two or more resource specifications apply to that option, the most specific one applies.

For example, suppose your options file has these two lines:

```
*background: LimeGreen
*Listbox*background: FloralWhite
```

Both specifications apply to the `background` option in a `Listbox` widget, but the second one is more specific, so it will win.

In general, the names in a resource specification are a sequence $n_1, n_2, n_3, ..., o$ where each n_i is a class or instance name. The class names are ordered from the highest to the lowest level, and o is the name of an option.

However, when *Tkinter* is creating a widget, all it has is the class name and the instance name of that widget.

Here are the precedence rules for resource specifications:

1. The name of the option must match the *o* part of the *option-pattern*. For example, if the rule is

   ```
   xparrot*indicatoron: 0
   ```

   this will match only options named `indicatoron`.

2. The tight-binding operator (`.`) is more specific than the loose-binding operator (*). For example, a line for `*Button.font` is more specific than a line for `*Button*font`.

3. References to instances are more specific than references to classes. For example, if you have a button whose instance name is `panicButton`, a rule for `*panicButton*font` is more specific than a rule for `*Button*font`.

4. A rule with more levels is more specific. For example, a rule for `*Button*font` is more specific than a rule for `*font`.

5. If two rules have same number of levels, names earlier in the list are more specific than later names. For example, a rule for `xparrot*font` is more specific than a rule for `*Button*font`.

28. *ttk*: Themed widgets

Starting with Tk 8.5, the *ttk* module became available. This module replaces much (but not all) of the original *Tkinter* machinery. Use this module to gain these advantages:

- *Platform-specific appearance*. In releases before Tk 8.5, one of the commonest complaints about Tk applications was that they did not conform to the style of the various platforms.

  The *ttk* module allows you to write your application in a generic way, yet your application can look like a Windows application under Windows, like a MacOS app under MacOS, and so on, without any change to your program.

  Each possible different appearance is represented by a named *ttk theme*. For example, the `classic` theme gives you the appearance of the original *Tkinter* widgets described in the previous sections.

- *Simplification and generalization of state-specific widget behavior*. In the basic *Tkinter* world, there are a lot of widget options that specify how the widget should look or behave depending on various conditions.

  For example, the `tk.Button` widget has several different options that control the foreground (text) color.
  - The `activeforeground` color option applies when the cursor is over the button.
  - The `disabledforeground` color is used when the widget is disabled.
  - The widget will have the `foreground` color when the other conditions don't apply.

  The *ttk* module collapses a lot of these special cases into a simple two-part system:
  - Every widget has a number of different states, and each state can be turned on or off independently of the others. Examples of states are: `disabled`, `active`, and `focus`.
  - You can set up a *style map* that specifies that certain options will be set to certain values depending on some state or some combination of the widget's states.

To use *ttk*, you will need to know these things.

- Section 28.1, "Importing *ttk*" (p. 109): Setting up your program to use *ttk*.
- Section 28.2, "The *ttk* widget set" (p. 110): The new and replaced *ttk* widgets.
- Section 47, "Customizing and creating *ttk* themes and styles" (p. 146).

28.1. Importing *ttk*

There are different ways to import the *ttk* module.

- If you prefer that all the widgets and other features of *Tkinter* and *ttk* be in your global namespace, use this form of import:

```
from Tkinter import *
from ttk import *
```

  It is important to do these two imports in this order, so that all the widget types from *ttk* replace the equivalent widgets from *Tkinter*. For example, all your `Button` widgets will come from *ttk* and not *Tkinter*.

- In more complex applications, where you are using more than one imported module, it can greatly improve the readability of your code if you practice *safe namespace hygiene*: import all your modules using the "`import modulename`" syntax. This requires just a bit more typing, but it has the great advantage that you can look at a reference to something and tell where it came from.

  We recommend this form of import:

```
import ttk
```

  So after this import, `ttk.Label` is the `Label` widget constructor, `ttk.Button` is a `Button`, and so on.

If you need to refer to items from the *Tkinter* module, it is available as `ttk.Tkinter`. For example, the anchor code for "northeast" is `ttk.Tkinter.NE`.

You may instead import *Tkinter* separately in this way:

```
import Tkinter as tk
```

After this form of import, the code for "northeast" is `tk.NE`.

28.2. The *ttk* widget set

The *ttk* module contains different versions of most of the standard *Tkinter* widgets and a few new ones. These widgets replace the ones from *Tkinter* of the same name:

- Section 29, "*ttk*`.Button`" (p. 110).
- Section 30, "*ttk*`.Checkbutton`" (p. 112).
- Section 32, "*ttk*`.Entry`" (p. 116).
- Section 33, "*ttk*`.Frame`" (p. 118).
- Section 34, "*ttk*`.Label`" (p. 119).
- Section 35, "*ttk*`.LabelFrame`" (p. 122).
- Section 36, "*ttk*`.Menubutton`" (p. 124).
- Section 38, "*ttk*`.PanedWindow`" (p. 129).
- Section 40, "*ttk*`.Radiobutton`" (p. 131).
- Section 41, "*ttk*`.Scale`" (p. 133).
- Section 42, "*ttk*`.Scrollbar`" (p. 135).

These widgets are new, and specific to *ttk*:

- Section 31, "*ttk*`.Combobox`" (p. 115).
- Section 37, "*ttk*`.Notebook`" (p. 126).
- Section 39, "*ttk*`.Progressbar`" (p. 130).
- Section 43, "*ttk*`.Separator`" (p. 137).

29. *ttk*`.Button`

This widget is the *ttk* version of Section 7, "The `Button` widget" (p. 18). To create a *ttk*`.Button` widget:

```
    w = ttk.Button(parent, option=value, ...)
```

Here are the options for the *ttk*`.Button` widget. Compare these to the *Tkinter* version discussed in Section 7, "The `Button` widget" (p. 18).

Table 35. *ttk*`.Button` options

`class_`	The widget class name. This may be specified when the widget is created, but cannot be changed later. For an explanation of widget classes, see Section 27, "Standardizing appearance" (p. 105).
`command`	A function to be called when the button is pressed.
`compound`	If you provide both `image` and `text` options, the `compound` option specifies the position of the image relative to the text. The value may be `tk.TOP` (image above text), `tk.BOTTOM` (image below text), `tk.LEFT` (image to the left of the text), or `tk.RIGHT` (image to the right of the text).

	When you provide both `image` and `text` options but don't specify a `compound` option, the image will appear and the text will not.
`cursor`	The cursor that will appear when the mouse is over the button; see Section 5.8, "Cursors" (p. 13).
`image`	An image to appear on the button; see Section 5.9, "Images" (p. 14).
`style`	The style to be used in rendering this button; see Section 49, "Using and customizing *ttk* styles" (p. 147).
`takefocus`	By default, a *ttk*`.Button` will be included in focus traversal; see Section 53, "Focus: routing keyboard input" (p. 155). To remove the widget from focus traversal, use `takefocus=False`.
`text`	The text to appear on the button, as a string.
`textvariable`	A variable that controls the text that appears on the button; see Section 52, "Control variables: the values behind the widgets" (p. 153).
`underline`	If this option has a nonnegative value *n*, an underline will appear under the character at position *n*.
`width`	If the label is text, this option specifies the absolute width of the text area on the button, as a number of characters; the actual width is that number multiplied by the average width of a character in the current font. For image labels, this option is ignored. The option may also be configured in a style.

These options of the *Tkinter* `Button` widget are *not* supported by the *ttk*`.Button` constructor:

Table 36. *Tkinter* `Button` options not in *ttk*`.Button`

`activebackground`	Use a style map to control the `background` option; see Section 50.2, "*ttk* style maps: dynamic appearance changes" (p. 151).
`activeforeground`	Use a style map to control the `foreground` option.
`anchor`	Configure this option using a style; see Section 49, "Using and customizing *ttk* styles" (p. 147). Use this option to specify the position of the text when the `width` option allocates extra horizontal space. For example, if you specify options `width=20` and `compound=tk.RIGHT` on a button that displays both text and and image, and a style that specifies `anchor=tk.E` (east), the image will be at the right-hand end of the twenty-character space, with the text just to its left. When the button displays an image but no text, this option is ignored.
`background` or `bg`	Configure the `background` option using a style. The `bg` abbreviation is not supported.
`bitmap`	Not supported.
`borderwidth` or `bd`	Configure the `borderwidth` option using a style. The `bd` abbreviation is not supported.
`cursor`	The cursor that will appear when the mouse is over the checkbutton; see Section 5.8, "Cursors" (p. 13).
`default`	Not supported; see Section 50.2, "*ttk* style maps: dynamic appearance changes" (p. 151).
`disabledforeground`	Use a style map for the `foreground` option; see Section 50.2, "*ttk* style maps: dynamic appearance changes" (p. 151).

`font`	Configure this option using a style.
`foreground` or `fg`	Configure this option using a style.
`height`	Not supported.
`highlightback-ground`	To control the color of the focus highlight when the button does not have focus, use a style map to control the `highlightcolor` option; see Section 50.2, "*ttk* style maps: dynamic appearance changes" (p. 151).
`highlightcolor`	You may specify the default focus highlight color by setting this option in a style. You may also control the focus highlight color using a style map.
`highlightthickness`	Configure this option using a style. This option may not work in all themes.
`justify`	If the `text` contains newline (`'\n'`) characters, the text will occupy multiple lines on the button. The `justify` option controls how each line is positioned horizontally. Configure this option using a style; values may be `tk.LEFT`, `tk.CENTER`, or `tk.RIGHT` for lines that are left-aligned, centered, or right-aligned, respectively.
`overrelief`	Use a style map to control the `relief` option; see Section 50.2, "*ttk* style maps: dynamic appearance changes" (p. 151).
`padx`	Not supported.
`pady`	Not supported.
`relief`	Configure this option using a style; see Section 49, "Using and customizing *ttk* styles" (p. 147).
`repeatdelay`	Not supported.
`repeatinterval`	Not supported.
`state`	In *ttk*, there is no option with this name. The state mechanism has been generalized; see Section 50.2, "*ttk* style maps: dynamic appearance changes" (p. 151).
`wraplength`	If you use a style with this option set to some dimensions, the `text` will be sliced into pieces no longer than that dimension.

Methods on a *ttk*`.Button` include all those described in Section 46, "Methods common to all *ttk* widgets" (p. 145), plus:

`.invoke()`
Calls the button's `command` callback, and returns what that function returns. Has no effect if the button is disabled or there is no callback.

The `.flash()` method of `Tkinter.Button` is not supported by the *ttk*`.Button` widget.

30. *ttk*`.Checkbutton`

This widget is the *ttk* version of Section 9, "The `Checkbutton` widget" (p. 38). To create a *ttk*`.Checkbutton` widget as the child of a given *`parent`* widget:

```
w = ttk.Checkbutton(parent, option=value, ...)
```

Here are the options for the *ttk*`.Checkbutton` widget. Compare these to the *Tkinter* version discussed in Section 7, "The `Button` widget" (p. 18).

Table 37. *ttk*.Checkbutton options

class_	The widget class name. This may be specified when the widget is created, but cannot be changed later. For an explanation of widget classes, see Section 27, "Standardizing appearance" (p. 105).
command	A function to be called whenever the state of this checkbutton changes.
compound	This option specifies the relative position of the image relative to the text when you specify both. The value may be tk.TOP (image above text), tk.BOTTOM (image below text), tk.LEFT (image to the left of the text), or tk.RIGHT (image to the right of the text). If you provide both image and text options but do not specify a value for compound, only the image will appear.
cursor	The cursor that will appear when the mouse is over the checkbutton; see Section 5.8, "Cursors" (p. 13).
image	An image to appear on the checkbutton; see Section 5.9, "Images" (p. 14).
offvalue	By default, when a checkbutton is in the off (unchecked) state, the value of the associated variable is 0. You can use the offvalue option to specify a different value for the off state.
onvalue	By default, when a checkbutton is in the on (checked) state, the value of the associated variable is 1. You can use the onvalue option to specify a different value for the on state.
style	The style to be used in rendering this checkbutton; see Section 49, "Using and customizing *ttk* styles" (p. 147).
takefocus	By default, a *ttk*.Checkbutton will be included in focus traversal; see Section 53, "Focus: routing keyboard input" (p. 155). To remove the widget from focus traversal, use takefocus=False.
text	The text to appear on the checkbutton, as a string.
textvariable	A variable that controls the text that appears on the checkbutton; see Section 52, "Control variables: the values behind the widgets" (p. 153).
underline	If this option has a nonnegative value *n*, an underline will appear under the text character at position *n*.
variable	A control variable that tracks the current state of the checkbutton; see Section 52, "Control variables: the values behind the widgets" (p. 153). Normally you will use an IntVar here, and the off and on values are 0 and 1, respectively. However, you may use a different control variable type, and specify the offvalue and onvalue options using values of that type.
width	Use this option to specify a fixed width or a minimum width. The value is specified in characters; a positive value sets a fixed width of that many average characters, while a negative width sets a minimum width. For example, if an average character in the selected font is 10 pixels wide, option width=8 will make the text label exactly 80 pixels wide; option width=-8 will use 80 pixels or the length of the text, whichever is larger. You may also specify a width value in an associated style. If values are specified both in the widget constructor call and in the style, the former takes priority.

These options of the *Tkinter* Checkbutton widget are *not* supported by the *ttk*.Checkbutton widget constructor:

Table 38. *Tkinter* `Checkbutton` options not in *ttk*`.Checkbutton`

`activebackground`	Use a style map to control the `background` option; see Section 50.2, "*ttk* style maps: dynamic appearance changes" (p. 151).
`activeforeground`	Use a style map to control the `foreground` option.
`anchor`	Configure this option using a style; see Section 49, "Using and customizing *ttk* styles" (p. 147). Use this option to specify the position of the text when the `width` option allocates extra horizontal space. For example, if you specify options `width=20` and `compound=tk.TOP` on a checkbutton that displays both text and and image, and a style that specifies `anchor=tk.E` (east), the image will be at the right-hand end of the twenty-character space, with the text just below it. When a checkbutton displays an image but no text, this option is ignored.
`background` or `bg`	Configure the `background` option using a style. The `bg` abbreviation is not supported.
`bitmap`	Not supported.
`borderwidth` or `bd`	Configure this option using a style.
`disabledforeground`	Use a style map for the `foreground` option; see Section 50.2, "*ttk* style maps: dynamic appearance changes" (p. 151).
`font`	Configure this option using a style.
`foreground` or `fg`	Configure this option using a style.
`height`	Not supported.
`highlightbackground`	To control the color of the focus highlight when the checkbutton does not have focus, use a style map to control the `highlightcolor` option; see Section 50.2, "*ttk* style maps: dynamic appearance changes" (p. 151).
`highlightcolor`	You may specify the default focus highlight color by setting this option in a style. You may also control the focus highlight color using a style map.
`highlightthickness`	Configure this option using a style. This option may not work in all themes.
`indicatoron`	Not supported.
`justify`	Controls how multiple lines are positioned horizontally relative to each other. Configure this option using a style; values may be `tk.LEFT`, `tk.CENTER`, or `tk.RIGHT` for left-aligned, centered, or right-aligned, respectively.
`offrelief`	Not supported.
`overrelief`	Use a style map to control the `relief` option; see Section 50.2, "*ttk* style maps: dynamic appearance changes" (p. 151).
`padx`	Not supported.
`pady`	Not supported.
`relief`	Use a style map to control the `relief` option; see Section 50.2, "*ttk* style maps: dynamic appearance changes" (p. 151).
`selectcolor`	Not supported.
`selectimage`	Not supported.

`state`	In *ttk*, there is no option with this name. The state mechanism has been generalized; see Section 50.2, "*ttk* style maps: dynamic appearance changes" (p. 151).
`wraplength`	If you use a style that has this option set to some dimension, the `text` will be sliced into pieces no longer than that dimension.

Methods on a *ttk*`.Checkbutton` include all those described in Section 46, "Methods common to all *ttk* widgets" (p. 145), plus:

`.invoke()`
This method toggles the state of the checkbutton. If there is a `command` callback, it calls that callback, and returns whatever value the callback returned.

Not supported are the following methods of the *Tkinter* `Checkbutton` widget: `.deselect()`, `.flash()`, `.select()`, and `.toggle()`. To change the state of a checkbutton through program control, use the `.set()` method of the associated control `variable`.

31. *ttk*`.Combobox`

This widget is a combination of an `Entry` and a drop-down menu. In your application, you will see the usual text entry area, with a downward-pointing arrow. When the user clicks on the arrow, a drop-down menu appears. If the user clicks on one, that choice replaces the current contents of the entry. However, the user may still type text directly into the entry (when it has focus), or edit the current text.

