**Python - GUI Programming**

[Previous](https://www.tutorialspoint.com/python/python_xml_processing.htm)

[Quiz](https://www.tutorialspoint.com/python/quiz_on_python_gui_programming.htm)

[Next](https://www.tutorialspoint.com/python/python_command_line_arguments.htm)

Python provides various options for developing graphical user interfaces (GUIs). The most important features are listed below.

* **Tkinter** − Tkinter is the Python interface to the Tk GUI toolkit shipped with Python. We would look at this option in this chapter.
* **wxPython** − This is an open-source Python interface for wxWidgets GUI toolkit. You can find a complete tutorial on WxPython [here](https://www.tutorialspoint.com/wxpython/index.htm).
* **PyQt** − This is also a Python interface for a popular cross-platform Qt GUI library. TutorialsPoint has a very good tutorial on PyQt5 [here](https://www.tutorialspoint.com/pyqt/index.htm).
* **PyGTK** − PyGTK is a set of wrappers written in Python and C for GTK + GUI library. The complete PyGTK tutorial is available [here](https://www.tutorialspoint.com/pygtk/index.htm).
* **PySimpleGUI** − PySimpleGui is an open source, cross-platform GUI library for Python. It aims to provide a uniform API for creating desktop GUIs based on Python's Tkinter, PySide and WxPython toolkits. For a detaile PySimpleGUI tutorial, click [here](https://www.tutorialspoint.com/pysimplegui/index.htm).
* **Pygame** − Pygame is a popular Python library used for developing video games. It is free, open source and cross-platform wrapper around Simple DirectMedia Library (SDL). For a comprehensive tutorial on Pygame, [visit](https://www.tutorialspoint.com/pygame/index.htm) this link.
* **Jython** − Jython is a Python port for Java, which gives Python scripts seamless access to the Java class libraries on the local machinehttp: [//www.jython.org](http://www.jython.org/).

There are many other interfaces available, which you can find them on the net.

Tkinter Programming

Tkinter is the standard GUI library for Python. Python when combined with Tkinter provides a fast and easy way to create GUI applications. Tkinter provides a powerful object-oriented interface to the Tk GUI toolkit.

The tkinter package includes following modules −

* **Tkinter** − Main Tkinter module.
* **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.scrolledtext** − Text widget with a vertical scroll bar built in.
* **tkinter.simpledialog** − Basic dialogs and convenience functions.
* **tkinter.ttk** − Themed widget set introduced in Tk 8.5, providing modern alternatives for many of the classic widgets in the main tkinter module.

Creating a GUI application using Tkinter is an easy task. All you need to do is perform the following steps.

* Import the Tkinter module.
* Create the GUI application main window.
* Add one or more of the above-mentioned widgets to the GUI application.
* Enter the main event loop to take action against each event triggered by the user.

Example

# note that module name has changed from Tkinter in Python 2

# to tkinter in Python 3

import tkinter

top = tkinter.Tk()

# Code to add widgets will go here...

top.mainloop()

This would create a following window −



When the program becomes more complex, using an object-oriented programming approach makes the code more organized.

import tkinter as tk

class App(tk.Tk):

def \_\_init\_\_(self):

super().\_\_init\_\_()

app = App()

app.mainloop()

Advertisement

-[PS](https://playstream.media/)

ReplaySkip backward 5 secondsSkip forward 5 seconds

Mute

Fullscreen

Tkinter Widgets

Tkinter provides various controls, such as buttons, labels and text boxes used in a GUI application. These controls are commonly called widgets.

There are currently 15 types of widgets in Tkinter. We present these widgets as well as a brief description in the following table −

|  |  |
| --- | --- |
| **Sr.No.** | **Operator & Description** |
| 1 | [**Button**](https://www.tutorialspoint.com/python/tk_button.htm)  The Button widget is used to display the buttons in your application. |
| 2 | [**Canvas**](https://www.tutorialspoint.com/python/tk_canvas.htm)  The Canvas widget is used to draw shapes, such as lines, ovals, polygons and rectangles, in your application. |
| 3 | [**Checkbutton**](https://www.tutorialspoint.com/python/tk_checkbutton.htm)  The Checkbutton widget is used to display a number of options as checkboxes. The user can select multiple options at a time. |
| 4 | [**Entry**](https://www.tutorialspoint.com/python/tk_entry.htm)  The Entry widget is used to display a single-line text field for accepting values from a user. |
| 5 | [**Frame**](https://www.tutorialspoint.com/python/tk_frame.htm)  The Frame widget is used as a container widget to organize other widgets. |
| 6 | [**Label**](https://www.tutorialspoint.com/python/tk_label.htm)  The Label widget is used to provide a single-line caption for other widgets. It can also contain images. |
| 7 | [**Listbox**](https://www.tutorialspoint.com/python/tk_listbox.htm)  The Listbox widget is used to provide a list of options to a user. |
| 8 | [**Menubutton**](https://www.tutorialspoint.com/python/tk_menubutton.htm)  The Menubutton widget is used to display menus in your application. |
| 9 | [**Menu**](https://www.tutorialspoint.com/python/tk_menu.htm)  The Menu widget is used to provide various commands to a user. These commands are contained inside Menubutton. |
| 10 | [**Message**](https://www.tutorialspoint.com/python/tk_message.htm)  The Message widget is used to display multiline text fields for accepting values from a user. |
| 11 | [**Radiobutton**](https://www.tutorialspoint.com/python/tk_radiobutton.htm)  The Radiobutton widget is used to display a number of options as radio buttons. The user can select only one option at a time. |
| 12 | [**Scale**](https://www.tutorialspoint.com/python/tk_scale.htm)  The Scale widget is used to provide a slider widget. |
| 13 | [**Scrollbar**](https://www.tutorialspoint.com/python/tk_scrollbar.htm)  The Scrollbar widget is used to add scrolling capability to various widgets, such as list boxes. |
| 14 | [**Text**](https://www.tutorialspoint.com/python/tk_text.htm)  The Text widget is used to display text in multiple lines. |
| 15 | [**Toplevel**](https://www.tutorialspoint.com/python/tk_toplevel.htm)  The Toplevel widget is used to provide a separate window container. |
| 16 | [**Spinbox**](https://www.tutorialspoint.com/python/tk_spinbox.htm)  The Spinbox widget is a variant of the standard Tkinter Entry widget, which can be used to select from a fixed number of values. |
| 17 | [**PanedWindow**](https://www.tutorialspoint.com/python/tk_panedwindow.htm)  A PanedWindow is a container widget that may contain any number of panes, arranged horizontally or vertically. |
| 18 | [**LabelFrame**](https://www.tutorialspoint.com/python/tk_labelframe.htm)  A labelframe is a simple container widget. Its primary purpose is to act as a spacer or container for complex window layouts. |
| 19 | [**tkMessageBox**](https://www.tutorialspoint.com/python/tk_messagebox.htm)  This module is used to display message boxes in your applications. |

Let us study these widgets in detail.

