Tkinter reference: a GUI for Python

John W. Shipman

2006-12-05 16:41

Table of Contents

1. What is Tkinter?

Tkinter is a GUI (graphical user interface) widget set for Python. This document contains only the commoner features.

This document applies to Python 1.5 and Tkinter 8.0.4 running in the X Window system under Linux. Your version may vary.

Pertinent references:

- *An Introduction to Tkinter*[1] by Fredrik Lundh
- *Python and Tkinter Programming* by John Grayson (Manning, 2000, ISBN 1-884777-81-3).
- *Python 2.2 quick reference*[2]: general information about the Python language.

[1] http://www.pythonware.com/library/tkinter/introduction/index.htm
[2] http://www.nmt.edu/tcc/help/pubs/python22/

- This publication is available in Web form[3] and also as a PDF document[4]. Please forward any comments to **tcc-doc@nmt.edu**.

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/local/bin/python      1
from Tkinter import *        2

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

    def createWidgets(self):
        self.quitButton = 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 the Python interpreter at path `/usr/local/bin/python`.
2. This line imports the entire Tkinter package into your program's namespace.
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. Layout management

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

[3] http://www.nmt.edu/tcc/help/pubs/tkinter/
[4] http://www.nmt.edu/tcc/help/pubs/tkinter/tkinter.pdf

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:

```
thing = Constructor(master, ...)
thing.grid(...)
```

where **`Constructor`** is one of the widget classes like **`Button`**, **`Frame`**, and so on, and **`master`** 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.

3.1. The `.grid()` method

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

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

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.

Here are the options to the **`.grid()`** geometry management method:

`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.
`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=NE** (top right), **SE** (bottom right), **SW** (bottom left), or **NW** (top left).
- You can position the widget centered against one side of the cell by using **sticky=N** (top center), **E** (right center), **S** (bottom center), or **W** (left center).
- Use **sticky=N+S** to stretch the widget vertically but leave it centered horizontally.
- Use **sticky=E+W** to stretch it horizontally but leave it centered vertically.
- Use **sticky=N+E+S+W** to stretch the widget both horizontally and vertically to fill the cell.
- The other combinations will also work. For example, **sticky=N+S+W** will stretch the widget vertically and place it against the west (left) wall.

3.2. Other grid management methods

These grid-related methods are defined on all widgets:

***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_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.

3.3. 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 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.

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.

3.4. 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 3.3, “Configuring column and row sizes” (p. 5), 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. 3), 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=N+S+E+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 19, “Universal widget methods” (p. 64).