To create a *ttk*`.Combobox` widget as the child of a given *`parent`* widget:

```
w = ttk.Combobox(parent, option=value, ...)
```

Options:

Table 39. *ttk*`.Combobox` options

`class_`	The widget class name. This may be specified when the widget is created, but cannot be changed later. For an explanation of widget classes, see Section 27, "Standardizing appearance" (p. 105).
`cursor`	The cursor that will appear when the mouse is over the checkbutton; see Section 5.8, "Cursors" (p. 13).
`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`.
`height`	Use this option to specify a maximum number of rows that will appear in the drop-down menu; the default is 20. If there are more `values` than this number, the drop-down menu will automatically include a vertical scrollbar.
`justify`	This option specifies how the text will be positioned within the entry area when it does not completely fill the area. Values may be `tk.LEFT` to left-justify; `tk.CENTER` to center; or `tk.RIGHT` to right-justify.
`postcommand`	You may use this option to supply a callback function that will be invoked when the user clicks on the down-arrow. This callback may change the `values` option; if so, the changes will appear in the drop-down menu.
`style`	The style to be used in rendering this checkbutton; see Section 49, "Using and customizing *ttk* styles" (p. 147).

`takefocus`	By default, a *ttk*`.Checkbutton` will be included in focus traversal; see Section 53, “Focus: routing keyboard input” (p. 155). To remove the widget from focus traversal, use `takefocus=False`.
`textvariable`	A variable that controls the text that appears in the entry area; see Section 52, “Control variables: the values behind the widgets” (p. 153).
`validate`	You may use this option to request dynamic validation of the widget's text content. See Section 10.2, “Adding validation to an `Entry` widget” (p. 45).
`validatecommand`	You may use this option to specify a callback function that dynamically validates the widget's text content. See Section 10.2, “Adding validation to an `Entry` widget” (p. 45).
`values`	The choices that will appear in the drop-down menu, as a sequence of strings.
`width`	This option specifies the width of the entry area as a number of characters. The actual width will be this number times the average width of a character in the effective font. The default value is 20.
`xscrollcommand`	If the widget has an associated horizontal scrollbar, set this option to the `.set` method of that scrollbar.

Methods on a *ttk*`.Combobox` include all those described in Section 46, “Methods common to all *ttk* widgets” (p. 145), plus all the methods on the *Tkinter* widget described in Section 10, “The `Entry` widget” (p. 41), plus:

`.current([`*`index`*`])`
: To select one of the elements of the `values` option, pass the index of that element as the argument to this method. If you do not supply an argument, the returned value is the index of the current entry text in the `values` list, or -1 if the current entry text is not in the `values` list.

`.set(`*`value`*`)`
: Set the current text in the widget to *`value`*.

The states of a *ttk*`.Combobox` widget affect its operation. To interrogate or change states, see the `.instate()` and `.state()` methods in Section 46, “Methods common to all *ttk* widgets” (p. 145).

- If the widget is in the `disabled` state, no user action will change the contents.
- If the widget is in the `!disabled` state and also the `readonly` state, the user may change the contents by using the drop-down menu, but may not directly edit the contents.

32. *ttk*`.Entry`

The purpose of an `Entry` widget is to allow the user to enter or edit a single line of text. This is the *ttk* version of Section 10, “The `Entry` widget” (p. 41).

To create a *ttk*`.Entry` widget as the child of a given *`parent`* widget:

```
    w = ttk.Entry(parent, option=value, ...)
```

Options:

Table 40. *ttk*`.Entry` options

`class_`	The widget class name. This may be specified when the widget is created, but cannot be changed later. For an explanation of widget classes, see Section 27, “Standardizing appearance” (p. 105).

`cursor`	The cursor that will appear when the mouse is over the checkbutton; see Section 5.8, “Cursors” (p. 13).
`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`.
`font`	Use this option to specify the font of the text that will appear in the widget; see Section 5.4, “Type fonts” (p. 10). For reasons that are unclear to the author, this option cannot be specified with a style.
`invalidcommand`	You may set this option to a callback function that will be called whenever validation fails (that is, when the `validatecommand` returns a 0). See Section 10.2, “Adding validation to an `Entry` widget” (p. 45).
`justify`	This option specifies how the text will be positioned within the entry area when it does not completely fill the area. Values may be `tk.LEFT` to left-justify; `tk.CENTER` to center; or `tk.RIGHT` to right-justify.
`show`	To protect fields such as passwords from being visible on the screen, set this option to a string, whose first character will be substituted for each of the actual characters in the field. For example, if the field contains “ `sesame`” but you have specified `show='*'`, the field will appear as “`******`”.
`style`	The style to be used in rendering this checkbutton; see Section 49, “Using and customizing *ttk* styles” (p. 147).
`takefocus`	By default, a *ttk*`.Checkbutton` will be included in focus traversal; see Section 53, “Focus: routing keyboard input” (p. 155). To remove the widget from focus traversal, use `takefocus=False`.
`textvariable`	A variable that controls the text that appears in the entry area; see Section 52, “Control variables: the values behind the widgets” (p. 153).
`validate`	You may use this option to specify a callback function that dynamically validates the widget's text content. See Section 10.2, “Adding validation to an `Entry` widget” (p. 45).
`validatecommand`	See Section 10.2, “Adding validation to an `Entry` widget” (p. 45).
`width`	This option specifies the width of the entry area as a number of characters. The actual width will be this number times the average width of a character in the effective font. The default value is 20.
`xscrollcommand`	If the widget has an associated horizontal scrollbar, set this option to the `.set` method of that scrollbar.

These options of the *Tkinter* `Entry` widget are *not* supported by the *ttk*`.Entry` widget constructor:

Table 41. *Tkinter* `Entry` options not in *ttk*`.Entry`

`background` or `bg`	Configure the `background` option using a style; see Section 47, “Customizing and creating *ttk* themes and styles” (p. 146). The `bg` abbreviation is not supported.
`borderwidth` or `bd`	Configure this option using a style.
`disabledbackground`	Use a style map for the `background` option; see Section 50.2, “*ttk* style maps: dynamic appearance changes” (p. 151).
`disabledforeground`	Use a style map for the `foreground` option; see Section 50.2, “*ttk* style maps: dynamic appearance changes” (p. 151).
`foreground` or `fg`	Configure this option using a style.

highlightbackground	To control the color of the focus highlight when the checkbutton does not have focus, use a style map to control the highlightcolor option; see Section 50.2, "*ttk* style maps: dynamic appearance changes" (p. 151).
highlightcolor	You may specify the default focus highlight color by setting this option in a style. You may also control the focus highlight color using a style map.
highlightthickness	Configure this option using a style. This option may not work in all themes.
insertbackground	Not supported.
insertborderwidth	Not supported.
insertofftime	Not supported.
insertontime	Not supported.
insertwidth	Not supported.
readonlybackground	Use a style map to control the background option; see Section 50.2, "*ttk* style maps: dynamic appearance changes" (p. 151).
relief	Configure this option using a style; see Section 47, "Customizing and creating *ttk* themes and styles" (p. 146).
selectbackground	Use a style map to control the background option; see Section 50.2, "*ttk* style maps: dynamic appearance changes" (p. 151).
selectborderwidth	Use a style map to control the borderwidth option; see Section 50.2, "*ttk* style maps: dynamic appearance changes" (p. 151).
selectforeground	Use a style map to control the foreground option; see Section 50.2, "*ttk* style maps: dynamic appearance changes" (p. 151).

Methods on a *ttk*.Entry include all those described in Section 46, "Methods common to all *ttk* widgets" (p. 145), plus all the methods on the *Tkinter* widget described in Section 10, "The Entry widget" (p. 41).

33. *ttk*.Frame

Like the *Tkinter* Frame widget, the *ttk*.Frame widget is a rectangular container for other widgets. To create a Frame widget as the child of a given *parent* widget:

```
    w = ttk.Frame(parent, option=value, ...)
```

Options include:

Table 42. *ttk*.Frame options

borderwidth	Use this option to specify the width of the border element; the default is zero.
class_	You may provide a widget class name when you create this widget. This name may be used to customize the widget's appearance; see Section 27, "Standardizing appearance" (p. 105). Once the widget is created, the widget class name cannot be changed.
cursor	Use this option to specify the appearance of the mouse cursor when it is over the widget; see Section 5.8, "Cursors" (p. 13). The default value (an empty string) specifies that the cursor is inherited from the parent widget.

`height`	This option is a dimension that sets the height of the frame. If you want to force the frame to have a specific height, call the `.grid_propagate(0)` on the widget; see Section 4.2, "Other grid management methods" (p. 7).
`padding`	To create an empty area inside the frame and outside of the contained widgets, set this option to the desired dimension. For example, `padding='0.5i'` would clear a half-inch-wide area inside the frame and around the outside of the widgets inside it.
`relief`	Specifies the relief style for the border; see Section 5.6, "Relief styles" (p. 12). This has no effect unless you also increase the `borderwidth`.
`style`	Use this option to specify a custom widget style name; see Section 47, "Customizing and creating *ttk* themes and styles" (p. 146).
`takefocus`	Use this option to specify whether a widget is visited during focus traversal; see Section 53, "Focus: routing keyboard input" (p. 155). Specify `takefocus=True` if you want the visit to accept focus; specify `takefocus=False` if the widget is not to accept focus. The default value is an empty string; by default, *ttk*`.Frame` widgets do not get focus.
`width`	This option is a dimension that sets the width of the frame. If you want to force the frame to have a specific width, call the `.grid_propagate(0)` on the widget; see Section 4.2, "Other grid management methods" (p. 7).

These options on the *Tkinter* `Frame` widget are *not* available as options on the *ttk*`.Frame` constructor:

Table 43. *Tkinter* `Frame` options not in *ttk*`.Frame`

`background` or `bg`	Configure this option with a style; see Section 47, "Customizing and creating *ttk* themes and styles" (p. 146).
`highlightbackground`	To control the color of the focus highlight when the frame does not have focus, use a style map to control the `highlightcolor` option; see Section 50.2, "*ttk* style maps: dynamic appearance changes" (p. 151).
`highlightcolor`	You may specify the default focus highlight color by setting this option in a style. You may also control the focus highlight color using a style map.
`highlightthickness`	Configure this option using a style. This option may not work in all themes.
`padx`	Not supported.
`pady`	Not supported.

34. *ttk*`.Label`

The purpose of this widget is to display text, an image, or both. Generally the content is static, but your program can change the text or the image.

To create a *ttk*`.Label` widget as the child of a given *`parent`* widget:

```
    w = ttk.Label(parent, option=value, ...)
```

Options include:

Table 44. *ttk*`.Label` options

`anchor`	If the text and/or image are smaller than the specified `width`, you can use the `anchor` option to specify where to position them: `tk.W`, `tk.CENTER`, or `tk.E`

<table>
<tr><td></td><td>for left, centered, or right alignment, respectively. You may also specify this option using a style.</td></tr>
<tr><td>background</td><td>Use this option to set the background color. You may also specify this option using a style.</td></tr>
<tr><td>borderwidth</td><td>To add a border around the label, set this option to the width dimension. You may also specify this option using a style.</td></tr>
<tr><td>class_</td><td>You may provide a widget class name when you create this widget. This name may be used to customize the widget's appearance; see Section 27, “Standardizing appearance” (p. 105). Once the widget is created, the widget class name cannot be changed.</td></tr>
<tr><td>compound</td><td>If you provide both text and image options, the compound option specifies how to display them.

<table><tr><td>'bottom'</td><td>Display the image below the text.</td></tr><tr><td>'image'</td><td>Display only the image, not the text.</td></tr><tr><td>'left'</td><td>Display the image to the left of the text.</td></tr><tr><td>'none'</td><td>Display the image if there is one, otherwise display the text. This is the default value.</td></tr><tr><td>'right'</td><td>Display the image to the right of the text.</td></tr><tr><td>'text'</td><td>Display the text, not the image.</td></tr><tr><td>'top'</td><td>Display the image above the text.</td></tr></table></td></tr>
<tr><td>cursor</td><td>Use this option to specify the appearance of the mouse cursor when it is over the widget; see Section 5.8, “Cursors” (p. 13). The default value (an empty string) specifies that the cursor is inherited from the parent widget.</td></tr>
<tr><td>font</td><td>Use this option to specify the font style for the displayed text. You may also specify this option using a style.</td></tr>
<tr><td>foreground</td><td>Use this option to specify the color of the displayed text. You may also specify this option using a style.</td></tr>
<tr><td>image</td><td>This option specifies an image or images to be displayed either in addition to or instead of text. The value must be an image as specified in Section 5.9, “Images” (p. 14). See the compound option above for what happens when you supply both image and text.

You may specify multiple images that will appear on the widget depending on the state of the widget (see Section 50.2, “ttk style maps: dynamic appearance changes” (p. 151) for a discussion of widget states). To do this, supply as the value of this option a tuple <code>(i₀, s₁, i₁, s₂, i₂, ...)</code>, where:

• <code>i₀</code> is the default image to be displayed on the widget.
• For each pair of values after the first, <code>s_i</code> specifies a state or combination of states, and <code>i₁</code> specifies the image to be displayed when the widget's state matches <code>s_i</code>.

Each state specifier <code>s_i</code> may be a single state name, optionally preceded by '!', or a sequence of such names. The ! specifies that the widget must not be in that state.</td></tr>
</table>

	For example, suppose you have three `PhotoImage` instances named `im1`, `im2`, and `im3`, and in your call to the `Label` constructor you include this option: `self.w = ttk.Label(self, ..., image=(im1, 'selected', im2, ('!disabled', 'alternate'), im3), ...)` The widget will display image `im2` if it is in the `selected` state. If it is not in the `selected` state or the `disabled` state but it is in the `alternate` state, it will display image `im3`. Otherwise it will display image `im1`.
`justify`	If the `text` you provide contains newline (`'\n'`) characters, this option specifies how each line will be positioned horizontally: `tk.LEFT` to left-justify; `tk.CENTER` to center; or `tk.RIGHT` to right-justify each line. You may also specify this option using a style.
`padding`	To add more space around all four sides of the text and/or image, set this option to the desired dimension. You may also specify this option using a style.
`relief`	Set this option to a relief style to create a 3-d effect. You will need to increase the `borderwidth` to make this effect appear. You may also specify this option using a style.
`style`	Use this option to specify a custom widget style name; see Section 47, "Customizing and creating *ttk* themes and styles" (p. 146).
`takefocus`	Use this option to specify whether the widget is visited during focus traversal; see Section 53, "Focus: routing keyboard input" (p. 155). Specify `takefocus=True` if you want the visit to accept focus; specify `takefocus=False` if the widget is not to accept focus. The default value is an empty string; by default, *ttk*`.Label` widgets do not get focus.
`text`	A string of text to be displayed in the widget.
`textvariable`	A `StringVar` instance (see Section 52, "Control variables: the values behind the widgets" (p. 153)); the text displayed on the widget will be its value. If both `text` and `textvariable` are specified, the `text` option will be ignored.
`underline`	You can request that one of the letters in the text string be underline by setting this option to the position of that letter. For example, the options `text='Quit'` and `underline=0` would underline the Q. Using this option doesn't change anything functionally. If you want the application to react to the Q key or some variation like control-shift-Q, you'll need to set up the bindings using the event system.
`width`	To specify a fixed width, set this option to the number of characters. To specify a minimum width, set this option to minus the number of characters. If you don't specify this option, the size of the label area will be just enough to accommodate the current text and/or image. For text displayed in a proportional font, the actual width of the widget will be based on the average width of a character in the font, and not a specific number of characters. This option may also be specified through a style.

`wraplength`	If you set this option to some dimension, all the text will be chopped into lines no longer than this dimension. This option may also be specified through a style.

The following options of the *Tkinter* version of `Label` are *not* supported by the *ttk*`.Label` constructor.

Table 45. *Tkinter* `Label` options not in *ttk*`.Label`

`activebackground`	Use a style map to control the `background` option; see Section 50.2, "*ttk* style maps: dynamic appearance changes" (p. 151).
`activeforeground`	Use a style map to control the `foreground` option.
`bitmap`	Not supported.
`disabledforeground`	Use a style map for the `foreground` option; see Section 50.2, "*ttk* style maps: dynamic appearance changes" (p. 151).
`height`	Not supported.
`highlightbackground`	To control the color of the focus highlight when the label does not have focus, use a style map to control the `highlightcolor` option; see Section 50.2, "*ttk* style maps: dynamic appearance changes" (p. 151).
`highlightcolor`	You may specify the default focus highlight color by setting this option in a style. You may also control the focus highlight color using a style map.
`highlightthickness`	Configure this option using a style. This option may not work in all themes.
`padx`	Not supported.
`pady`	Not supported.

35. *ttk*`.LabelFrame`

This is the *ttk* version of the basic *Tkinter* widget described in Section 13, "The `LabelFrame` widget" (p. 50).

To create a new *ttk*`.LabelFrame` widget as a child of a given *`parent`* widget:

```
w = ttk.LabelFrame(parent, option=value, ...)
```

Options include:

Table 46. *ttk*`.LabelFrame` options

`borderwidth`	Use this option to set the width of the border around the widget to a given dimension. This option may also be configured using a style.
`class_`	You may provide a widget class name when you create this widget. This name may be used to customize the widget's appearance; see Section 27, "Standardizing appearance" (p. 105). Once the widget is created, the widget class name cannot be changed.
`cursor`	Use this option to specify the appearance of the mouse cursor when it is over the widget; see Section 5.8, "Cursors" (p. 13). The default value (an empty string) specifies that the cursor is inherited from the parent widget.
`height`	This option can be set to some dimension to specify the height of the frame. If you don't call the `.grid_propagate(0)` method, this option will be ignored; see Section 4.2, "Other grid management methods" (p. 7).

