

An Introduction to Tkinter

Fredrik Lundh

An Introduction to Tkinter
by Fredrik Lundh

Copyright © 1999 by Fredrik Lundh

Table of Contents

Preface	i
I. Introducing Tkinter.....	1
What's Tkinter?.....	1
Hello, Tkinter.....	2
Running the Example	2
Details	2
Hello, Again	4
Running the Example	4
Details	4
More on widget references	5
More on widget names.....	6
Tkinter Classes.....	7
Widget classes.....	7
Mixins	8
Implementation mixins.....	8
Geometry mixins	8
Widget configuration management	8
Widget Configuration	9
Configuration Interface.....	9
widgetclass.....	9
cget	9
configure	9
keys.....	9
Compatibility	10
Widget Styling.....	11
Colors.....	11
Color Names	11
RGB Specifications	11
Fonts	12
Font descriptors	12
Font names	13
System fonts	14
X Font Descriptors	14
Text Formatting	14
Borders.....	15
Relief.....	15
Focus Highlights	15
Cursors.....	15
Events and Bindings	16
Events	16
The Event Object.....	18
Instance and Class Bindings.....	18
Protocols.....	19
Other Protocols	20
Application Windows.....	21
Base Windows.....	21
The Work Area	21

Menus	21
Toolbars	21
Status Bars.....	21
Standard Dialogs.....	23
Message Boxes	23
Message Box Options	24
Data Entry	24
Strings.....	25
Numeric Values.....	25
File Names	26
Colors.....	27
Dialog Windows	28
Grid Layouts	32
Validating Data	34
II. Tkinter Reference	35
The Button Widget	35
Button Patterns.....	35
Methods	36
flash	36
invoke.....	36
Helpers.....	36
Options.....	36
The Canvas Widget.....	39
Concepts.....	39
Items.....	39
Coordinate Systems	39
Item Specifiers	40
Printing	40
Patterns.....	41
Methods	41
create_arc.....	41
create_bitmap.....	41
create_image	41
create_line.....	41
create_oval.....	41
create_polygon.....	41
create_rectangle.....	41
create_text.....	41
create_window.....	41
delete.....	41
itemcget.....	42
itemconfig.....	42
coords.....	42
coords.....	42
bbox	42
canvasx.....	42
tag_bind.....	42
tag_unbind	42
type	42
lift	43
lower	43

move	43
postscript.....	43
scale	44
Searching for Items	44
find_above.....	44
find_all.....	44
find_below.....	44
find_closest	44
find_enclosed	44
find_overlapping.....	44
find_withtag.....	44
Manipulating Tags.....	44
addtag_above.....	45
addtag_all.....	45
addtag_below	45
addtag_closest	45
addtag_enclosed	45
addtag_overlapping.....	45
addtag_withtag.....	45
dtag.....	45
gettags.....	45
Special Methods for Text Items	45
dchars.....	45
focus.....	46
icursor.....	46
index	46
insert.....	46
select_adjust.....	46
select_clear	46
select_from.....	46
select_item	46
select_to.....	46
Scrolling	46
scan_mark, scan_dragto	46
xview, yview.....	47
xview, yview.....	47
Options.....	47
The Canvas Arc Item.....	49
Methods	49
create_arc.....	49
delete.....	49
coords.....	49
itemconfigure	49
Options.....	49
The Canvas Bitmap Item.....	51
Methods	51
create_bitmap	51
delete.....	51
coords.....	51
itemconfigure	51
Options.....	51

The Canvas Image Item.....	53
Methods	53
create_image	53
delete.....	53
coords.....	53
itemconfigure	53
Options.....	53
The Canvas Line Item	54
Methods	54
create_line	54
delete.....	54
coords.....	54
itemconfigure	54
Options.....	54
The Canvas Oval Item	56
Methods	56
create_oval	56
delete.....	56
coords.....	56
itemconfigure	56
Options.....	56
The Canvas Polygon Item.....	57
Methods	57
create_polygon.....	57
delete.....	57
coords.....	57
itemconfigure	57
Options.....	57
The Canvas Rectangle Item	59
Methods	59
create_rectangle.....	59
delete.....	59
coords.....	59
itemconfigure	59
Options.....	59
The Canvas Text Item	60
Methods	60
create_text.....	60
delete.....	60
coords.....	60
itemconfigure	60
Options.....	60
The Canvas Window Item	62
Methods	62
create_window.....	62
delete.....	62
coords.....	62
itemconfigure	62
Options.....	62
The Checkbutton Widget.....	63
Checkbutton Patterns	63

Methods	64
deselect.....	64
flash	64
invoke	64
select	64
toggle	64
Options.....	64
The Entry Widget	67
Concepts.....	67
Indexes.....	67
Patterns.....	67
Methods	67
insert.....	67
delete.....	67
icursor.....	68
get.....	68
index	68
Selection Methods	68
selection_adjust	68
selection_clear	68
selection_from	68
selection_present	68
selection_range	68
selection_to	68
Scrolling Methods.....	68
scan_mark, scan_dragto	69
xview.....	69
xview_moveto	69
xview_scroll.....	69
Options.....	69
The Font Class.....	71
Patterns.....	71
Methods	71
copy	71
actual	71
cget	71
config, configure.....	71
measure.....	71
metrics	71
Functions.....	71
families.....	71
names.....	71
Options.....	72
The Frame Widget.....	73
Patterns.....	73
Methods	73
Options.....	73
The Grid Geometry Manager	75
When to use the Grid Manager	75
Patterns.....	75
Widget Methods	77

grid	77
grid_forget.....	77
grid_info	77
grid_remove	77
Manager Methods.....	77
columnconfigure, rowconfigure	77
grid_location	78
grid_propagate	78
grid_size	78
grid_slaves	78
Options.....	78
The Label Widget	80
Patterns.....	80
Methods	80
Options.....	80
The Listbox Widget.....	83
Patterns.....	83
Methods	85
activate.....	85
bbox	86
curselection.....	86
delete.....	86
get.....	86
index	86
insert.....	86
nearest.....	86
see.....	86
size.....	86
Selection Methods	86
select_adjust.....	87
select_anchor.....	87
select_clear	87
select_includes.....	87
select_set.....	87
Scrolling Methods.....	87
scan_mark, scan_dragto	87
xview, yview	87
xview, yview	87
xview, yview	87
xview, yview	88
Options.....	88
The Menu Widget.....	90
Patterns.....	90
Methods	90
add.....	90
add_cascade	91
add_checkbutton	91
add_command.....	91
add_radiobutton.....	91
add_separator.....	91
delete.....	91

entryconfig.....	91
entryconfigure.....	91
index	91
insert.....	91
insert_cascade	91
insert_checkbutton.....	91
insert_command.....	91
insert_radiobutton.....	92
insert_separator.....	92
invoke	92
post	92
unpost	92
yposition.....	92
Helpers	92
Options.....	92
The Menubutton Widget	94
Patterns.....	94
Methods	94
Options.....	94
The Message Widget.....	95
Patterns.....	95
Methods	95
Options.....	95
The Pack Geometry Manager.....	97
When to use the Pack Manager	97
Patterns.....	97
Methods	97
Widget Methods	97
pack.....	97
pack_forget.....	97
pack_info()	97
Manager Methods.....	97
pack_propagate()	98
pack_slaves()	98
Options.....	98
The PhotoImage Class.....	99
Patterns.....	99
Methods	99
configure	99
cget	99
width, height.....	99
type	99
get.....	99
put	99
read.....	99
write.....	99
blank	100
copy	100
zoom	100
subsample.....	100
Options.....	100

The Place Geometry Manager.....	102
When to use place.....	102
Patterns.....	102
Methods	103
place.....	103
place_forget.....	103
place_info.....	103
place_slaves	104
Options.....	104
The Radiobutton Widget.....	105
Radiobutton Patterns.....	105
Methods	106
deselect.....	106
flash	106
invoke	106
select	106
Options.....	106
The Scale Widget.....	109
Patterns.....	109
Methods	109
get, set	109
Options.....	109
The Scrollbar Widget.....	111
Patterns.....	111
Methods	111
delta	111
destroy	111
fraction.....	111
get.....	111
identify	111
keys.....	111
set	111
Options.....	111
The Text Widget	114
Concepts.....	114
Indexes.....	114
Marks	116
Tags.....	117
Patterns.....	119
Basic Methods.....	121
insert.....	121
delete.....	121
get.....	121
dump.....	121
see.....	121
index	121
compare.....	121
Methods for Marks	121
mark_set	122
mark_unset	122
index	122

mark_gravity	122
mark_names.....	122
Methods for Embedded Windows.....	122
window_create.....	122
index	123
delete.....	123
window_cget.....	123
window_config.....	123
window_names	123
Methods for Embedded Images	124
image_create	124
index	124
delete.....	124
image_cget	124
image_config	125
image_names.....	125
Methods for Tags.....	125
tag_add	125
tag_remove.....	125
tag_delete	125
tag_config.....	125
tag_cget.....	125
tag_bind.....	126
tag_unbind	126
tag_names	126
tag_nextrange.....	126
tag_prevrange.....	126
tag_lower.....	126
tag_raise.....	126
tag_ranges.....	127
Methods for Selections.....	127
Methods for Rendering	127
bbox	127
dlineinfo	127
Methods for Printing	127
Methods for Searching.....	128
search.....	128
Methods for Scrolling	128
scan_mark, scan_dragto	128
xview, yview.....	128
xview, yview.....	129
xview, yview.....	129
yview_pickplace	129
Options.....	129
The Toplevel Widget.....	132
Methods	132
Options.....	132
Basic Widget Methods	134
Configuration.....	134
config	134
config	134

cget	134
keys.....	134
Event processing	135
mainloop	135
quit	135
update	135
update_idletasks	135
focus_set	135
focus_displayof.....	135
focus_force	135
focus_get	135
focus_lastfor.....	135
tk_focusNext.....	135
tk_focusPrev	136
grab_current.....	136
grab_release	136
grab_set	136
grab_set_global	136
grab_status.....	136
wait_variable	136
wait_visibility	136
wait_window	136
Event callbacks.....	136
bind.....	137
unbind	137
bind_all.....	137
unbind_all.....	137
bind_class	137
unbind_class.....	137
bindtags.....	137
bindtags.....	137
Alarm handlers and other non-event callbacks	137
after	137
after_cancel.....	138
after	138
after_idle	138
Window management.....	138
lift	138
lower	138
Window Related Information.....	138
winfo_cells	139
winfo_children.....	139
winfo_class.....	139
winfo_colormapfull	139
winfo_containing	139
winfo_depth	139
winfo_exists.....	139
winfo_pixels	139
winfo_geometry	139
winfo_width, winfo_height	140
winfo_id.....	140

wininfo_ismapped.....	140
wininfo_manager.....	140
wininfo_name.....	140
wininfo_parent	140
wininfo_pathname	140
wininfo_reqheight, wininfo_reqwidth	140
wininfo_rootx, wininfo_rooty	141
wininfo_screen	141
wininfo_screencells.....	141
wininfo_screendepth.....	141
wininfo_screenwidth, wininfo_screenheight	141
wininfo_screenmmwidth, wininfo_screenmmheight	141
wininfo_screenuisual.....	141
wininfo_toplevel	141
wininfo_visual	141
wininfo_x, wininfo_y	142
Miscellaneous	142
bell.....	142
clipboard_append	142
clipboard_clear	142
selection_clear	142
selection_get.....	142
selection_handle	142
selection_own.....	142
selection_own_get	142
tk_focusFollowsMouse	142
tk_strictMotif.....	142
wininfo_rgb	143
Tkinter Interface Methods.....	143
getboolean	143
getdouble	143
getint.....	143
register	143
wininfo_atom	143
wininfo_atomname	143
Option Database.....	143
option_add	143
option_clear.....	144
option_get	144
option_readfile	144
Toplevel Window Methods.....	145
Visibility Methods	145
deiconify	145
iconify	145
withdraw	145
state	145
Style Methods	145
title.....	145
group.....	145
transient	145
overrideredirect.....	146

Window Geometry Methods.....	146
geometry	146
geometry	146
aspect	146
maxsize.....	146
minsize.....	146
resizable	146
Icon Methods	147
iconbitmap.....	147
iconmask	147
iconname.....	147
iconposition.....	147
iconwindow.....	147
Property Access Methods	147
client	147
colormapwindows	147
command.....	147
focusmodel	148
frame.....	148
positionfrom	148
protocol.....	148
sizefrom.....	148

Preface

This is yet another draft version of my ever-growing Tkinter documentation.

The biggest change since the last version is that I'm using a new editor to edit this document. As a result, the HTML version of this document looks a bit different. I've tried to keep most of the old chapter names, but most chapters have been split into several subpages. The styling has also changed; the document now looks better than before if you have a CSS-capable browser, but worse if you don't.

By the way, might be interested in hearing that O'Reilly & Associates will be publishing a Tkinter book (tentatively called *Programming Tkinter in Python*) later this year. This book features lots of brand new material written by yours truly, giving you a more thorough description of Tkinter (and many other things) than you can find anywhere else.

</F>

This document was last updated Jun 21, 1999.

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 most recent release (8.0), Tk also offers native look and feel on all platforms.

Tkinter consists of a number of modules. The Tk interface is located in a binary module named *_tkinter* (this was *tkinter* in earlier versions). 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.

In addition to the Tk interface module, Tkinter includes a number of Python modules. The two most important modules are the *Tkinter* itself, and a module called *Tkconstants*. The former automatically imports the latter, so to use Tkinter, all you need to do is to import one module:

```
import Tkinter
```

Or, more often:

```
from Tkinter import *
```

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:

Example 1. 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.

Figure 1. 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, which tells it to size itself to fit the given text, and make itself visible. But before this happens, we have to enter 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.

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*.

Example 1. File: hello2.py

```
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()
```

Running the Example

When you run this example, the following window appears.

Figure 1. 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.

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()
```

The last call is to the `mainloop` method on the root widget. It 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.

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.

Tkinter Classes

Widget classes

Tkinter support 15 core widgets:

Table 1. Tkinter Widget Classes

Widget	Description
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 behaviour.
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.

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.

Table 2. Geometry mixins

Manager	Description
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 <i>grid</i> 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 <i>pack</i> 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 <i>place</i> 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.

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

`widgetclass(master, option=value, ...)`. 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. Note that the name option can only be set when the widget is created.

cget

`cget(option)`. 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.

configure

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

`keys()`. 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).

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.

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 Eng
ms sans Serif 11 points: Pack my box with fiven dozen
ms serif 16 points: The quick brown fox ju
σμβολ 12 ποντα: Φιγυρε τηισ ουτ, ωονδερ βοω
system 10 points: Hello, Harry. Now there's the ε
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:

Table 1.

Option	Type	Description
family	string	Font family.
size	integer	Font size in points. To give the size in pixels, use a negative value.

weight	constant	Font thickness. Use one of <i>NORMAL</i> or <i>BOLD</i> . Default is <i>NORMAL</i> .
slant	constant	Font slant. Use one of <i>NORMAL</i> or <i>ITALIC</i> . Default is <i>NORMAL</i> .
underline	flag	Font underlining. If 1 (true), the font is underlined. Default is 0 (false).
overstrike	flag	Font strikeout. If 1 (true), a line is drawn over text written with this font. Default is 0 (false).

FIXME: *add more information about the tkFont module, or at least a link to the reference page*

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

justify, wrap

Borders

Relief

The relief settings control how to draw the widget border:

borderwidth, relief

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:

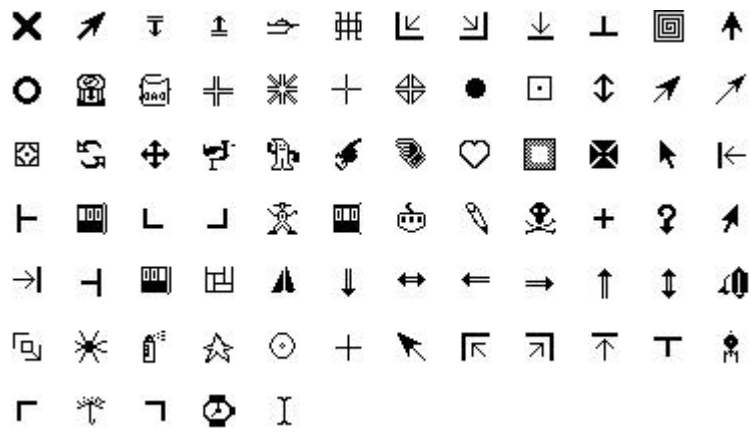
highlightcolor, highlightbackground, highlightthickness

Cursors

The cursor setting control 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, sets this option by default.

cursor



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:

Example 1. File: `bind1.py`

```
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.

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:

Table 1.