Standard Attributes

Let us look at how some of the common attributes, such as sizes, colors and fonts are specified.

* [Dimensions](https://www.tutorialspoint.com/python/tk_dimensions.htm)
* [Colors](https://www.tutorialspoint.com/python/tk_colors.htm)
* [Fonts](https://www.tutorialspoint.com/python/tk_fonts.htm)
* [Anchors](https://www.tutorialspoint.com/python/tk_anchors.htm)
* [Relief styles](https://www.tutorialspoint.com/python/tk_relief.htm)
* [Bitmaps](https://www.tutorialspoint.com/python/tk_bitmaps.htm)
* [Cursors](https://www.tutorialspoint.com/python/tk_cursors.htm)

Let us study them briefly −

Geometry Management

All Tkinter widgets have access to the specific geometry management methods, which have the purpose of organizing widgets throughout the parent widget area. Tkinter exposes the following geometry manager classes: pack, grid, and place.

* [**The pack() Method**](https://www.tutorialspoint.com/python/tk_pack.htm) − This geometry manager organizes widgets in blocks before placing them in the parent widget.
* [**The grid() Method**](https://www.tutorialspoint.com/python/tk_grid.htm) − This geometry manager organizes widgets in a table-like structure in the parent widget.
* [**The place() Method**](https://www.tutorialspoint.com/python/tk_place.htm) − This geometry manager organizes widgets by placing them in a specific position in the parent widget.

Let us study the geometry management methods briefly −

SimpleDialog

The simpledialog module in tkinter package includes a dialog class and convenience functions for accepting user input through a modal dialog. It consists of a label, an entry widget and two buttons Ok and Cancel. These functions are −

* **askfloat(title, prompt, \*\*kw)** − Accepts a floating point number.
* **askinteger(title, prompt, \*\*kw)** − Accepts an integer input.
* **askstring(title, prompt, \*\*kw)** − Accepts a text input from the user.

The above three functions provide dialogs that prompt the user to enter a value of the desired type. If Ok is pressed, the input is returned, if Cancel is pressed, None is returned.

askinteger

from tkinter.simpledialog import askinteger

from tkinter import \*

from tkinter import messagebox

top = Tk()

top.geometry("100x100")

def show():

num = askinteger("Input", "Input an Integer")

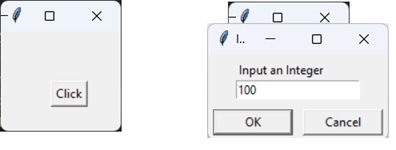
print(num)

B = Button(top, text ="Click", command = show)

B.place(x=50,y=50)

top.mainloop()

It will produce the following **output** −



askfloat

from tkinter.simpledialog import askfloat

from tkinter import \*

top = Tk()

top.geometry("100x100")

def show():

num = askfloat("Input", "Input a floating point number")

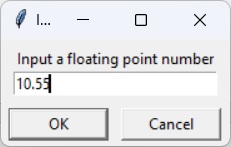
print(num)

B = Button(top, text ="Click", command = show)

B.place(x=50,y=50)

top.mainloop()

It will produce the following **output** −



askstring

from tkinter.simpledialog import askstring

from tkinter import \*

top = Tk()

top.geometry("100x100")

def show():

name = askstring("Input", "Enter you name")

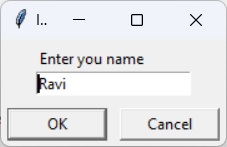
print(name)

B = Button(top, text ="Click", command = show)

B.place(x=50,y=50)

top.mainloop()

It will produce the following **output** −



The FileDialog Module

The filedialog module in Tkinter package includes a FileDialog class. It also defines convenience functions that enable the user to perform open file, save file, and open directory activities.

* filedialog.asksaveasfilename()
* filedialog.asksaveasfile()
* filedialog.askopenfilename()
* filedialog.askopenfile()
* filedialog.askdirectory()
* filedialog.askopenfilenames()
* filedialog.askopenfiles()

askopenfile

This function lets the user choose a desired file from the filesystem. The file dialog window has Open and Cancel buttons. The file name along with its path is returned when Ok is pressed, None if Cancel is pressed.

from tkinter.filedialog import askopenfile

from tkinter import \*

top = Tk()

top.geometry("100x100")

def show():

filename = askopenfile()

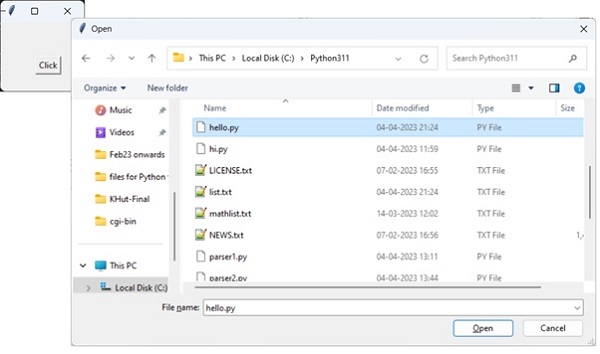
print(filename)

B = Button(top, text ="Click", command = show)

B.place(x=50,y=50)

top.mainloop()

It will produce the following **output** −



ColorChooser

The colorchooser module included in tkinter package has the feature of letting the user choose a desired color object through the color dialog. The askcolor() function presents with the color dialog with predefined color swatches and facility to choose custome color by setting RGB values. The dialog returns a tuple of RGB values of chosen color as well as its hex value.