`labelanchor`	Use this option to specify the position of the label on the widget's border. The default position is `'nw'`, which places the label at the left end of the top border. For the nine possible label positions, refer to Section 13, "The `LabelFrame` widget" (p. 50).
`labelwidget`	Instead of a text label, you can use any widget as the label in a *ttk*`.LabelFrame`. Create some widget *w* but do not register it with the `.grid()` method. Then create the `LabelFrame` with `labelwidget=`*w*. If you specify this option as well as the `text` option, the latter is ignored. For example, if you don't like the rather small and plain default font used for the label, you can use this option to display a `Label` widget with the font and other appearance of your choice.
`padding`	To add extra clear area around the contents of this widget, set this option to a dimension. This option may also be specified by a style.
`relief`	Use this option to specify a 3-d border style; see Section 5.6, "Relief styles" (p. 12). You will need to specify a nonzero `borderwidth` for this effect to appear. This option may also be specified by a style.
`style`	Use this option to specify a custom widget style name; see Section 47, "Customizing and creating *ttk* themes and styles" (p. 146).
`takefocus`	Use this option to specify whether the widget is visited during focus traversal; see Section 53, "Focus: routing keyboard input" (p. 155). Specify `takefocus=True` if you want the visit to accept focus; specify `takefocus=False` if the widget is not to accept focus. The default value is an empty string; by default, *ttk*`.Label` widgets do not get focus.
`text`	The value of this option is a string that will appear as part of the border.
`underline`	You can request that one of the letters in the text string be underline by setting this option to the position of that letter. For example, if you specified `text='Panic'` and `underline=2`, an underline would appear under the `'n'`. Using this option doesn't change anything functionally. If you want the application to react to the Q key or some variation like control-shift-Q, you'll need to set up the bindings using the event system.
`width`	This option can be set to some dimension to specify the width of the frame. If you don't call the `.grid_propagate(0)` method, this option will be ignored; see Section 4.2, "Other grid management methods" (p. 7).

The following options available for the *Tkinter* `LabelFrame` widget are *not* available as constructor arguments.

Table 47. *Tkinter* `LabelFrame` options not in *ttk*`.LabelFrame`

`background` or `bg`	Configure the `background` option using a style; see Section 47, "Customizing and creating *ttk* themes and styles" (p. 146). The `bg` abbreviation is not supported.
`highlightbackground`	To control the color of the focus highlight when the `LabelFrame` does not have focus, use a style map to control the `highlightcolor` option; see Section 50.2, "*ttk* style maps: dynamic appearance changes" (p. 151).
`highlightcolor`	You may specify the default focus highlight color by setting this option in a style. You may also control the focus highlight color using a style map.
`highlightthickness`	Configure this option using a style. This option may not work in all themes.

The *ttk*.LabelFrame widget supports all the methods described in Section 46, "Methods common to all *ttk* widgets" (p. 145).

36. *ttk*.Menubutton

A Menubutton widget is the part of a drop-down menu that is always visible. It is always used in combination with a Menu widget that controls what appears when the user clicks on the Menubutton.

There is no *ttk* version of the Menu widget. Use the regular *Tkinter* widget described in Section 15, "The Menu widget" (p. 56).

To create a new *ttk*.Menubutton widget as the child of some *parent* widget, use this constructor:

```
w = ttk.Menubutton(parent, option=value, ...)
```

Options include:

Table 48. *ttk*.Menubutton options

class_	The widget class name. This may be specified when the widget is created, but cannot be changed later. For an explanation of widget classes, see Section 27, "Standardizing appearance" (p. 105).
compound	If you provide both image and text options, the compound option specifies the position of the image relative to the text. The value may be tk.TOP (image above text), tk.BOTTOM (image below text), tk.LEFT (image to the left of the text), or tk.RIGHT (image to the right of the text). When you provide both image and text options but don't specify a compound option, the image will appear and the text will not.
cursor	The cursor that will appear when the mouse is over the button; see Section 5.8, "Cursors" (p. 13).
direction	This option specifies the position where the drop-down menu appears, relative to the menubutton. above — The menu will appear just above the menubutton. below — The menu will appear just below the menubutton. flush — The menu will appear over the menubutton, so that the menu's northwest corner coincides with the menubutton's northwest corner. left — The menu will appear just to the left of the menubutton. right — The menu will appear just to the right of the menubutton.
image	An image to appear on the menubutton; see Section 5.9, "Images" (p. 14).
menu	The related Menu widget. See Section 15, "The Menu widget" (p. 56) for the procedure used to establish this mutual connection.
style	The style to be used in rendering this menubutton; see Section 49, "Using and customizing *ttk* styles" (p. 147).
takefocus	By default, a *ttk*.Menubutton will be included in focus traversal; see Section 53, "Focus: routing keyboard input" (p. 155). To remove the widget from focus traversal, use takefocus=False.
text	The text to appear on the menubutton, as a string.

`textvariable`	A variable that controls the text that appears on the menubutton; see Section 52, “Control variables: the values behind the widgets” (p. 153).
`underline`	If this option has a nonnegative value *n*, an underline will appear under the character at position *n*.
`width`	If the label is text, this option specifies the absolute width of the text area on the button, as a number of characters; the actual width is that number multiplied by the average width of a character in the current font. For image labels, this option is ignored. The option may also be configured in a style.

The following options of the *Tkinter* `Menubutton` button, described in Section 16, “The `Menubutton` widget” (p. 61), are *not* supported by *ttk*`.Menubutton`:

Table 49. *Tkinter* `Menubutton` options not in *ttk*`.Menubutton`

`activebackground`	Use a style map to control the `background` option; see Section 50.2, “*ttk* style maps: dynamic appearance changes” (p. 151).
`activeforeground`	Use a style map to control the `foreground` option.
`anchor`	Configure this option using a style; see Section 49, “Using and customizing *ttk* styles” (p. 147). Use this option to specify the position of the text when the `width` option allocates extra horizontal space.
`bitmap`	Not supported.
`borderwidth` or `bd`	Configure the `borderwidth` option using a style. The `bd` abbreviation is not supported.
`buttonbackground`	Not supported.
`buttoncursor`	Not supported.
`buttondownrelief`	Not supported.
`buttonup`	Not supported.
`disabledforeground`	Use a style map for the `foreground` option; see Section 50.2, “*ttk* style maps: dynamic appearance changes” (p. 151).
`font`	Configure this option using a style.
`foreground` or `fg`	Configure the `foreground` option using a style.
`height`	Not supported.
`highlightbackground`	To control the color of the focus highlight when the menubutton does not have focus, use a style map to control the `highlightcolor` option; see Section 50.2, “*ttk* style maps: dynamic appearance changes” (p. 151).
`highlightcolor`	You may specify the default focus highlight color by setting this option in a style. You may also control the focus highlight color using a style map.
`highlightthickness`	Configure this option using a style.
`justify`	If the `text` contains newline (`'\n'`) characters, the text will occupy multiple lines on the menubutton. The `justify` option controls how each line is positioned horizontally. Configure this option using a style; values may be `tk.LEFT`, `tk.CENTER`, or `tk.RIGHT` for lines that are left-aligned, centered, or right-aligned, respectively.
`padx`	Not supported.
`pady`	Not supported.

`relief`	Configure this option using a style; see Section 49, "Using and customizing *ttk* styles" (p. 147).
`wraplength`	If you use a style with this option set to some dimension, the `text` will be sliced into pieces no longer than that dimension.

37. *ttk*.Notebook

The purpose of a `Notebook` widget is to provide an area where the user can select pages of content by clicking on tabs at the top of the area, like these:

- Each time the user clicks on one of these tabs, the widget will display the *child pane* associated with that tab. Typically, each pane will be a `Frame` widget, although a pane can be any widget.
- The tab for the child pane that is currently showing is referred to as the *selected* tab.
- You will use the `Notebook` widget's `.add()` method to attach a new tab, and its related content. Other methods let you remove or temporarily hide tabs.
- Each tab has its own set of options that control its appearance and behavior. These options are described in Table 51, "Tab options for the *ttk*.Notebook widget" (p. 128).
- A number of the methods of this widget use the idea of a *tabId* to refer to one of the tabs. Different values for a *tabId* may be any of:
  - Integer values refer to the position of the tab: 0 for the first tab, 1 for the second, and so forth.
  - You can always refer to a tab by using the child widget itself.
  - A string of the form `"@x,y"` refers to the tab that currently contains the point `(x,y)` relative to the widget. For example, the string `"@37,0"` would specify the tab containing a point 37 pixels from the left side of the widget, along the top edge of the tab.
  - The string `"current"` refers to whichever tab is currently selected.
  - In a call to the `Notebook` widget's `.index()` method, use the string `"end"` to determine the current number of tabs displayed.

To create a `Notebook` widget as the child of some *parent* widget, use this constructor:

```
w = ttk.Notebook(parent, option=value, ...)
```

Options include:

Table 50. *ttk*.Notebook options

`class_`	The widget class name. This may be specified when the widget is created, but cannot be changed later. For an explanation of widget classes, see Section 27, "Standardizing appearance" (p. 105).
`cursor`	The cursor that will appear when the mouse is over the notebook; see Section 5.8, "Cursors" (p. 13).
`height`	The height in pixels to be allocated to the widget.
`padding`	To add some extra space around the outside of the widget, set this option to that amount of space as a dimension.
`style`	The style to be used in rendering this menubutton; see Section 49, "Using and customizing *ttk* styles" (p. 147).

`takefocus`	By default, a *ttk*`.Notebook` will be included in focus traversal; see Section 53, "Focus: routing keyboard input" (p. 155). To remove the widget from focus traversal, use `takefocus=False`.
`width`	The width in pixels to be allocated to the widget.

Methods on a *ttk*`.Notebook` widget include all those described in Section 46, "Methods common to all *ttk* widgets" (p. 145), plus:

`.add(`*`child`*`, **kw)`

The *`child`* argument is a widget, usually a `Frame`, that wraps the content of one child pane. If *`child`* is not one of the `Notebook` widget's child panes, *`child`* is added as the next available tab, and the keyword arguments `kw` define the tab options for the new pane. These options are defined it Table 51, "Tab options for the *ttk*`.Notebook` widget" (p. 128).

If *`child`* is a currently hidden pane, that tab will reappear in its former position.

`.enable_traversal()`

Once you call this method, a few additional key bindings will work:

- *Control-Tab* will select the tab after the one currently selected. If the last tab was currently selected, selection will rotate back to the first tab.
- *Shift-Control-Tab* does the reverse: it moves to the previous tab, wrapping around to the last tab if the first tab was selected.
- You can configure a particular hot key that directly selects a tab. To do this, use the `text` and `underline` tab options to underline one of the characters in each tab. Then the user can jump to a tab by typing *Alt-x* where *`x`* is the underlined character on that tab.

If you have multiple `Notebook` widgets in the same application, these features will not work correctly unless each child pane widget is created with its `Notebook` widget as the parent.

`.forget(`*`child`*`)`

This method permanently removes the specified *`child`* from the widget's set of tabs.

`.hide(`*`tabId`*`)`

The tab identified by *`tabId`* is temporarily removed from the set of visible tabs in the `Notebook`. You may reinstate it by calling the `.add()` method again.

`.index(`*`tabId`*`)`

For a given *`tabId`*, this method returns the numeric index of the corresponding tab. There is one exception: if the argument is the string `"end"`, the method will return the total number of tabs.

`.insert(`*`where`*`, `*`child`*`,**kw)`

This method inserts the widget *`child`* at the position specified by *`where`*, using any keyword arguments to describe the new tab and pane. For the keyword options, see Table 51, "Tab options for the *ttk*`.Notebook` widget" (p. 128).

The *`where`* argument may be any of:

- `"end"` to place the new tab after all the existing tabs.
- An existing child widget; in this case the new *`child`* is inserted just before that existing widget.

`.select([`*`tabId`*`])`

If you call this method without an argument, it will return the window name of the widget whose tab is currently displayed.

To display a specific pane in the `Notebook`, call this method with a *`tabId`* as the argument.

.tab(tabId, option=None, **kw)

Use this method either to set tab options for the child panes described by *tabId*, or to find out what options are set for that child pane. The tab options are described in Table 51, "Tab options for the *ttk*.Notebook widget" (p. 128).

If you call the method with no keyword arguments, it will return a dictionary of the tab options currently in effect for the pane specified by *tagId*.

To find out the current value of a specific tab option *X*, call this method with the argument "option=*X*", and the method will return the value of that tab option.

To set one or more tab options for the child described by *tagId*, call this method with keyword arguments. For example, if *self.nb* is a Notebook, this call would change the text displayed on the first tab:

```
    self.nb.tab(0, text='Crunchy frog')
```

.tabs()

This method returns a list of the window names of the Notebook's child panes, in order from first to last.

Here are the tab options used in the .add() and .tab() methods.

Table 51. Tab options for the *ttk*.Notebook widget

compound	If you supply both image and text to be displayed on the tab, the compound option specifies how to display them. Allowable values describe the position of the image relative to the text, and may be any of tk.BOTTOM, tk.TOP, tk.LEFT, tk.RIGHT, or tk.CENTER. For example, compound=tk.LEFT would position the image to the left of the text.
padding	Use this option to add extra space around all four sides of the panel's content. The value is a dimension. For example, padding='0.1i' would add a 0.1" space around each side of the panel content.
sticky	Use this option to specify where the panel content is positioned if it does not completely fill the panel area. Values are the same as for the sticky argument described in Section 4.1, "The .grid() method" (p. 6). For example, sticky=tk.E+tk.S would position the content in the lower right (southeast) corner.
image	To make a graphic image appear on the tab, supply an image as the value of this option. Refer to the compound option above to specify the relative positions of image and text when you specify both.
text	The text to appear on the tab.
underline	If this option has a nonnegative value *n*, an underline will appear under the character at position *n* of the text on the tab.

37.1. Virtual events for the *ttk*.Notebook widget

Whenever the selected tab changes in a *ttk*.Notebook widget, it generates a "<<NotebookTabChanged>>" virtual event; see Section 54.8, "Virtual events" (p. 165).

38. *ttk*.PanedWindow

This is the *ttk* version of Section 19, "The PanedWindow widget" (p. 65). To create a *ttk*.PanedWindow widget as the child of a given *parent* widget:

```
    w = ttk.PanedWindow(parent, option=value, ...)
```

The options for this constructor are given in Table 52, "*ttk*.PanedWindow options" (p. 129).

Table 52. *ttk*.PanedWindow options

class_	The widget class name. This may be specified when the widget is created, but cannot be changed later. For an explanation of widget classes, see Section 27, "Standardizing appearance" (p. 105).
cursor	The cursor that will appear when the mouse is over the checkbutton; see Section 5.8, "Cursors" (p. 13).
height	The height dimension of the widget.
orient	To stack child widgets side by side, use orient=tk.HORIZONTAL. To stack them top to bottom, use orient=tk.VERTICAL. The default option is tk.VERTICAL.
style	The style to be used in rendering this widget; see Section 49, "Using and customizing *ttk* styles" (p. 147).
takefocus	By default, a *ttk*.PanedWindow will not be included in focus traversal; see Section 53, "Focus: routing keyboard input" (p. 155). To add the widget to focus traversal, use takefocus=True.
width	The width dimension of the widget.

These options of the *Tkinter*.PanedWindow widget are *not* supported by the *ttk*.PanedWindow constructor:

Table 53. *Tkinter* PanedWindow options not in *ttk*.PanedWindow

background or bg	Configure the background option using a style. The bg abbreviation is not supported.
borderwidth or bd	Not supported.
cursor	The cursor that will appear when the mouse is over the widget; see Section 5.8, "Cursors" (p. 13).
handlepad	Not supported.
handlesize	Not supported.
opaqueresize	Not supported.
relief	Not supported.
sashrelief	Not supported.
sashwidth	Not supported.
showhandle	Not supported.

Methods on a *ttk*.PanedWindow include all those described in Section 46, "Methods common to all *ttk* widgets" (p. 145), plus:

.add(*w*[, weight=*N*])
Add a new pane to the window, where *w* is any widget (but typically a `Frame`). If you provide a `weight` option, it describes the size of the pane in the stacking dimension, relative to the other panes. For example, for `orient=tk.VERTICAL`, if pane 0 has `weight=1` and pane 1 has `weight=3`, initially the first pane will have 1/4 of the height and the second pane will have 3/4.

.forget(*what*)
Remove a pane. The argument may be either the index of the pane, counting from zero, or the child widget.

.insert(*where*, *w*[, weight=*N*])
Add a new pane *w* to the window at the position specified by *where*, where *where* may be either an index or the pane widget before which you want to insert the new pane.

.panes()
This method returns a list of the `PanedWindow`'s child widgets.

39. *ttk*.Progressbar

The purpose of this widget is to reassure the user that something is happening. It can operate in one of two modes:

- In `determinate` mode, the widget shows an indicator that moves from beginning to end under program control.
- In `indeterminate` mode, the widget is animated so the user will believe that something is in progress. In this mode, the indicator bounces back and forth between the ends of the widget.

