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 *
```

Application Windows

Base Windows

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

```
from Tkinter import *

root = Tk()

# create window contents as children to root...

root.mainloop()
```

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

```
from Tkinter import *

root = Tk()

# create root window contents...

top = Toplevel()

# create top window contents...

root.mainloop()
```

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

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()
```

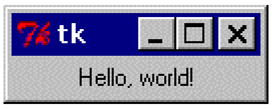
Running the Example

To run the program, run the script as usual:

```
$ python hello1.py
```

The following window appears.

Figure 1. Running the program



Hello, world!

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.

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 *grid* method.
Pack	The pack geometry manager lets you create a layout by "packing" the widgets into a parent widget, by treating them as rectangular blocks placed in a frame. To use this geometry manager for a widget, use the *pack* method on that widget to set things up.
Place	The place geometry manager lets you explicitly place a widget in a given position. To use this geometry manager, use the *place* method.

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.

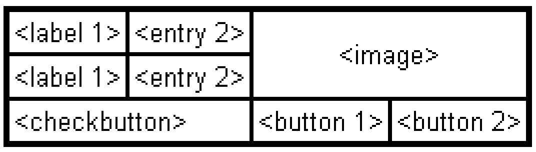
When to use the Grid Manager

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.



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)
```

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.

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 to much work to get things working as they should. Use the *pack* or *grid* managers for such purposes.

However, *place* has its uses in more specialized cases. Most importantly, it can be used by compound widget containers to implement various custom geometry managers. Another use is to position control buttons in dialogs.

Patterns

Let's look at some usage patterns. The following command centers a widget in its parent:

```
w.place(relx=0.5, rely=0.5, anchor=CENTER)
```

Here's another variant. It packs a *Label* widget in a frame widget, and then places a *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")
```

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 *wraplength* option to make the label wrap text by itself. When wrapping text, you might wish to use the *anchor* and *justify* options to make things look exactly as you wish. An example:

```
w = Label(master, text=longtext, anchor=W, justify=LEFT)
```

You can associate a 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.

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 *wraplength* option to make the button wrap text by itself. When wrapping text, use the *anchor*, *justify*, and possibly*padx* options to make things look exactly as you wish. An example:

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

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

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

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.

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:

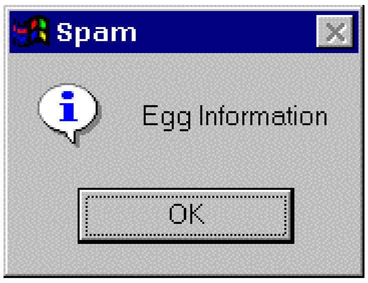
tkMessageBox.function(title, message [, options]). The *title* argument is shown in the window title, and the message in the dialog body. You can use newline characters ("\n") in the message to make it occupy multiple lines. The options can be used to modify the look; they are explained later in this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

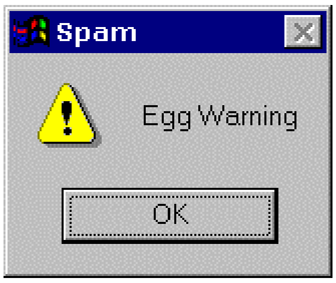
The first group of standard dialogs is used to present information. You provide the title and the message, the function displays these using an appropriate icon, and returns when the user has pressed OK. The return value should be ignored.

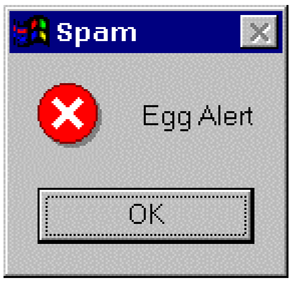
Here's an example:

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

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

Figure 2. askquestion dialog

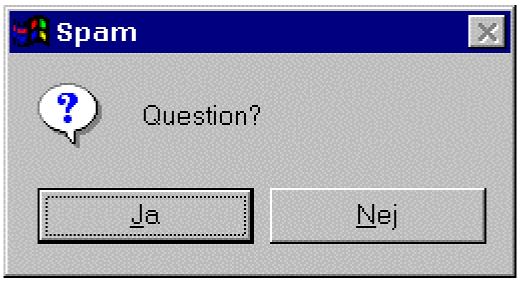
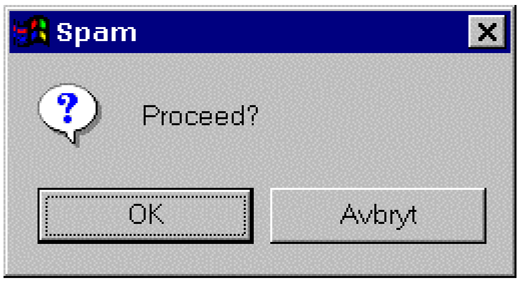
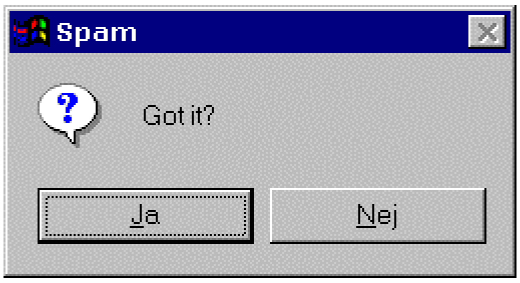
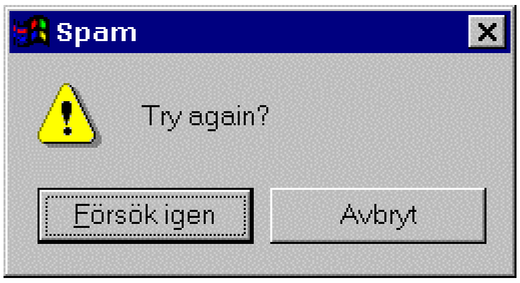