Event	Description
<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 mouse events will then be sent to the current widget as long as the mouse button is held down. The current position of the mouse pointer (relative to the widget) is provided in the <i>x</i> and <i>y</i> members of the event object passed to the callback. You can use <i>ButtonPress</i> instead of <i>Button</i> , 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 <i>x</i> and <i>y</i> members of the event object passed to the callback.
<Button-Release-1>	Button 1 was released. The current position of the mouse pointer is provided in the <i>x</i> and <i>y</i> members of the event object passed to the callback.
<Double-Button-1>	Button 1 was double clicked. You can use <i>Double</i> or <i>Triple</i> 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 <i>Enter</i> key!).
<Leave>	The mouse pointer left the 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 <i>Cancel</i> (the Break key), <i>BackSpace</i> , <i>Tab</i> , <i>Return</i> (the Enter key), <i>Shift_L</i> (any Shift key), <i>Control_L</i> (any Control key), <i>Alt_L</i> (any Alt key), <i>Pause</i> , <i>Caps_Lock</i> , <i>Escape</i> , <i>Prior</i> (Page Up), <i>Next</i> (Page Down), <i>End</i> , <i>Home</i> , <i>Left</i> , <i>Up</i> , <i>Right</i> , <i>Down</i> , <i>Print</i> , <i>Insert</i> , <i>Delete</i> , F1, F2, F3, F4, F5, F6, F7, F8, F9, F10, F11, F12, <i>Num_Lock</i> , and <i>Scroll_Lock</i> .
<Key>	The user pressed any key. The key is provided in the <i>char</i> 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 <i>Alt</i> , <i>Shift</i> , and <i>Control</i>
<Configure>	The widget changed size (or location, on some platforms). The new size is provided in the <i>width</i> and <i>height</i> attributes of the event object

Event	Description
	passed to the callback.

The Event Object

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

Table 2.

Attribute	Description
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.
char	The character code (keyboard events only).
keysym	The key symbol (keyboard events only).
keycode	The key code (keyboard events only)
width, height	The new size of the widget, in pixels (Configure events only).

FIXME: this list is not complete. See my mail on this topic for more details.

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 "best 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 behaviour 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 behaviour 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 behaviour 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 behaviour 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:

Example 2. File: protocol1.py

```
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 version 1.63, 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.

FIXME: *has this been changed in Python 1.5.2?*

Other Protocols

Window managers 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.

Application Windows

Coming soon.

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).

The Work Area

Menus

Toolbars

Status Bars

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)
```

The following class wraps a status bar label, providing *set* and *clear* methods:

Example 1. 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()
```

Note that 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.

[For compatibility with future versions of Tkinter, we've decided that it is not politically correct to inherit from any widgets other than the Toplevel and Frame widgets. Should definitely explain why somewhere in this document]

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 `askretryignore`. 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
```

Figure 1. showinfo, showwarning, showerror dialogs





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 if the user gave a positive answer (ok, yes, and retry, respectively).

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

Figure 2. askquestion dialog

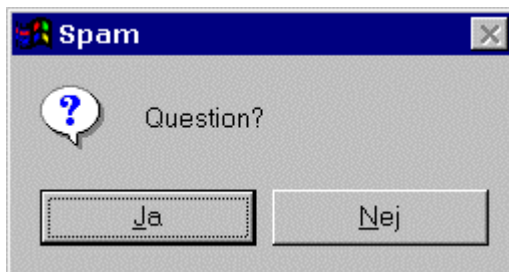
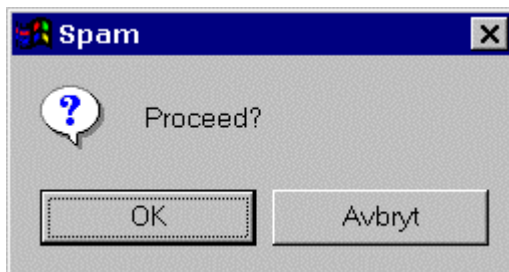


Figure 3. askokcancel, askyesno, askretryignore dialogs





[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).

Table 1. Message Box Options

Option	Type	Description
default	constant	Which button to make default: <i>ABORT</i> , <i>RETRY</i> , <i>IGNORE</i> , <i>OK</i> , <i>CANCEL</i> , <i>YES</i> , or <i>NO</i> (the constants are defined in the <i>tkMessageBox</i> module).
icon	constant	Which icon to display: <i>ERROR</i> , <i>INFO</i> , <i>QUESTION</i> , or <i>WARNING</i>
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: <i>ABORTRETRYIGNORE</i> , <i>OK</i> , <i>OKCANCEL</i> , <i>RETRYCANCEL</i> , <i>YESNO</i> , or <i>YESNOCANCEL</i> .

Data Entry

The *tkSimpleDialog* module provides an interface to the following simple dialogs.

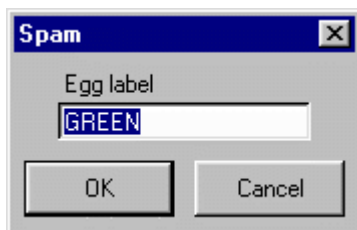
FIXME: where to find this module?

Strings

The *askstring* function in the *tkSimpleDialog* module prompts the user for a string. You specify the dialog title and the prompt string, and the function returns when the user closes the dialog. The prompt string may include newline characters.

`tkSimpleDialog.askstring(title, prompt [,options])`. Ask the user to enter a string value. If the user pressed Enter, or clicked OK, the function returns the string. If the user closed the dialog by pressing Escape, clicking Cancel, or explicitly via the window manager, this function returns *None*.

Figure 4. askstring



The following options can be used with this function:

Table 2. askstring Options

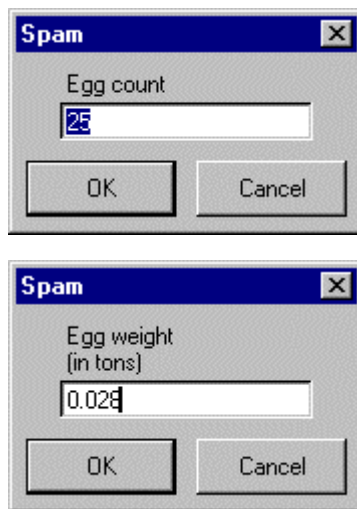
Option	Type	Description
initialvalue	string	Initial value, if any. Default is an empty string.
parent	widget	Which window to place the dialog on top of. When the dialog is closed, the focus is returned to the parent window.

Numeric Values

The *askinteger* and *askfloat* functions are similar to *askstring*, but they only accept integers and float values, respectively. You can also use the *minvalue* and *maxvalue* options to limit the input range:

`tkSimpleDialog.askinteger(title, prompt [,options])`. Ask the user to enter an integer value. If the entered value is not a valid integer or floating point value, a message box is displayed, and the dialog is not closed. As with the *askstring* function, the function returns *None* if the dialog box is cancelled.

`tkSimpleDialog.askfloat(title, prompt [,options])`. Same, but returns a floating point value.

Figure 5. askinteger, askfloat

The following options can be used with these functions:

Table 3. askinteger and askfloat options

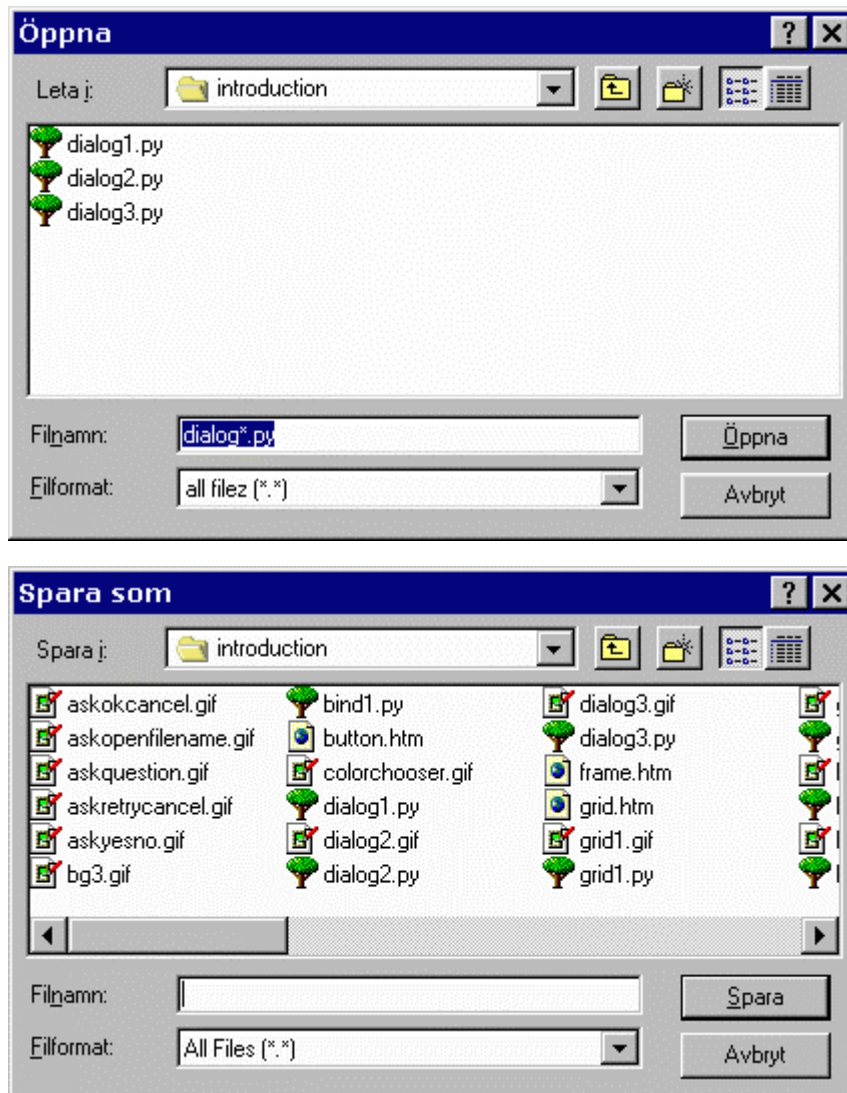
Option	Type	Description
initialvalue	integer or float	Initial value, if any. Default is an empty string.
parent	widget	Which window to place the dialog on top of. When the dialog is closed, the focus is returned to the parent window.
minvalue	integer or float	Minimum value. If exceeded, a message box is shown when the user clicks OK, and the dialog will not be closed. Default is no check.
maxvalue	integer or float	Maximum value. If exceeded, a message box is shown when the user clicks OK, and the dialog will not be closed. Default is no check.

File Names

The *tkFileDialog* module (included in the standard dialog kit described earlier) can be used to get a filename from the user. The module provides two convenience functions, one to get an existing filename so you can open it, and one to get a new filename, to save things into.

`tkFileDialog.askopenfilename([options])`. If the dialog is cancelled by the user, the function returns *None*.

`tkFileDialog.asksaveasfilename([options])`.

Figure 6. askopenfilename, asksaveasfilename

The following options can be used with the *askopenfilename* and *asksavefilename* functions:

Table 4. askopenfilename options

Option	Type	Description
defaultextension	string	An extension to add to the filename, if not explicitly given by the user. The string should include the leading dot (ignored by the open dialog).
filetypes	list	Sequence of (label, pattern) tuples. The same label may occur with several patterns. Use "*" as the pattern to indicate all files.
initialdir	string	Initial directory.
initialfile	string	Initial file (ignored by the open dialog)

Option	Type	Description
parent	widget	Which window to place the message box on top of. When the dialog is closed, the focus is returned to the parent window.
title	string	Message box title.

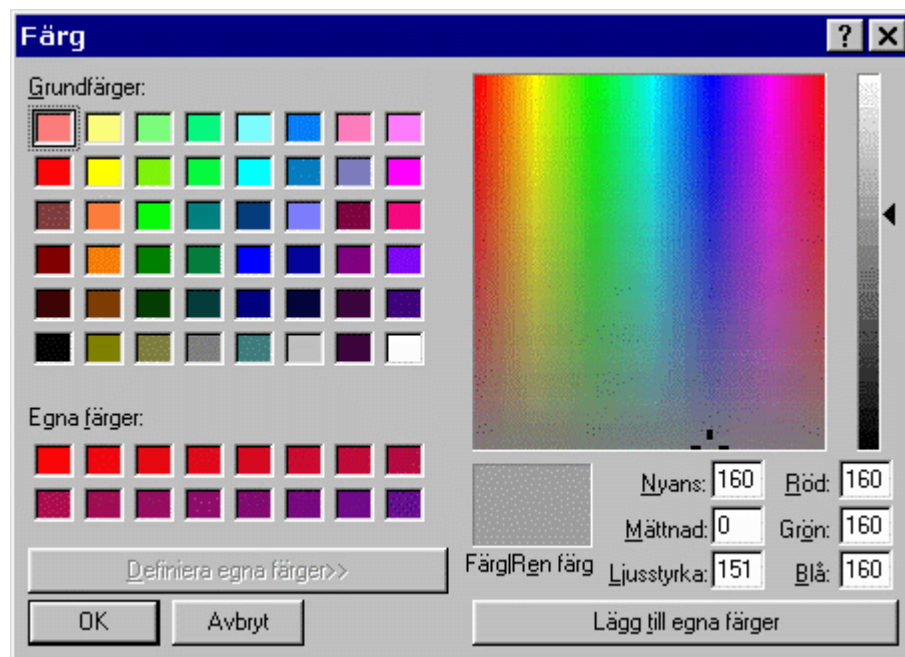
Colors

The `tkColorChooser` module (included in the standard dialog kit described earlier) can be used to specify an RGB color value.

`tkColorChooser.askcolor([color [,options]])`. The convenience function returns two values; the first is the color as a RGB triplet (a 3-tuple containing the red, green and blue values as integers in the range 0-255), the second a Tk color string. To preset a color when you display the dialog, you can pass a color (in either format) to the function.

If the dialog is cancelled, the function returns *(None, None)*

Figure 7. askcolor (in Swedish)



The following options can be used with the `askcolor` function:

Table 5. askcolor Options

Option	Type	Description
initialcolor	color	Color to mark as selected when dialog is displayed (given as an RGB triplet or a Tk color string). (the first argument to the convenience function).
parent	widget	Which window to place the message box on top of. When the dialog is closed, the focus is returned to

Option	Type	Description
		the parent window.
title	string	Message box title.

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.

Example 1. File: `dialog1.py`

```
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.

Example 2. 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).

Example 3. File: `dialog2.py`

```

class MyDialog(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)

```

```

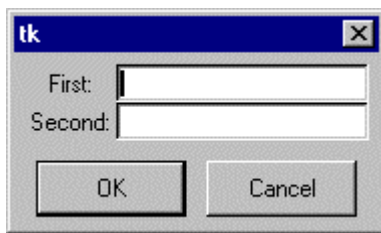
self.e1.grid(row=0, column=1)
self.e2.grid(row=1, column=1)
return self.e1 # initial focus

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

```

And here's the resulting dialog:

Figure 1. 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 = string.atoi(self.e1.get())
    second = string.atoi(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:

Figure 2. 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 manager takes care of the rest. The following *body* method shows how to get the above layout:

Example 4. File: dialog3.py

```
def body(self, master):

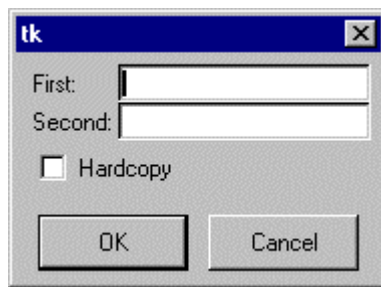
    Label(master, text="First:").grid(row=0, sticky=W)
    Label(master, text="Second:").grid(row=1, sticky=W)

    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:

Figure 3. 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 behaviour. 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 behaviour of the resulting layout. These include *padx* and *pady* which are used to add extra padding to widget cells, and many others. The *Grid Geometry Manager* 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 = string.atoi(self.e1.get())
        second = string.atoi(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= string.atoi(self.e1.get())
        second = string.atoi(self.e2.get())
        self.result = first, second
        return 1
    except ValueError:
        tkinterMessageBox.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.

The 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.