In either mode, the current position of the indicator has a numeric value. You can specify a `maximum` value, and you can set the indicator value directly. You may also specify that the indicator value moves a given amount every time a given time interval passes.

To create a new *ttk*`.Progressbar` widget as the child of a given *parent* widget:

```
    w = ttk.Progressbar(parent, option=value, ...)
```

The options for this constructor are given in Table 54, "*ttk*`.Progressbar` options" (p. 130).

Table 54. *ttk*.Progressbar options

`class_`	The widget class name. This may be specified when the widget is created, but cannot be changed later. For an explanation of widget classes, see Section 27, "Standardizing appearance" (p. 105).
`cursor`	The cursor that will appear when the mouse is over the checkbutton; see Section 5.8, "Cursors" (p. 13).
`length`	The size of the widget along its long axis as a dimension.
`maximum`	The maximum value of the indicator; default is 100.
`mode`	If your program cannot accurately depict the relative progress that this widget is supposed to display, use `mode='indeterminate'`. In this mode, a rectangle bounces back and forth between the ends of the widget once you use the `.start()` method. If your program has some measure of relative progress, use `mode='determinate'`. In this mode, your program can move the indicator to a specified position along the widget's track.

`orient`	This options specifies the orientation: use `orient=tk.HORIZONTAL` or `orient=tk.VERTICAL`.
`style`	The style to be used in rendering this widget; see Section 49, "Using and customizing *ttk* styles" (p. 147).
`takefocus`	By default, a *ttk*`.Progressbar` will not be included in focus traversal; see Section 53, "Focus: routing keyboard input" (p. 155). To add the widget to focus traversal, use `takefocus=True`.
`variable`	Use this option to link a control variable to the widget so that you can get or set the current value of the indicator.

Methods on a *ttk*`.Progressbar` include those described in Section 46, "Methods common to all *ttk* widgets" (p. 145) plus:

`.start([`*`interval`*`])`
Start moving the indicator every *`interval`* milliseconds; the default is 50ms. Each time, the indicator is moved as if you called the `.step()` method.

`.step([`*`delta`*`])`
This method increases the indicator value by *`delta`*; the default increment is 1.0. In determinate mode, the indicator will never exceed the value of the `maximum` option. In indeterminate mode, the indicator will reverse direction and count down once it reaches the maximum value.

`.stop()`
This method stops the automatic progress that was started by calling the `.start()` method.

40. *ttk*`.Radiobutton`

This widget is the *ttk* version of Section 20, "The `Radiobutton` widget" (p. 68). To create a *ttk*`.Radiobutton` widget as the child of a given *`parent`* widget, where the *`option`* values are given in Table 55, "*ttk*`.Radiobutton` options" (p. 131):

```
w = ttk.Radiobutton(parent, option=value, ...)
```

Table 55. *ttk*`.Radiobutton` options

`class_`	The widget class name. This may be specified when the widget is created, but cannot be changed later. For an explanation of widget classes, see Section 27, "Standardizing appearance" (p. 105).
`command`	A function to be called whenever the state of this radiobutton changes.
`compound`	This option specifies the relative position of the image relative to the text when you specify both. The value may be `tk.TOP` (image above text), `tk.BOTTOM` (image below text), `tk.LEFT` (image to the left of the text), or `tk.RIGHT` (image to the right of the text). If you provide both `image` and `text` options but do not specify a value for `compound`, only the image will appear.
`cursor`	The cursor that will appear when the mouse is over the radiobutton; see Section 5.8, "Cursors" (p. 13).
`image`	An image to appear on the radiobutton; see Section 5.9, "Images" (p. 14).
`style`	The style to be used in rendering this radiobutton; see Section 49, "Using and customizing *ttk* styles" (p. 147).

`takefocus`	By default, a *ttk*`.Radiobutton` will be included in focus traversal; see Section 53, "Focus: routing keyboard input" (p. 155). To remove the widget from focus traversal, use `takefocus=False`.
`text`	The text to appear next to the radiobutton, as a string.
`textvariable`	A variable that controls the text that appears on the radiobutton; see Section 52, "Control variables: the values behind the widgets" (p. 153).
`underline`	If this option has a nonnegative value *n*, an underline will appear under the `text` character at position *n*.
`value`	The value associated with this radiobutton. When the radiobutton is the one selected in the group, the value of this option will be stored in the control variable for the group.
`variable`	A control variable that is shared by the other radiobuttons in the group; see Section 52, "Control variables: the values behind the widgets" (p. 153). The type of this variable will be the same as the type you specify for the `value` options for the radiobuttons in the group.
`width`	Use this option to specify a fixed width or a minimum width. The value is specified in characters; a positive value sets a fixed width of that many average characters, while a negative width sets a minimum width. You may also specify a `width` value in an associated style. If values are specified both in the widget constructor call and in the style, the former takes priority.

These options of the *Tkinter* `Radiobutton` widget are *not* supported by the *ttk*`.Radiobutton` constructor:

Table 56. *ttk* `Radiobutton` options not in *ttk*`.Radiobutton`

`activebackground`	Use a style map to control the `background` option; see Section 50.2, "*ttk* style maps: dynamic appearance changes" (p. 151).
`activeforeground`	Use a style map to control the `foreground` option.
`anchor`	Configure this option using a style; see Section 49, "Using and customizing *ttk* styles" (p. 147). Use this option to specify the position of the text when the `width` option allocates extra horizontal space. For example, if you specify options `width=30` and `compound=tk.BOTTOM` on a radiobutton that displays both text and and image, and a style that specifies `anchor=tk.W` (west), the image will be at the left-hand end of the thirty-character space, with the text just above it. When a radiobutton displays an image but no text, this option is ignored.
`background` or `bg`	Configure the `background` option using a style. The `bg` abbreviation is not supported.
`bitmap`	Not supported.
`borderwidth` or `bd`	Configure this option using a style.
`disabledforeground`	Use a style map for the `foreground` option; see Section 50.2, "*ttk* style maps: dynamic appearance changes" (p. 151).
`font`	Configure this option using a style.
`foreground` or `fg`	Configure the `foreground` option using a style. The `fg` abbreviation is not supported.

`height`	Not supported.
`highlightbackground`	To control the color of the focus highlight when the menubutton does not have focus, use a style map to control the `highlightcolor` option; see Section 50.2, "*ttk* style maps: dynamic appearance changes" (p. 151).
`highlightcolor`	You may specify the default focus highlight color by setting this option in a style. You may also control the focus highlight color using a style map.
`highlightthickness`	Configure this option using a style.
`indicatoron`	Not supported.
`justify`	Controls how multiple lines are positioned horizontally relative to each other. Configure this option using a style; values may be `tk.LEFT`, `tk.CENTER`, or `tk.RIGHT` for left-aligned, centered, or right-aligned, respectively.
`offrelief`	Not supported.
`overrelief`	Not supported.
`padx`	Not supported.
`pady`	Not supported.
`relief`	Configure this option using a style.
`selectcolor`	Not supported.
`selectimage`	Not supported.
`state`	In *ttk*, there is no option with this name. The state mechanism has been generalized; see Section 50.2, "*ttk* style maps: dynamic appearance changes" (p. 151).
`wraplength`	If you use a style that has this option set to some dimension, the `text` will be sliced into pieces no longer than that dimension.

Methods on a *ttk*`.Radiobutton` include all those described in Section 46, "Methods common to all *ttk* widgets" (p. 145), plus:

`.invoke()`
When you call this method on a *ttk*`.Radiobutton`, the result is the same as if the user clicked on it: the indicator is set and the associated `variable` is set to the radiobutton's `value`. If there is a `command` associated with this radiobutton, that function is called, and the `.invoke()` method returns whatever the function returned; otherwise it returns `None`.

41. *ttk*`.Scale`

This is the *ttk* version of Section 21, "The `Scale` widget" (p. 71). To create a *ttk*`.Scale` widget as the child of a given *`parent`* widget, where the *`option`* values are given in Table 57, "*ttk*`.Scale` options" (p. 133):

```
    w = ttk.Scale(parent, option=value, ...)
```

Table 57. *ttk*`.Scale` options

`class_`	The widget class name. This may be specified when the widget is created, but cannot be changed later. For an explanation of widget classes, see Section 27, "Standardizing appearance" (p. 105).

`command`	A function to be called whenever the state of this widget changes. This function will receive one argument, the new value shown on the widget, as a `float`.
`cursor`	The cursor that will appear when the mouse is over the scale; see Section 5.8, "Cursors" (p. 13).
`from_`	Use this option in combination with the `to` option (described below) to constrain the values to a numeric range. For example, `from_=-10` and `to=10` would allow only values between –10 and 20 inclusive. See also the `increment` option below.
`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. For allowable values, see Section 5.1, "Dimensions" (p. 9).
`orient`	Set `orient=tk.HORIZONTAL` if you want the scale to run along the *x* dimension, or `orient=tk.VERTICAL` to run parallel to the *y*-axis. Default is vertical.
`style`	The style to be used in rendering this radiobutton; see Section 49, "Using and customizing *ttk* styles" (p. 147).
`takefocus`	By default, a *ttk*`.Scale` widget will be included in focus traversal; see Section 53, "Focus: routing keyboard input" (p. 155). To remove the widget from focus traversal, use `takefocus=False`.
`to`	A `float` 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. The default value is 100.
`value`	Use this option to set the initial value of the widget's `variable`; the default is 0.0.
`variable`	Use this option to associate a control variable with the widget. Typically this will be a `tk.DoubleVar` instance, which holds a value of type `float`. You may instead use a `tk.IntVar` instance, but values stored in it will be truncated as type `int`.

These options of the *Tkinter* `Scale` widget are *not* supported by the *ttk*`.Scale` widget constructor:

Table 58. *Tkinter* `Scale` options not in *ttk*`.Scale`

`activebackground`	Use a style map to control the `background` option; see Section 50.2, "*ttk* style maps: dynamic appearance changes" (p. 151).
`background` or `bg`	Configure the `background` option using a style; this option controls the color of the slider. The `bg` abbreviation is not supported.
`borderwidth` or `bd`	Configure this option using a style.
`digits`	Not supported.
`font`	Not supported.
`foreground` or `fg`	Not supported.
`highlightbackground`	Not supported.
`highlightcolor`	Not supported.
`highlightthickness`	Not supported.
`label`	Not supported.
`relief`	Not supported.
`repeatdelay`	Not supported.
`repeatinterval`	Not supported.

`resolution`	Not supported.
`showvalue`	Not supported.
`sliderlength`	Configure this option using a style.
`sliderrelief`	Configure this option using a style.
`state`	In *ttk*, there is no option with this name. The state mechanism has been generalized; see Section 50.2, "*ttk* style maps: dynamic appearance changes" (p. 151).
`tickinterval`	Not supported.
`troughcolor`	Configure this option using a style.
`width`	Configure this option using the `sliderthickness` option in a style.

Methods on a *ttk*`.Scale` include all those described in Section 46, "Methods common to all *ttk* widgets" (p. 145), plus:

`.get()`
Returns the current value shown on the widget.

`.set(`*`newValue`*`)`
Change the widget's current value to *`newValue`*.

42. *ttk*`.Scrollbar`

This is the *ttk* version of Section 22, "The `Scrollbar` widget" (p. 74). To create a *ttk*`.Scrollbar` as the child of a given *`parent`* widget, where the *`option`* values are given in Table 59, "*ttk*`.Scrollbar` options" (p. 135):

```
    w = ttk.Scrollbar(parent, option=value, ...)
```

Table 59. *ttk*`.Scrollbar` options

`class_`	The widget class name. This may be specified when the widget is created, but cannot be changed later. For an explanation of widget classes, see Section 27, "Standardizing appearance" (p. 105).
`command`	A procedure to be called whenever the scrollbar is moved. For a discussion of the calling sequence, see Section 22.1, "The `Scrollbar` *`command`* callback" (p. 77).
`cursor`	The cursor that will appear when the mouse is over the scrollbar; see Section 5.8, "Cursors" (p. 13).
`orient`	Set `orient=tk.HORIZONTAL` for a horizontal scrollbar, `orient=tk.VERTICAL` for a vertical one (the default orientation).
`style`	The style to be used in rendering this scrollbar; see Section 49, "Using and customizing *ttk* styles" (p. 147).
`takefocus`	By default, a *ttk*`.Scrollbar` will not be included in focus traversal; see Section 53, "Focus: routing keyboard input" (p. 155). To add the widget to focus traversal, use `takefocus=True`.

These options of a *Tkinter* `Scrollbar` widget are *not* supported by the *ttk*`.Scrollbar` constructor:

Table 60. *Tkinter* options not in *ttk*.Scrollbar

activebackground	Use a style map to control the background option; see Section 50.2, "*ttk* style maps: dynamic appearance changes" (p. 151).
activerelief	Use a style map to control the relief option; see Section 50.2, "*ttk* style maps: dynamic appearance changes" (p. 151).
background or bg	Configure the background option using a style; this option controls the color of the slider. The bg abbreviation is not supported.
borderwidth or bd	Configure the borderwidth option using a style. The bd abbreviation is not supported.
elementborderwidth	Not supported.
highlightbackground	Not supported.
highlightcolor	Not supported.
highlightthickness	Not supported.
jump	Not supported.
relief	Configure this option using a style.
repeatdelay	Not supported.
repeatinterval	Not supported.
troughcolor	Configure this option using a style.
width	Configure this option using a style. You may find that configuring arrowsize is a better choice; in some themes, increasing the width may not increase the size of the arrowheads.

Methods on a *ttk*.Scrollbar include all those described in Section 46, "Methods common to all *ttk* widgets" (p. 145), plus:

.delta(*dx*, *dy*)
Given a mouse movement of (*dx*, *dy*) in pixels, this method returns the float value that should be added to the current slider position to achieve that same movement. The value must be in the closed interval [-1.0, 1.0].

.fraction(*x*, *y*)
Given a pixel location (*x*, *y*), this method returns the corresponding normalized slider position in the interval [0.0, 1.0] that is closest to that location.

.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. Each value is in the interval [0.0, 1.0] where 0.0 is the leftmost or top position and 1.0 is the rightmost or bottom position. For example, if the slider extends from halfway to three-quarters of the way along the trough, you might get back the tuple (0.5,0.75).

.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. Please note that moving the scrollbar's slider does *not* move the corresponding widget.

43. *ttk*.Separator

Use this widget to place a horizontal or vertical bar that separates other widgets. The widget is rendered as a 2-pixel wide line. Be sure to use the `sticky` options to the `.grid()` method to stretch the widget, or it will appear as only a single pixel.

To create a *ttk*.Separator as the child of a given *parent* widget, where the *option* values are given in Table 61, "*ttk*.Separator options" (p. 137):

```
w = ttk.Separator(parent, option=value, ...)
```

Table 61. *ttk*.Separator options

class_	The widget class name. This may be specified when the widget is created, but cannot be changed later. For an explanation of widget classes, see Section 27, "Standardizing appearance" (p. 105).
orient	Set orient=tk.HORIZONTAL for a horizontal separator, orient=tk.VERTICAL for a vertical one (the default orientation).
style	The style to be used in rendering this scrollbar; see Section 49, "Using and customizing *ttk* styles" (p. 147). The only style feature you can configure is background, which specifies the color of the separator bar; the default color is a dark gray.

The only methods available on a *ttk*.Separator widgets are the ones listed in Section 46, "Methods common to all *ttk* widgets" (p. 145).

44. *ttk*.Sizegrip

Use this widget to provide a widget that the user can use to resize the entire application window. Typically this widget will be located in the bottom right corner of the application. Be sure to make the entire application resizeable using the techniques described in Section 4.3, "Configuring column and row sizes" (p. 7) and Section 4.4, "Making the root window resizeable" (p. 8).

To create a *ttk*.Sizegrip as the child of a given *parent* widget, where the *option* values are given in Table 62, "*ttk*.Sizegrip options" (p. 137):

```
w = ttk.Sizegrip(parent, option=value, ...)
```

Table 62. *ttk*.Sizegrip options

class_	The widget class name. This may be specified when the widget is created, but cannot be changed later. For an explanation of widget classes, see Section 27, "Standardizing appearance" (p. 105).
style	The style to be used in rendering this widget; see Section 49, "Using and customizing *ttk* styles" (p. 147). The only style feature you can configure is background, which specifies the color of the widget.

45. *ttk*.Treeview

The purpose of the *ttk*.Treeview widget is to present a hierarchical structure so that the user can use mouse actions to reveal or hide any part of the structure.

The association with the term “tree” is due to programming practice: tree structures are a commonplace in program design. Strictly speaking, the hierarchy shown in a `Treeview` widget is a forest: there is no one root, just a collection of top-level *nodes*, each of which may contain second-level nodes, each of which may contain third-level nodes, and so on.

You may have encountered this particular presentation as a way of browsing a directory or folder hierarchy. The entire hierarchy is displayed like an indented outline, where each directory is on a separate line, and the subdirectories of each directory are displayed underneath that line, indented:

The user can click on the icon for a directory to *collapse* (close) it, hiding all of the items in it. They can also click again on the icon to *expand* (open) it, so that the items in the directory or folder are shown.

The `Treeview` widget generalizes this concept so that you can use it to display any hierarchical structure, and the reader can collapse or expand subtrees of this structure with the mouse.