from tkinter.colorchooser import askcolor

from tkinter import \*

top = Tk()

top.geometry("100x100")

def show():

color = askcolor()

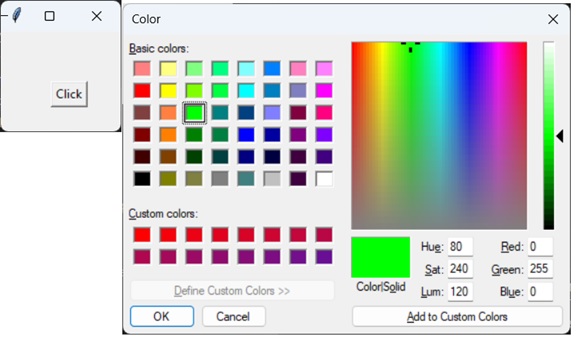
print(color)

B = Button(top, text ="Click", command = show)

B.place(x=50,y=50)

top.mainloop()

It will produce the following **output** −



((0, 255, 0), '#00ff00')

ttk module

The term ttk stands from Tk Themed widgets. The ttk module was introduced with Tk 8.5 onwards. It provides additional benefits including anti-aliased font rendering under X11 and window transparency. It provides theming and styling support for Tkinter.

The ttk module comes bundled with 18 widgets, out of which 12 are already present in Tkinter. Importing ttk over-writes these widgets with new ones which are designed to have a better and more modern look across all platforms.

The 6 new widgets in ttk are, the Combobox, Separator, Sizegrip, Treeview, Notebook and ProgressBar.

To override the basic Tk widgets, the import should follow the Tk import −

from tkinter import \*

from tkinter.ttk import \*

The original Tk widgets are automatically replaced by tkinter.ttk widgets. They are Button, Checkbutton, Entry, Frame, Label, LabelFrame, Menubutton, PanedWindow, Radiobutton, Scale and Scrollbar.

New widgets which gives a better look and feel across platforms; however, the replacement widgets are not completely compatible. The main difference is that widget options such as "fg", "bg" and others related to widget styling are no longer present in Ttk widgets. Instead, use the ttk.Style class for improved styling effects.

The new widgets in ttk module are −

* **Notebook** − This widget manages a collection of "tabs" between which you can swap, changing the currently displayed window.
* **ProgressBar** − This widget is used to show progress or the loading process through the use of animations.
* **Separator** − Used to separate different widgets using a separator line.
* **Treeview** − This widget is used to group together items in a tree-like hierarchy. Each item has a textual label, an optional image, and an optional list of data values.
* **ComboBox** − Used to create a dropdown list of options from which the user can select one.
* **Sizegrip** − Creates a little handle near the bottom-right of the screen, which can be used to resize the window.

Combobox Widget

The Python ttk Combobox presents a drop down list of options and displays them one at a time. It is a sub class of the widget Entry. Hence it inherits many options and methods from the Entry class.

Syntax

from tkinter import ttk

Combo = ttk.Combobox(master, values.......)

The get() function to retrieve the current value of the Combobox.

Example

from tkinter import \*

from tkinter import ttk

top = Tk()

top.geometry("200x150")

frame = Frame(top)

frame.pack()

langs = ["C", "C++", "Java",

"Python", "PHP"]

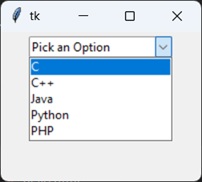
Combo = ttk.Combobox(frame, values = langs)

Combo.set("Pick an Option")

Combo.pack(padx = 5, pady = 5)

top.mainloop()

It will produce the following **output** −



Progressbar

The ttk ProgressBar widget, and how it can be used to create loading screens or show the progress of a current task.

Syntax

ttk.Progressbar(parent, orient, length, mode)

Parameters

* **Parent** − The container in which the ProgressBar is to be placed, such as root or a Tkinter frame.
* **Orient** − Defines the orientation of the ProgressBar, which can be either vertical of horizontal.
* **Length** − Defines the width of the ProgressBar by taking in an integer value.
* **Mode** − There are two options for this parameter, determinate and indeterminate.

Example

The code given below creates a progressbar with three buttons which are linked to three different functions.

The first function increments the "value" or "progress" in the progressbar by 20. This is done with the step() function which takes an integer value to change progress amount. (Default is 1.0)

The second function decrements the "value" or "progress" in the progressbar by 20.

The third function prints out the current progress level in the progressbar.

import tkinter as tk

from tkinter import ttk

root = tk.Tk()

frame= ttk.Frame(root)

def increment():

progressBar.step(20)

def decrement():

progressBar.step(-20)

def display():

print(progressBar["value"])

progressBar= ttk.Progressbar(frame, mode='determinate')

progressBar.pack(padx = 10, pady = 10)

button= ttk.Button(frame, text= "Increase", command= increment)

button.pack(padx = 10, pady = 10, side = tk.LEFT)

button= ttk.Button(frame, text= "Decrease", command= decrement)

button.pack(padx = 10, pady = 10, side = tk.LEFT)

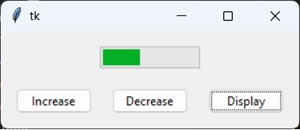
button= ttk.Button(frame, text= "Display", command= display)

button.pack(padx = 10, pady = 10, side = tk.LEFT)

frame.pack(padx = 5, pady = 5)

root.mainloop()

It will produce the following **output** −



Notebook

Tkinter ttk module has a new useful widget called Notebook. It is a of collection of of containers (e.g frames) which have many widgets as children inside.

Each "tab" or "window" has a tab ID associated with it, which is used to determine which tab to swap to.

You can swap between these containers like you would on a regular text editor.

Syntax

notebook = ttk.Notebook(master, \*options)

Example

In this example, add 3 windows to our Notebook widget in two different ways. The first method involves the add() function, which simply appends a new tab to the end. The other method is the insert() function which can be used to add a tab to a specific position.

The add() function takes one mandatory parameter which is the container widget to be added, and the rest are optional parameters such as text (text to be displayed as tab title), image and compound.

