

# Tkinter reference: A GUI for Python



---

## Contents

---

Part I: The face of your application	3
1. A minimal application	3
2. Layout management	3
2.1 The .grid() method	4
2.2 Other grid management methods	5
3. Standard attributes	6
3.1 Dimensions	6
3.2 Coordinate system	6
3.3 Colors	7
3.4 Fonts	7
3.5 Bitmaps	8
3.6 Cursors	9
3.7 Images	9
4. The Button widget	10
5. The Canvas widget	11
5.1 The canvas arc object	17
5.2 The canvas bitmap object	18
5.3 The canvas image object	18
5.4 The canvas line object	19
5.5 The canvas oval object	19
5.6 The canvas polygon object	20
5.7 The canvas rectangle object	21
5.8 The canvas text object	22
5.9 The canvas window object	22
6. The Checkbutton widget	23
7. The Entry widget	26
7.1 Scrolling an Entry widget	28
8. The Frame widget	28
9. The Label widget	29
10. The Listbox widget	31
10.1 Scrolling a Listbox widget	34
11. The Menu widget	34
12. The Menubutton widget	38
13. The Radiobutton widget	40
14. The Scale widget	43
15. The Scrollbar widget	46
15.1 The scrollbar command callback	48
15.2 Connecting scrollbars to other widgets	49
16. The Text widget	49
16.1 Indices in text widgets	50

16.2 Marks in text widgets . . . . .	50
16.3 Images in text widgets . . . . .	51
16.4 Windows in text widgets . . . . .	51
16.5 Tags in text widgets . . . . .	51
16.6 Methods on text widgets . . . . .	52
17. Toplevel: Top-level window methods . . . . .	60
18. Universal widget methods . . . . .	61
19. Standardizing appearance and the option database . . . . .	69
19.1 How to name a widget class . . . . .	70
19.2 How to name a widget instance . . . . .	70
19.3 Resource specification lines . . . . .	70
19.4 Rules for resource matching . . . . .	72
Part II: Connecting the interface to the rest of your program . . . . .	73
20. Control variables: the values behind the widgets . . . . .	73
21. Focus: routing keyboard input . . . . .	75
22. Events: responding to stimuli . . . . .	76
22.1 Levels of binding . . . . .	77
22.2 Event sequences . . . . .	78
22.3 Event types . . . . .	78
22.4 Event modifiers . . . . .	79
22.5 Key names . . . . .	80
22.6 Writing an event handler . . . . .	82
22.7 The extra arguments trick . . . . .	83
22.8 Virtual events . . . . .	84

---

Tkinter is a GUI widget set for Python. This document contains only the commoner features. For complete documentation, see:

<http://www.pythonware.com/library.htm>

or the book *Python and Tkinter Programming* by John Grayson (Manning, 2000, ISBN 1-884777-81-3).

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

Refer to the author's companion publication, *Python language quick reference*, or to Web site <http://www.python.org/>, for general information about the Python language.

## Part I: The face of your application

---

We'll start by looking at the visible part of Tkinter: creating the widgets and arranging them on the screen. In Part II, below, we will talk about how to connect the face—the “front panel”—of the application to the logic behind it.

### 1. A minimal application

---

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

```
#!/usr/local/bin/python
from Tkinter import *      # Interface to Tk widgets

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

    def createWidgets(self):
        self.quitButton = Button ( self, text="Quit",
                                   command=self.quit )
        self.quitButton.grid()

app = Application()         # Instantiate the application class
app.master.title("Sample application")
app.mainloop()             # Wait for events
```

### 2. Layout management

---

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

Although there are three different “geometry managers” in Tkinter, the author strongly prefers the Grid geometry manager for pretty much everything. This manager treats every window or frame as a table—a gridwork of rows and columns.

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

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

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

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

## 2.1 The `.grid()` method

```
w.grid ( *options )
```

This method registers a widget *w* with the Grid geometry manager—if you don't do this, the widget will exist but it will not be visible on the screen.

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

<code>column</code>	The column number where you want the widget gridded, counting from zero. The default value is zero.
<code>columnspan</code>	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 <code>columnspan</code> option to the number of cells. For example, <code>w.grid(row=0, column=2, columnspan=3)</code> would place widget <i>w</i> in a cell that spans columns 2, 3, and 4 of row 0.
<code>ipadx</code>	Internal <i>x</i> padding. This dimension is added inside the widget inside its left and right sides.
<code>ipady</code>	Internal <i>y</i> padding. This dimension is added inside the widget inside its top and bottom borders.
<code>padx</code>	External <i>x</i> padding. This dimension is added to the left and right outside the widget.
<code>pady</code>	External <i>y</i> padding. This dimension is added above and below the widget.
<code>row</code>	The row number into which you want to insert the widget, counting from 0. The default is the next higher-numbered unoccupied row.
<code>rowspan</code>	Normally a widget occupies only one cell in the grid. You can grab multiple adjacent cells of a column, however, by setting the <code>rowspan</code> option to the number of cells to grab. This option can be used in combination with the <code>columnspan</code> option to grab a block of cells. For example, <code>w.grid(row=3, column=2, rowspan=4, columnspan=5)</code> would place widget <i>w</i> in an area formed by merging 20 cells, with row numbers 3–6 and column numbers 2–6.
<code>sticky</code>	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 for a discussion.

- 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 setting `sticky` to NE (top right), SE (bottom right), SW (bottom left), or NW (top left).
- You can position the widget against one side of the cell by setting `sticky` to N (top center), E (right center), S (bottom center), or W (left center).
- Set `sticky` to N+S to stretch the widget vertically but leave it centered horizontally.
- Set `sticky` to E+W to stretch it horizontally but leave it centered vertically.
- Set `sticky=N+E+S+W` to stretch the widget both horizontally and vertically to fill the cell.

## 2.2 Other grid management methods

`w.grid_forget()`

This method makes widget `w` disappear from the screen. It still exists, it just isn't visible. You can `.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. 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 *Universal widget methods*, below, for more on these methods.

#### 3.1 Dimensions

Various lengths, widths, and other dimensions of widgets can be described in various 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 ( $\frac{1}{72.27}$ " )

#### 3.2 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 *Dimensions*, above.

### 3.3 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
#rrrrggggbbbb	Sixteen 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 strings "white", "black", "red", "green", "blue", "cyan", "yellow", and "magenta" will also be available. Others depending on your local installation.

### 3.4 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")	16-point Helvetica regular
("Times", "24", "bold italic")	24-pt 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 ( *options )
```

where the options include:

family	The font family as a string.
size	The font height as an integer in points. To get a font <i>n</i> pixels high, use <i>-n</i> .
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, this font

```
"*-lucidatypewriter-medium-r-*--*-140-*--*-*-*-"
```

is the author's favorite fixed-width font for use in X windows. 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 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 ( **options )
```