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=N+S+E+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):
        Frame.__init__(self, master)
        self.grid(sticky=N+S+E+W)
        self.createWidgets()
```

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

4. 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, and retrieve the current setting of any option by using the widget's **`.cget()`** method. See Section 19, “Universal widget methods” (p. 64) for more on these methods.

4.1. Dimensions

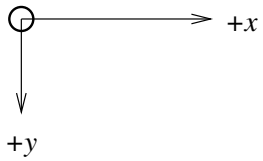
Various lengths, widths, and other dimensions of widgets can be described in many different units.

- 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:

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

4.2. 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 4.1, “Dimensions” (p. 7).

4.3. 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.

4.4. 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, 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.

- 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 the author's favorite 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:

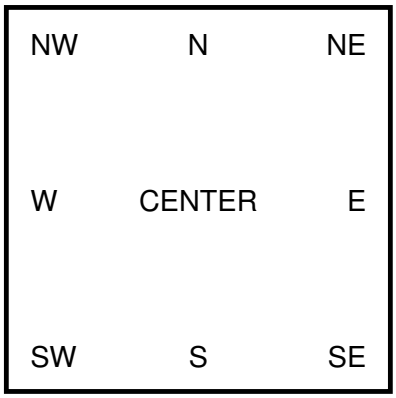
ascent	Number of pixels of height between the baseline and the top of the highest ascender.
descent	Number of pixels of height between the baseline and the bottom of the lowest ascender.
fixed	This value is **0** for a variable-width font and **1** for a monospaced font.
linespace	Number of pixels of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

4.5. Anchors

Constants are defined in the **Tkinter** package 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.

The anchor constants are shown in this diagram:

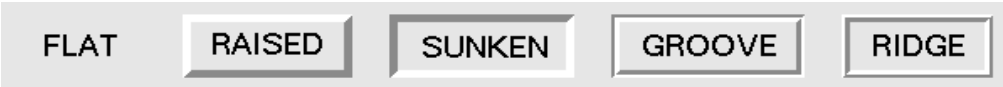


For example, if you create a small widget inside a large frame and use the **anchor=SE** option, the widget will be placed in the bottom right corner of the frame. If you used **anchor=N** instead, the widget would be centered along the top edge.

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

4.6. Relief styles

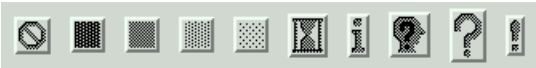
The *relief style* of a widget refers to certain simulated 3-D effects around the outside of the widget. Here is a screen shot of a row of buttons exhibiting all the possible relief styles:



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

4.7. Bitmaps

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



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.

4.8. 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.

 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	

4.9. Images

To use graphic images in a Tkinter application, Tkinter must be configured to include the Python Imaging Library (PIL).

Refer to the author's companion document for PIL documentation: *Python Imaging Library (PIL) quick reference*[5]. Objects of the **ImageTk** class can be used in Tkinter applications.

4.10. 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"**.

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

- 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.

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.

4.11. 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"**.

The path name for any widget ***w*** can be determined by calling **str(*w*)**.

See also Section 19, "Universal widget methods" (p. 64) for methods you can use to operate on window names, especially the **.winfo_name**, **.winfo_parent**, and **.winfo_pathname** methods.

5. The Button widget

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

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

The constructor returns the new button object. Its options include:

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 4.5, "Anchors" (p. 9). For example, **anchor=NE** would position the text at the top right corner of the button.
bd or **borderwidth**	Border width in pixels. Default is 2.

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	**NORMAL** is the default; use **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: **LEFT** to left-justify each line; **CENTER** to center them; or **RIGHT** to right-justify.
padx	Additional padding left and right of the text. See Section 4.1, "Dimensions" (p. 7) for the possible values for padding.
pady	Additional padding above and below the text.
relief	Specifies the relief type for the button (see Section 4.6, "Relief styles" (p. 10)). The default relief is **RAISED**.
state	Set this option to **DISABLED** to gray out the button and make it unresponsive. Has the value **ACTIVE** when the mouse is over it. Default is **NORMAL**.
takefocus	Normally, keyboard focus does visit buttons (see Section 23, "Focus: routing keyboard input" (p. 77)), 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 22, "Control variables: the values behind the widgets" (p. 75).
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 4.1, "Dimensions" (p. 7).

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 callback, and returns what that function returns. Has no effect if the button is disabled or there is no callback.

6. 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 6.3, “The canvas arc object” (p. 21).
- **.create_bitmap()**: An image as a bitmap. See Section 6.4, “The canvas bitmap object” (p. 22).
- **.create_image()**: A graphic image. See Section 6.5, “The canvas image object” (p. 22).
- **.create_line()**: One or more line segments. See Section 6.6, “The canvas line object” (p. 23).
- **.create_oval()**: An ellipse; use this also for drawing circles, which are a special case of an ellipse. See Section 6.7, “The canvas oval object” (p. 23).
- **.create_polygon()**: A polygon. See Section 6.8, “The canvas polygon object” (p. 24).
- **.create_rectangle()**: A rectangle. See Section 6.9, “The canvas rectangle object” (p. 25).
- **.create_text()**: Text annotation. See Section 6.10, “The canvas text object” (p. 26).
- **.create_window()**: A rectangular window. See Section 6.11, “The canvas window object” (p. 26).

To create a **Canvas** object:

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

The constructor returns the new canvas widget. Supported options include:

bd or **borderwidth**	Border width in 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 4.8, “Cursors” (p. 11).
height	Size of the canvas in the Y dimension. See Section 4.1, “Dimensions” (p. 7).
highlightbackground	Color of the focus highlight when the widget does not have focus. See Section 23, “Focus: routing keyboard input” (p. 77).
highlightcolor	Color shown in the focus highlight.
highlightthickness	Thickness of the focus highlight.

relief	The relief style of the canvas. Default is **FLAT**. See Section 4.6, "Relief styles" (p. 10).
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 23, "Focus: routing keyboard input" (p. 77)) will cycle through this widget with the tab key only if there are keyboard bindings set for it (see Section 24, "Events" (p. 78) 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 4.1, "Dimensions" (p. 7).
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 16, "The **Scrollbar** widget" (p. 49).
xscrollcommand	If the canvas is scrollable, this attribute should be the **.set()** method of the horizontal scrollbar.
yscrollincrement	Works like **xscrollincrement**, but governs vertical movement.
yscrollcommand	If the canvas is scrollable, this attribute should be the **.set()** method of the vertical scrollbar.

6.1. Canvas concepts

Before we discuss the methods on canvases, we need to define some terms:

6.1.1. Canvas and window 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.

6.1.2. The display list; "above" and "below"

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.

6.1.3. Object ID

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.

6.1.4. 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.

6.1.5. 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.
- 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).

6.2. Methods on Canvas objects

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. This allows you to search through all the overlapping objects sequentially.

.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 (*screenx*, 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** are deleted, where those values can be integer indices or the string **"end"** to mean the end of the text.

.delete (*tagOrId*)
Deletes the object or objects selected by ***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.

.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.

.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*, *index*)
Returns the integer index of the given ***index*** in the object specified by ***tagOrId*** (the lowest one that allows text insertion, if ***tagOrId*** specifies multiple objects). The ***index*** argument is an integer or the string **"end"**.

.insert (*tagOrId*, *beforeThis*, *text*)
Inserts the given ***string*** in the object or objects specified by ***tagOrId***, before index ***beforethis*** (which can be an integer or the string **"end"**).

.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 all.
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 all.

.scale (*tagOrId*, *xOrigin*, *yOrigin*, *xScale*, *yScale*)
Scale all objects according to their distance from a point P=(***xOrigin***, ***yOrigin***). 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***.

.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 24, "Events" (p. 78).

.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***.

.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***.

.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 24, "Events" (p. 78).

.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 (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 (SCROLL, *n*, *what*)
This method moves the canvas left or right: the ***what*** argument specifies how much to move and can be either **UNITS** or **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 **UNITS** is given by the value of the canvas's **xscrollincrement** option; see Section 16, "The **Scrollbar** widget" (p. 49).

For movements by **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(MOVETO, *fraction*)**.

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

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

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

6.3. The canvas arc object

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 6.7, "The canvas oval object" (p. 23) for more on the geometry of the ellipse drawn.

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

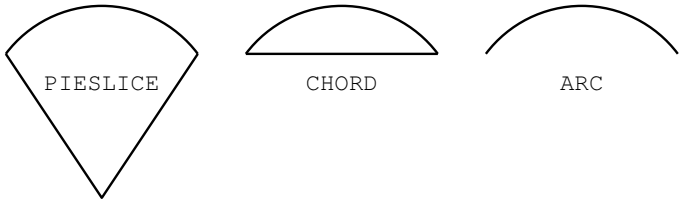
```
id = C.create_arc ( x0, y0, x1, y1, option, ... )
```

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:

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.
outline	The color of the border around the outside of the slice. Default is black.
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.
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=PIESLICE** for this style. To draw only the circular arc at the edge of the slice, use **style=ARC**. To draw the circular arc and the chord (a straight line connecting the endpoints of the arc), use **style=CHORD**.

	
width	Width of the border around the outside of the arc. Default is 1 pixel.

6.4. The canvas bitmap object

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:

anchor	The bitmap is positioned relative to point (***x***, ***y***). The default is anchor=**CENTER**, meaning that the bitmap is centered on the (***x***, ***y***) position. See Section 4.5, "Anchors" (p. 9) for the various **anchor** option values. For example, if you specify **anchor=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 4.7, "Bitmaps" (p. 10).
foreground	The color that will appear where there are 1 values in the bitmap. The default is **foreground="black"**.
tags	The tags to be associated with the object, as a sequence of strings.

6.5. The canvas image object

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:

anchor	The default is **anchor=CENTER**, meaning that the image is centered on the (***x***, ***y***) position. See Section 4.5, "Anchors" (p. 9) for the possible values of this option. For example, if you specify **anchor=S**, the image will be positioned so that point (***x***, ***y***) is located at the center of the bottom (south) edge of the image.
image	The image to be displayed. See Section 4.9, "Images" (p. 12), above, for information about how to create images that can be loaded onto canvases.

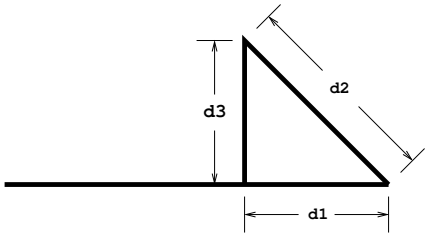
tags	The tags to be associated with the object, as a sequence of strings. See Section 6.1.4, “Canvas tags” (p. 17).

6.6. The canvas line object

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:

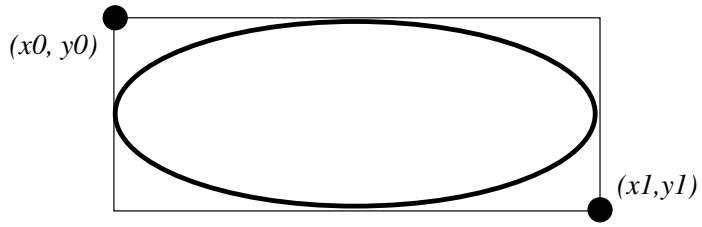
```
id = C.create_line ( x0, y0, x1, y1, ..., xn, yn, option, ... )
```

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

arrow	The default is for the line to have no arrowheads. Use **arrow=FIRST** to get an arrowhead at the (***x0***, ***y0***) end of the line. Use **arrow=LAST** to get an arrowhead at the far end. Use **arrow=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)**. 
fill	The color to use in drawing the line. Default is **fill="black"**.
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**.
stipple	To draw a stippled line, set this option to a bitmap that specifies the stippling pattern, such as **stipple="gray25"**. See Section 4.7, “Bitmaps” (p. 10) for the possible values.
tags	The tags to be associated with the object, as a sequence of strings. See Section 6.1.4, “Canvas tags” (p. 17).
width	The line's width. Default is 1 pixel. See Section 4.1, “Dimensions” (p. 7) for possible values.

6.7. The canvas oval object

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 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, ... )
```