First, some definitions:

item
: One of the entities being displayed in the widget. For a file browser, an item might be either a directory or a file.

  Each item is associated with a textual label, and may also be associated with an image.

iid
: Every item in the tree has a unique identifier string called the *iid*. You can supply the iid values yourself, or you can let *ttk* generate them.

child
: The items directly below a given item in a hierarchy. A directory, for example, may have two kinds of children: files and subdirectories.

parent
: For a given item, if it is at the top of the hierarchy it is said to have no parent; if it is not at the top level, the parent is the item that contains it.

ancestor
: The ancestors of an item include its parent, its parent's parent, and so on up to the top level of the tree.

visible
: Top-level items are always visible. Otherwise, an item is visible only if all its ancestors are expanded.

descendant
: The descendants of an item include its children, its childrens' children, and so on. Another way of saying this is that the subtree of an item includes all its descendants.

tag
: Your program can associate one or more *tag* strings with each item. You can use these tags to control the appearance of an item. For example, you could tag directories with the tag `'d'` and files with the tag `'f'`, and then specify that items with tag `'d'` use a boldface font.

You may also associate events with tags, so that certain events will cause certain handlers to be called for all items that have that tag. For example, you could set up a file browser so that when a user clicks on a directory, the browser updated its contents to reflect the current file structure.

Your `Treeview` widget will be structured with multiple columns. The first column, which we'll call the *icon column*, displays the icons that collapse or expand items. In the remaining columns, you may display whatever information you like.

For example, a simple file browser widget might use two columns, with the directory icons in the first column and the directory or file name in the second columns. Or you might wish to display file sizes, permissions, and other related data in additional columns.

The operations of the `Treeview` widget even allow you to use it as a tree editor. Your program can remove an entire subtree from its location in the main tree and then attach it later at an entirely different point.

Here is the general procedure for setting up a `Treeview` widget.

1. Create the widget with the *ttk*`.Treeview` constructor. Use the `columns` keyword argument to specify the number of columns to be displayed and to assign symbolic names to each column.
2. Use the `.column()` and `.heading()` methods to set up column headings (if you want them) and configure column properties such as size and stretchability.
3. Starting with the top-level entries, use the `.insert()` method to populate the tree. Each call to this method adds one item to the tree. Use the `open` keyword argument of this method to specify whether the item is initially expanded or collapsed.

   If you want to supply the iid value for this item, use the `iid` keyword argument. If you omit this argument, *ttk* will make one up and return it as the result of the `.insert()` method call.

   Use the `values` keyword argument of this method to specify what should appear in each column of this item when it is visible.

To create a `Treeview` widget within a given *`parent`* widget:

```
w = ttk.Treeview(parent, option=value, ...)
```

The constructor returns the new `Treeview` widget. Its options include:

`class_`	You may provide a widget class name when you create this widget. This name may be used to customize the widget's appearance; see Section 27, “Standardizing appearance” (p. 105). Once the widget is created, the widget class name cannot be changed.
`columns`	A sequence of column identifier strings. These strings are used internally to identify the columns within the widget. The icon column, whose identifier is always `'#0'`, contains the collapse/expand icons and is always the first column. The columns you specify with the `columns` argument are in addition to the icon column. For example, if you specified `columns=('Name', 'Size')`, three columns would appear in the widget: first the icon column, then two more columns whose internal identifiers are `'Name'` and `'Size'`.
`cursor`	Use this option to specify the appearance of the mouse cursor when it is over the widget; see Section 5.8, “Cursors” (p. 13). The default value (an empty string) specifies that the cursor is inherited from the parent widget.

<table>
<tr><td>displaycolumns</td><td>Selects which columns are actually displayed and determines the order of their presentation. Values may be:<ul><li>'#all' to select all columns and display them in the order defined by the columns argument.</li><li>A list of column numbers (integer positions, counting from 0) or column identifiers from the columns argument.
For example, suppose you specify columns=('Name', 'Size', 'Date'). This means each call to the .insert() method will require an argument values=(name, size, date) to supply the values that will be displayed. Let's call this sequence the logical column sequence.
Further suppose that in the constructor you specify columns=(2,0). The physical column sequence, the columns that will actually appear in the widget, will be three: the icon column will be first, followed by the date column (index 2 in the logical column sequence), followed by the name column (logical column index 0). The size column will not appear.
You could get the same effect by specifying column identifiers instead of logical column positions: columns=('Date', 'Name').</li></ul></td></tr>
<tr><td>height</td><td>The desired height of the widget, in rows.</td></tr>
<tr><td>padding</td><td>Use this argument to place extra space around the contents inside the widget. You may provide either a single dimension or a sequence of up to four dimensions, interpreted according to this table:<table><tr><th>Values given</th><th>Left</th><th>Top</th><th>Right</th><th>Bottom</th></tr><tr><td>a</td><td>a</td><td>a</td><td>a</td><td>a</td></tr><tr><td>a b</td><td>a</td><td>b</td><td>a</td><td>b</td></tr><tr><td>a b c</td><td>a</td><td>c</td><td>b</td><td>c</td></tr><tr><td>a b c d</td><td>a</td><td>b</td><td>c</td><td>d</td></tr></table></td></tr>
<tr><td>selectmode</td><td>This option controls what the user is allowed to select with the mouse. Values can be:<table><tr><td>selectmode='browse'</td><td>The user may select only one item at a time.</td></tr><tr><td>selectmode='extended'</td><td>The user may select multiple items at once.</td></tr><tr><td>selectmode='none'</td><td>The user cannot select items with the mouse.</td></tr></table></td></tr>
<tr><td>show</td><td>To suppress the labels at the top of each column, specify show='tree'. The default is to show the column labels.</td></tr>
<tr><td>style</td><td>Use this option to specify a custom widget style name; see Section 47, “Customizing and creating ttk themes and styles” (p. 146).</td></tr>
<tr><td>takefocus</td><td>Use this option to specify whether a widget is visited during focus traversal; see Section 53, “Focus: routing keyboard input” (p. 155). Specify takefocus=True if you want the visit to accept focus; specify takefocus=False if the widget is not to accept focus. The default value is an empty string; by default, ttk.Treeview widgets do get focus.</td></tr>
</table>

Here are the methods available on a Treeview widget.

.bbox(*item*, column=None)

For the item with iid *item*, if the item is currently visible, this method returns a tuple (*x*, *y*, *w*, *h*), where (*x*, *y*) are the coordinates of the upper left corner of that item relative to the widget, and *w* and *h* are the width and height of the item in pixels. If the item is not visible, the method returns an empty string.

If the optional column argument is omitted, you get the bounding box of the entire row. To get the bounding box of one specific column of the item's row, use column=*C* where *C* is either the integer index of the column or its column identifier.

.column(*cid*, option=None, ***kw*)

This method configures the appearance of the logical column specified by *cid*, which may be either a column index or a column identifier. To configure the icon column, use a *cid* value of '#0'.

Each column in a Treeview widget has its own set of options from this table:

anchor	The anchor that specifies where to position the content of the column. The default value is 'w'.
id	The column name. This option is read-only and set when the constructor is called.
minwidth	Minimum width of the column in pixels; the default value is 20.
stretch	If this option is True, the column's width will be adjusted when the widget is resized. The default setting is 1.
width	Initial width of the column in pixels; the default is 200.

- If no *option* value or any other keyword argument is supplied, the method returns a dictionary of the column options for the specified column.
- To interrogate the current value of an option named *X* , use an argument option=*X*.
- To set one or more column options, you may pass keyword arguments using the option names shown above, e.g., anchor=tk.CENTER to center the column contents.

.delete(items*)**

The arguments are iid values. All the items in the widget that have matching iid values are destroyed, along with all their descendants.

.detach(items*)**

The arguments are iid values. All the items in the widget that have matching iid values are removed from the visible widget, along with all their descendants.

The items are not destroyed. You may reattach them to the visible tree using the .move() method described below.

.exists(*iid*)

Returns True if there exists an item in the widget with the given *iid*, or False otherwise. If an item is not currently visible because it was removed with the .detach() method, it is still considered to exist for the purposes of the .exists() method.

.focus([*iid*])

If you don't provide an argument to this method, you get back either the iid of the item that currently has focus, or '' if no item has focus.

You can give focus to an item by passing its iid as the argument to this method.

.get_children([*item*])

Returns a tuple of the iid values of the children of the item specified by the *item* argument. If the argument is omitted, you get a tuple containing the iid values of the top-level items.

.heading(*cid*, option=None, ***kw*)

Use this method to configure the column heading that appears at the top of the widget for the column specified by *cid*, which may be either a column index or a column identifier. Use a *cid* argument value of `'#0'` to configure the heading over the icon column.

Each heading has its own set of options with these names and values:

`anchor`	An anchor that specifies how the heading is aligned within the column; see Section 5.5, "Anchors" (p. 12). The default value is `tk.W`.
`command`	A procedure to be called when the user clicks on this column heading.
`image`	To present a graphic in the column heading (either with or instead of a text heading), set this option to an image, as specified in Section 5.9, "Images" (p. 14).
`text`	The text that you want to appear in the column heading.

- If you supply no keyword arguments, the method will return a dictionary showing the current settings of the column heading options.
- To interrogate the current value of some heading option *X*, use an argument of the form `option=`*X*; the method will return the current value of that option.
- You can set one or more heading options by supplying them as keyword arguments such as "`anchor=tk.CENTER`".

.identify_column(*x*)

Given an *x* coordinate, this method returns a string of the form `'#`*n*`'` that identifies the column that contains that *x* coordinate.

Assuming that the icon column is displayed, the value of *n* is 0 for the icon column; 1 for the second physical column; 2 for the third physical column; and so on. Recall that the physical column number may be different from the logical column number in cases where you have rearranged them using the `displaycolumns` argument to the `Treeview` constructor.

If the icon column is not displayed, the value of *n* is 1 for the first physical column, 2 for the second, and so on.

.identify_element(*x*, *y*)

Returns the name of the element at location `(`*x*`, `*y*`)` relative to the widget, or `''` if no element appears at that position. Element names are discussed in Section 50, "The *ttk* element layer" (p. 149).

.identify_region(*x*, *y*))

Given the coordinates of a point relative to the widget, this method returns a string indicating what part of the widget contains that point. Return values may include:

`'nothing'`	The point is not within a functional part of the widget.
`'heading'`	The point is within one of the column headings.
`'separator'`	The point is located within the column headings row, but on the separator between columns. Use the `.identify_column()` method to determine which column is located just to the left of this separator.
`'tree'`	The point is located within the icon column.
`'cell'`	The point is located within an item row but not within the icon column.

.identify_row(*y*)

If *y*-coordinate *y* is within one of the items, this method returns the iid of that item. If that vertical coordinate is not within an item, this method returns an empty string.

.index(*iid*)

This method returns the index of the item with the specified *iid* relative to its parent, counting from zero.

.set_children(*item*, **newChildren*)

Use this method to change the set of children of the item whose iid is *item*. The *newChildren* argument is a sequence of iid strings. Any current children of *item* that are not in *newChildren* are removed.

.insert(*parent*, *index*, iid=None, **kw)

This method adds a new item to the tree, and returns the item's iid value. Arguments:

parent	To insert a new top-level item, make this argument an empty string. To insert a new item as a child of an existing item, make this argument the parent item's iid.
index	This argument specifies the position among this parent's children where you want the new item to be added. For example, to insert the item as the new first child, use a value of zero; to insert it after the parent's first child, use a value of 1; and so on. To add the new item as the last child of the parent, make this argument's value `'end'`.
`iid`	You may supply an iid for the item as a string value. If you don't supply an iid, one will be generated automatically and returned by the method.

You may also specify a number of item options as keyword arguments to this method.

`image`	You may display an image just to the right of the icon for this item's row by providing an `image=`*I* argument, where *I* is an image as specified in Section 5.9, “Images” (p. 14).
`open`	This option specifies whether this item will be open initially. If you supply `open=False`, this item will be closed. If you supply `open=True`, the item's children will be visible whenever the item itself is visible. The default value is `False`.
`tags`	You may supply one or more tag strings to be associated with this item. The value may be either a single string or a sequence of strings.
`text`	You may supply text to be displayed within the icon column of this item. If given, this text will appear just to the right of the icon, and also to the right of the image if provided.
`values`	This argument supplies the data items to be displayed in each column of the item. The values are supplied in logical column order. If too few values are supplied, the remaining columns will be blank in this item; if too many values are supplied, the extras will be discarded.

.item(*iid*[, *option*[, ***kw*]])

Use this method to set or retrieve the options within the item specified by *iid*. Refer to the `.insert()` method above for the names of the item options.

With no arguments, it returns a dictionary whose keys are the option names and the corresponding values are the settings of those options. To retrieve the value of a given option, pass the option's name as its second argument. To set one or more options, pass them as keyword arguments to the method.

.move(*iid*, *parent*, *index*)

Move the item specified by *iid* to the values under the item specified by *parent* at position *index*. The *parent* and *index* arguments work the same as those arguments to the `.index()` method.

.next(*iid*)

If the item specified by *iid* is not the last child of its parent, this method returns the iid of the following child; if it is the last child of its parent, this method returns an empty string. If the specified

item is a top-level item, the method returns the iid of the next top-level item, or an empty string if the specified item is the last top-level item.

.parent(*iid*)
If the item specified by *iid* is a top-level item, this method returns an empty string; otherwise it returns the iid of that item's parent.

.prev(*iid*)
If the item specified by *iid* is not the first child of its parent, this method returns the iid of the previous child; otherwise it returns an empty string. If the specified item is a top-level item, this method returns the iid of the previous top-level item, or an empty string if it is the first top-level item.

.see(*iid*)
This method ensures that the item specified by *iid* is visible. Any of its ancestors that are closed are opened. The widget is scrolled, if necessary, so that the item appears.

.selection_add(*items*)
In addition to any items already selected, add the specified *items*. The argument may be either a single iid or a sequence of iids.

.selection_remove(*items*)
Unselect any items specified by the argument, which may be a single iid or a sequence of iids.

.selection_set(*items*)
Only the specified *items* will be selected; if any other items were selected before, they will become unselected.

.selection_toggle(*items*)
The argument may be a single iid or a sequence of iids. For each item specified by the argument, if it was selected, unselect it; if it was unselected, select it.

.set(*iid*, column=None, value=None)
Use this method to retrieve or set the column values of the item specified by *iid*. With one argument, the method returns a dictionary: the keys are the column identifiers, and each related value is the text in the corresponding column.

With two arguments, the method returns the data value from the column of the selected item whose column identifier is the `column` argument. With three arguments, the item's value for the specified column is set to the third argument.

.tag_bind(*tagName*, sequence=None, callback=None)
This method binds the event handler specified by the `callback` argument to all items that have tag *tagName*. The `sequence` and `callback` arguments work the same as the `sequence` and `func` arguments of the `.bind()` method described in Section 26, "Universal widget methods" (p. 97).

.tag_configure(*tagName*, option=None, ***kw*)
This method can either interrogate or set options that affect the appearance of all the items that have tag `tagName`. Tag options include:

`'background'`	The background color.
`'font'`	The text font.
`'foreground'`	The foreground color.
`'image'`	An image to be displayed in items with the given tag.

When called with one argument, it returns a dictionary of the current tag options. To return the value of a specific option *X*, use *X* as the second argument.

To set one or more options, use keyword arguments such as `foreground='red'`.

`.tag_has(`*`tagName`*`[, `*`iid`*`])`
Called with one argument, this method returns a list of the iid values for all items that carry tag *`tagName`*. If you provide an iid as the second argument, the method returns `True` if the item with that iid has tag *`tagName`*, `False` otherwise.

`.xview(*args)`
This is the usual method for connecting a horizontal scrollbar to a scrollable widget. For details, see Section 22.1, "The `Scrollbar` *`command`* callback" (p. 77).

`.yview(*args)`
This is the usual method for connecting a vertical scrollbar to a scrollable widget. For details, see Section 22.1, "The `Scrollbar` *`command`* callback" (p. 77).

45.1. Virtual events for the *ttk* `Treeview` widget

Certain state changes within a `Treeview` widget generate virtual events that you can use to respond to these changes; see Section 54.8, "Virtual events" (p. 165).

- Whenever there is a change in the selection, either by items becoming selected or becoming unselected, the widget generates a "`<<TreeviewSelect>>`" event.
- Whenever an item is opened, the widget generates a "`<<TreeviewOpen>>`" event.
- Whenever an item is closed, the widget generates a "`<<TreeviewClose>>`" event.

46. Methods common to all *ttk* widgets

The methods shown here are available on all the *ttk* widgets.

`.cget(`*`option`*`)`
This method returns the value for the specified *`option`*.

`.configure(`*`option=value, ...`*`)`
To set one or more widget options, use keyword arguments of the form *`option=value`*. For example, to set a widget's `font`, you might use an argument such as "`font=('serif', 12)`".

If you provide no arguments, the method will return a dictionary of all the widget's current option values. In this dictionary, the keys will be the option names, and each related value will be a tuple (*`name`*, *`dbName`*, *`dbClass`*, *`default`*, *`current`*):

`name`	The option name.
`dbName`	The database name of the option.
`dbClass`	The database class of the option.
`default`	The default value of the option.
`current`	The current value of the option.