Button Patterns

Plain buttons are pretty straightforward to use. Simply specify the button contents (text, bitmap, or image) and a callback to call when the button is pressed:

```
b = Button(master, text="OK", command=self.ok)
```

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 actually specify the size in pixels even for text buttons, but it takes 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
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 the *wrplength* 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)
```

Methods

The *Button* widgets support the standard Tkinter Widget interface, plus the following methods:

flash

`flash()`. Redraw the button several times, alternating between active and normal appearance.

invoke

`invoke()`. Call the command associated with the button.

Helpers

The following methods are only relevant if you're implementing your own keyboard bindings. They are not documented in this version of the Tkinter overview.

`tkButtonDown()`.

`tkButtonEnter()`.

`tkButtonInvoke()`.

`tkButtonLeave()`.


`tkButtonUp()`. (*Button* only). These can be used in customized event bindings. All these methods accept zero or more dummy arguments.

Options

The *Button* widgets support the following options:

Table 1.

Option	Type	Description
activebackground, activeforeground	color	The color to use when the button is activated.
anchor	constant	Controls where in the button the text (or image) should be located. Use one of <i>N</i> , <i>NE</i> , <i>E</i> , <i>SE</i> , <i>S</i> , <i>SW</i> , <i>W</i> , <i>NW</i> , or <i>CENTER</i> . Default is <i>CENTER</i> . If you change this, it is probably a good idea to add some padding as well, using the <i>padx</i> and/or <i>pady</i> options.
background, foreground	color	The button color. The default is platform specific.
bitmap	bitmap	The bitmap to display in the widget. If the <i>image</i> option is given, this option is ignored.

Option	Type	Description
		<p>The following bitmaps are available on all platforms: "error", "gray75", "gray50", "gray25", "gray12", "hourglass", "info", "questhead", "question", and "warning".</p>  <p>The following additional bitmaps are available on the Macintosh only: "document", "stationery", "edition", "application", "accessory", "folder", "pfolder", "trash", "floppy", "ramdisk", "cdrom", "preferences", "querydoc", "stop", "note", and "caution".</p> <p>You can also load the bitmap from an XBM file. Just prefix the filename with an at-sign, for example "@sample.xbm".</p>
borderwidth (bd)	int	The width of the button border. The default is platform specific, but is usually 1 or 2 pixels.
command	callback	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.
cursor	cursor	The cursor to show when the mouse is moved over the button.
default	int	If set, the button is a default button. Tk will indicate this by drawing a platform specific indicator (usually an extra border). NOTE: The syntax has changed in 8.0b2!!!
disabledforeground	color	The color to use when the button is disabled. The background is shown in the <i>background</i> color.
font	font	The font to use in the button. The button can only contain text in a single font.
highlightbackground, highlightcolor	color	Controls how to draw the focus highlight border. When the widget has focus, the border is drawn in the <i>highlightcolor</i> color. Otherwise, it is drawn in the <i>highlightbackground</i> color. The defaults are system specific.
highlightthickness	distance	Controls the width of the focus highlight border. Default is typically one or two pixels.
image	image	The image to display in the widget. If specified, this takes precedence over the <i>text</i> and <i>bitmap</i> options.
justify	constant	Defines how to align multiple lines of text. Use <i>LEFT</i> , <i>RIGHT</i> , or <i>CENTER</i> .
padx, pady	distance	Button padding. These options specify the

Option	Type	Description
		horizontal and vertical padding between the text or image, and the button border.
relief	constant	Border decoration. Usually, the button is <i>SUNKEN</i> when pressed, and <i>RAISED</i> otherwise. Other possible values are <i>GROOVE</i> , <i>RIDGE</i> , and <i>FLAT</i> .
state	constant	The button state: <i>NORMAL</i> , <i>ACTIVE</i> or <i>DISABLED</i> . Default is <i>NORMAL</i> .
takefocus	flag	Indicates that the user can use the <i>Tab</i> 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).
text	string	The text to display in the button. The text can contain newlines. If the <i>bitmap</i> or <i>image</i> options are used, this option is ignored.
textvariable	variable	Associates a Tkinter variable (usually a <i>StringVar</i>) to the button. If the variable is changed, the button text is updated.
underline	int	Default is -1.
width, height	distance	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.
wrplength	distance	Determines when a button's text should be wrapped into multiple lines. This is given in screen units. Default is no wrapping.

The Canvas Widget

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

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 items.

Concepts

To be added.

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 are drawn as both an outline and an interior, either of which can be made transparent (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 in which 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 *canvay* methods:

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

Item Specifiers

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
- tags
- *ALL*
- *CURRENT*

Item handles are integer values that are 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.

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. All such operations are ignored.

You can either specify the tags via an option to the item create method, set them via the *itemconfig* method, or add them using the *addtag_withtag* method. The *tags* option take either a single 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 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 "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

To be added.

Patterns

To be added.

Methods

The first group of methods are used to create and configure items on a canvas.

create_arc

create_arc(bbox, options). Create an *arc* canvas item. Returns the item handle.

create_bitmap

create_bitmap(position, options). Create a *bitmap* canvas item. Returns the item handle.

create_image

create_image(position, options). Create an *image* canvas item. Returns the item handle.

create_line

create_line(coords, options). Create a *line* canvas item. Returns the item handle.

create_oval

create_oval(bbox, options). Create an *oval* canvas item. Returns the item handle.

create_polygon

create_polygon(coords, options). Create a *polygon* canvas item. Returns the item handle.

create_rectangle

create_rectangle(bbox, options). Create a *rectangle* canvas item. Returns the item handle.

create_text

create_text(position, options). Create a *text* canvas item. Returns the item handle.

create_window

create_window(position, options). Place a Tkinter widget on the canvas. Returns the item handle.

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.

delete

delete(items). Delete all matching items. It is not an error to give an item specifier that doesn't match any items.

itemcget

`itemcget(item, option)`. Get the current value for an option. If *item* refers to more than one items, this method returns the option value for the first item found.

itemconfig

`itemconfig(item, options)`, `itemconfigure(item, options)`. Change one or more options for all matching items.

coords

`coords(item)`. Return the coordinates for the given item. If *item* refers to more than one items, this method returns the type of the first item found.

coords

`coords(item, x0, y0, x1, y1, ..., xn, yn)`. Change the coordinates for the given item. This method updates all matching items.

bbox

`bbox(items)`, `bbox()`. Returns the bounding box for the given items. If the specifier is omitted, the bounding box for all items are returned. Note that the bounding box is approximate and may differ a few pixels from the real value.

canvasx

`canvasx(screenx)`, `canvasy(screeny)`. Convert a window coordinate (for example, the *x* and *y* coordinates from the structure passed to an event handler) to a canvas coordinate.

tag_bind

`tag_bind(item, sequence, callback)`, `tag_bind(item, sequence, callback, "+")`. Add an event binding to all matching items. Usually, the new binding replaces any existing binding for the same event sequence. The second form can be used to add the new callback to the existing binding.

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.

tag_unbind

`tag_unbind(item, sequence)`. Remove the binding, if any, for the given event sequence. This applies to all matching items.

type

`type(item)`. Return the type of the given item: "arc", "bitmap", "image", "line", "oval", "polygon", "rectangle", "text", or "window". If *item* refers to more than one items, this method returns the type of the first item found.

lift

`lift(item)`, `tkraise(item)`. Move the given item to the top of the canvas stack. If multiple item matches, 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.

lower

`lower(item)`. Move the given item to the bottom of the canvas stack. If multiple item matches, 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.

move

`move(item, dx, dy)`. Move all items *dx* canvas units to the right, and *dy* canvas units downwards. Both coordinates can be negative.

postscript

`postscript(options)`. Generate a Postscript rendering of the canvas contents. Images and embedded widgets are not included.

Table 1.

Option	Type	Description
<code>colormap</code>	None	
<code>colormode</code>	None	
<code>file</code>	None	
<code>fontmap</code>	None	
<code>height</code>	None	
<code>pageanchor</code>	None	
<code>pageheight</code>	None	
<code>pagewidth</code>	None	
<code>pagex</code>	None	
<code>pagey</code>	None	
<code>rotate</code>	None	
<code>width</code>	None	
<code>x</code>	None	
<code>y</code>	None	

scale

`scale(item, xscale, yscale)`. Scale all items according to the given scale factors. Note that this method modifies the item coordinates; you may lose precision if you use this method multiple times on the same items.

Searching for Items

The following methods are used to find certain groups of items, for later processing. Note that for each *find* method, there is a corresponding *addtag* method. Instead of processing the individual items returned by a *find* method, you can often get better performance by adding a temporary tag to a group of items, process all items with that tag in one go, and then remove the tag.

find_above

`find_above(item)`. Returns the item just above the given item.

find_all

`find_all()`. Return a list containing the identity 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)`.

find_below

`find_below(item)`. Returns the item just below the given item.

find_closest

`find_closest(x, y)`. 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.

find_enclosed

`find_enclosed(x1, y1, x2, y2)`. Returns a list of all items completely enclosed by the rectangle (x1, y1, x2, y2).

find_overlapping

`find_overlapping(x1, y1, x2, y2)`. Returns a list of all items that overlap the given rectangle, or that are completely enclosed by it.

find_withtag

`find_withtag(item)`. Returns a list of all items having the given specifier.

Manipulating Tags

The following methods are used to manipulate the tags, rather than the items themselves.

addtag_above

addtag_above(newtag, item). Add *newtag* to the item just above the given item.

addtag_all

addtag_all(newtag). Add *newtag* to all items on the canvas. This is shortcut for addtag_withtag(newtag, ALL).

addtag_below

addtag_below(newtag, item). Add *newtag* to the item just below the given item.

addtag_closest

addtag_closest(newtag, x, y). Add *newtag* to the item closest to the given coordinate. See *find_closest* for more information.

addtag_enclosed

addtag_enclosed(newtag, x1, y1, x2, y2). Add *newtag* to all items enclosed by the given rectangle. See *find_enclosed* for more information.

addtag_overlapping

addtag_overlapping(newtag, x1, y1, x2, y2). Add *newtag* to all items overlapping the given rectangle. See *find_overlapping* for more information.

addtag_withtag

addtag_withtag(newtag, tag). Add *newtag* to all items having the given tag.

dtag

dtag(item, tag). Remove 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.

gettags

gettags(item). Return all tags associated with the item.

Special Methods for Text Items

The following methods can be used with text items, as well as with any extension item type that supports a keyboard focus and an insertion cursor.

dchars

dchars().

focus

`focus()`.

icursor

`icursor()`.

index

`index()`.

insert

`insert()`.

select_adjust

`select_adjust(item, index)`.

select_clear

`select_clear()`.

select_from

`select_from(item, index)`.

select_item

`select_item()`.

select_to

`select_to(item, index)`.

Scrolling

The following methods are used to scroll the canvas in various ways. The *scan* methods can be used to implement fast mouse pan/roam operations, while the *xview* and *yview* methods are used with standard scrollbars.

scan_mark, scan_dragto

`scan_mark(x, y)`, `scan_dragto(x, y)`. *scan_mark* sets the scanning anchor for fast horizontal scrolling to the given mouse coordinate. *scan_dragto* 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.

xview, yview

xview(MOVETO, offset), yview(MOVETO, offset). Adjust the canvas so that the given offset is at the left (top) edge of the canvas. Offset 0.0 is the beginning of the *scrollregion*, 1.0 the end. These methods are used by the *Scrollbar* bindings.

The *MOVETO* constant is not defined in Python 1.5.2 and earlier. For compatibility, use the string "moveto" instead.

xview, yview

xview(SCROLL, step, what), yview(SCROLL, step, what). Scroll the canvas horizontally (vertically) by the given amount. The what argument can be either UNITS (lines) or PAGES. These methods are used by the *Scrollbar* bindings.

These constants are not defined in Python 1.5.2 and earlier. For compatibility, use the strings "scroll", "units", and "pages" instead.

Options

Table 2.

Option	Type	Description
background (bg)	color	
borderwidth (bd)	distance	
closeenough		
confine		
cursor	cursor	
height	distance	
highlightbackground, highlightcolor	color	Controls how to draw the focus highlight border. When the widget has focus, the border is drawn in the <i>highlightcolor</i> color. Otherwise, it is drawn in the <i>highlightbackground</i> color. The defaults are system specific.
highlightthickness	distance	Controls the width of the focus highlight border. Default is one or two pixels. Note that the focus highlight border is drawn on top of the canvas coordinate systems; if you don't use scrollbars, a one pixel border covers items drawn at canvas coordinate (0, 0).
insertbackground	color	Color used for the insertion cursor.
insertborderwidth	distance	Borderwidth for the insertion cursor.

Option	Type	Description
insertofftime, insertontime	time	Controls cursor blinking.
insertwidth	distance	Width of the insertion cursor.
relief	constant	Border decoration. The default is <i>FLAT</i> . Other possible values are <i>SUNKEN</i> , <i>RAISED</i> , <i>GROOVE</i> and <i>RIDGE</i> . Note that to show the border, you need to change the <i>borderwidth</i> from it's default value of 0. Also note that the border is drawn on top of the canvas coordinate system.
scrollregion	4-tuple	The bounding box of the scrollable area. If this option is not set, the scrolling is not bounded.
selectbackground	color	
selectborderwidth	distance	
selectforeground	color	
takefocus	flag	Indicates that the user can use the <i>Tab</i> key to move to this widget. Default is an empty string, which means that the canvas accepts focus only if it has any keyboard bindings (default is off, in other words).
width	distance	
xscrollcommand	callback	
xscrollincrement	distance	
yscrollcommand	callback	
yscrollincrement	distance	

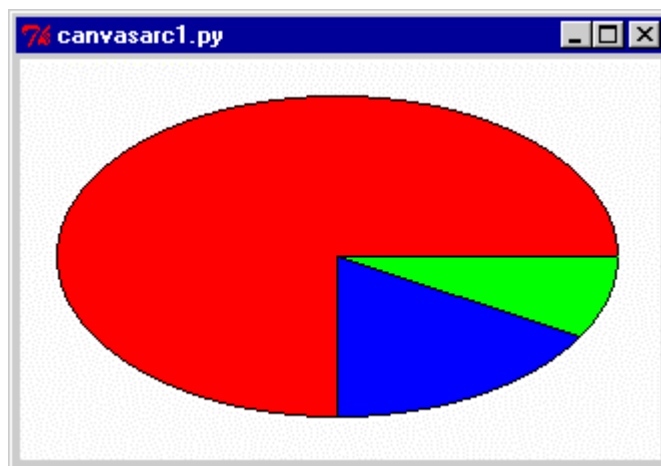
The Canvas Arc Item

An arc item is a section of oval, delimited by two angles (*start* and *extent*). An arc item can be drawn in one of three ways:

- *pieslice* (lines are drawn from the perimeter to the oval's center)
- *chord* (the ends are connected with a straight line)
- *arc* (only the perimeter section is drawn)

Pieslices and chords can be filled.

Figure 1. Pieslice Example (see methods section for corresponding code)



Methods

The following methods are used to create and configure *arc* items:

create_arc

`create_arc(x0, y0, x1, y1, options...)`, `create_arc(box, options...)`. Create a arc item enclosed by the given rectangle. The *start* and *extent* options control which section to draw. If they are set to 0.0 and 360.0, a full oval is drawn which touches the rectangle's four edges.

```
xy = 20, 20, 300, 180
canvas.create_arc(xy, start=0, extent=270, fill="red")
canvas.create_arc(xy, start=270, extent=60, fill="blue")
canvas.create_arc(xy, start=330, extent=30, fill="green")
```

delete

`delete(item)`. Delete an arc item.

coords

`coords(item, x0, y0, x1, y1)`. Change the enclosing rectangle for one or more arc items.

itemconfigure

`itemconfigure(item, options...)`. Change the options for one or more arc items.

Options

The *arc* item supports the following options, via the *create_arc* method, and the *itemconfig* and *itemcget* configuration methods.

Table 1. Canvas Arc Options

Option	Type	Description
style	constant	Specifies how to draw the arc item (see above). Use one of <i>PIESLICE</i> , <i>CHORD</i> , or <i>ARC</i> . The default is <i>PIESLICE</i> . These constants are not defined in Python 1.5.2 and earlier. For compatibility, use the strings "pieslice", "chord", and "arc" instead.
start, extent	angle	The arc is drawn from the start angle (measured counter-clockwise from three o'clock) to the start angle plus the extent. Both angles are given in degrees, and can be negative. By default, the arc starts at 0.0 degrees (three o'clock), and extends 90.0 degrees counter-clockwise (twelve o'clock).
fill	color	The color to use for the arc's interior. If an empty string is given, the interior is not drawn. Note that arc's having the <i>arc</i> style cannot be filled. Default is empty (transparent).
stipple	bitmap	The name of a bitmap which is used as a stipple brush when filling the arc's interior. Typical values are "gray12", "gray25", "gray50", or "gray75". Default is a solid brush (no bitmap). As of Tk 8.0p2, the stipple option is ignored on the Windows platform. To draw stippled pieslices or chords, you have to create corresponding polygons.
outline	color	The color to use for the arc's outline. If an empty string is given, the outline is not drawn. Default is "black".
outlinestipple	bitmap	The name of a bitmap which is used as a stipple brush when drawing the arc's outline. Typical values are "gray12", "gray25", "gray50", or "gray75". Default is a solid brush (no bitmap).
width	distance	The width of the arc's outline. Default is 1 pixel.

Option	Type	Description
tags	tuple	One or more tags to associate with this item. If only a single tag is to be used, you can use a single string instead of a tuple of strings.


itemconfigure

itemconfigure(item, options...). Change the options for one or more bitmap items.

Options

The *bitmap* item supports the following options, via the *create_bitmap* method, and the *itemconfig* and *itemcget* configuration methods.

Table 1.

Option	Type	Description
bitmap	bitmap	<p>The name of the bitmap.</p> <p>The following bitmaps are available on all platforms: "error", "gray75", "gray50", "gray25", "gray12", "hourglass", "info", "questhead", "question", and "warning".</p>  <p>The following additional bitmaps are available on the Macintosh only: "document", "stationery", "edition", "application", "accessory", "folder", "pfolder", "trash", "floppy", "ramdisk", "cdrom", "preferences", "querydoc", "stop", "note", and "caution".</p> <p>You can also load the bitmap from an XBM file. Just prefix the filename with an at-sign, for example "@sample.xbm".</p>
anchor	constant	Specifies which part of the bitmap that should be placed at the given position. Use one of <i>N</i> , <i>NE</i> , <i>E</i> , <i>SE</i> , <i>S</i> , <i>SW</i> , <i>W</i> , <i>NW</i> , or <i>CENTER</i> . Default is <i>CENTER</i> .
foreground	color	The color to use for the bitmap's foreground pixels (that is, non-zero pixels). If an empty string is given, the foreground pixels are not drawn. Default is "black".
background	color	The color to use for the bitmap's background pixels (that is, zero pixels). If an empty string is given, the background pixels are not drawn. Default is empty (transparent).
tags	tuple	One or more tags to associate with this item. If only a single tag is to be used, you can use a single string instead of a tuple of strings.

The Canvas Image Item

Methods

The following methods are used to create and configure *image* items:

create_image

`create_image(x0, y0, options...)`. Create a image item placed relative to the given position. Note that the image itself is given by the *image* option.

Example:

```
photo = PhotoImage(file="sample.gif")
item = canvas.create_image(10, 10, anchor=NW, image=photo)
```

[FIXME: add note on image ownership]

delete

`delete(item)`. Delete an image item.

coords

`coords(item, x0, y0)`. Move one or more image items.

itemconfigure

`itemconfigure(item, options...)`. Change the options for one or more image (or other) items.

Options

The *image* item supports the following options, via the *create_image* method, and the *itemconfig* and *itemcget* configuration methods.

Table 1.

Option	Type	Description
image	image	The image object (a Tkinter <i>PhotoImage</i> or <i>BitmapImage</i> instance, or instances of the corresponding Python Imaging Library classes).
anchor	constant	Specifies which part of the image that should be placed at the given position. Use one of <i>N</i> , <i>NE</i> , <i>E</i> , <i>SE</i> , <i>S</i> , <i>SW</i> , <i>W</i> , <i>NW</i> , or <i>CENTER</i> . Default is <i>CENTER</i> .
tags	tuple	One or more tags to associate with this item. If only a single tag is to be used, you can use a single string instead of a tuple of strings.

The Canvas Line Item

Methods

create_line

`create_line(x0, y0, x1, y1, ..., xn, yn, options...)`. Create a line item.

delete

`delete(item)`. Delete a line item.

coords

`coords(item, x0, y0, x1, y1, ..., xn, yn)`. Change the coordinates for one or more line items.

itemconfigure

`itemconfigure(item, options...)`. Change the options for one or more line items.

Options

The *line* item supports the following options, via the *create_line* method, and the *itemconfig* and *itemcget* configuration methods.

Table 1.

Option	Type	Description
width	distance	The width of the line. Default is 1 pixel.
fill	color	The color to use for the line. Default is "black".
stipple	bitmap	The name of a bitmap which is used as a stipple brush when drawing the line. Typical values are "gray12", "gray25", "gray50", or "gray75". Default is a solid brush (no bitmap).
arrow	constant	If set to a value other than <i>NONE</i> , the line is drawn as an arrow. The option value defines where to draw the arrow head: <i>FIRST</i> , <i>LAST</i> , or <i>BOTH</i> . Default is <i>NONE</i> . The <i>FIRST</i> and <i>LAST</i> constants are not defined in Python 1.5.2 and earlier. For compatibility, use the strings "first" and "last" instead.
arrowshape	3-tuple	Controls the shape of the arrow. Default is (8, 10, 3).
capstyle	constant	For wide lines, this option controls how to draw the line ends. Use one of <i>BUTT</i> , <i>PROJECTING</i> , <i>ROUND</i> . Default is <i>BUTT</i> .

Option	Type	Description
		These constants are not defined in Python 1.5.2 and earlier. For compatibility, use the strings "butt", "projecting", and "round" instead.
joinstyle	const	For wide lines, this option controls how to draw the joins between edges. Use one of <i>BEVEL</i> , <i>MITER</i> , or <i>ROUND</i> . Default is <i>ROUND</i> . These constants are not defined in Python 1.5.2 and earlier. For compatibility, use the strings "bevel", "miter", and "round" instead.
smooth	flag	If non-zero, the given coordinates are interpreted as b-spline vertices.
splinsteps	int	The number of steps to use when smoothing this line. Default is 12.
tags	tags	One or more tags to associate with this item. If only a single tag is to be used, you can use a single string instead of a tuple of strings.

The Canvas Oval Item

Methods

create_oval

`create_oval(x0, y0, options...)`. Create a oval item at the given position, using the given options. Note that the oval string itself is given by the *oval* option.

delete

`delete(item)`. Delete an oval item.

coords

`coords(item, x0, y0)`. Move one or more oval items.

itemconfigure

`itemconfigure(item, options...)`. Change the options for one or more oval (or other) items.

Options

The *oval* item supports the following options, via the *create_oval* method, and the *itemconfig* and *itemcget* configuration methods.

Table 1.

Option	Type	Description
fill	colour	The colour to use for the interior. If an empty string is given, the interior is not drawn. Default is empty (transparent).
stipple	bitmap	The name of a bitmap which is used as a stipple brush when filling the oval's interior. Typical values are "gray12", "gray25", "gray50", or "gray75". Default is a solid brush (no bitmap). As of Tk 8.0p2, the stipple option is ignored on the Windows platform. To draw stippled ovals, you have to create corresponding polygons.
outline	colour	The colour to use for the outline. If an empty string is given, the outline is not drawn. Default is "black".
width	distance	The width of the outline. Default is 1 pixel.
tags	tuple	One or more tags to associate with this item. If only a single tag is to be used, you can use a single string instead of a tuple of strings.

The Canvas Polygon Item

Methods

The following methods are used to create and configure *polygon* items:

create_polygon

`create_polygon(xy, options...)`.

`create_polygon(x0, y0, x1, y1, x2, y2, ..., xn, yn, options...)`. Create a polygon item. You must specify at least 3 vertices when you create a new polygon.

delete

`delete(item)`. Delete a polygon item.

coords

`coords(item, x0, y0, x1, y1, x2, y2, ..., xn, yn)`. Change the coordinates for one or more polygon items. Note that the coordinates must be given as separate arguments; you cannot use a sequence as with *create_polygon*.

itemconfigure

`itemconfigure(item, options...)`. Change the options for one or more polygon items.

Options

The *polygon* item supports the following options, via the *create_polygon* method, and the *itemconfig* and *itemcget* configuration methods.

Table 1.

Option	Type	Description
fill	None	The colour to use for the polygon interior. If an empty string is given, the interior is not drawn. Default is empty (transparent).
stipple	bitmap	The name of a bitmap which is used as a stipple brush when filling the polygon's interior. Typical values are "gray12", "gray25", "gray50", or "gray75". Default is a solid brush (no bitmap).
outline	None	The colour to use for the polygon outline. If an empty string is given, the outline is not drawn. Default is "black".
width	distance	The width of the polygon's outline. Default is 1 pixel.
smooth	None	If non-zero, the given coordinates are interpreted as

Option	Type	Description
		b-spline vertices.
splinsteps	None	The number of steps to use when smoothing the polygon outline. Default is 12.
tags	tuple	One or more tags to associate with the polygon. If only a single tag is to be used, you can use a single string instead of a tuple of strings.

The Canvas Rectangle Item

Methods

The following methods are used to create and configure *rectangle* items:

create_rectangle

`create_rectangle(x0, y0, x1, y1, options...)`. Create a rectangle item between the given coordinates. The rectangle item is created with the given options.

delete

`delete(item)`. Delete a rectangle item.

coords

`coords(item, x0, y0, x1, y1)`. Change the coordinates for one or more rectangle items. The *item* argument can match one or more rectangle items, rectangles, or any other item taking exactly four coordinates.

itemconfigure

`itemconfigure(item, options...)`. Change the options for one or more rectangle items.

Options

The *rectangle* item supports the following options, via the *create_rectangle* method, and the *itemconfig* and *itemcget* configuration methods.

Table 1.

Option	Type	Description
fill	None	The colour to use for the rectangle interior. If an empty string is given, the interior is not drawn. Default is empty (transparent).
outline	None	The colour to use for the outline. If an empty string is given, the outline is not drawn. Default is "black".
stipple	None	The name of a bitmap which is used as a stipple brush when filling the rectangle's interior. Typical values are "gray12", "gray25", "gray50", or "gray75". Default is a solid brush (no bitmap).
tags	None	One or more tags to associate with the rectangle. If only a single tag is to be used, you can use a single string instead of a tuple of strings.
width	distance	The width of the rectangle's outline. Default is 1 pixel.

The Canvas Text Item

Methods

The following methods are used to create and configure *text* items:

create_text

`create_text(x0, y0, options...)`. Create a text item at the given position, using the given options. Note that the text string itself is given by the *text* option.

delete

`delete(item)`. Delete a text item.

coords

`coords(item, x0, y0)`. Move one or more text items.

itemconfigure

`itemconfigure(item, options...)`. Change the options for one or more text (or other) items.

Options

The *text* item supports the following options, via the *create_text* method, and the *itemconfig* and *itemcget* configuration methods.

Table 1.

Option	Type	Description
anchor	constant	Specifies which part of the text (the text's bounding box, more exactly) that should be placed at the given position. Use one of <i>N</i> , <i>NE</i> , <i>E</i> , <i>SE</i> , <i>S</i> , <i>SW</i> , <i>W</i> , <i>NW</i> , or <i>CENTER</i> . Default is <i>CENTER</i> .
fill	colour	The colour to use for the text. If an empty string is given, the text is not drawn. Default is empty (transparent).
font	font	
justify	constant	
stipple	bitmap	
tags	tuple	One or more tags to associate with the text. If only a single tag is to be used, you can use a single string instead of a tuple of strings.
text	string	The text string.

Option	Type	Description
width	distance	

The Canvas Window Item

Methods

The following methods are used to create and configure *window* items:

create_window

`create_window(x0, y0, options...)`. Embed a window at the given position, using the given options. Note that the widget to use is given by the *window* option.

delete

`delete(item)`. Delete a window item.

coords

`coords(item, x0, y0)`. Move one or more window items.

itemconfigure

`itemconfigure(item, options...)`. Change the options for one or more window (or other) items.

Options

The *window* item supports the following options, via the *create_window* method, and the *itemconfig* and *itemcget* configuration methods.

Table 1.

Option	Type	Description
window	window	The widget to embed in the canvas.
anchor	constant	Specifies which part of the window that should be placed at the given position. Use one of <i>N</i> , <i>NE</i> , <i>E</i> , <i>SE</i> , <i>S</i> , <i>SW</i> , <i>W</i> , <i>NW</i> , or <i>CENTER</i> . Default is <i>CENTER</i> .
height, width	distance	The height and width of the window. If omitted, the height and width defaults to the actual window size.
tags	tuple	One or more tags to associate with the window. If only a single tag is to be used, you can use a single string instead of a tuple of strings.

The Checkbutton Widget

The *Checkbutton* widget is a standard Tkinter widget 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 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 *Checkbutton* widget should be associated with a variable.

Checkbutton Patterns

(Also see the *Button* patterns).

To use a *Checkbutton*, you must create a Tkinter variable:

```
var = IntVar()
c = Checkbutton(master, text="Expand", variable=var)
```

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()
```

Methods

The *Checkbutton* widgets support the standard Tkinter Widget interface, plus the following methods:

deselect

`deselect()`. Deselect the button.

flash

`flash()`. Redraw the button several times, alternating between active and normal appearance.

invoke

`invoke()`. Call the command associated with the button.

select

`select()`. Select the button.

toggle


`toggle()`. Toggle the selection state.

Options

The *Checkbutton* widgets support the following options:

Table 1.

Option	Type	Description
activebackground, activeforeground	color	The color to use when the button is activated.
anchor	constant	Controls where in the button the text (or image) should be located. Use one of <i>N</i> , <i>NE</i> , <i>E</i> , <i>SE</i> , <i>S</i> , <i>SW</i> , <i>W</i> , <i>NW</i> , or <i>CENTER</i> . Default is <i>CENTER</i> . If you change this, it is probably a good idea to add some padding as well, using the <i>padx</i> and/or <i>pady</i> options.
background, foreground	color	The button color. The default is platform specific.
bitmap	bitmap	The bitmap to display in the widget. If the <i>image</i> option is given, this option is ignored. The following bitmaps are available on all platforms: "error", "gray75", "gray50", "gray25", "gray12", "hourglass", "info", "questhead",

Option	Type	Description
		<p>"question", and "warning".</p>  <p>The following additional bitmaps are available on the Macintosh only: "document", "stationery", "edition", "application", "accessory", "folder", "pfolder", "trash", "floppy", "ramdisk", "cdrom", "preferences", "querydoc", "stop", "note", and "caution".</p> <p>You can also load the bitmap from an XBM file. Just prefix the filename with an at-sign, for example "@sample.xbm".</p>
borderwidth (bd)	int	The width of the button border. The default is platform specific.
command	callback	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.
cursor	cursor	The cursor to show when the mouse is moved over the button.
default	int	If set, the button is a default button. Tk will indicate this by drawing a platform specific indicator (usually an extra border). NOTE: The syntax has changed in 8.0b2!!!
disabledforeground	color	The color to use when the button is disabled. The background is shown in the <i>background</i> color.
font	font	The font to use in the button. The button can only contain text in a single font.
highlightbackground, highlightcolor	color	Controls how to draw the focus highlight border. When the widget has focus, the border is drawn in the <i>highlightcolor</i> color. Otherwise, it is drawn in the <i>highlightbackground</i> color. The defaults are system specific.
highlightthickness	distance	Controls the width of the focus highlight border. Default is typically one or two pixels.
image	image	The image to display in the widget. If specified, this takes precedence over the <i>text</i> and <i>bitmap</i> options.
indicatoron	bool	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 <i>SUNKEN</i> instead of <i>RAISED</i> .
justify	constant	Defines how to align multiple lines of text. Use

Option	Type	Description
		<i>LEFT</i> , <i>RIGHT</i> , or <i>CENTER</i> .
offvalue, onvalue	value	The values corresponding to a non-checked or checked button, respectively. Defaults are 0 and 1.
padx, pady	distance	Button padding. These options specify the horizontal and vertical padding between the text or image, and the button border.
relief	constant	Border decoration. This is usually <i>FLAT</i> for checkbuttons, unless they use the border as indicator (via the <i>indicatoron</i> option).
selectcolor	color	Color to use for the selector.
selectimage	image	Graphic image to use for the selector.
state	constant	The button state: <i>NORMAL</i> , <i>ACTIVE</i> or <i>DISABLED</i> . Default is <i>NORMAL</i> .
takefocus	flag	Indicates that the user can use the <i>Tab</i> 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).
text	string	The text to display in the button. The text can contain newlines. If the <i>bitmap</i> or <i>image</i> options are used, this option is ignored.
textvariable	variable	Associates a Tkinter variable (usually a <i>StringVar</i>) to the button. If the variable is changed, the button text is updated. Also see the <i>variable</i> option.
underline	int	Default is -1.
variable	variable	Associates a Tkinter variable to the button. When the button is pressed, the variable is toggled between <i>offvalue</i> and <i>onvalue</i> . Explicit changes to the variable are automatically reflected by the buttons.
width, height	distance	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.
wrplength	distance	Determines when a button's text should be wrapped into multiple lines. This is given in screen units. Default is no wrapping.

The Entry Widget

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

Concepts

Indexes

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

- Numerical indexes
- *ANCHOR*
- *END*
- *INSERT*
- Mouse coordinates

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 "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.

Patterns

Methods

The *Entry* widget support the standard Tkinter Widget interface, plus the following methods:

insert

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

delete

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

icursor

`icursor(index)`. Move the insertion cursor to the given index. This also sets the *INSERT* index.

get

`get()`. Get the current contents of the entry field.

index

`index(index)`. Return the numerical position corresponding to the given index.

Selection Methods

selection_adjust

`selection_adjust(index)`, `select_adjust(index)`. Adjust the selection to include also the given character. If index is already selected, do nothing.

selection_clear

`selection_clear()`, `select_clear()`. Clear the selection.

selection_from

`selection_from(index)`, `select_from(index)`. Starts a new selection. This also sets the *ANCHOR* index.

selection_present

`selection_present()`, `select_present()`. Returns true (non-zero) if some part of the text is selected.

selection_range

`selection_range(start, end)`, `select_range(start, end)`. Explicitly set the selection. Start must be smaller than end. Use `selection_range(0, END)` to select all text in the widget.

selection_to

`selection_to(index)`, `select_to(index)`. Select all text between *ANCHOR* and the given index.

Scrolling Methods

These methods are used to scroll the entry widget in various ways. The *scan* methods can be used to implement fast mouse panning operations (they are bound to the middle mouse button, if available), while the *xview* method is used with a standard scrollbar widget.

scan_mark, scan_dragto

`scan_mark(x)`, `scan_dragto(x)`. *scan_mark* sets the scanning anchor for fast horizontal scrolling to the given mouse coordinate. *scan_dragto* 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.

xview

`xview(index)`. Make sure the given index is visible. The widget is scrolled if necessary.

xview_moveto

`xview_moveto(fraction)`.

xview_scroll

`xview_scroll(number, what)`.

Options

The *Entry* widget support the following options:

Table 1.

