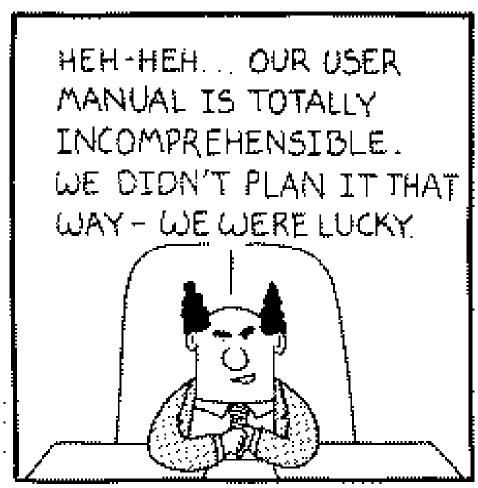
A Tkinter Life Preserver

a wee bit of documentation for the Python/Tk toolkit
until the real thing comes along



Panel from Scott Adams's daily comic strip *Dilbert*, Sept. 1994

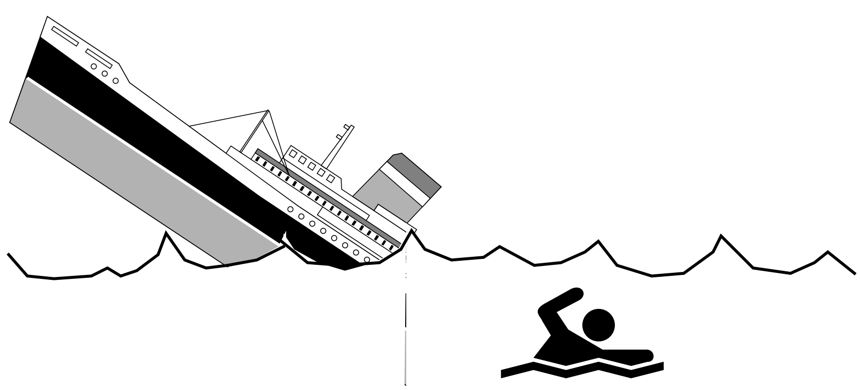


Table of Contents

Introduction

Tkinter is a Python module that gives programmers access to the Tk GUI toolkit from inside the Python programming language.

This document is not designed to be an exhaustive tutorial on either Tk or Tkinter. Rather, this paper is a stop gap effort to help answer some questions for programmers who have decided to use this package but who need something more helpful than "read the source code."

This manual assumes that you are familiar with the Python programming language and that you have access to the Tk toolkit man pages.

Python is available from : `http://www.cwi.nl/~guido/Python.html`
Tcl/Tk is available from : `ftp://ftp.cs.berkeley.edu/ucb/tcl`

Tkinter was written by Steen Lumholt and Guido van Rossum (the inventor of the Python programming language). Tk was written by John Ousterhout while at Berkeley, now at Sun Microsystems. This manual was written by Matt Conway at the University of Virginia. Many thanks go to Tommy Burnette, Shuichi Koga, Joe Shochet, and Richard Stoakley, for their excellent suggestions regarding the format and content of this document. Special thanks go to Joe Shochet for helping to produce Tkinter example programs.

How To Use This Manual

This manual is designed in two parts: the first half (roughly) covers background material, while the last half (starting on page 15) can be taken to the keyboard as a handy reference.

When trying to answer questions of the form "how do I do *blah*", it is often best to find out how to do "*blah*" in straight Tk, and then convert this back into the corresponding Tkinter call. Steen Lumholt has done an admirable job in keeping the correspondence between Tk and Tkinter close, so that Python programmers can often guess at the correct Python command by looking at the Tk documentation. This means that in order to use Tkinter, you will have to know a little bit about Tk. This document can't fulfill that role, so the best we can do is point you to the best documentation that exists. Here are some hints:

- ***I strongly suggest getting a copy of the Tk man pages.*** Specifically, the man pages in the "mann" directory are most useful. The "man3" man pages describe the C interface to the Tk library and thus are not especially helpful for script writers.
- ***Example programs*** that show very basic operations with Tkinter are currently available at `http://uvacs.cs.virginia.edu/~mjc4y/tkinter_examples`. This collection is dynamic and we appreciate any suggestions you might have for programs to include. For people who would like one-stop shopping, I will also make this document and the associated example files available to Guido van Rossum for inclusion at the official Python FTP site.
- ***Addison-Wesley publishes a book*** called *Tcl and the Tk Toolkit* by John Ousterhout (ISBN 0-201-63337-X) which is a good introduction to Tcl and Tk for the novice. The book is not exhaustive, and for many details, it defers to the man pages.
- ***Tkinter.py*** is a last resort for most, but can be a good place to go when nothing else makes sense.

A Simple Hello World Program


```
from Tkinter import *

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

    def createWidgets(self):
        self.QUIT = Button(self)
        self.QUIT["text"] = "QUIT"
        self.QUIT["fg"]   = "red"
        self.QUIT["command"] =  self.quit

        self.QUIT.pack({"side": "left"})

        self.hi_there = Button(self)
        self.hi_there["text"] = "Hello",
        self.hi_there["command"] = self.say_hi

        self.hi_there.pack({"side": "left"})


    def __init__(self, master=None):
        Frame.__init__(self, master)
            self.pack()
        self.createWidgets()

app = Application()
app.mainloop()
```

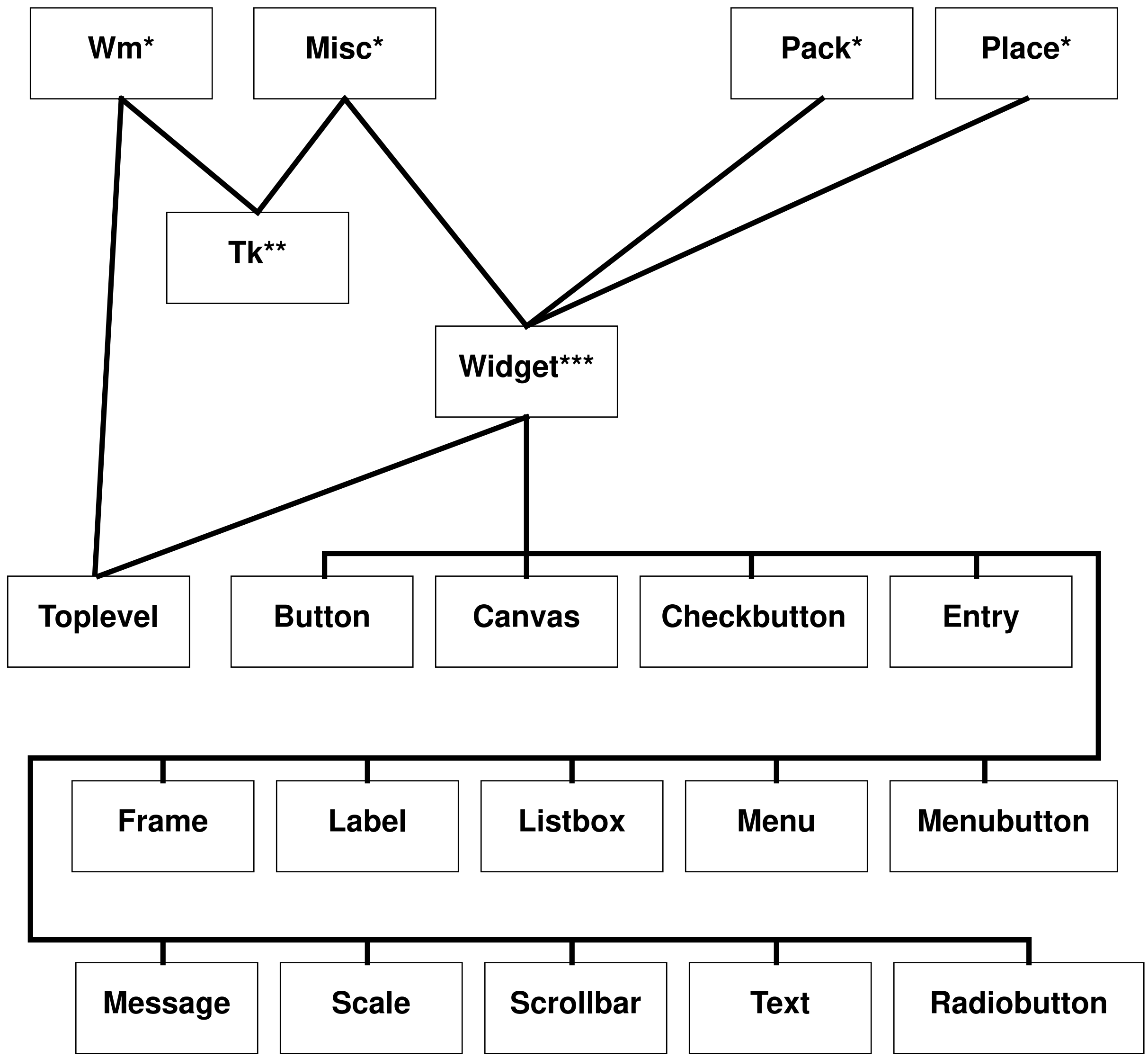
Notes On Hello World

Line		Notes
1		Every Tkinter program imports the Tkinter module. Usually, programmers will import the contents of the entire module into the local namespace with `import *`.
3		The application is modeled as a subclass of an existing widget, in this case, a Frame, which is just an empty container. The constructor for this class will create the widgets for this application The methods of this class are the methods that make sense for this application, and the callbacks to the various application widgets in this case, just the "say hi" method.
7-19		A method, called by the constructor, to create all the application widgets.
	8	Here, we create a new Button, and we call the button QUIT. The `self` parameter is passed into the Button constructor as the master of the new button. Put simply, if widget M is the master of widget W, then W is contained entirely inside M.
	9	We give the new widget a label. Note how we are treating the new widget as if it were a dictionary (see Python programming manual). You can configure an object's attributes ("options" in Tk parlance) by assigning to that object as if it was a dictionary, with the attribute name given as a dictionary key. Legal keys are discussed in the section on attribute setting on page 16.
	10	Again, setting an attribute, making the button's foreground color red.
	11	Another attribute setting. The "command" attribute points to a function to call when the button is pressed. In this case, we call the (inherited) self.quit method which will quit the program.
	13	Pack the quit button as far to the left as possible inside the button's master. This is a call to the geometry manager called the Packer. The packer is discussed briefly on page 12, with more information available in John Ousterhout's book.
	15	Creating a new Button instance.
	16	Setting the new button's label.
	17	This button's command is an application method defined in this class. It might have also been a regular function that takes no arguments.
	19	Pack this button as far to the left as possible, in this case, up against the quit button.
22-25		The constructor for the application.
27		Create the application.
28		Start the application.

Hello World Summary:

- Tkinter.py holds all the class descriptions for the Tk toolkit
- Applications are created by subclassing an existing widget. Frames and Canvases are common choices from which to subclass.
- Callbacks to widgets can either be methods or regular functions.
- Widget constructors take a parameter that denotes the master of the new widget. A widget's master will control where the widget appears in the application and how large it is.
- The actual location and size of a widget is determined by the Packer.
- Before a widget appears on the screen, you must call either pack or place on it.
- `somewidget.mainloop()` starts the program running
- Other simple example programs can be found in `http://uvacs.cs.virginia.edu/~mjc4y/tkinter_examples`

An Overview of The Tkinter Classes



The class hierarchy looks complicated, but in point of fact, application programmer almost always refer to the classes at the very bottom of the hierarchy.

Notes:

* These classes are provided for the purposes of organizing certain functions under one namespace. They aren't meant to be instantiated independently.

** The Tk class is meant to be instantiated only once in an application. Application programmers need not instantiate one explicitly, the system creates one whenever any of the other classes are instantiated.

*** The Widget class is not meant to be instantiated, it is meant only for subclassing to make "real" widgets. (in C++, this is called an 'abstract class')

A (Very) Quick Look at Tcl/Tk

This section is included because unfortunately, most of the documentation for Tkinter comes in the form of the Tk man pages and in John Ousterhout's book. To make use of this reference material, there will be times when you will need to know how to read short passages of Tk and how to identify the various parts of a Tk command.

Tk scripts are Tcl programs. Like all Tcl programs, Tk scripts are just lists of tokens separated by spaces. A Tk widget is just its **class**, the **options** that help configure it, and the **actions** that make it do useful things.

To make a widget in Tk, the command is always of the form:

```
classCommand newPathname options
```

The **classCommand** denotes which kind of widget to make (a button, a label, a menu...)
The **newPathname** is the new name for this widget. All names in Tk must be unique. To help enforce this, widgets in Tk are named with **pathnames**, just like files in a file system. The top level widget, the root, is called "." (period) and children are delimited by more periods. For example, `.myApp.controlPanel.okButton` might be the name of a widget. The **options** configure the widget's appearance and in some cases, its behavior. The options come in the form of a list of flags and values. Flags are proceeded by a -, like a unix shell command flags, and values are put in quotes if they are more than one word. For example:

```
button .fred -fg red -text "hi there"
```

class
command

new widget

options
(in the form: **-optionName** *value*)

Once created, the pathname to the widget becomes a new command. This new **widget command** is the programmer's handle for getting the new widget to perform some **action**. In C, you'd express this as someAction(fred, someOptions), in C++, you would express this as fred.someAction(someOptions), and in Tk, you say:

```
.fred someAction someOptions
```

Note that the object name, .fred, starts with a dot.

As you'd expect, the legal values for *someAction* will depend on the widget's class: `.fred disable` works if fred is a button (fred gets greyed out), but does not work if fred is a label (disabling of labels is not supported in Tk). The legal values of *someOptions* is action dependent. Some actions, like `disable`, require no arguments, others, like a text-entry box's `delete` command, would need arguments to specify what range of text to delete. Unfortunately, John Ousterhout uses the terms "widget command" and "action" interchangeably (p 154). This usually doesn't cause much confusion in practice, but it does muddy the waters for the beginner.

Mapping Basic Tk into Tkinter

Class commands in Tk correspond to class constructors in Tkinter.

Tk		Tkinter
`button .fred`	→	`fred = Button()`

The master of an object is implicit in the new name given to it at creation time. In Tkinter, masters are specified explicitly.

Tk		Tkinter
`button .panel.fred`	→	`fred = Button(panel)`

The configuration options in Tk are given in lists of hyphened tags followed by values. In Tkinter, options are specified either by treating the class instance as an associative array or by passing a dictionary of (tag, value) pairs in at the point of creation. See the section on setting options on page 11.

Tk		Tkinter
`button .fred -fg red`	→	`fred = Button(panel, {"fg" : "red"})`

`.fred configure -fg red`	→	`fred["fg"] = red`
		`OR`
		`fred.config({"fg" : "red"})`

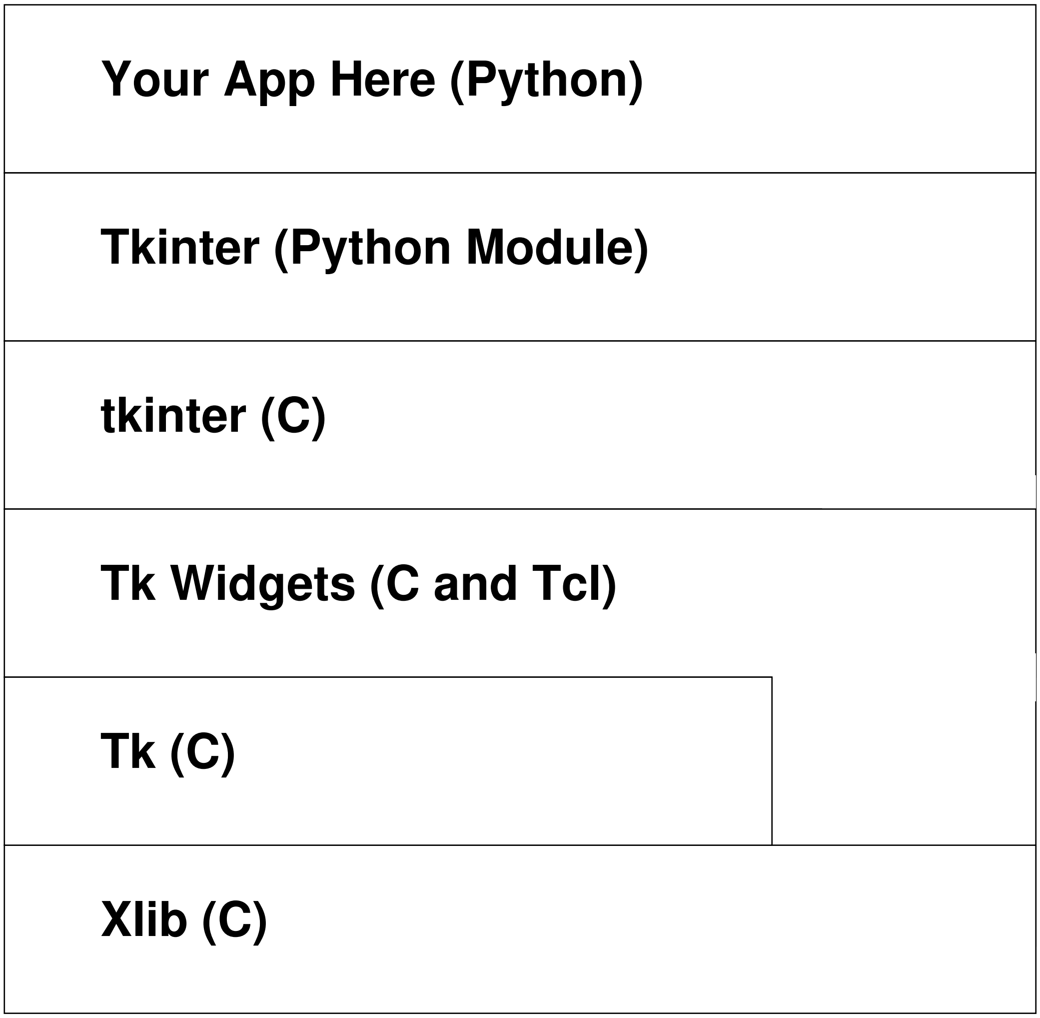
In Tk, to perform an action on a widget, use the widget name as a command, and follow it with an action name, possibly with arguments (options). In Tkinter, you call methods on the class instance to invoke actions on the widget. The actions (methods) that a given widget can perform are listed in the Tkinter.py module.

Tk		Tkinter
`.fred invoke`	→	`fred.invoke()`

To give a widget to the packer (geometry manager), you call pack with optional arguments. In Tkinter, the Pack class holds all this functionality, and the various forms of the pack command are implemented as methods. All widgets in Tkinter are subclassed from the Packer, and so inherit all the packing methods.

Tk		Tkinter
`pack .fred -side left`	→	`fred.pack({"side" : "left"})`

How Tk and Tkinter are Related



From the top down: a Python application makes a Tkinter call. This call (say, for example, creating a button widget), is implemented in the Tkinter module, which is written in Python. This Python function will parse the commands and the arguments and convert them into a form that makes them look as if they had come from a Tk script instead of a Python script. These commands and their arguments will be passed to a C function in the tkinter (note the lowercase) module. This C function is allowed to make calls into other C modules, including the C functions that make up the Tk library. Tk is implemented in C and some Tcl. The Tcl part of the Tk widgets is used to bind certain default behaviors to widgets, and is executed once at the point where the Python Tkinter module is imported. (The user never sees this stage). The C calls made into the Tk library will, in turn, make calls into the Xlib library to draw graphics on the screen.

Setting Options

Options control things like the color and border width of a widget. Options can be set in three ways:

- At object creation time with a dictionary

```
fred = Button(self, {"fg" : "red", "bg" : "blue"})
```

- After object creation, treating the option name like a dictionary index:

```
fred["fg"] = "red"
fred["bg"] = "blue"
```

- After object creation, you can update lots of attributes at once with the `config()` method:

```
fred.config({"fg" : "red", "bg" : "blue"})
```

To find out which options pertain to a given widget, consult the widget reference at the back of this document, starting on page 19. Some options expect certain types; see pages 15 and 16 for details. For a complete explanation of a given option and its behavior, see the man pages for the widget in question.

Note that the man pages list "STANDARD OPTIONS" and "WIDGET SPECIFIC OPTIONS" for each widget. The former is a list of options that are common to many widgets, the latter are the options that are pertinent to that specific widget. The Standard Options are documented on the "options" man page. No distinction is made in this document. Some options don't apply to some kinds of widgets. Whether a given widget responds to a particular option depends on the class of the widget: buttons have a `"command"` option, labels do not.

The options supported by a given widget are listed in that widget's man page, or can be queried at runtime by calling the `config()` method with no dictionary, or by calling the `keys()` method on that widget. The return value of these calls is a dictionary whose key is the name of the option (e.g. 'relief') and whose values are 5 tuples. [1]

Element #	***Value***	***Example***
0	option name	"relief"
1	option name for database lookup[2]	"relief"
2	option class for database lookup	"Relief"
3	default value	"raised"
4	current value	"groove"

Example:

```
print fred.config()

        {"relief" : ("relief", "relief", "Relief", "raised", "groove")}
```

Of course, the dictionary printed will include ***all*** the options available and their values. This is meant only as an example.

1. Some options, like "bg" are synonyms for common options with hard-to-type names ("bg' is shorthand for "background"). Passing the `config()` method the name of a shorthand option will return a 2-tuple, not 5-tuple. The 2-tuple passed back will contain the name of the synonym "real" option. ("bg", "background")
2. Database lookups happen in the internal "options database" or the Xresources database. See John Ousterhout's book, page 253 for details.

The Packer

The packer is one of two geometry managers that come with the Tk toolkit.[1] It is used to define the positions and sizes of the widgets in your interface. Widgets will not appear until they have had the `pack()` method called on them.

The pack() method can be called with a dictionary of (option, value) pairs that control where the widget is to appear and how it is to behave when the main application window is resized. Examples:

```
fred.pack()              #defaults to {"side" : "top"}

fred.pack({"side" : "left"})

fred.pack({"expand" : 1})
```

Every widget in Tk has a master; the size of the master is determined by the size of the "slave widgets" inside. The packer is used to control where slave widgets appear inside the master into which they are packed. You can pack widgets into frames, and frames into other frames in order to achieve the kind of layout you desire.

For more information on the packer and the options that it can take, see the man pages and page 183 of John Ousterhout's book.

Packer Options

For more information on the Packer, see the man pages and Ousterhout's book.

anchor:	Anchor type. Denotes where the packer is to place each slave in its parcel.
expand:	Boolean, 0 or 1
fill:	Legal values: `"x", "y", "both", "none".`
ipadx and ipady:	A distance. Specifies internal padding on each side of the slave widget.
padx and pady:	A distance. Specified external padding on each side of the slave widget.
side:	Legal values are : `"left", "right", "top", "bottom"`.

1.In fact, there is another, less popular geometry manager called the "placer", which is not discussed here, but its behavior is relatively straightforward. The placer is easy to use for small applications, but can make for difficult to create code when the number of widgets is large.

Widgets Setting Variables Automatically

The "current value" of some widgets (like text entry widgets) can be connected directly to application variables via special options. These options are `"variable"` , `"textvariable"`, `"onvalue"`, `"offvalue"`, and `"value"`. This connection works both ways: if the variable changes for any reason, the widget it's connected to will be updated to reflect the new value.

Unfortunately, in the current implementation of Tkinter it is not possible to hand over any arbitrary Python variable to a widget through a `"variable"` or `"textvariable"` option. The only kinds of variables for which this works are variables that are subclassed from a class called `Variable`, defined in the Tkinter module. There are many useful subclasses of `Variable` already defined: `StringVar`, `IntVar`, `DoubleVar`, and `BooleanVar`. To read the current value of such a variable, say, one called `myval`, you call the `get()` method on it, and to change its value you call the `set()` method. If you follow this protocol, the widget will always track the value of the variable, with no further intervention on your part.

For example:

```
class App(Frame):
       def __init__(self, master=None):
              Frame.__init__(self, master)
              self.pack()

              self.entrythingy = Entry()
              self.entrythingy.pack()

              self.button.pack()
              # here is the application variable
              self.contents = StringVar()
              # set it to some value
              self.contents.set("this is a variable")
              # tell the entry widget to watch this variable
              self.entrythingy["textvariable"] = self.contents

              # and here we get a callback when the user hits return.
              # we will have the program print out the value of the application
              # variable when the user hits return
              self.entrythingy.bind('<Key-Return>', self.print_contents)

       def print_contents(self, event):
              print "hi. contents of entry is now ---->", self.contents.get()
```

The Window Manager

In Tk, there is a utility command, Wm, for interacting with the window manager. Options to the wm command allow you to control things like titles, placement. icon bitmaps and the like. In Tkinter, these commands have been implemented as methods on the Wm class. Toplevel widgets are subclassed from the Wm class, and so can call the Wm methods directly.

To get at the toplevel window that contains a given widget, you can often just refer to the widget's master. Of course if the widget has been packed inside of a frame, the master won't represent a toplevel window. To get at the toplevel window that contains an arbitrary widget fred, you can call the `_root()` method. This method begins with an underscore to denote the fact that this function is part of the implementation, and not an interface to Tk functionality.

Here are some examples of typical usage:

```
import Tkinter
class App(Frame):
    def __init__(self, master=None):
       Frame.__init__(self, master)
       self.pack()


# create the application
myapp = App()

####
#### here are method calls to the window manager class
####
myapp.master.title("My Do-Nothing Application")
myapp.master.maxsize(1000, 400)

# start the program
myapp.mainloop()
```

Options Table

Option	Value
activebackground	color
activeborderwidth	distance
activeforeground	color
anchor	anchor type
aspect	integer: 100 * (width/ height)
background (shorthand: bg)	color
bd	distance
bg	color
bitmap	bitmap type
borderwidth (shorthand: bd)	distance
closeenough	float
command	callback type
confine	boolean
cursor	cursor
disabledforeground	color
exportselection	boolean
fg	color
font	font type
foreground (shorthand: fg)	color
from	integer
geometry	geometry type
height	distance
insertbackground	color
insertborderwidth	distance
insertofftime	integer: milliseconds
insertontime	integer: milliseconds
insertwidth	distance
justify	justify type
label	string
length	distance
menu	a Tkinter menu
offvalue	see page 13

Option	Value
onvalue	see page 13
orient	"horizontal" or "vertical"
padx	distance
pady	distance
postcommand	callback type
relief	relief type
repeatdelay	integer: milliseconds
repeatinterval	integer: milliseconds
scrollcommand	scroll command type
scrollincrement	distance
scrollregion	region type
selectbackground	color
selectborderwidth	distance
selectforeground	color
selector	color
setgrid	boolean
showvalue	boolean
sliderforeground	color
sliderlength	distance
state	see widget man page
text	string
textvariable	see page 13
tickinterval	integer
to	integer
underline	integer
value	see page 13
variable	see page 13
width	distance
wrap	wrap type
xscrollcommand	scroll command type
yscrollcommand	scroll command type

Tk Option Data Types

anchor: Legal values are points of the compass:
`"n", "ne", "e", "se", "s", "sw", "w", nw", "center".`

bitmap: There are eight built-in, named bitmaps: `"error","gray25", "gray50", "hourglass", "info", "questhead", "question", "warning".` To specify an X bitmap filename, give the full path to the file, preceded with an `"@"`, as in `"@/usr/contrib/bitmap/gumby.bit".` See page 164 of Ousterhout's book for details.

boolean: You can pass integers 0 or 1 or the stings "yes" or "no"

callback: This is any python function that takes no arguments. For example:

```
def print_it():
    print "hi there"
fred["command"] = print_it
```

color: Colors can be given as the names of X colors in the rgb.txt file, or as strings representing RGB values in 4 bit: "#RGB", 8 bit "#RRGGBB", 12 bit "#RRRGGGBBB", or 16 bit "#RRRRGGGGBBBB" ranges, where R,G,B here represent any legal hex digit. See page 160 of Ousterhout's book for details.

cursor: The standard X cursor names from cursorfont.h can be used, without the XC_. For example to get a hand cursor ("XC_hand2"), use the string "hand2". You can also specify a bitmap and mask file of your own. See page 179 of Ousterhout's book.

distance: Screen distances can be specified in either pixels or absolute distances. Pixels are given as numbers and absolute distances as strings, with the trailing character denoting units: "c" for centimeters, "i" for inches, "m" for millimeters, "p" for printer's points. Ex: 3.5 inches is expressed as `"3.5i"`

font: Tk uses the standard X font name format. (e.g. `"-*-times-*-r-*-*-*-90-*-*-*-*-*-*")`. The `xfontsel` program for X windows can be handy in browsing and selecting fonts. See page 162-163 of Ousterhout's book for details.

geometry: This is a string of the form `"widhtxheight",` where width and height are measured in pixels for most widgets (in characters for widgets displaying text). Example: `fred["geometry"] = "200x100"`

justify: Legal values are the strings: `"left", "center", "right",` and `"fill"`

region: This is a string with four space-delimited elements, each of which is a legal distance (see above). For example: `"2 3 4 5"` and `"3i 2i 4.5i 2i"` and `"3c 2c 4c 10.43c"` are all legal regions.

relief: Determines what the border style of a widget will be. Legal values are:
`"raised", "sunken", "flat", "groove", "ridge"`

scrollcommand: This is almost always the `set()` method of some scrollbar Tkinter widget, but can be any widget method that takes a single argument. See example files for canvas-with-scrollbars.py for details.

wrap: Must be one of: `"none"`, `"char"` or `"word"`.

Bindings and Events

The `bind` method from the widget command allows you to watch for certain events and to have a callback function trigger when that event type occurs. The form of the bind method is:

```
def bind(self, sequence, func, add=''):
```

where :

`sequence` is a string that denotes which kind of event you are looking for (see the bind man page and page 201 of John Ousterhout's book for details.

`func` is a python function that takes one argument. This function will be passed an Event instance when the function is called.

`add` is either "" or "+". Passing an empty string denotes that this binding is to replace any other bindings that this event is associated with. Passing "+" means that this function is to be added to the list of functions bound to this event type.

For example:

```
def turnRed(self, event):
    event.widget["activeforeground"] = "red"

self.button.bind("<Enter>", self.turnRed)
```

Notice how the `widget` field of the event is being accesed in the `turnRed` callback. This field contains the widget that caught the X event. The following table lists the other event fields you can access, and how they are denoted in Tk, which can be useful when referring to the Tk man pages.

Tk	Tkinter Event Field
%f	focus
%h	height
%k	keycode
%s	state
%t	time
%w	width
%x	x
%y	y

Tk	Tkinter Event Field
%A	char
%E	send_event
%K	keysym
%N	keysym_num
%T	type
%W	widget
%X	x_root
%Y	y_root

Indexes[1]

A number of widgets require "index" parameters to be passed. These are used to point at a specific place in a Text widget, or to particular characters in an Entry widget, or to particular menu items in a Menu widget.

Entry widget indexes (index, view index, etc.)

Entry widgets have options that refer to character positions in the text being displayed. You can use these Tkinter functions to access these special points in text widgets:

- `AtEnd()` refers to the last position in the text
- `AtInsert()` refers to the point where the text cursor is,
- `AtSelFirst()` indicates the beginning point of the selected text,
- `AtSelLast()` denotes the last point of the selected text and finally,
- `At(x, y=None)` refers to the character at pixel location x, y (with y not used in the case of a text entry widget, which is one line of text).

Text widget indexes

The index notation for Text widgets is very rich and is best described in the Tk man pages.

Menu indexes (menu.invoke(), menu.entryconfig(), etc.)

Some options and methods for menus manipulate specific menu entries. Anytime a menu index is needed for an option or a parameter, you may pass in:

- an `integer` which refers to the numeric position of the entry in the widget, counted from the top, starting with 0;
- the string `"active"`, which refers to the menu position that is currently under the cursor; `"last"` which refers to the last menu item;
- An integer preceded by `"@"`, as in `"@6"` where the integer is interpreted as a y pixel coordinate in the menu's coordinate system;
- `"none"`, which indicates no menu entry at all, most often used with menu.activate() to deactivate all entries, and finally,
- a text `string` that is pattern matched against the label of the menu entry, as scanned from the top of the menu to the bottom. Note that this index type is considered after all the others, which might interfere with menu items labelled "last", "active" or "none".

1.My webster says "indexes" is ok: in-dex \'in-,deks\ n, pl in-dex-es or in-di-ces \-de-,se^-z\

The Tkinter Class Interfaces

class Misc:

```
def tk_strictMotif(self, boolean=None):
def tk_menuBar(self, *args):
def wait_variable(self, name='PY_VAR'):
waitvar = wait_variable # XXX b/w compat
def wait_window(self, window=None):
def wait_visibility(self, window=None):
def setvar(self, name='PY_VAR', value='1'):
def getvar(self, name='PY_VAR'):
def getint(self, s):
def getdouble(self, s):
def getboolean(self, s):
def focus_set(self):
def focus_default_set(self):
def focus_default_none(self):
def focus_none(self):
def focus_get(self):
def after(self, ms, func=None, *args):
def grab_current(self):
def grab_release(self):
def grab_set(self):
def grab_set_global(self):
def grab_status(self):
def lower(self, belowThis=None):
def selection_clear(self):
def selection_get(self, type=None):
def selection_handle(self, func, type=None, format=None):
def selection_own(self, func=None):
def selection_own_get(self):
def send(self, interp, cmd, *args):
def lower(self, belowThis=None):
def tkraise(self, aboveThis=None):
def colormodel(self, value=None):
def winfo_atom(self, name):
def winfo_atomname(self, id):
def winfo_cells(self):
def winfo_children(self):
def winfo_class(self):
def winfo_containing(self, rootX, rootY):
def winfo_depth(self):
def winfo_exists(self):
def winfo_fpixels(self, number):
def winfo_geometry(self):
def winfo_height(self):
def winfo_id(self):
def winfo_interps(self):
def winfo_ismapped(self):
def winfo_name(self):
def winfo_parent(self):
def winfo_pathname(self, id):
def winfo_pixels(self, number):
def winfo_reqheight(self):
def winfo_reqwidth(self):
def winfo_rgb(self, color):
```

```
def winfo_rootx(self):
def winfo_rooty(self):
def winfo_screen(self):
def winfo_screencells(self):
def winfo_screendepth(self):
def winfo_screenheight(self):
def winfo_screenmmheight(self):
def winfo_screenmmwidth(self):
def winfo_screenvisual(self):
def winfo_screenwidth(self):
def winfo_toplevel(self):
def winfo_visual(self):
def winfo_vrootheight(self):
def winfo_vrootwidth(self):
def winfo_vrootx(self):
def winfo_vrooty(self):
def winfo_width(self):
def winfo_x(self):
def winfo_y(self):
def update(self):
def update_idletasks(self):
def bind(self, sequence, func, add=''):
def bind_all(self, sequence, func, add=''):
def bind_class(self, className, sequence, func, add=''):
def mainloop(self):
def quit(self):
```

---

class Wm:

```
def aspect(self,
def client(self, name=None):
def command(self, value=None):
def deiconify(self):
def focusmodel(self, model=None):
def frame(self):
def geometry(self, newGeometry=None):
def grid(self,
def group(self, pathName=None):
def iconbitmap(self, bitmap=None):
def iconify(self):
def iconmask(self, bitmap=None):
def iconname(self, newName=None):
def iconposition(self, x=None, y=None):
def iconwindow(self, pathName=None):
def maxsize(self, width=None, height=None):
def minsize(self, width=None, height=None):
def overrideredirect(self, boolean=None):
def positionfrom(self, who=None):
def protocol(self, name=None, func=None):
def sizefrom(self, who=None):
def state(self):
def title(self, string=None):
def transient(self, master=None):
def withdraw(self):
```

class Tk(Misc, Wm):

```
def __init__(self, screenName=None, baseName=None, className='Tk'):
def destroy(self):
def __str__(self):
```

class Pack:

```
def config(self, cnf={}):
pack = config
def __setitem__(self, key, value):
def forget(self):
def newinfo(self):
info = newinfo
def propagate(self, boolean=None):
def slaves(self):
```

class Place:

```
def config(self, cnf={}):
place = config
def __setitem__(self, key, value):
def forget(self):
def info(self):
def slaves(self):
```

class Widget(Misc, Pack, Place):

```
def __init__(self, master, widgetName, cnf={}, extra=()):
def config(self, cnf=None):
def __getitem__(self, key):
def __setitem__(self, key, value):
def keys(self):
def __str__(self):
def destroy(self):
```

class Toplevel(Widget, Wm):

```
# this widget is a window managed by the window manager.
# See also the Wm class for the pertinent methods
    def __init__(self, master=None, cnf={}):
```

class Button(Widget):

options:

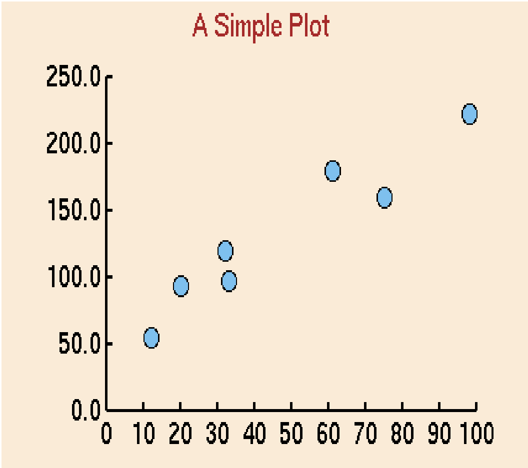
```
activebackground, activeforeground, anchor, background, bd,
bg, bitmap, borderwidth, command,
cursor, disabledforeground, fg, font, foreground,
height, padx, pady, relief, state,
text, textvariable, width
```

methods:

```
def __init__(self, master=None, cnf={}):
def tk_butEnter(self, *dummy):
def tk_butLeave(self, *dummy):
def tk_butDown(self, *dummy):
def tk_butUp(self, *dummy):
def flash(self):
def invoke(self):
```

class Canvas(Widget):

```
    # draws lines, circles, text etc. Objects drawn can respond to
    # events like "enter", "leave", "click", etc.
    options:
    background, bd, bg, borderwidth, closeenough, confine, cursor, height,
    insertbackground, insertborderwidth, insertofftime, insertontime, insertwidth,
    relief, scrollincrement, scrollregion, selectbackground, selectborderwidth,
    selectforeground, width, xscrollcommand, yscrollcommand
    methods:
        def __init__(self, master=None, cnf={}):
        def addtag(self, *args):
        def addtag_above(self, tagOrId):
        def addtag_all(self):
        def addtag_below(self, tagOrId):
        def addtag_closest(self, x, y, halo=None, start=None):
        def addtag_enclosed(self, x1, y1, x2, y2):
        def addtag_overlapping(self, x1, y1, x2, y2):
        def addtag_withtag(self, tagOrId):
        def bbox(self, *args):
        def bind(self, tagOrId, sequence, func, add='')
        def canvasx(self, screenx, gridspacing=None):
        def canvasy(self, screeny, gridspacing=None):
        def coords(self, *args):
        def create_arc(self, *args):
        def create_bitmap(self, *args):
        def create_line(self, *args):
        def create_oval(self, *args):
        def create_polygon(self, *args):
        def create_rectangle(self, *args):
        def create_text(self, *args):
        def create_window(self, *args):
        def dchars(self, *args):
        def delete(self, *args):
        def dtag(self, *args):
        def find(self, *args):
        def find_above(self, tagOrId):
        def find_all(self):
        def find_below(self, tagOrId):
        def find_closest(self, x, y, halo=None, start=None):
        def find_enclosed(self, x1, y1, x2, y2):
        def find_overlapping(self, x1, y1, x2, y2):
        def find_withtag(self, tagOrId):
        def focus(self, *args):
        def gettags(self, *args):
        def icursor(self, *args):
        def index(self, *args):
        def insert(self, *args):
        def itemconfig(self, tagOrId, cnf=None):
        def lower(self, *args):
        def move(self, *args):
        def postscript(self, cnf={}):
        def tkraise(self, *args):
        lift = tkraise
        def scale(self, *args):
        def scan_mark(self, x, y):
        def scan_dragto(self, x, y):
        def select_adjust(self, tagOrId, index):
        def select_clear(self):
```



```
def select_from(self, tagOrId, index):
def select_item(self):
def select_to(self, tagOrId, index):
def type(self, tagOrId):
def xview(self, index):
def yview(self, index):
```

class Checkbutton(Widget):

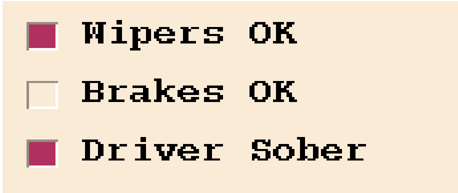
```
# turn a choice on or off. See also radiobuttons.
```

options:

activebackground, activeforeground, anchor, background, bd, bg, bitmap, borderwidth, command, cursor, disabledforeground, fg, font, foreground, height, offvalue, onvalue, padx, pady, relief, selector, state, text, textvariable, variable, width

methods:

```
def __init__(self, master=None, cnf={}):
def deselect(self):
def flash(self):
def invoke(self):
def select(self):
def toggle(self):
```



class Entry(Widget):

options:

background, bd, bg, borderwidth, cursor, exportselection, fg, font, foreground, insertbackground, insertborderwidth, insertofftime, insertontime, insertwidth, relief, scrollcommand, selectbackground, selectborderwidth, selectforeground, state, textvariable, width

methods:

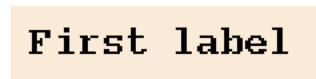
```
def __init__(self, master=None, cnf={}):
def tk_entryBackspace(self):
def tk_entryBackword(self):
def tk_entrySeeCaret(self):
def delete(self, first, last=None):
def get(self):
def icursor(self, index):
def index(self, index):
def insert(self, index, string):
def scan_mark(self, x):
def scan_dragto(self, x):
def select_adjust(self, index):
def select_clear(self):
def select_from(self, index):
def select_to(self, index):
def select_view(self, index):
```

class Frame(Widget):

```
# a container used for holding other widgets
options:
background, bd, bg, borderwidth, cursor, geometry, height, relief, width
methods:
    def __init__(self, master=None, cnf={}):
    def tk_menuBar(self, *args):
```

class Label(Widget):

```
options:
    anchor, background, bd, bg, bitmap,
    borderwidth, cursor, fg, font,
    foreground, height, padx, pady, relief,
    text, textvariable, width
methods:
    def __init__(self, master=None, cnf={}):
```

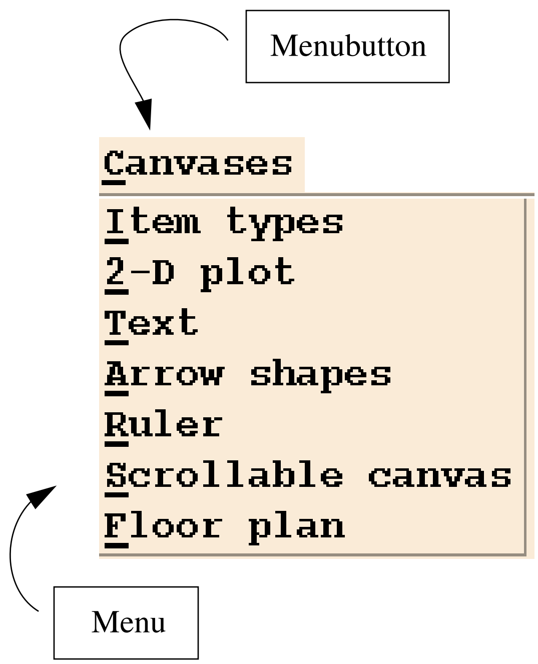


class Listbox(Widget):

```
options:
    background, bd, bg, borderwidth, cursor,
    exportselection, fg, font, foreground, geometry,
    relief, selectbackground, selectborderwidth, selectforeground, setgrid,
    xscrollcommand, yscrollcommand
methods:
    def __init__(self, master=None, cnf={}):
    def tk_listboxSingleSelect(self):
    def curselection(self):
    def delete(self, first, last=None):
    def get(self, index):
    def insert(self, index, *elements):
    def nearest(self, y):
    def scan_mark(self, x, y):
    def scan_dragto(self, x, y):
    def select_adjust(self, index):
    def select_clear(self):
    def select_from(self, index):
    def select_to(self, index):
    def size(self):
    def xview(self, index):
    def yview(self, index):
```

class Menu(Widget):

```
options:
    activebackground, activeborderwidth, activeforeground, background, bd,
    bg, borderwidth, cursor, disabledforeground, fg,
    font, foreground, postcommand, selector
methods:
    def __init__(self, master=None, cnf={}):
    def tk_bindForTraversal(self):
    def tk_mbPost(self):
    def tk_mbUnpost(self):
    def tk_traverseToMenu(self, char):
    def tk_traverseWithinMenu(self, char):
    def tk_getMenuButtons(self):
    def tk_nextMenu(self, count):
    def tk_nextMenuEntry(self, count):
    def tk_invokeMenu(self):
    def tk_firstMenu(self):
    def tk_mbButtonDown(self):
    def activate(self, index):
    def add(self, itemType, cnf={}):
    def add_cascade(self, cnf={}):
    def add_checkbutton(self, cnf={}):
    def add_command(self, cnf={}):
    def add_radiobutton(self, cnf={}):
    def add_separator(self, cnf={}):
    def delete(self, index1, index2=None):
    def entryconfig(self, index, cnf={}):
    def index(self, index):
    def invoke(self, index):
    def post(self, x, y):
    def unpost(self):
    def yposition(self, index):
```



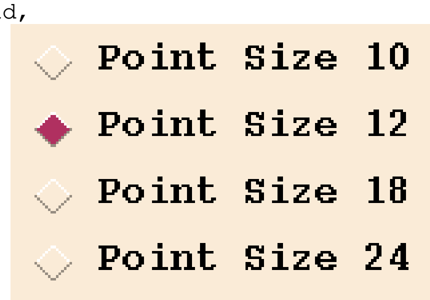
class Menubutton(Widget):

```
options:
    activebackground, activeforeground, anchor, background, bd,
    bg, bitmap, borderwidth, cursor,
    disabledforeground, fg, font, foreground, height,
    menu, padx, pady, relief, state,
    text, textvariable, underline, width
methods:
    def __init__(self, master=None, cnf={}):
```

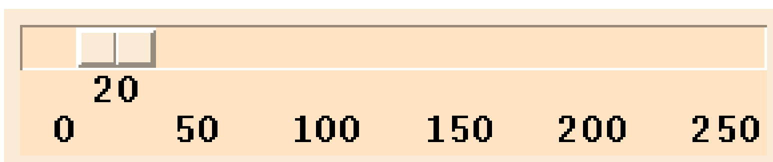
class Message(Widget):

```
# a multiline label
options:
    anchor, aspect, background, bd, bg,
    borderwidth, cursor, fg, font, foreground,
    justify, padx, pady, relief, text,
    textvariable, width
methods:
    def __init__(self, master=None, cnf={}):
```

```
class Radiobutton(Widget):
    # select one from the set of mutually exclusive choices
    options:
        activebackground, activeforeground, anchor, background, bd,
        bg, bitmap, borderwidth, command,
        cursor, disabledforeground, fg, font, foreground,
        height, padx, pady, relief, selector,
        state, text, textvariable, value, variable,
        width
    methods:
        def __init__(self, master=None, cnf={}):
        def deselect(self):
        def flash(self):
        def invoke(self):
        def select(self):
```




```
class Scale(Widget):
    options:
        activeforeground, background, bd, bg, borderwidth,
        command, cursor, fg, font, foreground,
        from, label, length, orient, relief,
        showvalue, sliderforeground, sliderlength, state, tickinterval,
        to, width
    methods:
        def __init__(self, master=None, cnf={}):
        def get(self):
        def set(self, value):
```




```
class Scrollbar(Widget):
    options:
        activeforeground, background, bd, bg, borderwidth,
        command, cursor, fg, foreground, orient,
        relief, repeatdelay, repeatinterval, width
    methods:
        def __init__(self, master=None, cnf={}):
        def get(self):
        def set(self, totalUnits, windowUnits, firstUnit, lastUnit):
```

```
class Text(Widget):
    # a text editor
    options:
        background, bd, bg, borderwidth, cursor,
        exportselection, fg, font, foreground, height,
        insertbackground, insertborderwidth, insertofftime, insertontime, insertwidth,
        padx, pady, relief, selectbackground, selectborderwidth,
        selectforeground, setgrid, state, width, wrap,
        yscrollcommand
    methods:
        def __init__(self, master=None, cnf={}):
        def tk_textSelectTo(self, index):
        def tk_textBackspace(self):
        def tk_textIndexCloser(self, a, b, c):
        def tk_textResetAnchor(self, index):
        def compare(self, index1, op, index2):
        def debug(self, boolean=None):
        def delete(self, index1, index2=None):
        def get(self, index1, index2=None):
        def index(self, index):
        def insert(self, index, chars):
        def mark_names(self):
        def mark_set(self, markName, index):
        def mark_unset(self, markNames):
        def scan_mark(self, y):
        def scan_dragto(self, y):
        def tag_add(self, tagName, index1, index2=None):
        def tag_bind(self, tagName, sequence, func, add=''):
        def tag_config(self, tagName, cnf={}):
        def tag_delete(self, *tagNames):
        def tag_lower(self, tagName, belowThis=None):
        def tag_names(self, index=None):
        def tag_nextrange(self, tagName, index1, index2=None):
        def tag_raise(self, tagName, aboveThis=None):
        def tag_ranges(self, tagName):
        def tag_remove(self, tagName, index1, index2=None):
        def yview(self, what):
        def yview_pickplace(self, what):
```

Text widgets like this one allow you to display information in a variety of styles. Display styles are controlled using a mechanism called *tags*. Tags are just textual names that you can apply to one or more ranges of characters within a text widget. You can configure tags with various display styles. If you do this, then the tagged characters will be displayed with the styles you chose. The available display styles are:

1. Font. You can choose any X font, large or small.

2. Color. You can change either the background or foreground color, or both.

3. Stippling. You can cause either the background or foreground information to be drawn with a stipple fill instead of a solid fill.

4. Underlining. You can underline ranges of text.

5. 3-D effects. You can arrange for the background to be drawn with a border that makes characters appear either raised or sunken.