Figure 3. askokcancel, askyesno, askretryignore dialogs







[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

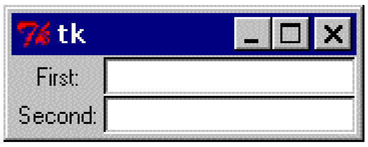
Option	Type	Description
default	constant	Which button to make default: *ABORT*, *RETRY*, *IGNORE*, *OK*, *CANCEL*, *YES*, or *NO* (the constants are defined in the *tkMessageBox* module).
icon	constant	Which icon to display: *ERROR*, *INFO*, *QUESTION*, or *WARNING*
message	string	The message to display (the second argument to the convenience functions). May contain newlines.
parent	widget	Which window to place the message box on top of. When the message box is closed, the focus is returned to the parent window.
title	string	Message box title (the first argument to the convenience functions).
type	constant	Message box type; that is, which buttons to display: *ABORTRETRYIGNORE*, *OK*, *OKCANCEL*, *RETRYCANCEL*, *YESNO*, or *YESNOCANCEL*.

```
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

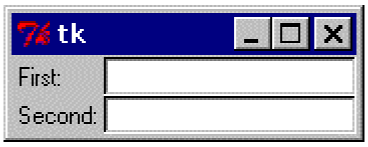


Empty rows and columns are ignored. The result would have been the same if you had placed the widgets in row 10 and 20 instead.

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)

checkbutton.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)
```

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:

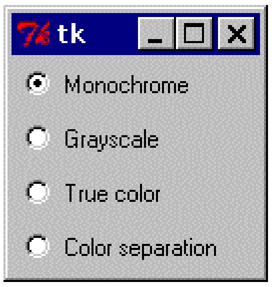
```
v = IntVar()
Radiobutton(master, text="One", variable=v, value=1).pack(anchor=W)
Radiobutton(master, text="Two", variable=v, value=2).pack(anchor=W)
```

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:

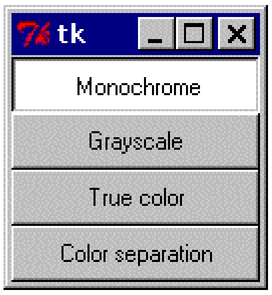
```
MODES = [
    ("Monochrome", "1"),
    ("Grayscale", "L"),
    ("True color", "RGB"),
    ("Color separation", "CMYK"),
]

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.

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 specifed 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:

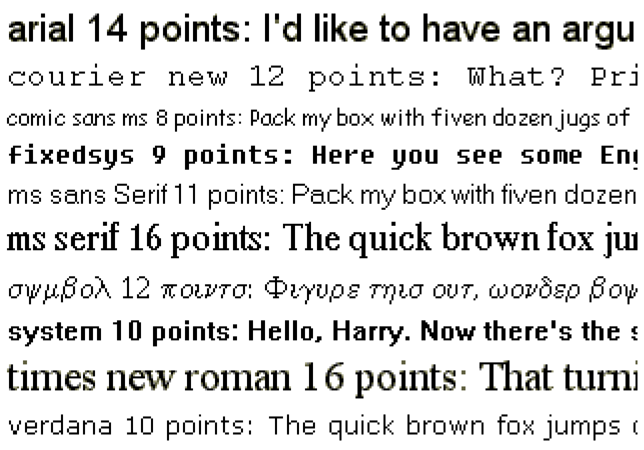
```
("Times", 10, "bold")
("Helvetica", 10, "bold italic")
("Symbol", 8)
```

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 En
ms sans Serif 11 points: Pack my box with fiven dozen
ms serif 16 points: The quick brown fox ju
σψμβολ 12 ποιντσ: Φιγυρε τηισ ουτ, ωονδερ βοψ
system 10 points: Hello, Harry. Now there's the
times new roman 16 points: That turni
verdana 10 points: The quick brown fox jumps

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