Option	Type	Description
background (bg)	color	Widget background.
borderwidth (bd)	distance	Border width.
cursor	cursor	Widget cursor. The default is a text insertion cursor (typically an "I beam" cursor, e.g. <i>xterm</i>).
exportselection	flag	If true, selected text is automatically exported to the clipboard. Default is true.
font	font	Widget font. The default is system specific.
foreground (fg)	color	Text color.
highlightbackground, highlightcolor	color	Controls how to draw the focus highlight border. When the widget has focus, the border is drawn in the <i>highlightcolor</i> color. Otherwise, it is drawn in the <i>highlightbackground</i> color. The defaults are system specific.
highlightthickness	distance	Controls the width of the focus highlight border. Default is typically one or two pixels.
insertbackground	color	Color used for the insertion cursor.

Option	Type	Description
insertborderwidth	color	
insertofftime, insertontime	int	Controls cursor blinking.
insertwidth	int	Width of the insertion cursor.
justify	const	
relief	const	Border decoration. The default is <i>FLAT</i> . Other possible values are <i>SUNKEN</i> , <i>RAISED</i> , <i>GROOVE</i> and <i>RIDGE</i> .
selectbackground	color	Selection background color. The default is system and display specific.
selectborderwidth	int	Selection border width. The default is system specific.
selectforeground	color	Selection text color. The default is system and display specific.
show	character	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, use "*".
state	const	One of <i>NORMAL</i> or <i>DISABLED</i> . Default is <i>NORMAL</i> . Note that if you set this to <i>DISABLED</i> , calls to <i>insert</i> or <i>delete</i> are ignored.
takefocus	flag	Indicates that the user can use the <i>Tab</i> key to move to this widget. Default is an empty string, which means that the canvas accepts focus only if it has any keyboard bindings (default is on, in other words).
textvariable	variable	
width	int	
xscrollcommand	callback	

The Font Class

Patterns

Methods

copy

`copy()`. Return a distinct copy of the current font.

actual

`actual()`, `actual(option)`. Return actual font attributes. If no option is given, returns all actual font attributes as a dictionary.

cget

`cget(option)`. Get configured font attribute.

config, configure

`config()`, `configure()`. Get full set of configured font attributes as a dictionary.

`config(options)`, `configure(options...)`. Modify one or more font attributes.

measure

`measure(text)`. Return text width.

metrics

`metrics()`, `metrics(options...)`. Return one or more font metrics. If no arguments are given, all metrics are returned as a dictionary.

For best performance, make sure that this font is in use before calling this method. If necessary, you can create a dummy widget using the font.

Functions

families

`families()`. Get a list of available font families.

names

`names()`. Get a list of the names of names of all user-defined fonts.

Options

The constructor and the *config* method supports the following options.

Table 1.

Option	Type	Description
font	font	Font specifier (name, system font, or (family, size, style)-tuple). If you use this option,
family	string	Font family.
size	integer	Font size in points. To give the size in pixels, use a negative value.
weight	constant	Font thickness. Use one of <i>NORMAL</i> or <i>BOLD</i> . Default is <i>NORMAL</i> . Note that these constants are defined in the <i>tkFont</i> module.
slant	constant	Font slant. Use one of <i>NORMAL</i> or <i>ITALIC</i> . Default is <i>NORMAL</i> . Note that these constants are defined in the <i>tkFont</i> module.
underline	flag	Font underlining. If 1 (true), the font is underlined. Default is 0 (false).
overstrike	flag	Font strikeout. If 1 (true), a line is drawn over text written with this font. Default is 0 (false).

The 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.

Patterns

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())
```

Methods

Except for the standard widget interface (*config*, etc), the *Frame* widget has no methods.

Options

The *Frame* widget supports the following options:

Table 1.

Option	Type	Description
height, width	distance	Frame size.
background (bg)	color	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.
colormap	window	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.
cursor	cursor	The cursor to show when the mouse pointer is placed over the button widget. Default is a system

Option	Type	Description
		specific arrow cursor.
relief	constant	Border decoration. The default is <i>FLAT</i> . Other possible values are <i>SUNKEN</i> , <i>RAISED</i> , <i>GROOVE</i> and <i>RIDGE</i> . Note that to show the border, you need to change the <i>borderwidth</i> from it's default value of 0.
borderwidth (bd)	distance	Border width. Defaults to 0 (no border).
takefocus	flag	Indicates that the user can use the <i>Tab</i> key to move to this widget. Default is an empty string, which means that the frame accepts focus only if it has any keyboard bindings (default is off, in other words).
highlightbackground, highlightcolor	color	Controls how to draw the focus highlight border. When any child to the frame has focus, the border is drawn in the <i>highlightcolor</i> color. Otherwise, it is drawn in the <i>highlightbackground</i> color. The defaults are system specific.
highlightthickness	distance	Controls the width of the focus highlight border. Default is 0 (no border).

The 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 (I tend to use two; one for the dialog body, and one for the button box at the bottom), and use the grid manager to get them all where you want them.

Consider the following example:

Figure 1.

<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:

Figure 2. Figure: 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)
```

Figure 3. Figure: 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)

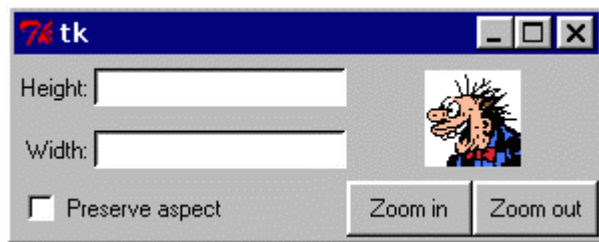
checkboxbutton.grid(columnspan=2, sticky=W)

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 checkbutton 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:

Figure 4. Figure: using column and row spans



Widget Methods

The following methods are available on widgets managed by the grid manager:

grid

`grid(option=value, ...)`, `grid_configure(option=value, ...)`. Place the widget in a grid as described by the options (see below).

grid_forget

`grid_forget()`. Remove the widget. The widget is not destroyed, and can be displayed again by *grid* or any other manager.

grid_info

`grid_info()`. Return a dictionary containing the current options.

grid_remove

`grid_remove()`. Remove the widget. The widget is not destroyed, and can be displayed again by *grid* or any other manager.

Manager Methods

The following methods are available on widgets that are used as grid managers (that is, the geometry *masters* for widgets managed by the grid manager).

columnconfigure, rowconfigure

`columnconfigure(column, option=value, ...)`, `rowconfigure(row, option=value, ...)`. Set options for the given column (or row).

To change this for a given widget, you have to call this method on the widget's parent.

Table 1.

Option	Type	Description
minsize	integer	Defines the minimum size for the column (row). Note that if a column or row is completely empty, it will not be displayed, even if this option is set.
pad	integer	Padding to add to the size of the largest widget in the column (row) when setting the size of the whole column.
weight	integer	A relative weight used to distribute additional space between columns (rows). 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_location

grid_location(x, y). Returns the grid cell under (or closest to) the given pixel coordinate. The result is a 2-tuple: (column, row).

grid_propagate

grid_propagate().

grid_size

grid_size(). Returns the current grid size. This is defined as indexes of the first empty column and row in the grid, in that order. The result is a 2-tuple: (column, row).

grid_slaves

grid_slaves(). Returns a list of the "slave" widgets managed by this widget. The widgets are returned as Tkinter widget references.

Options

The following options can be used with the *grid* and *grid_configure* methods:

Table 2.

Option	Type	Description
column	integer	Insert the widget at this column. Column numbers start with 0. If omitted, defaults to 0.
columnspan	integer	If given, indicates that the widget cell should span more than one column.
in (in_)	widget	Place widget inside to the given widget. You can only place a widget inside its parent, or in any

Option	Type	Description
		decendant of its parent. If this option is not given, it defaults to the parent. Note that <i>in</i> is a reserved word in Python. To use it as a keyword option, append an underscore (<i>in_</i>).
ipadx, ipady	distance	Optional internal padding. Works like <i>padx</i> and <i>pady</i> , but the padding is added <i>inside</i> the widget borders. Default is 0.
padx, pady	distance	Optional padding to place around the widget in a cell. Default is 0.
row	integer	Insert the widget at this row. Row numbers start with 0. If omitted, defaults to the first empty row in the grid.
rowspan	integer	If given, indicates that the widget cell should span more than one row.
sticky	constant	Defines how to expand the widget if the resulting cell is larger than the widget itself. This can be any combination of the constants <i>S</i> , <i>N</i> , <i>E</i> , and <i>W</i> , or <i>NW</i> , <i>NE</i> , <i>SW</i> , and <i>SE</i> . For example, <i>W</i> (west) means that the widget should be aligned to the left cell border. <i>W+E</i> means that the widget should be stretched horizontally to fill the whole cell. <i>W+E+N+S</i> means that the widget should be expanded in both directions. Default is to center the widget in the cell.

The Label Widget

The *Label* widget is a standard Tkinter widget used to display a text or image on the screen. 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.

Patterns

To use a label, you just have to specify what to display in it (this can be text, a bitmap, or an image):

```
w = Label(master, text="Hello, world!")
```

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 *wrplength* 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 variable with the label. When the contents of the variable changes, the label is automatically updated:

```
v = StringVar()
Label(master, textvariable=v).pack()
v.set("New Text!")
```


Methods

The *Label* widget supports the standard Tkinter Widget interface. There are no additional methods.

Options

The following options can be used for the *Label* widget.

Table 1.

Option	Type	Description
text	string	The text to display in the label. The text can contain newlines. If the <i>bitmap</i> or <i>image</i> options are used, this option is ignored.
bitmap	bitmap	<p>The bitmap to display in the widget. If the <i>image</i> option is given, this option is ignored.</p> <p>The following bitmaps are available on all platforms: "error", "gray75", "gray50", "gray25", "gray12", "hourglass", "info", "questhead", "question", and "warning".</p>  <p>The following additional bitmaps are available on the Macintosh only: "document", "stationery", "edition", "application", "accessory", "folder", "pfolder", "trash", "floppy", "ramdisk", "cdrom", "preferences", "querydoc", "stop", "note", and "caution".</p> <p>You can also load the bitmap from an XBM file. Just prefix the filename with an at-sign, for example "@sample.xbm".</p>
image	image	The image to display in the widget. If specified, this takes precedence over the <i>text</i> and <i>bitmap</i> options.
width, height	int	The size 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 omitted, it is calculated based on the label contents.
relief	constant	<p>Border decoration. The default is <i>FLAT</i>. Other possible values are <i>SUNKEN</i>, <i>RAISED</i>, <i>GROOVE</i> and <i>RIDGE</i>.</p> <p>Note that to show the border, you need to change the <i>borderwidth</i> from its default value of 0.</p>
borderwidth	dimension	The width of the label border. The default is 0 (no border).
background (bg), foreground (fg)	color	The label color (the foreground value is used for text and bitmap labels only). The default is platform specific.
font	font	The font to use in the label. The label can only contain text in a single font.
justify	constant	Defines how to align multiple lines of text. Use <i>LEFT</i> , <i>RIGHT</i> , or <i>CENTER</i> .

Option	Type	Description
anchor	constant	Controls where in the label the text (or image) should be located. Use one of <i>N</i> , <i>NE</i> , <i>E</i> , <i>SE</i> , <i>S</i> , <i>SW</i> , <i>W</i> , <i>NW</i> , or <i>CENTER</i> . Default is <i>CENTER</i> .
wrplength	distance	Determines when a label's text should be wrapped into multiple lines. This is given in screen units. Default is no wrapping.
textvariable	variable	Associates a Tkinter variable (usually a <i>StringVar</i>) to the label. If the variable is changed, the label text is updated.
underline	int	Default is -1.
cursor	cursor	The cursor to show when the mouse is moved over the label.

The 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.

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.

```
listbox = Listbox(master)

listbox.insert(END, "a list entry")

for item in ["one", "two", "three", "four"]:
    listbox.insert(END, item)
```

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 "checklist" behaviour, 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 1.101 (Python 1.5.1) and earlier versions causes this list to be returned as a list of strings, instead of integers. This will most likely 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 = list.curselection()
try:
```

```

    items = map(string.atoi, items)
except ValueError: pass

```

In Python 1.5 and later, you can use the builtin *int* function instead of *string.atoi*, but that's actually slightly slower.

Use the *get* method to get the list item corresponding to a given index.

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 indexed by the selection list:

```

items = self.lb.curselection()
try:
    items = map(string.atoi, 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)

```

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:

```

def __init__(self, master):
    self.list = Listbox(selectmode=EXTENDED)
    self.list.pack()
    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)

```

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)
```

Methods

The *Listbox* widget supports the standard Tkinter Widget interface, plus the following methods:

activate

`activate(index)`. Activate the given index (it will be marked with an underline). The active item can be referred to using the *ACTIVE* index.

bbox

`bbox(index)`. Get the bounding box of the given item text. The bounding box is returned as a 4-tuple giving (*xoffset*, *yoffset*, *width*, *height*). If the item is not visible, this method returns *None*.

curselection

`curselection()`. Get 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 Tkinter 1.101 (Python 1.5.1), 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.

delete

`delete(index)`, `delete(first, last)`. Delete one or more items. Use `delete(0, END)` to delete all items in the list.

get

`get(index)`, `get(first, last)`. Get 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 *ACTIVE* to get the active (underlined) item.

index

`index(index)`. Return 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.

insert

`insert(index, items)`. Insert 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.

nearest

`nearest(y)`. Return the index nearest to the given coordinate (a widget-relative pixel coordinate).

see

`see(index)`. Make sure the given list index is visible. You can use an integer index, or *END*.

size

`size()`. Return the number of items in the list. The valid index range goes from 0 to `size()-1`.

Selection Methods

The following methods are used to manipulate the listbox selection.

select_adjust

`select_adjust(index)`. Extend the selection to include the given index.

select_anchor

`select_anchor(index)`. Set the selection anchor to the given index. The anchor can be referred to using the *ANCHOR* index.

select_clear

`select_clear()`. Clear the selection.

select_includes

`select_includes(index)`. Returns true if the given item is selected.

select_set

`select_set(index)`, `select_set(first, last)`. Add one or more items to the selection.

Scrolling Methods

These methods are used to scroll the listbox widget in various ways. The *scan* methods can be used to implement fast mouse scrolling operations (they are bound to the middle mouse button, if available), while the *yview* method is used with a standard scrollbar widget.

scan_mark, scan_dragto

`scan_mark(x, y)`, `scan_dragto(x, y)`. *scan_mark* sets the scanning anchor for fast horizontal scrolling to the given mouse coordinate. *scan_dragto* 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.

xview, yview

`xview()`, `yview()`. Determine which part of the full list that is visible in the horizontal (vertical) 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). These methods are used by the *Scrollbar* bindings.

xview, yview

`xview(column)`, `yview(index)`. Adjust the list so that the given character column (list item) is at the left (top) edge of the listbox. To make sure that a given item is visible, use the *see* method instead.

xview, yview

`xview(MOVETO, offset)`, `yview(MOVETO, offset)`. Adjust 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.

The *MOVETO* constant is not defined in Python 1.5.2 and earlier. For compatibility, use the string "moveto" instead.

xview, yview

`xview(SCROLL, step, what)`, `yview(SCROLL, step, what)`. Scroll the list horizontally (vertically) by the given amount. The `what` argument can be either *UNITS* (lines) or *PAGES*. These methods are used by the *Scrollbar* bindings when the user clicks on a scrollbar arrow or in the trough.

These constants are not defined in Python 1.5.2 and earlier. For compatibility, use the strings "scroll", "units", and "pages" instead.

Options

The *Listbox* widget supports the following options:

Table 1.

Option	Type	Description
background, foreground	color	The listbox color. The default is platform specific.
cursor	cursor	The cursor to show when the mouse is placed over the listbox.
exportselection	bool	If set, the list selection is automatically exported via the X selection mechanism. The default is on. If you have more than one list in the same dialog, it is probably best to disable this mechanism.
font	font	The font to use in the listbox. The listbox can only contain text in a single font.
relief	constant	Border decoration. The default is <i>SUNKEN</i> . Other possible values are <i>FLAT</i> , <i>RAISED</i> , <i>GROOVE</i> , and <i>RIDGE</i> .
borderwidth (bd)	distance	The width of the listbox border. The default is platform specific, but is usually 1 or 2 pixels.
selectbackground, selectforeground	color	Selection color settings.
selectborderwidth	dimension	Selection border width. The selection is always raised.
selectmode	constant	Selection mode. One of <i>SINGLE</i> , <i>BROWSE</i> , <i>MULTIPLE</i> , or <i>EXTENDED</i> . Default is <i>BROWSE</i> . Use <i>MULTIPLE</i> to get checklist behaviour, <i>EXTENDED</i> if the user usually selects one item, but sometimes would like to select one or more ranges of items. See the patterns section for more

Option	Type	Description
		information.
setgrid	bool	
takefocus	bool	Indicates that the user can use the <i>Tab</i> key to move to this widget. Default is an empty string, which means that the listbox accepts focus only if it has any keyboard bindings (default is on, in other words).
width, height	distance	The size of the listbox, in text units.
xscrollcommand, yscrollcommand	command	Used to connect a listbox to a scrollbar. These options should be set to the <i>set</i> methods of the corresponding scrollbars.

The Menu Widget

Patterns

Methods

add

add(itemType, options...).

Table 1.