`.identify(`*`x, y`*`)`
Use this to determine what element is at a given location within the widget. If the point (*`x, y`*) relative to the widget is somewhere within the widget, this method returns the name of the element at that position; otherwise it returns an empty string.

.instate(*stateSpec*, callback=None, *args, **kw)

The purpose of this to determine whether the widget is in a specified state or combination of states.

If you provide a callable value as the `callback` argument, and the widget matches the state or combination of states specified by the *`stateSpec`* argument, that callable will be called with positional arguments `*args` and keyword arguments `**kw`. If the widget's state does not match *`stateSpec`*, the `callback` will not be called.

If you don't provide a `callback` argument, the method will return `True` if the widget's state matches *`stateSpec`*, `False` otherwise.

For the structure of the *`stateSpec`* argument, see Section 46.1, "Specifying widget states in *ttk*" (p. 146).

.state(stateSpec=None)

Use this item either to query a widget to determine its current states, or to set or clear one state.

If you provide a `stateSpec` argument of the form described in Section 46.1, "Specifying widget states in *ttk*" (p. 146), the method will set or clear states in the widget according to that argument.

For example, for a widget `w`, the method `w.state(['!disabled', 'selected'])` would clear the widget's `'disabled'` set and set its `'selected'` state.

46.1. Specifying widget states in *ttk*

Several methods within *ttk* require a *`stateSpec`* argument that specifies a particular widget state or combination of states. This argument may be any of the following:

- A single state name such as `'pressed'`. A `ttk.Button` widget is in this state, for example, when the mouse cursor is over the button and mouse button 1 is down.
- A single state name preceded with an exclamation point (`!`); this matches the widget state only when that state is *off*.

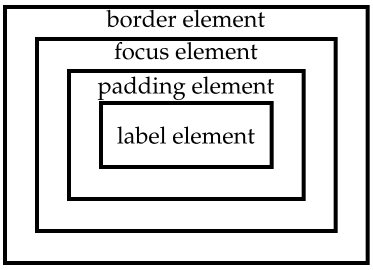
  For example, a *`stateSpec`* argument `'!pressed'` specifies a widget that is not currently being pressed.
- A sequence of state names, or state names preceded by an `'!'`. Such a *`stateSpec`* matches only when all of its components match. For example, a *`stateSpec`* value of `('!disabled', 'focus')` matches a widget only when that widget is not disabled and it has focus.

47. Customizing and creating *ttk* themes and styles

Designing with *ttk* widgets involves three levels of abstraction:

- A *theme* is a complete "look and feel," customizing the appearance of all the widgets.
- A *style* is the description of the appearance of one kind of widget. Each theme comes with a predefined set of styles, but you can customize the built-in styles or create your own new styles.

  The phrase "kind of widget" in the previous paragraph technically refers to the "class" of a widget. However, in the *ttk* world, this is different from Python classes. Within *ttk*, the class of a widget is a character string. For example, the *ttk* class of a stock `Button` widget is the string `'TButton'`.
- Each style is composed of one or more *elements*. For example, the style of a typical button has four elements: a border around the outside; a focus element that changes color when the widget has input focus; a padding element; and the button's label (text, image, or both).



We will discuss the discovery, use and customization of each of these layers in separate sections.

- Section 48, “Finding and using *ttk* themes” (p. 147).
- Section 49, “Using and customizing *ttk* styles” (p. 147).
- Section 50, “The *ttk* element layer” (p. 149).

48. Finding and using *ttk* themes

A number of operations related to themes require that you have available an instance of the `ttk.Style()` class (in the Python sense of class). For example, to obtain a list of the available themes in your installation:

```
>>> import ttk
>>> s=ttk.Style()
>>> s.theme_names()
('clam', 'alt', 'default', 'classic')
```

The `.theme_names()` method returns a tuple containing the names of the available styles. The `'classic'` theme gives you the original, pre-*ttk* appearance.

To determine which theme you get by default, use the `.theme_use()` method with no arguments. To change the current theme, call this same method with the desired theme name as the argument:

```
>>> s.theme_use()
'default'
>>> s.theme_use('alt')
>>> s.theme_use()
'alt'
```

49. Using and customizing *ttk* styles

Within a given theme, every widget has a default *widget class*; we use this term to distinguish *ttk* classes from Python classes.

Each widget also has a *style*. The default style for a widget is determined by its widget class, but you may specify a different style.

In *ttk*, widget classes and styles are specified as strings. In all but one case, the default style name of a widget is `'T'` prefixed to the widget name; for example, the default button widget class is `'TButton'`. There are some exceptions:

Table 63. Style names for *ttk* widget classes

Widget class	Style name
`Button`	`TButton`
`Checkbutton`	`TCheckbutton`
`Combobox`	`TCombobox`
`Entry`	`TEntry`
`Frame`	`TFrame`
`Label`	`TLabel`
`LabelFrame`	`TLabelFrame`
`Menubutton`	`TMenubutton`
`Notebook`	`TNotebook`
`PanedWindow`	`TPanedwindow` (*not* `TPanedWindow`!)
`Progressbar`	`Horizontal.TProgressbar` or `Vertical.TProgressbar`, depending on the `orient` option.
`Radiobutton`	`TRadiobutton`
`Scale`	`Horizontal.TScale` or `Vertical.TScale`, depending on the `orient` option.
`Scrollbar`	`Horizontal.TScrollbar` or `Vertical.TScrollbar`, depending on the `orient` option.
`Separator`	`TSeparator`
`Sizegrip`	`TSizegrip`
`Treeview`	`Treeview` (*not* `TTreview`!)

At runtime, you can retrieve a widget's widget class by calling its `.winfo_class()` method.

```
>>> b=ttk.Button(None)
>>> b.winfo_class()
'TButton'
>>> t=ttk.Treeview(None)
>>> t.winfo_class()
'Treeview'
>>> b.__class__    # Here, we are asking for the Python class
<class ttk.Button at 0x21c76d0>
```

The name of a style may have one of two forms.

- The built-in styles are all a single word: `'TFrame'` or `'TRadiobutton'`, for example.
- To create a new style derived from one of the built-in styles, use a style name of the form `'*newName.oldName*'`. For example, to create a new style of `Entry` widget to hold a date, you might call it `'Date.TEntry'`.

Every style has a corresponding set of *options* that define its appearance. For example, buttons have a `foreground` option that changes the color of the button's text.

To change the appearance of a style, use its `.configure()` method. The first argument of this method is the name of the style you want to configure, followed by keyword arguments specifying the option names and values you want to change. For example, to make all your buttons use green text, where `s` is in instance of the `ttk.Style` class:

```
s.configure('TButton', foreground='green')
```

To create a new style based on some style *oldName*, first create an instance of `ttk.Style`, then call its `.configure()` method using a name of the form `'newName.oldName'`. For example, suppose you don't want to use maroon text on all your buttons, but you do want to create a new style that does use maroon text, and you want to call the new style `'Kim.TButton'`:

```
s = ttk.Style()
s.configure('Kim.TButton', foreground='maroon')
```

Then to create a button in the new class you might use something like this:

```
self.b = ttk.Button(self, text='Friday', style='Kim.TButton',
    command=self._fridayHandler)
```

You can even build entire hierarchies of styles. For example, if you configure a style named `'Panic.Kim.TButton'`, that style will inherit all the options from the `'Kim.TButton'` style, that is, any option you don't set in the `'Panic.Kim.TButton` style will be the same as that option in the `'Kim.TButton'` style.

When *ttk* determines what value to use for an option, it looks first in the `'Panic.Kim.TButton'` style; if there is no value for that option in that style, it looks in the `'Kim.TButton'` style; and if that style doesn't define the option, it looks in the `'TButton'` style.

There is a *root style* whose name is `'.'`. To change some feature's default appearance for every widget, you can configure this style. For example, let's suppose that you want all text to be 12-point Helvetica (unless overriden by another style or `font` option). This configuration would do it:

```
    s = ttk.Style()
    s.configure('.', font=('Helvetica', 12))
```

50. The *ttk* element layer

A *ttk* element is one of the pieces that make up a widget. In order to understand how elements are assembled into styles, read these sections.

- Section 50.1, "*ttk* layouts: Structuring a style" (p. 149): the static structure of elements within a widget.
- Section 50.2, "*ttk* style maps: dynamic appearance changes" (p. 151): states of a widget and how those states affect its appearance.

50.1. *ttk* layouts: Structuring a style

In general, the pieces of a widget are assembled using the idea of a *cavity*, an empty space that is to be filled with elements.

For example, in the `classic` theme, a button has four concentric elements. From the outside in, they are the *focus highlight*, *border*, *padding*, and *label* elements.

Each of these elements has a `'sticky'` attribute that specifies how many of the four sides of the cavity it "sticks" to. For example, if an element has a `sticky='ew'` attribute, that means it must stretch in order to stick to the left (west) and right (east) sides of its cavity, but it does not have to stretch vertically.

Most of the built-in *ttk* styles use the idea of a *layout* to organize the different layers that make up a widget. Assuming that *S* is an instance of `ttk.Style`, to retrieve that style's layout use a method call of this form, where *widgetClass* is the name of the widget class.

```
S.layout(widgetClass)
```

Some widget classes don't have a layout; in those cases, this method call will raise a `tk.TclError` exception.

For the widget classes that have a layout, the returned value is a list of tuples `(eltName, d)`. Within each tuple, *`eltName`* is the name of an element and *`d`* is a dictionary that describes the element.

This dictionary may have values for the following keys:

`'sticky'`
: A string that defines how this element is to be positioned within its parent. This string may contain zero or more of the characters `'n'`, `'s'`, `'e'`, and `'w'`, referring to the sides of the box with the same conventions as for anchors. For example, the value `sticky='nsw'` would stretch this element to adhere to the north, south, and west sides of the cavity within its parent element.

`'side'`
: For elements with multiple children, this value defines how the element's children will be positioned inside it. Values may be `'left'`, `'right'`, `'top'`, or `'bottom`.

`'children'`
: If there are elements inside this element, this entry in the dictionary is the layout of the child elements using the same format as the top-level layout, that is, a list of two-element tuples `(eltName, d)`.

Let's dissect the layout of the stock `Button` widget of the `'classic'` theme in this conversational example.

```
>>> import ttk
>>> s = ttk.Style()
>>> s.theme_use('classic')
>>> b = ttk.Button(None, text='Yo')
>>> bClass = b.winfo_class()
>>> bClass
'TButton'
>>> layout = s.layout('TButton')
>>> layout
[('Button.highlight', {'children': [('Button.border', {'border':
'1', 'children': [('Button.padding', {'children': [('Button.label',
{'sticky': 'nswe'})], 'sticky': 'nswe'})], 'sticky': 'nswe'})],
'sticky': 'nswe'})]
```

All those parentheses, brackets, and braces make that structure a bit hard to understand. Here it is in outline form:

- The outermost element is the focus highlight; it has style `'Button.highlight'`. Its `'sticky'` attribute is `'nswe'`, meaning it should expand in all four directions to fill its cavity.
- The only child of the focus highlight is the border element, with style `'Button.border'`. It has a `'border'` width of 1 pixel, and its `'sticky'` attribute also specifies that it adheres to all four sides of its cavity, which is defined by the inside of the highlight element.
- Inside the border is a layer of padding, with style `'Button.padding'`. Its `sticky` attribute also specifies that it fills its cavity.
- Inside the padding layer is the text (or image, or both) that appears on the button. Its style is `'Button.label'`, with the usual `sticky='nswe'` attribute.

Each element has a dictionary of *element options* that affect the appearance of that element. The names of these options are all regular *Tkinter* options such as `'anchor'`, `'justify'`, `'background'`, or `'highlightthickness'`.

To obtain the list of option names, use a method call of this form, where *S* is an instance of class `ttk.Style`:

```
S.element_options(styleName)
```

The result is a sequence of option strings, each preceded by a hyphen. Continuing our conversational above, where `s` is an instance of `ttk.Style`:

```
>>> d = s.element_options('Button.highlight')
>>> d
('-highlightcolor', '-highlightthickness')
```

To find out what attributes are associated with an element option, use a method call of this form:

```
s.lookup(layoutName, optName)
```

Continuing our example:

```
>>> s.lookup('Button.highlight', 'highlightthickness')
1
>>> s.lookup('Button.highlight', 'highlightcolor')
'#d9d9d9'
>>> print s.element_options('Button.label')
('-compound', '-space', '-text', '-font', '-foreground', '-underline',
'-width', '-anchor', '-justify', '-wraplength', '-embossed', '-image',
'-stipple', '-background')
>>> s.lookup('Button.label', 'foreground')
'black'
```

50.2. *ttk* style maps: dynamic appearance changes

The *ttk* widgets can change their appearance during the execution of the program. For example, when a widget is *disabled*, it will not respond to mouse or keyboard actions. Typically a disabled widget presents a different appearance so that the user might realize that the widget will not respond to the mouse.

In general, every *ttk* widget has a set of *state flags* that you can use to make the appearance of a widget change during execution. Each state may be set (turned on) or reset (turned off) independently of the other states. The states and their meanings:

`active`	The mouse is currently within the widget.
`alternate`	This state is reserved for application use.
`background`	Under Windows or MacOS, the widget is located in a window that is not the foreground window.
`disabled`	The widget will not respond to user actions.
`focus`	The widget currently has focus.
`invalid`	The contents of the widget are not currently valid.
`pressed`	The widget is currently being pressed (e.g., a button that is being clicked).
`readonly`	The widget will not allow any user actions to change its current value. For example, a read-only `Entry` widget will not allow editing of its content.

`selected`	The widget is selected. Examples are checkbuttons and radiobuttons that are in the "on" state.

Some states will change in response to user actions, for example, the `pressed` state of a `Button`. Your program can interrogate, clear, or set any state by using functions described in Section 46, "Methods common to all *ttk* widgets" (p. 145).

The logic that changes the appearance of a widget is tied to one of its elements. To interrogate or set up dynamic behavior for a specific style, given an instance *s* of *ttk*`.Style`, use this method, where *styleName* is the element's name, e.g., `'Button.label'` or `'border'`.

```
    s.map(styleName, *p, **kw)
```

To determine the dynamic behavior of one option of a given style element, pass the option name as the second positional argument, and the method will return a list of state change specifications.

Each state change specification is a sequence (s_0, s_1, n). This sequence means that when the widget's current state matches all the s_i parts, set the option to the value n. Each item s_i is either a state name, or a state name preceded by a "`!`". To match, the widget must be in all the states described by items that don't start with "`!`", and it must *not* be in any of the states that start with "`!`".

For example, suppose you have an instance *s* of class `ttk.Style`, and you call it like this:

```
    changes = s.map('TCheckbutton', 'indicatorcolor')
```

Further suppose that the return value is:

```
[('pressed', '#ececec'), ('selected', '#4a6984')]
```

This means that when a checkbutton is in the `pressed` state, its `indicatorcolor` option should be set to the color `'#ececec'`, and when the checkbutton is in the `selected` state, its `indicatorcolor` option should be set to `'#4a6984'`.

You may also change the dynamic behavior of an element by passing one or more keyword arguments to the `.map()` method. For example, to get the behavior of the above example, use this method call:

```
    s.map('TCheckbutton',
        indicatoron=[('pressed', '#ececec'), ('selected', '#4a6984')])
```

Here's a more complex example. Suppose you want to create a custom button style based on the standard `TButton` class. We'll name our style `Wild.TButton`; because our name ends with "`.TButton`", it automatically inherits the standard style features. Here's how to set up this new style:

```
    s = ttk.Style()
    s.configure('Wild.TButton',
        background='black',
        foreground='white',
        highlightthickness='20',
        font=('Helvetica', 18, 'bold'))
    s.map('Wild.TButton',
        foreground=[('disabled', 'yellow'),
                    ('pressed', 'red'),
                    ('active', 'blue')],
        background=[('disabled', 'magenta'),
                    ('pressed', '!focus', 'cyan'),
                    ('active', 'green')],
        highlightcolor=[('focus', 'green'),
```

```
                      ('!focus', 'red')],
        relief=[('pressed', 'groove'),
                ('!pressed', 'ridge')])
```

- This button will initially show white text on a black background, with a 20-pixel-wide focus highlight.
- If the button is in `'disabled'` state, it will show yellow text on a magenta background.
- If the button is currently being pressed, the text will be red; provided the button does *not* have focus, the background will be cyan. The tuple `('pressed', '!focus', 'cyan')` is an example of how you can make an attribute dependent on a combination of states.
- If the button is active (under the cursor), the text will be blue on a green background.
- The focus highlight will be green when the button has focus and red when it does not.
- The button will show ridge relief when it is not being pressed, and groove relief when it is being pressed.

51. Connecting your application logic to the widgets

The preceding sections talked about how to arrange and configure the widgets—the front panel of the application.

Next, we'll talk about how to connect up the widgets to the logic that carries out the actions that the user requests.

- To make your application respond to events such as mouse clicks or keyboard inputs, there are two methods:
  - Some controls such as buttons have a `command` attribute that lets you specify a procedure, called a *handler*, that will be called whenever the user clicks that control.