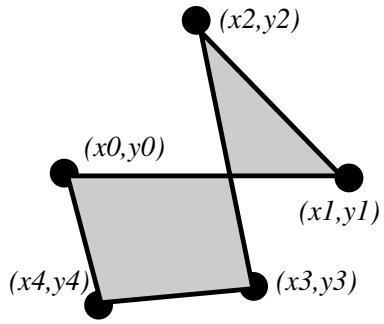
which returns the object ID of the new oval object on canvas ***C***.

Options for ovals:

fill	The default appearance of ellipse 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 4.3, “Colors” (p. 8).
outline	The color of the border around the outside of the ellipse. Default is **outline="black"**.
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 4.7, “Bitmaps” (p. 10).
tags	The tags to be associated with the object, as a sequence of strings. See Section 6.1.4, “Canvas tags” (p. 17).
width	Width of the border around the outside of the ellipse. Default is 1 pixel; see Section 4.1, “Dimensions” (p. 7) 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.

6.8. The canvas polygon object

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:

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 4.3, “Colors” (p. 8).
outline	Color of the outline; defaults to **outline="black"**. Use **outline=""** to make the outline transparent.
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**.
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 4.7, “Bitmaps” (p. 10).
tags	The tags to be associated with the object, as a sequence of strings. See Section 6.1.4, “Canvas tags” (p. 17).
width	Width of the outline; defaults to 1. See Section 4.1, “Dimensions” (p. 7).

6.9. The canvas rectangle object

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

Rectangles are drawn in two parts:

- The outline lies outside the canvas on its top and left sides, but *inside* the canvas on its bottom and right side. The default appearance is a 1-pixel-wide black outline.
- 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:

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 4.3, “Colors” (p. 8).
outline	The color of the border. Default is **outline="black"**.
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 4.7, “Bitmaps” (p. 10).

tags	The tags to be associated with the object, as a sequence of strings. See Section 6.1.4, “Canvas tags” (p. 17).
width	Width of the border. Default is 1 pixel. Use **width=0** to make the border invisible. See Section 4.1, “Dimensions” (p. 7).

6.10. The canvas text object

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:

anchor	The default is **anchor=CENTER**, meaning that the text is centered vertically and horizontally around position (***x***, ***y***). See Section 4.5, “Anchors” (p. 9) for possible values. For example, if you specify **anchor=SW**, the text will be positioned so its lower left corner is at point (***x***, ***y***).
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 4.3, “Colors” (p. 8).
font	If you don't like the default font, set this option to any font value. See Section 4.4, “Type fonts” (p. 8).
justify	For multi-line textual displays, this option controls how the lines are justified: **LEFT** (the default), **CENTER**, or **RIGHT**.
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 4.7, “Bitmaps” (p. 10).
tags	The tags to be associated with the object, as a sequence of strings. See Section 6.1.4, “Canvas tags” (p. 17).
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 4.1, “Dimensions” (p. 7).

6.11. The canvas window object

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:

anchor	The default is **anchor=CENTER**, meaning that the window is centered on the (***x***, ***y***) position. See Section 4.5, "Anchors" (p. 9) for the possible values. For example, if you specify **anchor=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 4.1, "Dimensions" (p. 7) for possible values.
tags	The tags to be associated with the object, as a sequence of strings. See Section 6.1.4, "Canvas tags" (p. 17).
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..

7. 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 22, "Control variables: the values behind the widgets" (p. 75), below.
- You can also use event bindings to react to user actions on the checkbutton; see Section 24, "Events" (p. 78), 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 ***master***:

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

The constructor returns a new **Checkbutton** object. Options include:

activebackground	Background color when the checkbutton is under the cursor. See Section 4.3, "Colors" (p. 8).
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=CENTER**. See Section 4.5, "Anchors" (p. 9) for the allowable values. For ex-

	ample, 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 4.3, “Colors” (p. 8).
bitmap	To display a monochrome image on a button, set this option to a bitmap; see Section 4.7, “Bitmaps” (p. 10).
bd or **borderwidth**	The size of the border around the indicator. Default is 2 pixels. For possible values, see Section 4.1, “Dimensions” (p. 7).
command	A procedure to be called every time the user changes the state of this checkbutton.
cursor	If you set this option to a cursor name (see Section 4.8, “Cursors” (p. 11)), 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 4.4, “Type fonts” (p. 8).
fg or **foreground**	The color used to render the ***text***.
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 23, “Focus: routing keyboard input” (p. 77).
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 4.9, “Images” (p. 12).
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: **CENTER**, **LEFT**, or **RIGHT**.
offvalue	Normally, a checkbutton's associated control variable will be set to 0 when it is cleared (off). You can supply an alternate value for the off state by setting **offvalue** to that value.
onvalue	Normally, a checkbutton's associated control variable will be set to 1 when it is set (on). You can supply an alternate value for the on state by setting **onvalue** to that value.
padx	How much space to leave to the left and right of the checkbutton and text. Default is 1 pixel. For possible values, see Section 4.1, “Dimensions” (p. 7).
pady	How much space to leave above and below the checkbutton and text. Default is 1 pixel.

relief	With the default value, **relief=FLAT**, the checkbutton does not stand out from its background. You may set this option to any of the other styles (see Section 4.6, "Relief styles" (p. 10)), or use **relief=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 4.9, "Images" (p. 12).
state	The default is **state=NORMAL**, but you can use **state=DISABLED** to gray out the control and make it unresponsive. If the cursor is currently over the checkbutton, the state is **ACTIVE**.
takefocus	The default is that the input focus (see Section 23, "Focus: routing keyboard input" (p. 77)) 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 22, "Control variables: the values behind the widgets" (p. 75)) 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 22, "Control variables: the values behind the widgets" (p. 75). 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.

8. 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 17, "The **Text** widget" (p. 53).
- If you want to display one or more lines of text that *cannot* be modified by the user, see Section 10, "The **Label** widget" (p. 34).

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 **END** refers to the position after the existing text.
  - The constant **INSERT** refers to the current position of the insertion cursor.
  - The constant **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 ***master***:

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

This constructor returns the entry widget object. Options include:

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. Default is 2.
cursor	The cursor used when the mouse is within the entry widget; see Section 4.8, "Cursors" (p. 11).
font	The font used for text entered in the widget by the user. See Section 4.4, "Type fonts" (p. 8).
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.

highlightbackground	Color of the focus highlight when the widget does not have focus. See Section 23, “Focus: routing keyboard input” (p. 77).
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 4.3, “Colors” (p. 8).
insertborderwidth	By default, the insertion cursor is a simple rectangle. You can get the cursor with the **RAISED** relief effect (see Section 4.6, “Relief styles” (p. 10)) by setting **insertborderwidth** to the dimension of the 3-d border. If you do, make sure that the **insertwidth** attribute 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 attribute 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 **LEFT** (the default), **CENTER**, or **RIGHT**.
relief	Selects three-dimensional shading effects around the text entry. See Section 4.6, “Relief styles” (p. 10). The default is **relief=SUNKEN**.
selectbackground	The background color to use displaying selected text. See Section 4.3, “Colors” (p. 8).
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 attribute to disable the **Entry** widget so that the user can't type anything into it. Use **state=DISABLED** to disable the widget, **state=NORMAL** to allow user input again. Your program can also find out whether the cursor is currently over the widget by interrogating this attribute; it will have the value **ACTIVE** when the mouse is over it.
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 23, “Focus: routing keyboard input” (p. 77).
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 22, “Control variables: the values behind the widgets” (p. 75). You can retrieve the text using ***v*.get()**, or set it using ***v*.set()**, where ***v*** is the associated control variable.

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 8.1, "Scrolling an **Entry** widget" (p. 33).

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***.

.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 **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, END)**.

.select_to (*index*)
Selects all the text from the **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 8.1, "Scrolling an **Entry** widget" (p. 33).

.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 **UNITS**, to scroll by character widths, or **PAGES**, to scroll by chunks the size of the entry widget. The ***number*** is positive to scroll left to right, negative to scroll right to left. For example, for an entry widget **e**, **e.xview_scroll(-1, PAGES)** would move the text one "page" to the right, and **e.xview_scroll(4, UNITS)** would move the text four characters to the left.

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

    self.entryScroll = Scrollbar ( self, orient=HORIZONTAL,
        command=self.__scrollHandler )
    self.entryScroll.grid(row=1, sticky=E+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 )
```

9. 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 ***master***:

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

The constructor returns the new frame widget. Options:

bg or **background**	The frame's background color. See Section 4.3, “Colors” (p. 8).
bd or **borderwidth**	Width of the frame's border. The default is 0 (no border). For permitted values, see Section 4.1, “Dimensions” (p. 7).
cursor	The cursor used when the mouse is within the frame widget; see Section 4.8, “Cursors” (p. 11).
height	The vertical dimension of the new frame. This will be ignored unless you also call **.grid_propagate(0)** on the frame.
highlightbackground	Color of the focus highlight when the frame does not have focus. See Section 23, “Focus: routing keyboard input” (p. 77).
highlightcolor	Color shown in the focus highlight when the frame has the focus.
highlightthickness	Thickness of the focus highlight.
relief	The default relief for a frame is **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 4.6, “Relief styles” (p. 10).
takefocus	Normally, frame widgets are not visited by input focus (see Section 23, “Focus: routing keyboard input” (p. 77) 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 24, “Events” (p. 78) for more on events and bindings.
width	The horizontal dimension of the new frame. See Section 4.1, “Dimensions” (p. 7). This value be ignored unless you also call **.grid_propagate(0)** on the frame.

10. 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 ***master***:

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

The constructor returns the label widget. Options include:

anchor	This options controls where the text is positioned if the widget has more space than the text needs. The default is **anchor=CENTER**, which centers the text in the available space. For other values, see Section 4.5, “Anchors” (p. 9). For example, if you use **anchor=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 4.3, “Colors” (p. 8).
bitmap	Set this option equal to a bitmap or image object and the label will display that graphic. See Section 4.7, “Bitmaps” (p. 10) and Section 4.9, “Images” (p. 12).
bd or **borderwidth**	Width of the border around the label. Default is 2.

cursor	Cursor that appears when the mouse is over this label. See Section 4.8, “Cursors” (p. 11).
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 4.4, “Type fonts” (p. 8).
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 4.3, “Colors” (p. 8).
height	Height of the label in *lines* (not pixels!). If this option is not set, the label will be sized to fit its contents.
image	To display a static image in the label widget, set this option to an image object. See Section 4.9, “Images” (p. 12).
justify	Specifies how multiple lines of text will be aligned with respect to each other: **LEFT** for flush left, **CENTER** for centered (the default), or **RIGHT** for right-justified.
padx	Extra space added to the left and right of the text within the widget. Default is 1.
pady	Extra space added above and below the text within the widget. Default is 1.
relief	Specifies the appearance of a decorative border around the label. The default is **FLAT**; for other values, see Section 4.6, “Relief styles” (p. 10).
takefocus	Normally, focus does not cycle through **Label** widgets; see Section 23, “Focus: routing keyboard input” (p. 77). 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 22, “Control variables: the values behind the widgets” (p. 75).
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 19, “Universal widget methods” (p. 64)).

11. 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. If you need something more like a text editor, see Section 17, "The **Text** widget" (p. 53).

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

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

This constructor returns the new listbox widget. Options:

bg or **background**	The background color in the listbox.
bd or **borderwidth**	The width of the border around the listbox. Default is 2. For possible values, see Section 4.1, "Dimensions" (p. 7).
cursor	The cursor that appears when the mouse is over the listbox. See Section 4.8, "Cursors" (p. 11).
font	The font used for the text in the listbox. See Section 4.4, "Type fonts" (p. 8).
fg or **foreground**	The color used for the text in the listbox. See Section 4.3, "Colors" (p. 8).
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 23, "Focus: routing keyboard input" (p. 77).
highlightcolor	Color shown in the focus highlight when the widget has the focus.
highlightthickness	Thickness of the focus highlight.
relief	Selects three-dimensional border shading effects. The default is **SUNKEN**. For other values, see Section 4.6, "Relief styles" (p. 10).
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 **RAISED** relief (see Section 4.6, "Relief styles" (p. 10)).
selectforeground	The foreground color to use displaying selected text.
selectmode	Determines how many items can be selected, and how mouse drags affect the selection: • **BROWSE**: Normally, you can only select one line out of a listbox. If you click on an item and then drag to a different line, the selection will follow the mouse. This is the default. • **SINGLE**: You can only select one line, and you can't drag the mouse—wherever you click button 1, that line is selected. • **MULTIPLE**: You can select any number of lines at once. Clicking on any line toggles whether or not it is selected. • **EXTENDED**: You can select any adjacent group of lines at once by clicking on the first line and dragging to the last line.

takefocus	Normally, the focus will tab through listbox widgets. Set this option to 0 to take the widget out of the sequence. See Section 23, "Focus: routing keyboard input" (p. 77).
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 11.1, "Scrolling a **Listbox** widget" (p. 38) 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 11.1, "Scrolling a **Listbox** widget" (p. 38).

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 **END** refers to the last line in the listbox.
- Index **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***.

.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 **END** as the first argument if you want to add new lines to the end of the listbox.

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

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

.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 11.1, "Scrolling a **Listbox** widget" (p. 38).

.xview_moveto (*fraction*)
Scroll the listbox so that the leftmost ***fraction*** of the width of its longest line is outside the left side of the listbox. Fraction is in the range [0,1].

.xview_scroll (*number*, *what*)
Scrolls the listbox horizontally. For the ***what*** argument, use either **UNITS** to scroll by characters, or **PAGES** to scroll by pages, that is, by the width of the listbox. The ***number*** argument tells how many to scroll; 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 11.1, "Scrolling a **Listbox** widget" (p. 38).

.yview_moveto (*fraction*)
Scroll the listbox so that the top ***fraction*** of the width of its longest line is outside the left side of the listbox. Fraction is in the range [0,1].

.yview_scroll (*number*, *what*)
Scrolls the listbox vertically. For the ***what*** argument, use either **UNITS** to scroll by lines, or **PAGES** to scroll by pages, that is, by the height of the listbox. The ***number*** argument tells how many to scroll; negative values move the text downward inside the listbox, and positive values move the text up.

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

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

    self.listbox = Listbox ( self,
```

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

12. 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 the rest of the time.

- 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 13, "The **Menubutton** widget" (p. 42), 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 7, "The **Checkbutton** widget" (p. 27)).
- A group of radiobuttons (see Section 14, "The **Radiobutton** widget" (p. 44)).

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

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

This constructor returns a new menu widget. Options include:

activebackground	The background color that will appear on a choice when it is under the mouse. See Section 4.3, "Colors" (p. 8).
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 4.1, "Dimensions" (p. 7).
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. Default is 1.
cursor	The cursor that appears when the mouse is over the choices, but only when the menu has been torn off. See Section 4.8, "Cursors" (p. 11).
disabledforeground	The color of the text for items whose **state** is **DISABLED**.

font	The default font for textual choices. See Section 4.4, “Type fonts” (p. 8).
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=RAISED**. For other options, see Section 4.6, “Relief styles” (p. 10).
selectcolor	Specifies the color displayed in checkbuttons and radiobuttons when they are selected.
tearoff	Normally, a menu can be torn off, the first position (position 0) in the list of choices is occupied by the tear-off element, and the additional choices are added starting at position 1. If you set **tearoff=0**, the menu will not have a tear-off feature, and choices will be added starting at position 0.
title	Normally, the title of a tear-off menu window will be the same as the text of the menubutton or cascade that lead to this menu. If you want to change the title of that window, set the **title** option to that string.
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.

These methods are available on **Menu** objects. The ones that create choices on the menu have their own particular options; see Section 12.1, “Menu item creation (**coption**) options” (p. 41).

.add_cascade (*coption*, ...)
Add a new cascade element as the next available choice in self. 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 12.1, “Menu item creation (**coption**) options” (p. 41).

.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 14, “The **Radiobutton** widget” (p. 44).

.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(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.

.type (*index*)
Returns the type of the choice specified by ***index***: either **"cascade"**, **"checkbutton"**, **"command"**, **"radiobutton"**, **"separator"**, or **"tearoff"**.

12.1. Menu item creation (coption) options

These are the possible values of the ***coption*** choice options used in the methods above:

activebackground	The background color used for choices when they are under the mouse.
activeforeground	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 4.7, “Bitmaps” (p. 10).
columnbreak	Normally all the choices are displayed in one long column. If you set **columnbreak=1**, this choice will start a new column to the right of the one containing the previous choice.
command	A procedure to be called when this choice is activated.
font	The font used to render the **label** text. See Section 4.4, “Type fonts” (p. 8)

foreground	The foreground color used for choices when they are *not* under the mouse. Note that this *cannot* be abbreviated as **fg**.
image	Display an image for this choice; see Section 4.9, “Images” (p. 12).
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 22, “Control variables: the values behind the widgets” (p. 75).
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 4.3, “Colors” (p. 8).
state	Normally, all choices react to mouse clicks, but you can set **state=DISABLED** to gray it out and make it unresponsive.
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 22, “Control variables: the values behind the widgets” (p. 75)) 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 22, “Control variables: the values behind the widgets” (p. 75).

13. 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 ***master***:

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

The constructor returns the new menubutton widget. Options:

activebackground	The background color when the mouse is over the menubutton. See Section 4.3, “Colors” (p. 8).
activeforeground	The foreground color when the mouse is over the menubutton.
anchor	This options controls where the text is positioned if the widget has more space than the text needs. The default is **anchor=CENTER**, which centers the text. For other options, see Section 4.5, “Anchors” (p. 9). For example,

	if you use **anchor=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 4.7, “Bitmaps” (p. 10).
bd or **borderwidth**	Width of the border around the menubutton. Default is 2 pixels. For possible values, see Section 4.1, “Dimensions” (p. 7).
cursor	The cursor that appears when the mouse is over this menubutton. See Section 4.8, “Cursors” (p. 11).
direction	Normally, the menu will appear below the menubutton. Set **direction=LEFT** to display the menu to the left of the button; use **direction=RIGHT** to display the menu to the right of the button; or use **direction=ABOVE** to place the menu above the button.
disabledforeground	The foreground color shown on this menubutton when it is disabled.
fg or **foreground**	The foreground color when the mouse is not over the menubutton.
height	The height of the menubutton in *lines* of text (not pixels!). The default is to fit the menubutton's size to its contents.
highlightbackground	Color of the focus highlight when the widget does not have focus. See Section 23, “Focus: routing keyboard input” (p. 77).
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 4.9, “Images” (p. 12).
justify	This option controls where the text is located when the text doesn't fill the menubutton: use **justify=LEFT** to left-justify the text (this is the default); use **justify=CENTER** to center it, or **justify=RIGHT** to right-justify.
menu	To associate the menubutton with a set of choices, set this option to the **Menu** object containing those choices. That menu object must have been created by passing the associated menubutton to the constructor as its first argument. 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 **RAISED** appearance. For other 3-d effects, see Section 4.6, “Relief styles” (p. 10).
state	Normally, menubuttons respond to the mouse. Set **state=DISABLED** to gray out the menubutton and make it unresponsive.
text	To display text on the menubutton, set this option to the string containing the desired text. Newlines (**"\n"**) within the string will cause line breaks.
textvariable	You can associate a control variable of class **StringVar** with this menubutton. Setting that control variable will change the displayed text. See Section 22, “Control variables: the values behind the widgets” (p. 75).

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 three checkboxes:

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

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

    self.mayoVar  = IntVar()
    self.ketchVar = 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.

14. 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 22, “Control variables: the values behind the widgets” (p. 75), 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 ***master***:

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

This constructor returns the new radiobutton widget. Options:

activebackground	The background color when the mouse is over the radiobutton. See Section 4.3, "Colors" (p. 8).
activeforeground	The foreground color when the mouse is over the radiobutton.
anchor	If the widget inhabits a space larger than it needs, this option specifies where the radiobutton will sit in that space. The default is **anchor=CENTER**. For other positioning options, see Section 4.5, "Anchors" (p. 9). For example, if you set **anchor=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 4.7, "Bitmaps" (p. 10).
borderwidth	The size of the border around the indicator part itself. Default is 2 pixels. For possible values, see Section 4.1, "Dimensions" (p. 7).
command	A procedure to be called every time the user changes the state of this radiobutton.
cursor	If you set this option to a cursor name (see Section 4.8, "Cursors" (p. 11)), 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 4.4, "Type fonts" (p. 8).
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 23, "Focus: routing keyboard input" (p. 77).
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 4.9, "Images" (p. 12). 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: **CENTER** (the default), **LEFT**, or **RIGHT**.
padx	How much space to leave to the left and right of the radiobutton and text. Default is 1.
pady	How much space to leave above and below the radiobutton and text. Default is 1.
relief	By default, a radiobutton will have **FLAT** relief, so it doesn't stand out from its background. See Section 4.6, "Relief styles" (p. 10) for more 3-d effect options. You can also use **relief=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 4.9, "Images" (p. 12).
state	The default is **state=NORMAL**, but you can set **state=DISABLED** to gray out the control and make it unresponsive. If the cursor is currently over the radiobutton, the state is **ACTIVE**.
takefocus	By default, the input focus (see Section 23, "Focus: routing keyboard input" (p. 77)) 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 22, "Control variables: the values behind the widgets" (p. 75)) 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 22, "Control variables: the values behind the widgets" (p. 75). 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.

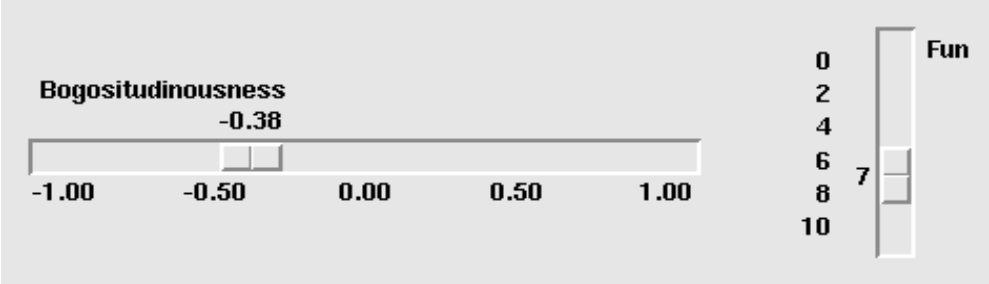
.flash()
Flashes the radiobutton a few times between its active and normal colors, but leaves it the way it started.

.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.

15. The Scale widget

The purpose of a scale widget is to allow the user to set some integer 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 ***master***:

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

The constructor returns the new scale widget. Options:

activebackground	The color of the slider when the mouse is over it. See Section 4.3, “Colors” (p. 8).

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 2 pixels. For acceptable values, see Section 4.1, “Dimensions” (p. 7).
command	A procedure to be called every time the slider is moved. 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 4.8, “Cursors” (p. 11).
digits	The way your program reads the current value shown in a scale widget is through a control variable; see Section 22, “Control variables: the values behind the widgets” (p. 75). The control variable for a scale can be an **IntVar**, a **DoubleVar** (float), or a **StringVar**. If it is a string variable, the **digits** option controls how many digits to use when the numeric scale value is converted to a string.
font	The font used for the label and annotations. See Section 4.4, “Type fonts” (p. 8).
fg or **foreground**	The color of the text used for the label and annotations.
from_	A float or integer value that defines one end of the scale's range. 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. 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 23, “Focus: routing keyboard input” (p. 77).
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 4.1, “Dimensions” (p. 7).
orient	Set **orient=HORIZONTAL** if you want the scale to run along the x dimension, or **orient=VERTICAL** to run parallel to the y-axis. Default is horizontal.
relief	With the default **relief=FLAT**, the scale does not stand out from its background. You may also use **relief=SOLID** to get a solid black frame around the scale, or any of the other relief types described in Section 4.6, “Relief styles” (p. 10).
repeatdelay	This option controls how long button 1 has to be held down in the trough before the slider starts moving in that direction repeatedly. Default is **repeatdelay=300**, and the units are milliseconds.
resolution	Normally, the user will only be able to change the scale in whole units. Set this option to some other value to change the smallest increment of

	the scale's value. For example, if **from_=-1.0** and **to=1.0**, and you set **resolution=0.5**, the scale will have 5 possible values: -1.0, -0.5, 0.0, +0.5, and +1.0. All smaller movements will be ignored.
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 4.1, "Dimensions" (p. 7).
state	Normally, scale widgets respond to mouse events, and when they have the focus, also keyboard events. Set **state=DISABLED** to make the widget unresponsive.
takefocus	Normally, the focus will cycle through scale widgets. Set this option to 0 if you don't want this behavior. See Section 23, "Focus: routing keyboard input" (p. 77).
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 **tick-interval=0.25**, labels will be displayed along the scale at values 0.0, 0.25, 0.50, 0.75, and 1.00. These labels appear below the scale if horizontal, to its left if vertical. Default is 0, which suppresses display of ticks.
to	A float or integer value that defines one end of the scale's range; the other end is defined by the **from_** option, discussed above. The **to** value can be either greater than or less than the **from_** value. For vertical scales, the **to** value defines the bottom of the scale; for horizontal scales, the right end.
troughcolor	The color of the trough.
variable	The control variable for this scale, if any; see Section 22, "Control variables: the values behind the widgets" (p. 75). Control variables may be from class **IntVar**, **DoubleVar** (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=HORIZONTAL**. Default is 15 pixels.

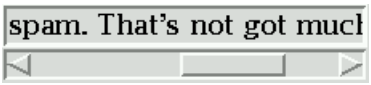
Scale objects have these methods:

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

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

16. 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:



- 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 *trough* is the sunken-looking area visible behind the arrowheads and slider.
- 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'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.

To create a new scrollbar widget as the child of a root window or frame ***master***:

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

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

activebackground	The color of the slider and arrowheads when the mouse is over them. See Section 4.3, “Colors” (p. 8).
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 2-pixel border around the arrowheads and slider.
command	A procedure to be called whenever the scrollbar is moved. For a discussion of the calling sequence, see Section 16.1, “The scrollbar **command** callback” (p. 51).
cursor	The cursor that appears when the mouse is over the scrollbar. See Section 4.8, “Cursors” (p. 11).
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 23, "Focus: routing keyboard input" (p. 77).
highlightcolor	The color of the focus highlight when the scrollbar has the focus.
highlightthickness	The thickness of the focus highlight. Default is **1**. Set to **0** to suppress display of the focus highlight.
jump	This option controls what happens when a user drags the slider. Normally (**jump=0**), every small drag of the slider causes the **command** callback to be called. If you set this option to **1**, the callback isn't called until the user releases the mouse button.
orient	Set **orient=HORIZONTAL** for a horizontal scrollbar, **orient=VERTICAL** for a vertical one.
repeatdelay	This option controls how long button 1 has to be held down in the trough before the slider starts moving in that direction repeatedly. Default is **repeatdelay=300**, and the units are milliseconds.
repeatinterval	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 23, "Focus: routing keyboard input" (p. 77). 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 4.1, "Dimensions" (p. 7).

Methods on scrollbar objects include:

.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.

16.1. 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("scroll", -1, "units")
```

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

```
command("scroll", 1, "units")
```

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

```
command("scroll", -1, "pages")
```

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

```
command("scroll", 1, "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("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 8.1, "Scrolling an **Entry** widget" (p. 33).

16.2. Connecting scrollbars to other widgets

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

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

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

17. 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 17.3, "Images in text widgets" (p. 56).
- You can even embed in it a "window" containing any Tkinter widget—even a frame widget containing other widgets. A window is also treated as a single character. See Section 17.4, "Windows in text widgets" (p. 56).
- An *index* is a way of describing a specific position between two characters of a text widget. See Section 17.1, "Indices in text widgets" (p. 55).
- 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 attributes. See Section 17.5, "Tags in text widgets" (p. 56).
- A text widget may contain invisible *mark* objects between character positions. See Section 17.2, "Marks in text widgets" (p. 56).

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

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

The constructor returns the new text widget. Options include:

bg or **background**	The default background color of the text widget. See Section 4.3, "Colors" (p. 8).
bd or **borderwidth**	The width of the border around the text widget. Default is 2 pixels.
cursor	The cursor that will appear when the mouse is over the text widget. See Section 4.8, "Cursors" (p. 11).
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 4.4, "Type fonts" (p. 8).
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 23, "Focus: routing keyboard input" (p. 77).
highlightcolor	The color of the focus highlight when the text widget has the focus.
highlightthickness	The thickness of the focus highlight. Default is **1**. Set **highlightthickness=0** to suppress display of the focus highlight.
insertbackground	The color of the insertion cursor. Default is black.
insertborderwidth	Size of the 3-D border around the insertion cursor. Default is **0**.

insertofftime	The number of milliseconds the insertion cursor is off during its blink cycle. Set this option to zero to suppress blinking. Default is 300.
insertontime	The number of milliseconds the insertion cursor is on during its blink cycle. Default is 600.
insertwidth	Width of the insertion cursor (its height is determined by the tallest item in its line). Default is 2 pixels.
padx	The size of the internal padding added to the left and right of the text area. Default is one pixel. For possible values, see Section 4.1, "Dimensions" (p. 7).
pady	The size of the internal padding added above and below the text area. Default is one pixel.
relief	The 3-D appearance of the text widget. Default is **relief=SUNKEN**; for other values, see Section 4.6, "Relief styles" (p. 10).
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=NORMAL** to get this behavior. If you set **state=DISABLED**, the text widget will not respond, and you won't be able to modify its contents programmatically either.
tabs	This option controls how tab characters position text. See Section 17.6, "Setting tabs in a **Text** widget" (p. 57).
takefocus	Normally, focus will visit a text widget (see Section 23, "Focus: routing keyboard input" (p. 77)). Set **takefocus=0** if you do not want focus in the widget.
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=CHAR**, any line that gets too long will be broken at any character. • Set **wrap=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=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.

17.1. Indices in text widgets

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.

INSERT
The position of the insertion cursor in the text widget.

CURRENT
The position of the character closest to the mouse pointer.

END
The position after the last character of the text.

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 **TclError** exception will be raised.

SEL_LAST
The position after the end of the selection, if any. As with **SEL_FIRST**, you'll get a **TclError** exception if you use such an index and there is no selection.

"*markname*"
You can use a mark as an index; just pass its name where an index is expected. See Section 17.2, “Marks in text widgets” (p. 56).

"*tag*.first"
The position before the first character of the region tagged with name ***tag***; see Section 17.5, “Tags in text widgets” (p. 56).

"*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 17.3, “Images in text widgets” (p. 56) and Section 17.4, “Windows in text widgets” (p. 56).

17.2. Marks in text widgets

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. **`INSERT`** is the current position of the insertion cursor, and **`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 **`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 **`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 17.7, “Methods on **`Text`** widgets” (p. 57), below, to see how to use marks.

17.3. Images in text widgets

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 4.9, “Images” (p. 12) andSection 4.7, “Bitmaps” (p. 10).

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.

17.4. Windows in text widgets

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 17.7, “Methods on **`Text`** widgets” (p. 57).

17.5. Tags in text widgets

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 17.7, "Methods on **`Text`** widgets" (p. 57), below, for information on this and related methods.

17.6. 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 **`LEFT`**, **`RIGHT`**, **`CENTER`**, or **`NUMERIC`** in the list after a distance, and that will change the positioning of the text after each tab.
  - A **`LEFT`** tab stop has the default behavior.
  - A **`RIGHT`** tab stop will position the text so its right edge is on the stop.
  - A **`CENTER`** tab will center the following text on the tab stop.
  - A **`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", RIGHT, "1.2i", CENTER, "2i", 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.

17.7. Methods on `Text` widgets

These methods are available on all text widgets:

`.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.

.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.

.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 sequence of tag strings. Any tags that apply to existing characters around the insertion point are ignored.

.mark_gravity (*mark*, *gravity*=None)
Changes or queries the gravity of an existing mark; see Section 17.2, "Marks in text widgets" (p. 56), above, for an explanation of gravity.

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

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

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

.mark_unset (*mark*)
Removes the named mark.

.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 1 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.
regexp	Set this option to 1 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 24, "Events" (p. 78), 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 4.7, "Bitmaps" (p. 10)). 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 4.4, “Type fonts” (p. 8).
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: **LEFT** (the default), **CENTER**, or **RIGHT**.
lmargin1	How much to indent the first line of a chunk of text that has this tag. The default is **0**. See Section 4.1, “Dimensions” (p. 7)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 4.1, “Dimensions” (p. 7).
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=FLAT**; for other possible values see Section 4.6, “Relief styles” (p. 10).
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 17.6, “Setting tabs in a **Text** widget” (p. 57).
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 17.5, “Tags in text widgets” (p. 56), 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 17.5, "Tags in text widgets" (p. 56), 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 **[*s0*, *e0*, *s1*, *e1*, …]**, where each ***si*** is the index just before the first character of the range and ***ei*** 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=CENTER** (the default), which centers the widget vertically within the line; **align=TOP**, which places the top of the image at the top of the line; **align=BOTTOM**, which places the bottom of the image at the bottom of the line; and **align=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 (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 (SCROLL, *n*, *what*)
The second call moves the text left or right: the ***what*** argument specifies how much to move and can be either **UNITS** or **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(MOVETO, *fraction*)**.