Option	Type	Description
activebackground	color	
activeforeground	color	
accelerator	string	
background	color	
bitmap	bitmap	
columnbreak	flag	
command	callback	
font	font	
foreground	color	
hidemargin	flag	
image	image	
indicatoron	flag	
label	string	
menu	widget	
offvalue	value	
onvalue	value	
selectcolor	color	
selectimage	image	
state	constant	
underline	integer	
value	value	

variable	variable	
----------	----------	--

add_cascade

`add_cascade(options...)`.

add_checkbutton

`add_checkbutton(options...)`.

add_command

`add_command(options...)`.

add_radiobutton

`add_radiobutton(options...)`.

add_separator

`add_separator(options...)`.

delete

`delete(index)`, `delete(start, stop)`.

entryconfig

`entryconfig(index, options...)`.

entryconfigure

`entryconfigure(index, options...)`.

index

`index(index)`.

insert

`insert(index, itemType, options...)`.

insert_cascade

`insert_cascade(index, options...)`.

insert_checkbutton

`insert_checkbutton(index, options...)`.

insert_command

`insert_command(index, options...)`.

insert_radiobutton

`insert_radiobutton(index, options...)`.

insert_separator

`insert_separator(index, options...)`.

invoke

`invoke(index)`.

post

`post(x, y)`.

unpost

`unpost()`.

yposition

`yposition(index)`.

Helpers

The following methods are only relevant if you're implementing your own keyboard bindings. They are not documented in this version of the Tkinter overview.

`tk_bindForTraversal()`.

`tk_firstMenu()`.

`tk_getMenuButtons()`.

`tk_invokeMenu()`.

`tk_mbButtonDown()`.

`tk_mbPost()`.

`tk_mbUnpost()`.

`tk_nextMenu(count)`.

`tk_nextMenuEntry(count)`.

`tk_popup(x, y, entry='')`.

`tk_traverseToMenu(char)`.

`tk_traverseWithinMenu(char)`.

Options

Table 2.

Option	Type	Description
<code>activebackgrou</code>	<code>color</code>	

Option	Type	Description
nd		
activeborderwidth	distance	
activeforeground	color	
background (bg)	color	
borderwidth (bd)	distance	
cursor	cursor	The cursor to show when the mouse pointer is placed over the button widget. Default is a system specific arrow cursor.
disabledforeground	color	
font	font	
foreground (fg)	color	
postcommand	callback	
relief	constant	Border decoration. The default is <i>RAISED</i> . Other possible values are <i>FLAT</i> , <i>SUNKEN</i> , <i>GROOVE</i> and <i>RIDGE</i> .
selectcolor	color	
takefocus	flag	Indicates that the user can use the <i>Tab</i> key to move to this widget. Default is an empty string, which means that the menu accepts focus only if it has any keyboard bindings (default is on, in other words).
tearoff	flag	
tearoffcommand	callback	
title	string	
type	constant	

The Menubutton Widget

Patterns

Methods

Options

The Message Widget

Patterns

Methods

The *Message* widget supports the standard Tkinter Widget interface. There are no additional methods.

Options

The *Message* widget support the following options:

Table 1.

Option	Type	Description
anchor	constant	
aspect	value	
background (bg)	color	
cursor	cursor	The cursor to show when the mouse pointer is placed over the message widget. Default is a system specific arrow cursor.
font	font	
foreground (fg)	color	
highlightbackground, highlightcolor	color	Controls how to draw the focus highlight border. When the widget has focus, the border is drawn in the <i>highlightcolor</i> color. Otherwise, it is drawn in the <i>highlightbackground</i> color. The defaults are system specific.
highlightthickness	distance	Controls the width of the focus highlight border. Default is 0 (no border).
justify	constant	
padx, pady	distance	
relief	constant	Border decoration. The default is <i>FLAT</i> . Other possible values are <i>SUNKEN</i> , <i>RAISED</i> , <i>GROOVE</i> and <i>RIDGE</i> . Note that to show the border, you need to change the <i>borderwidth</i> from it's default value of 0.
borderwidth	distance	Border width. The default is 0 (no border).

Option	Type	Description
(bd)		
takefocus	flag	Indicates that the user can use the <i>Tab</i> key to move to this widget. Default is an empty string, which means that the message accepts focus only if it has any keyboard bindings (default is off, in other words).
text	string	
textvariable	variable	
width	distance	

The 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.

When to use the Pack Manager

To be added.

Warning

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

To be added.

Methods

The following methods are available on widgets managed by the pack manager:

Widget Methods

The following methods are available on widgets managed by the pack manager:

pack

`pack(option=value, ...)`, `pack_configure(option=value, ...)`. Pack the widget as described by the options (see below).

pack_forget

`pack_forget()`. Remove the widget. The widget is not destroyed, and can be displayed again by *pack* or any other manager.

pack_info()

`pack_info()`. Return a dictionary containing the current options.

Manager Methods

The following methods are available on widgets that are used as pack managers (that is, the geometry *masters* for widgets managed by the pack manager).

pack_propagate()

pack_propagate().

pack_slaves()

pack_slaves(). Returns a list of the "slave" widgets managed by this widget. The widgets are returned as Tkinter widget references.

Options

The following options can be used with the *pack* and *pack_configure* methods:

Table 1.

Option	Type	Description
side	constant	Specifies which side to pack the widget against. To pack widgets vertically, use <i>TOP</i> (default). To pack widgets horizontally, use <i>LEFT</i> . You can also pack widgets along the <i>BOTTOM</i> and <i>RIGHT</i> edges. You can mix sides in a single geometry manager, but the results may not be what you expect. While you can create pretty complicated layouts by nesting <i>Frame</i> widgets, you may prefer using the <i>grid</i> geometry manager for all non-trivial layouts.
fill	constant	Specifies whether the widget should occupy all the space given to it by the master. If <i>NONE</i> (default), keep the widget's original size. If <i>X</i> (horizontally), <i>Y</i> (vertically), or <i>BOTH</i> , fill the given space along that direction. To make a widget fill the entire master widget, set <i>fill</i> to <i>BOTH</i> and <i>expand</i> to a non-zero value.
expand	flag	Specifies whether the widgets should be expanded to fill any extra space in the geometry master. If zero (default), the widget is not expanded.
in (in_)	widget	Pack 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 <i>in</i> is a reserved word in Python. To use it as a keyword option, append an underscore (<i>in_</i>).

The PhotoImage Class

Patterns

Methods

configure

`configure(options)`, `config(options)`. Change one or more configuration options.

cget

`cget(option)`. Return the value of the given configuration option.

width, height

`width()`, `height()`. Returns the width (height) of the image, in pixels.

type

`type()`. Returns the string "photo".

get

`get(x, y)`. Fetch the pixel at the given position (where (0, 0) is in the upper left corner).

As of Python 1.5.2, this method returns a string containing one or three pixel components. Here's how to convert this string to either an integer or a 3-tuple of integers:

```
optionvalue = im.get(x, y)
if type(value) == type(""):
    try:
        value = int(value)
    except ValueError:
        value = tuple(map(int, string.split(value)))
```

put

`put(data)`, `put(data, bbox)`. Write pixel data to the image.

read

`read()`. Not supported in 1.5.2 or earlier.

write

`write(filename, options)`. Save the contents of the PhotoImage to a file using the given format. The following options can be used:

Table 1.

Option	Type	Description
format	string	Specifies the format handler to use when writing this image. This is typically "gif" or "ppm"
from_coords	tuple	Save only a part of the image. If a 2-tuple is given, <i>write</i> saves the rectangle between that position, and the lower right corner of the image. If a 4-tuple is given, it specifies which rectangle to save.

blank

`blank()`. Clears the image. The size is left as it is, but the contents are made completely transparent.

copy

`copy()`. Duplicate the current PhotoImage instance.

zoom

`zoom(xscale, yscale)`, `zoom(scale)`. Resize the image to $(xscale * width, yscale * height)$ pixels, using nearest neighbour resampling. In other words, each pixel in the source image will be expanded to $xscale * yscale$ pixels. If only one scale is given, it is used for both directions.

subsample

`subsample(xscale, yscale)`, `subsample(scale)`. Resize the image to $(xscale / width, yscale / height)$ pixels, using nearest neighbour resampling. If only one scale is given, it is used for both directions.

Options

The *PhotoImage* class supports the following options.

Table 2.

Option	Type	Description
file	string	Read image data from the given file. The file can contain GIF, PGM (grayscale), or PPM (truecolor) data. Transparent regions in the GIF file are made transparent. To handle other file formats, use the corresponding class in the Python Imaging Library.
data	string	Read image data from a string. In the current version of Tk, this only works for base64-encoded GIF files. If the <i>file</i> option is given, this option is ignored.

Option	Type	Description
width, height	integer	The width (height) of the image memory. Note that this is the requested size, not the actual size. To get the actual size, use the corresponding methods.
format	string	If several file handlers can handle the given file, this option can be used to specify which handler to use. If you haven't installed extra file handlers, there's no need to use this option.
gamma	float	The image gamma. To get fully accurate colors, this should be set to a combination of the gamma values for the image and display. Default is 1.0 (no gamma correction).
palette	integer or string	Specifies the number of palette entries to use when displaying this image. You can either use a single integer to display the image as a grayscale image with that number of grayscale levels, or a string with three numbers separated by slashes, to display the image as a color image with that number of red, green, and blue values. The default is system specific.

The 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 place

It is usually not a good idea to use *place* for ordinary window and dialog layouts; its simply too 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 *DrawnButton* in the upper right corner of the frame. The button will overlap the label.

```
pane = Frame(master)
Label(pane, text="Pane Title").pack()
b = DrawnButton(pane, (12, 12), 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)
```

Methods

place

`place(option=value, ...)`, `place_configure(option=value, ...)`. Place the widget as described by the options (see below).

`place_configure` is not available in the Tkinter version shipped with Python 1.4. For compatibility, use `place` instead.

place_forget

`place_forget()`. Remove the widget. The widget is not destroyed, and can be displayed again by `place` or any other manager.

place_info

`place_info()`. Return a dictionary containing the current options.

place_slaves

`place_slaves()`. Returns a list of the "slave" widgets managed by this widget. The widgets are returned as Tkinter widget references.

Options

The following options can be used with the `place` and `place_configure` methods:

Table 1.

Option	Type	Description
anchor	constant	Specifies which part of the widget that should be placed at the given position. Valid values are <i>N</i> , <i>NE</i> , <i>E</i> , <i>SE</i> , <i>SW</i> , <i>W</i> , <i>NW</i> , or <i>CENTER</i> . Default is <i>NW</i> (the upper left corner, that is).
bordermode	constant	If <i>INSIDE</i> , the size and position are relative to the reference widget's inner size, excluding any border. If <i>OUTSIDE</i> , it's relative to the outer size, including the border. Default is <i>INSIDE</i> . These constants are not defined in Python 1.5.2 and earlier. For compatibility, use the strings "inside" and "outside" instead.
in (in_)	widget	Place widget relative to the given widget. You can only place a widget relative to its parent, or to any descendant of its parent. If this option is not given, it defaults to the parent. Note that <i>in</i> is a reserved word in Python. To use it as a keyword option, append an underscore (<i>in_</i>).
relwidth, relheight	float	Size, relative to the reference widget.
relx, rely	float	Position, relative to the reference widget (usually the parent, unless otherwise specified by the <i>in</i> option). 0.0 is the left (upper) edge, 1.0 is the right (lower) edge.
width, height	integer	Size, in pixels. If omitted, it defaults to the widget's "natural" size.
x, y	integer	Absolute position, in pixels. If omitted, defaults to 0.

The 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.

Radiobutton Patterns

The *Radiobutton* widget is very similar to the check button. To get a proper radio behaviour, 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:

```
v = IntVar()
Radiobutton(master, text="One", variable=v, value=1).pack(anchor=W)
Radiobutton(master, text="Two", variable=v, value=2).pack(anchor=W)
```

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:

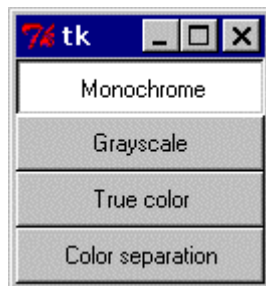
```
MODES = [
    ("Monochrome", "1"),
    ("Grayscale", "L"),
    ("True color", "RGB"),
    ("Color separation", "CMYK"),
]

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 1. 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 2. Using indicatoron=0

Methods

The *Radiobutton* widget supports the standard Tkinter Widget interface, plus the following methods:

deselect

`deselect()`. Deselect the button.

flash

`flash()`. Redraw the button several times, alternating between active and normal appearance.

invoke

`invoke()`. Call the command associated with the button.


select

`select()`. Select the button.

Options

The *Radiobutton* widget supports the following options:

Table 1.

Option	Type	Description
activebackground, activeforeground	color	The color to use when the button is activated.
anchor	constant	Controls where in the button the text (or image) should be located. Use one of <i>N</i> , <i>NE</i> , <i>E</i> , <i>SE</i> , <i>S</i> , <i>SW</i> , <i>W</i> , <i>NW</i> , or <i>CENTER</i> . Default is <i>CENTER</i> . If you change this, it is probably a good idea to add some padding as well, using the <i>padx</i> and/or <i>pady</i> options.
background, foreground	color	The button color. The default is platform specific.
bitmap	bitmap	<p>The bitmap to display in the widget. If the <i>image</i> option is given, this option is ignored.</p> <p>The following bitmaps are available on all platforms: "error", "gray75", "gray50", "gray25", "gray12", "hourglass", "info", "questhead", "question", and "warning".</p>  <p>The following additional bitmaps are available on the Macintosh only: "document", "stationery", "edition", "application", "accessory", "folder", "pfolder", "trash", "floppy", "ramdisk", "cdrom", "preferences", "querydoc", "stop", "note", and "caution".</p> <p>You can also load the bitmap from an XBM file. Just prefix the filename with an at-sign, for example "@sample.xbm".</p>
borderwidth (bd)	int	The width of the button border. The default is platform specific, but is usually 1 or 2 pixels.
command	callback	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.
cursor	cursor	The cursor to show when the mouse is moved over the button.
default	int	If set, the button is a default button. Tk will indicate this by drawing a platform specific indicator (usually an extra border). NOTE: The syntax has

Option	Type	Description
		changed in 8.0b2!!!
disabledforeground	color	The color to use when the button is disabled. The background is shown in the <i>background</i> color.
font	font	The font to use in the button. The button can only contain text in a single font.
highlightbackground, highlightcolor	color	Controls how to draw the focus highlight border. When the widget has focus, the border is drawn in the <i>highlightcolor</i> color. Otherwise, it is drawn in the <i>highlightbackground</i> color. The defaults are system specific.
highlightthickness	distance	Controls the width of the focus highlight border. Default is typically one or two pixels.
image	image	The image to display in the widget. If specified, this takes precedence over the <i>text</i> and <i>bitmap</i> options.
indicatoron	bool	Controls if the indicator should be drawn or not. For check and radio buttons, 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 <i>SUNKEN</i> instead of <i>RAISED</i> . For a menu button, this is off by default. Setting it to true draws a small indicator to the right. This is used by the <i>OptionMenu</i> widget.
justify	constant	Defines how to align multiple lines of text. Use <i>LEFT</i> , <i>RIGHT</i> , or <i>CENTER</i> .
padx, paxy	distance	Button padding. These options specify the horizontal and vertical padding between the text or image, and the button border.
relief	constant	Border decoration. Usually, the button is <i>SUNKEN</i> when pressed, and <i>RAISED</i> otherwise. Other possible values are <i>GROOVE</i> , <i>RIDGE</i> , and <i>FLAT</i> .
selectcolor	color	Color to use for the selector.
selectimage	image	Graphic image to use for the selector.
state	constant	The button state: <i>NORMAL</i> , <i>ACTIVE</i> or <i>DISABLED</i> . Default is <i>NORMAL</i> .
takefocus	flag	Indicates that the user can use the <i>Tab</i> 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).
text	string	The text to display in the button. The text can contain newlines. If the <i>bitmap</i> or <i>image</i> options are used, this option is ignored.

Option	Type	Description
textvariable	variable	Associates a Tkinter variable (usually a <i>StringVar</i>) to the button. If the variable is changed, the button text is updated.
underline	int	Default is -1.
value	None	The value to assign to the associated variable when the button is pressed.
variable	variable	Associates a Tkinter variable to the button. When the button is pressed, the variable is either toggled between <i>offvalue</i> and <i>onvalue</i> (for a <i>Checkbutton</i>), or set to <i>value</i> (for a <i>Radiobutton</i>). Explicit changes to the variable are automatically reflected by the buttons.
width, height	distance	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.
wrapplength	distance	Determines when a button's text should be wrapped into multiple lines. This is given in screen units. Default is no wrapping.

The Scale Widget

Patterns

Methods

get, set

get().

set(value).

Options

Table 1.

Option	Type	Description
activebackground	color	
background (bg)	color	
bigincrement	value	
command	callback	
cursor	cursor	The cursor to show when the mouse pointer is placed over the scale widget. Default is a system specific arrow cursor.
digits	value	
font	font	
foreground (fg)	color	
from (from_)	value	
highlightbackground, highlightcolor	color	Controls how to draw the focus highlight border. When the widget has focus, the border is drawn in the <i>highlightcolor</i> color. Otherwise, it is drawn in the <i>highlightbackground</i> color. The defaults are system specific.
highlightthickness	distance	Controls the width of the focus highlight border. Default is 0 (no border).
label	string	
length	distance	

Option	Type	Description
orient	constant	
relief	constant	Border decoration. The default is <i>FLAT</i> . Other possible values are <i>SUNKEN</i> , <i>RAISED</i> , <i>GROOVE</i> and <i>RIDGE</i> .
borderwidth (bd)	distance	The width of the button border. The default is platform specific, but is usually 1 or 2 pixels.
repeatdelay	time	
repeatinterval	time	
resolution	value	
showvalue	flag	
sliderlength	distance	
sliderrelief	constant	
state	constant	
takefocus	flag	Indicates that the user can use the <i>Tab</i> key to move to this widget. Default is an empty string, which means that the scale accepts focus only if it has any keyboard bindings (default is off, in other words).
tickinterval	time	
to	value	
troughcolor	color	
variable	variable	
width	distance	

The Scrollbar Widget

Patterns

Methods

delta

delta(deltax, deltay).

destroy

destroy().

fraction

fraction(x, y).

get

get().

identify

identify(x, y).

keys

keys().

set

set(args).

Options

Note that most options are ignored on Windows and Macintosh, where the scrollbar is drawn via the native UI toolbox. For best results, use only the *command* and *orient* options in your programs.

Table 1.

Option	Type	Description
orient	constant	Defines how to draw the scrollbar. Use one of <i>HORIZONTAL</i> or <i>VERTICAL</i> . Default is <i>VERTICAL</i> .
command	callback	Used to update the associated widget. This is typically the <i>xview</i> or <i>yview</i> method of the scrolled

Option	Type	Description
		<p>widget.</p> <p>If the user drags the scrollbar slider, the command is called as <i>callback(MOVETO, offset)</i>, 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.</p> <p>If the user clicks the arrow buttons, or clicks in the trough, the command is called as <i>callback(SCROLL, step, what)</i>. The second argument is either "-1" or "1" depending on the direction, and the third argument is <i>UNITS</i> to scroll lines (or other units relevant for the scrolled widget), or <i>PAGES</i> to scroll full pages.</p> <p>These constants are not defined in Python 1.5.2 and earlier. For compatibility, use the strings "moveto", "scroll", "units", and "pages" instead.</p>
activebackground	color	
activerelief	constant	
background (bg)	color	
cursor	cursor	The cursor to show when the mouse pointer is placed over the scrollbar widget. Default is a system specific arrow cursor.
elementborderwidth	distance	
highlightbackground, highlightcolor	color	Controls how to draw the focus highlight border. When the widget has focus, the border is drawn in the <i>highlightcolor</i> color. Otherwise, it is drawn in the <i>highlightbackground</i> color. The defaults are system specific.
highlightthickness	distance	Controls the width of the focus highlight border. Default is 0 (no border). Note that this option is ignored under Windows.
jump	constant	
relief	constant	Border decoration. The default is <i>SUNKEN</i> . Other possible values are <i>FLAT</i> , <i>RAISED</i> , <i>GROOVE</i> , and <i>RIDGE</i> . Note that this option is ignored under Windows.
borderwidth (bd)	distance	Border width. The default is 0 (no border). Note that this option is ignored under Windows.
repeatdelay	time	
repeatinterval	time	

Option	Type	Description
takefocus	flag	Indicates that the user can use the <i>Tab</i> key to move to this widget. Default is an empty string, which means that the scrollbar accepts focus only if it has any keyboard bindings (default is off, in other words).
troughcolor	color	
width	distance	

The 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.

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

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.

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.

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.

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 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 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 append one or more *modifiers*:

"+ *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*" moves 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 strings, you should 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.

Table 1.

Option	Type	Description
background	color	The background color to use for text having this tag. Note that the <i>bg</i> alias cannot be used with tags; it is interpreted as <i>bgstipple</i> rather than <i>background</i> .
bgstipple (bg)	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 <i>bd</i> alias cannot be used with tags.
fgstipple (fg)	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 <i>fg</i> alias cannot be used with tags; it is interpreted as <i>fgstipple</i> rather than <i>foreground</i> .
justify	constant	Controls text justification (the first character on a line determines how to justify the whole line). Use one of <i>LEFT</i> , <i>RIGHT</i> , or <i>CENTER</i> . Default is <i>LEFT</i> .
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 <i>SUNKEN</i> , <i>RAISED</i> , <i>GROOVE</i> , <i>RIDGE</i> , or <i>FLAT</i> . Default is <i>FLAT</i> (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 <i>NONE</i> , <i>CHAR</i> , or <i>WORD</i> .

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).

Basic Methods

The *Listbox* widget supports the standard Tkinter Widget interface, plus the following methods:

insert

`insert(index, text)`, `insert(index, text, tags)`. Insert text at the given position (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.

delete

`delete(index)`, `delete(start, stop)`. Delete 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.

get

`get(index)`, `get(start, stop)`. Return the character at the given position, or all text in the given range.

dump

`dump(index, options...)`, `dump(start, stop, options...)`. Return a list of widget contents at the given position, or for all text in the given range. This includes tags, marks, and embedded objects. Not implemented in Python 1.5.2 and earlier.

see

`see(index)`, `yview(index)`. If necessary, scroll the text widget to make sure the text at the given position is visible. The *see* method scrolls the widget only if the given position isn't visible at all, while *yview* always scrolls the widget to move the given position to the top of the window.

index

`index(index)`. Return the "line.column" index corresponding to the given index.

compare

`compare(index1, op, index2)`. Compare the two positions, and return true if the condition held. The *op* argument is one of "<", "<=", "==", ">=", ">", or "!=" (Python's "<>" syntax is not supported).

Methods for Marks

The following methods are used to manipulate builtin as well as user-defined marks.

mark_set

mark_set(mark, index). Move 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.

mark_unset

mark_unset(mark). Remove the given mark from the widget. You cannot remove the builtin *INSERT* and *CURRENT* marks.

index

index(mark). Return the line/column position corresponding to the given mark (or any other index specifier; see above).

mark_gravity

mark_gravity(mark). Return the current gravity setting for the given mark (*LEFT* or *RIGHT*).

mark_gravity(mark, gravity). 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*.

mark_names

mark_names(). Return a tuple containing 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).

Methods for Embedded Windows

The *Text* widget allows you to embed windows into the widget. Embedded windows occupy a single character position, and moves with the text flow.

window_create

window_create(index, options...). Insert 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 callback which is called when the window is first displayed.

Table 2.

Option	Type	Description
align	constant	Defines how to align the window on the line. Use one of <i>TOP</i> , <i>CENTER</i> , <i>BOTTOM</i> , or <i>BASELINE</i> . The last alignment means that the bottom of the window is aligned with the text baseline -- that is,

		the same alignment that is used for all text on the line).
create	callback	This callback is called when the window is first displayed by the text widget. It should create the window (as a child to the text widget), and return the resulting widget instance.
padx, pady	distance	Adds horizontal (vertical) padding between the window and the surrounding text. Default is 0 (no padding).
stretch	flag	If zero (or <i>OFF</i>), the window will be left as is also if the line is higher than the window. If non-zero (or <i>ON</i>), the window is stretched to cover the full line (in this case, the alignment is ignored).
window	widget	Gives the widget instance to insert into the text.

index

`index(window)`. Return the line/column position corresponding to the given window (or any other index specifier; see above).

delete

`delete(window)`. Remove the given window from the text widget, and destroy it.

window_cget

`window_cget(index, option)`. Return the current value of the given option. If there's no window on the given position, this method raises a *TclError* exception.

window_config

`window_config(index, options...)`, `window_configure(index, options...)`. Modifies one or more options. If there's no window on the given position, this method raises a *TclError* exception.

window_names

`window_names()`. Return 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. This will most likely be fixed in future versions.

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
```

Methods for Embedded Images

The *Text* widget allows you to embed images into the widget. Embedded images occupy a single character position, and moves with the text flow.

Note that the image interface is not available in early version of Tkinter (it's not implemented by Tk versions before 8.0). For such platforms, you can display images by embedding *Label* widgets instead.

image_create

`image_create(index, options...)`. Insert an image at the given position. The image is given by the *image* option, and 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.

Table 3.

Option	Type	Description
align	constant	Defines how to align the image on the line. Use one of <i>TOP</i> , <i>CENTER</i> , <i>BOTTOM</i> , or <i>BASELINE</i> . The last alignment means that the bottom of the image is aligned with the text baseline -- that is, the same alignment that is used for all text on the line).
image	image	Gives the image instance to insert into the text.
name	string	Gives the name to use when referring to this image in the text widget. The default is the name of the image object (which is usually generated by Tkinter).
padx, pady	distance	Adds horizontal (vertical) padding between the image and the surrounding text. Default is 0 (no padding).

index

`index(image)`. Return the line/column position corresponding to the given image (or any other index specifier; see above).

delete

`delete(image)`. Remove the given image from the text widget, and destroy it.

image_cget

`image_cget(index, option)`. Return 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.

image_config

`image_config(index, options...)`, `image_configure(index, options...)`. Modifies one or more 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.

image_names

`image_names()`. Return a tuple containing 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).

This method is not implemented in Python 1.5.2 and earlier.

Methods for Tags

The following methods are used to manipulate tags and tag ranges.

tag_add

`tag_add(tag, index)`, `tag_add(tag, start, top)`. Add *tag* to the character at the given position, or to the given range.

tag_remove

`tag_remove(tag, index)`, `tag_remove(tag, start, stop)`. Remove the tag from the character at the given position, or from the given range. The information associated with the tag is not removed (not even if you use `tag_remove(1.0, END)`).

tag_delete

`tag_delete(tag)`, `tag_delete(tags...)`. Remove the given tags from the widget. All style and binding information associated with the tags are also removed.

tag_config

`tag_config(tag, options...)`, `tag_configure(tag, options...)`. Set style options for the given tag. If the tag doesn't exist, it is created.

Note that the style options are associated with tags, not text ranges. Any text having a given tag will be rendered according to its style options, even if it didn't exist when the binding was created. If a text range has several tags associated with it, the *Text* widget combines the style options for all tags. Tags towards the top of the tag stack (created later, or raised using `tag_raise`) have precedence.

tag_cget

`tag_cget(tag, option)`. Get the current value for the given option.

tag_bind

`tag_bind(tag, sequence, func)`, `tag_bind(tag, sequence, func, "+")`. Add an event binding to the given tag. Tag bindings can use mouse- and keyboard-related events, plus *Enter* and *Leave*. If the tag doesn't exist, it is created. Usually, the new binding replaces any existing binding for the same event sequence. The second form can be used to add the new callback to the existing binding.

Note that the new bindings are associated with tags, not text ranges. Any text having the tag will fire events, even if it didn't exist when the binding was created. To remove bindings, use `tag_remove` or `tag_unbind`.

tag_unbind

`tag_unbind(tag, sequence)`. Remove the binding, if any, for the given tag and event sequence combination.

tag_names

`tag_names()`. Return a tuple containing all tags used in the widget. This includes the *SEL* selection tag.

`tag_names(index)`. Return a tuple containing all tags used by the character at the given position.

tag_nextrange

`tag_nextrange(tag, index)`, `tag_nextrange(tag, start, stop)`. Find the next occurrence of the given tag, starting at the given index. If two indexes are given, search only from *start* to *stop*. Note that this method looks for the start of a range, so if you happen to start on a character that has the given tag, this method will return that range *only* if that character is the first in the range. Otherwise, the current range is skipped.

tag_prevrange

`tag_prevrange(tag, index)`, `tag_prevrange(tag, start, stop)`. Find the next occurrence of the given tag, starting at the given index and searching towards the beginning of the text. If two indexes are given, search from *start* to *stop*. As for *nextrange*, this method looks for the start of a range, beginning at the start index. So if you start on a character that has the given tag, this method will return that range *unless* the search started on the first character in that tag range.

tag_lower

`tag_lower(tag)`, `tag_lower(tag, below)`. Move the given tag to the bottom of the tag stack (or place it just under the *below* tag). If multiple tags are defined for a range of text, options defined by tags towards the top of the stack have precedence.

tag_raise

`tag_raise(tag)`, `tag_raise(tag, above)`. Move the given tag to the top of the tag stack (or place it just over the *above* tag).

tag_ranges

`tag_ranges(tag)`. Return a tuple with start- and stop-indexes for each occurrence of the given tag. If the tag doesn't exist, this method returns an empty tuple. Note that the tuple contains two items for each range.

Methods for Selections

To manipulate the selection, use tag methods like `tag_add` and `tag_remove` on the *SEL* tag. There are no selection-specific methods provided by the *Text* widget.

But if you insist, here's how how to emulate the *Entry* widget selection methods:

```
def selection_clear(text):
    text.tag_remove(SEL, 1.0, END)

def selection_from(text, index):
    text._anchor = index

def selection_present(text):
    return len(text.tag_ranges(SEL)) != 0

def selection_range(text, start, end):
    text.tag_remove(SEL, 1.0, start)
    text.tag_add(SEL, start, end)
    text.tag_remove(SEL, end, END)

def selection_to(text, index):
    if text.compare(index, "<", text._anchor):
        selection_range(text, index, text._anchor)
    else:
        selection_range(text, text._anchor, index)
```

Methods for Rendering

The following methods only work if the text widget is updated. To make sure this is the case, call the `update_idletasks` method before you use any of these.

bbox

`bbox(index)`. Returns the bounding box for the given character, as a 4-tuple: (*x*, *y*, *width*, *height*). If the character is not visible, this method returns `None`.

dlineinfo

`dlineinfo(index)`. Returns the bounding box for the line containing the given character, as 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`.

Methods for Printing

The *Text* widget doesn't contain any builtin support for printing. To print the contents, use `get` or `dump` and pass the resulting text to a suitable output device.

If you have a Postscript printer, you can use PIL's *PSDraw* module.

Methods for Searching

search

`search(pattern, index, options...)`. Search for text in the widget. Returns the first matching position if successful, or an empty string if there was no match.

Table 4.

Option	Type	Description
forwards, backwards	flag	Search from the given position towards the end of the buffer (<i>forwards</i>), or the beginning (<i>backwards</i>). Default is <i>forwards</i> .
exact, regexp	flag	Interpret the pattern as a literal string (<i>exact</i>), or a Tcl-style regular expression (<i>regexp</i>). Default is <i>exact</i> .
nocase	flag	Enable case-insensitive search. Default is case sensitive.
stopindex	index	Don't search beyond this position. Default is to search the whole buffer, and wrap around if the search reaches the end of the buffer. To prevent wrapping, set <i>stopindex</i> to <i>END</i> when searching forwards, and <i>1.0</i> when searching backwards.
count	variable	Return the length of the match in the given variable. If given, this variable should be a Tkinter <i>IntVar</i> .

Methods for Scrolling

These methods are used to scroll the text widget in various ways. The *scan* methods can be used to implement fast mouse pan/roam operations (they are bound to the middle mouse button, if available), while the *xview* and *yview* methods are used with standard scrollbars.

scan_mark, scan_dragto

`scan_mark(x, y)`, `scan_dragto(x, y)`. *scan_mark* sets the scanning anchor for fast horizontal scrolling to the given mouse coordinate. *scan_dragto* 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.

xview, yview

`xview()`, `yview()`. Returns a tuple containing two values; the first value corresponds to the relative offset of the first visible line (column), and the second corresponds to the relative offset of the line (column) just after the last one visible on the screen. Offset 0.0 is the beginning of the text, 1.0 the end.

xview, yview

`xview(MOVETO, offset)`, `yview(MOVETO, offset)`. Adjust the text widget so that the given offset is at the left (top) edge of the text. Offset 0.0 is the beginning of the text, 1.0 the end. These methods are used by the *Scrollbar* bindings when the user drags the scrollbar slider.

The *MOVETO* constant is not defined in Python 1.5.2 and earlier. For compatibility, use the string "moveto" instead.

xview, yview

`xview(SCROLL, step, what)`, `yview(SCROLL, step, what)`. Scroll the text widget horizontally (vertically) by the given amount. The *what* argument can be either *UNITS* (lines, characters) or *PAGES*. These methods are used by the *Scrollbar* bindings when the user clicks at a scrollbar arrow or in the trough.

These constants are not defined in Python 1.5.2 and earlier. For compatibility, use the strings "scroll", "units", and "pages" instead.

yview_pickplace

`yview_pickplace(index)`. Same as *see*, but only handles the vertical position correctly. New code should use *see* instead.

Options

The *Text* widget supports the following options.

FIXME: *sort in relevance order*

Table 5.

Option	Type	Description
background (bg)	color	The background color for this widget. Default is system specific (usually "white"). If you change the background color, you should make sure to change the foreground color as well.
borderwidth (bd)	distance	Border width. Default is platform dependent, but is usually one or two pixels.
cursor	cursor	The cursor to show when the mouse pointer is placed over the text widget. The default is a text insertion cursor (typically an "I beam" cursor, e.g. <i>xterm</i>).
exportselection	flag	If true, selected text is automatically exported to the clipboard. Default is true.
font	font	Widget font. The default is system specific (usually "black").
foreground (fg)	color	Text color.
height	distance	Widget height, in text units.