    The sequence of events for using a `Button` widget is very specific, though. The user must move the mouse pointer onto the widget with mouse button 1 up, then press mouse button 1, and then release mouse button 1 while still on the widget. No other sequence of events will "press" a `Button` widget.
  - There is a much more general mechanism that can let your application react to many more kinds of inputs: the press or release of any keyboard key or mouse button; movement of the mouse into, around, or out of a widget; and many other events. As with `command` handlers, in this mechanism you write handler procedures that will be called whenever certain types of events occur. This mechanism is discussed under Section 54, "Events" (p. 157).
- Many widgets require you to use *control variables*, special objects that connect widgets together and to your program, so that you can read and set properties of the widgets. Control variables will be discussed in the next section.

52. Control variables: the values behind the widgets

A *Tkinter control variable* is a special object that acts like a regular Python variable in that it is a container for a value, such as a number or string.

One special quality of a control variable is that it can be shared by a number of different widgets, and the control variable can remember all the widgets that are currently sharing it. This means, in particular, that if your program stores a value *v* into a control variable *c* with its *c*.set(*v*) method, any widgets that are linked to that control variable are automatically updated on the screen.

Tkinter uses control variables for a number of important functions, for example:

- Checkbuttons use a control variable to hold the current state of the checkbutton (on or off).
- A single control variable is shared by a group of radiobuttons and can be used to tell which one of them is currently set. When the user clicks on one radiobutton in a group, the sharing of this control variable is the mechanism by which *Tkinter* groups radiobuttons so that when you set one, any other set radiobutton in the group is cleared.
- Control variables hold text string for several applications. Normally the text displayed in an `Entry` widget is linked to a control variable. In several other controls, it is possible to use a string-valued control variable to hold text such as the labels of checkbuttons and radiobuttons and the content of `Label` widgets.

  For example, you could link an `Entry` widget to a `Label` widget so that when the user changes the text in the entry and presses the *Enter* key, the label is automatically updated to show that same text.

To get a control variable, use one of these class constructors, depending on what type of values you need to store in it:

```
v = tk.DoubleVar()   # Holds a float; default value 0.0
v = tk.IntVar()      # Holds an int; default value 0
v = tk.StringVar()   # Holds a string; default value ''
```

All control variables have these two methods:

`.get()`
Returns the current value of the variable.

`.set(`*`value`*`)`
Changes the current value of the variable. If any widget options are slaved to this variable, those widgets will be updated when the main loop next idles; see .update_idletasks() in Section 26, “Universal widget methods” (p. 97) for more information on controlling this update cycle.

Here are some comments on how control variables are used with specific widgets:

`Button`
You can set its `textvariable` to a `StringVar`. Anytime that variable is changed, the text on the button will be updated to display the new value. This is not necessary unless the button's text is actually going to change: use the `text` attribute if the button's label is static.

`Checkbutton`
Normally, you will set the widget's `variable` option to an `IntVar`, and that variable will be set to 1 when the checkbutton is turned on and to 0 when it is turned off. However, you can pick different values for those two states with the `onvalue` and `offvalue` options, respectively.

You can even use a `StringVar` as the checkbutton's variable, and supply string values for the `offvalue` and `onvalue`. Here's an example:

```
self.spamVar = tk.StringVar()
self.spamCB  = tk.Checkbutton(self, text='Spam?',
    variable=self.spamVar, onvalue='yes', offvalue='no')
```

If this checkbutton is on, `self.spamVar.get()` will return the string `'yes'`; if the checkbutton is off, that same call will return the string `'no'`. Furthermore, your program can turn the checkbutton on by calling `.set('yes')`.

You can also the `textvariable` option of a checkbutton to a `StringVar`. Then you can change the text label on that checkbutton using the `.set()` method on that variable.

`Entry`
Set its `textvariable` option to a `StringVar`. Use that variable's `.get()` method to retrieve the text currently displayed in the widget. You can also the variable's `.set()` method to change the text displayed in the widget.

`Label`
You can set its `textvariable` option to a `StringVar`. Then any call to the variable's `.set()` method will change the text displayed on the label. This is not necessary if the label's text is static; use the `text` attribute for labels that don't change while the application is running.

`Menubutton`
If you want to be able to change the text displayed on the menu button, set its `textvariable` option to a `StringVar` and use that variable's `.set()` method to change the displayed text.

`Radiobutton`
The `variable` option must be set to a control variable, either an `IntVar` or a `StringVar`. All the radiobuttons in a functional group must share the same control variable.

Set the `value` option of each radiobutton in the group to a different value. Whenever the user sets a radiobutton, the variable will be set to the `value` option of that radiobutton, and all the other radiobuttons that share the group will be cleared.

You might wonder, what state is a group of radiobuttons in when the control variable has never been set and the user has never clicked on them? Each control variable has a default value: `0` for an `IntVar`, `0.0` for a `DoubleVar`, and `''` for a `StringVar`. If one of the radiobuttons has that `value`, that radiobutton will be set initially. If no radiobutton's `value` option matches the value of the variable, the radiobuttons will all appear to be cleared.

If you want to change the text label on a radiobutton during the execution of your application, set its `textvariable` option to a `StringVar`. Then your program can change the text label by passing the new label text to the variable's `.set()` method.

`Scale`
For a scale widget, set its `variable` option to a control variable of any class, and set its `from_` and `to` options to the limiting values for the opposite ends of the scale.

For example, you could use an `IntVar` and set the scale's `from_=0` and `to=100`. Then every user change to the widget would change the variable's value to some value between 0 and 100 inclusive.

Your program can also move the slider by using the `.set()` method on the control variable. To continue the above example, `.set(75)` would move the slider to a position three-fourths of the way along its trough.

To set up a `Scale` widget for `float` values, use a `DoubleVar`.

You can use a `StringVar` as the control variable of a `Scale` widget. You will still need to provide numeric `from_` and `to` values, but the numeric value of the widget will be converted to a string for storage in the `StringVar`. Use the scale's `digits` option to control the precision of this conversion.

53. Focus: routing keyboard input

To say a widget has *focus* means that keyboard input is currently directed to that widget.

- By *focus traversal*, we mean the sequence of widgets that will be visited as the user moves from widget to widget with the *tab* key. See below for the rules for this sequence.
- You can traverse backwards using *shift-tab*.

- The `Entry` and `Text` widgets are intended to accept keyboard input, and if an entry or text widget currently has the focus, any characters you type into it will be added to its text. The usual editing characters such as ← and → will have their usual effects.
- Because `Text` widgets can contain tab characters, you must use the special key sequence *control-tab* to move the focus past a text widget.
- Most of the other types of widgets will normally be visited by focus traversal, and when they have focus:
  - `Button` widgets can be "pressed" by pressing the spacebar.
  - `Checkbutton` widgets can be toggled between set and cleared states using the spacebar.
  - In `Listbox` widgets, the ↑ and ↓ keys scroll up or down one line; the *PageUp* and *PageDown* keys scroll by pages; and the spacebar selects the current line, or de-selects it if it was already selected.
  - You can set a `Radiobutton` widget by pressing the spacebar.
  - Horizontal `Scale` widgets respond to the ← and → keys, and vertical ones respond to ↑ and ↓.
  - In a `Scrollbar` widget, the *PageUp* and *PageDown* keys move the scrollbar by pageloads. The ↑ and ↓ keys will move vertical scrollbars by units, and the ← and → keys will move horizontal scrollbars by units.
- Many widgets are provided with an outline called the *focus highlight* that shows the user which widget has the highlight. This is normally a thin black frame located just outside the widget's border (if any). For widgets that don't normally have a focus highlight (specifically, frames, labels, and menus), you can set the `highlightthickness` option to a nonzero value to make the focus highlight visible.
- You can also change the color of the focus highlight using the `highlightcolor` option.
- Widgets of class `Frame`, `Label`, and `Menu` are not normally visited by the focus. However, you can set their `takefocus` options to `1` to get them included in focus traversal. You can also take any widget out of focus traversal by setting its `takefocus` option to `0`.

The order in which the *tab* key traverses the widgets is:

- For widgets that are children of the same parent, focus goes in the same order the widgets were created.
- For parent widgets that contain other widgets (such as frames), focus visits the parent widget first (unless its `takefocus` option is `0`), then it visits the child widgets, recursively, in the order they were created.

To sum up: to set up the focus traversal order of your widgets, create them in that order. Remove widgets from the traversal order by setting their `takefocus` options to 0, and for those whose default `takefocus` option is `0`, set it to `1` if you want to add them to the order.

The above describes the default functioning of input focus in *Tkinter*. There is another, completely different way to handle it—let the focus go wherever the mouse goes. Under Section 26, "Universal widget methods" (p. 97), refer to the `.tk_focusFollowsMouse()` method.

You can also add, change or delete the way any key on the keyboard functions inside any widget by using event bindings. See Section 54, "Events" (p. 157) for the details.

53.1. Focus in *ttk* widgets

If you create a *ttk* widget and do not specify its `takefocus` option, by default, all *ttk* widgets get focus except for `Frame`, `Label`, `LabelFrame`, `PanedWindow`, `Progressbar`, `Scrollbar`, `Separator`, and `Sizegrip`.

54. Events: responding to stimuli

An *event* is something that happens to your application—for example, the user presses a key or clicks or drags the mouse—to which the application needs to react.

The widgets normally have a lot of built-in behaviors. For example, a button will react to a mouse click by calling its `command` callback. For another example, if you move the focus to an entry widget and press a letter, that letter gets added to the content of the widget.

However, the event binding capability of *Tkinter* allows you to add, change, or delete behaviors.

First, some definitions:

- An *event* is some occurrence that your application needs to know about.
- An *event handler* is a function in your application that gets called when an event occurs.
- We call it *binding* when your application sets up an event handler that gets called when an event happens to a widget.

54.1. Levels of binding

You can bind a handler to an event at any of three levels:

1. Instance binding: You can bind an event to one specific widget. For example, you might bind the *PageUp* key in a canvas widget to a handler that makes the canvas scroll up one page. To bind an event of a widget, call the `.bind()` method on that widget (see Section 26, "Universal widget methods" (p. 97)).

   For example, suppose you have a canvas widget named `self.canv` and you want to draw an orange blob on the canvas whenever the user clicks the mouse button 2 (the middle button). To implement this behavior:

   ```
   self.canv.bind('<Button-2>', self.__drawOrangeBlob)
   ```

   The first argument is a *sequence descriptor* that tells *Tkinter* that whenever the middle mouse button goes down, it is to call the *event handler* named `self.__drawOrangeBlob`. (See Section 54.6, "Writing your handler: The `Event` class" (p. 162), below, for an overview of how to write handlers such as `.__drawOrangeBlob()`). Note that you omit the parentheses after the handler name, so that Python will pass in a reference the handler instead of trying to call it right away.

2. Class binding: You can bind an event to all widgets of a class. For example, you might set up all `Button` widgets to respond to middle mouse button clicks by changing back and forth between English and Japanese labels. To bind an event to all widgets of a class, call the `.bind_class()` method on any widget (see Section 26, "Universal widget methods" (p. 97), above).

   For example, suppose you have several canvases, and you want to set up mouse button 2 to draw an orange blob in any of them. Rather than having to call `.bind()` for every one of them, you can set them all up with one call something like this:

   ```
   self.bind_class('Canvas', '<Button-2>',
                       self.__drawOrangeBlob)
   ```

3. Application binding: You can set up a binding so that a certain event calls a handler no matter what widget has the focus or is under the mouse. For example, you might bind the *PrintScrn* key to all the widgets of an application, so that it prints the screen no matter what widget gets that key. To bind

an event at the application level, call the `.bind_all()` method on any widget (see Section 26, "Universal widget methods" (p. 97)).

Here's how you might bind the *PrintScrn* key, whose "key name" is `'Print'`:

```
    self.bind_all('<Key-Print>', self.__printScreen)
```

54.2. Event sequences

Tkinter has a powerful and general method for allowing you to define exactly which events, both specific and general, you want to bind to handlers.

In general, an event sequence is a string containing one or more *event patterns*. Each event pattern describes one thing that can happen. If there is more than one event pattern in a sequence, the handler will be called only when all the patterns happen in that same sequence.

The general form of an event pattern is:

```
<[modifier-]...type[-detail]>
```

- The entire pattern is enclosed inside **<…>**.
- The *event type* describes the general kind of event, such as a key press or mouse click. See Section 54.3, "Event types" (p. 158).
- You can add optional *`modifier`* items before the type to specify combinations such as the *shift* or *control* keys being depressed during other key presses or mouse clicks. Section 54.4, "Event modifiers" (p. 160)
- You can add optional *`detail`* items to describe what key or mouse button you're looking for. For mouse buttons, this is 1 for button 1, 2 for button 2, or 3 for button 3.
  - The usual setup has button 1 on the left and button 3 on the right, but left-handers can swap these positions.
  - For keys on the keyboard, this is either the key's character (for single-character keys like the `A` or * key) or the key's name; see Section 54.5, "Key names" (p. 160) for a list of all key names.

Here are some examples to give you the flavor of event patterns:

`<Button-1>`	The user pressed the first mouse button.
`<KeyPress-H>`	The user pressed the H key.
`<Control-Shift-KeyPress-H>`	The user pressed *control*-*shift*-H.

54.3. Event types

The full set of event types is rather large, but a lot of them are not commonly used. Here are most of the ones you'll need:

Type	Name	Description
36	`Activate`	A widget is changing from being inactive to being active. This refers to changes in the `state` option of a widget such as a button changing from inactive (grayed out) to active.

Type	Name	Description
4	`Button`	The user pressed one of the mouse buttons. The `detail` part specifies which button. For mouse wheel support under Linux, use `Button-4` (scroll up) and `Button-5` (scroll down). Under Linux, your handler for mouse wheel bindings will distinguish between scroll-up and scroll-down by examining the `.num` field of the `Event` instance; see Section 54.6, "Writing your handler: The `Event` class" (p. 162).
5	`ButtonRelease`	The user let up on a mouse button. This is probably a better choice in most cases than the `Button` event, because if the user accidentally presses the button, they can move the mouse off the widget to avoid setting off the event.
22	`Configure`	The user changed the size of a widget, for example by dragging a corner or side of the window.
37	`Deactivate`	A widget is changing from being active to being inactive. This refers to changes in the `state` option of a widget such as a radiobutton changing from active to inactive (grayed out).
17	`Destroy`	A widget is being destroyed.
7	`Enter`	The user moved the mouse pointer into a visible part of a widget. (This is different than the *enter* key, which is a `KeyPress` event for a key whose name is actually `'return'`.)
12	`Expose`	This event occurs whenever at least some part of your application or widget becomes visible after having been covered up by another window.
9	`FocusIn`	A widget got the input focus (see Section 53, "Focus: routing keyboard input" (p. 155) for a general introduction to input focus.) This can happen either in response to a user event (like using the *tab* key to move focus between widgets) or programmatically (for example, your program calls the `.focus_set()` on a widget).
10	`FocusOut`	The input focus was moved out of a widget. As with `FocusIn`, the user can cause this event, or your program can cause it.
2	`KeyPress`	The user pressed a key on the keyboard. The `detail` part specifies which key. This keyword may be abbreviated `Key`.
3	`KeyRelease`	The user let up on a key.
8	`Leave`	The user moved the mouse pointer out of a widget.
19	`Map`	A widget is being mapped, that is, made visible in the application. This will happen, for example, when you call the widget's `.grid()` method.
6	`Motion`	The user moved the mouse pointer entirely within a widget.
38	`MouseWheel`	The user moved the mouse wheel up or down. At present, this binding works on Windows and MacOS, but not under Linux. For Windows and MacOS, see the discussion of the `.delta` field of the `Event` instance in Section 54.6, "Writing your handler: The `Event` class" (p. 162). For Linux, see the note above under `Button`.
18	`Unmap`	A widget is being unmapped and is no longer visible. This happens, for example, when you use the widget's `.grid_remove()` method.
15	`Visibility`	Happens when at least some part of the application window becomes visible on the screen.

54.4. Event modifiers

The modifier names that you can use in event sequences include:

`Alt`	True when the user is holding the *alt* key down.
`Any`	This modifier generalizes an event type. For example, the event pattern `'<Any-KeyPress>'` applies to the pressing of any key.
`Control`	True when the user is holding the *control* key down.
`Double`	Specifies two events happening close together in time. For example, `<Double-Button-1>` describes two presses of button 1 in rapid succession.
`Lock`	True when the user has pressed *shift lock*.
`Shift`	True when the user is holding down the *shift* key.
`Triple`	Like `Double`, but specifies three events in rapid succession.

You can use shorter forms of the events. Here are some examples:

- `'<1>'` is the same as `'<Button-1>'`.
- `'x'` is the same as `'<KeyPress-x>'`.

Note that you can leave out the enclosing `'<…>'` for most single-character keypresses, but you can't do that for the space character (whose name is `'<space>'`) or the less-than (`<`) character (whose name is `'<less>'`).

54.5. Key names

The detail part of an event pattern for a `KeyPress` or `KeyRelease` event specifies which key you're binding. (See the `Any` modifier, above, if you want to get all keypresses or key releases).