The insert() function requires a tab\_id, which defines the location where it should be inserted. The tab\_id can be either an index value or it can be string literal like "end", which will append it to the end.

import tkinter as tk

from tkinter import ttk

root = tk.Tk()

nb = ttk.Notebook(root)

# Frame 1 and 2

frame1 = ttk.Frame(nb)

frame2 = ttk.Frame(nb)

label1 = ttk.Label(frame1, text = "This is Window One")

label1.pack(pady = 50, padx = 20)

label2 = ttk.Label(frame2, text = "This is Window Two")

label2.pack(pady = 50, padx = 20)

frame1.pack(fill= tk.BOTH, expand=True)

frame2.pack(fill= tk.BOTH, expand=True)

nb.add(frame1, text = "Window 1")

nb.add(frame2, text = "Window 2")

frame3 = ttk.Frame(nb)

label3 = ttk.Label(frame3, text = "This is Window Three")

label3.pack(pady = 50, padx = 20)

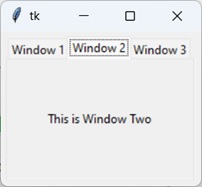
frame3.pack(fill= tk.BOTH, expand=True)

nb.insert("end", frame3, text = "Window 3")

nb.pack(padx = 5, pady = 5, expand = True)

root.mainloop()

It will produce the following **output** −



Treeview

The Treeview widget is used to display items in a tabular or hierarchical manner. It has support for features like creating rows and columns for items, as well as allowing items to have children as well, leading to a hierarchical format.

Syntax

tree = ttk.Treeview(container, \*\*options)

Options

|  |  |
| --- | --- |
| **Sr.No.** | **Option & Description** |
| 1 | **columns**  A list of column names |
| 2 | **displaycolumns**  A list of column identifiers (either symbolic or integer indices) specifying which data columns are displayed and the order in which they appear, or the string "#all". |
| 3 | **height**  The number of rows visible. |
| 4 | **padding**  Specifies the internal padding for the widget. Can be either an integer or a list of 4 values. |
| 5 | **selectmode**  One of "extended", "browse" or "none". If set to "extended" (default), multiple items can be selected. If "browse", only a single item can be selected at a time. If "none", the selection cannot be changed by the user. |
| 6 | **show**  A list containing zero or more of the following values, specifying which elements of the tree to display. The default is "tree headings", i.e., show all elements. |

Example

In this example we will create a simple Treeview ttk Widget and fill in some data into it. We have some data already stored in a list which will be reading and adding to the Treeview widget in our read\_data() function.

We first need to define a list/tuple of column names. We have left out the column "Name" because there already exists a (default) column with a blank name.

We then assign that list/tuple to the columns option in Treeview, followed by defining the "headings", where the column is the actual column, whereas the heading is just the title of the column that appears when the widget is displayed. We give each a column a name. "#0" is the name of the default column.

The tree.insert() function has the following parameters −

* **Parent** − which is left as an empty string if there is none.
* **Position** − where we want to add the new item. To append, use tk.END
* **Iid** − which is the item ID used to later track the item in question.
* **Text** − to which we will assign the first value in the list (the name).

Value we will pass the the other 2 values we obtained from the list.

The Complete Code

import tkinter as tk

import tkinter.ttk as ttk

from tkinter import simpledialog

root = tk.Tk()

data = [

["Bobby",26,20000],

["Harrish",31,23000],

["Jaya",18,19000],

["Mark",22, 20500],

]

index=0

def read\_data():

for index, line in enumerate(data):

tree.insert('', tk.END, iid = index,

text = line[0], values = line[1:])

columns = ("age", "salary")

tree= ttk.Treeview(root, columns=columns ,height = 20)

tree.pack(padx = 5, pady = 5)

tree.heading('#0', text='Name')

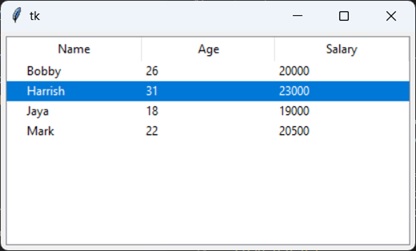
tree.heading('age', text='Age')

tree.heading('salary', text='Salary')

read\_data()

root.mainloop()

It will produce the following **output** −



Sizegrip

The Sizegrip widget is basically a small arrow-like grip that is typically placed at the bottom-right corner of the screen. Dragging the Sizegrip across the screen also resizes the container to which it is attached to.

Syntax

sizegrip = ttk.Sizegrip(parent, \*\*options)

Example

import tkinter as tk

import tkinter.ttk as ttk

root = tk.Tk()

root.geometry("100x100")

frame = ttk.Frame(root)

label = ttk.Label(root, text = "Hello World")

label.pack(padx = 5, pady = 5)

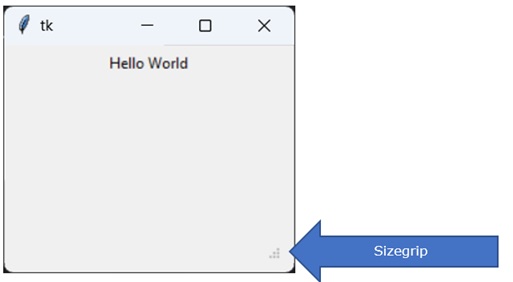
sizegrip = ttk.Sizegrip(frame)

sizegrip.pack(expand = True, fill = tk.BOTH, anchor = tk.SE)

frame.pack(padx = 10, pady = 10, expand = True, fill = tk.BOTH)

root.mainloop()

It will produce the following **output** −



Separator

The ttk Separator widget is a very simple widget, that has just one purpose and that is to help "separate" widgets into groups/partitions by drawing a line between them. We can change the orientation of this line (separator) to either horizontal or vertical, and change its length/height.

Syntax

separator = ttk.Separator(parent, \*\*options)

The "orient", which can either be tk.VERTICAL or tk.HORIZTONAL, for a vertical and horizontal separator respectively.