Option	Type	Description
highlightbackground, highlightcolor	color	Controls how to draw the focus highlight border. When the widget has focus, the border is drawn in the <i>highlightcolor</i> color. Otherwise, it is drawn in the <i>highlightbackground</i> color. The defaults are system specific.
highlightthickness	distance	Controls the width of the focus highlight border. Default is 0 (no border).
insertbackground	color	
insertborderwidth	distance	
insertofftime, insertontime	time	
insertwidth	distance	Controls cursor blinking and style. It's usually best to leave these as they are.
padx, pady	distance	Extra padding between the widget's inner border and the text body. Default is 0 (no padding).
relief	constant	Border decoration. The default is <i>SUNKEN</i> . Other possible values are <i>FLAT</i> , <i>RAISED</i> , <i>GROOVE</i> , and <i>RIDGE</i> .
selectbackground	color	Selection background color. The default is system and display specific.
selectborderwidth	distance	Selection border width. The default is system specific.
selectforeground	color	Selection text color. The default is system and display specific.
setgrid	flag	If true, Tkinter attempts to resize the window containing the text widget in full character steps (based on the <i>font</i> option).
spacing1	distance	Spacing to use above the first line in a block of text. Default is 0 (no extra spacing).
spacing2	distance	Spacing to use between the lines in a block of text wrapped by the widget. 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).
state	constant	One of <i>NORMAL</i> or <i>DISABLED</i> . Default is <i>NORMAL</i> . Note that if you set this to <i>DISABLED</i> , calls to <i>insert</i> or <i>delete</i> are ignored.
tabs	string	

Option	Type	Description
takefocus	flag	If true, you can use <i>Tab</i> to move focus to this widget (but not from it; the default bindings for the <i>Text</i> widget insert the tab character). Default is an empty string, which means that the text widget accepts focus only if it has any keyboard bindings (default is on, in other words).
width	distance	Widget width, in text units.
wrap	constant	Word wrap mode. Use one of <i>NONE</i> , <i>CHAR</i> , or <i>WORD</i> . Default is <i>NONE</i>
xscrollcommand, yscrollcommand	callback	Scrollbar callbacks. These options should be set to the <i>set</i> method for the corresponding scrollbar.

The 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".

Methods

Except for the standard widget interface (*config*, etc), the *Toplevel* widget has no methods.

Options

Table 1.

Option	Type	Description
height, width	distance	Toplevel window size.
background (bg)	color	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.
colormap	widget	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.
menu	widget	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.
cursor	cursor	The cursor to show when the mouse pointer is placed over the toplevel widget. Default is a system specific arrow cursor.
relief	constant	Border decoration: either <i>FLAT</i> , <i>SUNKEN</i> , <i>RAISED</i> , <i>GROOVE</i> , or <i>RIDGE</i> . The default is <i>FLAT</i> .
borderwidth (bd)	distance	Width of the 3D border. Defaults to 0 (no border).

Option	Type	Description
takefocus	flag	Indicates that the user can use the <i>Tab</i> key to move to this widget. Default is an empty string, which means that the toplevel accepts focus only if it has any keyboard bindings (default is off, in other words).
highlightbackground, highlightcolor	color	Controls how to draw the focus highlight border. When any child to the toplevel window has focus, the border is drawn in the <i>highlightcolor</i> . Otherwise, it is drawn in the <i>highlightbackground</i> color. The defaults are system specific.
highlightthickness	distance	Controls the width of the focus highlight border. Default is 0 (no border).
class (class_)	class	
visual	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.
screen	screen	
container	container	
use	widget	

Basic Widget Methods

The following methods are provided by all widgets (including the root window). In the method descriptions, *self* refer to the widget via which you reached the method.

The root window and other *Toplevel* windows provide additional methods. See the *Window Methods* section for more information.

Configuration

config

`config(options...)`, `configure(options...)`. Change one or more options for self.

config

`config()`, `configure()`. Return 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.

cget

`cget(option)`. Return the current value for the given 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.

keys

`keys()`. Return 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
```

Event processing

mainloop

`mainloop()`. Enter 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.

quit

`quit()`. Leaves Tkinter's main event loop. Note that you can have nested event loops; each call to *quit* terminates the outermost event loop.

update

`update()`. Process all pending events, call event callbacks, complete any pending geometry management, redraw widgets as necessary, and call 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.)

update_idletasks

`update_idletasks()`. Call 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.

focus_set

`focus_set()`, `focus()`. Move keyboard focus to self. This means that all keyboard events sent to the application will be routed to self.

focus_displayof

`focus_displayof()`.

focus_force

`focus_force()`. Force keyboard focus to self.

FIXME: *what's the difference between "moving" and "forcing"?*

focus_get

`focus_get()`.

focus_lastfor

`focus_lastfor()`.

tk_focusNext

`tk_focusNext()`. Return the next widget (following self) that should have focus. This is used by the default bindings for the *Tab* key.

tk_focusPrev

`tk_focusPrev()`. Return the previous widget (preceding self) that should have focus. This is used by the default bindings for the *Shift-Tab* key.

grab_current

`grab_current()`.

grab_release

`grab_release()`. Release the event grab.

grab_set

`grab_set()`. Route all events for this application to *self*.

grab_set_global

`grab_set_global()`. Route all events for the entire screen to *self*.

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

`grab_status()`.

wait_variable

`wait_variable(variable)`. Wait 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

`wait_visibility(widget)`. 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

`wait_window(widget)`. Wait 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.

Event callbacks

All event callbacks take one argument; an event descriptor. See the introduction for more information on this descriptor.

bind

`bind(sequence, callback)`, `bind(sequence, callback, "+")`. Add an event binding to self. Usually, the new binding replaces any existing binding for the same event sequence. The second form can be used to add the new callback to the existing binding.

unbind

`unbind(sequence)`. Remove any bindings for the given event sequence, for self.

bind_all

`bind_all(sequence, callback)`, `bind_all(sequence, callback, "+")`. Add an event binding to the application level. Usually, the new binding replaces any existing binding for the same event sequence. The second form can be used to add the new callback to the existing binding.

unbind_all

`unbind_all(sequence)`. Remove any bindings for the given event sequence, on the application level.

bind_class

`bind_class(class, sequence, func)`, `bind_class(class, sequence, func, "+")`. Add an event binding to the given widget class. Usually, the new binding replaces any existing binding for the same event sequence. The second form can be used to add the new callback to the existing binding.

unbind_class

`unbind_class(class, sequence)`. Remove any bindings for the given event sequence, for the given binding class.

bindtags

`bindtags()`. Return a tuple containing the binding search order used for self. By default, this tuple contains the self's widget 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.

bindtags

`bindtags(bindings)`. Modify the binding search order for self.

Alarm handlers and other non-event callbacks

after

`after(delay_ms, callback, args...)`. Register an alarm callback that is called after the 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 *after*. 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):
        ...
        self.master.after(100, self.poll)
```

You can provide one or more arguments which are passed to the callback. This method returns an alarm id which can be used with *after_cancel* to cancel the callback.

after_cancel

`after_cancel(id)`. Cancels the given alarm callback.

after

`after(delay_ms)`. Wait for the given number of milliseconds. Note that in the current version, this also blocks the event loop. In practice, this means that you might as well do:

```
time.sleep(delay_ms*0.001)
```

after_idle

`after_idle(callback, args...)`. Register an idle callback which is called when the system is idle (that is, when there are no more events to process in the mainloop). The callback is only called once for each call to *after_idle*.

Window management

lift

`lift()`, `tkraise()`, `lift(above)`, `tkraise(above)`. Move *self* to the top of the window stack. If *self* is a child window, it is moved to the top of its toplevel window. If *self* 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

`lower()`, `lower(below)`. Same as *lift*, but moves the widget to the bottom of the stack (or places it just under the *below* widget).

Window Related Information

This group of methods provide information related to the widget (*self*) to which the method belongs.

winfo_cells

`winfo_cells()`. Return the number of "cells" in the color map for self. This is typically a value between 2 and 256 (also for true color displays, by some odd reason).

winfo_children

`winfo_children()`. Return a list containing widget instances for all children of self. 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

`winfo_class()`. Returns the Tkinter widget class name for self. If self is a Tkinter base widget, `widget.winfo_class()` is the same as `widget.__class__.__name__`.

winfo_colormapfull

`winfo_colormapfull()`. Return true if the color map for self is full.

winfo_containing

`winfo_containing(x, y)`. Return 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.

winfo_depth

`winfo_depth()`. Return the bit depth used to display self. 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

`winfo_exists()`. Return true if there is Tk window corresponding to self. Unless you've done something really strange, this method should always return true.

winfo_pixels

`winfo_pixels(distance)`, `winfo_fpixels(distance)`. Convert the given distance (in any form accepted by Tkinter) to the corresponding number of pixels. *winfo_pixels* returns an integer value, *winfo_fpixels* a floating point value.

winfo_geometry

`winfo_geometry()`. Returns a string describing self's "geometry". The string has the following format:

```
"%dx%d%+d%+d" % (width, height, xoffset, yoffset)
```

where all coordinates are given in pixels.

winfo_width, winfo_height

`winfo_width()`, `winfo_height()`. Return the width (height) of self, in pixels. Note that if the window isn't managed by a geometry manager, these methods returns 1. To you get the real value, you may have to call `update_idletasks` first. You can also use `winfo_reqheight` to get the widget's requested height (that is, the "natural" size as defined by the widget itself based on it's contents).

winfo_id

`winfo_id()`. Return a string containing a system-specific window identifier corresponding to self. For Unix, this is the X window identifier. For Windows, this is the HWND cast to a long integer.

winfo_ismapped

`winfo_ismapped()`. Return true if there is window corresponding to self in the underlying window system (an X window, a Windows HWND, etc).

winfo_manager

`winfo_manager()`. Return the name of the geometry manager used to keep manage self (typically one of *grid*, *pack*, *place*, *canvas*, or *text*).

FIXME: *this is not implemented by Tkinter (or is it, in 1.5.2?)*

winfo_name

`winfo_name()`. Return the Tk widget name. This is the same as the last part of the full widget name (which you can get via `str(widget)`).

winfo_parent

`winfo_parent()`. Return the full widget name of self's parent, or an empty string if self doesn't have a parent (if self is the root window, 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 an instance.

winfo_pathname

`winfo_pathname(id)`. Return the full window name for the window having the given identity (see `winfo_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`.

winfo_reqheight, winfo_reqwidth

`winfo_reqheight()`, `winfo_reqwidth()`. Return the "natural" height (width) for self. 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.

winfo_rootx, winfo_rooty

`winfo_rootx()`, `winfo_rooty()`. Return the pixel coordinates for self's upper left corner, relative to the screen's upper left corner.

winfo_screen

`winfo_screen()`. Return the X window screen name for the current window. The string has the following format:

":%d.%d" % (display, screen)

On Windows and Macintosh, this is always ":0.0".

winfo_screencells

`winfo_screencells()`. Returns the number of "cells" in the default color map for self's screen.

winfo_screendepth

`winfo_screendepth()`. Return the default bit depth for self's screen.

winfo_screenwidth, winfo_screenheight

`winfo_screenwidth()`, `winfo_screenheight()`. Return the width (height) of self's screen, in pixels.

winfo_screenmmwidth, winfo_screenmmheight

`winfo_screenmmwidth()`, `winfo_screenmmheight()`. Return the width (height) of self's screen, in millimetres. This may not be accurate on all platforms.

FIXME: *does this take the logical resolution into account on Windows?*

winfo_screenuisual

`winfo_screenuisual()`. Return the "visual" type used for self. This is typically "pseudocolor" (for 256-color displays) or "truecolor" (for 16- or 24-bit displays).

Other possible values (on X window systems only) include "directcolor", "staticcolor", "grayscale", or "staticgray".

winfo_toplevel

`winfo_toplevel()`. Return the toplevel window (or root) window for self, as a widget instance.

winfo_visual

`winfo_visual()`. Return a string describing the display type (the X window "visual") for self's screen. This is one of *staticgray*, *grayscale*, *staticcolor*, *psuedocolor*, *directcolor*, or

truecolor. For most display devices, this is either *psuedocolor* (an 8-bit colormapped display), or *truecolor* (a 15- or 24-bit truecolor display).

winfo_x, winfo_y

`winfo_x()`, `winfo_y()`. Return the pixel coordinates for self's upper left corner, relative to its parent's upper left corner.

Miscellaneous

bell

`bell()`. Generate a system-dependent sound (typically a short beep).

clipboard_append

`clipboard_append(string)`. Add text to the clipboard.

clipboard_clear

`clipboard_clear()`. Clear the clipboard.

selection_clear

`selection_clear()`.

selection_get

`selection_get()`.

selection_handle

`selection_handle(command)`.

selection_own

`selection_own()`.

selection_own_get

`selection_own_get()`.

tk_focusFollowsMouse

`tk_focusFollowsMouse()`.

tk_strictMotif

`tk_strictMotif(flag)`. Under Unix, this method can be called to enforce strict Motif look and feel. To use this, create a root window by calling the *Tk* constructor, and then call this method with flag set to 1 before you create any other widgets. This method has no effect on other platforms.

winfo_rgb

`winfo_rgb(color)`. Convert a color string (in any form accepted by Tkinter) to a 3-tuple containing the corresponding red, green, and blue component. Note that the tuple contains 16-bit values (0..65535).

Tkinter Interface Methods

The following methods are used by Tkinter's inner workings. Don't use these unless you know exactly what you are doing, and why you should do that.

getboolean

`getboolean(s)`. Convert a string to a boolean (flag) value, using Tcl's conventions.

getdouble

`getdouble(s)`. Convert a string to a floating point value, using Tcl's conventions. In practice, this is equivalent to *float* and *string.atof*.

getint

`getint(s)`. Convert a string to an integer point value, using Tcl's conventions. In practice, this is equivalent to *int* and *string.atoi*.

register

`register(callback)`. Register 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.

winfo_atom

`winfo_atom(string)`. Map the given to a unique integer. Everytime you call this method with the same string, the same integer will be returned.

winfo_atomname

`winfo_atomname(id)`. Return 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 *winfo_atomname* to find out what they are used for.

Option Database

Not yet documented.

option_add

`option_add(pattern, value)`.

option_clear

`option_clear()`.

option_get

`option_get(name, className)`.

option_readfile

`option_readfile(fileName)`.

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.

Visibility Methods

deiconify

`deiconify()`. Display 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.

iconify

`iconify()`. Turn 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".

withdraw

`withdraw()`. Remove 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".

state

`state()`. Returns the current state of self. This is one of the values "normal", "iconic" (see *iconify*), "withdrawn" (see *withdraw*) or "icon" (see *iconwindow*).

Style Methods

title

`title(string)`, `title()`. Set (get) the window title.

group

`group(window)`. Adds *self* 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 behaviour depends on the window manager in use).

transient

`transient(master)`. Make *self* 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.

overrideredirect

`overrideredirect(flag)`, `overrideredirect()`. Set (get) 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.

Window Geometry Methods

geometry

`geometry()`. Returns a string describing self's "geometry". The string has the following format:

```
"%dx%d%+d%+d" % (width, height, xoffset, yoffset)
```

where all coordinates are given in pixels.

geometry

`geometry(geometry)`. Change the geometry for *self*. The string format is as described above.

aspect

`aspect(minNumer, minDenom, maxNumer, maxDenom)`, `aspect()`. Control 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.

maxsize

`maxsize(width, height)`, `maxsize()`. Set (get) the maximum size for this window.

minsize

`minsize(width, height)`, `minsize()`. Set (get) the minimum size for this window.

resizable

`resizable(width, height)`, `resizable()`. Set (get) 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.

Icon Methods

iconbitmap

`iconbitmap(bitmap)`, `iconbitmap()`. Set (get) the icon bitmap to use when this window is iconified. This method are 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 (see below).

iconmask

`iconmask(bitmap)`, `iconmask()`. Set (get) the icon bitmap mask to use when this window is iconified. This method are ignored by some window managers (including Windows).

iconname

`iconname(newName=None)`, `iconname()`. Set (get) the icon name to use when this window is iconified. This method are ignored by some window managers (including Windows).

iconposition

`iconposition(x, y)`, `iconposition()`. Set (get) the icon position hint to use when this window is iconified. This method are ignored by some window managers (including Windows).

iconwindow

`iconwindow(window)`, `iconwindow()`. Set (get) the icon window to use as an icon when this window is iconified. This method are ignored by some window managers (including Windows).

Property Access Methods

client

`client(name)`, `client()`. Set (get) 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.

colormapwindows

`colormapwindows(wlist...)`, `colormapwindows()`. Set (get) the `WM_COLORMAP_WINDOWS` property. This property is used by window managers under the X window system. It is ignored on other platforms.

command

`command(value)`, `command()`. Set (get) the `WM_COMMAND` 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.

focusmodel

focusmodel(model), focusmodel(). Set (get) the focus model.

frame

frame(). Return a string containing a system-specific window identifier corresponding to self'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 self.

positionfrom

positionfrom(who), positionfrom(). Set (get) the position controller.

protocol

protocol(name, function). Register *function* as a callback which will be called for the given protocol. The *name* argument is typically one of *BWM_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).

sizefrom

sizefrom(who), sizefrom(). Set (get) the size controller.