The table below shows several different ways to name keys. See Section 54.6, "Writing your handler: The `Event` class" (p. 162), below, for more information on `Event` objects, whose attributes will describe keys in these same ways.

- The `.keysym` column shows the "key symbol", a string name for the key. This corresponds to the `.keysym` attribute of the `Event` object.
- The `.keycode` column is the "key code." This identifies which key was pressed, but the code does not reflect the state of various modifiers like the shift and control keys and the *NumLock* key. So, for example, both `a` and `A` have the same key code.
- The `.keysym_num` column shows a numeric code equivalent to the key symbol. Unlike `.keycode`, these codes are different for different modifiers. For example, the digit 2 on the numeric keypad (key symbol `KP_2`) and the down arrow on the numeric keypad (key symbol `KP_Down`) have the same key code (88), but different `.keysym_num` values (65433 and 65458, respectively).
- The "Key" column shows the text you will usually find on the physical key, such as *tab*.

There are many more key names for international character sets. This table shows only the "Latin-1" set for the usual USA-type 101-key keyboard. For the currently supported set, see the manual page for Tk `keysym` values[10].

[10] http://www.tcl.tk/man/tcl8.4/TkCmd/keysyms.htm

.keysym	.keycode	.keysym_num	Key
Alt_L	64	65513	The left-hand *alt* key
Alt_R	113	65514	The right-hand *alt* key
BackSpace	22	65288	*backspace*
Cancel	110	65387	*break*
Caps_Lock	66	65549	*CapsLock*
Control_L	37	65507	The left-hand *control* key
Control_R	109	65508	The right-hand *control* key
Delete	107	65535	*Delete*
Down	104	65364	↓
End	103	65367	*end*
Escape	9	65307	*esc*
Execute	111	65378	*SysReq*
F1	67	65470	Function key *F1*
F2	68	65471	Function key *F2*
F_i	66+i	65469+i	Function key F_i
F12	96	65481	Function key *F12*
Home	97	65360	*home*
Insert	106	65379	*insert*
Left	100	65361	←
Linefeed	54	106	Linefeed (*control-J*)
KP_0	90	65438	0 on the keypad
KP_1	87	65436	1 on the keypad
KP_2	88	65433	2 on the keypad
KP_3	89	65435	3 on the keypad
KP_4	83	65430	4 on the keypad
KP_5	84	65437	5 on the keypad
KP_6	85	65432	6 on the keypad
KP_7	79	65429	7 on the keypad
KP_8	80	65431	8 on the keypad
KP_9	81	65434	9 on the keypad
KP_Add	86	65451	+ on the keypad
KP_Begin	84	65437	The center key (same key as 5) on the keypad
KP_Decimal	91	65439	Decimal (.) on the keypad
KP_Delete	91	65439	*delete* on the keypad
KP_Divide	112	65455	/ on the keypad
KP_Down	88	65433	↓ on the keypad
KP_End	87	65436	*end* on the keypad

.keysym	.keycode	.keysym_num	Key
KP_Enter	108	65421	*enter* on the keypad
KP_Home	79	65429	*home* on the keypad
KP_Insert	90	65438	*insert* on the keypad
KP_Left	83	65430	← on the keypad
KP_Multiply	63	65450	× on the keypad
KP_Next	89	65435	*PageDown* on the keypad
KP_Prior	81	65434	*PageUp* on the keypad
KP_Right	85	65432	→ on the keypad
KP_Subtract	82	65453	- on the keypad
KP_Up	80	65431	↑ on the keypad
Next	105	65366	*PageDown*
Num_Lock	77	65407	*NumLock*
Pause	110	65299	*pause*
Print	111	65377	*PrintScrn*
Prior	99	65365	*PageUp*
Return	36	65293	The *enter* key (*control-M*). The name `Enter` refers to a mouse-related event, not a keypress; see Section 54, “Events” (p. 157)
Right	102	65363	→
Scroll_Lock	78	65300	*ScrollLock*
Shift_L	50	65505	The left-hand *shift* key
Shift_R	62	65506	The right-hand *shift* key
Tab	23	65289	The *tab* key
Up	98	65362	↑

54.6. Writing your handler: The `Event` class

The sections above tell you how to describe what events you want to handle, and how to bind them. Now let us turn to the writing of the handler that will be called when the event actually happens.

The handler will be passed an `Event` object that describes what happened. The handler can be either a function or a method. Here is the calling sequence for a regular function:

```
def handlerName(event):
```

And as a method:

```
    def handlerName(self, event):
```

The attributes of the `Event` object passed to the handler are described below. Some of these attributes are always set, but some are set only for certain types of events.

`.char`	If the event was related to a `KeyPress` or `KeyRelease` for a key that produces a regular ASCII character, this string will be set to that character. (For special keys like *delete*, see the `.keysym` attribute, below.)
`.delta`	For `MouseWheel` events, this attribute contains an integer whose sign is positive to scroll up, negative to scroll down. Under Windows, this value will be a multiple of 120; for example, 120 means scroll up one step, and -240 means scroll down two steps. Under MacOS, it will be a multiple of 1, so 1 means scroll up one step, and -2 means scroll down two steps. For Linux mouse wheel support, see the note on the `Button` event binding in Section 54.3, “Event types” (p. 158).
`.height`	If the event was a `Configure`, this attribute is set to the widget's new height in pixels.
`.keycode`	For `KeyPress` or `KeyRelease` events, this attribute is set to a numeric code that identifies the key. However, it does not identify which of the characters on that key were produced, so that “`x`” and “`X`” have the same `.keyCode` value. For the possible values of this field, see Section 54.5, “Key names” (p. 160).
`.keysym`	For `KeyPress` or `KeyRelease` events involving a special key, this attribute is set to the key's string name, e.g., `'Prior'` for the *PageUp* key. See Section 54.5, “Key names” (p. 160) for a complete list of `.keysym` names.
`.keysym_num`	For `KeyPress` or `KeyRelease` events, this is set to a numeric version of the `.keysym` field. For regular keys that produce a single character, this field is set to the integer value of the key's ASCII code. For special keys, refer to Section 54.5, “Key names” (p. 160).
`.num`	If the event was related to a mouse button, this attribute is set to the button number (1, 2, or 3). For mouse wheel support under Linux, bind `Button-4` and `Button-5` events; when the mouse wheel is scrolled up, this field will be 4, or 5 when scrolled down.
`.serial`	An integer serial number that is incremented every time the server processes a client request. You can use `.serial` values to find the exact time sequence of events: those with lower values happened sooner.
`.state`	An integer describing the state of all the modifier keys. See the table of modifier masks below for the interpretation of this value.
`.time`	This attribute is set to an integer which has no absolute meaning, but is incremented every millisecond. This allows your application to determine, for example, the length of time between two mouse clicks.
`.type`	A numeric code describing the type of event. For the interpretation of this code, see Section 54.3, “Event types” (p. 158).
`.widget`	Always set to the widget that caused the event. For example, if the event was a mouse click that happened on a canvas, this attribute will be the actual `Canvas` widget.
`.width`	If the event was a `Configure`, this attribute is set to the widget's new width in pixels.
`.x`	The *x* coordinate of the mouse at the time of the event, relative to the upper left corner of the widget.
`.y`	The *y* coordinate of the mouse at the time of the event, relative to the upper left corner of the widget.
`.x_root`	The *x* coordinate of the mouse at the time of the event, relative to the upper left corner of the screen.
`.y_root`	The *y* coordinate of the mouse at the time of the event, relative to the upper left corner of the screen.

Use these masks to test the bits of the `.state` value to see what modifier keys and buttons were pressed during the event:

Mask	Modifier
`0x0001`	*Shift*.
`0x0002`	*Caps Lock*.
`0x0004`	*Control*.
`0x0008`	Left-hand *Alt*.
`0x0010`	*Num Lock*.
`0x0080`	Right-hand *Alt*.
`0x0100`	Mouse button 1.
`0x0200`	Mouse button 2.
`0x0400`	Mouse button 3.

Here's an example of an event handler. Under Section 54.1, "Levels of binding" (p. 157), above, there is an example showing how to bind mouse button 2 clicks on a canvas named `self.canv` to a handler called `self.__drawOrangeBlob()`. Here is that handler:

```
    def __drawOrangeBlob(self, event):
        '''Draws an orange blob in self.canv where the mouse is.
        '''
        r = 5   # Blob radius
        self.canv.create_oval(event.x-r, event.y-r,
            event.x+r, event.y+r, fill='orange')
```

When this handler is called, the current mouse position is `(event.x, event.y)`. The `.create_oval()` method draws a circle whose bounding box is square and centered on that position and has sides of length `2*r`.

54.7. The extra arguments trick

Sometimes you would like to pass other arguments to a handler besides the event.

Here is an example. Suppose your application has an array of ten checkbuttons whose widgets are stored in a list `self.cbList`, indexed by the checkbutton number in `range(10)`.

Suppose further that you want to write one handler named `.__cbHandler` for `<Button-1>` events in all ten of these checkbuttons. The handler can get the actual `Checkbutton` widget that triggered it by referring to the `.widget` attribute of the `Event` object that gets passed in, but how does it find out that checkbutton's index in `self.cbList`?

It would be nice to write our handler with an extra argument for the checkbutton number, something like this:

```
    def __cbHandler(self, event, cbNumber):
```

But event handlers are passed only one argument, the event. So we can't use the function above because of a mismatch in the number of arguments.

Fortunately, Python's ability to provide default values for function arguments gives us a way out. Have a look at this code:

```
    def __createWidgets(self):
        …
        self.cbList = []    # Create the checkbutton list
        for i in range(10):
            cb = tk.Checkbutton(self, …)
            self.cbList.append(cb)
            cb.grid( row=1, column=i)
            def handler(event, self=self, i=i):    1
                return self.__cbHandler(event, i)
            cb.bind('<Button-1>', handler)
        …
    def __cbHandler(self, event, cbNumber):
        …
```

1 These lines define a new function `handler` that expects three arguments. The first argument is the `Event` object passed to all event handlers, and the second and third arguments will be set to their default values—the extra arguments we need to pass it.

This technique can be extended to supply any number of additional arguments to handlers.

54.8. Virtual events

You can create your own new kinds of events called *virtual events*. You can give them any name you want so long as it is enclosed in double pairs of **<<…>>**.

For example, suppose you want to create a new event called `<<panic>>`, that is triggered either by mouse button 3 or by the *pause* key. To create this event, call this method on any widget *w*:

```
    w.event_add('<<panic>>', '<Button-3>',
                '<KeyPress-Pause>')
```

You can then use `'<<panic>>'` in any event sequence. For example, if you use this call:

```
    w.bind('<<panic>>', h)
```

any mouse button 3 or *pause* keypress in widget *w* will trigger the handler *h*.

See `.event_add()`, `.event_delete()`, and `.event_info()` under Section 26, "Universal widget methods" (p. 97) for more information about creating and managing virtual events.

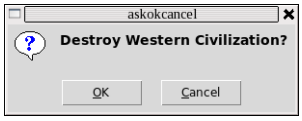
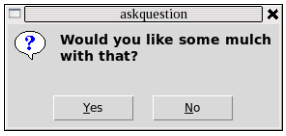
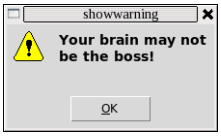
55. Pop-up dialogs

Tkinter provides three modules that can create pop-up dialog windows for you:

- Section 55.1, "The `tkMessageBox` dialogs module" (p. 165), provides an assortment of common pop-ups for simple tasks.
- Section 55.2, "The `tkFileDialog` module" (p. 167), allows the user to browse for files.
- Section 55.3, "The `tkColorChooser` module" (p. 168), allows the user to select a color.

55.1. The `tkMessageBox` dialogs module

Once you import the `tkMessageBox` module, you can create any of these seven common types of pop-up menu by calling functions from this table.

	`.askokcancel(`*`title`*`, `*`message`*`, `*`options`*`)`
	`.askquestion(`*`title`*`, `*`message`*`, `*`options`*`)`
	`.askretrycancel(`*`title`*`, `*`message`*`, `*`options`*`)`
	`.askyesno(`*`title`*`, `*`message`*`, `*`options`*`)`
	`.showerror(`*`title`*`, `*`message`*`, `*`options`*`)`
	`.showinfo(`*`title`*`, `*`message`*`, `*`options`*`)`
	`.showwarning(`*`title`*`, `*`message`*`, `*`options`*`)`

In each case, the *`title`* is a string to be displayed in the top of the window decoration. The *`message`* argument is a string that appears in the body of the pop-up window; within this string, lines are broken at newline (`'\n'`) characters.

The *`option`* arguments may be any of these choices.

`default`
: Which button should be the default choice? If you do not specify this option, the first button ("OK", "Yes", or "Retry") will be the default choice.

  To specify which button is the default choice, use `default=`*`C`*, where *`C`* is one of these constants defined in `tkMessageBox`: `CANCEL`, `IGNORE`, `OK`, `NO`, `RETRY`, or `YES`.

`icon`
: Selects which icon appears in the pop-up. Use an argument of the form `icon=`*`I`* where *`I`* is one of these constants defined in `tkMessageBox`: `ERROR`, `INFO`, `QUESTION`, or `WARNING`.

`parent`
: If you don't specify this option, the pop-up appears above your root window. To make the pop-up appear above some child window *`W`*, use the argument `parent=`*`W`*.

Each of the "`ask...`" pop-up functions returns a value that depends on which button the user pushed to remove the pop-up.

- `askokcancel`, `askretrycancel`, and `askyesno` all return a `bool` value: `True` for "OK" or "Yes" choices, `False` for "No" or "Cancel" choices.
- `askquestion` returns `u'yes'` for "Yes", or `u'no'` for "No".

55.2. The `tkFileDialog` module

The `tkFileDialog` module provides two different pop-up windows you can use to give the user the ability to find existing files or create new files.

`.askopenfilename(`*`option`*`=`*`value`*`, ...)`
Intended for cases where the user wants to select an existing file. If the user selects a nonexistent file, a popup will appear informing them that the selected file does not exist.

`.asksaveasfilename(`*`option`*`=`*`value`*`, ...)`
Intended for cases where the user wants to create a new file or replace an existing file. If the user selects an existing file, a pop-up will appear informing that the file already exists, and asking if they really want to replace it.

The arguments to both functions are the same:

`defaultextension=`*`s`*
The default file extension, a string starting with a period (`'.'`). If the user's reply contains a period, this argument has no effect. It is appended to the user's reply in case there are no periods.

For example, if you supply a `defaultextension='.jpg'` argument and the user enters `'gojiro'`, the returned file name will be `'gojiro.jpg'`.

`filetypes=[(`*`label`*$_1$`, `*`pattern`*$_1$`), (`*`label`*$_2$`, `*`pattern`*$_2$`), ...]`
A list of two-element tuples containing file type names and patterns that will select what appears in the file listing. In the screen picture below, note the pull-down menu labeled "Files of type:". The `filetypes` argument you supply will populate this pull-down list. Each *`pattern`* is a file type name ("PNG" in the example) and a pattern that selects files of a given type ("(*.png)" in the example).

`initialdir=`*`D`*
The path name of the directory to be displayed initially. The default directory is the current working directory.

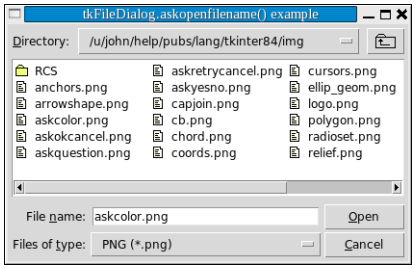
`initialfile=`*`F`*
The file name to be displayed initially in the "File name:" field, if any.

`parent=`*`W`*
To make the pop-up appear over some window *`W`*, supply this argument. The default behavior is that the pop-up will appear over your application's root window.

`title=`*`T`*
If specified, *`T`* is a string to be displayed as the pop-up window's title.

If the user selects a file, the returned value is the complete path name of the selected file. If the user uses the *Cancel* button, the function returns an empty string.

Here is an example:



55.3. The `tkColorChooser` module

To give your application's user a popup they can use to select a color, import the `tkColorChooser` module and call this function:

```
result = tkColorChooser.askcolor(color, option=value, ...)
```

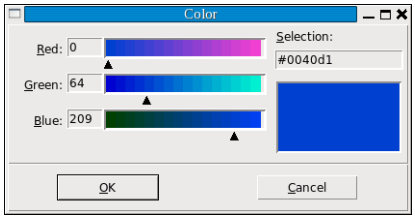
Arguments are:

color
The initial color to be displayed. The default initial color is a light gray.

`title=`*`text`*
The specified *`text`* appears in the pop-up window's title area. The default title is "Color".

`parent=`*`W`*
Make the popup appear over window *`W`*. The default behavior is that it appears over your root window.

If the user clicks the *OK* button on the pop-up, the returned value will be a tuple `(`*`triple`*`, color)`, where *`triple`* is a tuple `(`*`R, G, B`*`)` containing red, green, and blue values in the range [0,255] respectively, and *`color`* is the selected color as a regular *Tkinter* color object.

If the users clicks *Cancel*, this function will return `(None, None)`.

Here's what the popup looks like on the author's system: