

An Introduction To Tkinter

This is an incomplete draft version, last updated in **November 2005**.

You can find an earlier but more complete version in [the PythonWare library](#) (dead link).

Note: This is the source document used to generate the PythonWare version of **An Introduction To Tkinter**. Note that the links below point to documents that are being edited; in other words, they may be incomplete, broken, or otherwise messed up.

Part I. Introduction

Note: Some images and sample scripts are missing from this section. Some links may be broken.

[What's Tkinter?](#)
[Hello, Tkinter](#)
[Hello, Again](#)
[Tkinter Classes](#)
[Widget Configuration](#)
[Widget Styling](#)
[Events and Bindings](#)
[Application Windows](#)
[Standard Dialogs](#)
[Dialog Windows](#)

Part II. Class Reference

[The Button Widget](#)
[The Canvas Widget](#)
[The Checkbutton Widget](#)
[The Entry Widget](#)
[The Frame Widget](#)
[The Label Widget](#)
[The LabelFrame Widget](#)
[The Listbox Widget](#)
[The Menu Widget](#)
[The Menubutton Widget](#)
[The Message Widget](#)
[The OptionMenu Widget](#)
[The PanedWindow Widget](#)
[The Radiobutton Widget](#)
[The Scale Widget](#)
[The Scrollbar Widget](#)
[The Spinbox Widget](#)
[The Text Widget](#)
[The Toplevel Widget](#)
[Basic Widget Methods](#)
[Toplevel Window Methods](#)

[The Grid Geometry Manager](#)
[The Pack Geometry Manager](#)
[The Place Geometry Manager](#)

[The BitmapImage Class](#)
[The PhotoImage Class](#)

[Variable Wrappers \(BooleanVar, DoubleVar, IntVar, StringVar\)](#)

Part III. The BWidget Extension Library


Note: Incomplete.

[The BWidget Button Widget](#)
[The BWidget Entry Widget](#)
[The BWidget Label Widget](#)
[The BWidget Tree Widget](#)

Part IV. The Widget Construction Kit

Note: Incomplete.

[The Widget Construction Kit](#)

 rendered by a [django](#) application. hosted by [webfaction](#).

[back](#) [next](#)

What's Tkinter?

The **Tkinter** module (“Tk interface”) is the standard Python interface to the Tk GUI toolkit from [Scriptics](#) (formerly developed by Sun Labs).

Both Tk and Tkinter are available on most Unix platforms, as well as on Windows and Macintosh systems. Starting with the 8.0 release, Tk offers native look and feel on all platforms.

Tkinter consists of a number of modules. The Tk interface is provided by a binary extension module named **_tkinter**. This module contains the low-level interface to Tk, and should never be used directly by application programmers. It is usually a shared library (or DLL), but might in some cases be statically linked with the Python interpreter.

The public interface is provided through a number of Python modules. The most important interface module is the **Tkinter** module itself. To use Tkinter, all you need to do is to import the **Tkinter** module:

```
import Tkinter
```

Or, more often:

```
from Tkinter import *
```

The Tkinter module only exports widget classes and associated constants, so you can safely use the **from-in** form in most cases. If you prefer not to, but still want to save some typing, you can use **import-as**:

```
import Tkinter as Tk
```

[back](#) [next](#)

 rendered by a [django](#) application. hosted by [webfaction](#).

[back](#) [next](#)

Hello, Tkinter

But enough talk. Time to look at some code instead.

As you know, every serious tutorial should start with a “hello world”-type example. In this overview, we’ll show you not only one such example, but two.

First, let’s look at a pretty minimal version:

Our First Tkinter Program (File: hello1.py)

```
from Tkinter import *

root = Tk()

w = Label(root, text="Hello, world!")
w.pack()

root.mainloop()
```

Running the Example

To run the program, run the script as usual:

```
$ python hello1.py
```

The following window appears.

Running the program



To stop the program, just close the window.

Details

We start by importing the `Tkinter` module. It contains all classes, functions and other things needed to work with the Tk toolkit. In most cases, you can simply import everything from **Tkinter** into your module’s namespace:

```
from Tkinter import *
```

To initialize Tkinter, we have to create a `Tk` **root** widget. This is an ordinary window, with a title bar and other decoration provided by your window manager. You should only create one root widget for each program, and it must be created before any other widgets.

```
root = Tk()
```

Next, we create a **Label** widget as a child to the root window:

```
w = Label(root, text="Hello, world!")
w.pack()
```

A **Label** widget can display either text or an icon or other image. In this case, we use the **text** option to specify which text to display.

Next, we call the **pack** method on this widget. This tells it to size itself to fit the given text, and make itself visible. However, the window won’t appear until we’ve entered the Tkinter event loop:

```
root.mainloop()
```

The program will stay in the event loop until we close the window. The event loop doesn't only handle events from the user (such as mouse clicks and key presses) or the windowing system (such as redraw events and window configuration messages), it also handle operations queued by Tkinter itself. Among these operations are geometry management (queued by the **pack** method) and display updates. This also means that the application window will not appear before you enter the main loop.

[back](#) [next](#)



rendered by a [django](#) application. hosted by [webfaction](#).

[back](#) [next](#)

Hello, Again

When you write larger programs, it is usually a good idea to wrap your code up in one or more classes. The following example is adapted from the “hello world” program in Matt Conway’s [A Tkinter Life Preserver](#) (dead link).

Our Second Tkinter Program

```
from Tkinter import *

class App:

    def __init__(self, master):

        frame = Frame(master)
        frame.pack()

        self.button = Button(
            frame, text="QUIT", fg="red", command=frame.quit
        )
        self.button.pack(side=LEFT)

        self.hi_there = Button(frame, text="Hello", command=self.say_hi)
        self.hi_there.pack(side=LEFT)

    def say_hi(self):
        print "hi there, everyone!"

root = Tk()

app = App(root)

root.mainloop()
root.destroy() # optional; see description below
```

Running the Example

When you run this example, the following window appears.

Running the sample program (using Tk 8.0 on a Windows 95 box)



If you click the right button, the text “**hi there, everyone!**” is printed to the console. If you click the left button, the program stops.

Note: Some Python development environments have problems running Tkinter examples like this one. The problem is usually that the environment uses Tkinter itself, and the **mainloop** call and the **quit** calls interact with the environment’s expectations. Other environments may misbehave if you leave out the explicit **destroy** call. If the example doesn’t behave as expected, check for Tkinter-specific documentation for your development environment.

Details

This sample application is written as a class. The constructor (the **__init__** method) is called with a parent widget (the **master**), to which it adds a number of child widgets. The constructor starts by creating a **Frame** widget. A frame is a simple container, and

is in this case only used to hold the other two widgets.

```
class App:
    def __init__(self, master):
        frame = Frame(master)
        frame.pack()
```

The frame instance is stored in a local variable called **frame**. After creating the widget, we immediately call the **pack** method to make the frame visible.

We then create two **Button** widgets, as children to the frame.

```
self.button = Button(frame, text="QUIT", fg="red", command=frame.quit)
self.button.pack(side=LEFT)

self.hi_there = Button(frame, text="Hello", command=self.say_hi)
self.hi_there.pack(side=LEFT)
```

This time, we pass a number of **options** to the constructor, as keyword arguments. The first button is labelled “QUIT”, and is made red (**fg** is short for **foreground**). The second is labelled “Hello”. Both buttons also take a **command** option. This option specifies a function, or (as in this case) a bound method, which will be called when the button is clicked.

The button instances are stored in instance attributes. They are both packed, but this time with the **side=LEFT** argument. This means that they will be placed as far left as possible in the frame; the first button is placed at the frame’s left edge, and the second is placed just to the right of the first one (at the left edge of the *remaining space* in the frame, that is). By default, widgets are packed relative to their parent (which is **master** for the frame widget, and the frame itself for the buttons). If the side is not given, it defaults to **TOP**.

The “hello” button callback is given next. It simply prints a message to the console everytime the button is pressed:

```
def say_hi(self):
    print "hi there, everyone!"
```

Finally, we provide some script level code that creates a **Tk** root widget, and one instance of the **App** class using the root widget as its parent:

```
root = Tk()

app = App(root)

root.mainloop()
root.destroy()
```

The **mainloop** call enters the Tk event loop, in which the application will stay until the **quit** method is called (just click the QUIT button), or the window is closed.

The **destroy** call is only required if you run this example under certain development environments; it explicitly destroys the main window when the event loop is terminated. Some development environments won’t terminate the Python process unless this is done.

More on widget references

In the second example, the frame widget is stored in a local variable named **frame**, while the button widgets are stored in two instance attributes. Isn’t there a serious problem hidden in here: what happens when the **__init__** function returns and the **frame** variable goes out of scope?

Just relax; there’s actually no need to keep a reference to the widget instance. Tkinter automatically maintains a widget tree (via the **master** and **children** attributes of each widget instance), so a widget won’t disappear when the application’s last reference goes away; it must be explicitly destroyed before this happens (using the **destroy** method).

But if you wish to do something with the widget after it has been created, you better keep a reference to the widget instance yourself.

Note that if you don't need to keep a reference to a widget, it might be tempting to create and pack it on a single line:

```
Button(frame, text="Hello", command=self.hello).pack(side=LEFT)
```

Don't store the result from this operation; you'll only get disappointed when you try to use that value (the **pack** method returns **None**). To be on the safe side, it might be better to always separate construction from packing:

```
w = Button(frame, text="Hello", command=self.hello)
w.pack(side=LEFT)
```

More on widget names

Another source of confusion, especially for those who have some experience of programming Tk using Tcl, is Tkinter's notion of the *widget name*. In Tcl, you must explicitly name each widget. For example, the following Tcl command creates a **Button** named "ok", as a child to a widget named "dialog" (which in turn is a child of the root window, ".").

```
button .dialog.ok
```

The corresponding Tkinter call would look like:

```
ok = Button(dialog)
```

However, in the Tkinter case, **ok** and **dialog** are references to widget instances, not the actual names of the widgets. Since Tk itself needs the names, Tkinter automatically assigns a unique name to each new widget. In the above case, the dialog name is probably something like ".1428748," and the button could be named ".1428748.1432920". If you wish to get the full name of a Tkinter widget, simply use the **str** function on the widget instance:

```
>>> print str(ok)
.1428748.1432920
```

(if you print something, Python automatically uses the **str** function to find out what to print. But obviously, an operation like "name = ok" won't do the that, so make sure always to explicitly use **str** if you need the name).


If you really need to specify the name of a widget, you can use the **name** option when you create the widget. One (and most likely the only) reason for this is if you need to interface with code written in Tcl.

In the following example, the resulting widget is named "**.dialog.ok**" (or, if you forgot to name the dialog, something like "**.1428748.ok**"):

```
ok = Button(dialog, name="ok")
```

To avoid conflicts with Tkinter's naming scheme, don't use names which only contain digits. Also note that **name** is a "creation only" option; you cannot change the name once you've created the widget.

[back next](#)

 rendered by a [django](#) application. hosted by [webfaction](#).

[back](#) [next](#)

Tkinter Classes

Widget classes

Tkinter supports 15 core widgets:

Button

A simple button, used to execute a command or other operation.

Canvas

Structured graphics. This widget can be used to draw graphs and plots, create graphics editors, and to implement custom widgets.

Checkbutton

Represents a variable that can have two distinct values. Clicking the button toggles between the values.

Entry

A text entry field.

Frame

A container widget. The frame can have a border and a background, and is used to group other widgets when creating an application or dialog layout.

Label

Displays a text or an image.

Listbox

Displays a list of alternatives. The listbox can be configured to get radiobutton or checklist behavior.

Menu

A menu pane. Used to implement pulldown and popup menus.

Menubutton

A menubutton. Used to implement pulldown menus.

Message

Display a text. Similar to the label widget, but can automatically wrap text to a given width or aspect ratio.

Radiobutton

Represents one value of a variable that can have one of many values. Clicking the button sets the variable to that value, and clears all other radiobuttons associated with the same variable.

Scale

Allows you to set a numerical value by dragging a “slider”.

Scrollbar

Standard scrollbars for use with canvas, entry, listbox, and text widgets.

Text

Formatted text display. Allows you to display and edit text with various styles and attributes. Also supports embedded images and windows.

Toplevel

A container widget displayed as a separate, top-level window.

In Python 2.3 (Tk 8.4), the following widgets were added:

LabelFrame

A variant of the Frame widget that can draw both a border and a title.

PanedWindow

A container widget that organizes child widgets in resizable panes.

Spinbox

A variant of the Entry widget for selecting values from a range or an ordered set.

Also note that there's no widget class hierarchy in Tkinter; all widget classes are siblings in the inheritance tree.

All these widgets provide the **Misc** and geometry management methods, the configuration management methods, and additional methods defined by the widget itself. In addition, the **Toplevel** class also provides the window manager interface. This means that a typical widget class provides some 150 methods.

Mixins

The Tkinter module provides classes corresponding to the various widget types in Tk, and a number of mixin and other helper classes (a *mixin* is a class designed to be combined with other classes using multiple inheritance). When you use Tkinter, you should never access the mixin classes directly.

Implementation mixins

The [Misc](#) class is used as a mixin by the root window and widget classes. It provides a large number of Tk and window related services, which are thus available for all Tkinter core widgets. This is done by *delegation*; the widget simply forwards the request to the appropriate internal object.

The [Wm](#) class is used as a mixin by the root window and [Toplevel](#) widget classes. It provides window manager services, also by delegation.

Using delegation like this simplifies your application code: once you have a widget, you can access all parts of Tkinter using methods on the widget instance.

Geometry mixins

The [Grid](#), [Pack](#), and [Place](#) classes are used as mixins by the widget classes. They provide access to the various geometry managers, also via delegation.

[Grid](#)

The grid geometry manager allows you to create table-like layouts, by organizing the widgets in a 2-dimensional grid. To use this geometry manager, use the **grid** method.

[Pack](#)

The pack geometry manager lets you create a layout by “packing” the widgets into a parent widget, by treating them as rectangular blocks placed in a frame. To use this geometry manager for a widget, use the **pack** method on that widget to set things up.

[Place](#)

The place geometry manager lets you explicitly place a widget in a given position. To use this geometry manager, use the **place** method.

Widget configuration management

The **Widget** class mixes the **Misc** class with the geometry mixins, and adds configuration management through the **cget** and **configure** methods, as well as through a partial dictionary interface. The latter can be used to set and query individual options, and is explained in further detail in the next chapter.

[back](#) [next](#)

 rendered by a [django](#) application. hosted by [webfaction](#).

[back](#) [next](#)

Widget Configuration

To control the appearance of a widget, you usually use *options* rather than method calls. Typical options include text and color, size, command callbacks, etc. To deal with options, all core widgets implement the same configuration interface:

Configuration Interface

widgetclass(master, option=value, ...) => widget

(where *widgetclass* is one of the widget classes mentioned earlier)

Create an instance of this widget class, as a child to the given master, and using the given options. All options have default values, so in the simplest case, you only have to specify the master. You can even leave that out if you really want; Tkinter then uses the most recently created root window as master. Note that the **name** option can only be set when the widget is created.

cget("option") => string

Return the current value of an option. Both the option name, and the returned value, are strings. To get the **name** option, use **str(widget)** instead.

config(option=value, ...)

configure(option=value, ...)

Set one or more options (given as keyword arguments).

Note that some options have names that are reserved words in Python (**class**, **from**, ...). To use these as keyword arguments, simply append an underscore to the option name (**class_**, **from_**, ...). Note that you cannot set the **name** option using this method; it can only be set when the widget is created.

For convenience, the widgets also implement a partial dictionary interface. The **__setitem__** method maps to **configure**, while **__getitem__** maps to **cget**. As a result, you can use the following syntax to set and query options:

```
value = widget["option"]
widget["option"] = value
```

Note that each assignment results in one call to Tk. If you wish to change multiple options, it is usually a better idea to change them with a single call to **config** or **configure** (personally, I prefer to always change options in that fashion).

The following dictionary method also works for widgets:

keys() => list

Return a list of all options that can be set for this widget. The **name** option is not included in this list (it cannot be queried or modified through the dictionary interface anyway, so this doesn't really matter).

Backwards Compatibility

Keyword arguments were introduced in Python 1.3. Before that, options were passed to the widget constructors and **configure** methods using ordinary Python dictionaries. The source code could then look something like this:

```
self.button = Button(frame, {"text": "QUIT", "fg": "red", "command": frame.quit})
```

```
self.button.pack({"side": LEFT})
```

The keyword argument syntax is of course much more elegant, and less error prone. However, for compatibility with existing code, Tkinter still supports the older syntax. You shouldn't use this syntax in new programs, even if it might be tempting in some cases. For example, if you create a custom widget which needs to pass configuration options along to its parent class, you may come up with something like:


```
def __init__(self, master, **kw):  
    Canvas.__init__(self, master, kw) # kw is a dictionary
```

This works just fine with the current version of Tkinter, but it may not work with future versions. A more general approach is to use the **apply** function:

```
def __init__(self, master, **kw):  
    apply(Canvas.__init__, (self, master), kw)
```

The **apply** function takes a function (an unbound method, in this case), a tuple with arguments (which must include **self** since we're calling an unbound method), and optionally, a dictionary which provides the keyword arguments.

[back](#) [next](#)

 rendered by a [django](#) application. hosted by [webfaction](#).

[back](#) [next](#)

Widget Styling

All Tkinter standard widgets provide a basic set of “styling” options, which allow you to modify things like colors, fonts, and other visual aspects of each widget.

Colors

Most widgets allow you to specify the widget and text colors, using the **background** and **foreground** options. To specify a color, you can either use a color name, or explicitly specify the red, green, and blue (RGB) color components.

Color Names

Tkinter includes a color database which maps color names to the corresponding RGB values. This database includes common names like **Red**, **Green**, **Blue**, **Yellow**, and **LightBlue**, but also more exotic things like **Moccasin**, **PeachPuff**, etc.

On an X window system, the color names are defined by the X server. You might be able to locate a file named **xrgb.txt** which contains a list of color names and the corresponding RGB values. On Windows and Macintosh systems, the color name table is built into Tk.

Under Windows, you can also use the Windows system colors (these can be changed by the user via the control panel):

**SystemActiveBorder, SystemActiveCaption,
SystemAppWorkspace, SystemBackground,
SystemButtonFace, SystemButtonHighlight,
SystemButtonShadow, SystemButtonText, SystemCaptionText,
SystemDisabledText, SystemHighlight, SystemHighlightText,
SystemInactiveBorder, SystemInactiveCaption,
SystemInactiveCaptionText, SystemMenu, SystemMenuText,
SystemScrollbar, SystemWindow, SystemWindowFrame,
SystemWindowText.**

On the Macintosh, the following system colors are available:

**SystemButtonFace, SystemButtonFrame, SystemButtonText,
SystemHighlight, SystemHighlightText, SystemMenu,
SystemMenuActive, SystemMenuActiveText,
SystemMenuDisabled, SystemMenuText, SystemWindowBody.**

Color names are case insensitive. Many (but not all) color names are also available with or without spaces between the words. For example, “lightblue”, “light blue”, and “Light Blue” all specify the same color.

RGB Specifications

If you need to explicitly specify a color, you can use a string with the following format:

`#RRGGBB`

RR, GG, BB are hexadecimal representations of the red, green and blue values, respectively. The following sample shows how you can convert a color 3-tuple to a Tk color specification:

```
tk_rgb = "#%02x%02x%02x" % (128, 192, 200)
```

Tk also supports the forms “**#RGB**” and “**#RRRRGGGGBBBB**” to specify each value with 16 and 65536 levels, respectively.

You can use the **winfo_rgb** widget method to translate a color string (either a name or an RGB specification) to a 3-tuple:

```
rgb = widget.winfo_rgb("red")
red, green, blue = rgb[0]/256, rgb[1]/256, rgb[2]/256
```

Note that **winfo_rgb** returns 16-bit RGB values, ranging from 0 to 65535. To map them into the more common 0-255 range, you must divide each value by 256 (or shift them 8 bits to the right).

Fonts

Widgets that allow you to display text in one way or another also allows you to specify which font to use. All widgets provide reasonable default values, and you seldom have to specify the font for simpler elements like labels and buttons.

Fonts are usually specified using the **font** widget option. Tkinter supports a number of different font descriptor types:

- Font descriptors
- User-defined font names
- System fonts
- X font descriptors

With Tk versions before 8.0, only *X font descriptors* are supported (see below).

Font descriptors

Starting with Tk 8.0, Tkinter supports platform independent font descriptors. You can specify a font as tuple containing a family name, a height in points, and optionally a string with one or more styles. Examples:

```
("Times", 10, "bold")
("Helvetica", 10, "bold italic")
("Symbol", 8)
```

To get the default size and style, you can give the font name as a single string. If the family name doesn't include spaces, you can also add size and styles to the string itself:

```
"Times 10 bold"
"Helvetica 10 bold italic"
"Symbol 8"
```

Here are some families available on most Windows platforms:

Arial (corresponds to Helvetica), **Courier New** (Courier), **Comic Sans MS**, **Fixedsys**, **MS Sans Serif**, **MS Serif**, **Symbol**, **System**, **Times New Roman** (Times), and **Verdana**:

arial 14 points: I'd like to have an argu

courier new 12 points: What? Pri

comic sans ms 8 points: Pack my box with fiven dozen jugs of

fixedsys 9 points: Here you see some Eni

ms sans serif 11 points: Pack my box with fiven dozen

ms serif 16 points: The quick brown fox ju

σμβολ 12 ποντσ: Φιγυρε τηισ ουτ, ωονδερ βοψ

system 10 points: Hello, Harry. Now there's the s

times new roman 16 points: That turni

verdana 10 points: The quick brown fox jumps c

Note that if the family name contains spaces, you must use the tuple syntax described above.

The available styles are **normal**, **bold**, **roman**, **italic**, **underline**, and **overstrike**.

Tk 8.0 automatically maps **Courier**, **Helvetica**, and **Times** to their corresponding native family names on all platforms. In addition, a font specification can never fail under Tk 8.0; if Tk cannot come up with an exact match, it tries to find a similar font. If that fails, Tk falls back to a platform-specific default font. Tk's idea of what is "similar enough" probably doesn't correspond to your own view, so you shouldn't rely too much on this feature.

Tk 4.2 under Windows supports this kind of font descriptors as well. There are several restrictions, including that the family name must exist on the platform, and not all the above style names exist (or rather, some of them have different names).

Font names

In addition, Tk 8.0 allows you to create named fonts and use their names when specifying fonts to the widgets.

The **tkFont** module provides a [Font](#) class which allows you to create font instances. You can use such an instance everywhere Tkinter accepts a font specifier. You can also use a font instance to get font metrics, including the size occupied by a given string written in that font.

```
tkFont.Font(family="Times", size=10, weight=tkFont.BOLD)
tkFont.Font(family="Helvetica", size=10, weight=tkFont.BOLD,
            slant=tkFont.ITALIC)
tkFont.Font(family="Symbol", size=8)
```

If you modify a named font (using the **config** method), the changes are automatically propagated to all widgets using the font.

The **Font** constructor supports the following style options (note that the constants are defined in the **tkFont** module):

family

Font family.

size

Font size in points. To give the size in pixels, use a negative value.

weight

Font thickness. Use one of **NORMAL** or **BOLD**. Default is **NORMAL**.

slant

Font slant. Use one of **NORMAL** or **ITALIC**. Default is **NORMAL**.

underline

Font underlining. If 1 (true), the font is underlined. Default is 0 (false).

overstrike

Font strikeout. If 1 (true), a line is drawn over text written with this font. Default is 0 (false).

System fonts

Tk also supports system specific font names. Under X, these are usually font aliases like **fixed**, **6x10**, etc.

Under Windows, these include **ansi**, **ansifixed**, **device**, **oemfixed**, **system**, and **systemfixed**:

```
ansi: I didn't know ants had six legs, Marcus
ansifixed: Another merciless sweep
device: We like dressing up, ye
oemfixed: One day Ricky the magic p
system: Pretty strong meat there from Sam
systemfixed: Simon Zinc Trumpet Har
```

On the Macintosh, the system font names are **application** and **system**.

Note that the system fonts are full font names, not family names, and they cannot be combined with size or style attributes. For portability reasons, avoid using these names wherever possible.

X Font Descriptors

X Font Descriptors are strings having the following format (the asterisks represent fields that are usually not relevant. For details, see the Tk documentation, or an X manual):

```
--family-weight-slant-*--size-*--*--charset
```

The font *family* is typically something like **Times**, **Helvetica**, **Courier** or **Symbol**.

The *weight* is either **Bold** or **Normal**. Slant is either **R** for “roman” (normal), **I** for italic, or **O** for oblique (in practice, this is just another word for italic).

Size is the height of the font in decipoints (that is, points multiplied by 10). There are usually 72 points per inch, but some low-resolution displays may use larger “logical” points to make sure that small fonts are still legible. The *character set*, finally, is usually **ISO8859-1** (ISO Latin 1), but may have other values for some fonts.

The following descriptor requests a 12-point boldface Times font, using the ISO Latin 1 character set:

```
--Times-Bold-R-*--120-*--*--ISO8859-1
```

If you don’t care about the character set, or use a font like **Symbol** which has a special character set, you can use a single asterisk as the last component:

```
--Symbol-*--*--*--80--*
```

A typical X server supports at least **Times**, **Helvetica**, **Courier**, and a few more fonts, in sizes like 8, 10, 12, 14, 18, and 24 points, and in normal, bold, and italic (**Times**) or oblique (**Helvetica**, **Courier**) variants. Most servers also support freely scaleable fonts. You can use programs like **xlsfonts** and **xfontsel** to check which fonts you have access to on a given server.

This kind of font descriptors can also be used on Windows and Macintosh. Note that if you use Tk 4.2, you should keep in mind that the font family must be one supported by Windows (see above).

Text Formatting

While text labels and buttons usually contain a single line of text, Tkinter also supports multiple lines. To split the text across lines, simply insert newline characters (**\n**) where necessary.

By default, the lines are centered. You can change this by setting the **justify** option to **LEFT** or **RIGHT**. The default value is **CENTER**.

You can also use the **wraplength** option to set a maximum width, and let the widget wrap the text over multiple lines all by itself. Tkinter attempts to wrap on whitespace,

but if the widget is too narrow, it may break individual words across lines.

Borders

All Tkinter widgets have a border (though it's not visible by default for some widgets). The border consists of an optional 3D relief, and a focus highlight region.

Relief

The relief settings control how to draw the widget border:

borderwidth (or **bd**)

This is the width of the border, in pixels. Most widgets have a default borderwidth of one or two pixels. There's hardly any reason to make the border wider than that.

relief

This option controls how to draw the 3D border. It can be set to one of **SUNKEN**, **RAISED**, **GROOVE**, **RIDGE**, and **FLAT**.

Focus Highlights

The highlight settings control how to indicate that the widget (or one of its children) has keyboard focus. In most cases, the highlight region is a border outside the relief. The following options control how this extra border is drawn:

highlightcolor

This option is used to draw the highlight region when the widget has keyboard focus. It's usually black, or some other distinct contrast color.

highlightbackground

This option is used to draw the highlight region when the widget doesn't have focus. It's usually same as the widget background.

highlightthickness

This option is the width of the highlight region, in pixels. It is usually one or two pixels for widgets that can take keyboard focus.

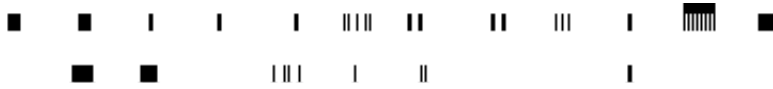
Cursors

cursor


This option controls which mouse cursor to use when the mouse is moved over the widget.

If this option isn't set, the widget uses the same mouse pointer as its parent.

Note that some widgets, including the [Text](#) and [Entry](#) widgets, set this option by default.



[back](#) [next](#)

 rendered by a [django](#) application. hosted by [webfaction](#).

[back](#) [next](#)

Events and Bindings

As was mentioned earlier, a Tkinter application spends most of its time inside an event loop (entered via the **mainloop** method). Events can come from various sources, including key presses and mouse operations by the user, and redraw events from the window manager (indirectly caused by the user, in many cases).

Tkinter provides a powerful mechanism to let you deal with events yourself. For each widget, you can **bind** Python functions and methods to events.

```
widget.bind(event, handler)
```

If an event matching the *event* description occurs in the widget, the given *handler* is called with an object describing the event.

Here's a simple example:

Capturing clicks in a window

```
from Tkinter import *

root = Tk()

def callback(event):
    print "clicked at", event.x, event.y

frame = Frame(root, width=100, height=100)
frame.bind("<Button-1>", callback)
frame.pack()

root.mainloop()
```

In this example, we use the **bind** method of the frame widget to bind a callback function to an event called **<Button-1>**. Run this program and click in the window that appears. Each time you click, a message like “**clicked at 44 63**” is printed to the console window.

Keyboard events are sent to the widget that currently owns the keyboard focus. You can use the **focus_set** method to move focus to a widget:

Capturing keyboard events

```
from Tkinter import *

root = Tk()

def key(event):
    print "pressed", repr(event.char)

def callback(event):
    frame.focus_set()
    print "clicked at", event.x, event.y

frame = Frame(root, width=100, height=100)
frame.bind("<Key>", key)
frame.bind("<Button-1>", callback)
frame.pack()

root.mainloop()
```

If you run this script, you'll find that you have to click in the frame before it starts receiving any keyboard events.

Events

Events are given as strings, using a special event syntax:

```
<modifier-type-detail>
```

The *type* field is the most important part of an event specifier. It specifies the kind of event that we wish to bind, and can be user actions like **Button**, and **Key**, or window manager events like **Enter**, **Configure**, and others. The modifier and detail fields are used to give additional information, and can in many cases be left out. There are also various ways to simplify the event string; for example, to match a keyboard key, you can leave out the angle brackets and just use the key as is. Unless it is a space or an angle bracket, of course.

Instead of spending a few pages on discussing all the syntactic shortcuts, let's take a look on the most common event formats:

Event Formats

<Button-1>

A mouse button is pressed over the widget. Button 1 is the leftmost button, button 2 is the middle button (where available), and button 3 the rightmost button. When you press down a mouse button over a widget, Tkinter will automatically “grab” the mouse pointer, and subsequent mouse events (e.g. Motion and Release events) will then be sent to the current widget as long as the mouse button is held down, even if the mouse is moved outside the current widget. The current position of the mouse pointer (relative to the widget) is provided in the **x** and **y** members of the event object passed to the callback.

You can use **ButtonPress** instead of **Button**, or even leave it out completely: **<Button-1>**, **<ButtonPress-1>**, and **<1>** are all synonyms. For clarity, I prefer the **<Button-1>** syntax.

<B1-Motion>

The mouse is moved, with mouse button 1 being held down (use B2 for the middle button, B3 for the right button). The current position of the mouse pointer is provided in the **x** and **y** members of the event object passed to the callback.

<ButtonRelease-1>

Button 1 was released. The current position of the mouse pointer is provided in the **x** and **y** members of the event object passed to the callback.

<Double-Button-1>

Button 1 was double clicked. You can use **Double** or **Triple** as prefixes. Note that if you bind to both a single click (**<Button-1>**) and a double click, both bindings will be called.

<Enter>

The mouse pointer entered the widget (this event doesn't mean that the user pressed the **Enter** key!).

<Leave>

The mouse pointer left the widget.

<FocusIn>

Keyboard focus was moved to this widget, or to a child of this widget.

<FocusOut>

Keyboard focus was moved from this widget to another widget.

<Return>

The user pressed the Enter key. You can bind to virtually all keys on the keyboard. For an ordinary 102-key PC-style keyboard, the special keys are **Cancel** (the Break key), **BackSpace**, **Tab**, **Return** (the Enter key), **Shift_L** (any Shift key), **Control_L** (any Control key), **Alt_L** (any Alt key), **Pause**, **Caps_Lock**, **Escape**, **Prior** (Page Up), **Next** (Page Down), **End**, **Home**, **Left**, **Up**, **Right**, **Down**, **Print**, **Insert**, **Delete**, **F1**, **F2**, **F3**, **F4**, **F5**, **F6**, **F7**, **F8**, **F9**, **F10**, **F11**, **F12**, **Num_Lock**, and **Scroll_Lock**.

<Key>

The user pressed any key. The key is provided in the **char** member of the event object passed to the callback (this is an empty string for special keys).

a

The user typed an “a”. Most printable characters can be used as is. The exceptions are space (**<space>**) and less than (**<less>**). Note that **1** is a keyboard binding, while **<1>** is a button binding.

<Shift-Up>

The user pressed the Up arrow, while holding the Shift key pressed. You can use prefixes like **Alt**, **Shift**, and **Control**.

<Configure>

The widget changed size (or location, on some platforms). The new size is provided in the **width** and **height** attributes of the event object passed to the callback.

The Event Object

The event object is a standard Python object instance, with a number of attributes describing the event.

Event Attributes

widget

The widget which generated this event. This is a valid Tkinter widget instance, not a name. This attribute is set for all events.

x, y

The current mouse position, in pixels.

x_root, y_root

The current mouse position relative to the upper left corner of the screen, in pixels.

char

The character code (keyboard events only), as a string.

keysym

The key symbol (keyboard events only).

keycode

The key code (keyboard events only).

num

The button number (mouse button events only).

width, height

The new size of the widget, in pixels (Configure events only).

type

The event type.

For portability reasons, you should stick to **char**, **height**, **width**, **x**, **y**, **x_root**, **y_root**, and **widgit**. Unless you know exactly what you're doing, of course...

Instance and Class Bindings

The **bind** method we used in the above example creates an instance binding. This means that the binding applies to a single widget only; if you create new frames, they will not inherit the bindings.

But Tkinter also allows you to create bindings on the class and application level; in fact, you can create bindings on four different levels:

- the widget instance, using **bind**.
- the widget's toplevel window (Toplevel or **root**), also using **bind**.
- the widget class, using **bind_class** (this is used by Tkinter to provide standard bindings).
- the whole application, using **bind_all**.

For example, you can use **bind_all** to create a binding for the F1 key, so you can provide help everywhere in the application. But what happens if you create multiple bindings for the same key, or provide overlapping bindings?

First, on each of these four levels, Tkinter chooses the “closest match” of the available bindings. For example, if you create instance bindings for the **<Key>** and **<Return>** events, only the second binding will be called if you press the **Enter** key.

However, if you add a **<Return>** binding to the toplevel widget, *both* bindings will be called. Tkinter first calls the best binding on the instance level, then the best binding on the toplevel window level, then the best binding on the class level (which is often a standard binding), and finally the best available binding on the application level. So in an extreme case, a single event may call four event handlers.

A common cause of confusion is when you try to use bindings to override the default behavior of a standard widget. For example, assume you wish to disable the Enter key in the text widget, so that the users cannot insert newlines into the text. Maybe the following will do the trick?

```
def ignore(event):
    pass
text.bind("<Return>", ignore)
```

or, if you prefer one-liners:

```
text.bind("<Return>", lambda e: None)
```

(the **lambda** function used here takes one argument, and returns **None**)

Unfortunately, the newline is still inserted, since the above binding applies to the instance level only, and the standard behavior is provided by a class level bindings.

You could use the **bind_class** method to modify the bindings on the class level, but that would change the behavior of *all* text widgets in the application. An easier solution is to prevent Tkinter from propagating the event to other handlers; just return the string **“break”** from your event handler:

```
def ignore(event):
    return "break"
text.bind("<Return>", ignore)
```

or

```
text.bind("<Return>", lambda e: "break")
```

By the way, if you really want to change the behavior of all text widgets in your application, here's how to use the **bind_class** method:

```
top.bind_class("Text", "<Return>", lambda e: None)
```

But there are a lot of reasons why you shouldn't do this. For example, it messes things up completely the day you wish to extend your application with some cool little UI component you downloaded from the net. Better use your own **Text** widget specialization, and keep Tkinter's default bindings intact:

```
class MyText(Text):
    def __init__(self, master, **kw):
        apply(Text.__init__, (self, master), kw)
        self.bind("<Return>", lambda e: "break")
```

Protocols

In addition to event bindings, Tkinter also supports a mechanism called *protocol handlers*. Here, the term protocol refers to the interaction between the application and the window manager. The most commonly used protocol is called **WM_DELETE_WINDOW**, and is used to define what happens when the user explicitly closes a window using the window manager.

You can use the **protocol** method to install a handler for this protocol (the widget must be a root or **Toplevel** widget):

```
widget.protocol("WM_DELETE_WINDOW", handler)
```

Once you have installed your own handler, Tkinter will no longer automatically close the window. Instead, you could for example display a message box asking the user if the current data should be saved, or in some cases, simply ignore the request. To close the window from this handler, simply call the **destroy** method of the window:

Capturing destroy events

```
from Tkinter import *
import tkMessageBox

def callback():
    if tkMessageBox.askokcancel("Quit", "Do you really wish to quit?"):
        root.destroy()

root = Tk()
root.protocol("WM_DELETE_WINDOW", callback)

root.mainloop()
```

Note that even you don't register an handler for **WM_DELETE_WINDOW** on a toplevel window, the window itself will be destroyed as usual (in a controlled fashion, unlike X). However, as of Python 1.5.2, Tkinter will not destroy the corresponding widget instance hierarchy, so it is a good idea to always register a handler yourself:

```
top = Toplevel(...)

# make sure widget instances are deleted
top.protocol("WM_DELETE_WINDOW", top.destroy)
```

Future versions of Tkinter will most likely do this by default.

Other Protocols

Window manager protocols were originally part of the X window system (they are defined in a document titled *Inter-Client Communication Conventions Manual*, or ICCCM). On that platform, you can install handlers for other protocols as well, like **WM_TAKE_FOCUS** and **WM_SAVE_YOURSELF**. See the ICCCM documentation for details.

[back](#) [next](#)



rendered by a [django](#) application. hosted by [webfaction](#).

[back](#) [next](#)

Application Windows

Base Windows

In the simple examples we've used this far, there's only one window on the screen; the root window. This is automatically created when you call the **Tk** constructor, and is of course very convenient for simple applications:

```
from Tkinter import *

root = Tk()

# create window contents as children to root...

root.mainloop()
```

If you need to create additional windows, you can use the **Toplevel** widget. It simply creates a new window on the screen, a window that looks and behaves pretty much like the original root window:

```
from Tkinter import *

root = Tk()

# create root window contents...

top = Toplevel()

# create top window contents...

root.mainloop()
```

There's no need to use **pack** to display the **Toplevel**, since it is automatically displayed by the window manager (in fact, you'll get an error message if you try to use **pack** or any other geometry manager with a **Toplevel** widget).

Menus

Tkinter provides a special widget type for menus. To create a menu, you create an instance of the **Menu** class, and use **add** methods to add entries to it:

- **add_command(label=string, command=callback)** adds an ordinary menu entry.
- **add_separator()** adds an separator line. This is used to group menu entries.
- **add_cascade(label=string, menu=menu instance)** adds a submenu (another **Menu** instance). This is either a pull-down menu or a fold-out menu, depending on the parent.

Here's an example:

Creating a small menu

```
from Tkinter import *

def callback():
    print "called the callback!"

root = Tk()

# create a menu
menu = Menu(root)
root.config(menu=menu)

filemenu = Menu(menu)
menu.add_cascade(label="File", menu=filemenu)
```

```

filemenu.add_command(label="New", command=callback)
filemenu.add_command(label="Open...", command=callback)
filemenu.add_separator()
filemenu.add_command(label="Exit", command=callback)

helpmenu = Menu(menu)
menu.add_cascade(label="Help", menu=helpmenu)
helpmenu.add_command(label="About...", command=callback)

mainloop()

```

In this example, we start out by creating a **Menu** instance, and we then use the **config** method to attach it to the root window. The contents of that menu will be used to create a menubar at the top of the root window. You don't have to pack the menu, since it is automatically displayed by Tkinter.

Next, we create a new **Menu** instance, using the menubar as the widget parent, and the **add_cascade** method to make it a pulldown menu. We then call **add_command** to add commands to the menu (note that all commands in this example use the same callback), and **add_separator** to add a line between the file commands and the exit command.

Finally, we create a small help menu in the same fashion.

Toolbars

Many applications place a toolbar just under the menubar, which typically contains a number of buttons for common functions like open file, print, undo, etc.

In the following example, we use a **Frame** widget as the toolbar, and pack a number of ordinary buttons into it.

Creating a simple toolbar

```

from Tkinter import *

root = Tk()

def callback():
    print "called the callback!"

# create a toolbar
toolbar = Frame(root)

b = Button(toolbar, text="new", width=6, command=callback)
b.pack(side=LEFT, padx=2, pady=2)

b = Button(toolbar, text="open", width=6, command=callback)
b.pack(side=LEFT, padx=2, pady=2)

toolbar.pack(side=TOP, fill=X)

mainloop()

```

The buttons are packed against the left side, and the toolbar itself is packed against the topmost side, with the **fill** option set to **X**. As a result, the widget is resized if necessary, to cover the full width of the parent widget.

Also note that I've used text labels rather than icons, to keep things simple. To display an icon, you can use the **PhotoImage** constructor to load a small image from disk, and use the **image** option to display it.

Status Bars

Finally, most applications sport a status bar at the bottom of each application window. Implementing a status bar with Tkinter is trivial: you can simply use a suitably configured **Label** widget, and reconfigure the **text** option now and then. Here's one

way to do it:

```
status = Label(master, text="", bd=1, relief=SUNKEN, anchor=W)
status.pack(side=BOTTOM, fill=X)
```

If you wish to be fancy, you can use the following class instead. It wraps a label widget in a convenience class, and provides **set** and **clear** methods to modify the contents.

A Status Bar Class (File: tkSimpleStatusBar.py)

```
class StatusBar(Frame):
    def __init__(self, master):
        Frame.__init__(self, master)
        self.label = Label(self, bd=1, relief=SUNKEN, anchor=W)
        self.label.pack(fill=X)

    def set(self, format, *args):
        self.label.config(text=format % args)
        self.label.update_idletasks()

    def clear(self):
        self.label.config(text="")
        self.label.update_idletasks()
```

The **set** method works like C's **printf** function; it takes a format string, possibly followed by a set of arguments (a drawback is that if you wish to print an arbitrary string, you must do that as **set("%s", string)**). Also note that this method calls the **update_idletasks** method, to make sure pending draw operations (like the status bar update) are carried out immediately.

But the real trick here is that we've inherited from the **Frame** widget. At the cost of a somewhat awkward call to the frame widget's constructor, we've created a new kind of custom widget that can be treated as any other widget. You can create and display the status bar using the usual widget syntax:

```
status = StatusBar(root)
status.pack(side=BOTTOM, fill=X)
```

We could have inherited from the Label widget itself, and just extended it with **set** and **clear** methods. This approach have a few drawbacks, though:

- It makes it harder to maintain the status bar's integrity. Some team members may cheat, and use **config** instead of **set**. That's not a big deal, until the day you decide to do some extra processing in the **set** method. Or the day you decide to use a **Canvas** widget to implement a fancier status bar.
- It increases the risk that your additional methods conflict with attributes or methods used by Tkinter. While the **Frame** and **Toplevel** widgets have relatively few methods, other widgets can have several dozens of widget specific attributes and methods.
- Future versions of Tkinter may use factory functions rather than class constructors for most widgets. However, it's more or less guaranteed that such versions will still provide **Frame** and **Toplevel** classes. Better safe than sorry, in other words.

[back](#) [next](#)

 rendered by a [django](#) application. hosted by [webfaction](#).

[back](#) [next](#)

Standard Dialogs

Before we look at what to put in that application work area, let's take a look at another important part of GUI programming: displaying dialogs and message boxes.

Starting with Tk 4.2, the Tk library provides a set of standard dialogs that can be used to display message boxes, and to select files and colors. In addition, Tkinter provides some simple dialogs allowing you to ask the user for integers, floating point values, and strings. Where possible, these standard dialogs use platform-specific mechanisms, to get the right look and feel.

Message Boxes

The **tkMessageBox** module provides an interface to the message dialogs.

The easiest way to use this module is to use one of the convenience functions: **showinfo**, **showwarning**, **showerror**, **askquestion**, **askokcancel**, **askyesno**, or **askretrycancel**. They all have the same syntax:

tkMessageBox.function(title, message [, options]).

The *title* argument is shown in the window title, and the message in the dialog body. You can use newline characters (“\n”) in the message to make it occupy multiple lines. The options can be used to modify the look; they are explained later in this section.

The first group of standard dialogs is used to present information. You provide the title and the message, the function displays these using an appropriate icon, and returns when the user has pressed OK. The return value should be ignored.

Here's an example:

```
try:
    fp = open(filename)
except:
    tkMessageBox.showwarning(
        "Open file",
        "Cannot open this file\n(%)s" % filename
    )
return
```

The showinfo dialog



The showwarning dialog



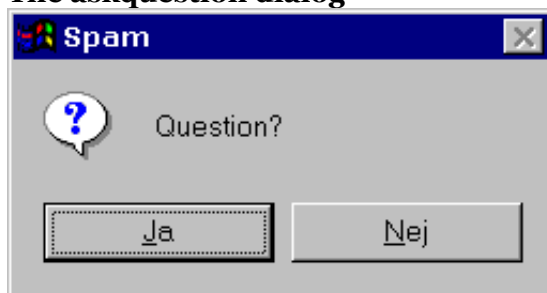
The showerror dialog



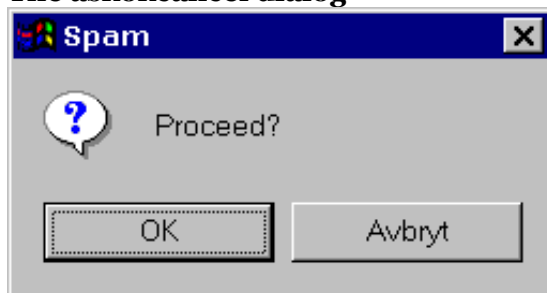
The second group is used to ask questions. The **askquestion** function returns the strings “yes” or “no” (you can use options to modify the number and type of buttons shown), while the others return a true value of the user gave a positive answer (**ok**, **yes**, and **retry**, respectively).

```
if tkMessageBox.askyesno("Print", "Print this report?"):
    report.print()
```

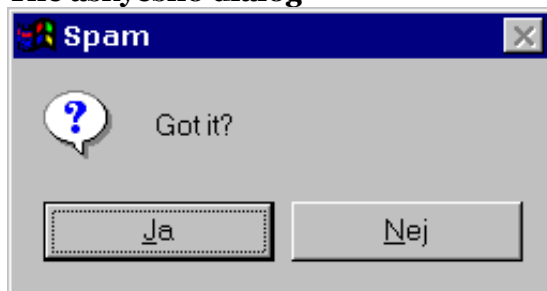
The askquestion dialog



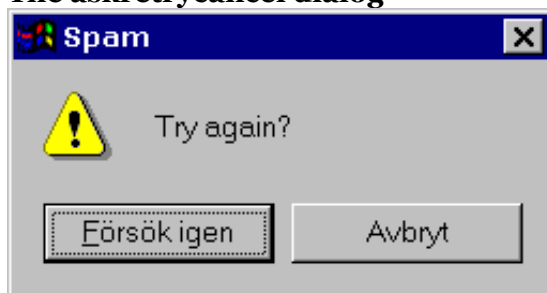
The askokcancel dialog



The askyesno dialog



The askretrycancel dialog



[Screenshots made on a Swedish version of Windows 95. Hope you don't mind...]

Message Box Options

If the standard message boxes are not appropriate, you can pick the closest alternative (**askquestion**, in most cases), and use options to change it to exactly suit your needs. You can use the following options (note that **message** and **title** are usually given as arguments, not as options).

default constant

Which button to make default: **ABORT**, **RETRY**, **IGNORE**, **OK**, **CANCEL**, **YES**, or **NO** (the constants are defined in the **tkMessageBox** module).

icon (constant)

Which icon to display: **ERROR**, **INFO**, **QUESTION**, or **WARNING**

message (string)

The message to display (the second argument to the convenience functions). May contain newlines.

parent (widget)

Which window to place the message box on top of. When the message box is closed, the focus is returned to the parent window.


title (string)

Message box title (the first argument to the convenience functions).

type (constant)

Message box type; that is, which buttons to display: **ABORTRETRYIGNORE**, **OK**, **OKCANCEL**, **RETRYCANCEL**, **YESNO**, or **YESNOCANCEL**.

[back](#) [next](#)

 rendered by a [django](#) application. hosted by [webfaction](#).

[back](#) [next](#)

Dialog Windows

While the standard dialogs described in the previous section may be sufficient for many simpler applications, most larger applications require more complicated dialogs. For example, to set configuration parameters for an application, you will probably want to let the user enter more than one value or string in each dialog.

Basically, creating a dialog window is no different from creating an application window. Just use the **Toplevel** widget, stuff the necessary entry fields, buttons, and other widgets into it, and let the user take care of the rest. (By the way, don't use the **ApplicationWindow** class for this purpose; it will only confuse your users).

But if you implement dialogs in this way, you may end up getting both your users and yourself into trouble. The standard dialogs all returned only when the user had finished her task and closed the dialog; but if you just display another toplevel window, everything will run in parallel. If you're not careful, the user may be able to display several copies of the same dialog, and both she and your application will be hopelessly confused.

In many situations, it is more practical to handle dialogs in a synchronous fashion; create the dialog, display it, wait for the user to close the dialog, and then resume execution of your application. The **wait_window** method is exactly what we need; it enters a local event loop, and doesn't return until the given window is destroyed (either via the **destroy** method, or explicitly via the window manager):

```
widget.wait_window(window)
```

(Note that the method waits until the window given as an argument is destroyed; the only reason this is a method is to avoid namespace pollution).

In the following example, the **MyDialog** class creates a **Toplevel** widget, and adds some widgets to it. The caller then uses **wait_window** to wait until the dialog is closed. If the user clicks OK, the entry field's value is printed, and the dialog is then explicitly destroyed.

Creating a simple dialog

```
from Tkinter import *

class MyDialog:

    def __init__(self, parent):

        top = self.top = Toplevel(parent)

        Label(top, text="Value").pack()

        self.e = Entry(top)
        self.e.pack(padx=5)

        b = Button(top, text="OK", command=self.ok)
        b.pack(pady=5)

    def ok(self):

        print "value is", self.e.get()

        self.top.destroy()

root = Tk()
Button(root, text="Hello!").pack()
root.update()

d = MyDialog(root)

root.wait_window(d.top)
```

If you run this program, you can type something into the entry field, and then click **OK**, after which the program terminates (note that we didn't call the **mainloop** method here; the local event loop handled by **wait_window** was sufficient). But there are a few problems with this example:

- The root window is still active. You can click on the button in the root window also when the dialog is displayed. If the dialog depends on the current application state, letting the users mess around with the application itself may be disastrous. And just being able to display multiple dialogs (or even multiple copies of one dialog) is a sure way to confuse your users.
- You have to explicitly click in the entry field to move the cursor into it, and also click on the OK button. Pressing **Enter** in the entry field is not sufficient.
- There should be some controlled way to cancel the dialog (and as we learned earlier, we really should handle the **WM_DELETE_WINDOW** protocol too).

To address the first problem, Tkinter provides a method called **grab_set**, which makes sure that no mouse or keyboard events are sent to the wrong window.

The second problem consists of several parts; first, we need to explicitly move the keyboard focus to the dialog. This can be done with the **focus_set** method. Second, we need to bind the **Enter** key so it calls the **ok** method. This is easy, just use the **bind** method on the **Toplevel** widget (and make sure to modify the **ok** method to take an optional argument so it doesn't choke on the event object).

The third problem, finally, can be handled by adding an additional **Cancel** button which calls the **destroy** method, and also use **bind** and **protocol** to do the same when the user presses **Escape** or explicitly closes the window.

The following **Dialog** class provides all this, and a few additional tricks. To implement your own dialogs, simply inherit from this class and override the **body** and **apply** methods. The former should create the dialog body, the latter is called when the user clicks OK.

A dialog support class (File: tkSimpleDialog.py)

```
from Tkinter import *
import os

class Dialog(Toplevel):

    def __init__(self, parent, title = None):

        Toplevel.__init__(self, parent)
        self.transient(parent)

        if title:
            self.title(title)

        self.parent = parent

        self.result = None

        body = Frame(self)
        self.initial_focus = self.body(body)
        body.pack(padx=5, pady=5)

        self.buttonbox()

        self.grab_set()

        if not self.initial_focus:
            self.initial_focus = self

        self.protocol("WM_DELETE_WINDOW", self.cancel)

        self.geometry("+%d+%d" % (parent.winfo_rootx()+50,
                                   parent.winfo_rooty()+50))
```



```

self.initial_focus.focus_set()

self.wait_window(self)

#
# construction hooks

def body(self, master):
    # create dialog body. return widget that should have
    # initial focus. this method should be overridden

    pass

def buttonbox(self):
    # add standard button box. override if you don't want the
    # standard buttons

    box = Frame(self)

    w = Button(box, text="OK", width=10, command=self.ok, default=ACTIVE)
    w.pack(side=LEFT, padx=5, pady=5)
    w = Button(box, text="Cancel", width=10, command=self.cancel)
    w.pack(side=LEFT, padx=5, pady=5)

    self.bind("<Return>", self.ok)
    self.bind("<Escape>", self.cancel)

    box.pack()

#
# standard button semantics

def ok(self, event=None):

    if not self.validate():
        self.initial_focus.focus_set() # put focus back
        return

    self.withdraw()
    self.update_idletasks()

    self.apply()

    self.cancel()

def cancel(self, event=None):

    # put focus back to the parent window
    self.parent.focus_set()
    self.destroy()

#
# command hooks

def validate(self):

    return 1 # override

def apply(self):

    pass # override

```

The main trickery is done in the constructor; first, **transient** is used to associate this window with a parent window (usually the application window from which the dialog was launched). The dialog won't show up as an icon in the window manager (it won't appear in the task bar under Windows, for example), and if you iconify the parent window, the dialog will be hidden as well. Next, the constructor creates the dialog body, and then calls **grab_set** to make the dialog modal, **geometry** to position the dialog relative to the parent window, **focus_set** to move the keyboard focus to the appropriate widget (usually the widget returned by the **body** method), and finally **wait_window**.

Note that we use the **protocol** method to make sure an explicit close is treated as a cancel, and in the **buttonbox** method, we bind the **Enter** key to OK, and **Escape** to Cancel. The **default=ACTIVE** call marks the OK button as a default button in a platform specific way.

Using this class is much easier than figuring out how it's implemented; just create the necessary widgets in the **body** method, and extract the result and carry out whatever you wish to do in the **apply** method. Here's a simple example (we'll take a closer look at the **grid** method in a moment).

Creating a simple dialog, revisited

```
import tkSimpleDialog

class MyDialog(tkSimpleDialog.Dialog):

    def body(self, master):

        Label(master, text="First:").grid(row=0)
        Label(master, text="Second:").grid(row=1)

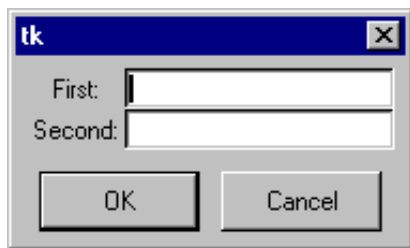
        self.e1 = Entry(master)
        self.e2 = Entry(master)

        self.e1.grid(row=0, column=1)
        self.e2.grid(row=1, column=1)
        return self.e1 # initial focus

    def apply(self):
        first = int(self.e1.get())
        second = int(self.e2.get())
        print first, second # or something
```

And here's the resulting dialog:

Running the dialog2.py script



Note that the **body** method may optionally return a widget that should receive focus when the dialog is displayed. If this is not relevant for your dialog, simply return **None** (or omit the return statement).

The above example did the actual processing in the **apply** method (okay, a more realistic example should probably do something with the result, rather than just printing it). But instead of doing the processing in the **apply** method, you can store the entered data in an instance attribute:

```
...

def apply(self):
    first = int(self.e1.get())
    second = int(self.e2.get())
    self.result = first, second

d = MyDialog(root)
print d.result
```

Note that if the dialog is cancelled, the **apply** method is never called, and the **result** attribute is never set. The **Dialog** constructor sets this attribute to **None**, so you can simply test the result before doing any processing of it. If you wish to return data in other attributes, make sure to initialize them in the **body** method (or simply set **result** to 1 in the **apply** method, and test it before accessing the other attributes).

Grid Layouts

While the **pack** manager was convenient to use when we designed application windows, it may not be that easy to use for dialogs. A typical dialog may include a

number of entry fields and check boxes, with corresponding labels that should be properly aligned. Consider the following simple example:

Simple Dialog Layout

First:	<entry field>
Second:	<entry field>
<checkboxbutton>	

To implement this using the **pack** manager, we could create a frame to hold the label “first:”, and the corresponding entry field, and use **side=LEFT** when packing them. Add a corresponding frame for the next line, and pack the frames and the checkbox into an outer frame using **side=TOP**. Unfortunately, packing the labels in this fashion makes it impossible to get the entry fields lined up, and if we use **side=RIGHT** to pack the entry field instead, things break down if the entry fields have different width. By carefully using **width** options, padding, **side** and **anchor** packer options, etc., we can get reasonable results with some effort. But there’s a much easier way: use the **grid** manager instead.

This manager splits the master widget (typically a frame) into a 2-dimensional grid, or table. For each widget, you only have to specify where in this grid it should appear, and the grid managers takes care of the rest. The following **body** method shows how to get the above layout:

Using the grid geometry manager

```
def body(self, master):
    Label(master, text="First:").grid(row=0, sticky=W)
    Label(master, text="Second:").grid(row=1, sticky=W)

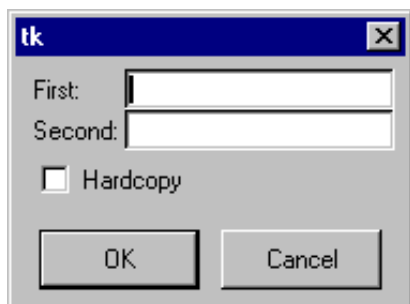
    self.e1 = Entry(master)
    self.e2 = Entry(master)

    self.e1.grid(row=0, column=1)
    self.e2.grid(row=1, column=1)

    self.cb = Checkbutton(master, text="Hardcopy")
    self.cb.grid(row=2, columnspan=2, sticky=W)
```

For each widget that should be handled by the grid manager, you call the **grid** method with the **row** and **column** options, telling the manager where to put the widget. The topmost row, and the leftmost column, is numbered 0 (this is also the default). Here, the checkbox is placed beneath the label and entry widgets, and the **columnspan** option is used to make it occupy more than one cell. Here’s the result:

Using the grid manager



If you look carefully, you’ll notice a small difference between this dialog, and the dialog shown by the **dialog2.py** script. Here, the labels are aligned to the left margin. If you compare the code, you’ll find that the only difference is an option called **sticky**.

When its time to display the frame widget, the grid geometry manager loops over all widgets, calculating a suitable width for each row, and a suitable height for each column. For any widget where the resulting cell turns out to be larger than the widget,

the widget is centered by default. The **sticky** option is used to modify this behavior. By setting it to one of **E**, **W**, **S**, **N**, **NW**, **NE**, **SE**, or **SW**, you can align the widget to any side or corner of the cell. But you can also use this option to stretch the widget if necessary; if you set the option to **E+W**, the widget will be stretched to occupy the full width of the cell. And if you set it to **E+W+N+S** (or **NW+SE**, etc), the widget will be stretched in both directions. In practice, the **sticky** option replaces the **fill**, **expand**, and **anchor** options used by the pack manager.

The grid manager provides many other options allowing you to tune the look and behavior of the resulting layout. These include **padx** and **pady** which are used to add extra padding to widget cells, and many others. See the [Grid Geometry Manager](#) chapter for details.

Validating Data

What if the user types bogus data into the dialog? In our current example, the **apply** method will raise an exception if the contents of an entry field is not an integer. We could of course handle this with a **try/except** and a standard message box:

```
...

def apply(self):
    try:
        first = int(self.e1.get())
        second = int(self.e2.get())
        dosomething((first, second))
    except ValueError:
        tkMessageBox.showwarning(
            "Bad input",
            "Illegal values, please try again"
        )
```

There's a problem with this solution: the **ok** method has already removed the dialog from the screen when the **apply** method is called, and it will destroy it as soon as we return. This design is intentional; if we carry out some potentially lengthy processing in the **apply** method, it would be very confusing if the dialog wasn't removed before we finished. The **Dialog** class already contain hooks for another solution: a separate **validate** method which is called before the dialog is removed.

In the following example, we simply moved the code from **apply** to **validate**, and changed it to store the result in an instance attribute. This is then used in the **apply** method to carry out the work.

```
...

def validate(self):
    try:
        first= int(self.e1.get())
        second = int(self.e2.get())
        self.result = first, second
        return 1
    except ValueError:
        tkMessageBox.showwarning(
            "Bad input",
            "Illegal values, please try again"
        )
        return 0

def apply(self):
    dosomething(self.result)
```

Note that if we left the processing to the calling program (as shown above), we don't even have to implement the **apply** method.

[back](#) [next](#)

 rendered by a [django](#) application. hosted by [webfaction](#).

An Introduction to Tkinter (Work in Progress)

This is the Tkinter introduction, last updated in **November 2005**. This is a work in progress.

Quick Navigation:

[Introduction](#)

[What's Tkinter](#)

[Events and Bindings](#)

[Standard Dialogs](#)

[Widget Class Reference](#)

[The Canvas Widget](#)

[The Entry Widget](#)

[The Listbox Widget](#)

[The Text Widget](#)

[More widgets...](#)

For the complete contents, see the [table of contents](#), or the document list below.

Overviews (1)

An Introduction To Tkinter [[tkinter-index](#)]

Articles (40)

[B](#) [C](#) [E](#) [F](#) [G](#) [L](#) [M](#) [O](#) [P](#) [R](#) [S](#) [T](#) [V](#) [W](#)

B

The Tkinter BitmapImage Class [[bitmapimage](#)]

The Tkinter Button Widget [[button](#)]

C

The Tkinter Canvas Widget [[canvas](#)]

The Tkinter Checkbutton Widget [[checkbutton](#)]

E

The Tkinter Entry Widget [[entry](#)]

F

The Tkinter Frame Widget [[frame](#)]

G

The Tkinter Grid Geometry Manager [[grid](#)]

L

The Tkinter Label Widget [[label](#)]

The Tkinter LabelFrame Widget [[labelframe](#)]

The Tkinter Listbox Widget [[listbox](#)]

M

The Tkinter Menu Widget [[menu](#)]
The Tkinter Menubutton Widget [[menubutton](#)]
The Tkinter Message Widget [[message](#)]

O

The Tkinter OptionMenu Widget [[optionmenu](#)]

P

The Tkinter Pack Geometry Manager [[pack](#)]
The Tkinter PanedWindow Widget [[panedwindow](#)]
The Tkinter PhotoImage Class [[photoimage](#)]
The Tkinter Place Geometry Manager [[place](#)]

R

The Tkinter Radiobutton Widget [[radiobutton](#)]

S

The Tkinter Scale Widget [[scale](#)]
The Tkinter Scrollbar Widget [[scrollbar](#)]
The Tkinter Spinbox Widget [[spinbox](#)]

T


The Tkinter Text Widget [[text](#)]
Application Windows [[tkinter-application-windows](#)]
Tkinter Classes [[tkinter-classes](#)]
Color Entry [[tkinter-color-dialogs](#)]
Dialog Windows [[tkinter-dialog-windows](#)]
Data Entry [[tkinter-entry-dialogs](#)]
Events and Bindings [[tkinter-events-and-bindings](#)]
File Dialogs [[tkinter-file-dialogs](#)]
Hello, Again [[tkinter-hello-again](#)]
Hello, Tkinter [[tkinter-hello-tkinter](#)]
Standard Dialogs [[tkinter-standard-dialogs](#)]
What's Tkinter? [[tkinter-whats-tkinter](#)]
Widget Configuration [[tkinter-widget-configuration](#)]
Widget Styling [[tkinter-widget-styling](#)]
The Tkinter Toplevel Widget [[toplevel](#)]

V

The Variable Classes (BooleanVar, DoubleVar, IntVar, StringVar) [[variable](#)]

W

Basic Widget Methods [[widget](#)]
Toplevel Window Methods [[wm](#)]

 rendered by a [django](#) application. hosted by [webfaction](#).

[back](#) [next](#)

The Tkinter Button Widget

The **Button** widget is a standard Tkinter widget used to implement various kinds of buttons. Buttons can contain text or images, and you can associate a Python function or method with each button. When the button is pressed, Tkinter automatically calls that function or method.

The button can only display text in a single font, but the text may span more than one line. In addition, one of the characters can be underlined, for example to mark a keyboard shortcut. By default, the **Tab** key can be used to move to a button widget.

When to use the Button Widget

Simply put, button widgets are used to let the user say “do this now!,” where *this* is either given by the text on the button, or implied by the icon displayed in the button. Buttons are typically used in toolbars, in application windows, and to accept or dismiss data entered into a dialog box.

For buttons suitable for data entry, see the [Checkbutton](#) and [Radiobutton](#) widgets.

Patterns

Plain buttons are pretty straightforward to use. All you have to do is to specify the button contents (text, bitmap, or image) and what function or method to call when the button is pressed:

```
from Tkinter import *

master = Tk()

def callback():
    print "click!"

b = Button(master, text="OK", command=callback)
b.pack()

mainloop()
```

A button without a callback is pretty useless; it simply doesn't do anything when you press the button. You might wish to use such buttons anyway when developing an application. In that case, it is probably a good idea to disable the button to avoid confusing your beta testers:

```
b = Button(master, text="Help", state=DISABLED)
```

If you don't specify a size, the button is made just large enough to hold its contents. You can use the **padx** and **pady** option to add some extra space between the contents and the button border.

You can also use the **height** and **width** options to explicitly set the size. If you display text in the button, these options define the size of the button in text units. If you display bitmaps or images instead, they define the size in pixels (or other screen units). You can specify the size in pixels even for text buttons, but that requires some magic. Here's one way to do it (there are others):

```
f = Frame(master, height=32, width=32)
f.pack_propagate(0) # don't shrink
f.pack()

b = Button(f, text="Sure!")
b.pack(fill=BOTH, expand=1)
```

Buttons can display multiple lines of text (but only in one font). You can use newlines, or use the **wraplength** option to make the button wrap text by itself. When wrapping

text, use the **anchor**, **justify**, and possibly **padx** options to make things look exactly as you wish. An example:

```
b = Button(master, text=longtext, anchor=W, justify=LEFT, padx=2)
```

To make an ordinary button look like it's held down, for example if you wish to implement a toolbox of some kind, you can simply change the relief from RAISED to SUNKEN:

```
b.config(relief=SUNKEN)
```

You might wish to change the background as well. Note that a possibly better solution is to use a **Checkbutton** or **Radiobutton** with the **indicatoron** option set to false:

```
b = Checkbutton(master, image=bold, variable=var, indicatoron=0)
```

In earlier versions of Tkinter, the **image** option overrides the **text** option. If you specify both, only the image is displayed. In later versions, you can use the **compound** option to change this behavior. To display text on top of an image, set **compound** to **CENTER**:

```
b = Button(master, text="Click me", image=pattern, compound=CENTER)
```

To display an icon along with the text, set the option to one of **LEFT**, **RIGHT**, **TOP**, or **BOTTOM**:

```
# put the icon to the left of the text label
b = Button(compound=LEFT, image=icon, text="Action")

# put the icon on top of the text
b = Button(compound=TOP, image=icon, text="Quit")
```

Reference

Button(master=None, **options) (class) [<#>]

A command button.

master

Parent widget.

***options*

Widget options. See the description of the [config](#) method for a list of available options.

config(options)** [<#>]

Modifies one or more widget options. If no options are given, the method returns a dictionary containing all current option values.

***options*

Widget options.

activebackground=

What background color to use when the button is active. The default is system specific. (the option database name is activeBackground, the class is Foreground)

activeforeground=

What foreground color to use when the button is active. The default is system specific. (activeForeground/Background)

anchor=

Controls where in the button the text (or image) should be located. Use one of **N**, **NE**, **E**, **SE**, **S**, **SW**, **W**, **NW**, or **CENTER**. Default is **CENTER**.

(anchor/Anchor)

background=
The background color. The default is system specific.
(background/Background)

bg=
Same as **background**.

bitmap=
The bitmap to display in the widget. If the **image** option is given, this option is ignored. (bitmap/Bitmap)

borderwidth=
The width of the button border. The default is platform specific, but is usually 1 or 2 pixels. (borderWidth/BorderWidth)

bd=
Same as **borderwidth**.

command=
A function or method that is called when the button is pressed. The callback can be a function, bound method, or any other callable Python object. If this option is not used, nothing will happen when the user presses the button. (command/Command)

compound=
Controls how to combine text and image in the button. By default, if an image or bitmap is given, it is drawn instead of the text. If this option is set to **CENTER**, the text is drawn on top of the image. If this option is set to one of **BOTTOM**, **LEFT**, **RIGHT**, or **TOP**, the image is drawn besides the text (use **BOTTOM** to draw the image under the text, etc.). Default is **NONE**. (compound/Compound)

cursor=
The cursor to show when the mouse is moved over the button.
(cursor/Cursor)

default=
If set, the button is a default button. Tkinter will indicate this by drawing a platform specific indicator (usually an extra border). The default is **DISABLED** (no default behavior). (default/Default)

disabledforeground=
The color to use when the button is disabled. The background is shown in the **background** color. The default is system specific.
(disabledForeground/DisabledForeground)

font=
The font to use in the button. The button can only contain text in a single font. The default is system specific. (font/Font)

foreground=
The color to use for text and bitmap content. The default is system specific. (foreground/Foreground)

fg=
Same as **foreground**.

height=
The height of the button. If the button displays text, the size is given in text units. If the button displays an image, the size is given in pixels (or screen units). If the size is omitted, it is calculated based on the button contents. (height/Height)

highlightbackground=
The color to use for the highlight border when the button does not have focus. The default is system specific.
(highlightBackground/HighlightBackground)

highlightcolor=
The color to use for the highlight border when the button has focus. The default is system specific. (highlightColor/HighlightColor)

highlightthickness=
The width of the highlight border. The default is system specific (usually one or two pixels). (highlightThickness/HighlightThickness)

image=
The image to display in the widget. If specified, this takes precedence over the **text** and **bitmap** options. (image/Image)

justify=
Defines how to align multiple lines of text. Use **LEFT**, **RIGHT**, or

CENTER. Default is **CENTER**. (justify/Justify)

overrelief=
Alternative relief to use when the mouse is moved over the widget. If empty, always use the **relief** value. (overRelief/OverRelief)

padx=
Extra horizontal padding between the text or image and the border. (padX/Pad)

pady=
Extra vertical padding between the text or image and the border. (padY/Pad)

relief=
Border decoration. Usually, the button is **SUNKEN** when pressed, and **RAISED** otherwise. Other possible values are **GROOVE**, **RIDGE**, and **FLAT**. Default is **RAISED**. (relief/Relief)

repeatdelay=
(repeatDelay/RepeatDelay)

repeatinterval=
(repeatInterval/RepeatInterval)

state=
The button state: **NORMAL**, **ACTIVE** or **DISABLED**. Default is **NORMAL**. (state/State)

takefocus=
Indicates that the user can use the **Tab** key to move to this button. Default is an empty string, which means that the button accepts focus only if it has any keyboard bindings (default is on, in other words). (takeFocus/TakeFocus)

text=
The text to display in the button. The text can contain newlines. If the **bitmap** or **image** options are used, this option is ignored (unless the **compound** option is used). (text/Text)

textvariable=
Associates a Tkinter variable (usually a **StringVar**) to the button. If the variable is changed, the button text is updated. (textVariable/Variable)

underline=
Which character to underline, in a text label. Default is -1, which means that no character is underlined. (underline/Underline)

width=
The width of the button. If the button displays text, the size is given in text units. If the button displays an image, the size is given in pixels (or screen units). If the size is omitted, or zero, it is calculated based on the button contents. (width/Width)

wraplength=
Determines when a button's text should be wrapped into multiple lines. This is given in screen units. Default is 0 (no wrapping). (wrapLength/WrapLength)


flash() [<#>]

Flash the button. This method redraws the button several times, alternating between active and normal appearance.

invoke() [<#>]

Call the command associated with the button.

[back](#) [next](#)

 rendered by a [django](#) application. hosted by [webfaction](#).

[back](#) [next](#)

The Tkinter Canvas Widget

The **Canvas** widget provides structured graphics facilities for Tkinter. This is a highly versatile widget which can be used to draw graphs and plots, create graphics editors, and implement various kinds of custom widgets.

When to use the Canvas Widget

The canvas is a general purpose widget, which is typically used to display and edit graphs and other drawings.

Another common use for this widget is to implement various kinds of custom widgets. For example, you can use a canvas as a completion bar, by drawing and updating a rectangle on the canvas.

Patterns

To draw things in the canvas, use the **create** methods to add new items.

```
from Tkinter import *

master = Tk()

w = Canvas(master, width=200, height=100)
w.pack()

w.create_line(0, 0, 200, 100)
w.create_line(0, 100, 200, 0, fill="red", dash=(4, 4))

w.create_rectangle(50, 25, 150, 75, fill="blue")

mainloop()
```

Note that items added to the canvas are kept until you remove them. If you want to change the drawing, you can either use methods like **coords**, **itemconfig**, and **move** to modify the items, or use **delete** to remove them.

```
i = w.create_line(xy, fill="red")

w.coords(i, new_xy) # change coordinates
w.itemconfig(i, fill="blue") # change color

w.delete(i) # remove

w.delete(ALL) # remove all items
```

Concepts

To display things on the canvas, you create one or more *canvas items*, which are placed in a stack. By default, new items are drawn on top of items already on the canvas.

Tkinter provides lots of methods allowing you to manipulate the items in various ways. Among other things, you can attach (*bind*) event callbacks to individual canvas items.

Canvas Items

The **Canvas** widget supports the following standard items:

- [arc](#) (arc, chord, or pieslice)
- [bitmap](#) (built-in or read from XBM file)
- [image](#) (a [BitmapImage](#) or [PhotoImage](#) instance)
- [line](#)
- [oval](#) (a circle or an ellipse)
- [polygon](#)

- [rectangle](#)
- [text](#)
- [window](#)

Chords, pieslices, ovals, polygons, and rectangles consist of both an outline and an interior area, either of which can be made transparent (and if you insist, you can make both transparent).

Window items are used to place other Tkinter widgets on top of the canvas; for these items, the Canvas widget simply acts like a geometry manager.

You can also write your own item types in C or C++ and plug them into Tkinter via Python extension modules.

Coordinate Systems

The **Canvas** widget uses two coordinate systems; the window coordinate system (with (0, 0) in the upper left corner), and a canvas coordinate system which specify where the items are drawn. By scrolling the canvas, you can specify which part of the canvas coordinate system to show in the window.

The **scrollregion** option is used to limit scrolling operations for the canvas. To set this, you can usually use something like:

```
canvas.config(scrollregion=canvas.bbox(ALL))
```

To convert from window coordinates to canvas coordinates, use the [canvasx](#) and [canvasy](#) methods:

```
def callback(event):
    canvas = event.widget
    x = canvas.canvasx(event.x)
    y = canvas.canvasy(event.y)
    print canvas.find_closest(x, y)
```

Item Specifiers: Handles and Tags

The **Canvas** widget allows you to identify items in several ways. Everywhere a method expects an item specifier, you can use one of the following:

- item handles (integers)
- tags
- **ALL**
- **CURRENT**

Item handles are integer values used to identify a specific item on the canvas. Tkinter automatically assigns a new handle to each new item created on the canvas. Item handles can be passed to the various canvas methods either as integers or as strings.

Tags are symbolic names attached to items. Tags are ordinary strings, and they can contain anything except whitespace (as long as they don't look like item handles).

An item can have zero or more tags associated with it, and the same tag can be used for more than one item. However, unlike the **Text** widget, the **Canvas** widget doesn't allow you to create bindings or otherwise configure tags for which there are no existing items. Tags are owned by the items, not the widget itself. All such operations are ignored.

You can either specify the tags via an option when you create the item, set them via the [itemconfig](#) method, or add them using the [addtag_withtag](#) method. The **tags** option takes either a single tag string, or a tuple of strings.

```
item = canvas.create_line(0, 0, 100, 100, tags="uno")
canvas.itemconfig(item, tags=("one", "two"))
canvas.addtag_withtag("three", "one")
```

To get all tags associated with a specific item, use **gettags**. To get the handles for all items having a given tag, use **find_withtag**.

```
>>> print canvas.gettags(item)
('one', 'two', 'three')
>>> print canvas.find_withtag("one")
(1,)
```

The **Canvas** widget also provides two predefined tags:

ALL (or the string “all”) matches all items on the canvas.

CURRENT (or “current”) matches the item under the mouse pointer, if any. This can be used inside mouse event bindings to refer to the item that triggered the callback.

Printing

The Tkinter widget supports printing to Postscript printers.

Performance Issues

The **Canvas** widget implements a straight-forward damage/repair display model. Changes to the canvas, and external events such as **Expose**, are all treated as “damage” to the screen. The widget maintains a **dirty rectangle** to keep track of the damaged area.

When the first damage event arrives, the canvas registers an idle task (using **after_idle**) which is used to “repair” the canvas when the program gets back to the Tkinter main loop. You can force updates by calling the **update_idletasks** method.

When it’s time to redraw the canvas, the widget starts by allocating a pixmap (on X windows, this is an image memory stored on the display) with the same size as the dirty rectangle.

It then loops over the canvas items, and redraws *all* items for which the bounding box touch the dirty rectangle (this means that diagonal lines may be redrawn also if they don’t actually cover the rectangle, but this is usually no big deal).

Finally, the widget copies the pixmap to the display, and releases the pixmap. The copy operation is a very fast operation on most modern hardware.

Since the canvas uses a *single* dirty rectangle, you can sometimes get better performance by forcing updates. For example, if you’re changing things in different parts of the canvas without returning to the main loop, adding explicit calls to **update_idletasks()** allows the canvas to update a few small rectangles, instead of a large one with many more objects.

Reference

Canvas(master=None, **options) (class) [<#>]

A structured graphics canvas.

master

Parent widget.

***options*

Widget options. See the description of the [config](#) method for a list of available options.

addtag(tag, method, *args) [<#>]

Adds a tag to a number of items. Application code should use more specific methods wherever possible (that is, use [addtag_above](#) instead of **addtag(“above”)**, and so on.

tag

The tag to add.

method

How to add a new tag. This can be one of “above”, “all”, “below”, “closest”, “enclosed”, “overlapping” or “withtag”.

**args*

Additional arguments. For details, see the description of the individual method.

addtag_above(tag, item) [<#>]

Adds a tag to the item just above the given item.

tag

The tag to add.

item

The tag or id of the reference item.

addtag_all(tag) [<#>]

Adds a tag to all items on the canvas. This is a shortcut for **addtag_withtag(newtag, ALL)**.

tag

The tag to add.

addtag_below(tag, item) [<#>]

Adds a tag to the item just below the given item.

tag

The tag to add.

item

The tag or id of the reference item.

addtag_closest(tag, x, y, halo=None, start=None) [<#>]

Adds a tag to the item closest to the given coordinate. Note that the position is given in canvas coordinates, and that this method always succeeds if there's at least one item in the canvas. To add tags to items within a certain distance from the position, use [add_overlapping](#) (dead link) with a small rectangle centered on the position.

tag

The tag to add.

x

The horizontal coordinate.

y

The vertical coordinate.

halo

Optional halo distance.

start

Optional start item.

addtag_enclosed(tag, x1, y1, x2, y2) [<#>]

Adds a tag to all items enclosed by the given rectangle.

tag

The tag to add.

x1

Left coordinate.

y1

Top coordinate.

x2

Right coordinate.

y2

Bottom coordinate.

addtag_overlapped(tag, x1, y1, x2, y2) [#]

Adds a tag to all items overlapping the given rectangle. This includes items that are completely enclosed by it.

tag

The tag to add.

x1

Left coordinate.

y1

Top coordinate.

x2

Right coordinate.

y2

Bottom coordinate.

addtag_withtag(tag, item) [#]

Adds a tag to all items having the given tag.

tag

The tag to add.

item

The reference item. If a tag is given, the new tag is added to all items that have this tag. You can also give an id, to add a tag to a single item.

bbox(item=None) [#]

Returns the bounding box for all matching items. If the tag is omitted, the bounding box for all items is returned. Note that the bounding box is approximate and may differ a few pixels from the real value.

item

Item specifier. If omitted, the bounding box for all elements on the canvas.

Returns:

The bounding box, as a 4-tuple.

canvasx(x, gridspacing=None) [#]

Converts a window coordinate to a canvas coordinate.

x

Screen coordinate.

gridspacing

Optional grid spacing. The coordinate is rounded to the nearest grid coordinate.

Returns:

Canvas coordinate.

canvasy(y, gridspacing=None) [#]

Converts a window coordinate to a canvas coordinate.

y

Screen coordinate.

gridspacing

Optional grid spacing. The coordinate is rounded to the nearest grid

coordinate.

Returns:

Canvas coordinate.

config(options) [#]**

Modifies one or more widget options. If no options are given, the method returns a dictionary containing all current option values.

****options**

Widget options.

background=

Canvas background color. Defaults to the standard widget background color. (the database name is background, the class is Background)

bg=

Same as **background**.

borderwidth=

Width of the canvas border. The default is 0 (no border). (borderWidth/BorderWidth)

bd=

Same as **borderwidth**.

closeenough=

The default value is 1. (closeEnough/CloseEnough)

confine=

The default value is 1. (confine/Confine)

cursor=

The cursor to use when the mouse is moved over the canvas. (cursor/Cursor)

height=

Canvas width. Default value is '7c'. (height/Height)

highlightbackground=

The color to use for the highlight border when the canvas does not have focus. The default is system specific. (highlightBackground/HighlightBackground)

highlightcolor=

The color to use for the highlight border when the canvas has focus. The default is system specific. (highlightColor/HighlightColor)

highlightthickness=

The width of the highlight border. The default is system specific (usually one or two pixels). (highlightThickness/HighlightThickness)

insertbackground=

The color to use for the text insertion cursor. The default is system specific. (insertBackground/Foreground)

insertborderwidth=

Width of the insertion cursor's border. If this is set to a non-zero value, the cursor is drawn using the **RAISED** border style. (insertBorderWidth/BorderWidth)

insertofftime=

Together with **insertontime**, this option controls cursor blinking. Both values are given in milliseconds. (insertOffTime/OffTime)

insertontime=

See **insertofftime**. (insertOnTime/OnTime)

insertwidth=

Width of the insertion cursor. Usually one or two pixels. (insertWidth/InsertWidth)

offset=

Default value is '0,0'. (offset/Offset)

relief=

Border style. The default is **FLAT**. Other possible values are **SUNKEN**, **RAISED**, **GROOVE**, and **RIDGE**. (relief/Relief)

scrollregion=

Canvas scroll region. No default value. (scrollRegion/ScrollRegion)

selectbackground=

Selection background color. The default is system and display specific.

(selectBackground/Foreground)
selectborderwidth=
 Selection border width. The default is system specific.
 (selectBorderWidth/BorderWidth)
selectforeground=
 Selection text color. The default is system specific.
 (selectForeground/Background)
state=
 Canvas state. One of NORMAL, DISABLED, or HIDDEN. The default is NORMAL. Note that this is a global setting, but individual canvas items can use the item-level **state** option to override this setting. (state/State)
takefocus=
 Indicates that the user can use the **Tab** key to move to this widget. Default is an empty string, which means that the canvas widget accepts focus only if it has any keyboard bindings. (takeFocus/TakeFocus)
width=
 Canvas width. Default value is '10c'. (width/Width)
xscrollcommand=
 Used to connect a canvas to a horizontal scrollbar. This option should be set to the **set** method of the corresponding scrollbar.
 (xScrollCommand/ScrollCommand)
xscrollincrement=
 Default value is 0. (xScrollIncrement/ScrollIncrement)
yscrollcommand=
 Used to connect a canvas to a vertical scrollbar. This option should be set to the **set** method of the corresponding scrollbar.
 (yScrollCommand/ScrollCommand)
yscrollincrement=
 Default value is 0. (yScrollIncrement/ScrollIncrement)

coords(item, *coords) [#]

Returns the coordinates for an item.

item

Item specifier (tag or id).

**coords*

Optional list of coordinate pairs. If given, the coordinates will replace the current coordinates for all matching items.

Returns:

If no coordinates are given, this method returns the coordinates for the matching item. If the item specifier matches more than one item, the coordinates for the first item found is returned.

create_arc(bbox, **options) [#]

Draws an arc, pieslice, or chord on the canvas. The new item is drawn on top of the existing items.

bbox

Bounding box for the full arc.

***options*

Arc options.

activedash=

activefill=

Fill color to use when the mouse pointer is moved over the item, if different from **fill**.

activeoutline=

activeoutlinestipple=

activestipple=

activewidth=

Default is 0.0.

dash=

Outline dash pattern, given as a list of segment lengths. Only the odd

segments are drawn.

dashoffset=
Default is 0.

disableddash=

disabledfill=
Fill color to use when the item is disabled, if different from **fill**.

disabledoutline=

disabledoutlinestipple=

disabledstipple=

disabledwidth=
Default is 0.0.

extent=
The size, relative to the **start** angle. Default is 90.0.

fill=
Fill color. An empty string means transparent.

offset=
Default is "0,0".

outline=
Outline color. Default is "black".

outlineoffset=
Default is "0,0".

outlinestipple=
Outline stipple pattern.

start=
Start angle. Default is 0.0.

state=
Item state. One of NORMAL, DISABLED, or HIDDEN.

stipple=
Stipple pattern.

style=
One of PIESLICE, CHORD, or ARC. Default is PIESLICE.

tags=
A tag to attach to this item, or a tuple containing multiple tags.

width=
Default is 1.0.

Returns:
The item id.

create_bitmap(position, **options) [<#>]

Draws a bitmap on the canvas.

position
Bitmap position, given as two coordinates.

***options*
Bitmap options.

activebackground=

activebitmap=

activeforeground=

anchor=
Where to place the bitmap relative to the given position. Default is CENTER.

background=
Background color, used for pixels that are "off". Use an empty string to make the background transparent. Default is transparent.

bitmap=
The bitmap descriptor. See [BitmapImage](#) for more information. (To display a **BitmapImage** object, use the [create_image](#) function.)

disabledbackground=

disabledbitmap=

disabledforeground=

foreground=
Foreground colors, used for pixels that are "on". Default is "black".

state=

Item state. One of NORMAL, DISABLED, or HIDDEN.

tags=

A tag to attach to this item, or a tuple containing multiple tags.

Returns:

The item id.

create_image(position, **options) [#]

Draws an image on the canvas.

position

Image position, given as two coordinates.

***options*

Image options.

activeimage=

anchor=

Where to place the image relative to the given position. Default is CENTER.

disabledimage=

image=

The image object. This should be a [PhotoImage](#) or [BitmapImage](#), or a compatible object (such as the PIL PhotoImage). The application must keep a reference to the image object.

state=

Item state. One of NORMAL, DISABLED, or HIDDEN.

tags=

A tag to attach to this item, or a tuple containing multiple tags.

Returns:

The item id.

create_line(coords, **options) [#]

Draws a line on the canvas.

coords

Image coordinates.

***options*

Line options.

activedash=

activefill=

Line color to use when the mouse pointer is moved over the item, if different from **fill**.

activestipple=

activewidth=

Default is 0.0.

arrow=

Default is NONE.

arrowshape=

Default is "8 10 3".

capstyle=

Default is BUTT.

dash=

Dash pattern, given as a list of segment lengths. Only the odd segments are drawn.

dashoffset=

Default is 0.

disableddash=

disabledfill=

Line color to use when the item is disabled, if different from **fill**.

disabledstipple=

disabledwidth=

Default is 0.0.

fill=

Line color. Default is "black".

joinstyle=
 Default is ROUND.
offset=
 Default is “o,o”.
smooth=
 Default is 0.
splinessteps=
 Default is 12.
state=
 Item state. One of NORMAL, DISABLED, or HIDDEN.
stipple=
 Stipple pattern.
tags=
 A tag to attach to this item, or a tuple containing multiple tags.
width=
 Default is 1.0.
 Returns:
 The item id.

create_oval(bbox, **options) [<#>]

Draws an ellipse on the canvas.

bbox
 Ellipse coordinates.
***options*
 Ellipse options.
activedash=
activefill=
 Fill color to use when the mouse pointer is moved over the item, if different from **fill**.
activeoutline=
activeoutlinestipple=
activestipple=
activewidth=
 Default is 0.0.
dash=
 Outline dash pattern, given as a list of segment lengths. Only the odd segments are drawn.
dashoffset=
 Default is 0.
disableddash=
disabledfill=
 Fill color to use when the item is disabled, if different from **fill**.
disabledoutline=
disabledoutlinestipple=
disabledstipple=
disabledwidth=
 Default is 0.
fill=
 Fill color. An empty string means transparent.
offset=
 Default is “o,o”.
outline=
 Outline color. Default is “black”.
outlineoffset=
 Default is “o,o”.
outlinestipple=
 Outline stipple pattern.
state=
 Item state. One of NORMAL, DISABLED, or HIDDEN.
stipple=
 Stipple pattern.
tags=

A tag to attach to this item, or a tuple containing multiple tags.

width=

Default is 1.0.

Returns:

The item id.

create_polygon(coords, **options) [#]

Draws a polygon on the canvas.

coords

Polygon coordinates.

****options**

Polygon options.

activedash=

activefill=

Fill color to use when the mouse pointer is moved over the item, if different from **fill**.

activeoutline=

activeoutlinestipple=

activestipple=

activewidth=

Default is 0.0.

dash=

Outline dash pattern, given as a list of segment lengths. Only the odd segments are drawn.

dashoffset=

Default is 0.

disableddash=

disabledfill=

Fill color to use when the item is disabled, if different from **fill**.

disabledoutline=

disabledoutlinestipple=

disabledstipple=

disabledwidth=

Default is 0.0.

fill=

Fill color. Default is "black".

joinstyle=

Default is ROUND.

offset=

Default is "0,0".

outline=

Outline color.

outlineoffset=

Default is "0,0".

outlinestipple=

Outline stipple pattern.

smooth=

Default is 0.

splinsteps=

Default is 12.

state=

Item state. One of NORMAL, DISABLED, or HIDDEN.

stipple=

Stipple pattern.

tags=

A tag to attach to this item, or a tuple containing multiple tags.

width=

Default is 1.0.

Returns:

The item id.

create_rectangle(bbox, **options) [<#>]

Draws a rectangle on the canvas.

bbox

Rectangle bounding box.

***options*

Rectangle options.

activedash=

activefill=

Fill color to use when the mouse pointer is moved over the item, if different from **fill**.

activeoutline=

activeoutlinestipple=

activestipple=

activewidth=

Default is 0.0.

dash=

Outline dash pattern, given as a list of segment lengths. Only the odd segments are drawn.

dashoffset=

Default is 0.

disableddash=

disabledfill=

Fill color to use when the item is disabled, if different from **fill**.

disabledoutline=

disabledoutlinestipple=

disabledstipple=

disabledwidth=

Default is 0.

fill=

Fill color. An empty string means transparent.

offset=

Default is "0,0".

outline=

Outline color. Default is "black".

outlineoffset=

Default is "0,0".

outlinestipple=

Outline stipple pattern.

state=

Item state. One of NORMAL, DISABLED, or HIDDEN.

stipple=

Stipple pattern.

tags=

A tag to attach to this item, or a tuple containing multiple tags.

width=

Default is 1.0.

Returns:

The item id.

create_text(position, **options) [<#>]

Draws text on the canvas.

position

Text position, given as two coordinates. By default, the text is centered on this position. You can override this with the **anchor** option. For example, if the coordinate is the upper left corner, set the **anchor** to **NW**.

***options*

Text options.

activefill=

Text color to use when the mouse pointer is moved over the item, if

different from **fill**.
activestipple=
anchor=
 Where to place the text relative to the given position. Default is CENTER.
disabledfill=
 Text color to use when the item is disabled, if different from **fill**.
disabledstipple=
fill=
 Text color. Default is “black”.
font=
 Font specifier. Default is system specific.
justify=
 Default is LEFT.
offset=
 Default is “o,o”.
state=
 Item state. One of NORMAL, DISABLED, or HIDDEN.
stipple=
 Stipple pattern.
tags=
 A tag to attach to this item, or a tuple containing multiple tags.
text=
 The text to display.
width=
 Maximum line length. Lines longer than this value are wrapped. Default is 0 (no wrapping).
 Returns:
 The item id.

create_window(position, **options) [<#>]

Places a Tkinter widget on the canvas. Note that widgets are drawn on top of the canvas (that is, the canvas acts like a geometry manager). You cannot draw other canvas items on top of a widget.

position
 Window position, given as two coordinates.
***options*
 Window options.
anchor=
 Where to place the widget relative to the given position. Default is CENTER.
height=
 Window height. Default is to use the window’s requested height.
state=
 Item state. One of NORMAL, DISABLED, or HIDDEN.
tags=
 A tag to attach to this item, or a tuple containing multiple tags.
width=
 Window width. Default is to use the window’s requested width.
window=
 Window object.
 Returns:
 The item id.

dchars(item, from, to=None) [<#>]

Deletes text from an editable item.

item
 Item specifier.
from
 Where to start deleting text.
to

Where to stop deleting text. If omitted, a single character is removed.

delete(item) [#]

Deletes all matching items. It is not an error to give an item specifier that doesn't match any items.

item

Item specifier (tag or id).

dtag(item, tag=None) [#]

Removes the given tag from all matching items. If the tag is omitted, all tags are removed from the matching items. It is not an error to give a specifier that doesn't match any items.

item

The item specifier (tag or id).

tag

The tag to remove from matching items. If omitted, all tags are removed.

find_above(item) [#]

Returns the item just above the given item.

item

Reference item.

find_all() [#]

Returns all items on the canvas. This method returns a tuple containing the identities of all items on the canvas, with the topmost item last (that is, if you haven't change the order using [lift](#) or [lower](#), the items are returned in the order you created them). This is shortcut for **find_withtag(ALL)**.

Returns:

A tuple containing all items on the canvas.

find_below(item) [#]

Returns the item just below the given item.

item

Reference item.

find_closest(x, y, halo=None, start=None) [#]

Returns the item closest to the given position. Note that the position is given in canvas coordinates, and that this method always succeeds if there's at least one item in the canvas. To find items within a certain distance from a position, use [find_overlapping](#) with a small rectangle centered on the position.

x

Horizontal screen coordinate.

y

Vertical screen coordinate.

halo

Optional halo distance.

start

Optional start item.

Returns:

An item specifier.

find_enclosed(x1, y1, x2, y2) [<#>]

Finds all items completely enclosed by the rectangle (x1, y1, x2, y2).

x1
Left edge.
y1
Upper edge.
x2
Right edge.
y2
Lower edge.
Returns:
A tuple containing all matching items.

find_overlapping(x1, y1, x2, y2) [<#>]

Finds all items that overlap the given rectangle, or that are completely enclosed by it.

x1
Left edge.
y1
Upper edge.
x2
Right edge.
y2
Lower edge.
Returns:
A tuple containing all matching items.

find_withtag(item) [<#>]

Finds all items having the given specifier.

item
Item specifier.

focus(item=None) [<#>]

Moves focus to the given item. If the item has keyboard bindings, it will receive all further keyboard events, given that the canvas itself also has focus. It's usually best to call `focus_set` on the canvas whenever you set focus to a canvas item.

To remove focus from the item, call this method with an empty string.

To find out what item that currently has focus, call this method without any arguments.

item
Item specifier. To remove focus from any item, use an empty string.
Returns:
If the item specifier is omitted, this method returns the item that currently has focus, or `None` if no item has focus.

gettags(item) [<#>]

Gets tags associated with an item.

item

Item specifier.

Returns:

A tuple containing all tags associated with the item.

icursor(item, index) [<#>]

Moves the insertion cursor to the given position. This method can only be used with editable items.

item

Item specifier.

index

Cursor index.

index(item, index) [<#>]

Gets the numerical cursor index corresponding to the given index. Numerical indexes work like Python's sequence indexes; 0 is just to the left of the first character, and len(text) is just to the right of the last character.

item

Item specifier.

index

An index. You can use a numerical index, or one of INSERT (the current insertion cursor), END (the length of the text), or SEL_FIRST and SEL_LAST (the selection start and end). You can also use the form "@x,y" where x and y are canvas coordinates, to get the index closest to the given coordinate.

Returns:

A numerical index (an integer).

insert(item, index, text) [<#>]

Inserts text into an editable item.

item

Item specifier.

index

Where to insert the text. This can be either a numerical index or a symbolic constant. See the description of the [index](#) method for details. If you insert text at the INSERT index, the cursor is moved along with the text.

text

The text to insert.

itemcget(item, option) [<#>]

Gets the current value for an item option.

item

Item specifier.

option

Item option.

Returns:

The option value. If the item specifier refers to more than one item, this method returns the option value for the first item found.

itemconfig(item, **options) [<#>]

Changes one or more options for all matching items.

item
Item specifier.

***options*
Item options.

itemconfigure(item, **options) [<#>]

Same as [itemconfig](#).

lift(item, **options) [<#>]

(Deprecated) Moves item to top of stack. Same as [tag_raise](#).

lower(item, **options) [<#>]

(Deprecated) Moves item to bottom of stack. Same as [tag_lower](#).

move(item, dx, dy) [<#>]

Moves matching items by an offset.

item
Item specifier.

dx
Horizontal offset.

dy
Vertical offset.

postscript(options) [<#>]**

Generates a Postscript rendering of the canvas contents. Images and embedded widgets are not included.

***options*
Postscript options.

scale(self, xscale, yscale, xoffset, yoffset) [<#>]

Resizes matching items by scale factor. The coordinates for each item are recalculated as $((\text{coord}-\text{offset}) * \text{scale} + \text{offset})$; in other words, each item first moved by $-\text{offset}$, then multiplied with the scale factor, and then moved back again. Note that this method modifies the item coordinates; you may lose precision if you use this method several times on the same items.

xscale
Horizontal scale.

yscale
Vertical scale.

xoffset
Horizontal offset, in canvas coordinates.

yoffset
Vertical scale, in canvas coordinates.

scan_dragto(x, y) [<#>]

Scrolls the widget contents relative to the scanning anchor. The contents are moved 10 times the distance between the anchor and the given position. Use [scan_mark](#) to set the anchor.

x
The horizontal coordinate.

y
The vertical coordinate.

scan_mark(x, y) [<#>]

Sets the scanning anchor. This sets an anchor that can be used for fast scrolling to the given mouse coordinate.

x
The horizontal coordinate.

y
The vertical coordinate.

select_adjust(item, index) [<#>]

Adjusts the selection, so that it includes the given index. This method also sets the selection anchor to this position. This is typically used by mouse bindings.

item
Item specifier.

index
Selection index.

select_clear() [<#>]

Removes the selection, if it is in this canvas widget.

select_from(item, index) [<#>]

Sets the selection anchor point. Use [select_adjust](#) or [select_to](#) to extend the selection.

item
Item specifier.

index
Selection anchor.

select_item() [<#>]

Returns the item that owns the text selection for this canvas widget.

Note that this method always returns None in some older versions of Tkinter. To work around this problem, replace the method call with **canvas.tk.call(canvas._w, "select", "item")**.

Returns:
Item specifier, or None if there's no selection.

select_to(item, index) [<#>]

Modifies the selection so it includes the region between the current selection anchor and the given index. The anchor is set by [select_from](#) or [select_adjust](#).

item
Item specifier.

index
Selection end point.

tag_bind(item, event=None, callback, add=None) [<#>]

Adds an event binding to all matching items.

Note that the new bindings are associated with the items, not the tag. For

example, if you attach bindings to all items having the **movable** tag, they will only be attached to any existing items with that tag. If you create new items tagged as **movable**, they will not get those bindings.

item

The item specifier (tag or id).

event

The event specifier.

callback

A callable object that takes one argument. This callback is called with an event descriptor, for events matching the given event specifier.

add

Usually, the new binding replaces any existing binding for the same event sequence. If this argument is present, and set to “+”, the new binding is added to to any existing binding.

tag_lower(item) [<#>]

Moves a canvas item to the bottom of the canvas stack. If multiple items match, they are all moved, with their relative order preserved.

This method doesn't work with window items. To change their order, use **lower** on the widget instance instead.

item

Item specifier.

tag_raise(item) [<#>]

Moves the given item to the top of the canvas stack. If multiple items match, they are all moved, with their relative order preserved.

This method doesn't work with window items. To change their order, use **lift** on the widget instance instead.

item

Item specifier.

tag_unbind(self, item, event) [<#>]

Removes the binding, if any, for the given event sequence. This applies to all matching items.

item

Item specifier.

sequence

Event specifier.

tkraise(item, **options) [<#>]

(Deprecated) Same as [tag_raise](#).

type(item) [<#>]

Returns the type of the given item. If item refers to more than one item, this method returns the type of the first item found.

item

Item specifier.

Returns:

A string, giving the item type. This can be one of “**arc**”, “**bitmap**”, “**image**”, “**line**”, “**oval**”, “**polygon**”, “**rectangle**”, “**text**”, or “**window**”.

xview(how, *args) [<#>]

Adjusts the canvas view horizontally.

how

How to adjust the canvas. This can be either “moveto” or “scroll”.

**args*

Additional arguments. For the “moveto” method, this is a single fraction. For the “scroll” method, this is a unit and a count. For details, see the descriptions of the [xview moveto](#) and [xview scroll](#) methods.

xview_moveto(fraction) [<#>]

Adjusts the canvas so that the given offset is at the left edge of the canvas.

fraction

Scroll offset. Offset 0.0 is the beginning of the **scrollregion**, 1.0 the end.

xview_scroll(number, what) [<#>]

Scrolls the canvas horizontally by the given amount.

number

Number of units.

what

What unit to use. This can be either “**units**” (small steps) or “**pages**”.

yview(how, *args) [<#>]

Adjusts the canvas view vertically.

how

How to adjust the canvas. This can be either “moveto” or “scroll”.

**args*

Additional arguments. For the “moveto” method, this is a single fraction. For the “scroll” method, this is a unit and a count. For details, see the descriptions of the [yview moveto](#) and [yview scroll](#) methods.

yview_moveto(fraction) [<#>]

Adjusts the canvas so that the given offset is at the top edge of the canvas.

fraction

Scroll offset. Offset 0.0 is the beginning of the **scrollregion**, 1.0 the end.

yview_scroll(number, what) [<#>]

Scrolls the canvas vertically by the given amount.

number

Number of units.

what

What unit to use. This can be either “**units**” (small steps) or “**pages**”.

[back](#) [next](#)

[back](#) [next](#)

The Tkinter Checkbutton Widget

The **Checkbutton** widget is a standard Tkinter widgets used to implement on-off selections. Checkbuttons can contain text or images, and you can associate a Python function or method with each button. When the button is pressed, Tkinter calls that function or method.

The button can only display text in a single font, but the text may span more than one line. In addition, one of the characters can be underlined, for example to mark a keyboard shortcut. By default, the **T**ab key can be used to move to a button widget.

Each Checkbutton widget should be associated with a variable.

When to use the Checkbutton Widget

The checkbutton widget is used to choose between two distinct values (usually switching something on or off). Groups of checkbuttons can be used to implement “many-of-many” selections.

To handle “one-of-many” choices, use [Radiobutton](#) and [Listbox](#) widgets.

Patterns

(Also see the [Button](#) patterns).

To use a **Checkbutton**, you must create a Tkinter variable. To inspect the button state, query the variable.

```
from Tkinter import *

master = Tk()

var = IntVar()

c = Checkbutton(master, text="Expand", variable=var)
c.pack()

mainloop()
```

By default, the variable is set to 1 if the button is selected, and 0 otherwise. You can change these values using the **onvalue** and **offvalue** options. The variable doesn't have to be an integer variable:

```
var = StringVar()
c = Checkbutton(
    master, text="Color image", variable=var,
    onvalue="RGB", offvalue="L"
)
```

If you need to keep track of both the variable and the widget, you can simplify your code somewhat by attaching the variable to the widget reference object.

```
v = IntVar()
c = Checkbutton(master, text="Don't show this again", variable=v)
c.var = v
```

If your Tkinter code is already placed in a class (as it should be), it is probably cleaner to store the variable in an attribute, and use a bound method as callback:

```
def __init__(self, master):
    self.var = IntVar()
    c = Checkbutton(
        master, text="Enable Tab",
        variable=self.var,
```



```

        command=self.cb)
    c.pack()

def cb(self, event):
    print "variable is", self.var.get()

```

Reference

Checkbutton(master=None, **options) (class) [<#>]

A toggle button.

master

Parent widget.

***options*

Widget options. See the description of the [config](#) method for a list of available options.

config(options)** [<#>]

Modifies one or more widget options. If no options are given, the method returns a dictionary containing all current option values.

***options*

Widget options.

activebackground=

The background color to use when the button is activated. Default value is system specific. (the database name is `activeBackground`, the class is `Foreground`)

activeforeground=

The foreground color to use when the button is activated. Default value is system specific. (`activeForeground/Background`)

anchor=

Controls where in the button the text (or image) should be located. Use one of **N**, **NE**, **E**, **SE**, **S**, **SW**, **W**, **NW**, or **CENTER**. Default is **CENTER**. If you change this, it is probably a good idea to add some padding as well, using the **padx** and/or **pady** options. (`anchor/Anchor`)

background=

The button background color. The default is system specific. (`background/Background`)

bg=

Same as **background**.

bitmap=

The bitmap to display in the widget. If the **image** option is given, this option is ignored. You can either use a built-in bitmap, or load a bitmap from an XBM file. To load the bitmap from file, just prefix the filename with an at-sign (for example `@sample.xbm`). (`bitmap/Bitmap`)

borderwidth=

The width of the button border. The default is system specific, but is usually one or two pixels. (`borderWidth/BorderWidth`)

bd=

Same as **borderwidth**.

command=

A function or method that is called when the button is pressed. The callback can be a function, bound method, or any other callable Python object. No default. (`command/Command`)

compound=

Default is **NONE**. (`compound/Compound`)

cursor=

The cursor to show when the mouse pointer is moved over the button. (`cursor/Cursor`)

disabledforeground=

The color to use when the button is disabled. The background is shown in the **background** color. (`disabledForeground/DisabledForeground`)

font=
The font to use in the button. The button can only contain text in a single font. The default is system specific. (font/Font)

foreground=
The button foreground color. The default is system specific. (foreground/Foreground)

fg=
Same as **foreground**.

height=
The size of the button. If the button displays text, the size is given in text units. If the button displays an image, the size is given in pixels (or screen units). If the size is omitted, it is calculated based on the button contents. (height/Height)

highlightbackground=
Default value is system specific. (highlightBackground/HighlightBackground)

highlightcolor=
Default value is system specific. (highlightColor/HighlightColor)

highlightthickness=
Default value is 1. (highlightThickness/HighlightThickness)

image=
The image to display in the widget. If specified, this takes precedence over the **text** and **bitmap** options. No default. (image/Image)

indicatoron=
Controls if the indicator should be drawn or not. This is on by default. Setting this option to false means that the relief will be used as the indicator. If the button is selected, it is drawn as **SUNKEN** instead of **RAISED**. (indicatorOn/IndicatorOn)

justify=
Defines how to align multiple lines of text. Use **LEFT**, **RIGHT**, or **CENTER** (default). (justify/Justify)

offrelief=
Default is raised. (offRelief/OffRelief)

offvalue=
The value corresponding to a non-checked button. The default is 0. (offValue/Value)

onvalue=
The value corresponding to a checked button. The default is 1. (onValue/Value)

overrelief=
No default value. (overRelief/OverRelief)

padx=
Button padding. Default value is 1. (padX/Pad)

pady=
Button padding. Default value is 1. (padY/Pad)

relief=
Border decoration. This is usually **FLAT** for checkbuttons, unless they use the border as indicator (via the **indicatoron** option). (relief/Relief)

selectcolor=
Color to use for the selector. Default value is system specific. (selectColor/Background)

selectimage=
Graphic image to use for the selector. No default. (selectImage/SelectImage)

state=
Button state. One of **NORMAL**, **ACTIVE** or **DISABLED**. Default is **NORMAL**. (state/State)

takefocus=
Indicates that the user can use the **Tab** key to move to this button. Default is an empty string, which means that the button accepts focus only if it has any keyboard bindings (default is on, in other words). (takeFocus/TakeFocus)

text=
The text to display in the button. The text can contain newlines. If the **bitmap** or **image** options are used, this option is ignored. (text/Text)

textvariable=

Associates a Tkinter variable (usually a **StringVar**) with the button. If the variable is changed, the button text is updated.

Also see the **variable** option. (textVariable/Variable)

underline=

Which character to underline, if any. Default value is -1 (no underline). (underline/Underline)

variable=

Associates a Tkinter variable to the button. When the button is pressed, the variable is toggled between **offvalue** and **onvalue**. Explicit changes to the variable are automatically reflected by the buttons.

(variable/Variable)

width=

The size of the button. See **height** for details. (width/Width)

wraplength=

Determines when a button's text should be wrapped into multiple lines. This is given in screen units. Default is no wrapping.

(wrapLength/WrapLength)

deselect() [<#>]

Deselects the checkbox; that is, sets the value to **offvalue**.

flash() [<#>]

Redraws the button several times, alternating between active and normal appearance.

invoke() [<#>]

Calls the command associated with the button.

select() [<#>]

Selects the button; that is, sets the value to **onvalue**.

toggle() [<#>]

Toggles the button.

[back](#) [next](#)

 rendered by a [django](#) application. hosted by [webfaction](#).

[back](#) [next](#)

The Tkinter Entry Widget

The **Entry** widget is a standard Tkinter widget used to enter or display a single line of text.

When to use the Entry Widget

The entry widget is used to enter text strings. This widget allows the user to enter one line of text, in a single font.

To enter multiple lines of text, use the [Text](#) widget.

Patterns

To add entry text to the widget, use the **insert** method. To replace the current text, you can call **delete** before you insert the new text.

```
e = Entry(master)
e.pack()

e.delete(0, END)
e.insert(0, "a default value")
```

To fetch the current entry text, use the **get** method:

```
s = e.get()
```

You can also bind the entry widget to a **StringVar** instance, and set or get the entry text via that variable:

```
v = StringVar()
e = Entry(master, textvariable=v)
e.pack()

v.set("a default value")
s = v.get()
```

This example creates an Entry widget, and a Button that prints the current contents:

```
from Tkinter import *

master = Tk()

e = Entry(master)
e.pack()

e.focus_set()

def callback():
    print e.get()

b = Button(master, text="get", width=10, command=callback)
b.pack()

mainloop()

e = Entry(master, width=50)
e.pack()

text = e.get()

def makeentry(parent, caption, width=None, **options):
    Label(parent, text=caption).pack(side=LEFT)
    entry = Entry(parent, **options)
    if width:
        entry.config(width=width)
    entry.pack(side=LEFT)
    return entry
```

```

user = makeentry(parent, "User name:", 10)
password = makeentry(parent, "Password:", 10, show="*")

content = StringVar()
entry = Entry(parent, text=caption, textvariable=content)

text = content.get()
content.set(text)

```

FIXME: More patterns to be added.

In newer versions, the Entry widget supports custom events. Document them, and add examples showing how to bind them.

Add [ValidateEntry](#) subclass as an example?

Concepts

Indexes

The *Entry* widget allows you to specify character positions in a number of ways:

- Numerical indexes
- **ANCHOR**
- **END**
- **INSERT**
- Mouse coordinates (“@x”)

Numerical indexes work just like Python list indexes. The characters in the string are numbered from 0 and upwards. You specify ranges just like you slice lists in Python: for example, (0, 5) corresponds to the first five characters in the entry widget.

ANCHOR (or the string “anchor”) corresponds to the start of the selection, if any. You can use the **select_from** method to change this from the program.

END (or “end”) corresponds to the position just after the last character in the entry widget. The range (0, END) corresponds to all characters in the widget.

INSERT (or “insert”) corresponds to the current position of the text cursor. You can use the **icursor** method to change this from the program.

Finally, you can use the mouse position for the index, using the following syntax:

```
"@%d" % x
```

where *x* is given in pixels relative to the left edge of the entry widget.

Reference

Entry(master=None, **options) (class) [<#>]

A text entry field.

master

Parent widget.

***options*

Widget options. See the description of the [config](#) method for a list of available options.

config(options)** [<#>]

Modifies one or more widget options. If no options are given, the method returns a dictionary containing all current option values.

***options*

Widget options.

background=
Widget background. The default is system specific. (the option database name is `background`, the class is `Background`)

bg=
Same as **background**.

borderwidth=
Border width. The default is system specific, but is usually a few pixels. (`borderWidth/BorderWidth`)

bd=
Same as **borderwidth**.

cursor=
Widget cursor. The default is a text insertion cursor (typically an “I-beam” cursor, e.g. **xterm**). (`cursor/Cursor`)

disabledbackground=
Background to use when the widget is disabled. If omitted or blank, the standard background is used instead. (`disabledBackground/DisabledBackground`)

disabledforeground=
Text color to use when the widget is disabled. If omitted or blank, the standard foreground is used instead. (`disabledForeground/DisabledForeground`)

exportselection=
If true, selected text is automatically exported to the clipboard. Default is true. (`exportSelection/ExportSelection`)

font=
Widget font. The default is system specific. (`font/Font`)

foreground=
Text color. (`foreground/Foreground`)

fg=
Same as **foreground**.

highlightbackground=
Together with **highlightcolor**, this option controls how to draw the focus highlight border. This option is used when the widget doesn’t have focus. The default is system specific. (`highlightBackground/HighlightBackground`)

highlightcolor=
Same as **highlightbackground**, but is used when the widget has focus. (`highlightColor/HighlightColor`)

highlightthickness=
The width of the focus highlight border. Default is typically a few pixels, unless the system indicates focus by modifying the button itself (like on Windows). (`highlightThickness/HighlightThickness`)

insertbackground=
Color used for the insertion cursor. (`insertBackground/Foreground`)

insertborderwidth=
Width of the insertion cursor’s border. If this is set to a non-zero value, the cursor is drawn using the **RAISED** border style. (`insertBorderWidth/BorderWidth`)

insertofftime=
Together with **insertontime**, this option controls cursor blinking. Both values are given in milliseconds. (`insertOffTime/OffTime`)

insertontime=
See **insertofftime**. (`insertOnTime/OnTime`)

insertwidth=
Width of the insertion cursor. Usually one or two pixels. (`insertWidth/InsertWidth`)

invalidcommand=
FIXME. No default. (`invalidCommand/InvalidCommand`)

invcmd=
Same as **invalidcommand**.

justify=
How to align the text inside the entry field. Use one of **LEFT**, **CENTER**, or **RIGHT**. The default is **LEFT**. (`justify/Justify`)

readonlybackground=
The background color to use when the state is “readonly”. If omitted or

blank, the standard background is used instead.
(readonlyBackground/ReadOnlyBackground)

relief=
Border style. The default is **SUNKEN**. Other possible values are **FLAT**, **RAISED**, **GROOVE**, and **RIDGE**. (relief/Relief)

selectbackground=
Selection background color. The default is system and display specific.
(selectBackground/Foreground)

selectborderwidth=
Selection border width. The default is system specific.
(selectBorderWidth/BorderWidth)

selectforeground=
Selection text color. The default is system specific.
(selectForeground/Background)

show=
Controls how to display the contents of the widget. If non-empty, the widget displays a string of characters instead of the actual contents. To get a password entry widget, set this option to “*”. (show/Show)

state=
The entry state: **NORMAL**, **DISABLED**, or “readonly” (same as **DISABLED**, but contents can still be selected and copied). Default is **NORMAL**. Note that if you set this to **DISABLED** or “readonly”, calls to **insert** and **delete** are ignored. (state/State)

takefocus=
Indicates that the user can use the **Tab** key to move to this widget. Default is an empty string, which means that the entry widget accepts focus only if it has any keyboard bindings (default is on, in other words).
(takeFocus/TakeFocus)

textvariable=
Associates a Tkinter variable (usually a **StringVar**) to the contents of the entry field. (textVariable/Variable)

validate=
Specifies when validation should be done. You can use “focus” to validate whenever the widget gets or loses the focus, “focusin” to validate only when it gets focus, “focusout” to validate when it loses focus, “key” on any modification, and ALL for all situations. Default is **NONE** (no validation). (validate/Validate)

validatecommand=
A function or method to call to check if the contents is valid. The function should return a true value if the new contents is valid, or false if it isn’t. Note that this option is only used if the **validate** option is not **NONE**.
(validateCommand/ValidateCommand)

vcmd=
Same as **validatecommand**.

width=
Width of the entry field, in character units. Note that this controls the size on screen; it does not limit the number of characters that can be typed into the entry field. The default width is 20 character.
(width/Width)

xscrollcommand=
Used to connect an entry field to a horizontal scrollbar. This option should be set to the **set** method of the corresponding scrollbar.
(xScrollCommand/ScrollCommand)

delete(first, last=None) [#]

Deletes the character at index, or within the given range. Use delete(0, END) to delete all text in the widget.

first

Start of range.

last

Optional end of range. If omitted, only a single character is removed.

get() [<#>]

Gets the current contents of the entry field.

Returns:

The widget contents, as a string.

icursor(index) [<#>]

Moves the insertion cursor to the given index. This also sets the **INSERT** index.

index

Where to move the cursor.

index(index) [<#>]

Gets the numerical position corresponding to the given index.

index

An index.

Returns:

The corresponding numerical index.

insert(index, string) [<#>]

Inserts text at the given index. Use `insert(INSERT, text)` to insert text at the cursor, `insert(END, text)` to append text to the widget.

index

Where to insert the text.

string

The text to insert.

scan_dragto(x) [<#>]

Sets the scanning anchor for fast horizontal scrolling to the given mouse coordinate.

x

Current horizontal mouse position.

scan_mark(x) [<#>]

Scrolls the widget contents sideways according to the given mouse coordinate. The text is moved 10 times the distance between the scanning anchor and the new position.

x

Current horizontal mouse position.

select_adjust(index) [<#>]

Same as [selection_adjust](#).

select_clear() [<#>]

Same as [selection_clear](#).

select_from(index) [<#>]

Same as [selection_from](#).

select_present() [<#>]

Same as [selection_present](#).

select_range(start, end) [<#>]

Same as [selection_range](#).

select_to(index) [<#>]

Same as [selection_to](#).

selection_adjust(index) [<#>]

Adjusts the selection to include also the given character. If index is already selected, do nothing.

index

The index.

selection_clear() [<#>]

Clears the selection.

selection_from(index) [<#>]

Starts a new selection. This also sets the **ANCHOR** index.

index

The index.

selection_present() [<#>]

Checks if text is selected.

Returns:

A true value if some part of the text is selected.

selection_range(start, end) [<#>]

Explicitly sets the selection range. Start must be smaller than end. Use **selection_range(o, END)** to select all text in the widget.

start

Start of selection range.

end

End of range.

selection_to(index) [<#>]

Selects all text between **ANCHOR** and the given index.

index

xview(index) [<#>]

Makes sure the given index is visible. The entry view is scrolled if necessary.

index

An index.

xview_moveto(fraction) [<#>]

Adjusts the entry view so that the given offset is at the left edge of the canvas. Offset 0.0 is the beginning of the entry string, 1.0 the end.

fraction

xview_scroll(number, what) [<#>]

Scrolls the entry view horizontally by the given amount.

number

Number of units.

what

What unit to use. This can be either “**units**” (characters) or “**pages**” (larger steps).

[back](#) [next](#)



rendered by a [django](#) application. hosted by [webfaction](#).

[back](#) [next](#)

The Tkinter Frame Widget

A frame is rectangular region on the screen. The frame widget is mainly used as a geometry master for other widgets, or to provide padding between other widgets.

When to use the Frame Widget

Frame widgets are used to group other widgets into complex layouts. They are also used for padding, and as a base class when implementing compound widgets.

Patterns

The frame widget can be used for decorations:

```
from Tkinter import *

master = Tk()

Label(text="one").pack()

separator = Frame(height=2, bd=1, relief=SUNKEN)
separator.pack(fill=X, padx=5, pady=5)

Label(text="two").pack()

mainloop()
```

The frame widget can be used as a place holder for video overlays and other external processes.

To use a frame widget in this fashion, set the background color to an empty string (this prevents updates, and leaves the color map alone), pack it as usual, and use the **window_id** method to get the window handle corresponding to the frame.

```
frame = Frame(width=768, height=576, bg="", colormap="new")
frame.pack()

video.attach_window(frame.window_id())
```

FIXME: add more patterns: gridded group, compound widget pattern

Reference

Frame(master=None, **options) (class) [<#>]

A widget container.

master

Parent widget.

***options*

Widget options. See the description of the [config](#) method for a list of available options.

config(options)** [<#>]

Modifies one or more widget options. If no options are given, the method returns a dictionary containing all current option values.

***options*

Widget options.

background=

The background color to use in this frame. This defaults to the

application background color. To prevent updates, set the color to an empty string. (the option database name is background, the class is Background)

bg=

Same as **background**.

borderwidth=

Border width. Defaults to 0 (no border). (borderWidth/BorderWidth)

bd=

Same as **borderwidth**.

class=

Default is Frame. (class/Class)

colormap=

Some displays support only 256 colors (some use even less). Such displays usually provide a color map to specify which 256 colors to use. This option allows you to specify which color map to use for this frame, and its child widgets.
By default, a new frame uses the same color map as its parent. Using this option, you can reuse the color map of another window instead (this window must be on the same screen and have the same visual characteristics). You can also use the value "new" to allocate a new color map for this frame.
You cannot change this option once you've created the frame.
(colormap/Colormap)

container=

Default is 0. (container/Container)

cursor=

The cursor to show when the mouse pointer is placed over this widget.
Default is a system specific arrow cursor. (cursor/Cursor)

height=

Default is 0. (height/Height)

highlightbackground=

Default is system specific. (highlightBackground/HighlightBackground)

highlightcolor=

Default value is system specific. (highlightColor/HighlightColor)

highlightthickness=

Default is 0. (highlightThickness/HighlightThickness)

padx=

Horizontal padding. Default is 0. (padX/Pad)

pady=

Vertical padding. Default is 0. (padY/Pad)

relief=

Border decoration. The default is **FLAT**. Other possible values are **SUNKEN**, **RAISED**, **GROOVE**, and **RIDGE**.
Note that to show the border, you need to change the **borderwidth** from its default value of 0. (relief/Relief)

takefocus=

If true, the user can use the **Tab** key to move to this widget. The default value is 0. (takeFocus/TakeFocus)

visual=

No default value. (visual/Visual)

width=

Default value is 0. (width/Width)

[back](#) [next](#)



rendered by a [django](#) application. hosted by [webfaction](#).

[back](#) [next](#)

The Tkinter Label Widget

The **Label** widget is a standard Tkinter widget used to display a text or image on the screen. The label can only display text in a single font, but the text may span more than one line. In addition, one of the characters can be underlined, for example to mark a keyboard shortcut.

When to use the Label Widget

Labels are used to display texts and images. The label widget uses double buffering, so you can update the contents at any time, without annoying flicker.

To display data that the user can manipulate in place, it's probably easier to use the [Canvas](#) widget.

Patterns

To use a label, you just have to specify what to display in it (this can be text, a bitmap, or an image):

```
from Tkinter import *

master = Tk()

w = Label(master, text="Hello, world!")
w.pack()

mainloop()
```

If you don't specify a size, the label is made just large enough to hold its contents. You can also use the **height** and **width** options to explicitly set the size. If you display text in the label, these options define the size of the label in text units. If you display bitmaps or images instead, they define the size in pixels (or other screen units). See the [Button](#) description for an example how to specify the size in pixels also for text labels.

You can specify which color to use for the label with the **foreground** (or **fg**) and **background** (or **bg**) options. You can also choose which font to use in the label (the following example uses Tk 8.0 font descriptors). Use colors and fonts sparingly; unless you have a good reason to do otherwise, you should stick to the default values.

```
w = Label(master, text="Rouge", fg="red")
w = Label(master, text="Helvetica", font=("Helvetica", 16))
```

Labels can display multiple lines of text. You can use newlines or use the **wraplength** option to make the label wrap text by itself. When wrapping text, you might wish to use the **anchor** and **justify** options to make things look exactly as you wish. An example:

```
w = Label(master, text=longtext, anchor=W, justify=LEFT)
```

You can associate a Tkinter variable with a label. When the contents of the variable changes, the label is automatically updated:

```
v = StringVar()
Label(master, textvariable=v).pack()

v.set("New Text!")
```

You can use the label to display [PhotoImage](#) and [BitmapImage](#) objects. When doing this, make sure you keep a reference to the image object, to prevent it from being garbage collected by Python's memory allocator. You can use a global variable or an instance attribute, or easier, just add an attribute to the widget instance:

```
photo = PhotoImage(file="icon.gif")
w = Label(parent, image=photo)
```

```
w.photo = photo
w.pack()
```

Reference

Label(master=None, **options) (class) [<#>]

Display a single line of text, or an image.

master

Parent widget.

****options**

Widget options. See the description of the [config](#) method for a list of available options.

config(options)** [<#>]

Modifies one or more widget options. If no options are given, the method returns a dictionary containing all current option values.

****options**

Widget options.

activebackground=

What background color to use when the label is active (set with the state option). The default is platform specific. (the option database name is activeBackground, the class is Foreground)

activeforeground=

What foreground color to use when the label is active. The default is platform specific. (activeForeground/Background)

anchor=

Controls where in the label the text (or image) should be located. Use one of **N**, **NE**, **E**, **SE**, **S**, **SW**, **W**, **NW**, or **CENTER**. Default is **CENTER**. (anchor/Anchor)

background=

The background color. The default is platform specific. (background/Background)

bg=

Same as **background**.

bitmap=

The bitmap to display in the widget. If the **image** option is given, this option is ignored. (bitmap/Bitmap)

borderwidth=

The width of the label border. The default is system specific, but is usually one or two pixels. (borderWidth/BorderWidth)

bd=

Same as **borderwidth**.

compound=

Controls how to combine text and image in the label. By default, if an image or bitmap is given, it is drawn instead of the text. If this option is set to **CENTER**, the text is drawn on top of the image. If this option is set to one of **BOTTOM**, **LEFT**, **RIGHT**, or **TOP**, the image is drawn besides the text (use **BOTTOM** to draw the image under the text, etc.). Default is **NONE**. (compound/Compound)

cursor=

What cursor to show when the mouse is moved over the label. The default is to use the standard cursor. (cursor/Cursor)

disabledforeground=

What foreground color to use when the label is disabled. The default is system specific. (disabledForeground/DisabledForeground)

font=

The font to use in the label. The label can only contain text in single font. The default is system specific. (font/Font)

foreground=

The label color, used for for text and bitmap labels. The default is system

specific. (foreground/Foreground)

fg= Same as **foreground**.

height= The height of the label. If the label displays text, the size is given in text units. If the label displays an image, the size is given in pixels (or screen units). If the size is set to 0, or omitted, it is calculated based on the label contents. (height/Height)

highlightbackground= What color to use for the highlight border when the widget does not have focus. The default is system specific, but usually the same as the standard background color. (highlightBackground/HighlightBackground)

highlightcolor= What color to use for the highlight border when the widget has focus. The default is system specific. (highlightColor/HighlightColor)

highlightthickness= The width of the highlight border. The default is 0 (no highlight border). (highlightThickness/HighlightThickness)

image= The image to display in the widget. The value should be a PhotoImage, BitmapImage, or a compatible object. If specified, this takes precedence over the **text** and **bitmap** options. (image/Image)

justify= Defines how to align multiple lines of text. Use **LEFT**, **RIGHT**, or **CENTER**. Note that to position the text inside the widget, use the **anchor** option. Default is **CENTER**. (justify/Justify)

padx= Extra horizontal padding to add around the text. The default is 1 pixel. (padX/Pad)

pady= Extra vertical padding to add around the text. The default is 1 pixel. (padY/Pad)

relief= Border decoration. The default is **FLAT**. Other possible values are **SUNKEN**, **RAISED**, **GROOVE**, and **RIDGE**. (relief/Relief)

state= Label state. This option controls how the label is rendered. The default is **NORMAL**. Other possible values are **ACTIVE** and **DISABLED**. (state/State)

takefocus= If true, the widget accepts input focus. The default is false. (takeFocus/TakeFocus)

text= The text to display in the label. The text can contain newlines. If the **bitmap** or **image** options are used, this option is ignored. (text/Text)

textvariable= Associates a Tkinter variable (usually a **StringVar**) with the label. If the variable is changed, the label text is updated. (textVariable/Variable)

underline= Used with the **text** option to indicate that a character should be underlined (e.g. for keyboard shortcuts). Default is -1 (no underline). (underline/Underline)

width= The width of the label. If the label displays text, the size is given in text units. If the label displays an image, the size is given in pixels (or screen units). If the size is set to 0, or omitted, it is calculated based on the label contents. (width/Width)

wraplength= Determines when a label's text should be wrapped into multiple lines. This is given in screen units. Default is 0 (no wrapping). (wrapLength/WrapLength)

[back](#) [next](#)



rendered by a [django](#) application. hosted by [webfaction](#).

[back](#) [next](#)

The Tkinter LabelFrame Widget

(new in Tk 8.4) The **LabelFrame** widget is a variant of the Tkinter Frame widget. By default, it draws a border around its child widgets, and it can also display a title.

When to use the LabelFrame Widget

The **LabelFrame** can be used when you want to group a number of related widgets, such as a number of radiobuttons.

Patterns

To display a group, create a LabelFrame, and add child widgets to the frame as usual. The widget draws a border around the children, and a text label above them.

```
from Tkinter import *

master = Tk()

group = LabelFrame(master, text="Group", padx=5, pady=5)
group.pack(padx=10, pady=10)

w = Entry(group)
w.pack()

mainloop()
```

You can use options to control where and how to draw the label, and how to draw the border. See below for details.

Reference

LabelFrame(master=None, **options) (class) [<#>]

A frame widget with an internal group border and an optional label.

master

Parent widget.

***options*

Widget options. See the description of the config method for a list of available options.

config(options)** [<#>]

Modifies one or more widget options. If no options are given, the method returns a dictionary containing all current option values.

***options*

Widget options.

background=

The background color to use in this frame. This defaults to the application background color. To prevent updates, set the color to an empty string. (the option database name is background, the class is Background)

bd=

Same as **borderwidth**.

bg=

Same as **background**.

borderwidth=

Border width. Defaults to 2 pixels. (borderWidth/BorderWidth)

class=

Default is LabelFrame. (class/Class)

colormap=

Some displays support only 256 colors (some use even less). Such displays usually provide a color map to specify which 256 colors to use. This option allows you to specify which color map to use for this frame, and its child widgets.

By default, a new frame uses the same color map as its parent. Using this option, you can reuse the color map of another window instead (this window must be on the same screen and have the same visual characteristics). You can also use the value “new” to allocate a new color map for this frame.

You cannot change this option once you’ve created the frame.

(colormap/Colormap)

container=

If true, this frame is a container widget. Defaults to false.

You cannot change this option once you’ve created the frame.

(container/Container)

cursor=

The cursor to show when the mouse pointer is placed over this widget.

Default is a system specific arrow cursor. (cursor/Cursor)

fg=

Same as **foreground**.

font=

Font to use for the label text. The default is system specific. (font/Font)

foreground=

Color to use for the label text. The default is system specific.

(foreground/Foreground)

height=

Frame height, in pixels. No default value. (height/Height)

highlightbackground=

Together with **highlightcolor**, this option controls how to draw the focus highlight border. This option is used when the widget doesn’t have focus. The default is system specific.

(highlightBackground/HighlightBackground)

highlightcolor=

Same as **highlightbackground**, but is used when the widget has focus.

(highlightColor/HighlightColor)

highlightthickness=

The width of the focus highlight border. The default is typically a few pixels. (highlightThickness/HighlightThickness)

labelanchor=

Where to draw the label text. Default is NW (upper left corner).

(labelAnchor/LabelAnchor)

labelwidget=

Widget to use for the label. If omitted, the frame uses the **text** option. No default value. (labelWidget/LabelWidget)

padx=

Horizontal internal padding. Default is 0. (padX/Pad)

pady=

Vertical internal padding. Default is 0. (padY/Pad)

relief=

Border style. The default is **GROOVE**. Other possible values are **FLAT**, **RAISED**, **SUNKEN**, and **RIDGE**. (relief/Relief)

takefocus=

If true, indicates that the user can use the **Tab** key to move to this widget.

Default is 0 (false). (takeFocus/TakeFocus)

text=

The label text. (text/Text)

visual=

Screen visual. No default value. (visual/Visual)

width=

Frame width. No default value. (width/Width)

[back](#) [next](#)



rendered by a [django](#) application. hosted by [webfaction](#).

[back](#) [next](#)

The Tkinter Listbox Widget

The **Listbox** widget is a standard Tkinter widget used to display a list of alternatives. The listbox can only contain text items, and all items must have the same font and color. Depending on the widget configuration, the user can choose one or more alternatives from the list.

When to use the Listbox Widget

Listboxes are used to select from a group of textual items. Depending on how the listbox is configured, the user can select one or many items from that list.

Patterns

When you first create the listbox, it is empty. The first thing to do is usually to insert one or more lines of text. The **insert** method takes an index and a string to insert. The index is usually an item number (0 for the first item in the list), but you can also use some special indexes, including **ACTIVE**, which refers to the “active” item (set when you click on an item, or by the arrow keys), and **END**, which is used to append items to the list.

```
from Tkinter import *

master = Tk()

listbox = Listbox(master)
listbox.pack()

listbox.insert(END, "a list entry")

for item in ["one", "two", "three", "four"]:
    listbox.insert(END, item)

mainloop()
```

To remove items from the list, use the **delete** method. The most common operation is to delete all items in the list (something you often need to do when updating the list).

```
listbox.delete(0, END)
listbox.insert(END, newitem)
```

You can also delete individual items. In the following example, a separate button is used to delete the **ACTIVE** item from a list.

```
lb = Listbox(master)
b = Button(master, text="Delete",
            command=lambda lb=lb: lb.delete(ANCHOR))
```

The listbox offers four different selection modes through the **selectmode** option. These are **SINGLE** (just a single choice), **BROWSE** (same, but the selection can be moved using the mouse), **MULTIPLE** (multiple item can be chosen, by clicking at them one at a time), or **EXTENDED** (multiple ranges of items can be chosen, using the **Shift** and **Control** keyboard modifiers). The default is **BROWSE**. Use **MULTIPLE** to get “checkboxlist” behavior, and **EXTENDED** when the user would usually pick only one item, but sometimes would like to select one or more ranges of items.

```
lb = Listbox(selectmode=EXTENDED)
```

To query the selection, use **curselection** method. It returns a list of item indexes, but a bug in Tkinter 1.160 (Python 2.2) and earlier versions causes this list to be returned as a list of strings, instead of integers. This may be fixed in later versions of Tkinter, so you should make sure that your code is written to handle either case. Here’s one way to do that:

```
items = map(int, list.curselection())
```

In versions before Python 1.5, use **string.atoi** instead of **int**.

Use the **get** method to get the list item corresponding to a given index. Note that **get** accepts either strings or integers, so you don't have to convert the indexes to integers if all you're going to do is to pass them to **get**.

You can also use a listbox to represent arbitrary Python objects. In the next example, we assume that the input data is represented as a list of tuples, where the first item in each tuple is the string to display in the list. For example, you could display a dictionary by using the **items** method to get such a list.

```
self.lb.delete(0, END) # clear
for key, value in data:
    self.lb.insert(END, key)
self.data = data
```

When querying the list, simply fetch the items from the **data** attribute, using the selection as an index:

```
items = self.lb.curselection()
items = [self.data[int(item)] for item in items]
```

In earlier versions of Python, you can use this instead:

```
items = self.lb.curselection()
try:
    items = map(int, items)
except ValueError: pass
items = map(lambda i,d=self.data: d[i], items)
```

Unfortunately, the listbox doesn't provide a **command** option allowing you to track changes to the selection. The standard solution is to bind a *double-click* event to the same callback as the OK (or Select, or whatever) button. This allows the user to either select an alternative as usual, and click OK to carry out the operation, or to select and carry out the operation in one go by double-clicking on an alternative. This solution works best in **BROWSE** and **EXTENDED** modes.

```
lb.bind("<Double-Button-1>", self.ok)
```

FIXME: show how to use bindtags to insert custom bindings *after* the standard bindings

FIXME: show how to use custom events in later versions of Tkinter

If you wish to track arbitrary changes to the selection, you can either rebind the whole bunch of selection related events (see the Tk manual pages for a complete list of Listbox event bindings), or, much easier, poll the list using a timer:

```
class Dialog(Frame):

    def __init__(self, master):
        Frame.__init__(self, master)
        self.list = Listbox(self, selectmode=EXTENDED)
        self.list.pack(fill=BOTH, expand=1)
        self.current = None
        self.poll() # start polling the list

    def poll(self):
        now = self.list.curselection()
        if now != self.current:
            self.list_has_changed(now)
            self.current = now
            self.after(250, self.poll)

    def list_has_changed(self, selection):
        print "selection is", selection
```

By default, the selection is exported via the X selection mechanism (or the clipboard, on Windows). If you have more than one listbox on the screen, this really messes things up for the poor user. If she selects something in one listbox, and then selects something in another, the original selection disappears. It is usually a good idea to

disable this mechanism in such cases. In the following example, three listboxes are used in the same dialog:

```
b1 = Listbox(exportselection=0)
for item in families:
    b1.insert(END, item)

b2 = Listbox(exportselection=0)
for item in fonts:
    b2.insert(END, item)

b3 = Listbox(exportselection=0)
for item in styles:
    b3.insert(END, item)
```

The listbox itself doesn't include a scrollbar. Attaching a scrollbar is pretty straightforward. Simply set the **xscrollcommand** and **yscrollcommand** options of the listbox to the **set** method of the corresponding scrollbar, and the **command** options of the scrollbars to the corresponding **xview** and **yview** methods in the listbox. Also remember to pack the scrollbars before the listbox. In the following example, only a vertical scrollbar is used. For more examples, see pattern section in the **Scrollbar** description.

```
frame = Frame(master)
scrollbar = Scrollbar(frame, orient=VERTICAL)
listbox = Listbox(frame, yscrollcommand=scrollbar.set)
scrollbar.config(command=listbox.yview)
scrollbar.pack(side=RIGHT, fill=Y)
listbox.pack(side=LEFT, fill=BOTH, expand=1)
```

With some more trickery, you can use a single vertical scrollbar to scroll several lists in parallel. This assumes that all lists have the same number of items. Also note how the widgets are packed in the following example.

```
def __init__(self, master):
    scrollbar = Scrollbar(master, orient=VERTICAL)
    self.b1 = Listbox(master, yscrollcommand=scrollbar.set)
    self.b2 = Listbox(master, yscrollcommand=scrollbar.set)
    scrollbar.config(command=self.yview)
    scrollbar.pack(side=RIGHT, fill=Y)
    self.b1.pack(side=LEFT, fill=BOTH, expand=1)
    self.b2.pack(side=LEFT, fill=BOTH, expand=1)

def yview(self, *args):
    apply(self.b1.yview, args)
    apply(self.b2.yview, args)
```

Reference

Listbox(master=None, **options) (class) [<#>]

A scrolling listbox.

master

Parent widget.

****options**

Widget options. See the description of the [config](#) method for a list of available options.

activate(index) [<#>]

Activates the given index (it will be marked with an underline). The active item can be referred to using the **ACTIVE** index.

index

Index specifier.

bbox(self, index) [<#>]

Gets the bounding box of the given item text.

index

Index specifier.

Returns:

The bounding box, as a 4-tuple (xoffset, yoffset, width, height). If the item is not visible, this method returns None. If the item is partially visible, the box may extend outside the visible area.

config(options) [<#>]**

Modifies one or more widget options. If no options are given, the method returns a dictionary containing all current option values.

***options*

Widget options.

activestyle=

Default is underline. (the option database name is activeStyle, the class is ActiveStyle)

background=

Default value is 'SystemButtonFace'. (background/Background)

bg=

Same as **background**.

borderwidth=

Default value is 2. (borderWidth/BorderWidth)

bd=

Same as **borderwidth**.

cursor=

No default value. (cursor/Cursor)

disabledforeground=

Default is system specific. (disabledForeground/DisabledForeground)

exportselection=

Default value is 1. (exportSelection/ExportSelection)

font=

Default value is system specific. (font/Font)

foreground=

Default value is system specific. (foreground/Foreground)

fg=

Same as **foreground**.

height=

Default value is 10. (height/Height)

highlightbackground=

Default value is system specific.
(highlightBackground/HighlightBackground)

highlightcolor=

Default value is system specific. (highlightColor/HighlightColor)

highlightthickness=

Default value is 1. (highlightThickness/HighlightThickness)

listvariable=

No default value. (listVariable/Variable)

relief=

Default is SUNKEN. (relief/Relief)

selectbackground=

Default is system specific. (selectBackground/Foreground)

selectborderwidth=

Default is 1. (selectBorderWidth/BorderWidth)

selectforeground=

Default is system specific. (selectForeground/Background)

selectmode=

Default is BROWSE. (selectMode/SelectMode)

setgrid=

Default is 0. (setGrid/SetGrid)

state=
Default is NORMAL. (state/State)
takefocus=
No default value. (takeFocus/TakeFocus)
width=
Default is 20. (width/Width)
xscrollcommand=
No default value. (xScrollCommand/ScrollCommand)
yscrollcommand=
No default value. (yScrollCommand/ScrollCommand)

curselection() [<#>]

Gets a list of the currently selected alternatives. The list contains the indexes of the selected alternatives (beginning with 0 for the first alternative in the list). In most Python versions, the list contains strings instead of integers. Since this may change in future versions, you should make sure your code can handle either case. See the patterns section for a suggested solution.

Returns:
A list of index specifiers.

delete(first, last=None) [<#>]

Deletes one or more items. Use **delete(0, END)** to delete all items in the list.

first
First item to delete.
last
Last item to delete. If omitted, a single item is deleted.

get(first, last=None) [<#>]

Gets one or more items from the list. This function returns the string corresponding to the given index (or the strings in the given index range). Use **get(0, END)** to get a list of all items in the list. Use **get(ACTIVE)** to get the active (underlined) item.

first
First item to return.
last
Last item to return. If omitted, a single item is returned.

Returns:
A list of strings.

index(index) [<#>]

Returns the numerical index (0 to size()-1) corresponding to the given index. This is typically **ACTIVE**, but can also be **ANCHOR**, or a string having the form “@x,y” where x and y are widget-relative pixel coordinates.

index
Index specifier.
Returns:
Numerical index.

insert(index, *elements) [<#>]

Inserts one or more items at given index (this works as for Python lists; index 0 is before the first item). Use **END** to append items to the list. Use **ACTIVE** to insert items before the the active (underlined) item.

index

Index specifier.

**elements*

One or more elements to add.

itemcget(index, option) [<#>]

Gets a configuration option for an individual listbox item.

index

option

itemconfig(index, **options) [<#>]

Modifies the configuration for an individual listbox item.

index

***options*

itemconfig(index, **options) [<#>]

Same as [itemconfig](#).

nearest(y) [<#>]

Returns the index nearest to the given coordinate (a widget-relative pixel coordinate).

y

Coordinate.

Returns:

An index.

scan_dragto(x, y) [<#>]

Scrolls the widget contents according to the given mouse coordinate. The text is moved 10 times the distance between the scanning anchor and the new position.

x

Mouse coordinate.

y

Mouse coordinate.

scan_mark(x, y) [<#>]

Sets the scanning anchor for fast horizontal scrolling to the given mouse coordinate.

x

Mouse coordinate.

y

Mouse coordinate.

see(index) [<#>]

Makes sure the given list index is visible. You can use an integer index, or **END**.

index

Index specifier.

select_anchor(index) [<#>]

Same as [selection_anchor](#).

select_clear(first, last=None) [<#>]

Same as [selection_clear](#).

select_includes(index) [<#>]

Same as [selection_includes](#).

select_set(first, last=None) [<#>]

Same as [selection_set](#).

selection_anchor(index) [<#>]

Sets the selection anchor to the given index. The anchor can be referred to using the **ANCHOR** index.

index

Index specifier.

selection_clear(first, last=None) [<#>]

Removes one or more items from the selection.

first

First item to remove.

last

Last item to remove. If omitted, only one item is removed.

selection_includes(index) [<#>]

Checks if an item is selected.

index

Index specifier.

Returns:

A true value if the item is selected.

selection_set(first, last=None) [<#>]

Adds one or more items to the selection.

first

First item to add.

last

Last item to add. If omitted, only one item is added.

size() [<#>]

Returns the number of items in the list. The valid index range goes from 0 to size()-1.

Returns:

The number of items in this list.

xview(column, *extra) [<#>]

Controls horizontal scrolling.

If called without an argument, this method determines which part of the full list that is visible in the horizontal direction. This is given as the offset and size of the visible part, given in relation to the full size of the list (1.0 is the full list).

If called with a single argument, this method adjusts the list so that the given character column is at the left edge of the listbox.

If called with the string “moveto” and a fraction, this method behaves like [xview_moveto](#). If called with the string “scroll” and two more arguments, this method behaves like [xview_scroll](#).

column

The column to place at the left edge, or a string specifying what subcommand to execute.

**extra*

Additional arguments for the “moveto” and “scroll” forms. See above for details.

Returns:

If called without any arguments, a 2-tuple containing the left offset and the view size (relative to the full width).

xview_moveto(fraction) [<#>]

Adjusts the list so that the given offset is at the left (top) edge of the listbox. Offset 0.0 is the beginning of the list, 1.0 the end. These methods are used by the **Scrollbar** bindings when the user drags the scrollbar slider.

fraction

Offset.

xview_scroll(number, what) [<#>]

Scrolls the list view horizontally by the given amount.

number

Number of units.

what

What unit to use. This can be either “**units**” (characters) or “**pages**” (larger steps).

yview(*what) [<#>]

Controls vertical scrolling. This method works like [xview](#), but controls vertical scrolling.

To make sure that a given item is visible, use the [see](#) method.

index

The index to place at the top edge, or a string specifying what subcommand to execute.

**extra*

Additional arguments for the “moveto” and “scroll” forms.

Returns:

If called without any arguments, a 2-tuple containing the top offset and the view size (relative to the list size).

yview_moveto(fraction) [<#>]

Adjusts the list view so that the given offset is at the left edge of the canvas. Offset 0.0 is the beginning of the entry string, 1.0 the end.

fraction

Offset.

yview_scroll(number, what) [<#>]

Scrolls the list view vertically by the given amount.

number

Number of units.

what

What unit to use. This can be either “**units**” (characters) or “**pages**” (larger steps).

[back](#) [next](#)



rendered by a [django](#) application. hosted by [webfaction](#).

[back](#) [next](#)

The Tkinter Menu Widget

The **Menu** widget is used to implement toplevel, pulldown, and popup menus.

When to use the Menu Widget

This widget is used to display all kinds of menus used by an application. Since this widget uses native code where possible, you shouldn't try to fake menus using buttons and other Tkinter widgets.

Patterns

Toplevel menus are displayed just under the title bar of the root or any other toplevel windows (or on Macintosh, along the upper edge of the screen). To create a toplevel menu, create a new Menu instance, and use **add** methods to add commands and other menu entries to it.

```
root = Tk()

def hello():
    print "hello!"

# create a toplevel menu
menubar = Menu(root)
menubar.add_command(label="Hello!", command=hello)
menubar.add_command(label="Quit!", command=root.quit)

# display the menu
root.config(menu=menubar)
```

Pulldown menus (and other submenus) are created in a similar fashion. The main difference is that they are attached to a parent menu (using **add_cascade**), instead of a toplevel window.

```
root = Tk()

def hello():
    print "hello!"

menubar = Menu(root)

# create a pulldown menu, and add it to the menu bar
filemenu = Menu(menubar, tearoff=0)
filemenu.add_command(label="Open", command=hello)
filemenu.add_command(label="Save", command=hello)
filemenu.add_separator()
filemenu.add_command(label="Exit", command=root.quit)
menubar.add_cascade(label="File", menu=filemenu)

# create more pulldown menus
editmenu = Menu(menubar, tearoff=0)
editmenu.add_command(label="Cut", command=hello)
editmenu.add_command(label="Copy", command=hello)
editmenu.add_command(label="Paste", command=hello)
menubar.add_cascade(label="Edit", menu=editmenu)

helpmenu = Menu(menubar, tearoff=0)
helpmenu.add_command(label="About", command=hello)
menubar.add_cascade(label="Help", menu=helpmenu)

# display the menu
root.config(menu=menubar)
```

Finally, a popup menu is created in the same way, but is explicitly displayed, using the **post** method:

```
root = Tk()

def hello():
```

```

print "hello!"

# create a popup menu
menu = Menu(root, tearoff=0)
menu.add_command(label="Undo", command=hello)
menu.add_command(label="Redo", command=hello)

# create a canvas
frame = Frame(root, width=512, height=512)
frame.pack()

def popup(event):
    menu.post(event.x_root, event.y_root)

# attach popup to canvas
frame.bind("<Button-3>", popup)

```

You can use the **postcommand** callback to update (or even create) the menu every time it is displayed.

```

counter = 0

def update():
    global counter
    counter = counter + 1
    menu.entryconfig(0, label=str(counter))

root = Tk()

menubar = Menu(root)

menu = Menu(menubar, tearoff=0, postcommand=update)
menu.add_command(label=str(counter))
menu.add_command(label="Exit", command=root.quit)

menubar.add_cascade(label="Test", menu=menu)

root.config(menu=menubar)

```

Reference

Menu(master=None, **options) (class) [<#>]

A menu pane.

__init__(master=None, **options) [<#>]

Creates a menu widget.

master

Parent widget.

***options*

Widget options. See the description of the config method for a list of available options.

activate(index) [<#>]

The activate method.

index

add(type, **options) [<#>]

Add (append) an entry of the given type to the menu.

type

What kind of entry to add. Can be one of “**command**”, “**cascade**” (submenu), “**checkbutton**”, “**radiobutton**”, or “**separator**”.

***options*

Menu options.

activebackground=
activeforeground=
accelerator=
background=
bitmap=
columnbreak=
command=
font=
foreground=
hidemargin=
image=
indicatoron=
label=
menu=
offvalue=
onvalue=
selectcolor=
selectimage=
state=
underline=
value=
variable=

add_cascade(options) [<#>]**

Adds a submenu. See [add](#) for a list of options.

***options*

add_checkbutton(options) [<#>]**

Adds a checkbox. See [add](#) for a list of options.

***options*

add_command(options) [<#>]**

Adds a command. See [add](#) for a list of options.

***options*

add_radiobutton(options) [<#>]**

Adds a radiobutton. See [add](#) for a list of options.

***options*

add_separator(options) [<#>]**

Adds a separator. See [add](#) for a list of options.

***options*

config(options) [<#>]**

Modifies one or more widget options. If no options are given, the method returns a dictionary containing all current option values.

***options*

Widget options.

activebackground=

Default value is 'SystemHighlight'. (the database name is activeBackground, the class is Foreground)

activeborderwidth=
Default value is 0. (activeBorderWidth/BorderWidth)

activeforeground=
Default value is 'SystemHighlightText'. (activeForeground/Background)

background=
Default value is 'SystemMenu'. (background/Background)

bg=
Same as background.

borderwidth=
Default value is 0. (borderWidth/BorderWidth)

bd=
Same as borderwidth.

cursor=
Default value is 'arrow'. (cursor/Cursor)

disabledforeground=
Default value is 'SystemDisabledText'.
(disabledForeground/DisabledForeground)

font=
Default value is 'MS Sans Serif 8'. (font/Font)

foreground=
Default value is 'SystemMenuText'. (foreground/Foreground)

fg=
Same as foreground.

postcommand=
No default value. (postCommand/Command)

relief=
Default value is 'flat'. (relief/Relief)

selectcolor=
Default value is 'SystemMenuText'. (selectColor/Background)

takefocus=
Default value is 0. (takeFocus/TakeFocus)

tearoff=
Default value is 1. (tearOff/TearOff)

tearoffcommand=
No default value. (tearOffCommand/TearOffCommand)

title=
No default value. (title/Title)

type=
Default value is 'normal'. (type/Type)

delete(index1, index2=None) [<#>]

Deletes one or more menu entries.

index1

The first entry to delete.

index2

The last entry to delete. If omitted, only one entry is deleted.

entrycget(index, option) [<#>]

The entrycget method.

index

option

entryconfig(index, **options) [<#>]

Reconfigures the given menu entry. Only the given options are changed; the rest are left as is. See [add](#) for a list of available options.

index
***options*

entryconfigure(index, **options) [<#>]

Same as [entryconfig](#) (dead link).

index(index) [<#>]

Converts an index (of any kind) to an integer index.

index
Returns:
 An integer index.

insert(index, itemType, **options) [<#>]

Inserts an entry of the given type in the menu. This is similar to [add](#), but inserts an entry

index
itemType
***options*

insert_cascade(index, **options) [<#>]

Inserts a submenu.

index
***options*

insert_checkbutton(index, **options) [<#>]

Inserts a checkbutton.

index
***options*

insert_command(index, **options) [<#>]

Inserts a command.

index
***options*

insert_radiobutton(index, **options) [<#>]

Inserts a radiobutton.

index
***options*

insert_separator(index, **options) [<#>]

Inserts a separator. def insert_separator(index, **options)

index
***options*

invoke(index) [<#>]

The invoke method.

index

post(x, y) [#]

Displays the menu at the given position. The position should be given in pixels, relative to the root window.

x

Menu position.

y

Menu position.

type(index) [#]

Gets the type of the given menu entry.

index

Index specifier.

Returns:

Item type.

unpost() [#]

Removes a posted menu.

yposition(index) [#]

Returns the vertical offset for the given entry. This can be used to position a popup menu so that a given entry is under the mouse when the menu appears.


index

Index specifier.

Returns:

The vertical offset, in screen coordinates.

[back](#) [next](#)

 rendered by a [django](#) application. hosted by [webfaction](#).

[back](#) [next](#)

The Tkinter Menubutton Widget

The **Menubutton** widget displays popup or pulldown menu when activated.

This widget is not documented in this version of this document. If you really need it, check the Tk documentation.

When to use the Menubutton Widget

This widget is used to implement various kinds of menus. In earlier versions of Tkinter, it was used to implement toplevel menus, but this is now done with the [Menu](#) widget.

Patterns

Nothing here.

Reference

Menubutton(master=None, **options) (class) [<#>]

A button used to post a menu pane.

master

Parent widget.

***options*

Widget options. See the description of the config method for a list of available options.

config(options)** [<#>]

Modifies one or more widget options. If no options are given, the method returns a dictionary containing all current option values.

***options*

Widget options.

activebackground=

Default is system specific. (the database name is activeBackground, the class is Foreground)

activeforeground=

Default is system specific. (activeForeground/Background)

anchor=

Default is CENTER. (anchor/Anchor)

background=

Default is system specific. (background/Background)

bg=

Same as **background**.

bitmap=

No default value. (bitmap/Bitmap)

borderwidth=

Default is 2. (borderWidth/BorderWidth)

bd=

Same as **borderwidth**.

compound=

Default is NONE. (compound/Compound)

cursor=

No default value. (cursor/Cursor)

direction=

Default is "below". (direction/Direction)

disabledforeground=

font= Default is system specific. (disabledForeground/DisabledForeground)

foreground= Default is system specific. (font/Font)

fg= Default is system specific. (foreground/Foreground)

Same as **foreground**.

height= Default is 0. (height/Height)

highlightbackground= Default is system specific. (highlightBackground/HighlightBackground)

highlightcolor= Default is system specific. (highlightColor/HighlightColor)

highlightthickness= Default is 0. (highlightThickness/HighlightThickness)

image= No default value. (image/Image)

indicatoron= Default is 0. (indicatorOn/IndicatorOn)

justify= Default is CENTER. (justify/Justify)

menu= No default value. (menu/Menu)

padx= Default is '4p'. (padX/Pad)

pady= Default is '3p'. (padY/Pad)

relief= Default is FLAT. (relief/Relief)

state= Default is NORMAL. (state/State)

takefocus= Default is 0. (takeFocus/TakeFocus)

text= No default value. (text/Text)


textvariable= No default value. (textVariable/Variable)

underline= Default is -1 (no underline). (underline/Underline)

width= Default is 0. (width/Width)

wraplength= Default is 0. (wrapLength/WrapLength)

[back](#) [next](#)

 rendered by a [django](#) application. hosted by [webfaction](#).

[back](#) [next](#)

The Tkinter Message Widget

The **Message** widget is a variant of the [Label](#), designed to display multiline messages. The message widget can wrap text, and adjust its width to maintain a given aspect ratio.

When to use the Message Widget

The widget can be used to display short text messages, using a single font. You can often use a plain [Label](#) instead. If you need to display text in multiple fonts, use a [Text](#) widget.

Patterns

To create a message, all you have to do is to pass in a text string. The widget will automatically break the lines, if necessary.

```
from Tkinter import *

master = Tk()

w = Message(master, text="this is a message")
w.pack()

mainloop()
```

If you don't specify anything else, the widget attempts to format the text to keep a given aspect ratio. If you don't want that behaviour, you can specify a width:

```
w = Message(master, text="this is a relatively long message", width=50)
w.pack()
```

Reference

Message(master=None, **options) (class) [<#>]

A multi-line text message.

master

Parent widget.

***options*

Widget options. See the description of the [config](#) method for a list of available options.

config(options)** [<#>]

Modifies one or more widget options. If no options are given, the method returns a dictionary containing all current option values.

***options*

Widget options.

anchor=

Where in the message widget the text should be placed. Use one of **N**, **NE**, **E**, **SE**, **S**, **SW**, **W**, **NW**, or **CENTER**. Default is **CENTER**. (the database name is anchor, the class is Anchor)

aspect=

Aspect ratio, given as the width/height relation in percent. The default is 150, which means that the message will be 50% wider than it is high. Note that if the **width** is explicitly set, this option is ignored.

(aspect/Aspect)
background=
Message background color. The default value is system specific.
(background/Background)
bg=
Same as **background**.
borderwidth=
Border width. Default value is 2. (borderWidth/BorderWidth)
bd=
Same as **borderwidth**.
cursor=
What cursor to show when the mouse is moved over the message widget.
The default is to use the standard cursor. (cursor/Cursor)
font=
Message font. The default value is system specific. (font/Font)
foreground=
Text color. The default value is system specific. (foreground/Foreground)
fg=
Same as **foreground**.
highlightbackground=
Together with **highlightcolor** and **highlightthickness**, this option
controls how to draw the highlight region.
(highlightBackground/HighlightBackground)
highlightcolor=
See **highlightbackground**. (highlightColor/HighlightColor)
highlightthickness=
See **highlightbackground**. (highlightThickness/HighlightThickness)
justify=
Defines how to align multiple lines of text. Use **LEFT**, **RIGHT**, or
CENTER. Note that to position the text inside the widget, use the
anchor option. Default is **LEFT**. (justify/Justify)
padx=
Horizontal padding. Default is -1 (no padding). (padX/Pad)
pady=
Vertical padding. Default is -1 (no padding). (padY/Pad)
relief=
Border decoration. The default is **FLAT**. Other possible values are
SUNKEN, **RAISED**, **GROOVE**, and **RIDGE**. (relief/Relief)
takefocus=
If true, the widget accepts input focus. The default is false.
(takeFocus/TakeFocus)
text=
Message text. The widget inserts line breaks if necessary to get the
requested aspect ratio. (text/Text)
textvariable=
Associates a Tkinter variable (usually a **StringVar**) with the message. If
the variable is changed, the message text is updated.
(textVariable/Variable)
width=
Widget width, in character units. If omitted, the widget picks a suitable
width based on the **aspect** setting. (width/Width)

[back](#) [next](#)



rendered by a [django](#) application. hosted by [webfaction](#).

[back](#) [next](#)

The Tkinter OptionMenu Widget

The **OptionMenu** class is a helper class that creates a popup menu, and a button to display it. The option menu is similar to the combobox widgets commonly used on Windows.

To get the currently selected value from an option menu, you have to pass in a Tkinter variable. See the patterns section for some examples.

Patterns

To create an option menu, call the **OptionMenu** class constructor, and pass in the variable and a list of options.

```
from Tkinter import *

master = Tk()

variable = StringVar(master)
variable.set("one") # default value

w = OptionMenu(master, variable, "one", "two", "three")
w.pack()

mainloop()
```

To get the selected option, use **get** on the variable:

```
from Tkinter import *

master = Tk()

var = StringVar(master)
var.set("one") # initial value

option = OptionMenu(master, var, "one", "two", "three", "four")
option.pack()

#
# test stuff

def ok():
    print "value is", var.get()
    master.quit()

button = Button(master, text="OK", command=ok)
button.pack()

mainloop()
```

The following example shows how to create an option menu from a list of options:

```
from Tkinter import *

# the constructor syntax is:
# OptionMenu(master, variable, *values)

OPTIONS = [
    "egg",
    "bunny",
    "chicken"
]


master = Tk()

variable = StringVar(master)
variable.set(OPTIONS[0]) # default value

w = apply(OptionMenu, (master, variable) + tuple(OPTIONS))
```

```
w.pack()  
mainloop()
```

[back](#) [next](#)

 rendered by a [django](#) application. hosted by [webfaction](#).

[back](#) [next](#)

The Tkinter PanedWindow Widget

(new in Tk 8.4) The **PanedWindow** widget is a geometry manager widget, which can contain one or more child widgets (“panes”). The child widgets can be resized by the user, by moving separator lines (“sashes”) using the mouse.

When to use the PanedWindow Widget

The PanedWindow widget can be used to implement common 2-pane and 3-pane layouts.

Patterns

Here’s how to create a 2-pane widget:

```
from Tkinter import *

m = PanedWindow(orient=VERTICAL)
m.pack(fill=BOTH, expand=1)

top = Label(m, text="top pane")
m.add(top)

bottom = Label(m, text="bottom pane")
m.add(bottom)

mainloop()
```

Here’s how to create a 3-pane widget:

```
from Tkinter import *

m1 = PanedWindow()
m1.pack(fill=BOTH, expand=1)

left = Label(m1, text="left pane")
m1.add(left)

m2 = PanedWindow(m1, orient=VERTICAL)
m1.add(m2)

top = Label(m2, text="top pane")
m2.add(top)

bottom = Label(m2, text="bottom pane")
m2.add(bottom)

mainloop()
```

Reference

PanedWindow(master=None, **options) (class) [<#>]

A paned window manager widget. This widget manages one or more child widgets, and allows the user to resize these widgets, by moving the separators between them.

master

Parent widget.

***options*

Widget options. See the description of the config method for a list of available options.

add(child, **options) [<#>]

Adds a child window to the paned window.

config(options) [<#>]**

Modifies one or more widget options. If no options are given, the method returns a dictionary containing all current option values.

***options*

Widget options.

background=

Default is system specific. (background/Background)

bd=

Same as **borderwidth**.

bg=

Same as **background**.

borderwidth=

Default is 2. (borderWidth/BorderWidth)

cursor=

No default value. (cursor/Cursor)

handlepad=

Default is 8. (handlePad/HandlePad)

handlesize=

Default is 8. (handleSize/HandleSize)

height=

No default value. (height/Height)

opaqueresize=

No default value. (opaqueResize/OpaqueResize)

orient=

Default is HORIZONTAL. (orient/Orient)

relief=

Default is FLAT. (relief/Relief)

sashcursor=

No default value. (sashCursor/Cursor)

sashpad=

Default is 2. (sashPad/SashPad)

sashrelief=

Default is RAISED. (sashRelief/Relief)

sashwidth=

Default is 2. (sashWidth/Width)

showhandle=

No default value. (showHandle/ShowHandle)

width=

No default value. (width/Width)

forget(child) [<#>]

Removes a child window.

identify(x, y) [<#>]

Identifies the widget element at the given position.

panecget(child, option) [<#>]

Gets a child window option.

paneconfig(child, **options) [<#>]

Same as [paneconfigure](#).

paneconfigure(child, **options) [<#>]

Set child window configuration options.

child

Child window.

****options**
Child window options.

after=
Insert after this widget.

before=
Insert before this widget.

height=
Widget height.

minsize=
Minimal size (width for horizontal panes, height for vertical panes).

padx=
Horizontal padding.

pady=
Vertical padding.

sticky=
Defines how to expand a child widget if the resulting pane is larger than the widget itself. This can be any combination of the constants **S**, **N**, **E**, and **W**, or **NW**, **NE**, **SW**, and **SE**.

width=
Widget width.

panes() [<#>]

Returns a list of child widgets.

Returns:

A list of widgets.

proxy_coord() [<#>]

Gets the most recent proxy position.

proxy_forget() [<#>]

Removes the proxy.

proxy_place(x, y) [<#>]

Places the proxy at the given position.

remove(child) [<#>]

Same as [forget](#).

sash_coord(index) [<#>]

Gets the current position for a sash (separator).

index

Sash index (0..n).

Returns:

The upper left corner of the sash, given as a 2-tuple (x, y).

sash_dragto(index, x, y) [<#>]

Drag the sash (separator) to a new position, relative to the mark. Together with [sash_mark](#), this method is used by the widget bindings to move a sash by dragging the mouse. The **mark** method is called when the mouse is pressed over a sash (you can use **identify** to figure out which sash to mark), and the **dragto** method is called repeatedly when the mouse pointer is moved.

Note that this method is missing from the Tkinter bindings in Python 2.3. You can use **sash("dragto", index, x, y)** instead.

index

Sash index (0..n).

Returns:

The upper left corner of the sash, given as a 2-tuple (x, y).

sash_mark(index, x, y) [<#>]

Registers the current mouse position. See [sash_dragto](#) for more information.

Note that this method only takes a single argument in the Tkinter bindings shipped with in Python 2.3. To pass in all three arguments, use **sash(“mark”, index, x, y)**.

index

Sash index (0..n).

x

Start position.

y

Start position.

sash_place(index, x, y) [<#>]

Moves the sash (separator) to a given position.

index

Sash index (0..n).


x

Sash position.

y

Sash position.

[back](#) [next](#)

 rendered by a [django](#) application. hosted by [webfaction](#).

[back](#) [next](#)

The Tkinter Radiobutton Widget

The **Radiobutton** is a standard Tkinter widget used to implement one-of-many selections. Radiobuttons can contain text or images, and you can associate a Python function or method with each button. When the button is pressed, Tkinter automatically calls that function or method.

The button can only display text in a single font, but the text may span more than one line. In addition, one of the characters can be underlined, for example to mark a keyboard shortcut. By default, the **Tab** key can be used to move to a button widget.

Each group of **Radiobutton** widgets should be associated with single variable. Each button then represents a single value for that variable.

When to use the Radiobutton Widget

The radiobutton widget is used to implement one-of-many selections. It's almost always used in groups, where all group members use the same variable.

Patterns

The **Radiobutton** widget is very similar to the check button. To get a proper radio behavior, make sure to have all buttons in a group point to the same variable, and use the **value** option to specify what value each button represents:

```
from Tkinter import *

master = Tk()

v = IntVar()

Radiobutton(master, text="One", variable=v, value=1).pack(anchor=W)
Radiobutton(master, text="Two", variable=v, value=2).pack(anchor=W)

mainloop()
```

If you need to get notified when the value changes, attach a **command** callback to each button.

To create a large number of buttons, use a loop:

```
MODES = [
    ("Monochrome", "1"),
    ("Grayscale", "L"),
    ("True color", "RGB"),
    ("Color separation", "CMYK"),
]

v = StringVar()
v.set("L") # initialize

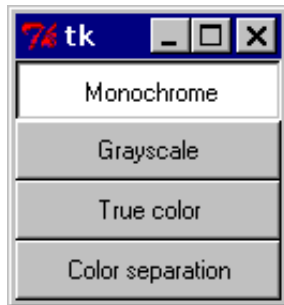
for text, mode in MODES:
    b = Radiobutton(master, text=text,
                    variable=v, value=mode)
    b.pack(anchor=W)
```

Figure: Standard radiobuttons



To turn the above example into a “button box” rather than a set of radio buttons, set the **indicatoron** option to 0. In this case, there’s no separate radio button indicator, and the selected button is drawn as **SUNKEN** instead of **RAISED**:

Figure: Using indicatoron=0



Reference

Radiobutton(master=None, **options) (class) [<#>]

A radio button. Radio buttons are usually used in groups, where all buttons in the group share a common **variable**.

__init__(master=None, **options) [<#>]

Create a radiobutton widget.

master

Parent widget.

***options*

Widget options. See the description of the config method for a list of available options.

config(options)** [<#>]

Modifies one or more widget options. If no options are given, the method returns a dictionary containing all current option values.

***options*

Widget options.

activebackground=

What background color to use when the button is active. The default is system specific. (the option database name is activeBackground, the class is Foreground)

activeforeground=

What foreground color to use when the button is active. The default is system specific. (activeForeground/Background)

anchor=

Controls where in the button the text (or image) should be located. Use one of **N**, **NE**, **E**, **SE**, **S**, **SW**, **W**, **NW**, or **CENTER**. Default is **CENTER**. (anchor/Anchor)

background=

The background color. The default is system specific.

(background/Background)

bg=
Same as background.

bitmap=
The bitmap to display in the widget. If the **image** option is given, this option is ignored. (bitmap/Bitmap)

borderwidth=
The width of the button border. The default is platform specific, but is usually 1 or 2 pixels. (borderWidth/BorderWidth)

bd=
Same as borderwidth.

command=
A function or method that is called when the button is pressed. The callback can be a function, bound method, or any other callable Python object. (command/Command)

compound=
Controls how to combine text and image in the button. By default, if an image or bitmap is given, it is drawn instead of the text. If this option is set to **CENTER**, the text is drawn on top of the image. If this option is set to one of **BOTTOM**, **LEFT**, **RIGHT**, or **TOP**, the image is drawn besides the text (use **BOTTOM** to draw the image under the text, etc.). Default is **NONE**. (compound/Compound)

cursor=
The cursor to show when the mouse is moved over the button. (cursor/Cursor)

disabledforeground=
The color to use when the button is disabled. The background is shown in the **background** color. The default is system specific. (disabledForeground/DisabledForeground)

font=
The font to use in the button. The button can only contain text in a single font. The default is system specific. (font/Font)

foreground=
The color to use for text and bitmap content. The default is system specific. (foreground/Foreground)

fg=
Same as foreground.

height=
The height of the button. If the button displays text, the size is given in text units. If the button displays an image, the size is given in pixels (or screen units). If the size is omitted, it is calculated based on the button contents. (height/Height)

highlightbackground=
The color to use for the highlight border when the button does not have focus. The default is system specific. (highlightBackground/HighlightBackground)

highlightcolor=
The color to use for the highlight border when the button has focus. The default is system specific. (highlightColor/HighlightColor)

highlightthickness=
The width of the highlight border. The default is system specific (usually one or two pixels). (highlightThickness/HighlightThickness)

image=
The image to display in the widget. If specified, this takes precedence over the **text** and **bitmap** options. (image/Image)

indicatoron=
If true, the widget uses the standard radio button look. If false, the selected button is drawn as SUNKEN instead. Default is true. (indicatorOn/IndicatorOn)

justify=
Defines how to align multiple lines of text. Use **LEFT**, **RIGHT**, or **CENTER**. Default is **CENTER**. (justify/Justify)

offrelief=
Default is raised. (offRelief/OffRelief)

overrelief=

Alternative relief to use when the mouse is moved over the widget. If empty, always use the **relief** value. (overRelief/OverRelief)

padx=

Extra horizontal padding between the text or image and the border. (padX/Pad)

pady=

Extra vertical padding between the text or image and the border. (padY/Pad)

relief=

Border decoration. One of **SUNKEN**, **RAISED**, **GROOVE**, **RIDGE**, or **FLAT**. Default is **FLAT** if indicatoron is true, otherwise **RAISED**. (relief/Relief)

selectcolor=

Default value is 'SystemWindow'. (selectColor/Background)

selectimage=

No default value. (selectImage/SelectImage)

state=

The button state: **NORMAL**, **ACTIVE** or **DISABLED**. Default is **NORMAL**. (state/State)

takefocus=

Indicates that the user can use the **Tab** key to move to this button. Default is an empty string, which means that the button accepts focus only if it has any keyboard bindings (default is on, in other words). (takeFocus/TakeFocus)

text=

The text to display in the button. The text can contain newlines. If the **bitmap** or **image** options are used, this option is ignored (unless the **compound** option is used). (text/Text)

textvariable=

Associates a Tkinter variable (usually a **StringVar**) to the button. If the variable is changed, the button text is updated. (textVariable/Variable)

underline=

Which character to underline, in a text label. Default is -1, which means that no character is underlined. (underline/Underline)

value=

The value associated with this radiobutton. All buttons in the same group should have distinct values. (value/Value)

variable=

The variable that's associated with this button. To get proper radio behaviour, you need to associate the same variable to all buttons in the same group. (variable/Variable)

width=

The width of the button. If the button displays text, the size is given in text units. If the button displays an image, the size is given in pixels (or screen units). If the size is omitted, or zero, it is calculated based on the button contents. (width/Width)

wraplength=

Determines when a button's text should be wrapped into multiple lines. This is given in screen units. Default is 0 (no wrapping). (wrapLength/WrapLength)

deselect() [<#>]

Deselects the button.

flash() [<#>]

Redraws the button a couple of times, alternating between active and normal appearance. This can be useful when debugging, or to indicate when some other user action has activate the button.


invoke() [<#>]

Calls the command associated with the button.

select() [<#>]

Selects the button.

[back](#) [next](#)

 rendered by a [django](#) application. hosted by [webfaction](#).

[back](#) [next](#)

The Tkinter Scale Widget

The **Scale** widget allows the user to select a numerical value by moving a “slider” knob along a scale. You can control the minimum and maximum values, as well as the resolution.

When to use the Scale Widget

You can use the **Scale** widget instead of an [Entry](#) widget, when you want the user to input a bounded numerical value.

Patterns

To create a scale with a specified range, use the **from** and **to** options. Note that to pass the **from** option as a keyword argument, you need to add a trailing underscore (**from** is a reserved keyword in Python).

```
from Tkinter import *

master = Tk()

w = Scale(master, from_=0, to=100)
w.pack()

w = Scale(master, from_=0, to=200, orient=HORIZONTAL)
w.pack()

mainloop()
```

To query the widget, call the **get** method:

```
w = Scale(master, from_=0, to=100)
w.pack()

print w.get()
```

The default resolution is **1**, which causes the widget to round all values to the nearest integer value. You can use the **resolution** option to specify another resolution; use **-1** to disable rounding.

```
w = Scale(from_=0, to=100, resolution=0.1)
```

Reference

Scale(master=None, **options) (class) [\[#\]](#)

A scale (slider).

master

Parent widget.

***options*

Widget options. See the description of the [config](#) method for a list of available options.

config(options)** [\[#\]](#)

Modifies one or more widget options. If no options are given, the method returns a dictionary containing all current option values.

***options*

Widget options.

activebackground=

Default value is system specific. (the database name is activeBackground, the class is Foreground)

background=
Default value is system specific. (background/Background)

bg=
Same as **background**.

bigincrement=
Default value is 0. (bigIncrement/BigIncrement)

borderwidth=
Default value is 2. (borderWidth/BorderWidth)

bd=
Same as **borderwidth**.

command=
No default value. (command/Command)

cursor=
No default value. (cursor/Cursor)

digits=
Default value is 0. (digits/Digits)

font=
Default value is system specific. (font/Font)

foreground=
Default value is system specific. (foreground/Foreground)

fg=
Same as **foreground**.

from=
Default value is 0. (from/From)

highlightbackground=
Default value is system specific.
(highlightBackground/HighlightBackground)

highlightcolor=
Default value is system specific. (highlightColor/HighlightColor)

highlightthickness=
Default value is 2. (highlightThickness/HighlightThickness)

label=
No default value. (label/Label)

length=
Default value is 100. (length/Length)

orient=
Default value is VERTICAL. (orient/Orient)

relief=
Default value is FLAT. (relief/Relief)

repeatdelay=
Default value is 300. (repeatDelay/RepeatDelay)

repeatinterval=
Default value is 100. (repeatInterval/RepeatInterval)

resolution=
Default value is 1. (resolution/Resolution)

showvalue=
Default value is 1. (showValue/ShowValue)

sliderlength=
Default value is 30. (sliderLength/SliderLength)

sliderrelief=
Default value is RAISED. (sliderRelief/SliderRelief)

state=
Default value is NORMAL. (state/State)

takefocus=
No default value. (takeFocus/TakeFocus)

tickinterval=
Default value is 0. (tickInterval/TickInterval)

to=
Default value is 100. (to/To)

troughcolor=
Default value is system specific. (troughColor/Background)

variable=
No default value. (variable/Variable)

width=
Default value is 15. (width/Width)

coords(value=None) [<#>]

Gets the screen coordinate corresponding to the given scale value.

value
A scale value. If omitted, this method uses the current setting.

Returns:
The corresponding screen coordinate, as a 2-tuple.

get() [<#>]

Gets the current scale value. Tkinter returns an integer if possible, otherwise a floating point value.

Returns:
The current value, as an integer or floating point value. To make sure you have a floating point value, use **float(scale.get())**.

identify(x, y) [<#>]

Checks if an active part of the scale is at the given screen location.

x
The horizontal screen coordinate.

y
The vertical screen coordinate.


Returns:
A string identifying what part of the scale is at the given location. This can be “slider”, “through1” (above or to the left of the slider), “through2” (below or to the right), or an empty string for any other part of the widget.

set(value) [<#>]

Sets the scale value.

value
The new scale value.

[back](#) [next](#)

 rendered by a [django](#) application. hosted by [webfaction](#).

[back](#) [next](#)

The Tkinter Scrollbar Widget

When to use the Scrollbar Widget

This widget is used to implement scrolled listboxes, canvases, and text fields.

Patterns

The **Scrollbar** widget is almost always used in conjunction with a **Listbox**, **Canvas**, or **Text** widget. Horizontal scrollbars can also be used with the **Entry** widget.

To connect a vertical scrollbar to such a widget, you have to do two things:

1. Set the widget's **yscrollcommand** callbacks to the **set** method of the scrollbar.
2. Set the scrollbar's **command** to the **yview** method of the widget.

```
from Tkinter import *

master = Tk()

scrollbar = Scrollbar(master)
scrollbar.pack(side=RIGHT, fill=Y)

listbox = Listbox(master, yscrollcommand=scrollbar.set)
for i in range(1000):
    listbox.insert(END, str(i))
listbox.pack(side=LEFT, fill=BOTH)

scrollbar.config(command=listbox.yview)

mainloop()
```

When the widget view is modified, the widget notifies the scrollbar by calling the **set** method. And when the user manipulates the scrollbar, the widget's **yview** method is called with the appropriate arguments.

Adding a horizontal scrollbar is as simple. Just use the **xscrollcommand** option instead, and the **xview** method.

For more examples, see the [Tkinter Scrollbar Patterns](#) article.

FIXME: Add [AutoScrollbar](#) pattern.

Reference

Scrollbar(master=None, **options) (class) [<#>]

A scrollbar.

master

Parent widget.

***options*

Widget options. See the description of the [config](#) method for a list of available options.

activate(element) [<#>]

Activates a scrollbar element.

element

The scrollbar element to activate. Use one of “arrow1”, “slider”, or

“arrow2”. If omitted, the method returns the currently active element, or an empty string if no element is active.

config(options) [#]**

Modifies one or more widget options. If no options are given, the method returns a dictionary containing all current option values.

****options**

Widget options.

activebackground=

Default value is system specific. (the database name is activeBackground, the class is Foreground)

activerelief=

Default value is RAISED. (activeRelief/Relief)

background=

Default value is system specific. (background/Background)

bg=

Same as **background**.

borderwidth=

Border width. Default value is 0. (borderWidth/BorderWidth)

bd=

Same as **borderwidth**.

command=

A callback used to update the associated widget. This is typically the **xview** or **yview** method of the scrolled widget.

If the user drags the scrollbar slider, the command is called as

callback(“moveto”, offset), where offset 0.0 means that the slider is in its topmost (or leftmost) position, and offset 1.0 means that it is in its bottommost (or rightmost) position.

If the user clicks the arrow buttons, or clicks in the trough, the command is called as **callback(“scroll”, step, what)**. The second argument is either “-1” or “1” depending on the direction, and the third argument is “units” to scroll lines (or other unit relevant for the scrolled widget), or “pages” to scroll full pages. (command/Command)

cursor=

The cursor to show when the mouse pointer is placed over the scrollbar widget. Default is a system specific arrow cursor. (cursor/Cursor)

elementborderwidth=

Default value is -1. (elementBorderWidth/BorderWidth)

highlightbackground=

Together with **highlightcolor** and **highlightthickness**, this option controls how to draw the highlight border. When the widget has focus, a **highlightthickness**-wide border is drawn in the **highlightcolor** color. Otherwise, it is drawn in the **highlightbackground** color. The default colors are system specific. (highlightBackground/HighlightBackground)

highlightcolor=

See **highlightbackground**. (highlightColor/HighlightColor)

highlightthickness=

See **highlightbackground**. Default value is 0 (no highlight border). (highlightThickness/HighlightThickness)

jump=

Default value is 0. (jump/Jump)

orient=

Defines how to draw the scrollbar. Use one of **HORIZONTAL** or **VERTICAL**. Default is **VERTICAL**. (orient/Orient)

relief=

Border decoration. The default is **SUNKEN**. Other possible values are **FLAT**, **RAISED**, **GROOVE**, and **RIDGE**. This option is ignored under Windows. (relief/Relief)

repeatdelay=

Default value is 300. (repeatDelay/RepeatDelay)

repeatinterval=

Default value is 100. (repeatInterval/RepeatInterval)

takefocus=

Default is an empty string. (takeFocus/TakeFocus)

troughcolor=

Default value is system specific. (troughColor/Background)

width=

Scrollbar width, in pixels. Default value is 16. (width/Width)

delta(deltax, deltay) [#]

Returns a floating point number that should be added to the current slider offsets in order to move the slider the given number of pixels. This is typically used by the mouse bindings to figure out how to move the slider when the user is dragging it around.

deltax

Horizontal delta, in screen coordinates.

deltay

Vertical delta, in screen coordinates.

Returns:

Value to add to the scrollbar offset.

fraction(x, y) [#]

Returns the slider position corresponding to a given mouse coordinate.

x

Horizontal screen coordinate.

y

Vertical screen coordinate.

Returns:

The scrollbar offset corresponding to the given coordinate (0.0 through 1.0).

getO [#]

Gets the current slider position.

Returns:

A tuple containing the relative offset for the upper (leftmost) and lower (rightmost) end of the scrollbar slider. Offset 0.0 means that the slider is in its topmost (or leftmost) position, and offset 1.0 means that it is in its bottommost (or rightmost) position.

identify(x, y) [#]

Identifies the scrollbar element at the given location.

x

Horizontal screen coordinate.

y

Vertical screen coordinate.

Returns:

One of “arrow1” (top/left arrow), “trough1”, “slider”, “trough2”, “arrow2” (bottom/right), or an empty string for any other part of the scrollbar widget.

set(lo, hi) [#]

Moves the slider to a new position.

lo

The relative offset for the upper (leftmost) end of the scrollbar slider.

hi

The relative offset for the lower (rightmost) end of the the scrollbar slider.

[back](#) [next](#)



rendered by a [django](#) application. hosted by [webfaction](#).

[back](#) [next](#)

The Tkinter Spinbox Widget

(new in Tk 8.4) The **Spinbox** widget is a variant of the standard Tkinter Entry widget, which can be used to select from a fixed number of values.

When to use the Spinbox Widget

The **Spinbox** widget can be used instead of an ordinary Entry, in cases where the user only has a limited number of ordered values to choose from.

Note that the spinbox widget is only available Python 2.3 and later, when linked against Tk 8.4 or later. Also note that several Tk spinbox methods appears to be missing from the Tkinter bindings in Python 2.3.

Patterns

The spinbox behaves pretty much like an ordinary Entry widget. The main difference is that you can specify what values to allow, either as a range, or using a tuple.

```
from Tkinter import *

master = Tk()

w = Spinbox(master, from_=0, to=10)
w.pack()

mainloop()
```

You can specify a set of values instead of a range:

```
w = Spinbox(values=(1, 2, 4, 8))
w.pack()
```

Reference

Spinbox(master=None, **options) (class) [\[#\]](#)

A spinbox widget.

master

Parent widget.

***options*

Widget options. See the description of the config method for a list of available options.

bbox(index) [\[#\]](#)

Returns the bounding box of a given character.

config(options)** [\[#\]](#)

Modifies one or more widget options. If no options are given, the method returns a dictionary containing all current option values.

***options*

Widget options.

activebackground=

Default is system specific. (activeBackground/Background)

background=

Default is system specific. (background/Background)

bd=

Same as **borderwidth**.

bg= Same as **background**.

borderwidth= Widget border width. The default is system specific, but is usually a few pixels. (borderWidth/BorderWidth)

buttonbackground= Button background color. Default is system specific. (Button.background/Background)

buttoncursor= Cursor to use when the mouse pointer is moved over the button part of this widget. No default value. (Button.cursor/Cursor)

buttondownrelief= The border style to use for the up button. Default is RAISED. (Button.relief/Relief)

buttonuprelief= The border style to use for the down button. Default is RAISED. (Button.relief/Relief)

command= A function or method that should be called when a button is pressed. No default value. (command/Command)

cursor= The cursor to use when the mouse pointer is moved over the entry part of this widget. Default is a text insertion cursor (usually XTERM). (cursor/Cursor)

disabledbackground= The background color to use when the widget is disabled. The default is system specific. (disabledBackground/DisabledBackground)

disabledforeground= The text color to use when the widget is disabled. The default is system specific. (disabledForeground/DisabledForeground)

exportselection= Default is True. (exportSelection/ExportSelection)

fg= Same as **foreground**.

font= The font to use in this widget. Default is system specific. (font/Font)

foreground= Text color. Default is system specific. (foreground/Foreground)

format= Format string. No default value. (format/Format)

from= The minimum value. Used together with **to** to limit the spinbox range.

Note that **from** is a reserved Python keyword. To use this as a keyword argument, add an underscore (**from_**). (from/From)

highlightbackground= Default is system specific. (highlightBackground/HighlightBackground)

highlightcolor= Default is system specific. (highlightColor/HighlightColor)

highlightthickness= No default value. (highlightThickness/HighlightThickness)

increment= Default is 1.0. (increment/Increment)

insertbackground= Color used for the insertion cursor. (insertBackground/Foreground)

insertborderwidth= Width of the insertion cursor's border. If this is set to a non-zero value, the cursor is drawn using the **RAISED** border style. (insertBorderWidth/BorderWidth)

insertofftime= Together with **insertontime**, this option controls cursor blinking. Both values are given in milliseconds. (insertOffTime/OffTime)

insertontime= See **insertofftime**. (insertOnTime/OnTime)

insertwidth=
Width of the insertion cursor. Usually one or two pixels.
(insertWidth/InsertWidth)

invalidcommand=
No default value. (invalidCommand/InvalidCommand)

invcmd=
Same as **invalidcommand**.

justify=
Default is LEFT. (justify/Justify)

readonlybackground=
Default is system specific. (readonlyBackground/ReadonlyBackground)

relief=
Default is SUNKEN. (relief/Relief)

repeatdelay=
Together with **repeatinterval**, this option controls button auto-repeat.
Both values are given in milliseconds. (repeatDelay/RepeatDelay)

repeatinterval=
See **repeatdelay**. (repeatInterval/RepeatInterval)

selectbackground=
Default is system specific. (selectBackground/Foreground)

selectborderwidth=
No default value. (selectBorderWidth/BorderWidth)

selectforeground=
Default is system specific. (selectForeground/Background)

state=
One of NORMAL, DISABLED, or “readonly”. Default is NORMAL.
(state/State)

takefocus=
Indicates that the user can use the **Tab** key to move to this widget.
Default is an empty string, which means that the entry widget accepts
focus only if it has any keyboard bindings (default is on, in other words).
(takeFocus/TakeFocus)

textvariable=
No default value. (textVariable/Variable)

to=
See **from**. (to/To)

validate=
Validation mode. Default is NONE. (validate/Validate)

validatecommand=
Validation callback. No default value.
(validateCommand/ValidateCommand)

values=
A tuple containing valid values for this widget. Overrides
from/to/increment. (values/Values)

vcmd=
Same as **validatecommand**.

width=
Widget width, in character units. Default is 20. (width/Width)

wrap=
If true, the up and down buttons will wrap around. (wrap/Wrap)

xscrollcommand=
Used to connect a spinbox field to a horizontal scrollbar. This option
should be set to the **set** method of the corresponding scrollbar.
(xScrollCommand/ScrollCommand)

delete(first, last=None) [<#>]

Deletes one or more characters from the spinbox.

getO [<#>]

Returns the current contents of the spinbox.

Returns:

The widget contents, as a string.

icursor(index) [<#>]

Moves the insertion cursor to the given index. This also sets the **INSERT** index.

index

Where to move the cursor.

identify(x, y) [<#>]

Identifies the widget element at the given location.

Returns:

One of “none”, “buttondown”, “buttonup”, or “entry”.

index(index) [<#>]

Gets the numerical position corresponding to the given index.

index

An index.

Returns:

The corresponding numerical index.

insert(index, text) [<#>]

Inserts text at the given index. Use insert(INSERT, text) to insert text at the cursor, insert(END, text) to append text to the spinbox.

index

Where to insert the text.

string

The text to insert.

invoke(element) [<#>]

Invokes a spinbox button.

element

What button to invoke. Must be one of “buttonup” or “buttondown”.

scan_dragto(x) [<#>]

Sets the scanning anchor for fast horizontal scrolling to the given mouse coordinate.

x

Current horizontal mouse position.

scan_mark(x) [<#>]

Scrolls the widget contents sideways according to the given mouse coordinate. The text is moved 10 times the distance between the scanning anchor and the new position.

x

Current horizontal mouse position.

selection_adjust(index) [<#>]

Adjusts the selection to include also the given character. If index is already selected, do nothing.

index

The index.

selection_clear() [<#>]

Clears the selection.

selection_element(element=None) [<#>]

Selects an element. If no element is specified, this method returns the current element.

[back](#) [next](#)



rendered by a [django](#) application. hosted by [webfaction](#).

[back](#) [next](#)

The Tkinter Text Widget

The **Text** widget provides formatted text display. It allows you to display and edit text with various styles and attributes. The widget also supports embedded images and windows.

When to use the Text Widget

The text widget is used to display text documents, containing either plain text or formatted text (using different fonts, embedded images, and other embellishments). The text widget can also be used as a text editor.

Concepts

The text widget stores and displays lines of text.

The text body can consist of characters, marks, and embedded windows or images. Different regions can be displayed in different styles, and you can also attach event bindings to regions.

By default, you can edit the text widget's contents using the standard keyboard and mouse bindings. To disable editing, set the **state** option to **DISABLED** (but if you do that, you'll also disable the **insert** and **delete** methods).

Indexes

Indexes are used to point to positions within the text handled by the text widget. Like Python sequence indexes, text widget indexes correspond to positions between the actual characters.

Tkinter provides a number of different index types:

- line/column (“line.column”)
- line end (“line.end”)
- **INSERT**
- **CURRENT**
- **END**
- user-defined marks
- user-defined tags (“tag.first”, “tag.last”)
- selection (**SEL_FIRST**, **SEL_LAST**)
- window coordinate (“@x,y”)
- embedded object name (window, images)
- expressions

Lines and columns

line/column indexes are the basic index type. They are given as strings consisting of a line number and column number, separated by a period. Line numbers start at 1, while column numbers start at 0, like Python sequence indexes. You can construct indexes using the following syntax:

```
"%d.%d" % (line, column)
```

It is not an error to specify line numbers beyond the last line, or column numbers beyond the last column on a line. Such numbers correspond to the line beyond the last, or the newline character ending a line.

Note that line/column indexes may look like floating point values, but it's seldom possible to treat them as such (consider position 1.25 vs. 1.3, for example). I sometimes use **1.0** instead of "1.0" to save a few keystrokes when referring to the first character in the buffer, but that's about it.

You can use the **index** method to convert all other kinds of indexes to the corresponding line/column index string.

Line endings

A *line end* index is given as a string consisting of a line number directly followed by the text **".end"**. A line end index correspond to the newline character ending a line.

Named indexes

INSERT (or "insert") corresponds to the insertion cursor.

CURRENT (or "current") corresponds to the character closest to the mouse pointer. However, it is only updated if you move the mouse without holding down any buttons (if you do, it will not be updated until you release the button).

END (or "end") corresponds to the position just after the last character in the buffer.

User-defined marks are named positions in the text. **INSERT** and **CURRENT** are predefined marks, but you can also create your own marks. See below for more information.

User-defined tags represent special event bindings and styles that can be assigned to ranges of text. For more information on tags, see below.

You can refer to the beginning of a tag range using the syntax **"tag.first"** (just before the first character in the text using that tag), and **"tag.last"** (just after the last character using that tag).

```
"%s.first" % tagname
"%s.last" % tagname
```

If the tag isn't in use, Tkinter raises a **TclError** exception.

The *selection* is a special tag named **SEL** (or "sel") that corresponds to the current selection. You can use the constants **SEL_FIRST** and **SEL_LAST** to refer to the selection. If there's no selection, Tkinter raises a **TclError** exception.

Coordinates

You can also use *window coordinates* as indexes. For example, in an event binding, you can find the character closest to the mouse pointer using the following syntax:

```
"@%d,%d" % (event.x, event.y)
```

Embedded objects

Embedded object name can be used to refer to windows and images embedded in the text widget. To refer to a window, simply use the corresponding Tkinter widget instance as an index. To refer to an embedded image, use the corresponding Tkinter **PhotoImage** or **BitmapImage** object.

Expressions

Expressions can be used to modify any kind of index. Expressions are formed by taking the string representation of an index (use **str** if the index isn't already a string), and appending one or more *modifiers* from the following list:

- “+ *count* **chars**” moves the index forward. The index will move over newlines, but not beyond the **END** index.
- “- *count* **chars**” moves the index backwards. The index will move over newlines, but not beyond index “1.0”.
- “+ *count* **lines**” and “- *count* **lines**” moves the index full lines forward (or backwards). If possible, the index is kept in the same column, but if the new line is too short, the index is moved to the end of that line.
- “**linestart**” moves the index to the first position on the line.
- “**lineend**” the index to the last position on the line (the newline, that is).
- “**wordstart**” and “**wordend**” moves the index to the beginning (end) of the current word. Words are sequences of letters, digits, and underline, or single non-space characters.

The keywords can be abbreviated and spaces can be omitted as long as the result is not ambiguous. For example, “+ **5 chars**” can be shortened to “+**5c**”.

For compatibility with implementations where the constants are not ordinary strings, you may wish to use **str** or formatting operations to create the expression string. For example, here's how to remove the character just before the insertion cursor:

```
def backspace(event):
    event.widget.delete("%s-1c" % INSERT, INSERT)
```

Marks

Marks are (usually) invisible objects embedded in the text managed by the widget. Marks are positioned between character cells, and moves along with the text.

- user-defined marks
- **INSERT**
- **CURRENT**

You can use any number of *user-defined marks* in a text widget. Mark names are ordinary strings, and they can contain anything except whitespace (for convenience, you should avoid names that can be confused with indexes, especially names containing periods). To create or move a mark, use the **mark_set** method.

Two marks are predefined by Tkinter, and have special meaning:

INSERT (or “insert”) is a special mark that is used to represent the insertion cursor. Tkinter draws the cursor at this mark's position, so it isn't entirely invisible.

CURRENT (or “current”) is a special mark that represents the character closest to the mouse pointer. However, it is only updated if you move the mouse without holding down any buttons (if you do, it will not be updated until you release the button).

Special marks can be manipulated as other user-defined marks, but they cannot be deleted.

If you insert or delete text before a mark, the mark is moved along with the other text. To remove a mark, you must use the **mark_unset** method. Deleting text around a mark doesn't remove the mark itself.

If you insert text *at* a mark, it may be moved to the end of that text or left where it was, depending on the mark's *gravity* setting (**LEFT** or **RIGHT**; default is **RIGHT**). You can use the **mark_gravity** method to change the gravity setting for a given mark.

In the following example, the “sentinel” mark is used to keep track of the original position for the insertion cursor.

```
text.mark_set("sentinel", INSERT)
text.mark_gravity("sentinel", LEFT)
```

You can now let the user enter text at the insertion cursor, and use **text.get(sentinel, INSERT)** to pick up the result.

Tags

Tags are used to associated a display style and/or event callbacks with ranges of text.

- user-defined tags
- **SEL**

You can define any number of *user-defined tags*. Any text range can have multiple tags, and the same tag can be used for many different ranges. Unlike the **Canvas** widget, tags defined for the text widget are not tightly bound to text ranges; the information associated with a tag is kept also if there is no text in the widget using it.

Tag names are ordinary strings, and they can contain anything except whitespace.

SEL (or “sel”) is a special tag which corresponds to the current selection, if any. There should be at most one range using the selection tag.

The following options are used with **tag_config** to specify the visual style for text using a certain tag.

background (color)

The background color to use for text having this tag.

Note that the **bg** alias cannot be used with tags; it is interpreted as **bgstipple** rather than **background**.

bgstipple (bitmap)

The name of a bitmap which is used as a stipple brush when drawing the background. Typical values are “gray12”, “gray25”, “gray50”, or “gray75”. Default is a solid brush (no bitmap).

borderwidth (distance)

The width of the text border. The default is 0 (no border).

Note that the **bd** alias cannot be used with tags.

fgstipple (bitmap)

The name of a bitmap which is used as a stipple brush when drawing the text. Typical values are “gray12”, “gray25”, “gray50”, or “gray75”. Default is a solid brush (no bitmap).

font (font)

The font to use for text having this tag.

foreground (color)

The color to use for text having this tag.

Note that the **fg** alias cannot be used with tags; it is interpreted as **fgstipple** rather than **foreground**.

justify (constant)

Controls text justification (the first character on a line determines how to justify the whole line). Use one of **LEFT**, **RIGHT**, or **CENTER**. Default is **LEFT**.

lmargin1 (distance)

The left margin to use for the first line in a block of text having this tag. Default is 0 (no left margin).

lmargin2 (distance)

The left margin to use for every line but the first in a block of text having this tag. Default is 0 (no left margin).

offset (distance)

Controls if the text should be offset from the baseline. Use a positive value for superscripts, a negative value for subscripts. Default is 0 (no offset).

overstrike (flag)

If non-zero, the text widget draws a line over the text that has this tag. For best results, you should use overstrike fonts instead.

relief (constant)

The border style to use for text having this tag. Use one of **SUNKEN**, **RAISED**, **GROOVE**, **RIDGE**, or **FLAT**. Default is **FLAT** (no border).

rmargin (distance)

The right margin to use for blocks of text having this tag. Default is 0 (no right margin).

spacing1 (distance)

Spacing to use above the first line in a block of text having this tag. Default is 0 (no extra spacing).

spacing2 (distance)

Spacing to use between the lines in a block of text having this tag. Default is 0 (no extra spacing).

spacing3 (distance)

Spacing to use after the last line of text in a block of text having this tag. Default is 0 (no extra spacing).

tabs (string)

underline (flag)

If non-zero, the text widget underlines the text that has this tag. For example, you can get the standard hyperlink look with (foreground="blue", underline=1). For best results, you should use underlined fonts instead.

wrap (constant)

The word wrap mode to use for text having this tag. Use one of **NONE**, **CHAR**, or **WORD**.

If you attach multiple tags to a range of text, style options from the most recently created tag override options from earlier tags. In the following example, the resulting text is blue on a yellow background.

```
text.tag_config("n", background="yellow", foreground="red")
text.tag_config("a", foreground="blue")
```

```
text.insert(contents, ("n", "a"))
```

Note that it doesn't matter in which order you attach tags to a range; it's the tag creation order that counts.

You can change the tag priority using the **tag_raise** and **tag_lower**. If you add a **text.tag_lower("a")** to the above example, the text becomes red.

The **tag_bind** method allows you to add event bindings to text having a particular tag. Tags can generate mouse and keyboard events, plus **Enter** and **Leave** events. For example, the following code snippet creates a tag to use for any hypertext links in the text:

```
text.tag_config("a", foreground="blue", underline=1)
text.tag_bind("Enter>", show_hand_cursor)
text.tag_bind("Leave>", show_arrow_cursor)
text.tag_bind("Button-1>", click)
text.config(cursor="arrow")

text.insert(INSERT, "click here!", "a")
```

Patterns

When you create a new text widget, it has no contents. To insert text into the widget, use the **insert** method and insert text at the **INSERT** or **END** indexes:

```
text.insert(END, "hello, ")
text.insert(END, "world")
```

You can use an optional third argument to the **insert** method to attach one or more tags to the newly inserted text:

```
text.insert(END, "this is a ")
text.insert(END, "link", ("a", "href"+href))
```

To insert embedded objects, use the **window_create** or **image_create** methods:

```
button = Button(text, text="Click", command=click)
text.window_create(INSERT, window=button)
```

To delete text, use the **delete** method. Here's how to delete all text from the widget (this also deletes embedded windows and images, but not marks):

```
text.delete(1.0, END)
```

To delete a single character (or an embedded window or image), you can use **delete** with only one argument:

```
text.delete(INSERT)
text.delete(button)
```

To make the widget read-only, you can change the **state** option from **NORMAL** to **DISABLED**:

```
text.config(state=NORMAL)
text.delete(1.0, END)
text.insert(END, text)
text.config(state=DISABLED)
```

Note that you must change the state back to **NORMAL** before you can modify the widget contents from within the program. Otherwise, calls to **insert** and **delete** will be silently ignored.

To fetch the text contents of the widget, use the **get** method:

```
contents = text.get(1.0, END)
```

FIXME: *add material on the dump method, and how to use it on 1.5.2 and earlier*

Here's a simple way to keep track of changes to the text widget:

```
import md5
def getsignature(contents):
    return md5.md5(contents).digest()

text.insert(END, contents) # original contents
signature = getsignature(contents)

...

contents = text.get(1.0, END)
if signature != getsignature(contents):
    print "contents have changed!"
```

FIXME: modify to handle ending linefeed added by text widget

The **index** method converts an index given in any of the supported formats to a line/column index. Use this if you need to store an “absolute” index.

```
index = text.index(index)
```

However, if you need to keep track of positions in the text even after other text is inserted or deleted, you should use marks instead.

```
text.mark_set("here", index)
text.mark_unset("here")
```

The following function converts any kind of index to a (line, column)-tuple. Note that you can directly compare positions represented by such tuples.

```
def getindex(text, index):
    return tuple(map(int, string.split(text.index(index), ".")))

if getindex(text, INSERT) == getindex(text, "sentinel"):
    text.mark_set(INSERT, "sentinel")
```

The following example shows how to enumerate all regions in the text that has a given tag.

```
ranges = text.tag_ranges(tag)
for i in range(0, len(ranges), 2):
    start = ranges[i]
    stop = ranges[i+1]
    print tag, repr(text.get(start, stop))
```

The **search** method allows you to search for text. You can search for an exact match (default), or use a Tcl-style regular expression (call with the **regexp** option set to true).

```
text.insert(END, "hello, world")

start = 1.0
while 1:
    pos = text.search("o", start, stopindex=END)
    if not pos:
        break
    print pos
    start = pos + "+1c"
```

Given an empty text widget, the above example prints **1.4** and **1.8** before it stops. If you omit the **stopindex** option, the search wraps around if it reaches the end of the text.

To search backwards, set the **backwards** option to true (to find all occurrences, start at **END**, set **stopindex** to 1.0 to avoid wrapping, and use “-1c” to move the start position).

Reference

Text(master=None, **options) (class) [<#>]

A display/editor widget for formatted text.

master

Parent widget.

****options**

Widget options. See the description of the config method for a list of available options.

bbox(index) [<#>]

Calculates the bounding box for the given character.

This method only works if the text widget is updated. To make sure this is the case, you can call the `update_idletasks` method first.

index

Character index.

Returns:

A 4-tuple (x, y, width, height), or None, if the character is not visible.

compare(index1, op, index2) [<#>]

Compares two indexes. The **op** argument is one of "<", "<=", "==", ">=", ">", or "!=" (Python's "<>" syntax is not supported).

index1

First index.

op

Operator (see above).

index2

Second index.

Returns:

A true value if the condition is true.

config(options) [<#>]**

Modifies one or more widget options. If no options are given, the method returns a dictionary containing all current option values.

****options**

Widget options.

autoseparators=

Default is 1. (autoSeparators/AutoSeparators)

background=

Default value is system specific. (the database name is background, the class is Background)

bg=

Same as background.

borderwidth=

Default value is 2. (borderWidth/BorderWidth)

bd=

Same as borderwidth.

cursor=

Default value is "xterm". (cursor/Cursor)

exportselection=

Default value is 1. (exportSelection/ExportSelection)

font=

Default value is system specific. (font/Font)

foreground=

Default value is system specific. (foreground/Foreground)

fg=

Same as foreground.

height=

Default value is 24. (height/Height)

highlightbackground=

Default value is system specific.
 (highlightBackground/HighlightBackground)
highlightcolor=
 Default value is system specific. (highlightColor/HighlightColor)
highlightthickness=
 Default value is 0. (highlightThickness/HighlightThickness)
insertbackground=
 Default value is system specific. (insertBackground/Foreground)
insertborderwidth=
 Default value is 0. (insertBorderWidth/BorderWidth)
insertofftime=
 Default value is 300. (insertOffTime/OffTime)
insertontime=
 Default value is 600. (insertOnTime/OnTime)
insertwidth=
 Default value is 2. (insertWidth/InsertWidth)
maxundo=
 Default is 0. (maxUndo/MaxUndo)
padx=
 Default value is 1. (padX/Pad)
pady=
 Default value is 1. (padY/Pad)
relief=
 Default value is SUNKEN. (relief/Relief)
selectbackground=
 Default value is system specific. (selectBackground/Foreground)
selectborderwidth=
 Default value is 0. (selectBorderWidth/BorderWidth)
selectforeground=
 Default value is system specific. (selectForeground/Background)
setgrid=
 Default value is 0. (setGrid/SetGrid)
spacing1=
 Default value is 0. (spacing1/Spacing)
spacing2=
 Default value is 0. (spacing2/Spacing)
spacing3=
 Default value is 0. (spacing3/Spacing)
state=
 Default value is NORMAL. (state/State)
tabs=
 No default value. (tabs/Tabs)
takefocus=
 No default value. (takeFocus/TakeFocus)
undo=
 Default is 0. (undo/Undo)
width=
 Default value is 80. (width/Width)
wrap=
 Default value is CHAR. (wrap/Wrap)
xscrollcommand=
 No default value. (xScrollCommand/ScrollCommand)
yscrollcommand=
 No default value. (yScrollCommand/ScrollCommand)

debug(boolean=None) [<#>]

Enables or disables debugging.

boolean

delete(start, end=None) [<#>]

Deletes the character (or embedded object) at the given position, or all text in the given range. Any marks within the range are moved to the beginning of the range.

start

Start index.

end

End index. If omitted, only one character is deleted.

dlineinfo(index) [<#>]

Calculates the bounding box for the line containing the given character.

This method only works if the text widget is updated. To make sure this is the case, you can call the `update_idletasks` method first.

index

Character index.

Returns:

A 5-tuple: (x, y, width, height, offset). The last tuple member is the offset from the top of the line to the baseline. If the line is not visible, this method returns None.

dump(index1, index2=None, command=None, **kw) [<#>]

Dumps the widget contents.

edit_modified(arg=None) [<#>]

The `edit_modified` method.

edit_redo() [<#>]

The `edit_redo` method.

edit_reset() [<#>]

The `edit_reset` method.

edit_separator() [<#>]

The `edit_separator` method.

edit_undo() [<#>]

The `edit_undo` method.

get(start, end=None) [<#>]

Returns the character at the given position, or all text in the given range.

start

Start position.

end

End position. If omitted, only one character is returned.

image_cget(index, option) [<#>]

Returns the current value of the given option. If there's no image on the given position, this method raises a **TclError** exception. Not implemented in Python 1.5.2 and earlier.

index

Index specifier.

option

Image option.

Returns:

Option value.

image_configure(index, **options) [<#>]

Modifies one or more image options. If there's no image on the given position, this method raises a **TclError** exception. Not implemented in Python 1.5.2 and earlier.

index

Index specifier.

***options*

Image options.

*align=**image=*

An image object. This must be a PhotoImage or BitmapImage object, or any compatible object.

*name=**padx=**pady=***image_create(index, cnf={}, **kw) [<#>]**

Inserts an image at the given position. The image must be a Tkinter **PhotoImage** or **BitmapImage** instance (or an instance of the corresponding PIL classes).

This method doesn't work with Tk versions before 8.0. As a workaround, you can put the image in a Label widget, and use `window_create` to insert the label widget.

index

Index specifier.

***options*Image options. See `image_config` for a complete list.**image_names() [<#>]**

Finds the names of all images embedded in the text widget. Tkinter doesn't provide a way to get the corresponding **PhotoImage** or **BitmapImage** objects, but you can keep track of those yourself using a dictionary (using **str(image)** as the key).

Returns:

A tuple containing image names.

index(index) [<#>]

Returns the "line.column" index corresponding to the given index.

index

Index specifier.

Returns:

The corresponding row/column, given as a "line.column" string.

insert(index, text, *tags) [<#>]

Inserts text at the given position. The index is typically **INSERT** or **END**. If you provide one or more tags, they are attached to the new text.

If you insert text on a mark, the mark is moved according to its gravity setting.

index

Where to insert the text.

text

The text to insert.

**tags*

Optional tags to attach to the text.

mark_gravity(self, name, direction=None) [#]

Sets the gravity for the given mark. The gravity setting controls how to move the mark if text is inserted exactly on the mark. If **LEFT**, the mark is not moved if text is inserted at the mark (that is, the text is inserted just after the mark). If **RIGHT**, the mark is moved to the right end of the text (that is, the text is inserted just before the mark). The default gravity setting is **RIGHT**.

name

The name of the mark.

direction

The gravity setting (**LEFT** or **RIGHT**). If this argument is omitted, the method returns the current gravity setting.

Returns:

The current gravity setting, if direction was omitted.

mark_names() [#]

Finds the names of all marks used in the widget. This includes the **INSERT** and **CURRENT** marks (but not **END**, which is a special index, not a mark).

Returns:

A tuple containing mark names.

mark_next(index) [#]

The mark_next method.

index

mark_previous(index) [#]

The mark_previous method.

index

mark_set(name, index) [#]

Moves the mark to the given position. If the mark doesn't exist, it is created (with gravity set to **RIGHT**). You also use this method to move the predefined **INSERT** and **CURRENT** marks.

name

Mark name.

index

New position.

mark_unset(name) [#]

Removes the given mark from the widget. You cannot remove the builtin **INSERT** and **CURRENT** marks.

name

Mark name.

scan_dragto(x, y) [<#>]

Scrolls the widget contents. The text view is moved 10 times the distance between the scanning anchor and the new position.

x

y

scan_mark(x, y) [<#>]

Sets the scanning anchor. This anchor is used for fast scrolling.

x

y

search(pattern, index, stopindex=None, forwards=None, backwards=None, exact=None, regexp=None, nocase=None, count=None) [<#>]

Searches for text strings or regular expressions.

pattern

index

stopindex

forwards

backwards

exact

regexp

nocase

count

see(index) [<#>]

Makes sure a given position is visible. If the index isn't visible, scroll the view as necessary.

index

Index specifier.

tag_add(tagName, index1, *args) [<#>]

The tag_add method.

tagName

index1

**args*

tag_bind(tagName, sequence, func, add=None) [<#>]

The tag_bind method.

tagName

sequence

func

add

tag_cget(tagName, option) [<#>]

The tag_cget method.

tagName
option

tag_config(tagName, cnf={}, **kw) [<#>]

The tag_config method.

tagName
cnf
***kw*

tag_configure(tagName, cnf={}, **kw) [<#>]

The tag_configure method.

tagName
cnf
***kw*

tag_delete(*tagNames) [<#>]

The tag_delete method.

**tagNames*

tag_lower(tagName, belowThis=None) [<#>]

The tag_lower method.

tagName
belowThis

tag_names(index=None) [<#>]

The tag_names method.

index

tag_nextrange(tagName, index1, index2=None) [<#>]

The tag_nextrange method.

tagName
index1
index2

tag_prevrange(tagName, index1, index2=None) [<#>]

The tag_prevrange method.

tagName
index1
index2

tag_raise(tagName, aboveThis=None) [<#>]

The tag_raise method.

tagName
aboveThis

tag_ranges(tagName) [<#>]

The tag_ranges method.

tagName

tag_remove(tagName, index1, index2=None) [<#>]

The tag_remove method.

tagName
index1
index2

tag_unbind(tagName, sequence, funcid=None) [<#>]

The tag_unbind method.

tagName
sequence
funcid

window_cget(index, option) [<#>]

Returns the current value of the given window option. If there's no window on the given position, this method raises a **TclError** exception.

index
option

window_config(index, **options) [<#>]

Modifies one or more window options. If there's no window on the given position, this method raises a **TclError** exception.

index
***options*
align=
create=
padx=
pady=
stretch=
window=

Window object.

window_configure(index, cnf=None, **kw) [<#>]

Same as [window_config](#).

window_create(index, **options) [<#>]

Inserts a widget at the given position. You can either create the widget (which should be a child of the text widget itself) first, and insert it using the **window** option, or provide a **create** callback which is called when the window is first displayed.

index
Index specifier.

*****options***

Window options. See `window_config` for a complete list.

window_names() [<#>]

Returns a tuple containing all windows embedded in the text widget. In 1.5.2 and earlier, this method returns the names of the widgets, rather than the widget instances.

Here's how to convert the names to a list of widget instances in a portable fashion:

```
windows = text.window_names()
try:
    windows = map(text.nametowidget, windows)
except TclError:
    pass
```

Returns:

A list of window names.

xview(*what) [<#>]

The xview method.

**what*

xview_moveto(fraction) [<#>]

The xview_moveto method.

fraction

xview_scroll(number, what) [<#>]

The xview_scroll method.

number

what

yview(*what) [<#>]

The yview method.

**what*

yview_moveto(fraction) [<#>]

The yview_moveto method.

fraction

yview_pickplace(*what) [<#>]

The yview_pickplace method.


**what*

yview_scroll(number, what) [<#>]

The yview_scroll method.

number
what

[back](#) [next](#)

 rendered by a [django](#) application. hosted by [webfaction](#).

[back](#) [next](#)

The Tkinter Toplevel Widget

The **Toplevel** widget work pretty much like **Frame**, but it is displayed in a separate, top-level window. Such windows usually have title bars, borders, and other “window decorations”.

When to use the Toplevel Widget

The Toplevel widget is used to display extra application windows, dialogs, and other “pop-up” windows.

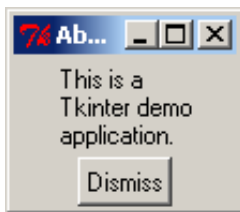
Patterns

```
...

top = Toplevel()
top.title("About this application...")

msg = Message(top, text=about_message)
msg.pack()

button = Button(top, text="Dismiss", command=top.destroy)
button.pack()
```



FIXME: add more patterns

Reference

Toplevel(master=None, **options) (class) [<#>]

A widget container placed in a new top level window.

master

Parent widget.

***options*

Widget options. See the description of the [config](#) method for a list of available options.

config(options)** [<#>]

Modifies one or more widget options. If no options are given, the method returns a dictionary containing all current option values.

***options*

Widget options.

background=

The background color to use in this toplevel. This defaults to the application background color. To prevent updates, set the color to an empty string. (the option database name is background, the class is Background)

bg=

Same as **background**.

borderwidth=

Width of the 3D border. Defaults to 0 (no border).
(borderWidth/BorderWidth)

bd=

Same as **borderwidth**.

class=

Default value is Toplevel. (class/Class)

colormap=

Some displays support only 256 colors (some use even less). Such displays usually provide a color map to specify which 256 colors to use. This option allows you to specify which color map to use for this toplevel window, and its child widgets.

By default, a new toplevel window uses the same color map as the root window. Using this option, you can reuse the color map of another window instead (this window must be on the same screen and have the same visual characteristics). You can also use the value “new” to allocate a new color map for this window.

You cannot change this option once you’ve created the window.
(colormap/Colormap)

container=

Default value is 0. (container/Container)

cursor=

The cursor to show when the mouse pointer is located over the toplevel widget. Default is a system specific arrow cursor. (cursor/Cursor)

height=

Window height, in pixels. If omitted, the widget is made large enough to hold its contents. (height/Height)

highlightbackground=

The color to use for the highlight region when the widget doesn’t have focus. The default is system specific.

(highlightBackground/HighlightBackground)

highlightcolor=

The color to use for the highlight region when the widget has focus. The default is system specific. (highlightColor/HighlightColor)

highlightthickness=

The width of the highlight region. Default is 0 (no region).
(highlightThickness/HighlightThickness)

menu=

A menu to associate with this toplevel window. On Unix and Windows, the menu is placed at the top of the toplevel window itself. On Macs, the menu is displayed at the top of the screen when the toplevel window is selected. (menu/Menu)

padx=

Horizontal padding. Default is 0. (padX/Pad)

pady=

Vertical padding. Default is 0. (padY/Pad)

relief=

Border decoration: either **FLAT**, **SUNKEN**, **RAISED**, **GROOVE**, or **RIDGE**. The default is **FLAT**. (relief/Relief)

screen=

No default value. (screen/Screen)

takefocus=

Indicates that the user can use the **Tab** key to move to this widget. Default is false. (takeFocus/TakeFocus)

use=

No default value. (use/Use)

visual=

Controls the “visual” type to use for this window. This option should usually be omitted. In that case, the visual type is inherited from the root window.

Some more advanced displays support “mixed visuals”. This typically means that the root window is a 256-color display (the “pseudocolor” visual type), but that individual windows can be displayed as true 24-bit color (the “truecolor” visual type). On such displays, you may wish to explicitly set the visual option to “truecolor” for any windows used to display full-color images.

Other possible values include “directcolor”, “staticcolor”, “grayscale”, or “staticgray”. See your X window documentation for details.

You cannot change this option once you’ve created the window.

(visual/Visual)

width=

Window width, in pixels. If omitted, the widget is made large enough to hold its contents.

[back](#) [next](#)



rendered by a [django](#) application. hosted by [webfaction](#).

[back](#) [next](#)

Basic Widget Methods

The following methods are provided by all widgets (including the root window).

The root window and other **Toplevel** windows provide additional methods. See the [Window Methods](#) section for more information.

Patterns

Configuration

```
w.config(option=value)
value = w.cget("option")
k = w.keys()
```

Event processing

```
mainloop()
w.mainloop()
w.quit()
w.wait_variable(var)
w.wait_visibility(window)
w.wait_window(window)
w.update()
w.update_idletasks()
```

Event callbacks

```
w.bind(event, callback)
w.unbind(event)
w.bind_class(event, callback)
w.bindtags()
w.bindtags(tags)
```

Alarm handlers and other non-event callbacks

```
id = w.after(time, callback)
id = w.after_idle(callback)
w.after_cancel(id)
```

Window management

```
w.lift()
w.lower()
```

Window-related information

```
w.winfo_width(), w.winfo_height()
w.winfo_reqwidth(), w.winfo_reqheight()
w.winfo_id()
```

The option database

```
w.option_add(pattern, value)
w.option_get(name, class)
```

Reference

Widget (class) [\[#\]](#)

Widget implementation class. This class is used as a mixin by all widget classes.

after(delay_ms, callback=None, *args) [\[#\]](#)

Registers an alarm callback that is called after a given time.

This method registers a callback function that will be called after a given number of milliseconds. Tkinter only guarantees that the callback will not be called earlier than that; if the system is busy, the actual delay may be much longer.

The callback is only called once for each call to this method. To keep calling the callback, you need to reregister the callback inside itself:

```
class App:
    def __init__(self, master):
        self.master = master
        self.poll() # start polling

    def poll(self):
        ... do something ...
        self.master.after(100, self.poll)
```

[after_cancel](#) to cancel the callback.

You can also omit the callback. If you do, this method simply waits for the given number of milliseconds, without serving any events (same as **time.sleep(delay_ms*0.001)**).

delay_ms

Delay, in milliseconds.

callback

The callback. This can be any callable object.

**args*

Optional arguments that are passed to the callback.

Returns:

An alarm identifier.

after_cancel(id) [<#>]

Cancels an alarm callback.

id

Alarm identifier.

after_idle(callback, *args) [<#>]

Registers a callback that is called when the system is idle. The callback will be called there are no more events to process in the mainloop. The callback is only called once for each call to [after_idle](#).

callback

The callback. This can be any callable object.

**args*

Optional arguments that are passed to the callback.

Returns:

An alarm identifier.

bbox(column=None, row=None, col2=None, row2=None) [<#>]

The bbox method.

column

row

col2

row2

bell(displayof=o) [<#>]

Generate a system-dependent sound (typically a short beep).

displayof

bind(sequence=None, func=None, add=None) [<#>]

Adds an event binding to this widget. Usually, the new binding replaces any existing binding for the same event sequence. By passing in “+” as the third argument, the new callback is added to the existing binding.

bind_all(sequence=None, func=None, add=None) [<#>]

Adds an event binding to the application level. Usually, the new binding replaces any existing binding for the same event sequence. By passing in “+” as the third argument, the new function is added to the existing binding.

bind_class(className, sequence=None, func=None, add=None) [<#>]

Adds an event binding to the given widget class. Usually, the new binding replaces any existing binding for the same event sequence. By passing in “+” as the third argument, the new function is added to the existing binding.

bindtags(tagList=None) [<#>]

Sets or gets the binding search order for this widget.

If called without an argument, this method returns a tuple containing the binding search order used for this widget. By default, this tuple contains the widget’s name (str(self)), the widget class (e.g. Button), the root window’s name, and finally the special name all which refers to the application level.

cget(key) [<#>]

Returns the current value for an option.

Note that option values are always returned as strings (also if you gave a nonstring value when you configured the widget). Use **int** and **float** where appropriate.

clipboard_append(string, **options) [<#>]

Adds text to the clipboard.

string

The text to add.

***options*

clipboard_clear(options) [<#>]**

Clears the clipboard.

***options*

colormodel(value=None) [<#>]

The colormodel method.

value

columnconfigure(index, cnf={}, **kw) [<#>]

The columnconfigure method.

index

cnf
***kw*

config(cnf=None, **kw) [<#>]

Modifies one or more widget options.

If called without an argument, this method returns a dictionary containing the current settings for all widget options. For each option key in the dictionary, the value is either a five-tuple (option, option database key, option database class, default value, current value), or a two-tuple (option alias, option). The latter case is used for aliases like **bg** (background) and **bd** (borderwidth).

Note that the value fields aren't correctly formatted for some option types. See the description of the [keys](#) method for more information, and a workaround.

configure(cnf=None, **kw) [<#>]

Same as [config](#).

deletecommand(name) [<#>]

The deletecommand method.

name

destroy() [<#>]

Destroys the widget. The widget is removed from the screen, and all resources associated with the widget are released.

event_add(virtual, *sequences) [<#>]

The event_add method.

virtual
**sequences*

event_delete(virtual, *sequences) [<#>]

The event_delete method.

virtual
**sequences*

event_generate(sequence, **kw) [<#>]

The event_generate method.

sequence
***kw*

event_info(virtual=None) [<#>]

The event_info method.

virtual

focus() [<#>]

The focus method.

focus_displayof() [<#>]

The focus_displayof method.

focus_force() [<#>]

The focus_force method.

focus_get() [<#>]

The focus_get method.

focus_lastfor() [<#>]

The focus_lastfor method.

focus_set() [<#>]

Moves the keyboard focus to this widget. This means that all keyboard events sent to the application will be routed to this widget.

getboolean(s) [<#>]

(Internal) Converts a Tk string to a boolean (flag) value.

getvar(name='PY_VAR') [<#>]

The getvar method.

name

grab_current() [<#>]

The grab_current method.

grab_release() [<#>]

Releases the event grab.

grab_set() [<#>]

Routes all events for this application to this widget.

grab_set_global() [<#>]

Routes all events for the entire screen to this widget.

This should only be used in very special circumstances, since it blocks all other applications running on the same screen. And that probably includes your development environment, so you better make sure your application won't crash or lock up until it has properly released the grab.

grab_status() [<#>]

The grab_status method.

image_names() [<#>]

The image_names method.

image_types() [<#>]

The image_types method.

keys() [<#>]

Returns a tuple containing the options available for this widget. You can use

[cget](#) to get the corresponding value for each option.

Note that the tuple currently include option aliases (like **bd**, **bg**, and **fg**). To avoid this, you can use [config](#) instead. On the other hand, [config](#) doesn't return valid option values for some option types (such as font names), so the best way is to use a combination of [config](#) and [cget](#):

```
for item in w.config():
    if len(item) == 5:
        option = item[0]
        value = w.cget(option)
        print option, value
```

lift(aboveThis=None) [<#>]

Moves the widget to the top of the window stack. If the widget is a child window, it is moved to the top of it's toplevel window. If it is a toplevel window (the root or a Toplevel window), it is moved in front of all other windows on the display. If an argument is given, the widget (or window) is moved so it's just above the given widget (window).

lower(belowThis=None) [<#>]

Moves the window to the bottom of the window stack. Same as [lift](#), but moves the widget to the bottom of the stack (or places it just under the belowThis widget).

mainloop(n=0) [<#>]

Enters Tkinter's main event loop. To leave the event loop, use the [quit](#) method. Event loops can be nested; it's ok to call mainloop from within an event handler.

nametowidget(name) [<#>]

Gets the widget object corresponding to a widget name.

name

The widget name.

Returns:

The corresponding widget object, if known.

Raises **KeyError**:

If the widget could not be found.

option_add(pattern, value, priority=None) [<#>]

The option_add method.

pattern

value

priority

option_clear() [<#>]

The option_clear method.

option_get(name, className) [<#>]

The option_get method.

name

className

option_readfile(fileName, priority=None) [<#>]

The option_readfile method.

fileName
priority

pack_propagate(flag=['_noarg_']) [<#>]

The pack_propagate method.

flag

pack_slaves() [<#>]

The pack_slaves method.

place_slaves() [<#>]

The place_slaves method.

propagate(flag=['_noarg_']) [<#>]

The propagate method.

flag

quit() [<#>]

The quit method.

register(func, subst=None, needcleanup=1) [<#>]

Registers a Tcl to Python callback. Returns the name of a Tcl wrapper procedure. When that procedure is called from a Tcl program, it will call the corresponding Python function with the arguments given to the Tcl procedure. Values returned from the Python callback are converted to strings, and returned to the Tcl program.

rowconfigure(index, cnf={}, **kw) [<#>]

The rowconfigure method.

index
cnf
***kw*

selection_clear(kw)** [<#>]

The selection_clear method.

***kw*

selection_get(kw)** [<#>]

The selection_get method.

***kw*

selection_handle(command, **kw) [<#>]

The selection_handle method.

command
***kw*

selection_own(kw)** [<#>]

The selection_own method.

***kw*

selection_own_get(kw)** [<#>]

The selection_own_get method.

***kw*

send(interp, cmd, *args) [<#>]

The send method.

interp
cmd
**args*

setvar(name='PY_VAR', value='1') [<#>]

The setvar method.

name
value

size() [<#>]

The size method.

slaves() [<#>]

The slaves method.

tk_bisque() [<#>]

The tk_bisque method.

tk_focusFollowsMouse() [<#>]

The tk_focusFollowsMouse method.

tk_focusNext() [<#>]

Returns the next widget (following self) that should have focus. This is used by the default bindings for the **Tab** key.

tk_focusPrev() [<#>]

Returns the previous widget (preceding self) that should have focus. This is used by the default bindings for the **Shift-Tab** key.

tk_menuBar(*args) [<#>]

The tk_menuBar method.

**args*

tk_setPalette(*args, **kw) [<#>]

The tk_setPalette method.

**args*
***kw*

tk_strictMotif(boolean=None) [<#>]

The tk_strictMotif method.

boolean

tkraise(aboveThis=None) [<#>]

The tkraise method.

aboveThis

unbind(sequence, funcid=None) [<#>]

Removes any bindings for the given event sequence, for this widget.

unbind_all(sequence) [<#>]

The unbind_all method.

sequence

unbind_class(className, sequence) [<#>]

The unbind_class method.

className

sequence

update() [<#>]

Processes all pending events, calls event callbacks, completes any pending geometry management, redraws widgets as necessary, and calls all pending idle tasks. This method should be used with care, since it may lead to really nasty race conditions if called from the wrong place (from within an event callback, for example, or from a function that can in any way be called from an event callback, etc.). When in doubt, use [update_idletasks](#) instead.

update_idletasks() [<#>]

Calls all pending idle tasks, without processing any other events. This can be used to carry out geometry management and redraw widgets if necessary, without calling any callbacks.

wait_variable(name) [<#>]

Waits for the given Tkinter variable to change. This method enters a local event loop, so other parts of the application will still be responsive. The local event loop is terminated when the variable is updated (setting it to its current value also counts).

wait_visibility(window=None) [<#>]

Wait for the given widget to become visible. This is typically used to wait until a new toplevel window appears on the screen. Like [wait_variable](#), this method enters a local event loop, so other parts of the application will still work as usual.

wait_window(window=None) [<#>]

Waits for the given widget to be destroyed. This is typically used to wait until a

destroyed window disappears from the screen. Like [wait_variable](#) and [wait_visibility](#), this method enters a local event loop, so other parts of the application will still work as usual.

waitvar(name='PY_VAR') [<#>]

The waitvar method.

name

winfo_atom(name, displayof=0) [<#>]

Maps the given string to a unique integer. Every time you call this method with the same string, the same integer will be returned.

winfo_atomname(id, displayof=0) [<#>]

Returns the string corresponding to the given integer (obtained by a call to [winfo_atom](#)). If the integer isn't in use, Tkinter raises a TclError exception. Note that Tkinter predefines a bunch of integers (typically 1-80 or so). If you're curious, you can use this method to find out what they are used for.

winfo_cells() [<#>]

Returns the number of "cells" in the color map for self. This is typically a value between 2 and 256 (also for true color displays, for some odd reason).

winfo_children() [<#>]

Returns a list containing widget instances for all children of this widget. The windows are returned in stacking order from bottom to top. If the order doesn't matter, you can get the same information from the children widget attribute (it's a dictionary mapping Tk widget names to widget instances, so widget.children.values() gives you a list of instances).

winfo_class() [<#>]

Returns the Tkinter widget class name for this widget. If the widget is a Tkinter base widget, widget.winfo_class() is the same as widget.__class__.__name__.

winfo_colormapfull() [<#>]

Returns true if the color map for this widget is full.

winfo_containing(rootX, rootY, displayof=0) [<#>]

Returns the widget at the given position, or None if there is no such window, or it isn't owned by this application. The coordinates are given relative to the screen's upper left corner.

rootX

rootY

displayof

winfo_depth() [<#>]

Returns the bit depth used to display this widget. This is typically 8 for a 256-color display device, 15 or 16 for a "hicolor" display, and 24 or 32 for a true color display.

winfo_exists() [<#>]

Returns true if there is Tk window corresponding to this widget. Unless you've done something really strange, this method should always return true.

wininfo_fpixels(distance) [#]

Converts the given distance (in any form accepted by Tkinter) to the corresponding number of pixels.

distance

The screen distance.

Returns:

The corresponding number of screen pixels, as a floating point number.

wininfo_geometry() [#]

Returns a string describing the widget's "geometry". The string has the following format:

```
"%dx%d%d%d" % (width, height, xoffset, yoffset)
```

wininfo_height() [#]

Get the height of this widget, in pixels. Note that if the window isn't managed by a geometry manager, this method returns 1. To you get the real value, you may have to call [update_idletasks](#) first. You can also use [wininfo_reqheight](#) to get the widget's requested height (that is, the "natural" size as defined by the widget itself based on it's contents).

Returns:

The widget's current height, in pixels.

wininfo_id() [#]

Get a system-specific window identifier for this widget. For Unix, this is the X window identifier. For Windows, this is the HWND cast to a long integer.

Returns:

The window identifier.

wininfo_interps(displayof=0) [#]

The wininfo_interps method.

displayof

wininfo_ismapped() [#]

Check if the window has been created. This method checks if Tkinter has created a window corresponding to the widget in the underlying window system (an X window, a Windows HWND, etc).

Returns:

A true value if a window has been created.

wininfo_manager() [#]

Return the name of the geometry manager used to keep manage this widget (typically one of grid, pack, place, canvas, or text).

wininfo_name() [#]

Get the Tk widget name. This is the same as the last part of the full widget name (which you can get via **str(widget)**).

Returns:

The widget name.

wininfo_parent() [#]

Get the full widget name of this widget's parent. This method returns an empty string if the widget doesn't have a parent (if it's a root window or a toplevel, that is).

To get the widget instance instead, you can simply use the **master** attribute instead of calling this method (the **master** attribute is **None** for the root window). Or if you insist, use **nametowidget** to map the full widget name to a widget instance.

Returns:

The widget name, as a string.

wininfo_pathname(id, displayof=0) [#]

Get the full window name for the window having the given identity (see **wininfo_id** for details). If the window doesn't exist, or it isn't owned by this application, Tkinter raises a **TclError** exception.

To convert the full name to a widget instance, use **nametowidget**.

id

The window identifier.

displayof

wininfo_pixels(distance) [#]

Convert the given distance (in any form accepted by Tkinter) to the corresponding number of pixels.

distance

The screen distance.

Returns:

The corresponding number of screen pixels, as an integer.

wininfo_pointerx() [#]

The wininfo_pointerx method.

wininfo_pointerxy() [#]

The wininfo_pointerxy method.

wininfo_pointery() [#]

The wininfo_pointery method.

wininfo_reqheight() [#]

Returns the “natural” height for this widget. The natural size is the minimal size needed to display the widget's contents, including padding, borders, etc. This size is calculated by the widget itself, based on the given options. The actual widget size is then determined by the widget's geometry manager, based on this value, the size of the widget's master, and the options given to the geometry manager.

wininfo_reqwidth() [#]

Returns the “natural” width for this widget. The natural size is the minimal size needed to display the widget's contents, including padding, borders, etc. This

size is calculated by the widget itself, based on the given options. The actual widget size is then determined by the widget's geometry manager, based on this value, the size of the widget's master, and the options given to the geometry manager.

wininfo_rgb(color) [<#>]

Convert a color string (in any form accepted by Tkinter) to an RGB tuple.

color

A colour string. This can be a colour name, a string containing an rgb specifier (“#rrggb”), or any other syntax supported by Tkinter.

Returns:

A 3-tuple containing the corresponding red, green, and blue components. Note that the tuple contains 16-bit values (0..65535).

wininfo_rootx() [<#>]

Get the pixel coordinate for the widget's left edge, relative to the screen's upper left corner.

Returns:

The root coordinate.

wininfo_rooty() [<#>]

Get the pixel coordinates for the widget's upper edge, relative to the screen's upper left corner.

Returns:

The root coordinate.

wininfo_screen() [<#>]

Get the X window screen name for the current window. The screen name is a string having the format “:display.screen”, where display and screen are decimal numbers.

On Windows and Macintosh, this is always “:0.0”.

Returns:

The screen name.

wininfo_screencells() [<#>]

Get the number of “color cells” in the default color map for this widget's screen.

Returns:

The number of color cells.

wininfo_screendepth() [<#>]

Get the default bit depth for this widget's screen.

Returns:

The bit depth.

wininfo_screenheight() [<#>]

Get the height of this widget's screen, in pixels.

Returns:
The height, in pixels.

wininfo_screenmmheight() [<#>]

Get the height of this widget's screen, in millimetres. This may not be accurate on all platforms.

Returns:
The height, in millimetres.

wininfo_screenmmwidth() [<#>]

Get the width of this widget's screen, in millimetres. This may not be accurate on all platforms.

Returns:
The width, in millimetres.

wininfo_screenvisual() [<#>]

Get the "visual" type used for this widget. This is typically "pseudocolor" (for 256-color displays) or "truecolor" (for 16- or 24-bit displays).

Returns:
A string containing the visual type. This is one of "pseudocolor", "truecolor", "directcolor", "staticcolor", "grayscale", or "staticgray".

wininfo_screenwidth() [<#>]

Get the width of this widget's screen, in pixels.

Returns:
The width, in pixels.

wininfo_server() [<#>]

The wininfo_server method.

wininfo_toplevel() [<#>]

Get the toplevel window (or root) window for this widget, as a widget instance.

Returns:
The toplevel parent widget. This is either a Tk instance, or a Toplevel instance.

wininfo_viewable() [<#>]

The wininfo_viewable method.

wininfo_visual() [<#>]

Same as wininfo_screenvisual.

wininfo_visualid() [<#>]

The wininfo_visualid method.

wininfo_visualsavailable(includeids=o) [<#>]

The wininfo_visualsavailable method.

includeids

winfo_vrootheight() [<#>]

The winfo_vrootheight method.

winfo_vrootwidth() [<#>]

The winfo_vrootwidth method.

winfo_vrootx() [<#>]

The winfo_vrootx method.

winfo_vrooty() [<#>]

The winfo_vrooty method.

winfo_width() [<#>]

Get the width of this widget, in pixels. Note that if the window isn't managed by a geometry manager, this method returns 1. To you get the real value, you may have to call [update_idletasks](#) first. You can also use [winfo_reqwidth](#) to get the widget's requested width (that is, the "natural" size as defined by the widget itself based on it's contents).

Returns:

The widget's current width, in pixels.

winfo_x() [<#>]

Returns the pixel coordinates for the widgets's left corner, relative to its parent's left corner.

winfo_y() [<#>]

Returns the pixel coordinates for the widgets's upper corner, relative to its parent's upper corner.

[back](#) [next](#)

 rendered by a [django](#) application. hosted by [webfaction](#).

[back](#) [next](#)

Toplevel Window Methods

This group of methods are used to communicate with the window manager. They are available on the root window (**Tk**), as well as on all **Toplevel** instances.

Note that different window managers behave in different ways. For example, some window managers don't support icon windows, some don't support window groups, etc.

Patterns

Displaying Windows

Window Properties

Icons

Reference

Wm (class) [<#>]

Wm implementation class. This class is used as a mixin by the root (**Tk**) and [Toplevel](#) (dead link) widgets.

aspect(minNumer=None, minDenom=None, maxNumer=None, maxDenom=None) [<#>]

Controls the aspect ratio (the relation between width and height) of this window. The aspect ratio is constrained to lie between minNumer/minDenom and maxNumer/maxDenom.

If no arguments are given, this method returns the current constraints as a 4-tuple, if any.

Same as [wm_aspect](#).

attributes(*args) [<#>]

Sets or gets window attributes. In Python 2.4, this wrapper is incomplete, and the application must use a non-standard syntax to modify and query the available attributes.

Same as [wm_attributes](#).

**args*

One or more attribute specifiers. The current version doesn't support keyword arguments. Instead, you have to prefix the attribute name with a hyphen, and pass in the value as a separate argument (e.g. to set the "disabled" option, use **attribute("-disabled", 1)** instead of **attribute(disabled=1)**). If only an attribute name is given, the method returns the current value. If no arguments are given, the method returns the current attribute settings (see below for details).

alpha=

(Windows, Mac) Controls window transparency. 0.0 means fully transparent, 1.0 means fully opaque. This isn't supported on all systems; where not supported, Tkinter always uses 1.0. Note that in this release, this attribute must be given as "-alpha".

disabled=

(Windows) If set, disables the entire window. Note that in this release, this attribute must be given as "-disabled".

modified=

(Mac) If set, the window is flagged as modified. Note that in this release, this attribute must be given as “-modified”.

titlepath=

(Mac) The path to the window proxy icon. Note that in this release, this attribute must be given as “-titlepath”.

toolwindow=

(Windows) If set, sets the window style to a “tool window”. Note that in this release, this attribute must be given as “-toolwindow”.

topmost=

(Windows) If set, this window is always placed on top of other windows. Note that in this release, this attribute must be given as “-topmost”.

Returns:

If an attribute value was specified, the return value is undefined. If a single attribute name was specified, this is the current attribute value (as a string). If no arguments were specified, this currently returns the current attribute values as a string, in the format “-attribute value - attribute value”. Future versions may return a dictionary instead.

client(name=None) [<#>]

Sets or gets the WM_CLIENT_MACHINE property. This property is used by window managers under the X window system. It is ignored on other platforms.

To remove the property, set it to an empty string.

Same as [wm_client](#).

name

The new value for this property. If omitted, the current value is returned.

colormapwindows(*wlist) [<#>]

Sets or gets the WM_COLORMAP_WINDOWS property. This property is used by window managers under the X window system. It is ignored on other platforms.

Same as [wm_colormapwindows](#).

wlist

The new value for this property. If omitted, the current value is returned.

command(value=None) [<#>]

Sets or gets the WM_COMMAND property. This property is used by window managers under the X window system. It is ignored on other platforms.

Same as [wm_command](#).

wlist

The new value for this property. If omitted, the current value is returned. To remove the property, pass in an empty string.

deiconify() [<#>]

Displays the window. New windows are displayed by default, so you only have to use this method if you have used [iconify](#) or [withdraw](#) to remove the window from the screen.

Same as [wm_deiconify](#).

focusmodel(model=None) [<#>]

Sets or gets the focus model.

Same as [wm_focusmodel](#).

frameO [<#>]

Returns a string containing a system-specific window identifier corresponding to the window's outermost parent. For Unix, this is the X window identifier. For Windows, this is the HWND cast to a long integer.

Note that if the window hasn't been reparented by the window manager, this method returns the window identifier corresponding to the window itself.

Same as [wm_frame](#).

geometry(geometry=None) [<#>]

Sets or gets the window geometry. If called with an argument, this changes the geometry. The argument should have the following format:

```
"%dx%d+%d+%d" % (width, height, xoffset, yoffset)
```

To convert a geometry string to pixel coordinates, you can use something like this:

```
import re
def parsegeometry(geometry):
    m = re.match("(\\d+)x(\\d+) ([-+]?\\d+) ([-+]?\\d+)", geometry)
    if not m:
        raise ValueError("failed to parse geometry string")
    return map(int, m.groups())
```

[wm_geometry](#).

geometry

The new geometry setting. If omitted, the current setting is returned.

grid(baseWidth=None, baseHeight=None, widthInc=None, heightInc=None) [<#>]

The grid method. Same as [wm_grid](#).

group(window=None) [<#>]

Adds window to the window group controlled by the given window. A group member is usually hidden when the group owner is iconified or withdrawn (the exact behavior depends on the window manager in use).

Same as [wm_group](#).

window

The group owner. If omitted, the current owner is returned.

iconbitmap(bitmap=None) [<#>]

Sets or gets the icon bitmap to use when this window is iconified. This method is ignored by some window managers (including Windows).

Note that this method can only be used to display monochrome icons. To display a color icon, put it in a Label widget and display it using the [iconwindow](#) method instead.

Same as [wm_iconbitmap](#).

iconifyO [<#>]

Turns the window into an icon (without destroying it). To redraw the window,

use [deiconify](#). Under Windows, the window will show up in the taskbar.

When the window has been iconified, the [state](#) method returns “iconic”.

Same as [wm iconify](#).

iconmask(bitmap=None) [#]

Sets or gets the icon bitmap mask to use when this window is iconified. This method is ignored by some window managers (including Windows).

Same as [wm iconmask](#).

iconname(newName=None) [#]

Sets or gets the icon name to use when this window is iconified. This method is ignored by some window managers (including Windows).

Same as [wm iconname](#).

iconposition(x=None, y=None) [#]

Sets or gets the icon position hint to use when this window is iconified. This method is ignored by some window managers (including Windows).

Same as [wm iconposition](#).

iconwindow(window=None) [#]

Sets or gets the icon window to use as an icon when this window is iconified. This method is ignored by some window managers (including Windows).

Same as [wm iconwindow](#).

window

The new icon window. If omitted, the current window is returned.

maxsize(width=None, height=None) [#]

Sets or gets the maximum size for this window.

Same as [wm maxsize](#).

minsize(width=None, height=None) [#]

Sets or gets the minimum size for this window.

Same as [wm minsize](#).

overrideredirect(flag=None) [#]

Sets or gets the override redirect flag. If non-zero, this prevents the window manager from decorating the window. In other words, the window will not have a title or a border, and it cannot be moved or closed via ordinary means.

Same as [wm overrideredirect](#).

positionfrom(who=None) [#]

Sets or gets the position controller

Same as [wm positionfrom](#).

protocol(name=None, func=None) [#]

Registers a callback function for the given protocol. The name argument is typically one of “WM_DELETE_WINDOW” (the window is about to be deleted),

“WM_SAVE_YOURSELF” (called by X window managers when the application should save a snapshot of its working set) or “WM_TAKE_FOCUS” (called by X window managers when the application receives focus).

Same as [wm_protocol](#).

resizable(width=None, height=None) [#]

Sets or gets the resize flags. The width flag controls whether the window can be resized horizontally by the user. The height flag controls whether the window can be resized vertically.

Same as [wm_resizable](#).

sizefrom(who=None) [#]

Sets or gets the size controller

Same as [wm_sizefrom](#).

state(newstate=None) [#]

Sets or gets the window state. This is one of the values “normal”, “iconic” (see [iconify](#)), “withdrawn” (see [withdraw](#)) or “icon” (see [iconwindow](#)).

Same as [wm_state](#).

title(string=None) [#]

Sets or gets the window title.

Same as [wm_title](#).

title

The new window title. If omitted, the current title is returned.

transient(master=None) [#]

Makes window a transient window for the given master (if omitted, master defaults to self's parent). A transient window is always drawn on top of its master, and is automatically hidden when the master is iconified or withdrawn. Under Windows, transient windows don't show up in the task bar.

Same as [wm_transient](#).

withdraw() [#]

Removes the window from the screen (without destroying it). To redraw the window, use [deiconify](#).

When the window has been withdrawn, the [state](#) method returns “withdrawn”.

Same as [wm_withdraw](#).

wm_aspect(minNumer=None, minDenom=None, maxNumer=None, maxDenom=None) [#]

Controls the aspect ratio. See [aspect](#) for details.

wm_attributes(*args) [#]

Sets or gets window attributes. See [attributes](#) for details.

wm_client(name=None) [#]

Sets or gets the WM_CLIENT_MACHINE property. See [client](#) for details.

wm_colormapwindows(*wlist) [<#>]

Sets or gets the WM_COLORMAP_WINDOWS property. See [colormapwindows](#) for details.

wm_command(value=None) [<#>]

Sets or gets the WM_COMMAND property. See [command](#) for details.

wm_deiconify() [<#>]

Displays the window. See [deiconify](#) for details.

wm_focusmodel(model=None) [<#>]

Sets or gets the focus model. See [focusmodel](#) for details.

wm_frame() [<#>]

Returns a window identifier corresponding for the window's outermost parent. See [frame](#) for details.

wm_geometry(newGeometry=None) [<#>]

Sets or gets the window geometry. See [geometry](#) for details.

wm_grid(baseWidth=None, baseHeight=None, widthInc=None, heightInc=None) [<#>]

See [grid](#) for details.

wm_group(pathName=None) [<#>]

Adds the window to a window group. See [group](#) for details.

wm_iconbitmap(bitmap=None) [<#>]

Sets or gets the icon bitmap. See [iconbitmap](#) for details.

wm_iconify() [<#>]

Turns the window into an icon. See [iconify](#) for details.

wm_iconmask(bitmap=None) [<#>]

Sets or gets the icon bitmap mask. See [iconmask](#) for details.

wm_iconname(newName=None) [<#>]

Sets or gets the icon name. See [iconname](#) for details.

wm_iconposition(x=None, y=None) [<#>]

Sets or gets the icon position hint. See [iconposition](#) for details.

wm_iconwindow(pathName=None) [<#>]

Sets or gets the icon window. See [iconwindow](#) for details.

wm_maxsize(width=None, height=None) [<#>]

Sets or gets the maximum size. See [maxsize](#) for details.

wm_minsize(width=None, height=None) [<#>]

Sets or gets the minimum size. See [minsize](#) for details.

wm_overrideredirect(boolean=None) [<#>]

Sets or gets the override redirect flag. See [overrideredirect](#) for details.

wm_positionfrom(who=None) [#]

See [positionfrom](#) for details.

wm_protocol(name=None, func=None) [#]

Registers a callback function for the given protocol. See [protocol](#) for details.

wm_resizable(width=None, height=None) [#]

Sets or gets the resize flags. See [resizable](#) for details.

wm_sizefrom(who=None) [#]

See [sizefrom](#) for details.

wm_state(newstate=None) [#]

Sets or gets the window state. See [state](#) for details.

wm_title(string=None) [#]

Sets or gets the window title. See [title](#) for details.

wm_transient(master=None) [#]

Makes window a transient window for a given master. See [transient](#) for details.

wm_withdraw() [#]

Removes the window from the screen. See [withdraw](#) for details.

[back](#) [next](#)

 rendered by a [django](#) application. hosted by [webfaction](#).

[back](#) [next](#)

The Tkinter Grid Geometry Manager

The **Grid** geometry manager puts the widgets in a 2-dimensional table. The master widget is split into a number of rows and columns, and each “cell” in the resulting table can hold a widget.

When to use the Grid Manager

The grid manager is the most flexible of the geometry managers in Tkinter. If you don’t want to learn how and when to use all three managers, you should at least make sure to learn this one.

The grid manager is especially convenient to use when designing dialog boxes. If you’re using the packer for that purpose today, you’ll be surprised how much easier it is to use the grid manager instead. Instead of using lots of extra frames to get the packing to work, you can in most cases simply pour all the widgets into a single container widget, and use the grid manager to get them all where you want them. (I tend to use two containers; one for the dialog body, and one for the button box at the bottom.)

Consider the following example:

<label 1>	<entry 2>	<image>	
<label 1>	<entry 2>		
<checkboxbutton>		<button 1>	<button 2>

Creating this layout using the pack manager is possible, but it takes a number of extra frame widgets, and a lot of work to make things look good. If you use the grid manager instead, you only need one call per widget to get everything laid out properly (see next section for the code needed to create this layout).

Warning: Never mix grid and pack in the same master window. Tkinter will happily spend the rest of your lifetime trying to negotiate a solution that both managers are happy with. Instead of waiting, kill the application, and take another look at your code. A common mistake is to use the wrong parent for some of the widgets.

Patterns

Using the grid manager is easy. Just create the widgets, and use the **grid** method to tell the manager in which row and column to place them. You don’t have to specify the size of the grid beforehand; the manager automatically determines that from the widgets in it.

```
Label(master, text="First").grid(row=0)
Label(master, text="Second").grid(row=1)

e1 = Entry(master)
e2 = Entry(master)

e1.grid(row=0, column=1)
e2.grid(row=1, column=1)
```

Note that the column number defaults to 0 if not given.

Running the above example produces the following window:

Simple grid example



Empty rows and columns are ignored. The result would have been the same if you had placed the widgets in row 10 and 20 instead.

Note that the widgets are centered in their cells. You can use the **sticky** option to change this; this option takes one or more values from the set **N, S, E, W**. To align the labels to the left border, you could use **W** (west):

```
Label(master, text="First").grid(row=0, sticky=W)
Label(master, text="Second").grid(row=1, sticky=W)

e1 = Entry(master)
e2 = Entry(master)

e1.grid(row=0, column=1)
e2.grid(row=1, column=1)
```

Using the sticky option



You can also have the widgets span more than one cell. The **columnspan** option is used to let a widget span more than one column, and the **rowspan** option lets it span more than one row. The following code creates the layout shown in the previous section:

```
label1.grid(sticky=E)
label2.grid(sticky=E)

entry1.grid(row=0, column=1)
entry2.grid(row=1, column=1)

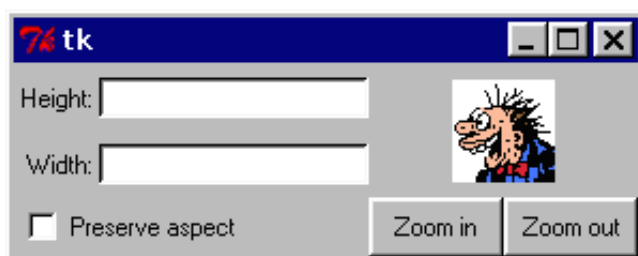
checkboxbutton.grid(columnspan=2, sticky=W)

image.grid(row=0, column=2, columnspan=2, rowspan=2,
           sticky=W+E+N+S, padx=5, pady=5)

button1.grid(row=2, column=2)
button2.grid(row=2, column=3)
```

There are plenty of things to note in this example. First, no position is specified for the label widgets. In this case, the column defaults to 0, and the row to the *first unused row in the grid*. Next, the entry widgets are positioned as usual, but the checkbox widget is placed on the next empty row (row 2, in this case), and is configured to span two columns. The resulting cell will be as wide as the label and entry columns combined. The image widget is configured to span both columns and rows at the same time. The buttons, finally, is packed each in a single cell:

Using column and row spans



Reference

Grid (class) [#]

Grid geometry manager. This is an implementation class; all the methods described below are available on all widget classes.

grid(options) [<#>]**

Place the widget in a grid as described by the options.

****options**

Geometry options.

column=

Insert the widget at this column. Column numbers start with 0. If omitted, defaults to 0.

columnspan=

If given, indicates that the widget cell should span multiple columns. The default is 1.

in=

Place widget inside to the given widget. You can only place a widget inside its parent, or in any descendant of its parent. If this option is not given, it defaults to the parent.

Note that **in** is a reserved word in Python. To use it as a keyword option, append an underscore (**in_**).

in_ =

Same as in. See above.

ipadx=

Optional horizontal internal padding. Works like **padx**, but the padding is added *inside* the widget borders. Default is 0.

ipady=

Optional vertical internal padding. Works like **pady**, but the padding is added *inside* the widget borders. Default is 0.

padx=

Optional horizontal padding to place around the widget in a cell. Default is 0.

pady=

Optional vertical padding to place around the widget in a cell. Default is 0.

row=

Insert the widget at this row. Row numbers start with 0. If omitted, defaults to the first empty row in the grid.

rowspan=

If given, indicates that the widget cell should span multiple rows. Default is 1.

sticky=

Defines how to expand the widget if the resulting cell is larger than the widget itself. This can be any combination of the constants **S**, **N**, **E**, and **W**, or **NW**, **NE**, **SW**, and **SE**.

For example, **W** (west) means that the widget should be aligned to the left cell border. **W+E** means that the widget should be stretched horizontally to fill the whole cell. **W+E+N+S** means that the widget should be expanded in both directions. Default is to center the widget in the cell.

grid_bbox(column=None, row=None, col2=None, row2=None) [<#>]

The grid_bbox method.

column

row

col2

row2

grid_columnconfigure(index, **options) [#]

Set options for a cell column.

To change this for a given widget, you have to call this method on the widget's parent.

index

Column index.

***options*

Column options.

minsize=

Defines the minimum size for the column. Note that if a column is completely empty, it will not be displayed, even if this option is set.

pad=

Padding to add to the size of the largest widget in the column when setting the size of the whole column.

weight=

A relative weight used to distribute additional space between columns. A column with the weight 2 will grow twice as fast as a column with weight 1. The default is 0, which means that the column will not grow at all.

grid_configure(options) [#]**

Same as [grid](#).

grid_forget() [#]

Remove this widget from the grid manager. The widget is not destroyed, and can be displayed again by **grid** or any other manager.

grid_info() [#]

Return a dictionary containing the current cell options for the cell used by this widget.

Returns:

A dictionary containing grid management options.

grid_location(x, y) [#]

Returns the grid cell under (or closest to) a given pixel.

x

y

Returns:

A tuple containing the column and row index.

grid_propagate(flag) [#]

Enables or disables geometry propagation. When enabled, a grid manager connected to this widget attempts to change the size of the widget whenever a child widget changes size. Propagation is always enabled by default.

flag

True to enable propagation.

grid_remove() [#]

Remove this widget from the grid manager. The widget is not destroyed, and can be displayed again by **grid** or any other manager.

grid_rowconfigure(index, **options) [#]

Set options for a row of cells.

To change this for a given widget, you have to call this method on the widget's parent.

index

Row index.

***options*

Row options.

minsize=

Defines the minimum size for the row. Note that if a row is completely empty, it will not be displayed, even if this option is set.

pad=

Padding to add to the size of the largest widget in the row when setting the size of the whole row.

weight=

A relative weight used to distribute additional space between rows. A row with the weight 2 will grow twice as fast as a row with weight 1. The default is 0, which means that the row will not grow at all.

grid_size() [#]

Returns the current grid size for the geometry manager attached to this widget. This is defined as indexes of the first empty column and row in the grid, in that order.

Returns:

A 2-tuple containing the number of columns and rows.

grid_slaves(row=None, column=None) [#]

Returns a list of the “slave” widgets managed by this widget. The widgets are returned as Tkinter widget references.

Returns:

A list of widgets.

[back](#) [next](#)



rendered by a [django](#) application. hosted by [webfaction](#).

[back](#) [next](#)

The Tkinter Pack Geometry Manager

The **Pack** geometry manager packs widgets in rows or columns. You can use options like **fill**, **expand**, and **side** to control this geometry manager.

The manager handles all widgets that are packed inside the same master widget. The packing algorithm is simple, but a bit tricky to describe in words; imagine a sheet of some elastic material, with a very small rectangular hole in the middle. For each widget, in the order they are packed, the geometry manager makes the hole large enough to hold the widget, and then place it against a given inner edge (default is the top edge). It then repeats the process for all widgets. Finally, when all widgets have been packed into the hole, the manager calculates the bounding box for all widgets, makes the master widget large enough to hold all widgets, and moves them all to the master.

When to use the Pack Manager

Compared to the **grid** manager, the pack manager is somewhat limited, but it's much easier to use in a few, but quite common situations:

1. Put a widget inside a frame (or any other container widget), and have it fill the entire frame
2. Place a number of widgets **on top of each other**
3. Place a number of widgets **side by side**

See the [Patterns](#) section for code examples.

If you need to create more complex layouts, you usually have to group the widgets using extra **Frame** widgets. You can also use the **grid** manager instead.

Note: Don't mix grid and pack in the same master window. Tkinter will happily spend the rest of your lifetime trying to negotiate a solution that both managers are happy with. Instead of waiting, kill the application, and take another look at your code. A common mistake is to use the wrong parent for some of the widgets.

Patterns

Filling the entire parent widget

A common situation is when you want to place a widget inside a container widget, and have it fill the entire parent. Here's a simple example: a listbox placed in the root window:

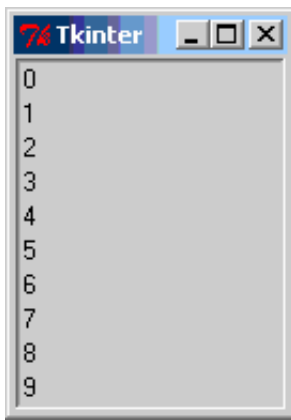
```
from Tkinter import *

root = Tk()

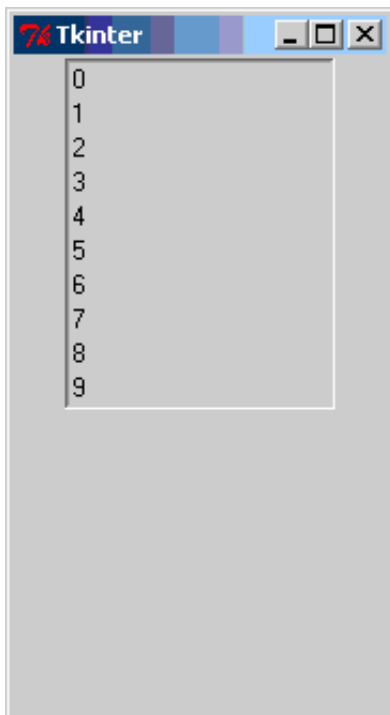
listbox = Listbox(root)
listbox.pack()

for i in range(20):
    listbox.insert(END, str(i))

mainloop()
```



By default, the listbox is made large enough to show 10 items. But this listbox contains twice as many. But if the user attempts to show them all by resizing the window, Tkinter will add padding around the listbox:



To make the widget fill the entire parent, also if the user resizes the window, add **fill** and **expand** options:

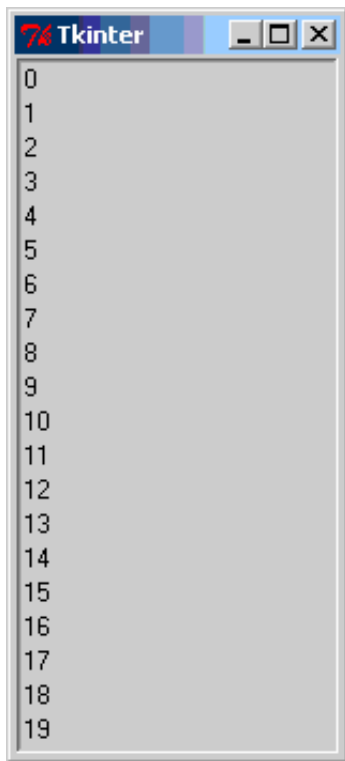
```
from Tkinter import *

root = Tk()

listbox = Listbox(root)
listbox.pack(fill=BOTH, expand=1)

for i in range(20):
    listbox.insert(END, str(i))

mainloop()
```



The **fill** option tells the manager that the widget wants fill the entire space assigned to it. The value controls how to fill the space; **BOTH** means that the widget should expand both horizontally and vertically, **X** means that it should expand only horizontally, and **Y** means that it should expand only vertically.

The **expand** option tells the manager to assign additional space to the widget box. If the parent widget is made larger than necessary to hold all packed widgets, any exceeding space will be distributed among all widgets that have the **expand** option set to a non-zero value.

Placing a number of widgets on top of each other

To put a number of widgets in a column, you can use the **pack** method without any options:

```
from Tkinter import *

root = Tk()

w = Label(root, text="Red", bg="red", fg="white")
w.pack()
w = Label(root, text="Green", bg="green", fg="black")
w.pack()
w = Label(root, text="Blue", bg="blue", fg="white")
w.pack()

mainloop()
```



You can use the **fill=X** option to make all widgets as wide as the parent widget:

```
from Tkinter import *

root = Tk()

w = Label(root, text="Red", bg="red", fg="white")
w.pack(fill=X)
w = Label(root, text="Green", bg="green", fg="black")
w.pack(fill=X)
```

```
w = Label(root, text="Blue", bg="blue", fg="white")
w.pack(fill=X)

mainloop()
```



Placing a number of widgets side by side

To pack widgets side by side, use the **side** option. If you wish to make the widgets as high as the parent, use the **fill=Y** option too:

```
from Tkinter import *

root = Tk()

w = Label(root, text="Red", bg="red", fg="white")
w.pack(side=LEFT)
w = Label(root, text="Green", bg="green", fg="black")
w.pack(side=LEFT)
w = Label(root, text="Blue", bg="blue", fg="white")
w.pack(side=LEFT)

mainloop()
```



Reference

Pack (class) [#]

Pack geometry manager. This is an implementation class; all the methods described below are available on all widget classes.

pack(**options) [#]

Pack the widget as described by the options.

****options**

Geometry options.

anchor=

Where the widget is placed inside the packing box. Default is CENTER.

expand=

Specifies whether the widgets should be expanded to fill any extra space in the geometry master. If false (default), the widget is not expanded.

fill=

Specifies whether the widget should occupy all the space provided to it by the master. If **NONE** (default), keep the widget's original size. If **X** (fill horizontally), **Y** (fill vertically), or **BOTH**, fill the given space along that direction.

To make a widget fill the entire master widget, set **fill** to **BOTH** and **expand** to a non-zero value.

in=

Pack this widget inside the given widget. You can only pack a widget inside its parent, or in any descendant of its parent. This option should usually be left out, in which case the widget is packed inside its parent. Note that **in** is a reserved word in Python. To use it as a keyword option, append an underscore (**in_**).

ipadx=

Internal padding. Default is 0.

ipady=

Internal padding. Default is 0.
padx= External padding. Default is 0.
pady= External padding. Default is 0.
side= Specifies which side to pack the widget against. To pack widgets vertically, use **TOP** (default). To pack widgets horizontally, use **LEFT**. You can also pack widgets along the **BOTTOM** and **RIGHT** edges. You can mix sides in a single geometry manager, but the results may not always be what you expect. While you can create pretty complicated layouts by nesting **Frame** widgets, you may prefer using the **grid** geometry manager for non-trivial layouts.

pack_configure(options)** [<#>]

Same as [pack](#).

pack_forget() [<#>]

Removes the widget from its current manager. The widget is not destroyed, and can be displayed again by **pack** or any other manager.

pack_info() [<#>]

Returns a dictionary containing the current packer options.

Returns:

A dictionary mapping packer option names to option values.

pack_propagate(flag) [<#>]

(Manager method) Controls geometry propagation. If enabled, the manager widget will be resized if not large enough to hold all the child widgets.

Note that this method should be called on the master widget, not on an individual child widget.

pack_slaves() [<#>]


(Manager method) Returns a list of all child (“slave”) widgets managed by the packer for this widget.

Note that this method should be called on the master widget, not on an individual child widget.

Returns:

A list of child widgets.

[back](#) [next](#)

 rendered by a [django](#) application. hosted by [webfaction](#).

[back](#) [next](#)

The Tkinter Place Geometry Manager

The **Place** geometry manager is the simplest of the three general geometry managers provided in Tkinter. It allows you explicitly set the position and size of a window, either in absolute terms, or relative to another window.

You can access the place manager through the **place** method which is available for all standard widgets.

When to use the Place Manager

It is usually not a good idea to use **place** for ordinary window and dialog layouts; its simply to much work to get things working as they should. Use the **pack** or **grid** managers for such purposes.

However, **place** has its uses in more specialized cases. Most importantly, it can be used by compound widget containers to implement various custom geometry managers. Another use is to position control buttons in dialogs.

Patterns

Let's look at some usage patterns. The following command centers a widget in its parent:

```
w.place(relx=0.5, rely=0.5, anchor=CENTER)
```

Here's another variant. It packs a **Label** widget in a frame widget, and then places a **Button** in the upper right corner of the frame. The button will overlap the label.

```
pane = Frame(master)
Label(pane, text="Pane Title").pack()
b = Button(pane, width=12, height=12,
          image=launch_icon, command=self.launch)
b.place(relx=1, x=-2, y=2, anchor=NE)
```

The following excerpt from a **Notepad** widget implementation displays a notepad page (implemented as a **Frame**) in the notepad body frame. It first loops over the available pages, calling **place_forget** for each one of them. Note that it's not an error to "unplace" a widget that it's not placed in the first case:

```
for w in self.__pages:
    w.place_forget()
self.__pages[index].place(in_=self.__body, x=bd, y=bd)
```

You can combine the absolute and relative options. In such cases, the relative option is applied first, and the absolute value is then added to that position. In the following example, the widget *w* is almost completely covers its parent, except for a 5 pixel border around the widget.

```
w.place(x=5, y=5, relwidth=1, relheight=1, width=-10, height=-10)
```

You can also place a widget outside another widget. For example, why not place two widgets on top of each other:

```
w2.place(in_=w1, relx=0.5, y=-2, anchor=S, bordermode="outside")
```

Note the use of **relx** and **anchor** options to center the widgets vertically. You could also use (relx=0, anchor=SW) to get left alignment, or (relx=1, anchor=SE) to get right alignment.

By the way, why not combine this way to use the packer with the launch button example shown earlier. The following example places two buttons in the upper right corner of the *pane*:

```
b1 = DrawnButton(pane, (12, 12), launch_icon, command=self.launch)
b1.place(relx=1, x=-2, y=2, anchor=NE)
b2 = DrawnButton(pane, (12, 12), info_icon, command=self.info)
b2.place(in_=b1, x=-2, anchor=NE, bordermode="outside")
```

Finally, let's look at a piece of code from an imaginary **SplitWindow** container widget. The following piece of code splits *frame* into two subframes called *f1* and *f2*.

```
f1 = Frame(frame, bd=1, relief=SUNKEN)
f2 = Frame(frame, bd=1, relief=SUNKEN)
split = 0.5
f1.place(rely=0, relheight=split, relwidth=1)
f2.place(rely=split, relheight=1.0-split, relwidth=1)
```

To change the split point, set *split* to something suitable, and call the **place** method again. If you haven't changed an option, you don't have to specify it again.

```
f1.place(relheight=split)
f2.place(rely=split, relheight=1.0-split)
```

You could add a small frame to use as a dragging handle, and add suitable bindings to it, e.g:

```
f3 = Frame(frame, bd=2, relief=RAISED, width=8, height=8)
f3.place(relx=0.9, rely=split, anchor=E)
f3.bind("B1-Motion>", self.adjust)
```

Reference

Place (class) [#]

Place geometry manager. This is an implementation class; all the methods described below are available on all widget classes.

place(**options) [#]

Place this widget relative to its parent.

****options**

Geometry options.

anchor=

Default is NW.

bordermode=

Default is INSIDE.

height=

No default value.

in=

Default is ..

relheight=

No default value.

relwidth=

No default value.

relx=

Default is 0.

rely=

Default is 0.

width=

No default value.

x=

Default is 0.

y=

Default is 0.

place_configure(options)** [<#>]

Same as [place](#).

place_forget() [<#>]

The place_forget method.

place_info() [<#>]

The place_info method.


place_slaves() [<#>]

The place_slaves method.

slaves() [<#>]

Same as [place_slaves](#).

[back](#) [next](#)

 rendered by a [django](#) application. hosted by [webfaction](#).

[back](#) [next](#)

The Tkinter BitmapImage Class

The **BitmapImage** class provides a simple image class, for monochrome (two-color) images.

When to use the BitmapImage Class

This class can be used to display bitmap images in labels, buttons, canvases, and text widgets.

The bitmap loader reads X11 bitmap files. To use other formats, use the [PhotoImage](#) class.

Patterns

An X11 bitmap image consists of a C fragment that defines a width, a height, and a data array containing the bitmap. To embed a bitmap in a Python program, you can put it inside a triple-quoted string:

```
BITMAP = """
#define im_width 32
#define im_height 32
static char im_bits[] = {
0xaf,0x6d,0xeb,0xd6,0x55,0xdb,0xb6,0x2f,
0xaf,0xaa,0x6a,0x6d,0x55,0x7b,0xd7,0x1b,
0xad,0xd6,0xb5,0xae,0xad,0x55,0x6f,0x05,
0xad,0xba,0xab,0xd6,0xaa,0xd5,0x5f,0x93,
0xad,0x76,0x7d,0x67,0x5a,0xd5,0xd7,0xa3,
0xad,0xbd,0xfe,0xea,0x5a,0xab,0x69,0xb3,
0xad,0x55,0xde,0xd8,0x2e,0x2b,0xb5,0x6a,
0x69,0x4b,0x3f,0xb4,0x9e,0x92,0xb5,0xed,
0xd5,0xca,0x9c,0xb4,0x5a,0xa1,0x2a,0x6d,
0xad,0x6c,0x5f,0xda,0x2c,0x91,0xbb,0xf6,
0xad,0xaa,0x96,0xaa,0x5a,0xca,0x9d,0xfe,
0x2c,0xa5,0x2a,0xd3,0x9a,0x8a,0x4f,0xfd,
0x2c,0x25,0x4a,0x6b,0x4d,0x45,0x9f,0xba,
0x1a,0xaa,0x7a,0xb5,0xaa,0x44,0x6b,0x5b,
0x1a,0x55,0xfd,0x5e,0x4e,0xa2,0x6b,0x59,
0x9a,0xa4,0xde,0x4a,0x4a,0xd2,0xf5,0xaa
};
"""
```

To create X11 bitmaps, you can use the X11 **bitmap** editor provided with most Unix systems, or draw your image in some other drawing program and convert it to a bitmap using e.g. [the Python Imaging Library](#).

The BitmapImage class can read X11 bitmaps from strings or text files:

```
bitmap = BitmapImage(data=BITMAP)

bitmap = BitmapImage(file="bitmap.xbm")
```

By default, foreground (non-zero) pixels in the bitmap are drawn in black, and background (zero) pixels are made transparent. You can use the **foreground** and **background** options to override this behaviour:

```
bitmap = BitmapImage(
    data=BITMAP,
    foreground="white", background="black"
)
```

You can draw two-colour transparent bitmaps by associating a mask image to the bitmap. The mask must be an X11 bitmap of the same size as the main bitmap. Background (zero) pixels in the mask are always made transparent, independent of the foreground and background colour settings:

```
bitmap = BitmapImage(
```

```
data=BITMAP,  
foreground="black", background="yellow",  
maskdata=MASK_BITMAP  
)
```

You can use a BitmapImage instance everywhere Tkinter accepts an image object. An example:

```
label = Label(image=bitmap)  
label.pack()
```

Note: When a BitmapImage object is garbage-collected by Python (e.g. when you return from a function which stored a bitmap in a local variable), the bitmap is cleared even if it's displayed by a Tkinter widget.

To avoid this, the program must keep an extra reference to the bitmap object. One way to do this is to assign the bitmap to a widget attribute, like this:

```
label = Label(image=bitmap)  
label.image = bitmap # keep a reference!  
label.pack()
```

You can change the bitmap options after you've created the object. To modify an option, use the **config** method, or the `[]` operator. To get the current value of an option, use the `[]` operator. The **cget** method cannot be used for bitmap objects.

```
bitmap.config(foreground="blue")  
bitmap["foreground"] = "red"  
print bitmap["foreground"]
```

[back](#) [next](#)

 rendered by a [django](#) application. hosted by [webfaction](#).

[back](#) [next](#)

The Tkinter PhotoImage Class

The **PhotoImage** class is used to display images (either grayscale or true color images) in labels, buttons, canvases, and text widgets.

When to use the PhotoImage Class

You can use the PhotoImage class whenever you need to display an icon or an image in a Tkinter application.

Patterns

The PhotoImage class can read GIF and PGM/PPM images from files:

```
photo = PhotoImage(file="image.gif")

photo = PhotoImage(file="lenna.pgm")
```

The PhotoImage can also read base64-encoded GIF files from strings. You can use this to embed images in Python source code (use functions in the **base64** module to convert binary data to base64-encoded strings):

```
photo = """
R0lGODdhEAAQAIcAAAAAAEBAQICAgMDAwQEBAUFBQYGBGcHBwgICAkJCQoKCgsLCwwMDA0NDQ4O
Dg8PDxAQEERERERISEhMTExQUFBUVFRYWfHcXFxgYGBkZGRoaGhsbGxwcHB0dHR4eHh8fHyAgICEh
...
AfjHtqlbAP/i/gPwry4AAP/yAtj77x+Af4ABAwDwrzAAAP8SA/j3DwCAfwAA/Jsm4J/lfwD+/QMA
4B8AAP9Ci/4HoLTpfwD+qV4NoHVAADs=
"""

photo = PhotoImage(data=photo)
```

If you need to work with other file formats, the [Python Imaging Library](#) (PIL) contains classes that lets you load images in over 30 formats, and convert them to Tkinter-compatible image objects:

```
from PIL import Image, ImageTk

image = Image.open("lenna.jpg")
photo = ImageTk.PhotoImage(image)
```

You can use a PhotoImage instance everywhere Tkinter accepts an image object. An example:

```
label = Label(image=photo)
label.image = photo # keep a reference!
label.pack()
```


You must keep a reference to the image object in your Python program, either by storing it in a global variable, or by attaching it to another object.

Note: When a PhotoImage object is garbage-collected by Python (e.g. when you return from a function which stored an image in a local variable), the image is cleared even if it's being displayed by a Tkinter widget.

To avoid this, the program must keep an extra reference to the image object. A simple way to do this is to assign the image to a widget attribute, like this:

```
label = Label(image=photo)
label.image = photo # keep a reference!
label.pack()
```

[back](#) [next](#)

 rendered by a [django](#) application. hosted by [webfaction](#).

[back](#)

The Variable Classes (BooleanVar, DoubleVar, IntVar, StringVar)

If you program Tk using the Tcl language, you can ask the system to let you know when a variable is changed. The Tk toolkit can use this feature, called *tracing*, to update certain widgets when an associated variable is modified.

There's no way to track changes to Python variables, but Tkinter allows you to create variable wrappers that can be used wherever Tk can use a traced Tcl variable.

When to use the Variable Classes

Variables can be used with most entry widgets to track changes to the entered value. The Checkbutton and Radiobutton widgets require variables to work properly.

Variables can also be used to validate the contents of an entry widget, and to change the text in label widgets.

Patterns

To create a Tkinter variable, call the corresponding constructor:

```
var = StringVar()
```

Note that the constructor takes an optional widget argument, but no value argument; to set the value, call the **set** method:

```
var = StringVar()
var.set("hello")
```

The constructor argument is only relevant if you're running Tkinter with multiple Tk instances (which you shouldn't do, unless you really know what you're doing).

You can use the **trace** method to attach "observer" callbacks to the variable. The callback is called whenever the contents change:

```
def callback(*args):
    print "variable changed!"

var = StringVar()
var.trace("w", callback)
var.set("hello")
```

FIXME: add Entry/Label/OptionMenu patterns

Methods

get/set

get() => value

set(string)

The **get** method returns the current value of the variable, as a Python object. For **BooleanVar** variables, the returned value is 0 for false, and 1 for true. For **DoubleVar** variables, the returned value is a Python float. For **IntVar**, it's an integer. For **StringVar**, it's either an ASCII string or a Unicode string, depending on the contents.

The **set** method updates the variable, and notifies all variable observers. You can either pass in a value of the right type, or a string.

trace

trace(mode, callback) => string

trace_variable(mode, callback)

Add a variable observer. Returns the internal name of the observer (you can use this to unregister the observer; see below).

The **mode** argument is one of “r” (call observer when variable is read by someone), “w” (call when variable is written by someone), or “u” (undefine; call when the variable is deleted).

FIXME: describe the mode argument and how the callback should look, and when it is called.

trace_vdelete

trace_vdelete(mode, observer name)


Remove an observer. The observer name is the string returned by **trace_variable**, when the observer was first registered.

trace_vinfo

trace_vinfo() => list

FIXME: add description

[back](#)

 rendered by a [django](#) application. hosted by [webfaction](#).