Example

Here we have created two Label widgets, and then created a Horizontal Separator between them.

import tkinter as tk

import tkinter.ttk as ttk

root = tk.Tk()

root.geometry("200x150")

frame = ttk.Frame(root)

label = ttk.Label(frame, text = "Hello World")

label.pack(padx = 5)

separator = ttk.Separator(frame,orient= tk.HORIZONTAL)

separator.pack(expand = True, fill = tk.X)

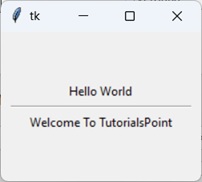
label = ttk.Label(frame, text = "Welcome To TutorialsPoint")

label.pack(padx = 5)

frame.pack(padx = 10, pady = 50, expand = True, fill = tk.BOTH)

root.mainloop()

It will produce the following **output** −



tkinter — Python interface to Tcl/Tk[¶](https://docs.python.org/3/library/tkinter.html#module-tkinter)

**Source code:** [Lib/tkinter/\_\_init\_\_.py](https://github.com/python/cpython/tree/3.13/Lib/tkinter/__init__.py)

The [tkinter](https://docs.python.org/3/library/tkinter.html#module-tkinter) package (“Tk interface”) is the standard Python interface to the Tcl/Tk GUI toolkit. Both Tk and [tkinter](https://docs.python.org/3/library/tkinter.html#module-tkinter) are available on most Unix platforms, including macOS, as well as on Windows systems.

Running python -m tkinter from the command line should open a window demonstrating a simple Tk interface, letting you know that [tkinter](https://docs.python.org/3/library/tkinter.html#module-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.

Tkinter supports a range of Tcl/Tk versions, built either with or without thread support. The official Python binary release bundles Tcl/Tk 8.6 threaded. See the source code for the [\_tkinter](https://docs.python.org/3/library/tkinter.html#module-_tkinter) module for more information about supported versions.

Tkinter is not a thin wrapper, but adds a fair amount of its own logic to make the experience more pythonic. This documentation will concentrate on these additions and changes, and refer to the official Tcl/Tk documentation for details that are unchanged.

**Note**

Tcl/Tk 8.5 (2007) introduced a modern set of themed user interface components along with a new API to use them. Both old and new APIs are still available. Most documentation you will find online still uses the old API and can be woefully outdated.

**See also**

* [**TkDocs**](https://tkdocs.com/)

Extensive tutorial on creating user interfaces with Tkinter. Explains key concepts, and illustrates recommended approaches using the modern API.

* [**Tkinter 8.5 reference: a GUI for Python**](https://www.tkdocs.com/shipman/)

Reference documentation for Tkinter 8.5 detailing available classes, methods, and options.

Tcl/Tk Resources:

* [**Tk commands**](https://www.tcl.tk/man/tcl8.6/TkCmd/contents.htm)

Comprehensive reference to each of the underlying Tcl/Tk commands used by Tkinter.

* [**Tcl/Tk Home Page**](https://www.tcl.tk/)

Additional documentation, and links to Tcl/Tk core development.

Books:

* [**Modern Tkinter for Busy Python Developers**](https://tkdocs.com/book.html)

By Mark Roseman. (ISBN 978-1999149567)

* [**Python GUI programming with Tkinter**](https://www.packtpub.com/en-us/product/python-gui-programming-with-tkinter-9781788835886)

By Alan D. Moore. (ISBN 978-1788835886)

* [**Programming Python**](https://learning-python.com/about-pp4e.html)

By Mark Lutz; has excellent coverage of Tkinter. (ISBN 978-0596158101)

* [**Tcl and the Tk Toolkit (2nd edition)**](https://www.amazon.com/exec/obidos/ASIN/032133633X)

By John Ousterhout, inventor of Tcl/Tk, and Ken Jones; does not cover Tkinter. (ISBN 978-0321336330)

Architecture

Tcl/Tk is not a single library but rather consists of a few distinct modules, each with separate functionality and its own official documentation. Python’s binary releases also ship an add-on module together with it.

Tcl

Tcl is a dynamic interpreted programming language, just like Python. Though it can be used on its own as a general-purpose programming language, it is most commonly embedded into C applications as a scripting engine or an interface to the Tk toolkit. The Tcl library has a C interface to create and manage one or more instances of a Tcl interpreter, run Tcl commands and scripts in those instances, and add custom commands implemented in either Tcl or C. Each interpreter has an event queue, and there are facilities to send events to it and process them. Unlike Python, Tcl’s execution model is designed around cooperative multitasking, and Tkinter bridges this difference (see [Threading model](https://docs.python.org/3/library/tkinter.html#threading-model) for details).

Tk

Tk is a [Tcl package](https://wiki.tcl-lang.org/37432) implemented in C that adds custom commands to create and manipulate GUI widgets. Each [Tk](https://docs.python.org/3/library/tkinter.html#tkinter.Tk) object embeds its own Tcl interpreter instance with Tk loaded into it. Tk’s widgets are very customizable, though at the cost of a dated appearance. Tk uses Tcl’s event queue to generate and process GUI events.

Ttk

Themed Tk (Ttk) is a newer family of Tk widgets that provide a much better appearance on different platforms than many of the classic Tk widgets. Ttk is distributed as part of Tk, starting with Tk version 8.5. Python bindings are provided in a separate module, [tkinter.ttk](https://docs.python.org/3/library/tkinter.ttk.html#module-tkinter.ttk).

Internally, Tk and Ttk use facilities of the underlying operating system, i.e., Xlib on Unix/X11, Cocoa on macOS, GDI on Windows.

When your Python application uses a class in Tkinter, e.g., to create a widget, the [tkinter](https://docs.python.org/3/library/tkinter.html#module-tkinter) module first assembles a Tcl/Tk command string. It passes that Tcl command string to an internal [\_tkinter](https://docs.python.org/3/library/tkinter.html#module-_tkinter) binary module, which then calls the Tcl interpreter to evaluate it. The Tcl interpreter will then call into the Tk and/or Ttk packages, which will in turn make calls to Xlib, Cocoa, or GDI.

Tkinter Modules

Support for Tkinter is spread across several modules. Most applications will need the main [tkinter](https://docs.python.org/3/library/tkinter.html#module-tkinter) module, as well as the [tkinter.ttk](https://docs.python.org/3/library/tkinter.ttk.html#module-tkinter.ttk) module, which provides the modern themed widget set and API:

**from** **tkinter** **import** \*

**from** **tkinter** **import** ttk

*class*tkinter.**Tk**(*screenName=None*, *baseName=None*, *className='Tk'*, *useTk=True*, *sync=False*, *use=None*)

Construct a toplevel Tk widget, which is usually the main window of an application, and initialize a Tcl interpreter for this widget. Each instance has its own associated Tcl interpreter.

The [Tk](https://docs.python.org/3/library/tkinter.html#tkinter.Tk) class is typically instantiated using all default values. However, the following keyword arguments are currently recognized:

*screenName*

When given (as a string), sets the DISPLAY environment variable. (X11 only)

*baseName*

Name of the profile file. By default, *baseName* is derived from the program name (sys.argv[0]).

*className*

Name of the widget class. Used as a profile file and also as the name with which Tcl is invoked (*argv0* in *interp*).

*useTk*

If True, initialize the Tk subsystem. The [tkinter.Tcl()](https://docs.python.org/3/library/tkinter.html#tkinter.Tcl) function sets this to False.

*sync*

If True, execute all X server commands synchronously, so that errors are reported immediately. Can be used for debugging. (X11 only)

*use*

Specifies the *id* of the window in which to embed the application, instead of it being created as an independent toplevel window. *id* must be specified in the same way as the value for the -use option for toplevel widgets (that is, it has a form like that returned by winfo\_id()).

Note that on some platforms this will only work correctly if *id* refers to a Tk frame or toplevel that has its -container option enabled.

[Tk](https://docs.python.org/3/library/tkinter.html#tkinter.Tk) reads and interprets profile files, named .*className*.tcl and .*baseName*.tcl, into the Tcl interpreter and calls [exec()](https://docs.python.org/3/library/functions.html#exec) on the contents of .*className*.py and .*baseName*.py. The path for the profile files is the HOME environment variable or, if that isn’t defined, then [os.curdir](https://docs.python.org/3/library/os.html#os.curdir).

**tk**

The Tk application object created by instantiating [Tk](https://docs.python.org/3/library/tkinter.html#tkinter.Tk). This provides access to the Tcl interpreter. Each widget that is attached the same instance of [Tk](https://docs.python.org/3/library/tkinter.html#tkinter.Tk) has the same value for its [tk](https://docs.python.org/3/library/tkinter.html#tkinter.Tk.tk) attribute.

**master**

The widget object that contains this widget. For [Tk](https://docs.python.org/3/library/tkinter.html#tkinter.Tk), the *master* is [None](https://docs.python.org/3/library/constants.html#None) because it is the main window. The terms *master* and *parent* are similar and sometimes used interchangeably as argument names; however, calling winfo\_parent() returns a string of the widget name whereas [master](https://docs.python.org/3/library/tkinter.html#tkinter.Tk.master) returns the object. *parent*/*child* reflects the tree-like relationship while *master*/*slave* reflects the container structure.

**children**

The immediate descendants of this widget as a [dict](https://docs.python.org/3/library/stdtypes.html#dict) with the child widget names as the keys and the child instance objects as the values.

tkinter.**Tcl**(*screenName=None*, *baseName=None*, *className='Tk'*, *useTk=False*)

The [Tcl()](https://docs.python.org/3/library/tkinter.html#tkinter.Tcl) function is a factory function which creates an object much like that created by the [Tk](https://docs.python.org/3/library/tkinter.html#tkinter.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()](https://docs.python.org/3/library/tkinter.html#tkinter.Tcl) object can have a Toplevel window created (and the Tk subsystem initialized) by calling its loadtk() method.

The modules that provide Tk support include:

[tkinter](https://docs.python.org/3/library/tkinter.html#module-tkinter)

Main Tkinter module.

[tkinter.colorchooser](https://docs.python.org/3/library/tkinter.colorchooser.html#module-tkinter.colorchooser)

Dialog to let the user choose a color.

[tkinter.commondialog](https://docs.python.org/3/library/dialog.html#module-tkinter.commondialog)

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

[tkinter.filedialog](https://docs.python.org/3/library/dialog.html#module-tkinter.filedialog)

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

[tkinter.font](https://docs.python.org/3/library/tkinter.font.html#module-tkinter.font)

Utilities to help work with fonts.

[tkinter.messagebox](https://docs.python.org/3/library/tkinter.messagebox.html#module-tkinter.messagebox)

Access to standard Tk dialog boxes.

[tkinter.scrolledtext](https://docs.python.org/3/library/tkinter.scrolledtext.html#module-tkinter.scrolledtext)

Text widget with a vertical scroll bar built in.

[tkinter.simpledialog](https://docs.python.org/3/library/dialog.html#module-tkinter.simpledialog)

Basic dialogs and convenience functions.

[tkinter.ttk](https://docs.python.org/3/library/tkinter.ttk.html#module-tkinter.ttk)

Themed widget set introduced in Tk 8.5, providing modern alternatives for many of the classic widgets in the main [tkinter](https://docs.python.org/3/library/tkinter.html#module-tkinter) module.

Additional modules:

[\_tkinter](https://docs.python.org/3/library/tkinter.html#module-_tkinter)

A binary module that contains the low-level interface to Tcl/Tk. It is automatically imported by the main [tkinter](https://docs.python.org/3/library/tkinter.html#module-tkinter) module, 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.

[idlelib](https://docs.python.org/3/library/idle.html#module-idlelib)

Python’s Integrated Development and Learning Environment (IDLE). Based on [tkinter](https://docs.python.org/3/library/tkinter.html#module-tkinter).

tkinter.constants

Symbolic constants that can be used in place of strings when passing various parameters to Tkinter calls. Automatically imported by the main [tkinter](https://docs.python.org/3/library/tkinter.html#module-tkinter) module.

[tkinter.dnd](https://docs.python.org/3/library/tkinter.dnd.html#module-tkinter.dnd)

(experimental) Drag-and-drop support for [tkinter](https://docs.python.org/3/library/tkinter.html#module-tkinter). This will become deprecated when it is replaced with the Tk DND.

[turtle](https://docs.python.org/3/library/turtle.html#module-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. For that, refer to one of the external resources noted earlier. Instead, this section provides a very quick orientation to what a Tkinter application looks like, identifies foundational Tk concepts, and explains how the Tkinter wrapper is structured.

The remainder of this section will help you to identify the classes, methods, and options you’ll need in your Tkinter application, and where to find more detailed documentation on them, including in the official Tcl/Tk reference manual.

A Hello World Program

We’ll start by walking through a “Hello World” application in Tkinter. This isn’t the smallest one we could write, but has enough to illustrate some key concepts you’ll need to know.

**from** **tkinter** **import** \*

**from** **tkinter** **import** ttk

root = Tk()

frm = ttk.Frame(root, padding=10)

frm.grid()

ttk.Label(frm, text="Hello World!").grid(column=0, row=0)

ttk.Button(frm, text="Quit", command=root.destroy).grid(column=1, row=0)

root.mainloop()

After the imports, the next line creates an instance of the Tk class, which initializes Tk and creates its associated Tcl interpreter. It also creates a toplevel window, known as the root window, which serves as the main window of the application.

The following line creates a frame widget, which in this case will contain a label and a button we’ll create next. The frame is fit inside the root window.

The next line creates a label widget holding a static text string. The grid() method is used to specify the relative layout (position) of the label within its containing frame widget, similar to how tables in HTML work.

A button widget is then created, and placed to the right of the label. When pressed, it will call the destroy() method of the root window.

Finally, the mainloop() method puts everything on the display, and responds to user input until the program terminates.

Important Tk Concepts

Even this simple program illustrates the following key Tk concepts:

widgets

A Tkinter user interface is made up of individual *widgets*. Each widget is represented as a Python object, instantiated from classes like ttk.Frame, ttk.Label, and ttk.Button.

widget hierarchy

Widgets are arranged in a *hierarchy*. The label and button were contained within a frame, which in turn was contained within the root window. When creating each *child* widget, its *parent* widget is passed as the first argument to the widget constructor.

configuration options

Widgets have *configuration options*, which modify their appearance and behavior, such as the text to display in a label or button. Different classes of widgets will have different sets of options.

geometry management

Widgets aren’t automatically added to the user interface when they are created. A *geometry manager* like grid controls where in the user interface they are placed.

event loop

Tkinter reacts to user input, changes from your program, and even refreshes the display only when actively running an *event loop*. If your program isn’t running the event loop, your user interface won’t update.

Understanding How Tkinter Wraps Tcl/Tk

When your application uses Tkinter’s classes and methods, internally Tkinter is assembling strings representing Tcl/Tk commands, and executing those commands in the Tcl interpreter attached to your application’s Tk instance.

Whether it’s trying to navigate reference documentation, trying to find the right method or option, adapting some existing code, or debugging your Tkinter application, there are times that it will be useful to understand what those underlying Tcl/Tk commands look like.

To illustrate, here is the Tcl/Tk equivalent of the main part of the Tkinter script above.

ttk::frame .frm -padding 10

grid .frm

grid [ttk::label .frm.lbl -text "Hello World!"] -column 0 -row 0

grid [ttk::button .frm.btn -text "Quit" -command "destroy ."] -column 1 -row 0

Tcl’s syntax is similar to many shell languages, where the first word is the command to be executed, with arguments to that command following it, separated by spaces. Without getting into too many details, notice the following:

* The commands used to create widgets (like ttk::frame) correspond to widget classes in Tkinter.
* Tcl widget options (like -text) correspond to keyword arguments in Tkinter.
* Widgets are referred to by a *pathname* in Tcl (like .frm.btn), whereas Tkinter doesn’t use names but object references.
* A widget’s place in the widget hierarchy is encoded in its (hierarchical) pathname, which uses a . (dot) as a path separator. The pathname for the root window is just . (dot). In Tkinter, the hierarchy is defined not by pathname but by specifying the parent widget when creating each child widget.
* Operations which are implemented as separate *commands* in Tcl (like grid or destroy) are represented as *methods* on Tkinter widget objects. As you’ll see shortly, at other times Tcl uses what appear to be method calls on widget objects, which more closely mirror what would is used in Tkinter.

How do I…? What option does…?

If you’re not sure how to do something in Tkinter, and you can’t immediately find it in the tutorial or reference documentation you’re using, there are a few strategies that can be helpful.

First, remember that the details of how individual widgets work may vary across different versions of both Tkinter and Tcl/Tk. If you’re searching documentation, make sure it corresponds to the Python and Tcl/Tk versions installed on your system.

When searching for how to use an API, it helps to know the exact name of the class, option, or method that you’re using. Introspection, either in an interactive Python shell or with [print()](https://docs.python.org/3/library/functions.html#print), can help you identify what you need.

To find out what configuration options are available on any widget, call its configure() method, which returns a dictionary containing a variety of information about each object, including its default and current values. Use keys() to get just the names of each option.

btn = ttk.Button(frm, ...)

print(btn.configure().keys())

As most widgets have many configuration options in common, it can be useful to find out which are specific to a particular widget class. Comparing the list of options to that of a simpler widget, like a frame, is one way to do that.

print(set(btn.configure().keys()) - set(frm.configure().keys()))

Similarly, you can find the available methods for a widget object using the standard [dir()](https://docs.python.org/3/library/functions.html#dir) function. If you try it, you’ll see there are over 200 common widget methods, so again identifying those specific to a widget class is helpful.

print(dir(btn))

print(set(dir(btn)) - set(dir(frm)))

Navigating the Tcl/Tk Reference Manual

As noted, the official [Tk commands](https://www.tcl.tk/man/tcl8.6/TkCmd/contents.htm) reference manual (man pages) is often the most accurate description of what specific operations on widgets do. Even when you know the name of the option or method that you need, you may still have a few places to look.

While all operations in Tkinter are implemented as method calls on widget objects, you’ve seen that many Tcl/Tk operations appear as commands that take a widget pathname as its first parameter, followed by optional parameters, e.g.

destroy .

grid .frm.btn -column 0 -row 0

Others, however, look more like methods called on a widget object (in fact, when you create a widget in Tcl/Tk, it creates a Tcl command with the name of the widget pathname, with the first parameter to that command being the name of a method to call).

