

# tkinter — Python interface to Tcl/Tk

**Source code:** [Lib/tkinter/\\_\\_init\\_\\_.py](#)

The `tkinter` package (“Tk interface”) is the standard Python interface to the Tk GUI toolkit. Both Tk and `tkinter` are available on most Unix platforms, as well as on Windows systems. (Tk itself is not part of Python; it is maintained at ActiveState.)

Running `python -m tkinter` from the command line should open a window demonstrating a simple Tk interface, letting you know that `tkinter` is properly installed on your system, and also showing what version of Tcl/Tk is installed, so you can read the Tcl/Tk documentation specific to that version.

**See also:** Tkinter documentation:

## Python Tkinter Resources

The Python Tkinter Topic Guide provides a great deal of information on using Tk from Python and links to other sources of information on Tk.

## TKDocs

Extensive tutorial plus friendlier widget pages for some of the widgets.

## Tkinter 8.5 reference: a GUI for Python

On-line reference material.

## Tkinter docs from effbot

Online reference for tkinter supported by effbot.org.

## Programming Python

Book by Mark Lutz, has excellent coverage of Tkinter.

## Modern Tkinter for Busy Python Developers

Book by Mark Roseman about building attractive and modern graphical user interfaces with Python and Tkinter.

## Python and Tkinter Programming

Book by John Grayson (ISBN 1-884777-81-3).

Tcl/Tk documentation:

## Tk commands

Most commands are available as `tkinter` or `tkinter.ttk` classes. Change ‘8.6’ to match the version of your Tcl/Tk installation.

## Tcl/Tk recent man pages

Recent Tcl/Tk manuals on [www.tcl.tk](http://www.tcl.tk).

## ActiveState Tcl Home Page

The Tk/Tcl development is largely taking place at ActiveState.

## Tcl and the Tk Toolkit

Book by John Ousterhout, the inventor of Tcl.

## Practical Programming in Tcl and Tk

Brent Welch's encyclopedic book.

# Tkinter Modules

Most of the time, `tkinter` is all you really need, but a number of additional modules are available as well. The Tk interface is located in a binary module named `_tkinter`. This module contains the low-level interface to Tk, and should never be used directly by application programmers. It is usually a shared library (or DLL), but might in some cases be statically linked with the Python interpreter.

In addition to the Tk interface module, `tkinter` includes a number of Python modules, `tkinter.constants` being one of the most important. Importing `tkinter` will automatically import `tkinter.constants`, so, usually, to use Tkinter all you need is a simple import statement:

```
import tkinter
```

Or, more often:

```
from tkinter import *
```

`class tkinter.Tk(screenName=None, baseName=None, className='Tk', useTk=1)`

The `Tk` class is instantiated without arguments. This creates a toplevel widget of Tk which usually is the main window of an application. Each instance has its own associated Tcl interpreter.

`tkinter.Tcl(screenName=None, baseName=None, className='Tk', useTk=0)`

The `Tcl()` function is a factory function which creates an object much like that created by the `Tk` class, except that it does not initialize the Tk subsystem. This is most often useful when driving the Tcl interpreter in an environment where one doesn't want to create extraneous toplevel windows, or where one cannot (such as Unix/Linux systems without an X server). An object created by the `Tcl()` object can have a Toplevel window created (and the Tk subsystem initialized) by calling its `loadtk()` method.

Other modules that provide Tk support include:

`tkinter.scrolledtext`

Text widget with a vertical scroll bar built in.

`tkinter.colorchooser`

Dialog to let the user choose a color.

`tkinter.commondialog`

Base class for the dialogs defined in the other modules listed here.

`tkinter.filedialog`

Common dialogs to allow the user to specify a file to open or save.

`tkinter.font`

Utilities to help work with fonts.

`tkinter.messagebox`

Access to standard Tk dialog boxes.

`tkinter.simpledialog`

Basic dialogs and convenience functions.

`tkinter.dnd`

Drag-and-drop support for `tkinter`. This is experimental and should become deprecated when it is replaced with the Tk DND.

`turtle`

Turtle graphics in a Tk window.

## Tkinter Life Preserver

This section is not designed to be an exhaustive tutorial on either Tk or Tkinter. Rather, it is intended as a stop gap, providing some introductory orientation on the system.

Credits:

- Tk was written by John Ousterhout while at Berkeley.
- Tkinter was written by Steen Lumholt and Guido van Rossum.
- This Life Preserver was written by Matt Conway at the University of Virginia.
- The HTML rendering, and some liberal editing, was produced from a FrameMaker version by Ken Manheimer.
- Fredrik Lundh elaborated and revised the class interface descriptions, to get them current with Tk 4.2.
- Mike Clarkson converted the documentation to LaTeX, and compiled the User Interface chapter of the reference manual.

## How To Use This Section

This section is designed in two parts: the first half (roughly) covers background material, while the second half 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. 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:

- The authors 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.

- 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/__init__.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

```
import tkinter as tk

class Application(tk.Frame):
    def __init__(self, master=None):
        super().__init__(master)
        self.master = master
        self.pack()
        self.create_widgets()

    def create_widgets(self):
        self.hi_there = tk.Button(self)
        self.hi_there["text"] = "Hello World\n(click me)"
        self.hi_there["command"] = self.say_hi
        self.hi_there.pack(side="top")

        self.quit = tk.Button(self, text="QUIT", fg="red",
                               command=self.master.destroy)
        self.quit.pack(side="bottom")

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

root = tk.Tk()
app = Application(master=root)
app.mainloop()
```

## A (Very) Quick Look at Tcl/Tk

The class hierarchy looks complicated, but in actual practice, application programmers 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').

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. (See

section [Mapping Basic Tk into Tkinter](#) for the [tkinter](#) equivalents of what's below.)

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
```

*classCommand*

denotes which kind of widget to make (a button, a label, a menu...)

*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.

*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 preceded by a '-', like 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      new          options
command   widget      (-opt val -opt val ...)
```

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.

## Mapping Basic Tk into Tkinter

Class commands in Tk correspond to class constructors in 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.

```
button .panel.fred    =====> fred = Button(panel)
```

The configuration options in Tk are given in lists of hyphenated tags followed by values. In Tkinter, options are specified as keyword-arguments in the instance constructor, and keyword-args for configure calls or as instance indices, in dictionary style, for established instances. See section [Setting Options](#) on setting options.

```
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 `tkinter/__init__.py`.

```
.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. See the `tkinter.tix` module documentation for additional information on the Form geometry manager.

```
pack .fred -side left   =====> fred.pack(side="left")
```

## How Tk and Tkinter are Related

From the top down:

Your App Here (Python)

A Python application makes a `tkinter` call.

tkinter (Python Package)

This call (say, for example, creating a button widget), is implemented in the `tkinter` package, 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.

`_tkinter` (C)

These commands and their arguments will be passed to a C function in the `_tkinter` - note the underscore - extension module.

## Tk Widgets (C and Tcl)

This C function is able 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` package is imported. (The user never sees this stage).

## Tk (C)

The Tk part of the Tk Widgets implement the final mapping to ...

## Xlib (C)

the Xlib library to draw graphics on the screen.

# Handy Reference

## 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, using keyword arguments

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

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

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

Use the `config()` method to update multiple attrs subsequent to object creation

```
fred.config(fg="red", bg="blue")
```

For a complete explanation of a given option and its behavior, see the Tk 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 idiosyncratic to that particular widget. The Standard Options are documented on the *options(3)* man page.

No distinction between standard and widget-specific options 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 without arguments, 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 as a string (for example, `'relief'`) and whose values are 5-tuples.

Some options, like `bg` are synonyms for common options with long 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 and the “real” option (such as `('bg', 'background')`).

Index	Meaning	Example
0	option name	'relief'
1	option name for database lookup	'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.

## The Packer

The packer is one of Tk’s geometry-management mechanisms. Geometry managers are used to specify the relative positioning of the positioning of widgets within their container - their mutual *master*. In contrast to the more cumbersome *placer* (which is used less commonly, and we do not cover here), the packer takes qualitative relationship specification - *above*, *to the left of*, *filling*, etc - and works everything out to determine the exact placement coordinates for you.

The size of any *master* widget 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. Additionally, the arrangement is dynamically adjusted to accommodate incremental changes to the configuration, once it is packed.

Note that widgets do not appear until they have had their geometry specified with a geometry manager. It’s a common early mistake to leave out the geometry specification, and then be surprised when the widget is created but nothing appears. A widget will appear only after it has had, for example, the packer’s `pack()` method applied to it.

The `pack()` method can be called with keyword-option/value pairs that control where the widget is to appear within its container, and how it is to behave when the main application window is resized. Here are some examples:

```
fred.pack()                # defaults to side = "top"  
fred.pack(side="left")  
fred.pack(expand=1)
```



## Packer Options

For more extensive information on the packer and the options that it can take, see the man pages and page 183 of John 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 - designating internal padding on each side of the slave widget.

padx and pady

A distance - designating external padding on each side of the slave widget.

side

Legal values are: 'left', 'right', 'top', 'bottom'.

## Coupling Widget Variables

The current-value setting of some widgets (like text entry widgets) can be connected directly to application variables by using 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 an 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 `tkinter`.

There are many useful subclasses of `Variable` already defined: `StringVar`, `IntVar`, `DoubleVar`, and `BooleanVar`. To read the current value of such a variable, 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):
        super().__init__(master)
        self.pack()

        self.entrythingy = Entry()
        self.entrythingy.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, 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 as tk

class App(tk.Frame):
    def __init__(self, master=None):
        super().__init__(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()

```

## Tk Option Data Types

## anchor

Legal values are points of the compass: "n", "ne", "e", "se", "s", "sw", "w", "nw", and also "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".

## boolean

You can pass integers 0 or 1 or the strings "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: "#RRRGGBBB", 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\_ prefix. 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 centimetres, i for inches, m for millimetres, p for printer's points. For example, 3.5 inches is expressed as "3.5i".

## font

Tk uses a list font name format, such as {courier 10 bold}. Font sizes with positive numbers are measured in points; sizes with negative numbers are measured in pixels.

## geometry

This is a string of the form widthxheight, where width and height are measured in pixels for most widgets (in characters for widgets displaying text). For 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", and "ridge".

#### scrollcommand

This is almost always the `set()` method of some scrollbar widget, but can be any widget method that takes a single argument.

#### 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 the target kind of event. (See the `bind` man page and page 201 of John Ousterhout's book for details).

##### func

is a Python function, taking one argument, to be invoked when the event occurs. An `Event` instance will be passed as the argument. (Functions deployed this way are commonly known as *callbacks*.)

##### add

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

For example:

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

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

Notice how the `widget` field of the event is being accessed in the `turn_red()` 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	Tk	Tkinter Event Field
----	---------------------	----	---------------------

Tk	Tkinter Event Field	Tk	Tkinter Event Field
%f	focus	%A	char
%h	height	%E	send_event
%k	keycode	%K	keysym
%s	state	%N	keysym_num
%t	time	%T	type
%w	width	%W	widget
%x	x	%X	x_root
%y	y	%Y	y_root

## The index Parameter

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:

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;
- the string "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;
- the string "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 means that matches for menu items labelled last, active, or none may be interpreted as the above literals, instead.

## Images

Images of different formats can be created through the corresponding subclass of `tkinter.Image`:

- `BitmapImage` for images in XBM format.
- `PhotoImage` for images in PGM, PPM, GIF and PNG formats. The latter is supported starting with Tk 8.6.

Either type of image is created through either the `file` or the `data` option (other options are available as well).

The image object can then be used wherever an image option is supported by some widget (e.g. labels, buttons, menus). In these cases, Tk will not keep a reference to the image. When the last Python reference to the image object is deleted, the image data is deleted as well, and Tk will display an empty box wherever the image was used.

**See also:** The [Pillow](#) package adds support for formats such as BMP, JPEG, TIFF, and WebP, among others.

## File Handlers

Tk allows you to register and unregister a callback function which will be called from the Tk mainloop when I/O is possible on a file descriptor. Only one handler may be registered per file descriptor. Example code:

```
import tkinter
widget = tkinter.Tk()
mask = tkinter.READABLE | tkinter.WRITABLE
widget.tk.createfilehandler(file, mask, callback)
...
widget.tk.deletefilehandler(file)
```

This feature is not available on Windows.

Since you don't know how many bytes are available for reading, you may not want to use the [BufferedIOBase](#) or [TextIOBase](#) `read()` or `readline()` methods, since these will insist on reading a predefined number of bytes. For sockets, the [recv\(\)](#) or [recvfrom\(\)](#) methods will work fine; for other files, use raw reads or `os.read(file.fileno(), maxbytecount)`.

`Widget.tk.createfilehandler(file, mask, func)`

Registers the file handler callback function *func*. The *file* argument may either be an object with a `fileno()` method (such as a file or socket object), or an integer file descriptor. The *mask* argument is an ORed combination of any of the three constants below. The callback is called as follows:

```
callback(file, mask)
```

`Widget.tk.deletefilehandler(file)`

Unregisters a file handler.

`tkinter.READABLE`

`tkinter.WRITABLE`

`tkinter.EXCEPTION`

Constants used in the *mask* arguments.