Use this method to change one or more options on a font. For example, if you have a Font object called `titleFont`, if can 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 ( *options )
```

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.

### 3.5 Bitmaps

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



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



### 3.6 Cursors

There are quite a number of different mouse cursors available. These are their names:

X_cursor	gobbler	sb_h_double_arrow
arrow	gumby	sb_left_arrow
based_arrow_down	hand1	sb_right_arrow
based_arrow_up	hand2	sb_up_arrow
boat	heart	sb_v_double_arrow
bogosity	icon	shuttle
bottom_left_corner	iron_cross	sizing
bottom_right_corner	left_ptr	spider
bottom_side	left_side	spraycan
bottom_tee	left_tee	star
box_spiral	leftbutton	target
center_ptr	ll_angle	tcross
circle	lr_angle	top_left_arrow
clock	man	top_left_corner
coffee_mug	middlebutton	top_right_corner
cross	mouse	top_side
cross_reverse	pencil	top_tee
crosshair	pirate	trek
diamond_cross	plus	ul_angle
dot	question_arrow	umbrella
dotbox	right_ptr	ur_angle
double_arrow	right_side	watch
draft_large	right_tee	xterm
draft_small	rightbutton	
draped_box	rtl_logo	
exchange	sailboat	
fleur	sb_down_arrow	

### 3.7 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, *Python Imaging Library (PIL) quick reference*, for PIL documentation. Objects of the ImageTk class can be used in Tkinter applications.

## 4. 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	Background color when the button is under the cursor.
anchor	Where the text is positioned on the button: one of CENTER (the default), N (centered along the top edge), NE (top right corner), E, SE, S, SW, W, or NW.
bd	Border width in pixels. Also borderwidth.
bg	Normal background color. Also background.
bitmap	Name of one of the standard bitmaps to display on the button (instead of text).
command	Function or method called when the button is clicked.
cursor	Cursor used over the button.
default	NORMAL is the default; use DISABLED if the button is to be initially disabled.
disabledforeground	Foreground color used when the button is disabled.
fg	Normal foreground (text) color. Also foreground.
font	Text font.
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	Color shown in the focus highlight.
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, CENTER (the default), or RIGHT.
padx	Additional padding left and right of the text.
pady	Additional padding above and below the text.
relief	Default is RAISED, changing to SUNKEN when pressed. May also be GROOVE, RIDGE, or FLAT.
state	Default is NORMAL. Set to DISABLED to gray out the button and make it unresponsive. Has the value ACTIVE when the mouse is over it.
takefocus	Normally, keyboard focus does visit buttons (see the section on <i>Focus</i> , below, for an overview of keyboard focus), and a SPACE character acts as the same as a mouse click, “pushing” the button. You can set the takefocus option to zero to disable focus from visiting the button.
text	Text displayed on the button. Use internal newlines to display multiple text lines.

<code>textvariable</code>	An instance of <code>StringVar()</code> that is associated with the text on this button. If the variable is changed, the new value will be displayed on the button.
<code>underline</code>	Default is <code>-1</code> , meaning that no character of the text is underlined. If nonnegative, the corresponding text character will be underlined.
<code>width</code>	Width of the button in letters (if displaying text) or pixels (if displaying an image).
<code>wraplength</code>	If this value is set to a positive number, the text lines will be wrapped to fit within this length.

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 not callback.

## 5. The Canvas *widget*

---

Creates a canvas in the given master window. See the following sections for the objects that can be created on canvases:

<code>.create_arc()</code>	A slice out of an ellipse.
<code>.create_bitmap()</code>	An image as a bitmap.
<code>.create_image()</code>	A general image.
<code>.create_line()</code>	A line.
<code>.create_oval()</code>	An ellipse.
<code>.create_polygon()</code>	A polygon.
<code>.create_rectangle()</code>	A rectangle.
<code>.create_text()</code>	Text annotation.
<code>.create_window()</code>	A rectangular window.

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.

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

The constructor returns the new canvas widget. Supported options:

<code>bd</code>	Border width in pixels. Also <code>borderwidth</code> .
<code>bg</code>	Background color of the canvas. Also <code>background</code> .
<code>closeenough</code>	A float that specifies how close the mouse must be to an item to be considered inside it. Default is <code>1.0</code> .
<code>confine</code>	If true (the default), the canvas cannot be scrolled outside of the <code>scrollregion</code> .

<code>cursor</code>	Cursor used in the canvas.
<code>height</code>	Size of the canvas in the Y dimension.
<code>highlightbackground</code>	Color of the focus highlight when the widget does not have focus.
<code>highlightcolor</code>	Color shown in the focus highlight.
<code>highlightthickness</code>	Thickness of the focus highlight.
<code>relief</code>	Default is RAISED, changing to SUNKEN when pressed. May also be GROOVE, RIDGE, or FLAT.
<code>scrollregion</code>	A tuple ( <i>w</i> , <i>n</i> , <i>e</i> , <i>s</i> ) that defines over how large an area the canvas can be scrolled, where <i>w</i> is the left side, <i>n</i> the top, <i>e</i> the right side, and <i>s</i> the bottom.
<code>selectbackground</code>	The background color to use displaying selected items.
<code>selectborderwidth</code>	The width of the border to use around selected items.
<code>selectforeground</code>	The foreground color to use displaying selected items.
<code>takefocus</code>	Normally, focus will cycle through this widget with the tab key only if there are keyboard bindings set for it (see <i>Events</i> , below, 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.
<code>width</code>	Size of the canvas in the X dimension.
<code>xscrollincrement</code>	Normally, canvases can be scrolled horizontally to any position. You can get this behavior by setting <code>xscrollincrement</code> to zero. If you set it some distance, the canvas can be positioned only on multiples of that distance, and the value will be used for scrolling by "units," such as when the user clicks on the arrows at the ends of a scrollbar.
<code>xscrollcommand</code>	If the canvas is scrollable, this attribute should be the <code>.set</code> method of the horizontal scrollbar.
<code>yscrollincrement</code>	Works like <code>xscrollincrement</code> , but governs vertical movement.
<code>yscrollcommand</code>	If the canvas is scrollable, this attribute should be the <code>.set</code> method of the vertical scrollbar.

Before we discuss the methods, some definitions:

- The *display list* is an ordering of the objects on the canvas from back (background) to front (foreground). 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.
- The *object ID* of an object on the canvas is the value returned by the constructor for that object.
- A *tag* is a string that you can associate with any object or any number of objects on the canvas.

- A `tagOrId` argument specifies one or more objects on the canvas. If a `tag` (string), it specifies all the objects with that tag. If an `Id` (integer), it specifies the object that has that ID number.

Methods on Canvas objects:

```
.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 a tag string.

```
.addtag_all ( newTag )
```

Attaches the given tag `newTag` to all the objects on the canvas.

```
.addtag_below ( newTag, tagOrId )
```

Attaches a new tag to the object just below the one specified by `tagOrId` in the display list. The `newTag` argument is a tag string.

```
.addtag_closest ( newTag, x, y, halo=None, start=None )
```

Adds a tag to the object closest to screen coordinate  $(x, y)$ . If there are two or more objects at the same distance, the one higher in the display list is selected.

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

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

```
.addtag_enclosed ( newTag, x1, y1, x2, y2 )
```

Tag all objects that occur completely within the rectangle whose top left corner is  $(x_1, y_1)$  and bottom right corner is  $(x_2, y_2)$ .

```
.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.

```
.cansay ( 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 ( tag0rId, x0, y0, x1, y1, ..., xn, yn )
```

If you pass only the `tag0rId` 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 ( tag0rId, 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 ( tag0rId )
```

Deletes the specified object or objects.

```
.dtag ( tag0rId, tagToDelete )
```

Removes the tag specified by `tagToDelete` from the object or objects specified by `tag0rId`.

```
.find_above ( tag0rId )
```

Returns the ID number of the object just above the specified object. If multiple objects match `tag0rId`, uses the highest one.

```
.find_all()
```

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

```
.find_below ( tag0rId )
```

Returns the ID number of the object just below the specified object. If multiple objects match `tag0rId`, uses the lowest one.

```
.find_closest ( x, y, halo=None, start=None )
```

Returns the ID number of the object closest to point  $(x, y)$ .

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 returns the highest qualifying object that is below `start` in the display list.

```
.find_enclosed ( x1, y1, x2, y2 )
```

Returns a list of the ID numbers 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 ID numbers of all objects that share at least one point with the given rectangle.

```
.find_withtag ( tag0rId )
```

Returns a list of the ID numbers of the object or objects specified by `tag0rId`.

`.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, string )`

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, options=None )`

If the `options` argument is omitted, 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 ( options )`

Generates an Encapsulated PostScript representation of the canvas's current contents. The options include:

<code>colormode</code>	Use "color" for color output, "gray" for grayscale, or "mono" for black and white.
<code>file</code>	If supplied, names a file where the PostScript will be written. If this option is not given, the PostScript is returned as a string.
<code>height</code>	How much of the Y size of the canvas to print. Default is all.

<code>rotate</code>	If false, the page will be rendered in portrait orientation; if true, in landscape.
<code>x</code>	Leftmost canvas coordinate of the area to print.
<code>y</code>	Topmost canvas coordinate of the area to print.
<code>width</code>	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 meaning 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.

`.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 specified 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 last or only object specified 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`.

`.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 )`  
`.xview ( SCROLL, n, what )`

This method scrolls the canvas relative to its image, and is intended for binding to the command option of a related scrollbar.

This method can be called in two different ways. The first call positions the canvas  $x$ -coordinate at a value given by `offset`, where 0.0 moves the canvas to its leftmost position and 1.0 to its rightmost position. The second call 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 (q.v.). Moves by page units use 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 )  
.yview ( SCROLL, n, what )  
.yview_moveto ( fraction )  
.yview_scroll ( n, what )
```

These are the same as the `.xview()` and related methods, only for vertical scrolling.

## 5.1 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 *The canvas oval object*, below, for more on the geometry of the ellipse drawn.

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

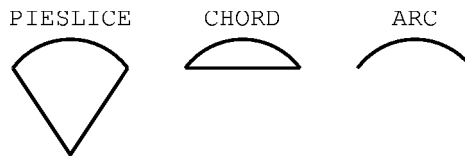
```
id = C.create_arc ( x0, y0, x1, y1, *options )
```

The constructor returns the integer ID number 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 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	The default appearance of an arc is that the interior is transparent, and a value of "" 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 this option to "".
start	Starting angle for the slice, in degrees, measured from the +x direction. If omitted, you get the entire ellipse (or circle).
stipple	A bitmap indicating how the interior fill of the arc will be stippled. Default is "" (solid). Typical value: "gray25". Has no effect unless fill has been set to some color.
style	The default is to draw the whole arc; use value PIESLICE for this style. Set it to ARC to draw only the circular arc at the edge of the slice. Set it to CHORD to draw the circular arc and the chord (a straight line connecting the endpoints of the arc).



width

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

## 5.2 The canvas bitmap object

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

The bitmap is positioned relative to point (x,y). See the anchor option, below, for positioning options.

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.

Options include:

anchor	The default is CENTER, meaning that the bitmap is centered on the (x,y) position. Other possible values are N, NE, E, SE, S, SW, W, and NW. 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 "", meaning transparent.
bitmap	The bitmap to be displayed.
foreground	The color that will appear where there are 1 values in the bitmap. The default is "black".
tags	The tags to be associated with the object, as a sequence of strings.

## 5.3 The canvas image object

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

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

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

The image is positioned relative to point (x,y). See the anchor option, below, for positioning options. Options include:

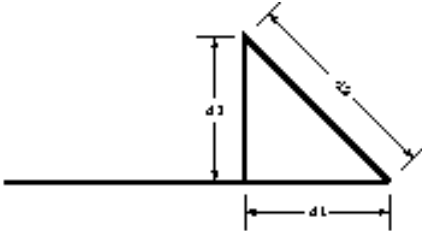
anchor	The default is CENTER, meaning that the image is centered on the (x,y) position. Other possible values are N, NE, E, SE, S, SW, W, and NW. For example, if you specify anchor=NE, the image will be positioned so that point (x,y) is located at the northeast (top right) corner of the bitmap.
image	The image to be displayed. See <i>Images</i> , 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.

## 5.4 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:

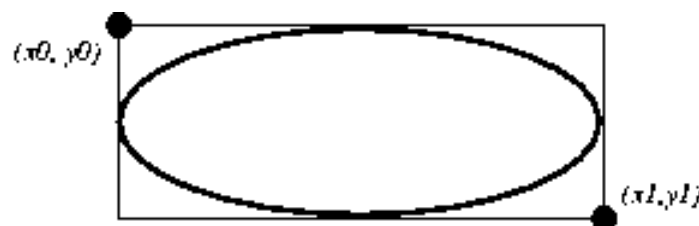
```
C.create_line ( x0, y0, x1, y1, ..., xn, yn, *options )
```

The line goes through the series of points  $(x_0, y_0), (x_1, y_1), \dots, (x_n, y_n)$ . Options include:

arrow	The default is for the line to have no arrowheads. Set this option to FIRST to get an arrowhead at the $(x_0, y_0)$ end of the line; set it to "last" to get an arrowhead at the far end; set it to "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 "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 12.
stipple	To draw a stippled line, set this option to a bitmap that specifies the stippling pattern, such as "gray25".
tags	The tags to be associated with the object, as a sequence of strings.
width	The line's width. Default is 1 pixel.

## 5.5 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  $(x_0, y_0)$  of the top left corner and the coordinates  $(x_1, y_1)$  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, *options )
```

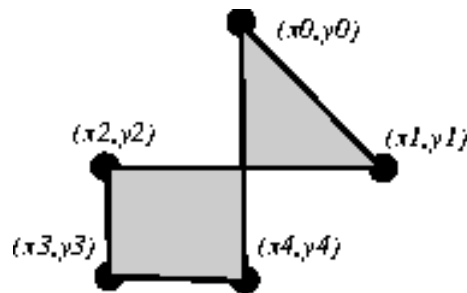
which returns the integer ID number of the new oval object on canvas *C*.

The options for ovals:

fill	The default appearance of ellipse is transparent, and a value of "" 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.
outline	The color of the border around the outside of the ellipse. Default is black.
stipple	A bitmap indicating how the interior of the ellipse will be stippled. Default is "" (solid). Typical value: "gray25". Has no effect unless fill has been set to some color.
tags	The tags to be associated with the object, as a sequence of strings.
width	Width of the border around the outside of the ellipse. Default is 1 pixel. If you set this to zero, the border will not appear. If you set this to zero and the fill transparent, you can make it disappear altogether.

## 5.6 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  $[(x_0, y_0), (x_1, y_1), \dots, (x_n, y_n)]$ , but the actual perimeter includes one more segment from  $(x_n, y_n)$  back to  $(x_0, y_0)$ . 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, ..., *options )
```

which returns an integer ID number for that object on canvas *C*. 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 the fill option to "" to get this behavior.
outline	Color of the outline; defaults to "black". Setting this option to "" makes the outline transparent.

<code>smooth</code>	The default outline uses straight lines to connect the vertices, and you can set this option to 0 to get that behavior. If you set it to 1, you get a continuous spline curve. Moreover, if you set <code>smooth</code> to 1, you can make any segment straight by duplicating the coordinates at each end of that segment.
<code>splinsteps</code>	If the <code>smooth</code> option is true, each spline is rendered as a number of straight line segments. The <code>splinsteps</code> option specifies the number of segments used to approximate each section of the line; the default is 12.
<code>stipple</code>	To render the fill color as a stippled pattern, set this option to a bitmap that specifies the stippling pattern, such as "gray25".
<code>tags</code>	The tags to be associated with the object, as a sequence of strings.
<code>width</code>	Width of the outline; defaults to 1.

### 5.7 The canvas rectangle object

Each rectangle is specified as two points:  $(x_0, y_0)$  is the top left corner, and  $(x_1, y_1)$  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, *options )
```

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

<code>fill</code>	By default, the interior of a rectangle is empty, and you can get this behavior by setting <code>fill</code> to "". You can also set the option to a color.
<code>outline</code>	The color of the border. Default is black.
<code>stipple</code>	If <code>fill</code> is set to a color, the default pattern is solid. Set <code>stipple</code> to a bitmap (such as "gray20") to get a stippled pattern.
<code>tags</code>	The tags to be associated with the object, as a sequence of strings.
<code>width</code>	Width of the border. Default is 1 pixel.

## 5.8 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, *options )
```

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

anchor	The default is CENTER, meaning that the text is centered on the (x,y) position. Other possible values are N, NE, E, SE, S, SW, W, and NW. For example, if you specify anchor=W, the text will be positioned so that point (x,y) is located at the center of the left edge of the rectangle containing the text.
fill	The default text color is black, but you can render it in any color by setting the fill option to that color.
font	If you don't like the default font, set this option to any font value. See the section on <i>Fonts</i> above.
justify	For multi-line textual displays, this option controls how the lines are justified: LEFT (the default), CENTER, or RIGHT.
stipple	If fill is set to a color, the default pattern is solid. Set stipple to a bitmap (such as "gray20") to get a stippled text pattern.
tags	The tags to be associated with the object, as a sequence of strings.
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.

## 5.9 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.

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

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

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

anchor	The default is CENTER, meaning that the window is centered on the (x,y) position. Other possible values are N, NE, E, SE, S, SW, W, and NW. For example, if you specify anchor=S, the window will be positioned so that point (x,y) is located at the center of the bottom edge of the window.
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.

tags	The tags to be associated with the object, as a sequence of strings.
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 <code>window=w</code> where <i>w</i> is the widget you want to place onto the canvas. If this is omitted initially, you can later call <code>C.itemconfigure(id, window=w)</code> to place the widget <i>w</i> onto the canvas.

## 6. The Checkbutton widget

---



The purpose of a checkbutton widget is to allow the user to read and set 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 *Control variables*, below.
- You can also use event bindings to react to user actions on the checkbutton; see *Events*, 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 raised when it is cleared.

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

```
w = Checkbutton ( master, *options )
```

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

activebackground	Background color when the checkbutton is under the cursor.
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 <code>CENTER</code> , but you can specify any of <code>N</code> , <code>NE</code> , <code>E</code> , <code>SE</code> , <code>S</code> , <code>SW</code> , <code>W</code> , or <code>NW</code> .
background	The normal background color. May be abbreviated <code>bg</code> .
bitmap	To display a monochrome image on a button, set this option to a bitmap; see <i>Bitmaps</i> , above.
borderwidth	The size of the border around the indicator. Default is 2 pixels.

<code>command</code>	A procedure to be called every time the user changes the state of this checkbox.
<code>cursor</code>	If you set this option to a cursor name (see <i>Cursors</i> , above), the mouse cursor will change to that pattern when it is over the checkbox.
<code>disabledforeground</code>	The foreground color used to render the text of a disabled checkbox. The default is a stippled version of the default foreground color.
<code>font</code>	The font used for the text.
<code>foreground</code>	The color used to render the text. Can be abbreviated <code>fg</code> .
<code>height</code>	The number of lines of text on the checkbox. Default is 1.
<code>highlightbackground</code>	The color of the focus highlight when the checkbox does not have focus. See <i>Focus</i> , below.
<code>highlightcolor</code>	The color of the focus highlight when the checkbox has the focus. See <i>Focus</i> , below.
<code>highlightthickness</code>	The thickness of the focus highlight. Default is 1. Set to 0 to suppress display of the focus highlight. See <i>Focus</i> , below.
<code>image</code>	To display a graphic image on the button, set this option to an image object. See <i>Images</i> , above.
<code>indicatoron</code>	Normally a checkbox displays an “indicator”—a box that shows whether the checkbox is set or not. You can get this behavior by setting <code>indicatoron</code> to 1. However, if you set it 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 <code>borderwidth</code> value to make it easier to see the state of such a control.
<code>justify</code>	If the text contains multiple lines, this option controls how the text is justified: <code>CENTER</code> , <code>LEFT</code> , or <code>RIGHT</code> .
<code>offvalue</code>	Normally, a checkbox’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 <code>offvalue</code> to that value.
<code>onvalue</code>	Normally, a checkbox’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 <code>onvalue</code> to that value.
<code>padx</code>	How much space to leave to the left and right of the checkbox and text. Default is 1.
<code>pady</code>	How much space to leave above and below the checkbox and text. Default is 1.
<code>relief</code>	With the default relief, <code>FLAT</code> , the checkbox does not stand out from its background. You may set this option to <code>RAISED</code> , <code>SUNKEN</code> , <code>RIDGE</code> , <code>GROOVE</code> , or <code>SOLID</code> (which gives you a solid black frame around it).
<code>selectcolor</code>	The color of the checkbox when it is set. Default is red.



<code>selectimage</code>	If you set this option to an image (see <i>Images</i> , above) that image will appear in the checkbox when it is set.
<code>state</code>	The default is NORMAL, but you can set it to DISABLED to gray out the control and make it unresponsive. If the cursor is currently over the checkbox, the state is ACTIVE.
<code>takefocus</code>	The default is that the input focus (see <i>Focus</i> , below) will pass through a checkbox. If you set this option to 0, focus will not pass through.
<code>text</code>	The label displayed next to the checkbox. Use newlines (" <code>\n</code> ") to display multiple lines of text.
<code>textvariable</code>	If you need to change the label on a checkbox during execution, create a <code>StringVar</code> (see <i>Control variables</i> , below) to manage the current value, and set this option to that control variable. Whenever the control variable's value changes, the checkbox's annotation will automatically change as well.
<code>underline</code>	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.
<code>variable</code>	The control variable that tracks the current state of the checkbox; see <i>Control variables</i> , below. Normally this variable is an <code>IntVar</code> , and 0 means cleared and 1 means set, but see the <code>offvalue</code> and <code>onvalue</code> options above.
<code>width</code>	The default width of a checkbox is determined by the size of the displayed image or text. You can set this option to a number of characters and the checkbox will always have room for that many characters.
<code>wraplength</code>	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 checkbox include:

`.deselect()`

Clears (turns off) the checkbox.

`.flash()`

Flashes the checkbox 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 checkbox to change its state.

`.select()`

Sets (turns on) the checkbox.

`.toggle()`

Clears the checkbox if set, sets it if cleared.

## 7. 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 the Text widget, below.
- If you want to display a single line of text that *cannot* be modified, see the Label widget, below.
- For multiple-line displays that can't be modified, see the Message widget, below.

To create a new Entry widget in a root window or frame *master*:

```
w = Entry ( master, *options )
```

This constructor returns the entry widget object. Options include:

background	The background color inside the entry area. May be abbreviated bg.
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 <i>Cursors</i> , above.
font	The font used for the text in the window; see <i>Fonts</i> , above.
foreground	The color used to render the text. Default is black. May be abbreviated as fg.
highlightbackground	Color of the focus highlight when the widget does not have focus.
highlightcolor	Color shown in the focus highlight when the widget has the focus.
highlightthickness	Thickness of the focus highlight.
justify	This option controls where the text is located when the text doesn't fill the widget: LEFT (the default), CENTER, or RIGHT.
relief	Selects three-dimensional shading effects. The default is SUNKEN. May also be RAISED, GROOVE, RIDGE, or FLAT.
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.
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 this option to "*".
state	The default is NORMAL, but you can set it to DISABLED to make the control unresponsive to user events. If the cursor is currently over the entry, the state is ACTIVE.
takefocus	By default, the focus will tab through entry widgets. Set this option to 0 to take the widget out of the sequence. See <i>Focus</i> , below.

<code>textvariable</code>	In order to be able to retrieve the current text from your entry widget, you must set this option to an instance of the <code>StringVar</code> class; see <i>Control variables</i> , below. You can then retrieve the text using <code>v.get()</code> and set it using <code>v.set()</code> , where <i>v</i> is the associated control variable.
<code>width</code>	The size of the entry in characters. For proportional fonts, the physical length of the widget will be based on the average width of a character times the value of the <code>width</code> option. The default is 20.
<code>xscrollcommand</code>	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 <code>.set</code> method of the scrollbar. See below for more on scrollable entry widgets.

Methods on Entry objects include:

```
.delete ( first, last=None )
```

Deletes characters from the entry. The `first` argument is the index of the first character to be deleted. The `last` argument can be:

- Omitted: the single character at position `first` is deleted.
- Set to an integer: characters up to but *not* including the one at position `last` are deleted.
- Set to the string "end": Characters are deleted up to the end.

```
.get()
```

Returns the entry's current text as a string.

```
.icursor ( index )
```

Set the insertion cursor just before position `index`.

```
.index ( index )
```

Shift the contents of the entry so that the character at position `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 position `index`.

```
.xview ( index )
```

Same as `.index()`.

```
.xview_moveto ( λ )
```

Positions the text in the entry so that the character at position `λ`, relative to the entire text, is positioned at the left edge of the window. The `λ` 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. Examples:

```
entry.xview_scroll(-1, "pages")    # 1 page to the right
entry.xview_scroll(4, "units")     # 4 chars to the left
```

## 7.1 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 )
```

## 8. 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 related widgets into a compound widget and pack 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 interactions between a group of related widgets.

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

```
w = Frame ( master, *options )
```

The constructor returns the new frame widget. Options:

background	The frame's background color. May be abbreviated bg.
borderwidth	Width of the frame's border in pixels. The default is 0 (no border).
cursor	The cursor used when the mouse is within the frame widget; see <i>Cursors</i> , above.
height	The vertical dimension of the new frame. This will be ignored unless you also call <code>.grid_propagate(0)</code> on the frame.

<code>highlightbackground</code>	Color of the focus highlight when the frame does not have focus.
<code>highlightcolor</code>	Color shown in the focus highlight when the frame has the focus.
<code>highlightthickness</code>	Thickness of the focus highlight.
<code>relief</code>	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 <code>borderwidth</code> to a positive value and set its <code>relief</code> to one of RAISED, SUNKEN, GROOVE, or RIDGE.
<code>takefocus</code>	Normally, frame widgets are not visited by input focus (see <i>Focus</i> , below, for an overview of this topic). However, you can set this option to 1 if you want the frame to receive keyboard input. To handle such input, you will need to create bindings for keyboard events; see <i>Events</i> , below, for more on events and bindings.
<code>width</code>	The horizontal dimension of the new frame. This will be ignored unless you also call <code>.grid_propagate(0)</code> on the frame.

## 9. 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, *options )
```

The constructor returns the label widget. Options include:

<code>anchor</code>	This options controls where the text is positioned if the widget has more space than the text needs. The default is CENTER, which centers the text in the available space. Suppose the width is set to 10 and the height is set to 5, and the text occupies only a 4×2 area: the anchor option tells Tkinter where in the full area to place that text. Setting it to SW would put it in the lower left corner, and a value of E centers it on the right side. Other values include W, NW, N, NE, and SE. We apologize in advance to our readers in the Southern Hemisphere who object to the assumption that north is up.
<code>background</code>	The background color of the label area. See the section on <i>Colors</i> , above. May be abbreviated as "bg".
<code>bitmap</code>	Set this option equal to a bitmap or image object and the label will display that graphic. See the sections on <i>Bitmaps</i> and <i>Images</i> , above.
<code>borderwidth</code>	Width of the border around the label. Default is 2.
<code>cursor</code>	Cursor that appears over this label. See <i>Cursors</i> , above.

font	If you are displaying text in this label (with the <code>text</code> or <code>textvariable</code> option, the <code>font</code> option specifies in what font that text will be displayed. See <i>Fonts</i> , above.
foreground	If you are displaying text or a bitmap in this label, the <code>foreground</code> option specifies the color of the text, or of the 1-bits in the bitmap. See <i>Colors</i> , above. May be abbreviated as <code>fg</code> .
height	Height of the label in <i>lines</i> (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 <i>Images</i> , above.
justify	Specifies how multiple lines of text will be aligned with respect to each other: <code>LEFT</code> , <code>CENTER</code> (the default), or <code>RIGHT</code> .
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 <code>FLAT</code> ; may also be <code>RIDGE</code> , <code>GROOVE</code> , <code>RAISED</code> , or <code>SUNKEN</code> .
takefocus	The default is that focus will not visit a label widget; set this option to 1 if you want this widget to be visited by focus. See <i>Focus</i> , below.
text	To display one or more lines of text in a label widget, set this option to a string containing the text. Internal newlines (" <code>\n</code> ") will force a line break.
textvariable	To slave the text displayed in a label widget to a control variable of class <code>StringVar</code> , set this option to that variable. See <i>Control variables</i> , below.
underline	You can display an underline ( <code>_</code> ) below the <i>n</i> th letter of the text, counting from 0, by setting this option to <i>n</i> . The default is <code>-1</code> , which means no underlining.
width	Width of the label in <i>characters</i> (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 length limit. 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 *Universal widget methods*, above).

## 10. 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 of a list of items. If you need something more like a text editor, see the Text widget, below.

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

```
w = Listbox ( master, *options )
```

This constructor returns the new listbox widget. Options:

background	The background color in the listbox. May be abbreviated as bg.
borderwidth	The width of the border around the listbox. Default is 2. May be abbreviated as bd.
cursor	The cursor that appears when the mouse is over the listbox.
font	The font used for the text in the listbox. See <i>Fonts</i> , above.
foreground	The color used for the text in the listbox. See <i>Colors</i> , above. May be abbreviated fg.
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.
highlightcolor	Color shown in the focus highlight when the widget has the focus.
highlightthickness	Thickness of the focus highlight.
relief	Selects three-dimensional shading effects. The default is SUNKEN. May also be RAISED, GROOVE, RIDGE, or FLAT.
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.
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 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.

<code>takefocus</code>	By default, the focus will tab through listbox widgets. Set this option to 0 to take the widget out of the sequence. See <i>Focus</i> , below.
<code>width</code>	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.
<code>xscrollcommand</code>	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 <code>.set</code> method of the scrollbar. See below for more on scrollable listbox widgets.
<code>yscrollcommand</code>	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 <code>.set</code> method of the scrollbar. See below for more on scrollable listbox widgets.

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

- 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 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 specified 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 *Scrolling a Listbox widget*, below.

```
.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 listbox leftward relative to the text, positive values rightward.

```
.yview()
```

To make the listbox vertically scrollable, set the `command` option of the associated vertical scrollbar to this method. See *Scrolling a Listbox widget*, below.

```
.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 listbox upward relative to the text, positive values downward.

## 10.1 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
```

## 11. The Menu widget

---

“Drop-down” menus are a very 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.

Definitions:

- 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 they click on 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 window appears containing the choices.

Refer to the section on the Menubutton widget, 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 checkbox.
- A group of radiobuttons.

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.
activeborderwidth	Specifies the width of a border drawn around a choice when it is under the mouse. Default is 1 pixel.
activeforeground	The foreground color that will appear on a choice when it is under the mouse.
background	The background color for choices not under the mouse. May be abbreviated bg.
borderwidth	The width of the border around all the choices. Default is 1. May be abbreviated as bd.
cursor	The cursor that appears when the mouse is over the choices, but only when the menu has been torn off.
disabledforeground	The color of the text for items whose state is DISABLED.
font	The default font for textual choices.
foreground	The foreground color used for choices not under the mouse. May be abbreviated fg.
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 RAISED. You can set it to FLAT, SUNKEN, RIDGE, or GROOVE.
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 this option to 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.

These methods are available on Menu objects. The ones that create choices on the menu have their own particular options, discussed below under `coption`.

```
.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 (described above).

```
.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 much the same way as you would set up a Radiobutton object (described below).

```
.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, option )
```

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

```
.entryconfigure ( index, *option )
```

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 `option=value` arguments.

```
.index ( index )
```

Returns the position of the choice specified by `index`. 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, *coptions )
```

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, *coptions )
```

Insert a new checkbutton at position index. Options are the same as for .add\_checkbutton().

```
.insert_command ( index, *coptions )
```

Insert a new command at position index. Options are the same as for .add\_command().

```
.insert_radiobutton ( index, *coptions )
```

Insert a new radiobutton at position index. Options are the same as for .add\_radiobutton().

```
.insert_separator ( index )
```

Insert a new separator at position index.

```
.invoke ( index )
```

Calls the command callback associated with the choice at position index. If a checkbutton, its state is toggled; if a radiobutton, that choice is set.

```
.type ( index )
```

Returns the type of the choice specified by index: one of "cascade", "checkbutton", "command", "radiobutton", "separator", or "tearoff".

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 <i>not</i> under the mouse. Note that this <i>cannot</i> be abbreviated as bg.
bitmap	Display a bitmap for this choice; see <i>Bitmaps</i> , above.
columnbreak	Normally all the choices are displayed in one long column. If you can set this option to 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.
foreground	The foreground color used for choices when they are <i>not</i> under the mouse. Note that this <i>cannot</i> be abbreviated as bg.
image	Display an image for this choice; see <i>Images</i> , above.
label	Display this string as the text 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 0 when the checkbutton is off. You can change the off value by setting this option to that value.

onvalue	Normally, the control variable for a checkbox is 1 when the checkbox is on. You can change the on value by setting this option to that value.
selectcolor	Normally, the color displayed in a set checkbox or radiobutton is red. Change that color by setting this option to the color you want.
state	Normally all choices react to mouse clicks, but you can set this option to 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 <i>Control variables</i> , below) 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 checkboxes or radiobuttons, this option should be set to the control variable associated with the checkbox or group of radiobuttons. See <i>Control variables</i> , below.

## 12. 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, *options )
```

The constructor returns the new menubutton widget. Options:

activebackground	The background color when the mouse is over the menubutton.
activeforeground	The foreground color when the mouse is over the menubutton.
anchor	This options controls where the text is positioned if the widget has more space than the text needs. The default is CENTER, which centers the text; other values include E, SE, S, SW, W, NW, N, and NE.
background	The background color when the mouse is not over the menubutton. May be abbreviated as bg.
bitmap	To display a bitmap on the menubutton, set this option to a bitmap name; see <i>Bitmaps</i> , above.
borderwidth	Width of the border around the menubutton. Default is 2 pixels. May be abbreviated as bd.
cursor	The cursor that appears when the mouse is over this menubutton.

<code>direction</code>	Normally, the menu will appear below the menubutton. Set this option to "left", "right", or "above" to change where menu appears relative to the menubutton.
<code>disabledforeground</code>	The foreground color shown on this menubutton when it is disabled.
<code>font</code>	The font used to display text on the button. See <i>Fonts</i> , above.
<code>foreground</code>	The foreground color when the mouse is not over the menubutton. May be abbreviated as fg.
<code>height</code>	The height of the menubutton in <i>lines</i> of text (not pixels!).a The default is to fit the menubutton's size to its contents.
<code>highlightbackground</code>	Color of the focus highlight when the widget does not have focus.
<code>highlightcolor</code>	Color shown in the focus highlight when the widget has the focus.
<code>highlightthickness</code>	Thickness of the focus highlight.
<code>image</code>	To make an image appear on this menubutton, set this option to the image object. See <i>Images</i> , above.
<code>justify</code>	This option controls where the text is located when the text doesn't fill the menubutton: LEFT (the default), CENTER, or RIGHT.
<code>menu</code>	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.
<code>padx</code>	How much space to leave to the left and right of the text of the menubutton. Default is 1.
<code>pady</code>	How much space to leave above and below the text of the menubutton. Default is 1.
<code>relief</code>	Normally, menubuttons will have RAISED appearance, but you can set this option to FLAT, SUNKEN, RIDGE, or GROOVE relief.
<code>state</code>	Normally, menubuttons respond to the mouse. Set this option to DISABLED to gray out the menubutton and make it unresponsive.
<code>text</code>	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.
<code>textvariable</code>	You can associate a control variable of class StringVar with this menubutton. Setting that control variable will change the displayed text. See <i>Control variables</i> , below.
<code>underline</code>	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.

<code>width</code>	Width of the menubutton in <i>characters</i> (not pixels!). If this option is not set, the label will be sized to fit its contents.
<code>wraplength</code>	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

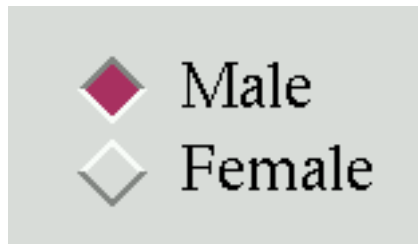
self.mayoVar = IntVar()
self.ketchVar = IntVar()
self.mb.menu.add_checkbutton ( label="mayo",
                              variable=self.mayoVar )
self.mb.menu.add_checkbutton ( label="ketchup",
                              variable=self.ketchVar )
```

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

### 13. The Radiobutton *widget*

---

Radiobuttons are sets of related widgets usually used to 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 araised.
- To form several radiobuttons into functional groups, create a single control variable (see *Control variables*, 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, *options )
```

This constructor returns the new radiobutton widget. Options:

activebackground	The background color when the mouse is over the radiobutton.
activeforeground	The foreground color when the mouse is over the radiobutton.
anchor	If the widget inhabits a space larger than it needs, this option specifies where the radiobutton will sit in that space. The default is CENTER, but you can specify any of N, NE, E, SE, S, SW, W, or NW.
background	The normal background color. May be abbreviated bg.
bitmap	To display a monochrome image on a radiobutton, set this option to a bitmap; see <i>Bitmaps</i> , above.
borderwidth	The size of the border around the indicator part itself. Default is 2 pixels.
command	A procedure to be called every time the user changes the state of this radiobutton.
cursor	If you set this option to a cursor name (see <i>Cursors</i> , above), 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.
foreground	The color used to render the text. Can be abbreviated fg.
height	The number of lines of text on the radiobutton. Default is 1.
highlightbackground	The color of the focus highlight when the radiobutton does not have focus. See <i>Focus</i> , below.
highlightcolor	The color of the focus highlight when the radiobutton has the focus. See <i>Focus</i> , below.
highlightthickness	The thickness of the focus highlight. Default is 1. Set to 0 to suppress display of the focus highlight. See <i>Focus</i> , below.
image	To display a graphic image instead of text for this radiobutton, set this option to an image object. See <i>Images</i> , above. The image appears when the radiobutton is <i>not</i> selected; compare <i>selectimage</i> , below.

<code>indicatoron</code>	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 <code>borderwidth</code> value to make it easier to see the state of such a control.
<code>justify</code>	If the text contains multiple lines, this option controls how the text is justified: <code>CENTER</code> , <code>LEFT</code> , or <code>RIGHT</code> .
<code>padx</code>	How much space to leave to the left and right of the radiobutton and text. Default is 1.
<code>pady</code>	How much space to leave above and below the radiobutton and text. Default is 1.
<code>relief</code>	With the default relief, <code>FLAT</code> , the radiobutton does not stand out from its background. You may set this option to <code>RAISED</code> , <code>SUNKEN</code> , <code>RIDGE</code> , <code>GROOVE</code> , or <code>SOLID</code> (which gives you a solid black frame around it).
<code>selectcolor</code>	The color of the radiobutton when it is set. Default is red.
<code>selectimage</code>	If you are using the <code>image</code> option to display a graphic instead of text when the radiobutton is cleared, you can set the <code>selectimage</code> option to a different image that will be displayed when the radiobutton is set.
<code>state</code>	The default is <code>NORMAL</code> , but you can set it to <code>DISABLED</code> to gray out the control and make it unresponsive. If the cursor is currently over the radiobutton, the state is <code>ACTIVE</code> .
<code>takefocus</code>	The default is that the input focus (see <i>Focus</i> , below) will pass through a radiobutton. If you set this option to 0, focus will not pass through.
<code>text</code>	The label displayed next to the radiobutton. Use newlines (“\n”) to display multiple lines of text.
<code>textvariable</code>	If you need to change the label on a radiobutton during execution, create a <code>StringVar</code> (see <i>Control variables</i> , below) 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 as well.
<code>underline</code>	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.
<code>value</code>	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 <code>IntVar</code> , give each radiobutton in the group a different integer value option. If the control variable is a <code>StringVar</code> , give each radiobutton a different string value option.

variable	The control variable that this radiobutton shares with the other radiobuttons in the group. This can be either an IntVar or a StringVar.
width	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 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.

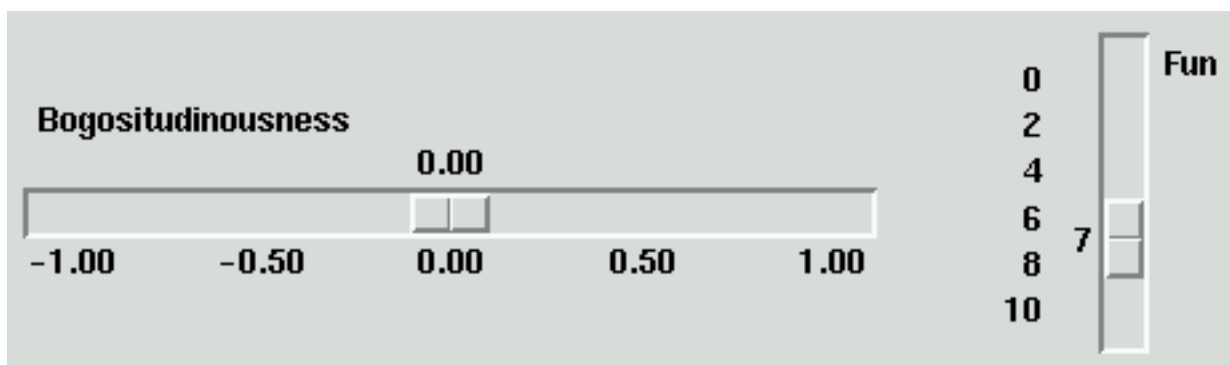
`.select()`

Sets (turns on) the radiobutton.

## 14. 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, *options )
```

The constructor returns the new scale widget. Options:

activebackground	The color of the slider when the mouse is over it.
background	The background color of the parts of the widget that are outside the trough. May be abbreviated bg.
borderwidth	Width of the 3-d border around the trough and slider. Default is 2. May be abbreviated bd.
command	A procedure to be called every time the slider is moved.
cursor	The cursor that appears when the mouse is over the scale. See <i>Cursors</i> , above.
digits	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.
foreground	The color of the text used for the label and annotations. May be abbreviated as fg.
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_. 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 <i>Focus</i> , below.
highlightcolor	The color of the focus highlight when the scale has the focus. See <i>Focus</i> , below.
highlightthickness	The thickness of the focus highlight. Default is 1. Set to 0 to suppress display of the focus highlight. See <i>Focus</i> , below.
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 <i>x</i> dimension if the scale is horizontal, or the <i>y</i> dimension if vertical. The default is 100 pixels.
orient	Set this option to HORIZONTAL if you want the scale to run along the <i>x</i> dimension, or VERTICAL to run along <i>y</i> . Default is horizontal.

<code>relief</code>	With the default relief, <code>FLAT</code> , the scale does not stand out from its background. You may set this option to <code>RAISED</code> , <code>SUNKEN</code> , <code>RIDGE</code> , <code>GROOVE</code> , or <code>SOLID</code> (which gives you a solid black frame around it).
<code>repeatdelay</code>	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 300, and the units are milliseconds.
<code>repeatinterval</code>	This option controls how often slider movement will repeat when button 1 is held down in the trough. Default is 100, and the units are milliseconds.
<code>resolution</code>	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 <code>from_=-1.0</code> and <code>to=1.0</code> , and you set <code>resolution=0.5</code> , the scale will have 5 possible values: -1.0, -0.5, 0.0, +0.5, and +1.0. All smaller movements will be ignored.
<code>showvalue</code>	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.
<code>sliderlength</code>	Normally the slider is 30 pixels along the length of the scale. You can change that length by setting the <code>sliderlength</code> option to your desired length.
<code>state</code>	Normally, scale widgets respond to mouse events, and when they have the focus, also keyboard events. Set this option to <code>DISABLED</code> to make the widget unresponsive.
<code>takefocus</code>	Normally, the focus will cycle through scale widgets. Set this option to 0 if you don't want this behavior. See <i>Focus</i> , below.
<code>tickinterval</code>	Normally, there are no "ticks" 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 <code>from_=0.0</code> , <code>to=1.0</code> , and <code>tickinterval=0.25</code> , 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.
<code>to</code>	A float or integer value that defines one end of the scale's range; the other end is defined by the <code>from_</code> option, discussed above. The <code>to</code> value can be either greater than or less than the <code>from_</code> value. For vertical scales, the <code>to</code> value defines the bottom of the scale; for horizontal scales, the right end.
<code>troughcolor</code>	The color of the trough.

variable	The control variable for this scale, if any. See <i>Control variables</i> , below. Control variables may be from class <code>IntVar</code> , <code>DoubleVar</code> (float), or <code>StringVar</code> . In the latter case, the numerical value will be converted to a string. See the <code>digits</code> option, above, for more information on this conversion.
width	The width of the trough part of the widget. This is the <i>x</i> dimension for vertical scales and the <i>y</i> dimension if the scale has horizontal orientation. Default is 15 pixels.

Scale objects have these methods:

`.get()`

This method returns the current value of the scale.

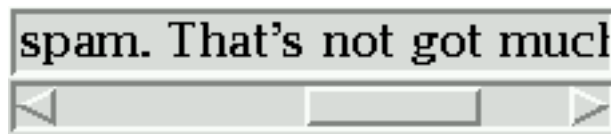
`.set(value)`

Sets the scale's value.

## 15. 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, like a canvas, 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, *options )
```

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.
background	The color of the slider and arrowheads when the mouse is not over them. May be abbreviated as bg.
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. May be abbreviated as bd.
command	A procedure to be called whenever the scrollbar is moved. See below for a discussion of the calling sequence.
cursor	The cursor that appears when the mouse is over the scrollbar. See <i>Cursors</i> , above.
elementborderwidth	The width of the borders around the arrowheads and slider. The default is -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 <i>Focus</i> , below.
highlightcolor	The color of the focus highlight when the scrollbar has the focus. See <i>Focus</i> , below.
highlightthickness	The thickness of the focus highlight. Default is 1. Set to 0 to suppress display of the focus highlight. See <i>Focus</i> , below.
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 this option to HORIZONTAL for a horizontal scrollbar, 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 300, and the units are milliseconds.

<code>repeatinterval</code>	This option controls how often slider movement will repeat when button 1 is held down in the trough. Default is 100, and the units are milliseconds.
<code>takefocus</code>	Normally, you can tab the focus through a scrollbar widget. Set this option to 0 if you don't want this behavior. The default key bindings for scrollbars allow the user to use the left and right arrow keys to move horizontal scrollbars and up and down arrow keys to move vertical scrollbars.
<code>troughcolor</code>	The color of the trough.
<code>width</code>	Width of the scrollbar (its <i>y</i> dimension if horizontal, and its <i>x</i> dimension if vertical). Default is 16.

Methods on scrollbar objects include:

```
.set ( first, last )
```

To connect a scrollbar to another widget *w*, set *w*'s `xscrollcommand` or `yscrollcommand` to the scrollbar's `.set` method. The `first` and `last` arguments tell the scrollbar where and how big to make the slider. For example, `.set(0.5, 0.75)` would make the slider extend from halfway along the trough to three-quarters of the way.

### 15.1 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 of  $\lambda$  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",  $\lambda$ )
```

These calling sequences match the arguments expected by the `.xview()` and `.yview()` methods of canvases, listboxes, and text widgets. However, for some reason, the entry widget does not have an `.xview()` method. See the section *Scrolling an Entry widget*, above.



## 15.2 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.

## 16. 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 in different fonts, colors, and backgrounds.
- You can intersperse embedded images with text. An image is treated as a single character.
- 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.
- A text widget may contain invisible *mark* objects between character positions.

To use a text widget, there are two other concepts you will need to understand:

- An *index* is a way of describing a specific position between two characters of a text widget.
- 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.

Refer to the separate sections below for more on the theory and practice of indices, marks, images, windows, and tags.

## 16.1 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:

<code>line.column</code>	The position just before the given <i>column</i> (counting from zero) on the given <i>line</i> (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.
<code>line.end</code>	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.
<i>markname</i>	You can use a mark as an index; just pass its name where an index is expected. See <i>Marks in text widgets</i> , below.
<code>tag.first</code>	The position before the first character of the region tagged with name <i>tag</i> ; see <i>Tags in text widgets</i> , below.
<code>tag.last</code>	The position after the last character of a tagged region.
<code>"@x,y"</code>	The position before the character closest to the coordinate ( <i>x</i> , <i>y</i> ).
<i>embedded-object</i>	If you have an image or widget embedded in the text widget, you can use the PhotoImage, BitmapImage, or embedded widget as an index.

## 16.2 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 floating 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 *Methods on text widgets*, below, to see how to use marks.

### 16.3 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 *Images* and *Bitmaps*, above.

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.

### 16.4 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. Refer to *Methods on text widgets*, below, for the calling sequence and related methods.

### 16.5 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 *Methods on text widgets*, below, for information on this and related methods.

## 16.6 Methods on text widgets

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

```
w = Text ( master, *options )
```

The constructor returns the new text widget. Options include:

background	The default background color of the text widget. May be abbreviated as bg.
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 <i>Cursors</i> , above.
exportselection	Normally, text selected within a text widget is exported to be the selection in the window manager. Set this option to 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 <i>Fonts</i> , above.
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. May be abbreviated as fg.
height	The height of the widget in <i>lines</i> (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 <i>Focus</i> , below.
highlightcolor	The color of the focus highlight when the text widget has the focus. See <i>Focus</i> , below.
highlightthickness	The thickness of the focus highlight. Default is 1. Set to 0 to suppress display of the focus highlight. See <i>Focus</i> , below.
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 in its line). Default is 2 pixels.
padx	The size of the internal padding added to the left and right of the text area. Default is one pixel.
pady	The size of the internal padding added above and below the text area. Default is one pixel.
relief	The 3-D appearance of the text widget. Default is <i>SUNKEN</i> .
selectbackground	The background color to use displaying selected text.
selectborderwidth	The width of the border to use around selected text.

<code>selectforeground</code>	The foreground color to use displaying selected text.
<code>spacing1</code>	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.
<code>spacing2</code>	This option specifies how much extra vertical space to add between displayed lines of text when a logical line wraps. Default is 0.
<code>spacing3</code>	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.
<code>state</code>	Normally, text widgets respond to keyboard and mouse events; set this option to <code>NORMAL</code> to get this behavior. If you set <code>state</code> to <code>DISABLED</code> , the text widget will not respond, but you won't be able to modify its contents programmatically either.
<code>tabs</code>	<p>This option controls how tab characters position text. 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.</p> <p>For example, setting <code>tabs</code> to (<code>"3c"</code>, <code>"5c"</code>, <code>"12c"</code>) would put tab stops 3, 5, and 12cm from the left side. Tabs past the last one you set have the same width as the distance between the last two existing tab stops.</p> <p>Normally, text after a tab character is aligned with its left edge on the tab stop, but you can include any of the keywords <code>LEFT</code>, <code>RIGHT</code>, <code>CENTER</code>, or <code>NUMERIC</code> in the list after a distance, and that will change the positioning of the text after each tab.</p> <p>A <code>LEFT</code> tab stop has the default behavior. A <code>RIGHT</code> tab stop will position the text so its right edge is on the stop, and a <code>CENTER</code> tab will center the following text on the tab stop. A <code>NUMERIC</code> 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 be positioned to its right.</p> <p>For example, setting <code>tabs</code> to the sequence (<code>"0.5i"</code>, <code>"0.8i"</code>, <code>RIGHT</code>, <code>"1.2i"</code>, <code>CENTER</code>, <code>"2i"</code>, <code>NUMERIC</code>) 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.</p>
<code>takefocus</code>	Normally, focus will visit a text widget (see <i>Focus</i> , below). Set <code>takefocus</code> to 0 if you do not want focus in the widget.
<code>width</code>	The width of the widget in <i>characters</i> (not pixels!), measured according to the current font size.

<code>wrap</code>	This option controls the display of lines that are too wide. With the default behavior, value <code>CHAR</code> , any line that gets too long will be broken at any character. Set the option to <code>WORD</code> 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 this option to <code>NONE</code> and provide a horizontal scrollbar.
<code>xscrollcommand</code>	To make the text widget horizontally scrollable, set this option to the <code>.set</code> method of the horizontal scrollbar.
<code>yscrollcommand</code>	To make the text widget vertically scrollable, set this option to the <code>.set</code> method of the vertical scrollbar.

Methods on 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, *options )`

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 `Marks in text widgets`, 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, *options )
```

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 <sub>1</sub> ...] (...) * + ? e <sub>1</sub>   e <sub>2</sub>
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 it is.

```
.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 *Events*, 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, *options )
```

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 "gray12", "gray25", "gray50", or "gray75" (from lighter to darker). 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 to one of the standard bitmap names specified above under bgstipple.
font	The font used to display text with this tag.
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.



<code>lmargin2</code>	How much to indent successive lines of a chunk of text that has this tag. The default is 0.
<code>offset</code>	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.
<code>overstrike</code>	Set this option to 1 to draw a horizontal line through the center of text with this tag.
<code>relief</code>	Which 3-D effect to use for text with this tag: one of FLAT (the default), SUNKEN, RAISED, GROOVE, or RIDGE.
<code>rmargin</code>	Size of the right margin for chunks of text with this tag. Default is 0.
<code>spacing1</code>	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.
<code>spacing2</code>	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.
<code>spacing3</code>	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.
<code>tabs</code>	How tabs are expanded on lines with this tag. See the description of the <code>tabs</code> option for text widgets, above.
<code>underline</code>	Set this option to 1 to underline text with this tag.
<code>wrap</code>	How long lines are wrapped in text with this tag. See the description of the <code>wrap</code> 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 *Tags in text widgets*, 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  $[i_0, i_1]$ , where  $i_0$  is the index of the first tagged character and  $i_1$  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 *Tags in text widgets*, above, for an explanation of the tag stack). If you pass two arguments, the tag with name `tagName` is moved to a position just above the tag with name `aboveThis`. If you pass only one argument, that tag is moved to the top of the tag stack.

```
.tag_ranges ( tagName )
```

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

```
.tag_remove ( tagName, index1, index2=None )
```

Removes the tag named `tagName` from all characters between `index1` and `index2`. If `index2` is omitted, the tag is removed from the single character after `index1`.

```
.tag_unbind ( tagName, sequence, funcid=None )
```

Remove the event binding for the given sequence from the tag named `tagName`. If there are multiple handlers for this sequence and tag, you can remove only one handler by passing it as the third argument.

```
.window_cget ( index, option )
```

Returns the value of the given option for the embedded widget at the given index.

```
.window_configure ( index, option )
```

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

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

```
.window_create ( index, *options )
```

This method creates a window where a widget can be embedded within a text widget. There are two ways to provide the embedded widget:

- you can use pass the widget to the window option in this method, or
- 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:

<code>align</code>	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: <code>CENTER</code> (the default), which centers the widget vertically within the line; <code>TOP</code> , which places the top of the image at the top of the line; <code>BOTTOM</code> , which places the bottom of the image at the bottom of the line; and <code>BASELINE</code> , which aligns the bottom of the image with the text baseline.
<code>create</code>	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.
<code>padx</code>	Extra space added to the left and right of the widget within the text line. Default is 0.
<code>pady</code>	Extra space added above and below the widget within the text line. Default is 0.
<code>stretch</code>	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 <code>stretch</code> to 1, the widget is stretched vertically to fill the height of the line, and the <code>align</code> option is ignored.
<code>window</code>	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 )
.xview ( SCROLL, n, what )
```

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 `offset`, where 0.0 moves the text to its leftmost position and 1.0 to its rightmost position. 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)
.yview(SCROLL, n, what)
.yview_moveto(fraction)
.yview_scroll(n, what)
```

These are the same as the `.xview()` and related methods, only for vertical scrolling. When scrolling vertically by `UNITS`, the units are lines.

## 17. Toplevel: *Top-level window methods*

---

You can get the top-level widget from any widget *w* using *w.wm\_toplevel()*.

`Toplevel ( options )`

Options include:

background	The background color of the window. May be abbreviated as bg.
borderwidth	Border width in pixels. May be abbreviated as bd.
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 <code>class_="Screamer"</code> . Then you can put a line in your option database like this:  <code>*Screamer*background: red</code>  and then, if you use the <code>.option_readfile()</code> method to read your option database, all widgets with that class name will default to a red background. This option is named <code>class_</code> because in Python <code>class</code> is a reserved word.
cursor	The cursor that appears when the mouse is in this window.
height	Window height in pixels.
relief	One of: FLAT (the default), SUNKEN, RAISED, GROOVE, or RIDGE.
width	Window width in pixels.

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 iconified, expand.

`.geometry ( newGeometry=None )`

Set the window geometry. The argument has the form "*w*x*h* ± *x* ± *y*"; if 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 ( string=None )
```

Set the window title. If the argument is omitted, returns the current title.

```
.withdraw()
```

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

## **18. 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 `ms`, 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 *Events*, below, 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 keep them both 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 *Events*, below, 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 ( string )
```

Appends the given 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).

```
.config ( option=value, ... )  
.configure ( option=value, ... )
```

Set the values of one or more options. For the options whose names are Python reserved words (class, tt from, in), use a trailing underbar: class\_, tt 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).

You can use either .config() or .configure(); the two names are equivalent.

```
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 *Events*, below, 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 *Events*, below, 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 *Focus*, below, for a general description of input focus.

```
w.focus_force()
```

Force the input focus to the widget. This is impolite. Better to wait for the window manager to give you the focus.

```
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, the `.focus_lastfor()` method 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 *Events*, below, 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 *Events*, below, 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.

`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*. The path name for any widget *x* is `str(x)`; see also the `.winfo_pathname()` method, below.

`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:

<code>widgetDefault</code>	Level 20, for global default properties of widgets.
<code>startupFile</code>	Level 40, for default properties of specific applications.
<code>userDefault</code>	Level 60, for options that come from user files such as their <code>.Xdefaults</code> file.
<code>interactive</code>	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 *Standardizing appearance and the option database*, below, 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 overridden 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 the section on *Standardizing appearance*, below, for more on matching keys to 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 the section on *Standardizing appearance*, below, 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 the section on *Focus*, below. 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 *Events*, 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 "List-box").

`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.wininfo_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.wininfo_depth()
```

Returns the number of bits per pixel in *w*'s display.

```
w.wininfo_fpixels ( number )
```

For any dimension number (see *Dimensions*, above), this method returns that distance in pixels on *w*'s display, as a floating-point number.

```
w.wininfo_geometry()
```

Returns the geometry string describing the size and on-screen location of *w* as a string "*w**x**h*+*x*+*y*", where *w* is the width, *h* is the height, and the upper left corner of the widget is at (*x*,*y*) relative to the upper left corner of its top-level window. 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.wininfo_height()
```

Returns the current height of *w* in pixels. See the remarks on geometry updating under .wininfo\_geometry(), above.

```
w.wininfo_id()
```

Returns an integer that uniquely identifies *w* within its top-level window. You will need this for the .wininfo\_pathname() method, below.

```
w.wininfo_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.wininfo_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.wininfo_name()
```

This method returns *w*'s name relative to its parent. Widget names have a hierarchical "path name" structure somewhat like path names in a file system. A widget that is the child of a top-level window has a name of the form ".*n*", where *n* is the widget's name as returned by .wininfo\_name(). Any other widget has a name of the form *p<sub>p</sub>*.*n*, where *p<sub>p</sub>* is the path name of its parent and *n* is its own name. See .wininfo\_pathname(), below, to find out how to obtain a widget's path name.

```
w.wininfo_parent()
```

Returns *w*'s parent's path name, or an empty string if *w* is a top-level window. See .wininfo\_name(), above, for more on widget path names.

```
w.winfo_pathname ( id, displayof=0 )
```

If the `displayof` argument is false, returns the “widget 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`. Under `.winfo_id()`, above, you will see how to obtain a widget’s identifier; under `.winfo_name()`, above, is an explanation 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()  
w.winfo_pointery()  
w.winfo_pointerxy()
```

The first two methods return the  $x$  and  $y$  coordinates, respectively, of the mouse pointer relative to `w`’s root window. If the mouse pointer isn’t on the same screen, each will return `-1`. The third method returns a tuple  $(x, y)$ , or  $(-1, -1)$ .

```
w.winfo_reqheight()  
w.winfo_reqwidth()
```

These methods return the requested height and width, respectively, of widget `w`. These dimensions are the ones necessary so that all of `w`’s contents have the room they need. The actual height and width may be different due to negotiations with the geometry manager.

```
w.winfo_rgb ( color )
```

For any given color (see *Colors*, above, for the various ways of specifying colors), 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  $(0, 65535, 0)$ .

```
w.winfo_rootx()  
w.winfo_rooty()
```

Returns the  $x$  and  $y$  coordinates, respectively, of the upper left-hand corner of `w`’s root window relative to `w`’s parent. If `w` has a border, this is the outer corner of the border.

```
w.winfo_screenheight()  
w.winfo_screenwidth()  
w.winfo_screenmmheight()  
w.winfo_screenmmwidth()
```

These methods retrieve the size of the screen. The first two return the height and width, respectively, in pixels. The latter two return the screen size in millimeters, as an integer.

```
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 the section on `Toplevel`, above.

```
w.winfo_x()  
w.winfo_y()
```

These methods return the  $x$  and  $y$  coordinates, respectively, of the top left corner of  $w$  relative to its parent. If  $w$  has a border, this is the outer corner of the border.

## 19. 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 `.option_readfile()` method, above, in the section on *Universal widget methods*. See the section on *Resource specification lines*, below, 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 in the section *Universal widget methods*, above.

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:

- 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 the section *How to name a widget class*, below, for details.
- You can also give any widget an *instance name*. The default name of a widget is usually a meaningless number (e.g., "135147872" is a value that might come up in the X implementation). However, as with widget classes, you can assign a name to any widget. See the section *How to name a widget instance*, below, 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 the `.winfo_pathname()` method in the section on universal widget methods, above, for more information on pathnames.

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 the section on *Resource specification lines*, below, we'll discuss how the options database really works.

### 19.1 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()
        ...
```

### 19.2 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", ...)
```

### 19.3 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.

## 19.4 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  $n_1, n_2, n_3, \dots, o$  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.



## Part II: Connecting the interface to the rest of your program

---

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.

### 20. 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 four class constructors, depending on what type of values you want 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 the section on *Universal widget methods*, above, 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.

## 21. 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.
- 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 using the spacebar.
  - In Listbox widgets, the up- and down-arrows 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 left- and right-arrow keys, and vertical ones respond to the up- and down-arrow.
  - In a Scrollbar widget, the PAGEUP and PAGEDOWN keys move the scrollbar by pageloads. The up- and down-arrows will move vertical scrollbars by units, and the left- and right-arrow 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 its `highlight-thickness` 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 *Universal widget methods*, above, 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 *Events*, below, for the details.

## 22. 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.

## 22.1 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 *Universal methods*, above).

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 the section *Writing an event handler*, 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 *Universal methods*, 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:

```
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 *Universal methods*, above).

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

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

## 22.2 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 *type* describes the general kind of event, such as a key press or mouse click.
- 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.
- 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 below 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.

## 22.3 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 <i>detail</i> part specifies which button.
ButtonRelease	The user let up on a mouse button.
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 <i>Focus</i> , above, 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 <code>.focus_set()</code> on a widget).
FocusOut	The input focus was moved out of a widget. Again, the user can cause this event, or your program can cause it.
KeyPress	The user pressed a key on the keyboard. The <i>detail</i> 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 <code>.grid()</code> 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 <code>.grid_remove()</code> method.
Visibility	Happens when at least some part of the application window becomes visible on the screen.

## 22.4 Event modifiers

The modifier names that you can use in event sequences include:

Alt	The user is holding the ALT key down.
Any	This modifier generalizes an event type. For example, the event pattern <code>&lt;Any-KeyPress&gt;</code> applies to the pressing of any key.
Control	The user is holding the CONTROL key down.
Double	Specifies two events happening close together in time. For example, <code>&lt;Double-Button-1&gt;</code> describes two presses of button 1 in rapid succession.
Lock	The user has pressed SHIFT LOCK.
Shift	The user is holding the SHIFT key down.
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>`).

## 22.5 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 *Writing your handler*, below, for more information on `Event` objects, whose members 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` member 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.

<code>.keysym</code>	<code>.keycode</code>	<code>.keysym_num</code>	Key
<code>Alt_L</code>	64	65513	The left-hand ALT key
<code>Alt_R</code>	113	65514	The right-hand ALT key
<code>BackSpace</code>	22	65288	<code>BACKSPACE</code>
<code>Cancel</code>	110	65387	<code>BREAK</code>
<code>Caps_Lock</code>	66	65549	<code>CAPSLOCK</code>
<code>Control_L</code>	37	65507	The left-hand CONTROL key
<code>Control_R</code>	109	65508	The right-hand CONTROL key
<code>Delete</code>	107	65535	<code>DELETE</code>
<code>Down</code>	104	65364	↓
<code>End</code>	103	65367	<code>END</code>
<code>Escape</code>	9	65307	<code>ESC</code>
<code>Execute</code>	111	65378	<code>SYSREQ</code>
<code>F1</code>	67	65470	Function key F1
<code>F2</code>	68	65471	Function key F2
<code>Fi</code>	<code>66+i</code>	<code>65469+i</code>	Function key <code>Fi</code>
...			
<code>F12</code>	96	65481	Function key F12
<code>Home</code>	97	65360	<code>HOME</code>
<code>Insert</code>	106	65379	<code>INSERT</code>
<code>Left</code>	100	65361	←



Linefeed	54	106	Linefeed (CONTROL-J)
KP_0	90	65456	0 on the keypad
KP_1	91	65457	1 on the keypad
KP_ <i>i</i>	90+ <i>i</i>	65456+ <i>i</i>	Digit <i>i</i> on the keypad
...			
KP_9	81	65465	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	65454	. 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 an enter event; see <i>Event types</i> , above
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	↑

## 22.6 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 members of the `Event` object passed to the handler are described below. Some of these members are always set, but some members are set only for certain types of events.

<code>.char</code>	If the event was related to a <code>KeyPress</code> or <code>KeyRelease</code> for a key that produces a regular ASCII character, this string will be set to that character. (For special keys like <code>DELETE</code> , see the <code>.keysym</code> member, below.)
<code>.height</code>	If the event was a <code>Configure</code> , this member is set to the widget's new height in pixels.
<code>.keysym</code>	For <code>KeyPress</code> or <code>KeyRelease</code> events involving a special key, this member is set to the key's string name, e.g., "Prior" for the <code>PAGEUP</code> key. See the <i>Key names</i> table, above, for a complete list of <code>.keysym</code> names.
<code>.keysym_num</code>	For <code>KeyPress</code> or <code>KeyRelease</code> events, this is set to a numeric version of the <code>.keysym</code> 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 the <i>Key names</i> table, above.
<code>.num</code>	If the event was related to a mouse button, this member is set to the button number (1, 2, or 3).
<code>.time</code>	This member 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.
<code>.widget</code>	Always set to the widget that caused the event. For example, if the event was a mouse click that happened on a canvas, this member will be the actual <code>Canvas</code> widget.
<code>.width</code>	If the event was a <code>Configure</code> , this member is set to the widget's new width in pixels.
<code>.x</code>	The <i>x</i> coordinate of the mouse at the time of the event, relative to the upper left corner of the widget.
<code>.y</code>	The <i>y</i> coordinate of the mouse at the time of the event, relative to the upper left corner of the widget.
<code>.x_root</code>	The <i>x</i> coordinate of the mouse at the time of the event, relative to the upper left corner of the screen.
<code>.y_root</code>	The <i>y</i> 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 *Levels of binding*, 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.

## 22.7 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 the range [0, 9].

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 member of the Event object that gets passed in, but how does it find out that checkbutton's index in the self.cbList array?

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:

```

1def __createWidgets ( self ):
2    ...
3    self.cbList = [] # Create the checkbutton list
4    for i in range(10):
5        cb = Checkbutton ( self, ... )
6        self.cbList.append ( cb )
7        cb.grid( row=1, column=i )
8        def handler ( event, self=self, i=i ):
9            return self.__cbHandler ( event, i )
10       cb.bind ( "<Button-1>", handler )
11    ...
12def __cbHandler ( self, event, cbNumber ):

```

Lines <sup>8-9</sup> 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.

## 22.8 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 *Universal widget methods*, above, for more information about creating and managing virtual events.

---

Written by John W. Shipman (tcc-doc@nmt.edu). This version printed 2003-09-12. Copyright © 2001 by the New Mexico Institute of Mining and Technology.