.frm.btn invoke

.frm.lbl configure -text "Goodbye"

In the official Tcl/Tk reference documentation, you’ll find most operations that look like method calls on the man page for a specific widget (e.g., you’ll find the invoke() method on the [ttk::button](https://www.tcl.tk/man/tcl8.6/TkCmd/ttk_button.htm) man page), while functions that take a widget as a parameter often have their own man page (e.g., [grid](https://www.tcl.tk/man/tcl8.6/TkCmd/grid.htm)).

You’ll find many common options and methods in the [options](https://www.tcl.tk/man/tcl8.6/TkCmd/options.htm) or [ttk::widget](https://www.tcl.tk/man/tcl8.6/TkCmd/ttk_widget.htm) man pages, while others are found in the man page for a specific widget class.

You’ll also find that many Tkinter methods have compound names, e.g., winfo\_x(), winfo\_height(), winfo\_viewable(). You’d find documentation for all of these in the [winfo](https://www.tcl.tk/man/tcl8.6/TkCmd/winfo.htm) man page.

**Note**

Somewhat confusingly, there are also methods on all Tkinter widgets that don’t actually operate on the widget, but operate at a global scope, independent of any widget. Examples are methods for accessing the clipboard or the system bell. (They happen to be implemented as methods in the base Widget class that all Tkinter widgets inherit from).

Threading model

Python and Tcl/Tk have very different threading models, which [tkinter](https://docs.python.org/3/library/tkinter.html#module-tkinter) tries to bridge. If you use threads, you may need to be aware of this.

A Python interpreter may have many threads associated with it. In Tcl, multiple threads can be created, but each thread has a separate Tcl interpreter instance associated with it. Threads can also create more than one interpreter instance, though each interpreter instance can be used only by the one thread that created it.

Each Tk object created by [tkinter](https://docs.python.org/3/library/tkinter.html#module-tkinter) contains a Tcl interpreter. It also keeps track of which thread created that interpreter. Calls to [tkinter](https://docs.python.org/3/library/tkinter.html#module-tkinter) can be made from any Python thread. Internally, if a call comes from a thread other than the one that created the Tk object, an event is posted to the interpreter’s event queue, and when executed, the result is returned to the calling Python thread.

Tcl/Tk applications are normally event-driven, meaning that after initialization, the interpreter runs an event loop (i.e. Tk.mainloop()) and responds to events. Because it is single-threaded, event handlers must respond quickly, otherwise they will block other events from being processed. To avoid this, any long-running computations should not run in an event handler, but are either broken into smaller pieces using timers, or run in another thread. This is different from many GUI toolkits where the GUI runs in a completely separate thread from all application code including event handlers.

If the Tcl interpreter is not running the event loop and processing events, any [tkinter](https://docs.python.org/3/library/tkinter.html#module-tkinter) calls made from threads other than the one running the Tcl interpreter will fail.

A number of special cases exist:

* Tcl/Tk libraries can be built so they are not thread-aware. In this case, [tkinter](https://docs.python.org/3/library/tkinter.html#module-tkinter) calls the library from the originating Python thread, even if this is different than the thread that created the Tcl interpreter. A global lock ensures only one call occurs at a time.
* While [tkinter](https://docs.python.org/3/library/tkinter.html#module-tkinter) allows you to create more than one instance of a Tk object (with its own interpreter), all interpreters that are part of the same thread share a common event queue, which gets ugly fast. In practice, don’t create more than one instance of Tk at a time. Otherwise, it’s best to create them in separate threads and ensure you’re running a thread-aware Tcl/Tk build.
* Blocking event handlers are not the only way to prevent the Tcl interpreter from reentering the event loop. It is even possible to run multiple nested event loops or abandon the event loop entirely. If you’re doing anything tricky when it comes to events or threads, be aware of these possibilities.
* There are a few select [tkinter](https://docs.python.org/3/library/tkinter.html#module-tkinter) functions that presently work only when called from the thread that created the Tcl interpreter.

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)*](https://manpages.debian.org/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 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](https://docs.python.org/3/library/tkinter.html#module-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](https://docs.python.org/3/library/tkinter.html#module-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:

**import** **tkinter** **as** **tk**

**class** **App**(tk.Frame):

**def** \_\_init\_\_(self, master):

super().\_\_init\_\_(master)

self.pack()

self.entrythingy = tk.Entry()

self.entrythingy.pack()

*# Create the application variable.*

self.contents = tk.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

*# Define a callback for when the user hits return.*

*# It prints the current value of the variable.*

self.entrythingy.bind('<Key-Return>',

self.print\_contents)

**def** print\_contents(self, event):

print("Hi. The current entry content is:",

self.contents.get())

root = tk.Tk()

myapp = App(root)

myapp.mainloop()

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](https://docs.python.org/3/library/tkinter.html#module-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: "#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\_ 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(3tk)*](https://manpages.debian.org/bind(3tk)) man page, and page 201 of John Ousterhout’s book, *Tcl and the Tk Toolkit (2nd edition)*, 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** |
| --- | --- | --- | --- |
| %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](https://docs.python.org/3/library/tkinter.html#module-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).

*Changed in version 3.13:*Added the PhotoImage method copy\_replace() to copy a region from one image to other image, possibly with pixel zooming and/or subsampling. Add *from\_coords* parameter to PhotoImage methods copy(), zoom() and subsample(). Add *zoom* and *subsample* parameters to PhotoImage method copy().

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](https://python-pillow.org/) 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](https://docs.python.org/3/library/io.html#io.BufferedIOBase) or [TextIOBase](https://docs.python.org/3/library/io.html#io.TextIOBase) [read()](https://docs.python.org/3/library/io.html#io.BufferedIOBase.read) or [readline()](https://docs.python.org/3/library/io.html#io.IOBase.readline) methods, since these will insist on reading a predefined number of bytes. For sockets, the [recv()](https://docs.python.org/3/library/socket.html#socket.socket.recv) or [recvfrom()](https://docs.python.org/3/library/socket.html#socket.socket.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()](https://docs.python.org/3/library/io.html#io.IOBase.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.