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

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

.yview(SCROLL, *n*, *what*)
The vertical scrolling equivalent of **.xview(SCROLL,…)**. When scrolling vertically by **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()**.

18. 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 = Toplevel ( option, ... )
```

Options include:

bg or **background**	The background color of the window. See Section 4.3, “Colors” (p. 8).
bd or **borderwidth**	Border width in pixels; default is **0**. For possible values, see Section 4.1, “Dimensions” (p. 7). 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 popups 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 4.8, “Cursors” (p. 11).
height	Window height; see Section 4.1, “Dimensions” (p. 7).
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 4.6, “Relief styles” (p. 10).
width	The desired width of the window; see Section 4.1, “Dimensions” (p. 7).

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 4.10, "Geometry strings" (p. 12). 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.

.resizable (*width*=None, *height*=None)
If ***width*** 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.

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

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

19. Universal widget methods

The methods are defined below on all widgets. In the descriptions, ***w*** can be any widget—a frame, a top-level window, whatever.

***w*.after (*ms*, *func*=None, **args*)**
Requests Tkinter to call function ***func*** with arguments ***args*** after a delay of at least ***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.

***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.

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 24, "Events" (p. 78) 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 24, "Events" (p. 78) 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).

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 24, "Events" (p. 78) 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 24, "Events" (p. 78) 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 23, "Focus: routing keyboard input" (p. 77) 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()
Get the name of the widget that has focus in this application, if any—otherwise return **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 24, "Events" (p. 78) 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 24, "Events" (p. 78) 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.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.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 4.11, "Window names" (p. 13).

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:

widgetDefault	Level 20, for global default properties of widgets.
startupFile	Level 40, for default properties of specific applications.
userDefault	Level 60, for options that come from user files such as their `.Xdefaults` file.
interactive	Level 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 20, "Standardizing appearance" (p. 71) 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 20, “Standardizing appearance” (p. 71) 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 **TclError**.

Refer to Section 20, “Standardizing appearance” (p. 71) for an introduction to the options database and the format of option files.

***w*.quit()**
This method exits the main loop. See **.mainloop()**, above, for a discussion of main loops.

***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 **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 **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 23, “Focus: routing keyboard input” (p. 77). 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*.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 24, “Events” (p. 78), 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.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_colormapfull()
Returns true if **w**'s window's color map is full, that is, if the last attempt to add a color to it failed and no colors have been removed since then.

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 4.1, "Dimensions" (p. 7)), this method returns that distance in pixels on **w**'s display, as a floating-point number.

w.winfo_geometry()
Returns the geometry string describing the size and on-screen location of **w**. See Section 4.10, "Geometry strings" (p. 12).

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.

***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, such as **"grid"**.

***w*.winfo_name()**
This method returns ***w***'s name relative to its parent. See Section 4.11, "Window names" (p. 13). 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 4.11, "Window names" (p. 13) 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 4.11, "Window names" (p. 13) 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_pointery()**
Returns the same value as the ***y*** 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_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 4.3, "Colors" (p. 8).

***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_screenwidth()**
Returns the width 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_toplevel()**
Returns the top-level window containing ***w***. That window supports all the methods on **Toplevel** widgets; see Section 18, "**Toplevel**: Top-level window methods" (p. 63).

***w*.winfo_width()**
Returns the current width of ***w*** in pixels. See the remarks on geometry updating under **.winfo_geometry()**, above.

***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.

20. 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 19, "Universal widget methods" (p. 64), 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 19, “Universal widget methods” (p. 64).

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 20.1, “How to name a widget class” (p. 72) 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 4.11, “Window names” (p. 13)). However, as with widget classes, you can assign a name to any widget. See the section Section 20.2, “How to name a widget instance” (p. 73) 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 4.11, “Window names” (p. 13) 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 20.3, “Resource specification lines” (p. 73), we'll discuss how the options database really works.

20.1. 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_`** attribute to the parent constructor in your new class's constructor. Here is a fragment of the code that defines the new class:

```
class Jukebox(Frame):
    def __init__(self, master):
        "Constructor for the Jukebox class"
        Frame.__init__ ( self, master, class_="Jukebox" )
        self.__createWidgets()
        ...
```

20.2. 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 = Button ( self, name="panicButton", text="Panic", ...)
```

20.3. 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** attribute 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.

20.4. 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**.

21. 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 24, "Events" (p. 78).
- 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.

22. 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 = DoubleVar()   # Holds a float; default value 0.0
v = IntVar()      # Holds an integer; default value 0
v = 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 19, "Universal widget methods" (p. 64) 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 = StringVar()
self.spamCB  = 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 floating 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.

23. 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 19, "Universal widget methods" (p. 64), 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 24, "Events" (p. 78) for the details.

24. 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.

24.1. 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 19, "Universal widget methods" (p. 64)).

   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 24.6, "Writing your handler" (p. 84), 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 19, "Universal widget methods" (p. 64), 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 19, "Universal widget methods" (p. 64)).

   Here's how you might bind the *PrintScrn* key, whose "key name" is **"Print"**:

   ```
   self.bind_all ( "<Key-Print>", self.__printScreen )
   ```

24.2. 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 24.3, "Event types" (p. 80).

- 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 24.4, “Event modifiers” (p. 81)
- 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 24.5, “Key names” (p. 81) 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.

24.3. 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:

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.
Button	The user pressed one of the mouse buttons. The **detail** part specifies which button.
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 it off the widget to avoid setting off the event.
Configure	The user changed the size of a widget, for example by dragging a corner or side of the window.
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).
Destroy	A widget is being destroyed.
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"**.)
Expose	This event occurs whenever at least some part of your application or widget becomes visible after having been covered up by another window.
FocusIn	A widget got the input focus (see Section 23, “Focus: routing keyboard input” (p. 77) 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).

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.
KeyPress	The user pressed a key on the keyboard. The **detail** part specifies which key. This keyword may be abbreviated **Key**.
KeyRelease	The user let up on a key.
Leave	The user moved the mouse pointer out of a widget.
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.
Motion	The user moved the mouse pointer entirely within a widget.
Unmap	A widget is being unmapped and is no longer visible. This happens, for example, when you use the widget's **.grid_remove()** method.
Visibility	Happens when at least some part of the application window becomes visible on the screen.

24.4. 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>"**).

24.5. 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 24.6, “Writing your handler” (p. 84), 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.

.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

.keysym	.keycode	.keysym_num	Key
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
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 24, "Events" (p. 78)
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**	↑

24.6. Writing your handler

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.)
.height	If the event was a **Configure**, this attribute is set to the widget's new height in pixels.
.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 24.5, “Key names” (p. 81) 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 24.5, “Key names” (p. 81).
.num	If the event was related to a mouse button, this attribute is set to the button number (1, 2, or 3).
.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.
.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.
.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.

Here's an example of an event handler. Under Section 24.1, “Levels of binding” (p. 78), 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**.

24.7. 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 = 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.

24.8. 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 19, "Universal widget methods" (p. 64) for more information about creating and managing virtual events.